

開發人員指南

AWS AppSync GraphQL



AWS AppSync GraphQL: 開發人員指南

Copyright © 2025 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon 的商標和商業外觀不得用於任何非 Amazon 的產品或服務，也不能以任何可能造成客戶混淆、任何貶低或使 Amazon 名譽受損的方式使用 Amazon 的商標和商業外觀。所有其他非 Amazon 擁有的商標均為其各自擁有者的財產，這些擁有者可能附屬於 Amazon，或與 Amazon 有合作關係，亦或受到 Amazon 贊助。

Table of Contents

什麼是 AWS AppSync	1
AWS AppSync 功能	1
您是第一次 AWS AppSync 使用嗎？	2
相關服務	2
的定價 AWS AppSync	2
GraphQL 和 AWS AppSync 架構	3
什麼是 API	4
用戶端	4
資源	4
什麼是 REST	4
統一界面	4
無狀態	5
分層系統	5
快取性	5
什麼是 RESTful API？	5
RESTful APIs如何運作？	6
什麼是 GraphQL	6
比較 REST 和 GraphQL	6
為什麼透過 REST 使用 GraphQL	8
GraphQL API 的元件	9
GraphQL 結構描述	10
資料來源	26
解析程式	36
GraphQL 的其他屬性	45
宣告式	45
階層	46
隱含性	47
強式輸入	48
入門：建立您的第一個 GraphQL API	50
啟動結構描述	51
導覽 AWS AppSync 主控台	54
結構描述設計工具	55
資料來源	56
查詢	56

設定	57
使用 GraphQL 變動將資料新增至 DynamoDB 資料表	57
使用 GraphQL 查詢從 DynamoDB 資料表擷取資料	62
補充區段	65
整合	65
補充讀取	65
設計 GraphQL APIs	66
建構 GraphQL API (空白或匯入APIs)	66
設計 GraphQL 結構描述	67
連接資料來源	92
設定 AWS AppSync 解析程式	103
搭配 CDK APIs	153
針對即時資料應用程式使用訂閱	170
GraphQL 結構描述訂閱指令	170
使用訂閱引數	173
建立採用無伺服器 WebSockets技術的通用 pub/sub APIs	176
定義增強型訂閱篩選條件	179
使用篩選條件取消訂閱 WebSocket 連線	189
建置即時 WebSocket 用戶端	193
合併 APIs	212
合併 APIs和聯合	213
合併 API 衝突解決	215
設定結構描述	223
設定授權模式	223
設定執行角色	224
使用 設定跨帳戶合併 APIs AWS RAM	225
合併	227
合併 APIs的其他支援	228
合併 API 限制	229
建立合併 APIs	229
使用 RDS 自我檢查建置 GraphQL APIs	230
使用自我檢查功能 (主控台)	231
使用自我檢查功能 (API)	234
建置用戶端應用程式	237
JavaScript 解析程式教學課程	240
使用 DynamoDB JavaScript 解析程式建立簡單的貼文應用程式	240

建立 GraphQL API	240
定義基本文章 API	241
設定 Amazon DynamoDB 資料表	242
設定 addPost 解析程式 (Amazon DynamoDB PutItem)	242
設定 getPost 解析程式 (Amazon DynamoDB GetItem)	246
建立 updatePost 變動 (Amazon DynamoDB UpdateItem)	248
建立投票變動 (Amazon DynamoDB UpdateItem)	252
設定 deletePost 解析程式 (Amazon DynamoDB DeleteItem)	255
設定 allPost 解析程式 (Amazon DynamoDB Scan)	261
設定 allPostsByAuthor 解析程式 (Amazon DynamoDB 查詢)	265
使用 集合	270
結論	277
使用 AWS Lambda 解析程式	277
建立 Lambda 函數	277
設定 Lambda 的資料來源	279
建立 GraphQL 結構描述	280
設定解析程式	104
測試 GraphQL API	282
傳回錯誤	283
進階使用案例：批次處理	286
使用本機解析程式	295
建立 pub/sub 應用程式	295
傳送和訂閱訊息	296
結合 GraphQL 解析程式	297
範例結構描述	297
透過解析程式變更資料	299
DynamoDB 和 OpenSearch Service	299
使用 OpenSearch Service 解析程式	301
建立新的 OpenSearch Service 網域	302
設定 OpenSearch Service 的資料來源	302
連接解析程式	304
修改您的搜尋	306
將資料新增至 OpenSearch Service	307
擷取單一文件	308
執行查詢和變動	309
最佳實務	309

執行 DynamoDB 交易	310
許可	310
資料來源	311
交易	312
使用 DynamoDB 批次操作	319
單一資料表批次	320
多資料表批次	324
錯誤處理	332
使用 HTTP 解析程式	337
建立 REST API	338
建立 GraphQL API	338
建立 GraphQL 結構描述	339
設定 HTTP 資料來源	339
設定解析程式	133
叫用 AWS 服務	343
搭配資料 API 使用 Aurora PostgreSQL	344
設定您的 Aurora PostgreSQL 資料庫	345
建立資料庫和資料表	346
建立 GraphQL 結構描述	347
RDS 的解析程式	349
刪除叢集	357
VTL 解析程式教學課程	358
使用 DynamoDB 解析程式建立簡單的貼文應用程式	359
設定 DynamoDB 資料表	359
建立 GraphQL API	338
定義基本文章 API	360
設定 DynamoDB 資料表的資料來源	361
設定 addPost 解析程式 (DynamoDB PutItem)	362
設定 getPost 解析程式 (DynamoDB GetItem)	367
建立 updatePost Mutation (DynamoDB UpdateItem)	370
修改 updatePost 解析程式 (DynamoDB UpdateItem)	373
建立 upvotePost 和 downvotePost 變動 (DynamoDB UpdateItem)	379
設定 deletePost Resolver (DynamoDB DeleteItem)	383
設定 allPost 解析程式 (DynamoDB Scan)	389
設定 allPostsByAuthor Resolver (DynamoDB 查詢)	394
使用集	270

使用清單和映射	406
結論	409
使用 AWS Lambda 解析程式	410
建立 Lambda 函數	410
設定 Lambda 的資料來源	412
建立 GraphQL 結構描述	339
設定解析程式	133
測試 GraphQL API	416
傳回錯誤	418
進階使用案例：批次處理	420
使用 OpenSearch Service 解析程式	430
一鍵設定	430
建立新的 OpenSearch Service 網域	430
設定 OpenSearch Service 的資料來源	431
連結解析程式	433
修改您的搜尋	435
將資料新增至 OpenSearch Service	436
擷取單一文件	437
執行查詢與變動	437
最佳實務	438
使用本機解析程式	438
建立分頁應用程式	439
傳送和訂閱頁面	440
結合 GraphQL 解析程式	441
範例結構描述	441
透過解析程式來修改資料	442
DynamoDB 和 OpenSearch Service	443
使用 DynamoDB 批次操作	447
許可	447
資料來源	448
單一資料表批次	449
多重資料表批次	452
錯誤處理	460
執行 DynamoDB 交易	465
許可	447
資料來源	448

交易	468
使用 HTTP 解析程式	477
一鍵設定	430
建立 REST API	338
建立 GraphQL API	338
建立 GraphQL 結構描述	339
設定您的 HTTP 資料來源	339
設定解析程式	133
叫用 AWS 服務	483
使用 Aurora Serverless v2 解析程式	484
設定資料庫叢集	486
啟用 Data API	487
建立資料庫及資料表	487
GraphQL 結構描述	488
將您的 API 連線至資料庫操作	133
透過 API 修改您的資料	496
擷取您的資料	497
保護您的資料存取	498
使用管道解析程式	500
一鍵設定	430
手動設定	501
測試您的 GraphQL API	416
在版本控制的資料來源上使用 Delta Sync 操作	514
一鍵設定	430
結構描述	515
變動	517
同步查詢	518
範例	518
組態和設定	525
設定伺服器端快取和 API 承載壓縮	525
執行個體類型	526
快取行為	527
快取加密	529
快取移出	529
移出快取項目	529
根據身分移出快取項目	531

壓縮 API 回應	532
設定 GraphQL 和即時 APIs 自訂網域名稱	533
註冊和設定網域名稱	534
在 AWS AppSync 中建立自訂網域名稱	534
AWS AppSync 中的萬用字元自訂網域名稱	535
DynamoDB 的版本控制、衝突偵測和同步操作	535
版本控制 DynamoDB 資料來源	536
衝突偵測和解決	539
在版本控制的資料來源上使用 DynamoDB 同步操作	548
使用 CloudWatch 監控和記錄 GraphQL API 資料	549
設定和組態	550
CloudWatch 指標	551
CloudWatch Logs	560
日誌類型參考	565
使用 CloudWatch Logs Insights 分析您的日誌	567
使用 OpenSearch Service 分析您的日誌	568
日誌格式遷移	569
在 中追蹤請求 AWS X-Ray	569
設定與組態	550
使用 X-Ray 追蹤您的 API	570
使用 AWS CloudTrail 記錄 API 呼叫	572
AWS AppSync CloudTrail 中的資訊	572
AWS AppSync CloudTrail 中的資料事件	573
了解 AWS AppSync 日誌檔案項目	574
使用 AWS AppSync 私有 APIs	582
建立 AWS AppSync 私有 APIs	583
建立的介面端點 AWS AppSync	584
進階 範例	585
使用 IAM 政策限制公有 API 建立	588
共用 GraphQL APIs	589
shared AWS AppSync GraphQL APIs 的先決條件	590
Share AWS AppSync GraphQL APIs	590
停止共用 AWS AppSync GraphQL APIs	592
跨帳戶事件	592
設定 GraphQL 執行複雜性、查詢深度和自我檢查	592
使用自我檢查功能	593

設定查詢深度限制	594
設定解析程式計數限制	596
使用 環境變數	597
設定環境變數 (主控台)	598
設定環境變數 (API)	598
設定環境變數 (CFN)	600
環境變數和合併 APIs	600
擷取環境變數	600
授權和驗證 GraphQL APIs	602
授權類型	602
API_KEY 授權	603
AWS_LAMBDA 授權	605
規避 SigV4 和 OIDC 字符授權限制	610
AWS_IAM 授權	610
OPENID_CONNECT 授權	612
AMAZON_COGNITO_USER_POOLS 授權	614
使用其他授權模式	615
精細定義存取控制	617
篩選資訊	620
資料來源存取	620
用於保護請求和回應安全的存取控制使用案例	621
概觀	621
讀取資料	622
寫入資料	625
公有和私有記錄	628
即時資料	629
使用 AWS WAF 保護 APIs	632
將 AppSync API 與 整合 AWS WAF	633
建立 Web ACL 的規則	634
安全	638
資料保護	638
動態加密	639
法規遵循驗證	639
基礎架構安全	640
恢復能力	641
身分與存取管理	641

目標對象	641
使用身分驗證	642
使用政策管理存取權	644
How AWS AppSync 可與 IAM 搭配使用	646
身分型政策	652
故障診斷	663
使用 Logging AWS AppSync API 呼叫 AWS CloudTrail	665
CloudTrail 中的 AWS AppSync 資訊	666
Understanding AWS AppSync 日誌檔案項目	666
最佳實務	438
了解身分驗證方法	669
了解 API 組態如何變更傳播	669
針對 HTTP 解析程式使用 TLS	669
盡可能使用具有最低許可的角色	669
IAM 政策最佳實務	670
解析程式參考 (JavaScript)	671
JavaScript 解析程式概觀	671
支援的執行時間功能	672
單位解析程式	672
JavaScript 管道解析程式的剖析	672
補充主題	676
使用 Amazon DynamoDB 的範例管道解析程式	677
設定 APPSYNC_JS 執行時間的公用程式	679
APPSYNC_JS 執行時間的綁定、TypeScript 和來源映射	682
測試您的解析程式和函數處理常式	688
從 VTL 遷移至 JavaScript	690
在直接資料來源存取和透過 Lambda 資料來源代理之間進行選擇	693
JavaScript 解析程式內容物件參考	695
正在存取 context	695
解析程式和函數的 JavaScript 執行期功能	705
支援的執行時間功能	706
內建公用程式	713
內建模組	716
執行期公用程式	738
util.time 中的時間協助程式	738
util.dynamodb 中的 DynamoDB 協助程式	740

util.http 中的 HTTP 協助程式	746
util.transform 中的轉換協助程式	747
util.str 中的字串協助程式	761
延伸模組	761
util.xml 中的 XML 協助程式	765
DynamoDB 的 JavaScript 解析程式函數參考	766
GetItem	768
PutItem	769
UpdateItem	772
DeleteItem	776
Query	779
Scan	783
Sync	786
BatchGetItem	789
BatchDeleteItem	792
BatchPutItem	794
TransactGetItems	796
TransactWriteItems	799
類型系統（請求映射）	805
類型系統（回應映射）	810
篩選條件	814
條件表達式	815
交易條件表達式	828
投影	830
OpenSearch 的 JavaScript 解析程式函數參考	831
請求	832
回應	832
operation 欄位	833
path 欄位	833
params 欄位	834
傳遞變數	836
Lambda 的 JavaScript 解析程式函數參考	836
請求物件	837
回應物件	840
Lambda 函數批次回應	840
EventBridge 資料來源的 JavaScript 解析程式函數參考	841

請求	832
回應	841
PutEvents 欄位	843
None 資料來源的 JavaScript 解析程式函數參考	844
請求	832
承載	839
回應	841
HTTP 的 JavaScript 解析程式函數參考	846
請求	832
方法	846
ResourcePath	847
參數欄位	847
回應	841
Amazon RDS 的 JavaScript 解析程式函數參考	849
SQL 標記範本	849
建立陳述式	850
擷取資料	851
公用程式函數	852
轉換	859
Amazon Bedrock 的 JavaScript 解析程式函數參考	861
請求物件	861
回應物件	868
長時間執行的叫用	868
類型參考	869
解析程式映射範本參考 (VTL)	876
解析程式映射範本概觀	876
單位解析程式	877
管道解析程式	149
範例 範本	881
評估的映射範本還原序列化規則	883
解析程式映射範本程式設計指南	884
設定	885
Variables	886
呼叫方法	888
Strings	889
迴圈	890

陣列	890
條件式檢查	891
運算子	892
Context	894
篩選	894
解析程式映射範本內容參考	899
正在存取 \$context	899
處理輸入	908
Resolver 映射範本公用程式參考	909
\$util 中的公用程式協助程式	910
AWS AppSync 指令	921
\$util.time 中的時間協助程式	922
在 \$util.list 中列出協助程式	924
在 \$util.map 中映射協助程式	925
\$util.dynamodb 中的 DynamoDB 協助程式	926
\$util.rds 中的 Amazon RDS 協助程式	935
\$util.http 中的 HTTP 協助程式	938
\$util.xml 中的 XML 協助程式	940
\$util.transform 中的轉換協助程式	942
\$util.math 中的數學協助程式	956
\$util.str 中的字串協助程式	957
延伸模組	957
DynamoDB 的解析程式映射範本參考	970
GetItem	971
PutItem	973
UpdateItem	976
DeleteItem	982
Query	985
Scan	989
Sync	993
BatchGetItem	996
BatchDeleteItem	999
BatchPutItem	1003
TransactGetItems	1006
TransactWriteItems	1010
類型系統（請求映射）	1017

類型系統 (回應映射)	1022
篩選條件	1026
條件表達式	1027
交易條件表達式	1039
投影	1041
RDS 的解析程式映射範本參考	1042
請求映射範本	1042
版本	1044
陳述式和 VariableMap	1044
VariableTypeHintMap	1045
OpenSearch 的解析程式映射範本參考	1046
請求映射範本	1042
回應映射範本	832
operation 欄位	833
path 欄位	833
params 欄位	834
傳遞變數	836
Lambda 的解析程式映射範本參考	1050
請求映射範本	1042
回應映射範本	832
Lambda 函數批次回應	1056
Direct Lambda 解析程式	1056
EventBridge 的解析程式映射範本參考	1062
請求映射範本	1042
回應映射範本	832
PutEvents 欄位	843
None 資料來源的解析程式映射範本參考	1066
請求映射範本	1042
版本	1044
承載	1054
回應映射範本	832
HTTP 的解析程式映射範本參考	1068
請求映射範本	1042
版本	1044
方法	1071
ResourcePath	1072

參數欄位	834
回應	1073
HTTPS AWS AppSync 端點的憑證授權機構 (CA) 由 認可	1074
解析程式映射範本變更日誌	1136
每種版本陣列的可用資料來源操作	1137
變更單位解析程式映射範本的版本	1138
變更函數的版本	1139
2018-05-29	1139
2017-02-28	1146
GraphQL 類型參考	1147
GraphQL 中的純量類型	1147
預設純量	1148
AWS AppSync 純量	1148
結構描述用量範例	1149
GraphQL 中的界面和聯集	1153
介面範例	1153
聯集範例	1157
類型解析 in AWS AppSync	1158
類型解析度範例	1158
故障診斷和常見錯誤	1164
不正確的 DynamoDB 索引鍵映射	1164
缺少解析程式	1164
映射範本錯誤	1165
不正確的傳回類型	1165
處理無效的請求	1166
.....	mclxvii

什麼是 AWS AppSync ?

AWS AppSync 可讓開發人員使用安全、無伺服器且高效能的 GraphQL 和 Pub/Sub APIs，將其應用程式和服務連線至資料和事件。您可以使用 AWS AppSync 執行下列動作：

- 從單一 GraphQL API 端點存取一或多個資料來源的資料。
- 將多個來源 GraphQL APIs 合併為單一合併的 GraphQL API。
- 發佈即時資料更新到您的應用程式。
- 利用內建的安全性、監控、記錄和追蹤，以及選用的低延遲快取。
- 只需支付 API 請求和任何即時訊息交付的費用。

Important

自 2025 年 3 月 13 日起，您可以使用 AWS AppSync Events 建置由 WebSockets 支援的即時 PubSub API。如需詳細資訊，請參閱《AppSync Events 開發人員指南》中的[透過 WebSocket 發佈事件](#)。AWS AppSync

主題

- [AWS AppSync 功能](#)
- [您是第一次 AWS AppSync 使用嗎？](#)
- [相關服務](#)
- [的定價 AWS AppSync](#)

AWS AppSync 功能

- 由 GraphQL 提供支援的簡化資料存取和查詢
- GraphQL 訂閱和 pub/sub 通道的無WebSockets
- 伺服器端快取可在高速記憶體內快取中提供資料，以降低延遲
- JavaScript 和 TypeScript 支援撰寫商業邏輯
- 使用私有 APIs來限制 API 存取和整合的企業安全 AWS WAF
- 內建授權控制，支援 API 金鑰、IAM、Amazon Cognito、OpenID Connect 提供者和自訂邏輯的 Lambda 授權。

- 合併 APIs 以支援聯合使用案例

如需這些功能的詳細資訊，請參閱 [AWS AppSync 功能](#)。

您是第一次 AWS AppSync 使用嗎？

我們建議第一次 AWS AppSync 使用的使用者先閱讀以下章節：

- 如果您不熟悉 GraphQL，請參閱 [入門：建立您的第一個 GraphQL API](#)。
- 如果您要建置使用 GraphQL APIs 的應用程式，請參閱 [建置用戶端應用程式](#) 和 [the section called “針對即時資料應用程式使用訂閱”](#)。
- 如果您要尋找 GraphQL 解析程式資訊，請參閱以下內容：

JavaScript/TypeScript

- [解析程式教學課程 \(JavaScript\)](#)
- [解析程式參考 \(JavaScript\)](#)

VTL

- [解析程式教學課程 \(VTL\)](#)
- [解析程式映射範本參考 \(VTL\)](#)
- 如果您要尋找 AWS AppSync 範例專案、更新等，請參閱 [AppSync 部落格](#)。

相關服務

如果您要從頭開始建置 Web 或行動應用程式，請考慮使用 [AWS Amplify](#)。Amplify 利用 AWS AppSync 和其他 AWS 服務，協助您建置更強大、功能強大的 Web 和行動應用程式，減少工作量。

的定價 AWS AppSync

AWS AppSync 根據數百萬個請求和更新定價。快取需支付額外費用。如需詳細資訊，請參閱 [AWS AppSync 定價](#)。

下列列出一般 AWS AppSync 定價的例外狀況：

- 授權和驗證失敗的要求不會收取費用。
- API 金鑰遺失或無效時，不會收取呼叫需要 API 金鑰之方法的費用。

GraphQL 和 AWS AppSync 架構

Note

本指南假設使用者具備 REST 架構樣式的工作知識。建議您在使用 GraphQL 和 之前，先檢閱此主題和其他前端主題 AWS AppSync。

GraphQL 是 APIs 的查詢和操作語言。GraphQL 提供靈活且直覺式的語法來描述資料需求和互動。它可讓開發人員詢問確切所需的內容，並取得可預測的結果。它也可以在單一請求中存取許多來源，減少網路呼叫次數和頻寬需求，從而節省應用程式的電池壽命和 CPU 週期。

資料更新會隨著變動而變得簡單，讓開發人員能夠描述資料應如何變更。GraphQL 也可透過訂閱快速設定即時解決方案。所有這些功能結合強大的開發人員工具，使 GraphQL 成為管理應用程式資料的必要條件。

GraphQL 是 REST 的替代方案。RESTful 架構目前是用戶端與伺服器通訊的熱門解決方案之一。它以 URL 公開的資源（資料）概念為中心。這些 URLs 可用於透過 CRUD（建立、讀取、更新、刪除）操作存取和操作資料，其形式為 HTTP 方法 POST，例如 GET、和 DELETE。REST 的優點是，學習和實作相對簡單。您可以快速設定 RESTful APIs 來呼叫各種服務。

不過，技術變得越來越複雜。隨著應用程式、工具和服務開始為全球受眾擴展，對快速、可擴展架構的需求至關重要。REST 在處理可擴展操作時有許多缺點。如需範例，請參閱此[使用案例](#)。

在下列各節中，我們將檢閱一些有關 RESTful APIs 概念。然後，我們將介紹 GraphQL 及其運作方式。

如需 GraphQL 和遷移到 之優點的詳細資訊 AWS，請參閱 [GraphQL 實作的決策指南](#)。

主題

- [什麼是 API？](#)
- [什麼是 REST？](#)
- [什麼是 GraphQL？](#)
- [比較 REST 和 GraphQL](#)
- [為什麼在 REST 上使用 GraphQL？](#)
- [GraphQL API 的元件](#)
- [GraphQL 的其他屬性](#)

什麼是 API？

應用程式程式設計界面 (API) 定義了您必須遵循的規則，以便與其他軟體系統通訊。開發人員公開或建立 APIs，以便其他應用程式可以程式設計方式與其應用程式通訊。例如，時間表應用程式公開了 API，要求員工的完整姓名和日期範圍。當它收到此資訊時，它會在內部處理員工的時間表，並傳回在該日期範圍內工作的時數。

您可以將 Web API 視為 Web 上用戶端和資源之間的閘道。

用戶端

用戶端是想要從 Web 存取資訊的使用者。用戶端可以是使用 API 的人員或軟體系統。例如，開發人員可以撰寫程式，從天氣系統存取天氣資料。或者，當您直接造訪天氣網站時，您可以從瀏覽器存取相同的資料。

資源

資源是不同應用程式提供給其用戶端的資訊。資源可以是影像、影片、文字、數字或任何類型的資料。提供資源給用戶端的機器也稱為伺服器。Organizations 使用 APIs 來共用資源並提供 Web 服務，同時維護安全性、控制和身分驗證。此外，APIs 可協助他們判斷哪些用戶端可以存取特定的內部資源。

什麼是 REST？

在高階，表示式狀態轉移 (REST) 是一種軟體架構，會針對 API 的運作方式施加條件。REST 最初是建立為管理網際網路等複雜網路上通訊的指導方針。您可以使用以 REST 為基礎的架構，大規模支援高效能和可靠的通訊。您可以輕鬆實作和修改它，為任何 API 系統提供可見性和跨平台可攜性。

API 開發人員可以使用數種不同的架構來設計 APIs。遵循 REST 架構樣式的 APIs 稱為 REST APIs。實作 REST 架構的 Web 服務稱為 RESTful Web 服務。RESTful API 一詞通常是指 RESTful Web APIs。不過，您可以交替使用 REST API 和 RESTful API。

以下是 REST 架構樣式的一些原則：

統一界面

統一界面是任何 RESTful Web 服務設計的基礎。它表示伺服器以標準格式傳輸資訊。格式化的資源稱為 REST 中的表示法。此格式可能與伺服器應用程式上資源的內部表示方式不同。例如，伺服器可以將資料儲存為文字，但以 HTML 表示格式傳送。

統一界面會施加四個架構限制：

1. 請求應識別資源。他們使用統一的資源識別符來執行此操作。
2. 用戶端在資源表示法中有足夠的資訊，可視需要修改或刪除資源。伺服器透過傳送進一步描述資源的中繼資料來滿足此條件。
3. 用戶端會收到如何進一步處理表示的資訊。伺服器透過傳送包含有關用戶端如何最佳使用中繼資料的自我描述訊息來實現此目標。
4. 用戶端會收到完成任務所需的所有其他相關資源的相關資訊。伺服器會透過傳送 表示法中的超連結來達成此目的，以使用戶端動態探索更多資源。

無狀態

在 REST 架構中，無狀態是指一種通訊方法，伺服器會在其中完成每個用戶端請求，獨立於所有先前的請求。用戶端可以按任何順序請求資源，而且每個請求都是無狀態的，或與其他請求隔離。此 REST API 設計限制表示伺服器每次都能完全了解和滿足請求。

分層系統

在分層系統架構中，用戶端可以連線至用戶端和伺服器之間的其他授權中介，而且仍然會收到伺服器的回應。伺服器也可以將請求傳遞至其他伺服器。您可以設計 RESTful Web 服務，在多個伺服器上執行，其中包含多個層級，例如安全性、應用程式和商業邏輯，共同滿足用戶端請求。這些層對用戶端而言是隱藏的。

快取性

RESTful Web 服務支援快取，這是將一些回應儲存在用戶端或媒介上以改善伺服器回應時間的程序。例如，假設您造訪的網站在每個頁面上都有通用的標頭和頁尾影像。每次您造訪新的網站頁面時，伺服器都必須重新傳送相同的映像。為了避免這種情況，用戶端會在第一次回應後快取或存放這些映像，然後直接從快取使用映像。RESTful Web 服務會使用將自己定義為可快取或不可快取的 API 回應來控制快取。

什麼是 RESTful API？

RESTful API 是兩個電腦系統用來透過網際網路安全地交換資訊的界面。大多數商業應用程式必須與其他內部和第三方應用程式通訊，才能執行各種任務。例如，若要產生每月薪資單，您的內部帳戶系統必須與客戶的銀行系統共用資料，以自動開立發票並與內部時間表應用程式通訊。RESTful APIs 支援此資訊交換，因為它們遵循安全、可靠且高效的軟體通訊標準。

RESTful APIs如何運作？

RESTful API 的基本函數與瀏覽網際網路相同。當用戶端需要資源時，會使用 API 聯絡伺服器。API 開發人員說明用戶端應該如何使用伺服器應用程式 API 文件中的 REST API。以下是任何 REST API 呼叫的一般步驟：

1. 用戶端會將請求傳送至伺服器。用戶端遵循 API 文件，以伺服器了解的方式格式化請求。
2. 伺服器會驗證用戶端，並確認用戶端有權提出該請求。
3. 伺服器會收到請求並在內部處理。
4. 伺服器會傳回回應給用戶端。回應包含告知用戶端請求是否成功的資訊。回應也包含用戶端請求的任何資訊。

REST API 請求和回應詳細資訊會因 API 開發人員設計 API 的方式而略有不同。

什麼是 GraphQL？

GraphQL 既是 APIs 的查詢語言，也是執行這些查詢的執行時間。GraphQL 可讓用戶端請求所需的資料，在許多情況下提供更靈活且更有效率的 REST 替代方案。與依賴預先定義端點的 REST 不同，GraphQL 使用單一端點，其中用戶端可以以查詢和變動的形式指定其資料需求。

如需 [GraphQL API 結構的詳細資訊](#)，請參閱 [GraphQL API 的元件](#)。GraphQL APIs

比較 REST 和 GraphQL

APIs（應用程式程式設計介面）在促進應用程式之間的資料交換方面扮演重要角色。如前所述，設計 APIs 兩種重要方法已出現：GraphQL 和 REST。雖然兩者都是為了實現用戶端與伺服器通訊的基本目的，但其實作和使用案例有很大的差異。

GraphQL 和 REST 有幾個關鍵特性：

1. 用戶端-伺服器模型：兩者都使用用戶端-伺服器架構進行資料交換。
2. 無狀態：兩個 都不會在請求之間維護用戶端工作階段資訊。
3. HTTP 型：兩者通常使用 HTTP 做為基礎通訊協定。
4. 資源導向設計：兩者都圍繞資源設計其資料交換，這是指用戶端可以透過 API 存取和操作的任何資料或物件。
5. 資料格式彈性：JSON 是兩者中最常使用的資料交換格式，但也支援其他格式，例如 XML 和 HTML。

6. 語言和資料庫無關：兩者都可以使用任何程式設計語言或資料庫結構，使其高度可互通。
7. 快取支援：兩者都支援快取，允許用戶端和伺服器存放經常存取的資料，以提高效能。

GraphQL 和 REST 共用一些基本原則時，其 API 設計和資料擷取方法有顯著差異：

1. 請求結構和資料擷取

REST 使用不同的 HTTP 方法 (GET、POST、PUT、DELETE) 對資源執行操作。這通常需要不同資源的多個端點，這可能會導致資料擷取的效率低下。例如，執行 GET 操作來擷取使用者的資料可能會導致資料過度擷取或擷取不足。若要取得正確的資料，可能會呼叫截斷或多個操作。

GraphQL 對所有操作使用單一端點。它依賴於擷取資料和變動以修改資料的查詢。用戶端可以使用查詢來擷取在單一請求中所需的確切資料，進而透過將資料傳輸降至最低來降低網路負荷。

2. 伺服器端結構描述

REST 不需要伺服器端結構描述，但可以選擇性地定義結構描述，以實現高效的 API 設計和文件。

GraphQL 使用強類型伺服器端結構描述來定義資料和資料服務。以 GraphQL 結構描述定義語言 (SDL) 撰寫的結構描述包含每個物件的物件類型和欄位，以及定義每個欄位操作的伺服器端解析程式函數。

3. 版本控制

REST 通常在 URL 中包含版本控制，這可能會導致同時維護多個 API 版本。版本控制不是強制性的，但有助於防止重大變更。

GraphQL 透過要求回溯相容性，提升 API 的持續演變，而無需明確版本控制。刪除的欄位會傳回錯誤訊息，而棄用標籤會逐步淘汰舊欄位並傳回警告訊息。

4. 錯誤處理

REST 類型較弱，需要將錯誤處理內建到周圍的程式碼中。這可能不會自動識別與類型相關的錯誤（例如，將數字剖析為文字）。

相較之下，GraphQL 是強式類型，需要全面的結構描述定義。這可讓您的服務自動識別許多具有高度細節的請求錯誤。

5. 使用案例

REST 更適合：

- 具有較不複雜資料需求的小型應用程式。

- 所有用戶端以類似方式使用資料和操作的案例。
- 沒有複雜資料查詢需求的應用程式。

GraphQL 更適合：

- 頻寬有限的案例，其中將請求和回應降至最低至關重要。
- 具有多個資料來源且需要在單一端點合併的應用程式。
- 用戶端請求有顯著差異且預期不同回應結構的案例。

請注意，您可以在單一應用程式中使用 GraphQL 和 REST APIs，以用於不同的功能領域。此外，您可以升級 RESTful API 以包含 GraphQL 功能，而無需完全重寫。如需範例，請參閱[如何為 AWS 資料來源建置 GraphQL 解析程式](#)。

為什麼在 REST 上使用 GraphQL？

REST 是 Web APIs 的基石架構樣式之一。不過，隨著世界變得更相互連結，開發強大且可擴展的應用程式的需求將成為更緊迫的問題。雖然 REST 通常用於建置 Web APIs，但已識別 RESTful 實作的多個重複缺點：

1. 資料請求：使用 RESTful APIs，您通常會透過端點請求所需的資料。當您的資料可能無法妥善封裝時，就會發生問題。您需要的資料可能位於多層抽象後面，而擷取資料的唯一方法是使用多個端點，這表示提出多個請求來擷取所有資料。
2. 過度擷取和低擷取：若要將新增至多個請求的問題，會嚴格定義每個端點的資料，這表示即使您在技術上不需要該 API，您還是會傳回為該 API 定義的任何資料。

這可能會導致過度擷取，這表示我們的請求會傳回多餘的資料。例如，假設您正在請求公司人員資料，並想要知道特定部門的員工名稱。傳回資料的端點將包含名稱，但也可能包含其他資料，例如職稱或出生日期。由於 API 是固定的，您無法只請求名稱，其餘資料也隨附於其中。

相反地，我們未傳回足夠資料的情況稱為擷取不足。若要取得所有請求的資料，您可能需要向服務提出多個請求。根據資料的結構方式，您可能會遇到效率不佳的查詢，導致如可可怕的 $n+1$ 問題等問題。

3. 緩慢的開發反覆運算：許多開發人員會量身打造 RESTful APIs，以符合其應用程式的流程。不過，隨著應用程式的成長，前端和後端可能需要大量變更。因此，APIs 可能不再以有效率或影響的方式符合資料的形狀。由於需要 API 修改，這會導致產品反覆運算速度變慢。
4. 大規模效能：由於這些複合問題，許多領域會影響可擴展性。應用程式端的效能可能會受到影響，因為您的請求會傳回太多資料或太少（導致更多請求）。這兩種情況都會對網路造成不必要的壓

力，導致效能不佳。在開發人員方面，開發速度可能會降低，因為您的 APIs 是固定的，不再符合他們請求的資料。

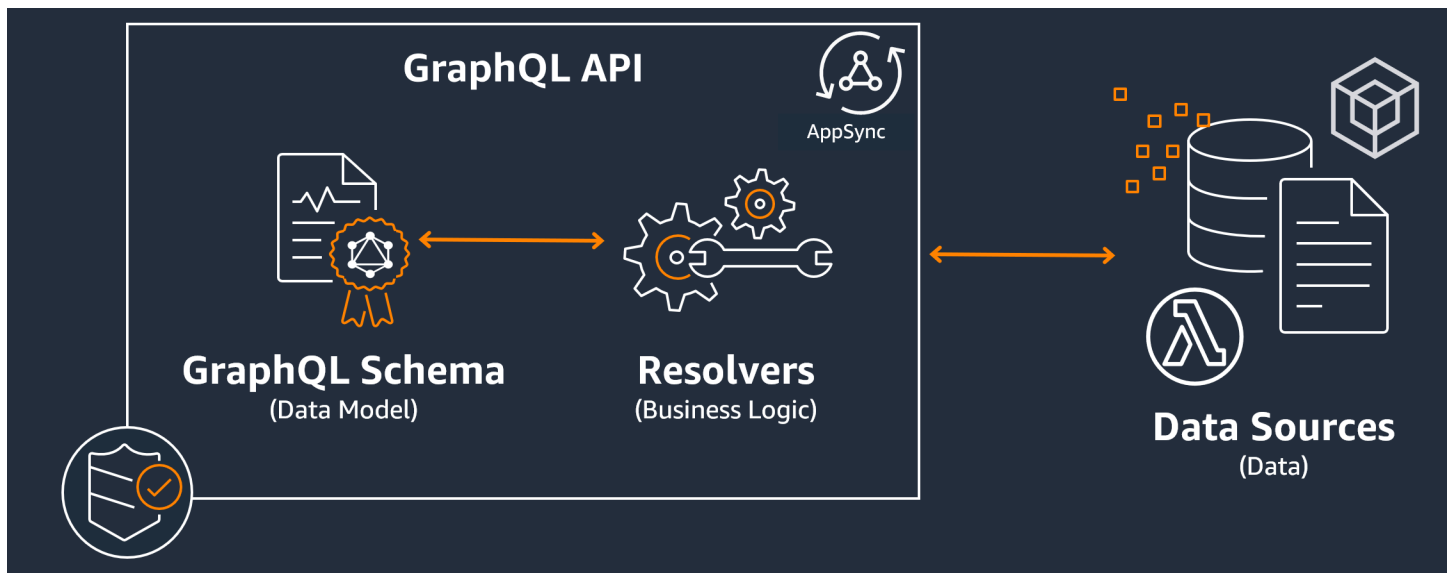
GraphQL 的賣點是克服 REST 的缺點。以下是 GraphQL 提供給開發人員的一些重要解決方案：

1. 單一端點：GraphQL 使用單一端點來查詢資料。您不需要建置多個 APIs 來符合資料的形狀。這會導致透過網路的請求較少。
2. 擷取：GraphQL 只需定義您需要的資料，即可解決擷取過多和不足的永久問題。GraphQL 可讓您將資料塑造成符合您的需求，因此您只會收到您要求的內容。
3. 抽象：GraphQL APIs 包含一些元件和系統，這些元件和系統使用語言無關的標準描述資料。換言之，資料的形狀和結構會標準化，因此前端和後端都知道如何透過網路傳送資料。這可讓兩端的開發人員使用 GraphQL 的系統，而不是在其中。
4. 快速反覆運算：由於資料標準化，可能不需要變更開發的一端。例如，前端呈現變更可能不會導致大量的後端變更，因為 GraphQL 允許立即修改資料規格。您可以直接定義或修改資料的形狀，以符合應用程式在成長時的需求。這會導致開發工作的可能性降低。

這些只是 GraphQL 的一些優點。在接下來的幾個區段中，您將了解 GraphQL 的結構方式，以及讓它成為 REST 獨特替代方案的屬性。

GraphQL API 的元件

標準 GraphQL API 由單一結構描述組成，可處理要查詢的資料形狀。您的結構描述會連結至一或多個資料來源，例如資料庫或 Lambda 函數。在兩者之間的中，有一或多個解析程式可處理您的請求的商業邏輯。每個元件在您的 GraphQL 實作中都扮演重要角色。以下各節將介紹這三個元件及其在 GraphQL 服務中扮演的角色。



主題

- [GraphQL 結構描述](#)
- [資料來源](#)
- [解析程式](#)

GraphQL 結構描述

GraphQL 結構描述是 GraphQL API 的基礎。它可做為定義資料形狀的藍圖。它也是用戶端和伺服器之間的合約，定義如何擷取和/或修改您的資料。

GraphQL 結構描述是以結構描述定義語言 (SDL) 撰寫。SDL 由具有已建立結構的類型和欄位組成：

- **類型：**類型是 GraphQL 如何定義資料的形狀和行為。GraphQL 支援許多類型，本節稍後會說明這些類型。結構描述中定義的每個類型都會包含自己的範圍。範圍內的一個或多個欄位，可以包含 GraphQL 服務中使用的值或邏輯。類型會填入許多不同的角色，最常見的是物件或純量（基本值類型）。
- **欄位：**欄位存在於類型的範圍內，並保留從 GraphQL 服務請求的值。這些與其他程式設計語言的變數非常相似。您在欄位中定義的資料形狀將決定資料在請求/回應操作中的結構。這可讓開發人員在不了解服務的後端實作方式的情況下預測將傳回的內容。

若要視覺化結構描述的外觀，讓我們檢閱簡單 GraphQL 結構描述的內容。在生產程式碼中，您的結構描述通常位於名為 `schema.graphql` 或 `schema.json` 的檔案中。假設我們正在對等至實作 GraphQL

服務的專案。此專案正在儲存公司人員資料，而 `schema.graphql` 檔案會用來擷取人員資料，並將新人員新增至資料庫。程式碼看起來可能如下所示：

`schema.graphql`

```
type Person {
  id: ID!
  name: String
  age: Int
}
type Query {
  people: [Person]
}
type Mutation {
  addPerson(id: ID!, name: String, age: Int): Person
}
```

我們可以看到結構描述中定義了三種類型：Person、Query和Mutation。查看Person，我們可以猜測這是公司員工執行個體的藍圖，這會讓此類型成為物件。在其範圍內，我們看到id、name和age。這些欄位定義的屬性Person。這表示我們的資料來源會將每個Person的String儲存name為純量（基本）類型和純量age Int（基本）類型。id可做為每個的特殊、唯一識別符Person。它也是以!符號表示的必要值。

接下來的兩個物件類型行為不同。GraphQL 會為特殊物件類型保留一些關鍵字，以定義資料在結構描述中填入的方式。Query 類型會從來源擷取資料。在我們的範例中，我們的查詢可能會從資料庫擷取Person物件。這可能會提醒您 RESTful 術語中的GET操作。Mutation 將修改資料。在我們的範例中，我們的變動可能會將更多Person物件新增至資料庫。這可能會提醒您 PUT或 等狀態變更操作POST。本節稍後將說明所有特殊物件類型的行為。

假設範例中Query的 會從資料庫擷取物件。如果我們查看 的欄位Query，就會看到一個名為 的欄位people。其欄位值為 [Person]。這表示我們希望擷取Person資料庫中的一些 執行個體。不過，新增括號表示我們想要傳回所有Person執行個體的清單，而不只是特定執行個體。

Mutation 類型負責執行變更狀態的操作，例如資料修改。變動負責對資料來源執行一些狀態變更操作。在我們的範例中，我們的變動包含名為 的操作addPerson，可將新Person物件新增至資料庫。變動會使用 ，Person並預期 id、name和age 欄位的輸入。

此時，您可能會想知道，如果沒有程式碼實作，像是 addPerson 操作的運作方式，因為它應該會執行一些行為，看起來就像是具有函數名稱和參數的函數。目前，它無法運作，因為結構描述僅做為宣告。若要實作 的行為addPerson，我們必須新增解析程式。解析程式是程式碼單位，每當呼叫其相關聯的

欄位時（在此案例中為 `addPerson` 操作）就會執行。如果您想要使用 操作，您必須在某個時間點新增解析程式實作。透過某種方式，您可以將結構描述操作視為函數宣告，並將解析程式視為定義。解析程式將在不同的章節中說明。

此範例僅顯示結構描述操作資料的最簡單方式。您可以利用 GraphQL 和 的功能來建置複雜、強大且可擴展的應用程式 AWS AppSync。在下一節中，我們將定義您可以在結構描述中使用的所有不同類型和欄位行為。

GraphQL 類型

GraphQL 支援許多不同的類型。如上一節所示，類型會定義資料的形狀或行為。它們是 GraphQL 結構描述的基本建置區塊。

類型可以分類為輸入和輸出。輸入是允許做為特殊物件類型 (`Query`、等) 引數傳入的類型 `Mutation`，而輸出類型會嚴格用於存放和傳回資料。類型及其分類的清單如下：

- 物件：物件包含描述實體的欄位。例如，物件可以是類似 的物件，`book` 其中包含描述其特性的欄位 `publishingYear`，例如 `authorName`、等。它們是嚴格輸出類型。
- 純量：這些是基本類型，例如 `int`、`string` 等。它們通常會指派給欄位。使用 `authorName` 欄位做為範例，可以指派 `String` 純量來存放名稱，例如 "John Smith"。純量可以是輸入和輸出類型。
- 輸入：輸入可讓您傳遞一組欄位做為引數。它們的結構與物件非常類似，但可以做為引數傳遞給特殊物件。輸入可讓您在範圍內定義純量、列舉和其他輸入。輸入只能是輸入類型。
- 特殊物件：特殊物件會執行狀態變更操作，並大量處理服務的繁重工作。有三種特殊物件類型：查詢、變動和訂閱。查詢通常會擷取資料；變動會操作資料；訂閱會開啟和維護用戶端和伺服器之間的雙向連線，以進行持續通訊。特殊物件的功能不是輸入或輸出。
- 列舉：列舉是預先定義的法律值清單。如果您呼叫列舉，其值只能是其範圍中定義的值。例如，如果您有一個名為 的列舉，`trafficLights` 描述流量訊號的清單，則其可能具有 `redLight` 和 等值，`greenLight` 但不會有 等值 `purpleLight`。真正的交通燈只會有這麼多訊號，因此您可以使用列舉來定義它們，並在參考 時強制它們成為唯一的法律值 `trafficLight`。列舉可以是輸入和輸出類型。
- 聯集/介面：聯集可讓您根據用戶端請求的資料，在請求中傳回一或多個物件。例如，如果您的 `Book` 類型具有 `title` 欄位，而 `Author` 類型具有 `name` 欄位，則可以在這兩種類型之間建立聯集。如果您的用戶端想要查詢資料庫以取得片語「Julius Caesar」，則聯集可以從 傳回 `Julius Caesar` (William Shakespeare 的播放)，`Booktitle` 並從 傳回 `Julius Caesar Author` (Anicarii de Bello Gallico 的作者) `name`。聯集只能是輸出類型。

介面是物件必須實作的一組欄位。這與 Java 等程式設計語言的介面略有相似，您必須在其中實作介面中定義的欄位。例如，假設您建立了名為 的介面 `Book`，其中包含 `title` 欄位。假設您稍後

建立了名為 `Novel` 的類型，該類型已實作 `Book`。您的 `Novel` 必須包含 `title` 欄位。不過，您的 `Novel` 也可以包含不在 界面中的其他欄位，例如 `pageCount` 或 `ISBN`。界面只能是輸出類型。

以下各節將說明每種類型如何在 GraphQL 中運作。

物件

GraphQL 物件是您將在生產程式碼中看到的主要類型。在 GraphQL 中，您可以將物件視為不同欄位的群組（類似於其他語言的變數），每個欄位都由可保留值的類型（通常是純量或其他物件）定義。物件代表可從您的服務實作擷取/操控的資料單位。

物件類型使用 `Type` 關鍵字宣告。讓我們稍微修改結構描述範例：

```
type Person {  
  id: ID!  
  name: String  
  age: Int  
  occupation: Occupation  
}  
  
type Occupation {  
  title: String  
}
```

此處的物件類型為 `Person` 和 `Occupation`。每個物件都有自己的欄位和自己的類型。GraphQL 的一項功能是能夠將欄位設定為其他類型。您可以在 中看到 `occupation` 欄位 `Person` 包含 `Occupation` 物件類型。我們可以建立此關聯，因為 GraphQL 只會描述資料，而不是服務的實作。

純量

純量基本上是保留值的基本類型。在 中 AWS AppSync，純量有兩種類型：預設 GraphQL 純量和 AWS AppSync 純量。純量通常用於在物件類型中存放欄位值。預設 GraphQL 類型包括 `Int`、`Float`、`Boolean`、`String` 和 `ID`。讓我們再次使用上一個範例：

```
type Person {  
  id: ID!  
  name: String  
  age: Int  
  occupation: Occupation  
}  
  
type Occupation {
```

```
title: String
}
```

拼出 `name` 和 `title` 欄位，兩者都保留純量。String Name 可能會傳回字串值，例如 "John Smith"，而標題可能會傳回類似 "firefighter" 的值。有些 GraphQL 實作也支援使用 Scalar 關鍵字的自訂純量，並實作類型的行為。不過，AWS AppSync 目前不支援自訂純量。如需純量清單，請參閱 [中的純量類型 AWS AppSync](#)。

輸入

由於輸入和輸出類型的概念，傳入引數時存在某些限制。通常需要傳入的類型會受到限制，特別是物件。您可以使用輸入類型來略過此規則。輸入是包含純量、列舉和其他輸入類型的類型。

輸入是使用 `input` 關鍵字定義：

```
type Person {
  id: ID!
  name: String
  age: Int
  occupation: Occupation
}

type Occupation {
  title: String
}

input personInput {
  id: ID!
  name: String
  age: Int
  occupation: occupationInput
}

input occupationInput {
  title: String
}
```

如您所見，我們可以有模擬原始類型的個別輸入。這些輸入通常會用於您的欄位操作，如下所示：

```
type Person {
  id: ID!
  name: String
}
```

```
    age: Int
    occupation: Occupation
  }

  type Occupation {
    title: String
  }

  input occupationInput {
    title: String
  }

  type Mutation {
    addPerson(id: ID!, name: String, age: Int, occupation: occupationInput): Person
  }
```

請注意，我們仍然會傳遞 `occupationInput` `Occupation` 來建立 `Person`。

這只是輸入的一個案例。它們不一定需要 1:1 複製物件，而且在生產程式碼中，您很可能不會像這樣使用物件。最佳實務是利用 GraphQL 結構描述，只需定義您需要輸入的內容做為引數即可。

此外，相同的輸入可用於多個操作，但我們不建議這樣做。每個操作理想情況下都應包含自己的唯一輸入複本，以防結構描述的需求變更。

特殊物件

GraphQL 會為特殊物件保留一些關鍵字，以定義結構描述如何擷取/處理資料的一些商業邏輯。結構描述中最多可以有其中一個關鍵字。它們可做為用戶端針對 GraphQL 服務執行之所有請求資料的進入點。

也會使用 `type` 關鍵字定義特殊物件。雖然它們的使用方式與一般物件類型不同，但其實作非常類似。

Queries

查詢與 `GET` 操作非常相似，因為它們執行唯讀擷取以從您的來源取得資料。在 GraphQL 中，`Query` 會定義用戶端對伺服器提出請求的所有進入點。`Query` 您的 GraphQL 實作中一律會有。

以下是我們在先前結構描述範例中使用的 `Query` 和 修改過的物件類型：

```
type Person {
  id: ID!
  name: String
  age: Int
```

```
    occupation: Occupation
  }
  type Occupation {
    title: String
  }
  type Query {
    people: [Person]
  }
```

我們的 Query 包含一個名為 `people` 的欄位，從資料來源傳回 `Person` 執行個體清單。假設我們需要變更應用程式的行為，現在我們需要針對一些不同的用途，僅傳回 `Occupation` 執行個體的清單。我們可以直接將其新增至查詢：

```
type Query {
  people: [Person]
  occupations: [Occupation]
}
```

在 GraphQL 中，我們可以將查詢視為請求的單一來源。如您所見，這比 RESTful 實作更簡單，這些實作可能會使用不同的端點來達成相同的物件 (`.../api/1/people` 和 `.../api/1/occupations`)。

假設我們對此查詢有解析程式實作，現在可以執行實際的查詢。當 Query 類型存在時，我們必須明確呼叫它，才能在應用程式的程式碼中執行。這可以使用 `query` 關鍵字來完成：

```
query getItems {
  people {
    name
  }
  occupations {
    title
  }
}
```

如您所見，此查詢稱為 `getItems` 並傳回 `people` (`Person` 物件清單) 和 `occupations` (`Occupation` 物件清單)。在 `people` 中，我們只會傳回每個的 `name` 欄位 `Person`，同時傳回每個的 `title` 欄位 `Occupation`。回應可能如下所示：

```
{
  "data": {
    "people": [
```

```

    {
      "name": "John Smith"
    },
    {
      "name": "Andrew Miller"
    },
    .
    .
    .
  ],
  "occupations": [
    {
      "title": "Firefighter"
    },
    {
      "title": "Bookkeeper"
    },
    .
    .
    .
  ]
}
}

```

範例回應顯示資料如何遵循查詢的形狀。擷取的每個項目都會列在 欄位的範圍內。people和occupations會以個別清單傳回物件。雖然有用，但修改查詢以傳回人員名稱和職業清單可能更為方便：

```

query getItems {
  people {
    name
    occupation {
      title
    }
  }
}

```

這是合法修改，因為我們的Person類型包含 類型的occupation欄位Occupation。當在 範圍內列出時people，我們會傳回每個 Person的 ，name以及其Occupation與 相關聯的 title。回應可能如下所示：

```

}
  "data": {

```

```

    "people": [
      {
        "name": "John Smith",
        "occupation": {
          "title": "Firefighter"
        }
      },
      {
        "name": "Andrew Miller",
        "occupation": {
          "title": "Bookkeeper"
        }
      },
      .
      .
      .
    ]
  }
}

```

Mutations

變動類似於狀態變更的操作，例如 PUT 或 POST。他們會執行寫入操作來修改來源中的資料，然後擷取回應。它們會定義資料修改請求的進入點。與查詢不同，根據專案的需求，變動可能會也可能不會包含在結構描述中。以下是結構描述範例中的變動：

```

type Mutation {
  addPerson(id: ID!, name: String, age: Int): Person
}

```

`addPerson` 欄位代表一個 `Person` 將新增至資料來源的進入點。`addPerson` 是欄位名稱；`id`、`name` 和 `age` 是參數；`Person` 是傳回類型。回顧 `Person` 類型：

```

type Person {
  id: ID!
  name: String
  age: Int
  occupation: Occupation
}

```

我們已新增 `occupation` 欄位。不過，我們無法直接將此欄位設定為 `Occupation` 因為物件無法以引數形式傳入；它們是嚴格輸出類型。我們應該改為傳遞具有與引數相同欄位的輸入：

```
input occupationInput {  
  title: String  
}
```

我們也可以在建立新Person執行個體時，輕鬆更新我們的 `addPerson` 以包含此參數：

```
type Mutation {  
  addPerson(id: ID!, name: String, age: Int, occupation: occupationInput): Person  
}
```

以下是更新的結構描述：

```
type Person {  
  id: ID!  
  name: String  
  age: Int  
  occupation: Occupation  
}  
  
type Occupation {  
  title: String  
}  
  
input occupationInput {  
  title: String  
}  
  
type Mutation {  
  addPerson(id: ID!, name: String, age: Int, occupation: occupationInput): Person  
}
```

請注意，`occupation`將從 傳入 `title` 欄位`occupationInput`，以完成 的建立，`Person`而不是原始`Occupation`物件。假設我們有適用於 的解析程式實作`addPerson`，我們現在可以執行實際的變動。當 `Mutation`類型存在時，我們必須明確呼叫它，才能在應用程式的程式碼中執行。這可以使用 `mutation`關鍵字來完成：

```
mutation createPerson {  
  addPerson(id: ID!, name: String, age: Int, occupation: occupationInput) {  
    name  
    age  
    occupation {
```

```

        title
      }
    }
  }
}
```

此變動稱為 `createPerson`，`addPerson` 是操作。若要建立新的 `Person`，我們可以輸入 `id`、`age`、`name` 和 `occupation` 的引數。在範圍內 `addPerson`，我們也可以看到其他欄位 `age`，例如 `name`、等。這是您的回應；這些是 `addPerson` 操作完成後將傳回的欄位。以下是範例的最後部分：

```

mutation createPerson {
  addPerson(id: "1", name: "Steve Powers", age: "50", occupation: "Miner") {
    id
    name
    age
    occupation {
      title
    }
  }
}
```

使用此變動時，結果可能如下所示：

```

{
  "data": {
    "addPerson": {
      "id": "1",
      "name": "Steve Powers",
      "age": "50",
      "occupation": {
        "title": "Miner"
      }
    }
  }
}
```

如您所見，回應會以與變動中定義的相同格式傳回我們請求的值。最好傳回所有修改過的值，以減少混淆，以及未來需要更多查詢。變動可讓您在範圍內包含多個操作。它們將依變動中列出的順序依序執行。例如，如果我們建立另一個名為 `addOccupation` 的操作，將任務標題新增至資料來源，我們可以在之後的變動中呼叫此 `addPerson`。`addPerson` 會先處理，再處理 `addOccupation`。

Subscriptions

訂閱使用 [WebSockets](#) 在伺服器及其用戶端之間開啟持久的雙向連線。一般而言，用戶端會訂閱或接聽伺服器。每當伺服器進行伺服器端變更或執行事件時，訂閱的用戶端都會收到更新。當訂閱多個用戶端且需要通知伺服器或其他用戶端發生變更時，此類型的通訊協定非常有用。例如，訂閱可用來更新社交媒體摘要。使用者 A 和使用者 B 可能有兩個使用者，他們每當收到直接訊息時都會訂閱自動通知更新。用戶端 A 上的使用者 A 可以傳送直接訊息給用戶端 B 上的使用者 B。使用者 A 的用戶端會傳送直接訊息，該訊息將由伺服器處理。然後，伺服器會在傳送自動通知至用戶端 B 時，將直接訊息傳送至使用者 B 的帳戶。

以下是Subscription我們可以新增至結構描述範例的 範例：

```
type Subscription {  
  personAdded: Person  
}
```

每當有新的 新增至資料來源Person時，personAdded 欄位就會傳送訊息給訂閱的用戶端。假設我們有適用於 的解析程式實作personAdded，我們現在可以使用訂閱。當 Subscription類型存在時，我們必須明確呼叫它，才能在應用程式的程式碼中執行。這可以使用 subscription關鍵字來完成：

```
subscription personAddedOperation {  
  personAdded {  
    id  
    name  
  }  
}
```

訂閱稱為 personAddedOperation，操作為 personAdded。personAdded會傳回新Person執行個體的 id和 name 欄位。查看變動範例，我們使用Person此操作新增：

```
addPerson(id: "1", name: "Steve Powers", age: "50", occupation: "Miner")
```

如果用戶端訂閱了新新增 的更新Person，則在addPerson執行後可能會看到：

```
{  
  "data": {  
    "personAdded": {  
      "id": "1",  
      "name": "Steve Powers"  
    }  
  }  
}
```

```
}  
}  
}
```

以下是訂閱提供的摘要：

訂閱是雙向通道，可讓用戶端和伺服器接收快速但穩定的更新。他們通常會使用 WebSocket 通訊協定，這會建立標準化且安全的連線。

訂閱可靈活運作，減少連線設定開銷。訂閱後，用戶端可以長時間繼續在該訂閱上執行。他們通常會允許開發人員量身打造訂閱的生命週期，並設定將請求哪些資訊，以有效率地使用運算資源。

一般而言，訂閱可讓用戶端一次進行多個訂閱。由於與相關 AWS AppSync，訂閱僅用於從 AWS AppSync 服務接收即時更新。它們無法用於執行查詢或變動。

訂閱的主要替代方案是輪詢，它會以設定的間隔傳送查詢以請求資料。此程序通常比訂閱效率低，並對用戶端和後端造成很大壓力。

我們的結構描述範例中未提及的一點是，您的特殊物件類型也必須在 schema 根中定義。因此，當您在中匯出結構描述時 AWS AppSync，其可能如下所示：

schema.graphql

```
schema {  
  query: Query  
  mutation: Mutation  
  subscription: Subscription  
}  
  
.  
.  
.  
  
type Query {  
  # code goes here  
}  
type Mutation {  
  # code goes here  
}  
type Subscription {  
  # code goes here  
}
```

列舉

列舉或列舉是限制類型或欄位可能具有之法律引數的特殊純量。這表示每當列舉在結構描述中定義時，其相關聯的類型或欄位將限制為列舉中的值。列舉會序列化為字串純量。請注意，不同的程式設計語言可能會以不同的方式處理 GraphQL 列舉。例如，JavaScript 沒有原生列舉支援，因此列舉值可以改為映射至整數值。

列舉是使用 `enum` 關鍵字定義。範例如下：

```
enum trafficSignals {  
  solidRed  
  solidYellow  
  solidGreen  
  greenArrowLeft  
  ...  
}
```

呼叫 `trafficLights` 列舉時，引數只能是 `solidRed`、`solidYellow`、`solidGreen` 等。使用列舉來描述具有不同但有限選擇數量的物件是很常見的。

聯集/界面

請參閱 [GraphQL 中的界面和聯集](#)。GraphQL

GraphQL 欄位

欄位存在於類型的範圍內，並保留從 GraphQL 服務請求的值。這些與其他程式設計語言的變數非常相似。例如，以下是 `Person` 物件類型：

```
type Person {  
  name: String  
  age: Int  
}
```

在此情況下，欄位分別為 `name` 和 `age` 並保留 `String` 和 `Int` 值。如上所示的物件欄位可以用作查詢和變動欄位（操作）中的輸入。例如，請參閱 `Query` 以下：

```
type Query {  
  people: [Person]  
}
```

people 欄位正在Person從資料來源請求 的所有執行個體。當您Person在 GraphQL 伺服器上新增或擷取 時，您可以預期資料會遵循您類型和欄位的格式，也就是說，結構描述中的資料結構會決定如何在回應中建構：

```

}
  "data": {
    "people": [
      {
        "name": "John Smith",
        "age": "50"
      },
      {
        "name": "Andrew Miller",
        "age": "60"
      },
      .
      .
      .
    ]
  }
}

```

欄位在建構資料中扮演重要角色。以下說明了幾個額外的屬性，可以套用至欄位以進行更多自訂。

清單

清單會傳回指定類型的所有項目。您可以使用括號 將清單新增至欄位的類型[]：

```

type Person {
  name: String
  age: Int
}
type Query {
  people: [Person]
}

```

在 中Query，括號Person表示您想要Person從資料來源傳回 的所有執行個體做為陣列。在回應中，每個的 name和 age值Person會以單一分隔清單傳回：

```

}
  "data": {

```

```

    "people": [
      {
        "name": "John Smith",      # Data of Person 1
        "age": "50"
      },
      {
        "name": "Andrew Miller",  # Data of Person 2
        "age": "60"
      },
      .
      .
      .
    ]
  }
}

```

您不限於特殊物件類型。您也可以在一般物件類型的欄位中使用清單。

非 Null

非 null 表示回應中不可為 null 的欄位。您可以使用 ! 符號將欄位設定為非空值：

```

type Person {
  name: String!
  age: Int
}
type Query {
  people: [Person]
}

```

name 欄位不能明確為 null。如果您要查詢資料來源，並為此欄位提供 null 輸入，則會擲回錯誤。

您可以結合清單和非 Null。比較這些查詢：

```

type Query {
  people: [Person!]      # Use case 1
}

.
.
.

type Query {

```

```

people: [Person]!    # Use case 2
}

.
.
.

type Query {
  people: [Person]!    # Use case 3
}

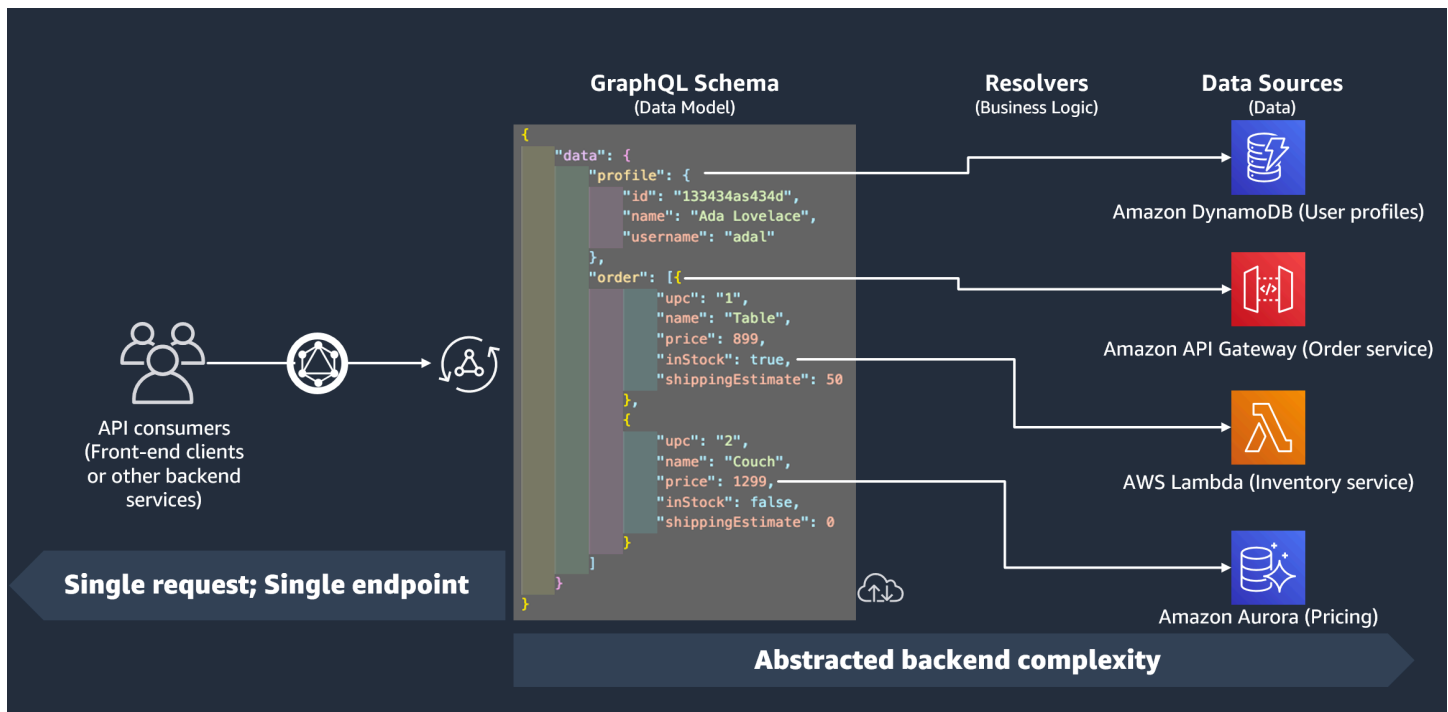
```

在使用案例 1 中，清單不能包含 null 項目。在使用案例 2 中，清單本身無法設定為 null。在使用案例 3 中，清單及其項目不能為 null。不過，無論如何，您仍然可以傳回空白清單。

如您所見，GraphQL 中有許多移動元件。在本節中，我們展示了簡單結構描述的結構，以及結構描述支援的不同類型和欄位。在下一節中，您將探索 GraphQL API 的其他元件，以及它們如何使用結構描述。

資料來源

在上一節中，我們了解到結構描述會定義資料的形狀。不過，我們從未說明該資料的來源。在實際專案中，您的結構描述就像是處理對伺服器提出之所有請求的閘道。提出請求時，結構描述會做為與用戶端連接的單一端點。結構描述將存取、處理資料，並將資料從資料來源轉送回用戶端。請參閱下列資訊圖表：



AWS AppSync 和 GraphQL 完美實作後端前端 (BFF) 解決方案。它們可串聯運作，透過抽象化後端來降低大規模複雜性。如果您的服務使用不同的資料來源和/或微服務，基本上您可以透過在單一結構描述（超級圖）中定義每個來源（子圖）的資料形狀，來抽象化一些複雜性。這表示您的 GraphQL API 不限於使用一個資料來源。您可以將任意數量的資料來源與 GraphQL API 建立關聯，並在程式碼中指定它們將如何與服務互動。

如您在資訊圖表中所見，GraphQL 結構描述包含用戶端請求資料所需的所有資訊。這表示一切都可以在單一請求中處理，而不是像使用 REST 的情況一樣處理多個請求。這些請求會經歷結構描述，這是服務的唯一端點。處理請求時，解析程式（在下一節中說明）會執行其程式碼來處理來自相關資料來源的資料。傳回回應時，與資料來源繫結的子圖形會填入結構描述中的資料。

AWS AppSync 支援許多不同的資料來源類型。在下表中，我們將描述每種類型、列出每種類型的一些優點，並提供實用連結以用於其他內容。

資料來源	描述	優勢	補充資訊
Amazon DynamoDB	「Amazon DynamoDB 是全受管的 NoSQL 資料庫服務，可提供快速且可預測的效能和無縫的可擴展性。DynamoDB 是全受管的 NoSQL 資料庫服務，可讓您卸下操作及擴展分散式資料庫的管理負擔，不再需要煩惱硬體佈建、設定和組態、複寫、軟體修補或叢集擴展。DynamoDB 也提供靜態加密，消除保護敏感資料所涉及的操作負擔和複雜性。」	- 大規模效能：DynamoDB 的設計是以任何規模的一致效能為目標。這可透過使用分割區來實現。DynamoDB 會自動將您的資料表分割為數個配置，這些配置將儲存在多個節點的多個 SSDs 中。這通常會增加網路輸送量並減少延遲。 - 大規模容量：DynamoDB 會監控您的流量，並允許您在網路長時間維持超載時自動擴展輸送量。 - 可用性和容錯能力：DynamoDB	DynamoDB 官方文件 分區 自動擴展 容錯能力 監控 安全性 GraphQL 和 DynamoDB DynamoDB 的解析程式操作 定價模型

資料來源	描述	優勢	補充資訊
		由數個實體隔離的區域支援，每個區域都包含數個實體隔離的可用區域。DynamoDB 會在服務中斷時自動切換到備份區域。您也可以手動備份和複寫資料，以確保資料安全。 • 記錄和監控： DynamoDB 為您的資料表提供數種分析工具。您可以監控資料表的效能並建立警示，以通知您服務的重大變更。 • 安全性： DynamoDB 遵循嚴格的通訊協定，以確保您的資料符合組織的安全需求。 • 與 AWS AppSync 的整合： DynamoDB 與我們的服務無縫整合。您可以建立新的 DynamoDB 資料表，並自動從它們產生結構描述，以簡化您的開發程序。我們也提供完整的操作集合，	

資料來源	描述	優勢	補充資訊
		可輕鬆地從解析程式中帳戶中現有的 DynamoDB 資料表請求資料。	

資料來源	描述	優勢	補充資訊
AWS Lambda	「AWS Lambda 是一種運算服務，可讓您執行程式碼，而無需佈建或管理伺服器。 Lambda 在高可用性的運算基礎設施上執行您的程式碼，並執行所有運算資源的管理，包括伺服器與作業系統維護、容量佈建與自動擴展以及記錄。使用 Lambda，您只需要在 Lambda 支援的其中一種語言執行時間中提供程式碼。」	• Pay-as-you-use 模型：Lambda 只會在您使用其資源時向您收費。它們也可讓您擴展用於應用程式需求的資源量。 • 自動擴展：有時您的應用程式可能需要針對特定程序使用額外的運算能力。Lambda 可讓您自動擴展運算資源，以符合應用程式的需求。 • 更快的部署時間：您可以透過部署套件簡化開發程序。使用套件將您的函數程式碼上傳至 Lambda 服務。然後，您可以使用其執行時間環境來測試和執行您的函數。 • 多樣化：Lambda 可用於多種使用案例。您可以無縫整合 Lambda 與第三方服務和 AWS 服務。一些範例包括 CI/CD 管道 和 大量郵寄服務。	• 官方文件 • 擴展 • 部署 • 執行時間 • Lambda 解析程式教學課程 • 定價模型

資料來源	描述	優勢	補充資訊
		與 AWS AppSync 的整合：您可以輕鬆地在解析程式中調用 Lambda 函數來處理請求。我們的服務提供簡化的請求操作，以執行 Lambda 呼叫。我們允許單一和批次呼叫。	

資料來源	描述	優勢	補充資訊
OpenSearch	「Amazon OpenSearch Service 是一項受管服務，可讓您輕鬆地在 AWS 雲端中部署、操作和擴展 OpenSearch 叢集。Amazon OpenSearch Service 支援 OpenSearch 和舊版 Elasticsearch OSS（最高 7.10，軟體的最終開放原始碼版本）。在您建立叢集時，您可選擇要使用的搜尋引擎。 OpenSearch 是一個完全開源的搜尋和分析引擎，適用於例如日誌分析、即時應用程式監控及點擊流分析等使用案例。如需詳細資訊，請參閱 OpenSearch 文件。 Amazon OpenSearch Service 會為您的 OpenSearch 叢集佈建所有資源並啟動它。它還會自動偵測和更換發生故障的 OpenSearch Service 節點，可降低與自我管理的基礎設施相關的營運成本。您可以	- 擴展：您可以透過 OpenSearch Serverless 輕鬆擴展服務以符合您的服務需求。 - 資料擷取：您可以使用 OpenSearch Ingestion 匯入、處理和分析資料。您可以在這裡找到許多用於資料擷取的應用程式。 - 安全：OpenSearch 可以管理您的 AWS 安全組態，包括 IAM、Cloud Trail、VPCs、身分驗證等。 - 可用性：OpenSearch 在其服務中也支援不同的區域和可用區域。 - 與 AWS AppSync 整合：In AWS AppSync，您可以使用 GraphQL APIs 來儲存和擷取帳戶中現有 OpenSearch Service 網域的資料。	官方文件 無伺服器 定價模型

資料來源	描述	優勢	補充資訊
	使用單一 API 呼叫或在主控台中按幾下滑鼠來擴展叢集。」		
HTTP 端點	您可以使用 HTTP 端點做為資料來源。AWS AppSync 可以傳送請求到端點，其中包含參數和承載等相關資訊。HTTP 回應將公開給解析程式，它將在完成其操作後傳回最終回應 (s)。	適用於未與 Lambda 等服務整合的簡單應用程式。	Resolver 參考

資料來源	描述	優勢	補充資訊
Amazon EventBridge	「EventBridge 是一種無伺服器服務，使用事件將應用程式元件連接在一起，讓您更輕鬆地建置可擴展的事件驅動型應用程式。使用它，將事件從諸如自家開發應用程式、AWS 服務和第三方軟體等來源路由到整個組織的消費者應用程式。EventBridge 提供簡單且一致的方法來擷取、篩選、轉換和交付事件，以便您可以快速建立新應用程式。」	• 事件驅動型架構：您可以利用事件驅動型架構。 • 排程：您可以使用 EventBridge 排程器，使用 Cron 表達式自動化任務和規則，或設定時間間隔作為事件模式的替代方案。 • 管道：使用 EventBridge 管道，您可以將事件匯流排取代為管道，其中包含額外的篩選事件模式，以及在將事件傳送至目標之前透過資料轉換擴充。 • 與 AWS AppSync：AWS AppSync 整合可讓您使用解析程式將事件傳送至事件匯流排。	• 官方文件 • 管道 • 排程器 • Resolver 參考 • 定價模型

資料來源	描述	優勢	補充資訊
關聯式資料庫	「Amazon Relational Database Service (Amazon RDS) 是一種 Web 服務，可讓您更輕鬆地在 AWS 雲端中設定、操作和擴展關聯式資料庫。它為產業標準關聯式資料庫提供經濟實惠、可擴展的容量，並管理常見的資料庫管理任務。」	- 輕鬆管理：RDS 會定期對其資源執行維護。維護通常涉及資料庫執行個體的基礎硬體、基礎作業系統 (OS) 或資料庫引擎版本的更新。在正常情況下，您可以決定何時執行更新（例外狀況包括安全性修補程式）。 - 建議：RDS 的建議功能提供自動化建議，以修正執行個體中的潛在問題。 - 可用性：RDS 適用於世界各地的不同實體區域。您可以輕鬆地將資料庫需求分散到不同的節點，為客戶提供更好的服務。 - 自訂：RDS 專為滿足大型企業的需求量身打造。RDS 提供各種運算、快速部署、可擴展性和儲存的選項。 - 安全性：RDS 與數種工具和服務整合，以維護使用者、資料庫和網路	官方文件 功能 Maintenance (維護) 建議 儲存選項 可用性 安全性 定價模型

資料來源	描述	優勢	補充資訊
		層級的資料庫安全性。 • 與 AWS AppSync 整合：如果您正在尋找成熟的後端解決方案，AWS AppSync 可讓您使用執行個體做為資料來源傳送、處理、存放和傳回資料。	
無資料來源	如果您不打算使用資料來源服務，您可以將其設定為 none。雖然 none 資料來源仍明確分類為資料來源，但 不是儲存媒體。儘管如此，它在某些執行個體中仍然適用於資料處理和傳遞。	• 可能適用於資料轉換等項目 • 在本機解析某個項目時很有用	• Resolver 參考

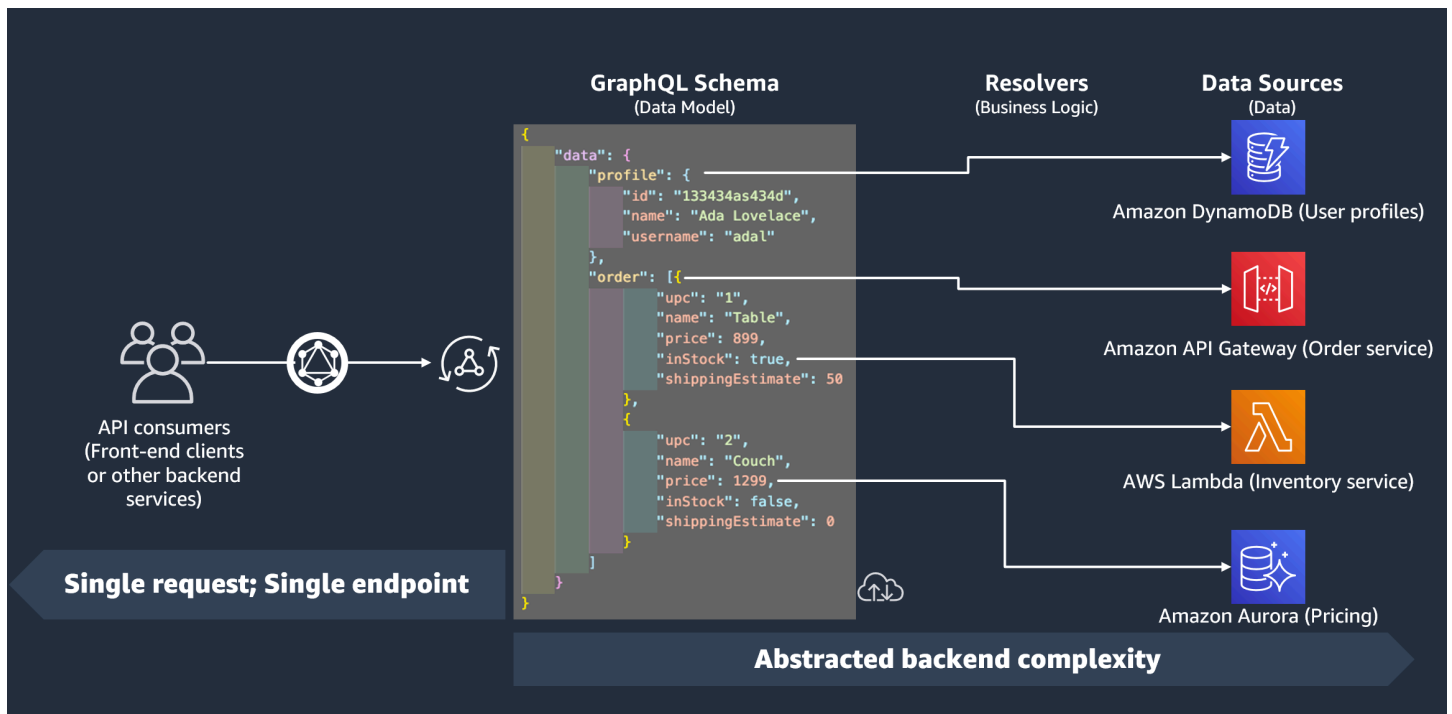
Tip

如需資料來源如何與 互動的詳細資訊 AWS AppSync，請參閱[連接資料來源](#)。

解析程式

從前幾節中，您已了解結構描述和資料來源的元件。現在，我們需要解決結構描述和資料來源的互動方式。一切都從解析程式開始。

解析程式是程式碼單位，可處理在向服務提出請求時，該欄位的資料將如何解析。解析程式會連接到您結構描述中類型內的特定欄位。它們最常用於實作查詢、變動和訂閱欄位操作的狀態變更操作。解析程式會處理用戶端的請求，然後傳回結果，其可以是一組輸出類型，例如物件或純量：



Resolver 執行時間

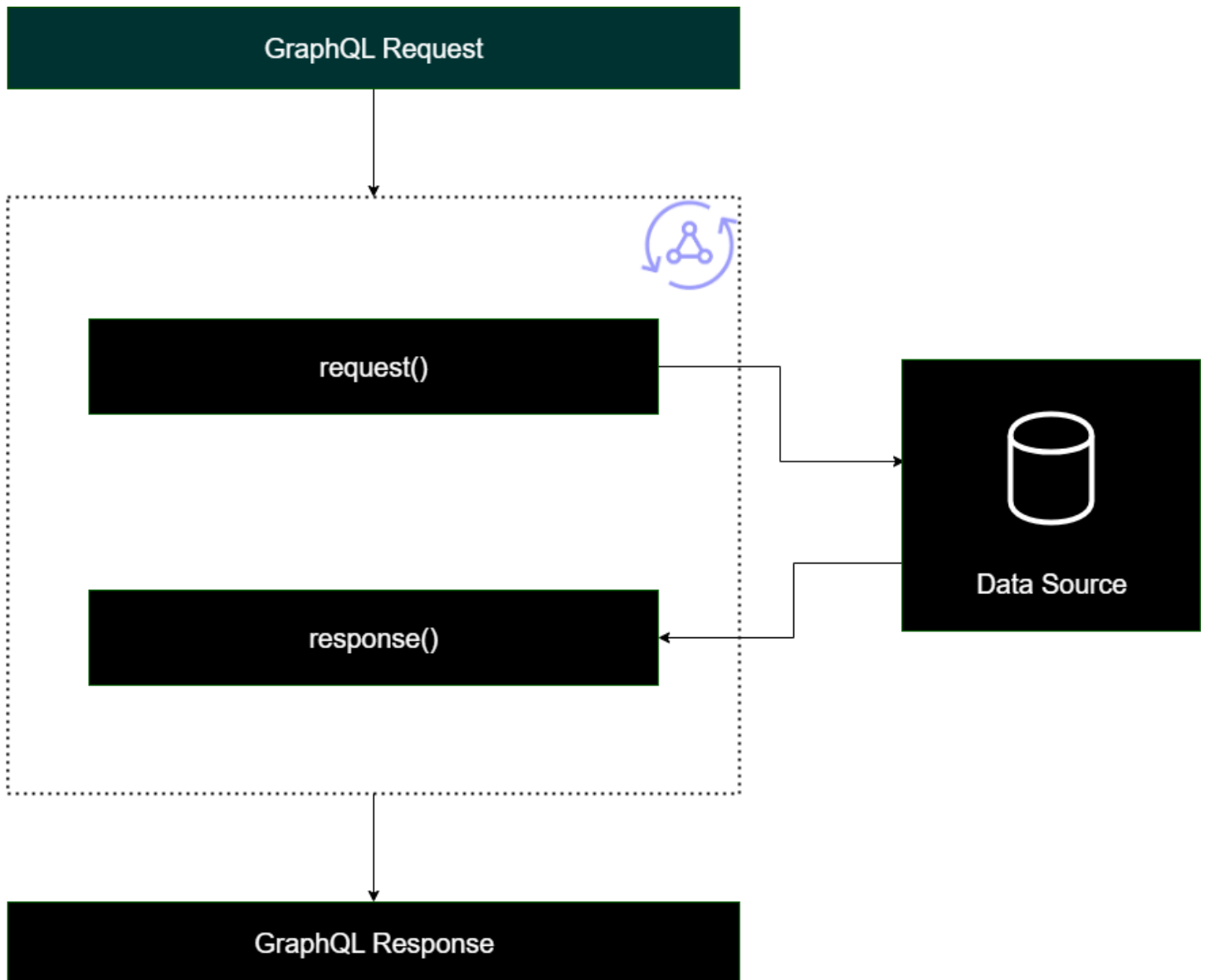
在 AWS AppSync 中，您必須先指定解析程式的執行時間。解析程式執行時間表示執行解析程式的環境。它還指定解析程式將寫入的語言。AWS AppSync 目前支援適用於 JavaScript 和 Velocity 範本語言 (VTL) 的 APPSYNC_JS。請參閱[適用於 JavaScript 的解析程式和函數的 JavaScript 執行期功能](#)，或適用於 VTL 的[解析程式映射範本公用程式參考](#)。JavaScript

解析程式結構

解析程式可以透過幾種方式進程式碼化建構。有單元和管道解析程式。

單位解析程式

單位解析程式由程式碼組成，可定義單一請求和回應處理常式，這些請求和回應處理常式會針對資料來源執行。請求處理常式會將內容物件做為引數，並傳回用來呼叫資料來源的請求承載。回應處理常式會從資料來源接收承載，其中包含執行請求的結果。回應處理常式會將承載轉換為 GraphQL 回應，以解析 GraphQL 欄位。



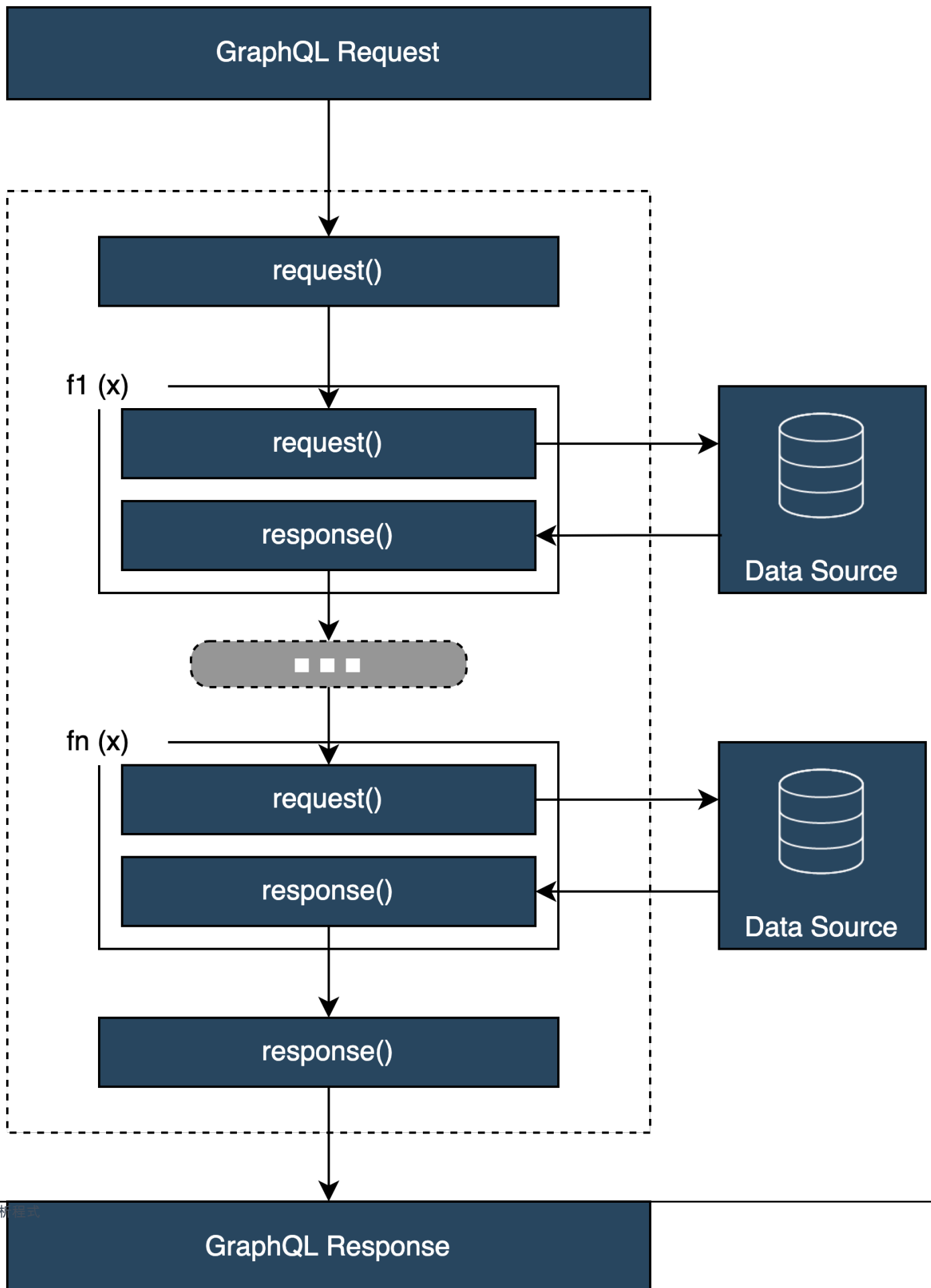
管道解析程式

實作管道解析程式時，會遵循一般結構：

- 在步驟之前：當用戶端提出請求時，所使用結構描述欄位的解析程式（通常是您的查詢、變動、訂閱）會傳遞請求資料。解析程式會開始在步驟處理常式之前處理請求資料，這允許在資料通過解析程式之前執行一些預先處理操作。
- Function(s)：在步驟執行之前，請求會傳遞至函數清單。清單中的第一個函數將針對資料來源執行。函數是解析程式程式碼的子集，其中包含自己的請求和回應處理常式。請求處理常式會取得請求資料，並對資料來源執行操作。回應處理常式會先處理資料來源的回應，再將其傳回清單。如果有多個

函數，請求資料將傳送至清單中要執行的下一個函數。清單中的函數將依開發人員定義的順序序列執行。執行所有函數後，最終結果會傳遞至步驟後。

- 後步驟：後步驟是處理常式函數，可讓您在最終函數的回應上執行一些最終操作，然後再將其傳遞至 GraphQL 回應。



Resolver 處理常式結構

處理常式通常是稱為 Request 和 的函數Response：

```
export function request(ctx) {  
  // Code goes here  
}  
  
export function response(ctx) {  
  // Code goes here  
}
```

在單位解析程式中，只有一組這些函數。在管道解析程式中，在步驟前後會有一組，每個函數還會有一組額外的。為了視覺化此狀況的外觀，讓我們檢閱簡單的Query類型：

```
type Query {  
  helloWorld: String!  
}
```

這是一個簡單的查詢，有一個欄位稱為helloWorld類型 String。假設我們一律希望此欄位傳回字串 "Hello World"。若要實作此行為，我們需要將解析程式新增至此欄位。在單位解析程式中，我們可以新增類似以下內容：

```
export function request(ctx) {  
  return {}  
}  
  
export function response(ctx) {  
  return "Hello World"  
}
```

request 可以只保留空白，因為我們沒有請求或處理資料。我們也可以假設資料來源為 None，表示此程式碼不需要執行任何調用。回應只會傳回「Hello World」。若要測試此解析程式，我們需要使用查詢類型提出請求：

```
query helloWorldTest {  
  helloWorld  
}
```

這是名為 `helloWorldTest` 的查詢，會傳回 `helloWorld` 欄位。執行時，`helloWorld` 欄位解析程式也會執行並傳回回應：

```
{
  "data": {
    "helloWorld": "Hello World"
  }
}
```

傳回像這樣的常數是最簡單的操作。實際上，您將傳回輸入、清單等。以下是更複雜的範例：

```
type Book {
  id: ID!
  title: String
}

type Query {
  getBooks: [Book]
}
```

在這裡，我們會傳回 `Books` 的清單。假設我們使用 `DynamoDB` 資料表來存放書籍資料。我們的處理常式可能如下所示：

```
/**
 * Performs a scan on the dynamodb data source
 */
export function request(ctx) {
  return { operation: 'Scan' };
}

/**
 * return a list of scanned post items
 */
export function response(ctx) {
  return ctx.result.items;
}
```

我們的請求使用內建掃描操作來搜尋資料表中的所有項目、將調查結果存放在內容中，然後將其傳遞給回應。回應接受了結果項目，並在回應中傳回它們：

```
{
  "data": {
```

```
"getBooks": {
  "items": [
    {
      "id": "abcdefgh-1234-1234-1234-abcdefghijkl",
      "title": "book1"
    },
    {
      "id": "aaaaaaa-bbbb-cccc-dddd-eeeeeeeeeeee",
      "title": "book2"
    },
    ...
  ]
}
```

解析程式內容

在解析程式中，處理常式鏈中的每個步驟都必須知道先前步驟的資料狀態。一個處理常式的結果可以儲存並傳遞給另一個處理常式做為引數。GraphQL 定義四個基本解析程式引數：

Resolver 基本引數	描述
obj、root、parent 等等	父系的結果。
args	提供給 GraphQL 查詢中 欄位的引數。
context	提供給每個解析程式並保留重要內容資訊的值，例如目前登入的使用者或資料庫的存取權。
info	值，其中包含與目前查詢相關的欄位特定資訊，以及結構描述詳細資訊。

在 中 AWS AppSync，[context](#)(ctx) 引數可以保留上述所有資料。它是每個請求建立的物件，包含授權登入資料、結果資料、錯誤、請求中繼資料等資料。內容是程式設計人員操作來自請求其他部分的資料的簡單方法。再次使用此程式碼片段：

```
/**
```

```
* Performs a scan on the dynamodb data source
*/
export function request(ctx) {
  return { operation: 'Scan' };
}

/**
 * return a list of scanned post items
 */
export function response(ctx) {
  return ctx.result.items;
}
```

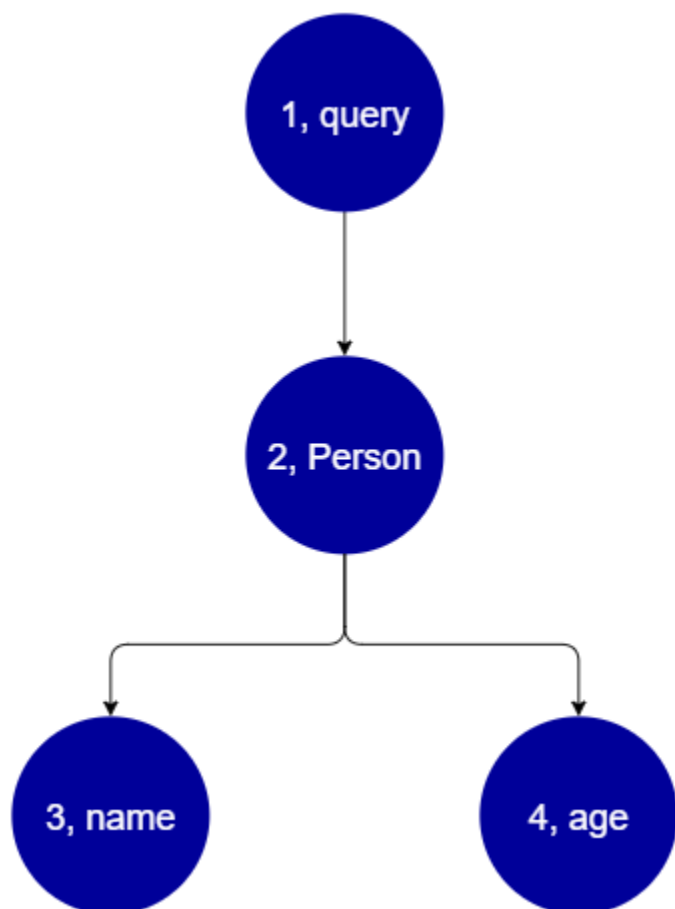
請求會指定內容 (ctx) 做為引數；這是請求的狀態。它會執行資料表中所有項目的掃描，然後將結果存放在 的內容中result。然後，內容會傳遞至回應引數，該引數會存取 result並傳回其內容。

請求和剖析

當您對 GraphQL 服務進行查詢時，必須先執行剖析和驗證程序，才能執行。您的請求將會剖析並翻譯為抽象語法樹。樹狀目錄的內容是透過針對結構描述執行多個驗證演算法來驗證。驗證步驟之後，會周遊和處理樹狀結構的節點。呼叫解析程式，結果會儲存在內容中，並傳回回應。例如，採取以下查詢：

```
query {
  Person { //object type
    name //scalar
    age  //scalar
  }
}
```

我們將Person傳回 name和 age 欄位。執行此查詢時，樹狀結構看起來會像這樣：



從樹狀目錄中，此請求似乎會在結構描述Query中搜尋的根目錄。在查詢內，Person 欄位將會解析。從先前的範例中，我們知道這可能是來自使用者的輸入、值清單等。Person 很可能繫結至保存所需欄位的物件類型 (name 和 age)。找到這兩個子欄位後，它們會依指定的順序 (name 後面接著 age) 解析。樹狀結構完全解決後，請求即完成，並將傳回用戶端。

GraphQL 的其他屬性

GraphQL 由數個設計原則組成，可大規模維持簡易性和穩健性。

宣告式

GraphQL 是宣告式，這表示使用者只會宣告要查詢的欄位，來描述（塑造）資料。回應只會傳回這些屬性的資料。例如，以下操作會擷取 DynamoDB 資料表中 ISBN 13 id值為 **9780199536061** 的Book物件：

```
{
  getBook(id: "9780199536061") {
    name
  }
}
```

```
    year
    author
  }
}
```

回應將傳回承載 (name、year 和 author) 中的欄位，而不會傳回其他欄位：

```
{
  "data": {
    "getBook": {
      "name": "Anna Karenina",
      "year": "1878",
      "author": "Leo Tolstoy",
    }
  }
}
```

由於此設計原則，GraphQL 消除了 REST APIs 在複雜系統中處理過度和擷取不足的常年問題。這可更有效率地收集資料並改善網路效能。

階層

GraphQL 具有彈性，因為請求的資料可由使用者塑造以符合應用程式的需求。請求的資料一律遵循 GraphQL API 中定義的屬性類型和語法。例如，以下程式碼片段顯示名為 `getBook` 操作 `quotes`，其會傳回所有與 `9780199536061` Book 連結的已儲存引號字串和頁面：

```
{
  getBook(id: "9780199536061") {
    name
    year
    author
    quotes {
      description
      page
    }
  }
}
```

執行此查詢會傳回下列結果：

```
{
```

```
"data": {
  "getBook": {
    "name": "Anna Karenina",
    "year": "1878",
    "author": "Leo Tolstoy",
    "quotes": [
      {
        "description": "The highest Petersburg society is essentially one: in it everyone knows everyone else, everyone even visits everyone else.",
        "page": 135
      },
      {
        "description": "Happy families are all alike; every unhappy family is unhappy in its own way.",
        "page": 1
      },
      {
        "description": "To Konstantin, the peasant was simply the chief partner in their common labor.",
        "page": 251
      }
    ]
  }
}
```

如您所見，連結至所請求書籍quotes的欄位會以陣列的形式傳回，其格式與查詢中所述的格式相同。雖然此處未顯示，但 GraphQL 具有額外優勢，其並未特別了解所擷取資料的位置。Books和quotes可以分開存放，但只要關聯存在，GraphQL 仍會擷取資訊。這表示您的查詢可以在單一請求中擷取多個獨立資料。

隱含性

GraphQL 是自我記錄或自我檢查。它支援數個內建操作，可讓使用者檢視結構描述中的基礎類型和欄位。例如，以下Foo類型具有 date和 description 欄位：

```
type Foo {
  date: String
  description: String
}
```

我們可以使用 `_type`操作來尋找結構描述下方的輸入中繼資料：

```
{
  __type(name: "Foo") {
    name                # returns the name of the type
    fields {           # returns all fields in the type
      name              # returns the name of each field
      type {            # returns all types for each field
        name            # returns the scalar type
      }
    }
  }
}
```

這將傳回回應：

```
{
  "__type": {
    "name": "Foo",          # The type name
    "fields": [
      {
        "name": "date",      # The date field
        "type": { "name": "String" } # The date's type
      },
      {
        "name": "description", # The description field
        "type": { "name": "String" } # The description's type
      },
    ]
  }
}
```

此功能可用來了解特定 GraphQL 結構描述支援的類型和欄位。GraphQL 支援各種這些自我檢查操作。如需詳細資訊，請參閱[自我檢查](#)。

強式輸入

GraphQL 支援透過其類型和欄位系統進行強式輸入。當您在結構描述中定義某個項目時，它必須具有可在執行時間之前驗證的類型。它還必須遵循 GraphQL 的語法規格。此概念與其他語言的程式設計並無不同。例如，以下是先前的 Foo 類型：

```
type Foo {
  date: String
```

```
description: String
}
```

我們可以看到 Foo 是將要建立的物件。在執行個體內 Foo，將會有一個 String date 和 description 欄位，兩者都是基本類型（純量）。總而言之，我們看到 Foo 已宣告，且其欄位存在於其範圍內。這種類型檢查和邏輯語法的組合可確保您的 GraphQL API 簡潔且自我可見。您可以在[此處](#)找到 GraphQL 的輸入和語法規格。

入門：在 AWS AppSync 中建立您的第一個 GraphQL API

您可以使用 AWS AppSync 主控台來設定和啟動 GraphQL API。GraphQL APIs 通常需要三個元件：

1. **GraphQL 結構描述** - 您的 GraphQL 結構描述是 API 的藍圖。它會定義您在執行操作時可請求的類型和欄位。若要將資料填入結構描述，您必須將資料來源連線至 GraphQL API。在本快速入門指南中，我們將使用預先定義的模型建立結構描述。
2. **資料來源** - 這些資源包含用於填入 GraphQL API 的資料。這可以是 DynamoDB 資料表、Lambda 函數等。AWS AppSync 支援多種資料來源，以建置強大且可擴展的 GraphQL APIs。資料來源會連結至結構描述中的欄位。每當在欄位上執行請求時，來源的資料就會填入欄位。此機制由解析程式控制。在本快速入門指南中，我們將使用預先定義的模型以及結構描述來建立資料來源。
3. **解析程式** - 解析程式負責將結構描述欄位連結至資料來源。它們會從來源擷取資料，然後根據欄位定義的內容傳回結果。AWS AppSync 支援 JavaScript 和 VTL，以寫入 GraphQL APIs 的解析程式。在本快速入門指南中，將根據結構描述和資料來源自動產生解析程式。在本節中，我們不會深入探討這一點。

AWS AppSync 支援建立和設定所有 GraphQL 元件。開啟主控台時，您可以使用下列方法來建立 API：

1. 透過預先定義的模型產生自訂的 GraphQL API，並設定新的 DynamoDB 資料表（資料來源）以支援它。
2. 設計具有空白結構描述且沒有資料來源或解析程式的 GraphQL API。
3. 使用 DynamoDB 資料表匯入資料並產生結構描述的類型和欄位。
4. 使用 AWS AppSync 的 WebSocket 功能和 Pub/Sub 架構來開發即時 APIs。
5. 使用現有的 GraphQL APIs（來源 APIs）連結至合併的 API。

Note

建議您先檢閱[設計結構描述區段](#)，再使用更進階的工具。這些指南將說明更簡單的範例，您可以在概念上使用這些範例，在 AWS AppSync 中建置更複雜的應用程式。

AWS AppSync 也支援數個非主控台選項來建立 GraphQL APIs。其中包含：

1. AWS Amplify

2. AWS SAM
3. AWS CloudFormation
4. CDK

下列範例將示範如何使用預先定義的模型和 DynamoDB 建立 GraphQL API 的基本元件。

主題

- [在 AWS AppSync 主控台中啟動結構描述](#)
- [導覽 AWS AppSync 主控台](#)
- [使用 GraphQL 變動將資料新增至 AWS AppSync 主控台內的 DynamoDB 資料表](#)
- [使用 GraphQL 查詢從 AWS AppSync 主控台內的 DynamoDB 資料表擷取資料](#)
- [AWS 適用於 AppSync 主控台的補充區段](#)

在 AWS AppSync 主控台中啟動結構描述

在此範例中，您將建立 Todo API，允許使用者建立每日家務提醒Todo的項目，例如####或####。此 API 將示範如何使用 GraphQL 操作，其中狀態會保留在 DynamoDB 資料表中。

實際上，建立第一個 GraphQL API 有三個主要步驟。您必須定義結構描述（類型和欄位）、將資料來源連接至欄位（然後），然後寫入處理商業邏輯的解析程式。不過，主控台體驗會變更此順序。首先，我們將定義資料來源與結構描述的互動方式，稍後再定義結構描述和解析程式。

建立 GraphQL API

1. 登入 AWS Management Console 並開啟 [AppSync 主控台](#)。
2. 在儀表板上，選擇 Create API (建立 API)。
3. 選取 GraphQL APIs時，選擇從頭開始設計。然後選擇下一步。
4. 針對 API 名稱，將預先填入的名稱變更為 **Todo API**，然後選擇下一步。

Note

這裡也有其他選項，但我們不會在此範例中使用這些選項。

5. 在指定 GraphQL 資源區段中，執行下列動作：
 - a. 立即選擇 DynamoDB 資料表支援的建立類型。

 Note

這表示我們將建立新的 DynamoDB 資料表，以做為資料來源連接。

- b. 在模型名稱欄位中，輸入 **Todo**。

 Note

我們的第一個要求是定義我們的結構描述。此模型名稱將是類型名稱，因此您實際執行的就是建立Todo將存在於結構描述中的 type：

```
type Todo {}
```

- c. 在欄位下，執行下列動作：

- i. 建立名為的欄位**id**，類型為 ID，必要設定為 Yes。

 Note

這些欄位將存在於您Todo類型的範圍內。系統會呼叫您這裡的欄位名稱id，類型為 ID！：

```
type Todo {  
  id: ID!  
}
```

AWS AppSync 支援不同使用案例的多個純量值。

- ii. 使用新增欄位，建立四個額外的欄位，並將Name值設定為 **name**、**where**、**when**和 **description**。其Type值為 String，而 Array和 Required值都會設定為 No。其會如下所示：

Model information

Model name

A model is a type with preconfigured queries, mutations, and subscriptions.

The model name must have 1 to 50 characters. Valid characters: A-Z, a-z, 0-9, and _

Fields

Models have fields. Fields have a name and a type.

Name	Type	Array	Required	
id	ID	No	Yes	Remove
name	String	No	No	Remove
when	String	No	No	Remove
where	String	No	No	Remove
description	String	No	No	Remove
Add new field				

Note

完整類型及其欄位如下所示：

```
type Todo {  
  id: ID!  
  name: String  
  when: String  
  where: String  
  description: String  
}
```

由於我們使用這個預先定義的模型建立結構描述，因此它也會根據 `create`、`delete` 和 `update` 等類型填入數個樣板變動，以協助您輕鬆填入資料來源。

- d. 在設定模型資料表下，輸入資料表名稱，例如 **TodoAPITable**。將主索引鍵設定為 `id`。

 Note

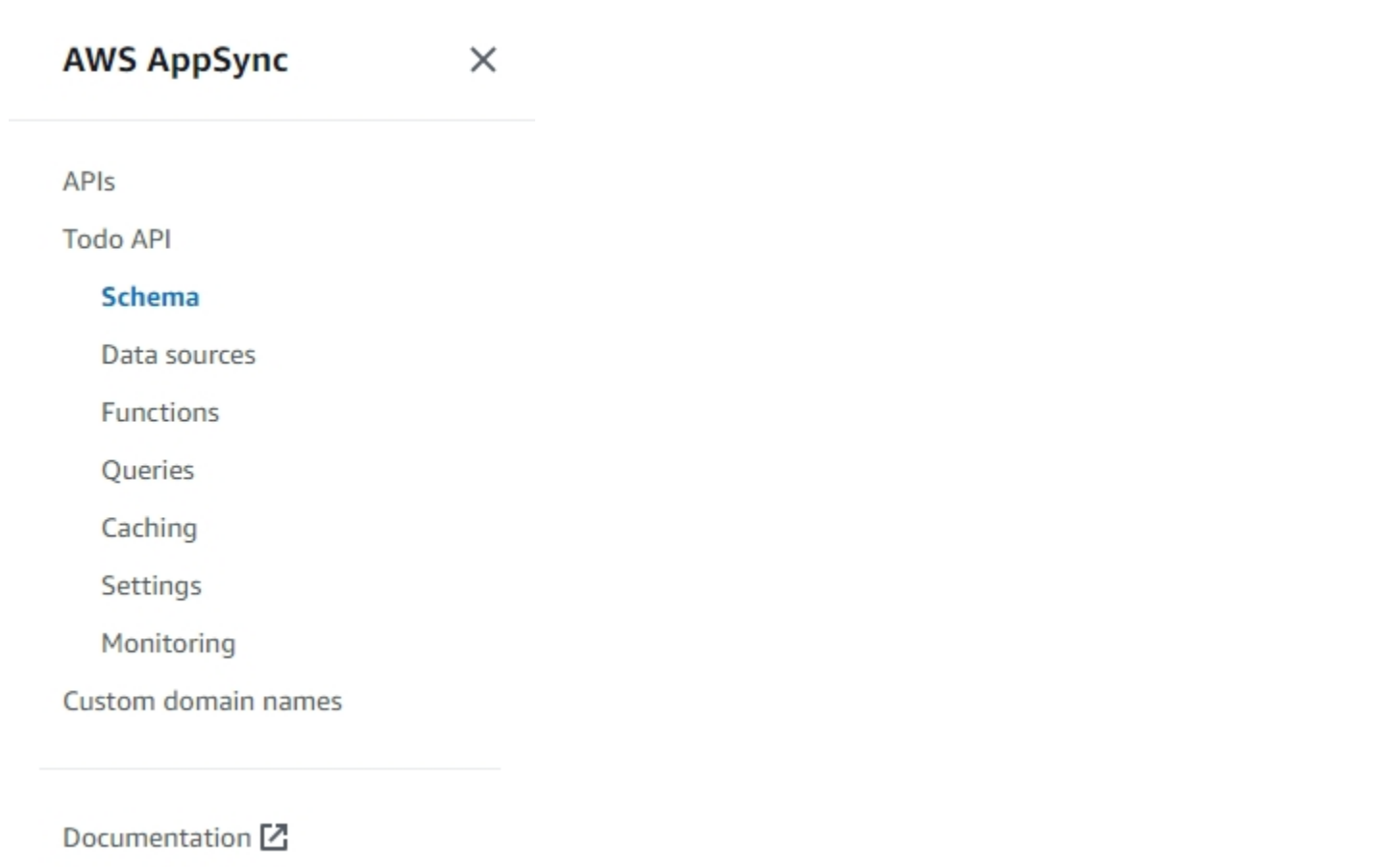
我們基本上正在建立新的 DynamoDB 資料表，稱為 *TodoAPITable*，此資料表將連接至 API 做為我們的主要資料來源。我們的主索引鍵會設定為我們在此之前定義的必要id欄位。請注意，這個新資料表是空白的，除了分割區索引鍵之外，不包含任何內容。

- e. 選擇 Next (下一步)。
6. 檢閱您的變更，然後選擇建立 API。等待片刻，讓 AWS AppSync 服務完成建立您的 API。

您已成功建立具有其結構描述和 DynamoDB 資料來源的 GraphQL API。為了總結上述步驟，我們選擇建立全新的 GraphQL API。我們定義了 API 的名稱，然後新增我們的第一個類型，以新增我們的結構描述定義。我們定義了 類型及其欄位，然後選擇透過建立新的 DynamoDB 資料表來將資料來源連接到其中一個欄位，其中沒有資料。

導覽 AWS AppSync 主控台

在將資料新增至 DynamoDB 資料表之前，我們應該檢閱 AWS AppSync 主控台體驗的基本功能。頁面左側的 AWS AppSync 主控台索引標籤可讓使用者輕鬆導覽至 AWS AppSync 提供的任何主要元件或組態選項：



結構描述設計工具

選擇結構描述以檢視您剛建立的結構描述。如果您檢閱結構描述的內容，您會注意到它已載入大量協助程式操作，以簡化開發程序。在結構描述編輯器中，如果您捲動程式碼，最終將到達您在上一節中定義的模型：

```
type Todo {  
  id: ID!  
  name: String  
  when: String  
  where: String  
  description: String  
}
```

您的模型會成為整個結構描述中使用的基本類型。我們將使用從此類型自動產生的變動，開始將資料新增至資料來源。

以下是關於結構描述編輯器的一些其他秘訣和事實：

1. 程式碼編輯器具有固定和錯誤檢查功能，您可以在撰寫自己的應用程式時使用。
2. 主控台右側顯示則會顯示已建立的 GraphQL 類型，以及不同最上層類型 (例如查詢) 的解析程式。
3. 將新類型新增至結構描述時 (例如 `type User { ... }`)，您可以擁有 AWS AppSync 佈建 DynamoDB 資源。這些包括最符合您 GraphQL 資料存取模式的適當主索引鍵、排序索引鍵和索引設計。如果您選擇最上方的 Create Resources (建立資源)，並從選單選取其中一個使用者定義的類型，您便可以在結構描述設計中選擇不同的欄位選項。我們將在[設計結構描述](#)區段中說明這一點。

解析程式組態

在結構描述設計工具中，解析程式區段包含結構描述中的所有類型和欄位。如果您捲動欄位清單，您會注意到您可以選擇連接，將解析程式連接到特定欄位。這會開啟程式碼編輯器，您可以在其中撰寫解析程式程式碼。AWS AppSync 支援 VTL 和 JavaScript 執行期，您可以透過選擇動作，然後選擇更新執行期來變更頁面頂端。在頁面底部，您也可以建立將依序執行數個操作的函數。不過，解析程式是進階主題，我們不會在本節涵蓋該主題。

資料來源

選擇資料來源以檢視您的 DynamoDB 資料表。透過選擇 Resource 選項 (如果可用)，您可以檢視資料來源的組態。在我們的範例中，這會導致 DynamoDB 主控台。您可以從那裡編輯資料。您也可以選擇資料來源，然後選擇編輯，直接編輯部分資料。如果您需要刪除資料來源，您可以選擇資料來源，然後選取刪除。最後，您可以選擇建立資料來源，然後設定名稱和類型，以建立新的資料來源。請注意，此選項用於將 AWS AppSync 服務連結至現有資源。您仍然需要使用相關服務在帳戶中建立資源，AWS AppSync 才會辨識該資源。

查詢

選擇查詢以檢視您的查詢和變動。當我們使用模型建立 GraphQL API 時，AWS AppSync 會自動產生一些協助程式變動和查詢，以供測試之用。在查詢編輯器中，左側包含 Explorer。這是顯示所有變動和查詢的清單。您可以按一下操作和欄位的名稱值，輕鬆啟用您想要在這裡使用的操作和欄位。這會導致程式碼自動出現在編輯器的中心部分。在這裡，您可以透過修改值來編輯變動和查詢。在編輯器底部，您有查詢變數編輯器，可讓您輸入操作輸入變數的欄位值。在編輯器頂端選擇執行將出現下拉式清單，以選取要執行的查詢/變動。此執行的輸出會出現在頁面右側。回到頂端的 Explorer 區段，您可以選擇操作 (查詢、變動、訂閱)，然後選擇 + 符號來新增該特定操作的新執行個體。在頁面頂端，將會有另一個下拉式清單，其中包含查詢執行的授權模式。不過，我們不會涵蓋本節中的該功能 (如需詳細資訊，請參閱[安全性](#))。

設定

選擇設定以檢視 GraphQL API 的一些組態選項。在這裡，您可以啟用一些選項，例如記錄、追蹤和 Web 應用程式防火牆功能。您也可以新增新的授權模式，以保護資料免於意外洩漏給大眾。不過，這些選項更為進階，不會在本節中涵蓋。

Note

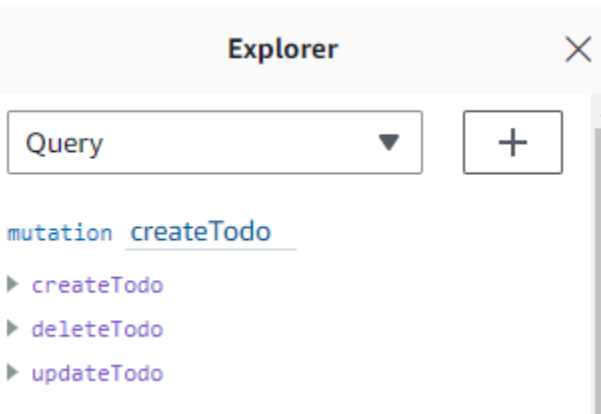
預設授權模式 API_KEY 使用 API 金鑰來測試應用程式。這是提供給所有新建立 GraphQL APIs 的基本授權。建議您使用不同的生產方法。為了本節中的範例，我們只會使用 API 金鑰。如需支援的授權方法的詳細資訊，請參閱 [安全性](#)。

使用 GraphQL 變動將資料新增至 AWS AppSync 主控台中的 DynamoDB 資料表

您的下一個步驟是使用 GraphQL 變動將資料新增至空白 DynamoDB 資料表。變動是 GraphQL 中的基本操作類型之一。它們在結構描述中定義，可讓您操作資料來源中的資料。就 REST APIs 而言，這些與 PUT 或 等操作非常類似 POST。

將資料新增至資料來源

1. 如果您尚未這麼做，請登入 AWS Management Console 並開啟 [AppSync 主控台](#)。
2. 從資料表中選擇您的 API。
3. 在左側的索引標籤中，選擇查詢。
4. 在資料表左側的 Explorer 索引標籤中，您可能會看到查詢編輯器中已定義的數個變動和查詢：



Note

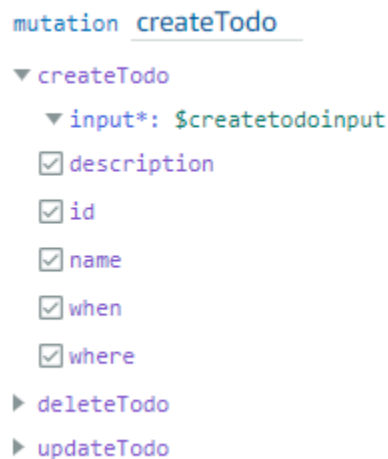
此變動實際上位於您的結構描述中做為 Mutation 類型。它具有 程式碼：

```
type Mutation {
  createTodo(input: CreateTodoInput!): Todo
  updateTodo(input: UpdateTodoInput!): Todo
  deleteTodo(input: DeleteTodoInput!): Todo
}
```

如您所見，此處的操作與查詢編輯器中的操作類似。

AWS AppSync 會自動從我們先前定義的模型產生這些項目。此範例將使用 createTodo 變動將項目新增至我們的 *TodoAPITable* 資料表。

5. 在createTodo變動下展開操作，以選擇createTodo操作：



```
mutation createTodo
  ▼ createTodo
    ▼ input*: $createtodoinput
      ☒ description
      ☒ id
      ☒ name
      ☒ when
      ☒ where
    ▶ deleteTodo
    ▶ updateTodo
```

啟用上圖等所有欄位的核取方塊。

Note

您在此處看到的屬性是變動的不同可修改元素。input 您可以將 視為 的 參數createTodo。具有核取方塊的各種選項是執行 操作後，將在回應中傳回的欄位。

6. 在畫面中央的程式碼編輯器中，您會注意到 操作出現在createTodo變動下方：

```
mutation createTodo($createtodoinput: CreateTodoInput!) {
  createTodo(input: $createtodoinput) {
    where
    when
    name
    id
    description
  }
}
```

Note

為了正確解釋此程式碼片段，我們也必須查看結構描述程式碼。宣告mutation `createTodo($createtodoinput: CreateTodoInput!){}`是具有其其中一個操作的變動，`createTodo`。完整變動位於結構描述中：

```
type Mutation {
  createTodo(input: CreateTodoInput!): Todo
  updateTodo(input: UpdateTodoInput!): Todo
  deleteTodo(input: DeleteTodoInput!): Todo
}
```

從編輯器返回變動宣告，參數是名為 `$createtodoinput` 且必要輸入類型為的物件 `CreateTodoInput`。請注意，`CreateTodoInput`（以及變動中的所有輸入）也會在結構描述中定義。例如，以下是的樣板程式碼 `CreateTodoInput`：

```
input CreateTodoInput {
  name: String
  when: String
  where: String
  description: String
}
```

它包含我們在模型中定義的欄位，即 `name`、`where`、`when`和 `description`。返回編輯器程式碼，在 `createTodo(input: $createtodoinput) {}`，我們會將輸入宣告為 `$createtodoinput`，這也用於變動宣告。我們這樣做是因為這允許 GraphQL 根據提供的類型驗證我們的輸入，並確保它們與正確的輸入一起使用。編輯器程式碼的最後部分顯示執行操作後，將在回應中傳回的欄位：

```
{
  where
  when
  name
  id
  description
}
```

在此編輯器下方的查詢變數索引標籤中，會有一般createtodoinput物件，其中可能有下列資料：

```
{
  "createtodoinput": {
    "name": "Hello, world!",
    "when": "Hello, world!",
    "where": "Hello, world!",
    "description": "Hello, world!"
  }
}
```

Note

這是我們為先前提到的輸入配置值的位置：

```
input CreateTodoInput {
  name: String
  when: String
  where: String
  description: String
}
```

透過新增要在 DynamoDB 資料表中放置的資訊createtodoinput來變更。在這種情況下，我們希望建立一些Todo項目作為提醒：

```
{
  "createtodoinput": {
```



```

    "name": "Shopping List",
    "when": "Friday",
    "where": "Home",
    "description": "I need to buy eggs"
  }
}

```

7. 選擇編輯器頂端的執行。在下拉式清單中選擇 `createTodo`。在編輯器的右側，您應該會看到回應。這可能看起來如下：

```

{
  "data": {
    "createTodo": {
      "where": "Home",
      "when": "Friday",
      "name": "Shopping List",
      "id": "abcdefgh-1234-1234-abcdefghijkl",
      "description": "I need to buy eggs"
    }
  }
}

```

如果您導覽至 DynamoDB 服務，您現在會在資料來源中看到包含此資訊的項目：

TodoAPITable

► **Scan or query items**

Expand to query or scan items.

✔ Completed. Read capacity units consumed: 2

Items returned (1)

☐	id	description	name	when	where
☐		I need to buy ...	Shopping List	Friday	Home

為了摘要說明操作，GraphQL 引擎會剖析記錄，而解析程式會將記錄插入您的 Amazon DynamoDB 資料表。同樣地，您可以在 DynamoDB 主控台中驗證這一點。請注意，您不需要傳入 `id` 值。在結果中 `id` 產生並傳回。這是因為範例在 GraphQL 解析程式中使用 `autoId()` 函數，用於 DynamoDB 資源上設定的分割區索引鍵。我們將介紹如何在不同的區段中建置解析程式。請記下傳回 `id` 的值；您將在下一節中使用它來擷取具有 GraphQL 查詢的資料。

使用 GraphQL 查詢從 AWS AppSync 主控台中的 DynamoDB 資料表擷取資料

現在您的資料庫中存在記錄，您會在執行查詢時取得結果。查詢是 GraphQL 的其他基本操作之一。它用於剖析資料來源的資訊，並從中擷取資訊。在 REST APIs 方面，這與 GET 操作類似。GraphQL 查詢的主要優點是能夠指定應用程式的確切資料需求，以便您適時擷取相關資料。

查詢資料來源

1. 如果您尚未這麼做，請登入 AWS Management Console 並開啟 [AppSync 主控台](#)。
2. 從資料表中選擇您的 API。
3. 在左側的索引標籤中，選擇查詢。
4. 在資料表左側的 Explorer 索引標籤中，在 `query` 下 `listTodos` 展開 `getTodo` 操作：

```
query listTodos
  ▼ getTodo
    ☒ id*
    ☒ description
    ☒ id
    ☒ name
    ☒ when
    ☒ where
  ► listTodos
```

5. 在程式碼編輯器中，您應該會看到操作程式碼：

```
query listTodos {
  getTodo(id: "") {
    description
    id
    name
    when
```

```
    where
  }
```

在 `中(id:"")`，填入您在變動操作結果中儲存的值。在我們的範例中，這將是：

```
query listTodos {
  getTodo(id: "abcdefgh-1234-1234-1234-abcdefghijkl") {
    description
    id
    name
    when
    where
  }
}
```

6. 選擇執行，然後選擇 `listTodos`。結果會出現在編輯器右側。我們的範例如下所示：

```
{
  "data": {
    "getTodo": {
      "description": "I need to buy eggs",
      "id": "abcdefgh-1234-1234-1234-abcdefghijkl",
      "name": "Shopping List",
      "when": "Friday",
      "where": "Home"
    }
  }
}
```

Note

查詢只會傳回您指定的欄位。您可以從傳回欄位刪除不需要的欄位，以取消選取這些欄位：

```
{
  description
  id
  name
  when
  where
}
```

您也可以取消勾選您要刪除之欄位旁的 Explorer 索引標籤中的方塊。

- 您也可以重複這些步驟，在資料來源中建立項目，然後使用 `listTodos` 操作重複查詢步驟，以嘗試 `listTodos` 操作。以下是我們新增第二個任務的範例：

```
{
  "createtodoinput": {
    "name": "Second Task",
    "when": "Monday",
    "where": "Home",
    "description": "I need to mow the lawn"
  }
}
```

透過呼叫 `listTodos` 操作，它會傳回舊項目和新項目：

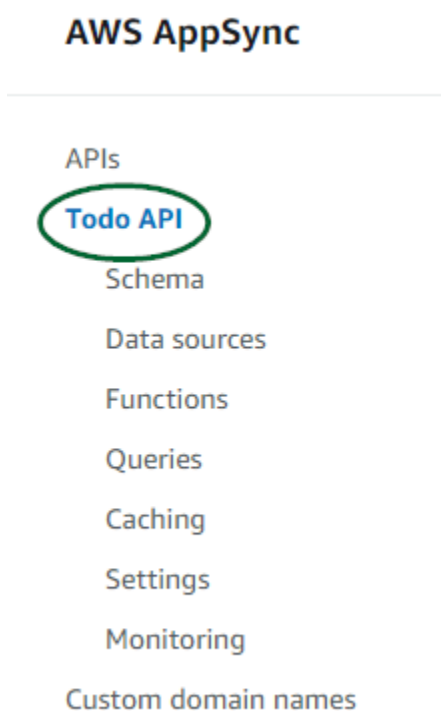
```
{
  "data": {
    "listTodos": {
      "items": [
        {
          "id": "abcdefgh-1234-1234-1234-abcdefghijkl",
          "name": "Shopping List",
          "when": "Friday",
          "where": "Home",
          "description": "I need to buy eggs"
        },
        {
          "id": "aaaaaaaa-bbbb-cccc-dddd-eeeeeeeeeeee",
          "name": "Second Task",
          "when": "Monday",
          "where": "Home",
          "description": "I need to mow the lawn"
        }
      ]
    }
  }
}
```

AWS 適用於 AppSync 主控台的補充區段

這些區段是更進階 AWS AppSync 主題的參考。建議您先遵循補充讀取區段，再執行任何其他動作。

整合

在主控台索引標籤中，如果您選擇 API 的名稱，則會顯示整合頁面：



它總結了設定 API 的步驟，並概述了建置用戶端應用程式的後續步驟。整合您的應用程式區段提供使用 [AWS Amplify 工具鏈](#) 的詳細資訊，透過組態和程式碼產生，自動化將 API 與 iOS、Android 和 JavaScript 應用程式連線的程序。Amplify 工具鏈提供從本機工作站建置專案的完整支援，包括 GraphQL 佈建和 CI/CD 的工作流程。

用戶端範例區段也會列出用於測試 end-to-end 體驗的範例用戶端應用程式（例如 JavaScript、iOS、Android）。您可以複製和下載這些範例，而且組態檔案具有開始使用所需的必要資訊（例如您的端點 URL）。遵循 [AWS Amplify 工具鏈](#) 頁面上的指示來執行您的應用程式。

補充讀取

- [設計 GraphQL APIs](#) - 這是使用不含資料來源或解析程式的空白結構描述建立 GraphQL 的完整指南。

使用 設計 GraphQL APIs AWS AppSync

AWS AppSync 可讓您使用主控台體驗建立 GraphQL APIs。您在[啟動範例結構描述](#)區段中發現了這一點。不過，該指南並未顯示您可以利用的完整選項和組態目錄 AWS AppSync。

當您選擇在主控台中建立 GraphQL API 時，有幾個選項可供探索。如果您遵循我們的[啟動範例結構描述](#)指南，我們會示範如何從預先定義的模型建立 API。在下列各節中，我們將引導您完成在其中建立 GraphQL APIs 的其餘選項和組態 AWS AppSync。

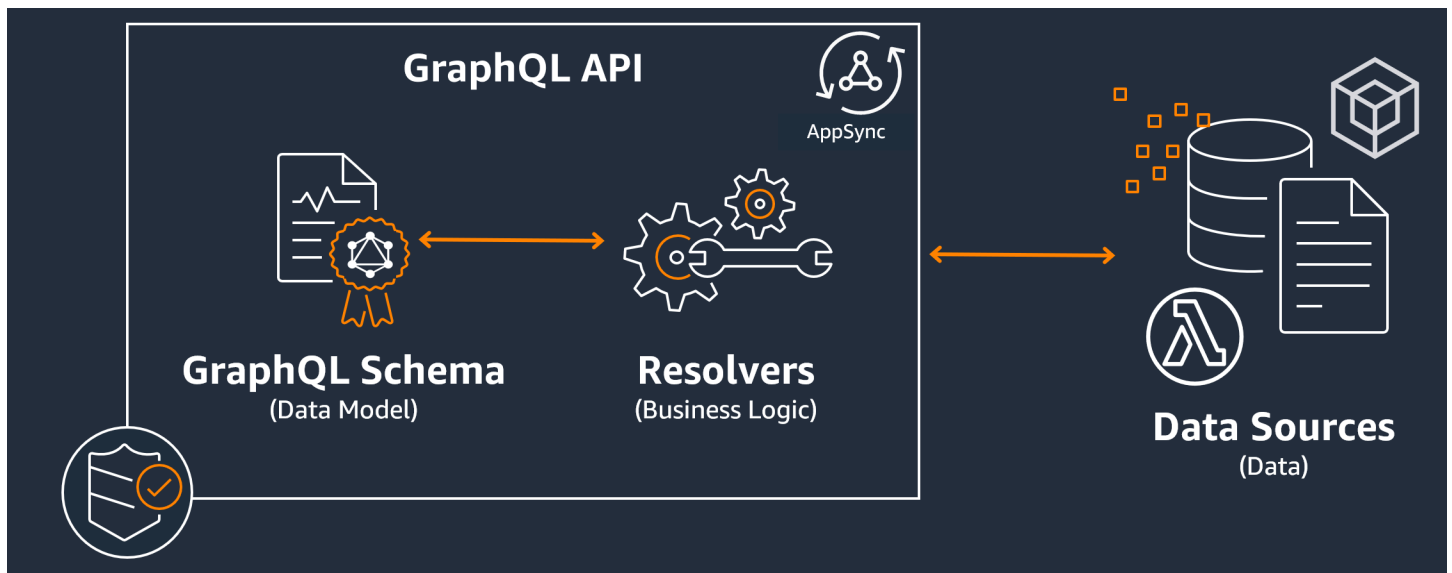
在本節中，您將檢閱下列概念：

1. [Blank APIs or imports](#)：本指南將執行整個建立 GraphQL API 的程序。您將了解如何從沒有模型的空白範本建立 GraphQL、設定結構描述的資料來源，以及將第一個解析程式新增至 欄位。
2. [Real-time data](#)：本指南將向您展示使用 WebSocket 引擎建立 API AWS AppSync 的潛在選項。
3. [Merged APIs](#)：本指南將示範如何透過關聯和合併來自多個現有 GraphQL APIs 的資料來建立新的 GraphQL APIs。
4. [the section called “使用 RDS 自我檢查建置 GraphQL APIs”](#)：本指南將說明如何使用資料 API 整合 Amazon RDS 資料表。

建構 GraphQL API (空白或匯入APIs)

從空白範本建立 GraphQL API 之前，檢閱 GraphQL 的相關概念會有幫助。GraphQL API 有三個基本元件：

1. 結構描述是包含資料形狀和定義的檔案。當用戶端向 GraphQL 服務提出請求時，傳回的資料將遵循結構描述的規格。如需詳細資訊，請參閱[GraphQL 結構描述](#)。
2. 資料來源會連接到您的結構描述。提出請求時，這是擷取和修改資料的位置。如需詳細資訊，請參閱[Data sources](#)。
3. 解析程式位於結構描述和資料來源之間。提出請求時，解析程式會對來源的資料執行操作，然後傳回結果做為回應。如需詳細資訊，請參閱[Resolvers](#)。



AWS AppSync 可讓您建立、編輯和存放結構描述和解析程式的程式碼，以管理您的 APIs。您的資料來源將來自外部儲存庫，例如資料庫、DynamoDB 資料表和 Lambda 函數。如果您使用 AWS 服務來存放資料，或打算這麼做，會在將資料從 AWS 您的帳戶關聯至 GraphQL APIs 時 AWS AppSync 提供近乎無縫的體驗。

在下一節中，您將了解如何使用 AWS AppSync 服務建立每個元件。

主題

- [設計 GraphQL 結構描述](#)
- [連接資料來源 in AWS AppSync](#)
- [在中設定解析程式 AWS AppSync](#)
- [搭配 AWS CDK 使用 AWS AppSync API](#)

設計 GraphQL 結構描述

GraphQL 結構描述是任何 GraphQL 伺服器實作的基礎。每個 GraphQL API 由單一結構描述定義，其中包含描述請求資料如何填入的類型和欄位。流經 API 的資料和執行的操作必須針對結構描述進行驗證。

一般而言，[GraphQL 類型系統](#)說明 GraphQL 伺服器的功能，並用來判斷查詢是否有效。伺服器類型系統通常稱為該伺服器的結構描述，並且可以包含不同的物件類型、純量類型、輸入類型等。GraphQL 同時是宣告式和強式類型，這表示類型將在執行時間妥善定義，並且只會傳回指定的類型。

AWS AppSync 可讓您定義和設定 GraphQL 結構描述。下一節說明如何使用 AWS AppSync 的服務從頭開始建立 GraphQL 結構描述。

建構 GraphQL 結構描述

Tip

建議您檢閱[結構描述](#)區段，然後再繼續。

GraphQL 是實作 API 服務的強大工具。根據 [GraphQL 的網站](#)，GraphQL 如下：

「GraphQL 是 APIs 的查詢語言，也是使用現有資料完成這些查詢的執行時間。GraphQL 提供 API 中資料的完整且易於理解的描述，讓用戶端能夠確切地詢問他們需要什麼，什麼都不做，讓隨著時間發展 APIs 變得更容易，並啟用強大的開發人員工具。」

本節涵蓋 GraphQL 實作的第一部分：結構描述。使用上面的引號，結構描述扮演「提供 API 中資料的完整且可理解描述」的角色。換句話說，GraphQL 結構描述是服務資料、操作及其之間關係的文字表示。結構描述會被視為 GraphQL 服務實作的主要進入點。令人意外的是，這通常是您在專案中所做的第一件事之一。建議您先檢閱[結構描述](#)區段再繼續。

若要引用[結構描述](#)區段，GraphQL 結構描述會以結構描述定義語言 (SDL) 撰寫。SDL 由具有已建立結構的類型和欄位組成：

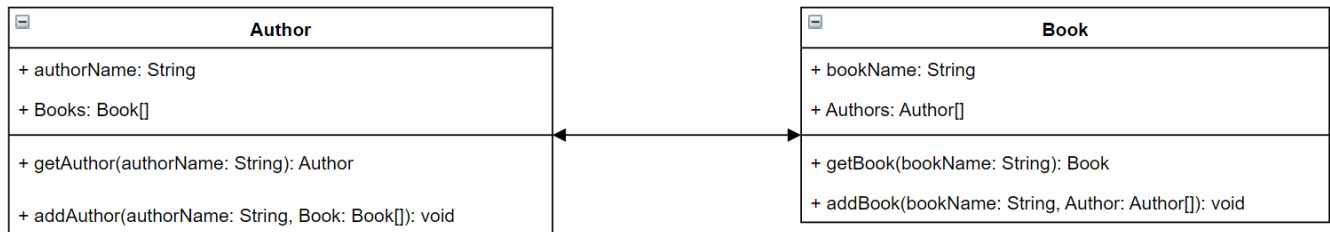
- **類型**：類型是 GraphQL 如何定義資料的形狀和行為。GraphQL 支援許多類型，本節稍後會說明這些類型。結構描述中定義的每個類型都會包含自己的範圍。範圍內的一個或多個欄位，可以包含 GraphQL 服務中使用的值或邏輯。類型會填入許多不同的角色，最常見的是物件或純量（基本值類型）。
- **欄位**：欄位存在於類型的範圍內，並保留從 GraphQL 服務請求的值。這些與其他程式設計語言的變數非常相似。您在欄位中定義的資料形狀將決定資料在請求/回應操作中的結構。這可讓開發人員在不知道服務後端如何實作的情況下，預測將傳回的內容。

最簡單的結構描述將包含三種不同的資料類別：

1. **結構描述根目錄**：根目錄定義結構描述的進入點。它會指向將對資料執行一些操作的欄位，例如新增、刪除或修改某些內容。
2. **類型**：這些是基本類型，用來代表資料的形狀。您幾乎可以將這些視為物件或具有已定義特性之物件的抽象表示法。例如，您可以製作代表資料庫中人員的 Person 物件。每個人的特性將在 中定義為 Person 欄位。它們可以是個人的姓名、年齡、工作、地址等。

3. 特殊物件類型：這些是定義結構描述中操作行為的類型。每個特殊物件類型每個結構描述定義一次。它們會先放置在結構描述根目錄，然後在結構描述內文中定義。特殊物件類型中的每個欄位都會定義您的解析程式要實作的單一操作。

若要將此納入考量，假設您正在建立可存放作者和他們所撰寫書籍的服務。每個作者都有一個名稱和他們撰寫的書籍陣列。每本書都有一個名稱和相關聯的作者清單。我們也希望能夠新增或擷取書籍和作者。此關係的簡單 UML 表示方式如下所示：



在 GraphQL 中，實體 **Author** 和 **Book** 代表結構描述中的兩種不同的物件類型：

```

type Author {
}

type Book {
}
  
```

Author 包含 `authorName` 和 `Books`，而 **Book** 包含 `bookName` 和 `Authors`。這些可以表示為您類型範圍內的欄位：

```

type Author {
  authorName: String
  Books: [Book]
}

type Book {
  bookName: String
  Authors: [Author]
}
  
```

如您所見，類型表示法非常接近圖表。不過，這些方法會變得較棘手。這些將放置在幾個特殊物件類型之一做為欄位。它們的特殊物件分類取決於其行為。GraphQL 包含三種基本的特殊物件類型：查詢、變動和訂閱。如需詳細資訊，請參閱[特殊物件](#)。

由於 `getAuthor` 和 `getBook` 都請求資料，因此它們將放置在 `Query` 特殊物件類型中：

```
type Author {
  authorName: String
  Books: [Book]
}

type Book {
  bookName: String
  Authors: [Author]
}

type Query {
  getAuthor(authorName: String): Author
  getBook(bookName: String): Book
}
```

操作會連結至查詢，而查詢本身會連結至結構描述。新增結構描述根將定義特殊物件類型 (`Query` 在此案例中為) 做為其中一個進入點。這可以使用 `schema` 關鍵字來完成：

```
schema {
  query: Query
}

type Author {
  authorName: String
  Books: [Book]
}

type Book {
  bookName: String
  Authors: [Author]
}

type Query {
  getAuthor(authorName: String): Author
  getBook(bookName: String): Book
}
```

查看最後兩種方法，`addAuthor`和`addBook`並將資料新增至您的資料庫，以便以`Mutation`特殊物件類型定義這些資料。不過，我們也知道不允許直接參考物件的輸入<https://docs.aws.amazon.com/appsync/latest/devguide/graphql-types.html#input-components>，因為它們是嚴格輸出類型。在這種情況下，我們無法使用 `Author` 或 `Book`，因此我們需要使用相同的欄位建立輸入類型。在此範例中，我們新增了 `AuthorInput` 和 `BookInput`，兩者都接受各自類型的相同欄位。然後，我們使用輸入做為參數來建立變動：

```
schema {  
  query: Query  
  mutation: Mutation  
}  
  
type Author {  
  authorName: String  
  Books: [Book]  
}  
  
input AuthorInput {  
  authorName: String  
  Books: [BookInput]  
}  
  
type Book {  
  bookName: String  
  Authors: [Author]  
}  
  
input BookInput {  
  bookName: String  
  Authors: [AuthorInput]  
}  
  
type Query {  
  getAuthor(authorName: String): Author  
  getBook(bookName: String): Book  
}  
  
type Mutation {  
  addAuthor(input: [BookInput]): Author  
  addBook(input: [AuthorInput]): Book  
}
```

讓我們檢閱剛執行的操作：

1. 我們建立了一個結構描述，其中包含 Book 和 Author 類型來代表我們的實體。
2. 我們新增了包含實體特性的欄位。
3. 我們新增了查詢，以從資料庫擷取此資訊。
4. 我們新增了變動來操作資料庫中的資料。
5. 我們新增輸入類型來取代變動中的物件參數，以符合 GraphQL 的規則。
6. 我們已將查詢和變動新增至根結構描述，以便 GraphQL 實作了解根類型位置。

如您所見，建立結構描述的程序通常需要資料建模（特別是資料庫建模）的許多概念。您可以將結構描述視為符合來源資料的形狀。它也可以做為解析程式將實作的模型。在下列各節中，您將了解如何使用各種 AWS 支援的工具和服務建立結構描述。

Note

以下各節中的範例並非要在實際應用程式中執行。它們只用來展示命令，讓您可以建置自己的應用程式。

建立結構描述

您的結構描述將位於名為 `schema.graphql` 的檔案中，AWS AppSync 的檔案中，允許使用者使用各種方法為其 GraphQL APIs 建立新的結構描述。在此範例中，我們將建立空白 API 以及空白結構描述。

Console

1. 登入 AWS Management Console 並開啟 [AppSync 主控台](#)。
 - a. 在儀表板上，選擇 Create API (建立 API)。
 - b. 在 API 選項下，選擇 GraphQL APIs、從頭開始設計，然後選擇下一步。
 - i. 針對 API 名稱，請將預先填入的名稱變更為應用程式需要的名稱。
 - ii. 如需聯絡詳細資訊，您可以輸入聯絡點來識別 API 的管理員。此為選用欄位。
 - iii. 在私有 API 組態下，您可以啟用私有 API 功能。私有 API 只能從設定的 VPC 端點 (VPCE) 存取。如需詳細資訊，請參閱[私有 APIs](#)。

不建議在此範例中啟用此功能。檢閱您的輸入後，請選擇下一步。

- c. 在建立 GraphQL 類型下，您可以選擇建立 DynamoDB 資料表以用作資料來源，或略過此項目，稍後再執行此動作。

在此範例中，稍後選擇建立 GraphQL 資源。我們將在單獨的區段中建立資源。

- d. 檢閱您的輸入，然後選擇建立 API。
2. 您將位於特定 API 的儀表板中。您可以指出，因為 API 的名稱將位於儀表板頂端。如果不是這種情況，您可以在 Sidebar 中選取 APIs，然後在 API 儀表板中選擇您的 APIs。
 - 在 API 名稱下方的側邊欄中，選擇結構描述。
 3. 在結構描述編輯器中，您可以設定 `schema.graphql` 檔案。它可能是空的，或填入從模型產生的類型。在右側，您有解析程式區段，用於將解析程式連接至您的結構描述欄位。我們不會查看本節中的解析程式。

CLI

Note

使用 CLI 時，請確定您擁有存取和建立服務中資源的正確許可。您可能想要為需要存取服務的非管理員使用者設定[最低權限](#)政策。如需 AWS AppSync 政策的詳細資訊，請參閱[Identity and Access Management AWS AppSync](#)。
此外，如果您尚未閱讀主控台版本，建議您先閱讀。

1. 如果您尚未這麼做，請 AWS [安裝](#) CLI，然後新增您的[組態](#)。
2. 執行 `create-graphql-api` 命令來建立 GraphQL API 物件。

您需要為此特定命令輸入兩個參數：

1. API name 的。
2. authentication-type，或用於存取 API 的登入資料類型 (IAM、OIDC 等)。

Note

Region 必須設定 等其他參數，但通常會預設為您的 CLI 組態值。

範例命令可能如下所示：

```
aws appsync create-graphql-api --name testAPI123 --authentication-type API_KEY
```

輸出將在 CLI 中傳回。範例如下：

```
{
  "graphqlApi": {
    "xrayEnabled": false,
    "name": "testAPI123",
    "authenticationType": "API_KEY",
    "tags": {},
    "apiId": "abcdefghijklmnopqrstuvwxyz",
    "uris": {
      "GRAPHQL": "https://zyxwvutsrqponmlkjihgfedcba.appsync-api.us-west-2.amazonaws.com/graphql",
      "REALTIME": "wss://zyxwvutsrqponmlkjihgfedcba.appsync-realtime-api.us-west-2.amazonaws.com/graphql"
    },
    "arn": "arn:aws:appsync:us-west-2:107289374856:apis/abcdefghijklmnopqrstuvwxyz"
  }
}
```

3.

 Note

這是選用的命令，採用現有的結構描述，並使用 base-64 blob 將其上傳至 AWS AppSync 服務。基於此範例，我們不會使用此命令。

執行 [start-schema-creation](#) 命令。

您需要為此特定命令輸入兩個參數：

1. 上一個步驟api-id的。
2. 結構描述definition是 base-64 編碼的二進位 Blob。

範例命令可能如下所示：

```
aws appsync start-schema-creation --api-id abcdefghijklmnopqrstuvwxyz --  
definition "aa1111aa-123b-2bb2-c321-12hgg76cc33v"
```

將傳回輸出：

```
{  
  "status": "PROCESSING"  
}
```

此命令不會在處理後傳回最終輸出。您必須使用單獨的命令 [get-schema-creation-status](#) 來查看結果。請注意，這兩個命令是非同步的，因此即使結構描述仍在建立中，您也可以檢查輸出狀態。

CDK

Tip

在使用 CDK 之前，我們建議您檢閱 CDK 的 [官方文件](#) 以及 CDK AWS AppSync 參考。

https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws_appsync-readme.html

下列步驟只會顯示用來新增特定資源的程式碼片段一般範例。這並非生產程式碼中的可行解決方案。我們也假設您已經有運作中的應用程式。

1. CDK 的起點略有不同。理想情況下，應該已建立您的 `schema.graphql` 檔案。您只需要建立副檔名為 `.graphql` 的新檔案。這可以是空檔案。
2. 一般而言，您可能需要將匯入指令新增至您正在使用的服務。例如，它可能會遵循以下表單：

```
import * as x from 'x'; # import wildcard as the 'x' keyword from 'x-service'  
import {a, b, ...} from 'c'; # import {specific constructs} from 'c-service'
```

若要新增 GraphQL API，您的堆疊檔案需要匯入 AWS AppSync 服務：

```
import * as appsync from 'aws-cdk-lib/aws-appsync';
```

Note

這表示我們會在 `appsync` 關鍵字下匯入整個服務。若要在您的應用程式中使用此項目，您的 AWS AppSync 建構會使用格式 `appsync.construct_name`。例如，如果我們想要製作 GraphQL API，我們會說 `new appsync.GraphqlApi(args_go_here)`。下列步驟說明此項目。

3. 最基本的 GraphQL API 將包含 API name 和 schema 路徑的。

```
const add_api = new appsync.GraphqlApi(this, 'API_ID', {
  name: 'name_of_API_in_console',
  schema: appsync.SchemaFile.fromAsset(path.join(__dirname,
    'schema_name.graphql')),
});
```

Note

讓我們檢閱此程式碼片段的功能。在範圍內 `api`，我們呼叫來建立新的 GraphQL API `appsync.GraphqlApi(scope: Construct, id: string, props: GraphqlApiProps)`。範圍為 `this`，是指目前的物件。ID 是 `API_ID`，其將在建立 AWS CloudFormation 時成為 GraphQL API 的資源名稱。 `GraphqlApiProps` 包含 GraphQL API name 的和 schema。schema 將透過搜尋 `graphql` 檔案的絕對路徑 (`SchemaFile.fromAsset`) (`schema_name.graphql`) 來產生結構描述 (`__dirname`)。在實際案例中，您的結構描述檔案可能位於 CDK 應用程式內。若要使用對 GraphQL API 所做的變更，您必須重新部署應用程式。

將類型新增至結構描述

現在您已新增結構描述，您可以開始同時新增輸入和輸出類型。請注意，此處的類型不應用於真實程式碼；它們只是協助您了解程序的範例。

首先，我們將建立物件類型。在真實程式碼中，您不需要從這些類型開始。只要您遵循 GraphQL 的規則和語法，就可以隨時進行您想要的類型。

 Note

接下來的幾個部分將使用結構描述編輯器，因此請保持開啟狀態。

Console

- 您可以使用 `type` 關鍵字以及 類型的名稱來建立物件類型：

```
type Type_Name_Goes_Here {}
```

在類型範圍內，您可以新增代表物件特性的欄位：

```
type Type_Name_Goes_Here {  
  # Add fields here  
}
```

範例如下：

```
type Obj_Type_1 {  
  id: ID!  
  title: String  
  date: AWSDateTime  
}
```

 Note

在此步驟中，我們新增了一個一般物件類型，其中包含存放為 `id` 的必要欄位 `ID!`、存放為 `title` 的欄位 `String`，以及存放為 `date` 的欄位 `AWSDateTime`。若要查看類型和欄位清單及其用途，請參閱[結構描述](#)。若要查看純量清單及其用途，請參閱[類型參考](#)。

CLI

 Note

如果您尚未閱讀主控台版本，建議您先閱讀。

- 您可以執行 [create-type](#) 命令來建立物件類型。

您需要為此特定命令輸入幾個參數：

1. API api-id 的。
2. definition 或 類型的內容。在主控台範例中，這是：

```
type Obj_Type_1 {  
  id: ID!  
  title: String  
  date: AWSDateTime  
}
```

3. format 您輸入的。在此範例中，我們使用 SDL。

範例命令可能如下所示：

```
aws appsync create-type --api-id abcdefghijklmnopqrstuvwxyz --definition "type  
Obj_Type_1{id: ID! title: String date: AWSDateTime}" --format SDL
```

輸出將在 CLI 中傳回。範例如下：

```
{  
  "type": {  
    "definition": "type Obj_Type_1{id: ID! title: String date:  
AWSDateTime}",  
    "name": "Obj_Type_1",  
    "arn": "arn:aws:appsync:us-west-2:107289374856:apis/  
abcdefghijklmnopqrstuvwxyz/types/Obj_Type_1",  
    "format": "SDL"  
  }  
}
```

Note

在此步驟中，我們新增了一個一般物件類型，其中包含存放為的必要id欄位ID、存放為 title 的欄位String，以及存放為 date 的欄位AWSDateTime。若要查看類型和欄位清單及其用途，請參閱[結構描述](#)。若要查看純量清單及其用途，請參閱[類型參考](#)。

此外，您可能已了解輸入定義直接適用於較小的類型，但無法用於新增較大的或多種類型。您可以選擇在 `.graphql` 檔案中新增所有項目，然後將其 做為輸入 傳遞。

CDK

Tip

在使用 CDK 之前，建議您檢閱 CDK 的 [官方文件](https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws_appsync-readme.html) 以及 CDK AWS AppSync 參考。 https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws_appsync-readme.html

下列步驟只會顯示用來新增特定資源的程式碼片段一般範例。這並非生產程式碼中的可行解決方案。我們也假設您已經有運作中的應用程式。

若要新增類型，您需要將其新增至您的 `.graphql` 檔案。例如，主控台範例為：

```
type Obj_Type_1 {  
  id: ID!  
  title: String  
  date: AWSDateTime  
}
```

您可以將類型直接新增至結構描述，就像任何其他檔案一樣。

Note

若要使用對 GraphQL API 所做的變更，您必須重新部署應用程式。

物件類型 具有 純量類型 的欄位，例如字串和整數。AWS AppSync 也可讓您除了基本 GraphQL 純量 `AWSDateTime` 之外，使用 `enum` 增強純量類型。此外，結尾為驚嘆號的任何欄位都是必要的。

特別是 ID 純量類型是唯一識別符，可以是 `String` 或 `Int`。您可以在解析程式碼中控制這些項目以進行自動指派。

特殊物件類型之間有相似之處，例如 `Query` 和上述範例的「一般」物件類型，因為它們都使用 `type` 關鍵字並被視為物件。不過，對於特殊物件類型 (`Query`、`Subscription` 和 `Mutation`)，其行為非常不同，因為它們公開為 API 的進入點。他們也更重視塑造操作，而不是資料。如需詳細資訊，請參閱 [查詢和變動類型](#)。

在特殊物件類型的主題上，下一個步驟可能是新增一或多個物件，以對成形的資料執行操作。在實際案例中，每個 GraphQL 結構描述至少必須具有請求資料的根查詢類型。您可以將查詢視為 GraphQL 伺服器的其中一個進入點（或端點）。讓我們新增查詢做為範例。

Console

- 若要建立查詢，您只需將其新增至結構描述檔案，就像任何其他類型一樣。查詢需要 Query 類型和根目錄中的項目，如下所示：

```
schema {  
  query: Name_of_Query  
}  
  
type Name_of_Query {  
  # Add field operation here  
}
```

請注意，生產環境中的 *Name_of_Query* Query 會在大多數情況下呼叫。建議您將其保持在此值。在查詢類型中，您可以新增欄位。每個欄位都會在請求中執行操作。因此，如果不是全部，這些欄位大部分都會連接到解析程式。不過，我們並不在意本節中的這一點。關於欄位操作的格式，可能如下所示：

```
Name_of_Query(params): Return_Type # version with params  
Name_of_Query: Return_Type # version without params
```

範例如下：

```
schema {  
  query: Query  
}  
  
type Query {  
  getObj: [Obj_Type_1]  
}  
  
type Obj_Type_1 {  
  id: ID!  
  title: String  
  date: AWSDateTime  
}
```

Note

在此步驟中，我們新增了 類型，並在schema根目錄中定義了該Query類型。我們的Query類型定義了傳回Obj_Type_1物件清單getObj的欄位。請注意，Obj_Type_1是上一個步驟的物件。在生產程式碼中，您的欄位操作通常會使用這類物件所塑造的資料Obj_Type_1。此外，等欄位getObj通常會有解析程式來執行商業邏輯。這將在不同的章節中涵蓋。

另請注意，會在匯出期間 AWS AppSync 自動新增結構描述根目錄，因此在技術上您不需要將其直接新增至結構描述。我們的服務會自動處理重複的結構描述。我們將它新增到這裡，作為最佳實務。

CLI

Note

如果您尚未閱讀主控台版本，建議您先閱讀。

1. 執行 `create-type` 命令，以建立具有 query 定義的 schema 根目錄。

您需要為此特定命令輸入幾個參數：

1. API api-id 的。
2. definition 或 類型的內容。在主控台範例中，這是：

```
schema {  
  query: Query  
}
```

3. 您輸入的 format。在此範例中，我們使用 SDL。

範例命令可能如下所示：

```
aws appsync create-type --api-id abcdefghijklmnopqrstuvwxyz --definition "schema  
{query: Query}" --format SDL
```

輸出將在 CLI 中傳回。範例如下：

```
{
  "type": {
    "definition": "schema {query: Query}",
    "name": "schema",
    "arn": "arn:aws:appsync:us-west-2:107289374856:apis/
    abcdefghijklmnopqrstuvwxyz/types/schema",
    "format": "SDL"
  }
}
```

Note

請注意，如果您未在 `create-type` 命令中正確輸入內容，您可以執行 [update-type](#) 命令來更新結構描述根目錄（或結構描述中的任何類型）。在此範例中，我們將暫時變更結構描述根目錄以包含 `subscription` 定義。

您需要為此特定命令輸入幾個參數：

1. API `api-id` 的。
2. `type-name` 您類型的。在主控台範例中，這是 `schema`。
3. `definition` 或類型的內容。在主控台範例中，這是：

```
schema {
  query: Query
}
```

新增後的結構描述 `subscription` 如下所示：

```
schema {
  query: Query
  subscription: Subscription
}
```

4. 您輸入的 `format`。在此範例中，我們使用 `SDL`。

範例命令可能如下所示：

```
aws appsync update-type --api-id abcdefghijklmnopqrstuvwxyz --type-name
schema --definition "schema {query: Query subscription: Subscription}"
--format SDL
```

輸出將在 CLI 中傳回。範例如下：

```
{
  "type": {
    "definition": "schema {query: Query subscription: Subscription}",
    "arn": "arn:aws:appsync:us-west-2:107289374856:apis/
abcdefghijklmnopqrstuvwxyz/types/schema",
    "format": "SDL"
  }
}
```

在此範例中，新增預先格式化的檔案仍然有效。

2. 執行 `create-type` 命令來建立 Query 類型。

您需要為此特定命令輸入幾個參數：

1. API `api-id` 的。
2. `definition` 或 類型的內容。在主控台範例中，這是：

```
type Query {
  getObj: [Obj_Type_1]
}
```

3. 您輸入的 `format`。在此範例中，我們使用 `SDL`。

範例命令可能如下所示：

```
aws appsync create-type --api-id abcdefghijklmnopqrstuvwxyz --definition "type
Query {getObj: [Obj_Type_1]}" --format SDL
```

輸出將在 CLI 中傳回。範例如下：

```
{
  "type": {
```

```

      "definition": "Query {getObj: [Obj_Type_1]}",
      "name": "Query",
      "arn": "arn:aws:appsync:us-west-2:107289374856:apis/
      abcdefghijklmnopqrstuvwxyz/types/Query",
      "format": "SDL"
    }
  }
}

```

Note

在此步驟中，我們新增了 類型，並在您的 schema 根中定義了該 Query 類型。我們的 Query 類型定義了傳回 Obj_Type_1 物件清單 getObj 的欄位。在 schema 根碼 中 query: Query，query: 部分表示已在結構描述中定義查詢，而 Query 部分表示實際的特殊物件名稱。

CDK

Tip

在使用 CDK 之前，建議您檢閱 CDK 的 [官方文件](https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws_appsync-readme.html) 以及 CDK AWS AppSync 參考。 https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws_appsync-readme.html

下列步驟只會顯示用來新增特定資源的程式碼片段一般範例。這並非生產程式碼中的可行解決方案。我們也假設您已經有運作中的應用程式。

您需要將查詢和結構描述根新增至 .graphql 檔案。我們的範例看起來像以下範例，但您想要將其取代為實際的結構描述程式碼：

```

schema {
  query: Query
}

type Query {
  getObj: [Obj_Type_1]
}

type Obj_Type_1 {
  id: ID!
  title: String
}

```



```
date: AWSDateTime
}
```

您可以將類型直接新增至結構描述，就像任何其他檔案一樣。

Note

更新結構描述根是選用的。我們在此範例中新增了此範例做為最佳實務。
若要使用對 GraphQL API 所做的變更，您必須重新部署應用程式。

您現在已看到建立物件和特殊物件（查詢）的範例。您也已了解這些如何互相連結來描述資料和操作。您可以只具有資料描述和一或多個查詢的結構描述。不過，我們希望新增另一個操作，將資料新增至資料來源。我們會新增另一個名為的特殊物件類型Mutation，以修改資料。

Console

- 變動將稱為 Mutation。如同 Query，中的欄位操作Mutation將描述操作，並將連接到解析程式。此外，請注意，我們需要在schema根目錄中定義它，因為它是一種特殊的物件類型。以下是變動的範例：

```
schema {
  mutation: Name_of_Mutation
}

type Name_of_Mutation {
  # Add field operation here
}
```

典型的變動會列在根目錄中，就像查詢一樣。變動是使用type關鍵字和名稱來定義。*Name_of_Mutation* 通常稱為 Mutation，因此我們建議您以這種方式保留它。每個欄位也會執行操作。關於欄位操作的格式，可能如下所示：

```
Name_of_Mutation(params): Return_Type # version with params
Name_of_Mutation: Return_Type # version without params
```

範例如下：

```
schema {
```

```
query: Query
mutation: Mutation
}

type Obj_Type_1 {
  id: ID!
  title: String
  date: AWSDateTime
}

type Query {
  getObj: [Obj_Type_1]
}

type Mutation {
  addObj(id: ID!, title: String, date: AWSDateTime): Obj_Type_1
}
```

Note

在此步驟中，我們新增了具有 addObj 欄位的Mutation類型。讓我們總結此欄位的功能：

```
addObj(id: ID!, title: String, date: AWSDateTime): Obj_Type_1
```

addObj 正在使用 Obj_Type_1 物件來執行 操作。由於欄位，這很明顯，但語法在：Obj_Type_1傳回類型中證明這一點。在 addObj，它接受Obj_Type_1來自 物件的 title、id和 date 欄位做為參數。如您所見，它看起來與方法宣告非常相似。不過，我們尚未說明方法的行為。如前所述，結構描述只用來定義資料和操作的內容，而不是它們的運作方式。當我們建立第一個解析程式時，實作實際的商業邏輯將稍後出現。

完成結構描述後，您可以選擇將其匯出為 schema.graphql 檔案。在結構描述編輯器中，您可以選擇匯出結構描述，以支援的格式下載檔案。

另請注意，會在匯出期間 AWS AppSync 自動新增結構描述根目錄，因此在技術上您不需要將其直接新增至結構描述。我們的服務會自動處理重複的結構描述。我們將它新增到這裡，作為最佳實務。

CLI

 Note

如果您尚未閱讀主控台版本，建議您先閱讀。

1. 執行 `update-type` 命令來更新您的根結構描述。

您需要為此特定命令輸入幾個參數：

1. API `api-id` 的。
2. `type-name` 您 類型的。在主控台範例中，這是 `schema`。
3. `definition` 或 類型的內容。在主控台範例中，這是：

```
schema {  
  query: Query  
  mutation: Mutation  
}
```

4. 您輸入的 `format`。在此範例中，我們使用 `SDL`。

範例命令可能如下所示：

```
aws appsync update-type --api-id abcdefghijklmnopqrstuvwxyz --type-name schema  
--definition "schema {query: Query mutation: Mutation}" --format SDL
```

輸出將在 CLI 中傳回。範例如下：

```
{  
  "type": {  
    "definition": "schema {query: Query mutation: Mutation}",  
    "arn": "arn:aws:appsync:us-west-2:107289374856:apis/  
abcdefghijklmnopqrstuvwxyz/types/schema",  
    "format": "SDL"  
  }  
}
```

2. 執行 `create-type` 命令來建立 `Mutation` 類型。

您需要為此特定命令輸入幾個參數：

1. API `api-id` 的。
2. `definition` 或 類型的內容。在主控台範例中，這是

```
type Mutation {  
  addObj(id: ID!, title: String, date: AWSDateTime): Obj_Type_1  
}
```

3. `format` 您輸入的。在此範例中，我們使用 `SDL`。

範例命令可能如下所示：

```
aws appsync create-type --api-id abcdefghijklmnopqrstuvwxyz --definition "type  
Mutation {addObj(id: ID! title: String date: AWSDateTime): Obj_Type_1}" --  
format SDL
```

輸出將在 CLI 中傳回。範例如下：

```
{  
  "type": {  
    "definition": "type Mutation {addObj(id: ID! title: String date:  
AWSDateTime): Obj_Type_1}",  
    "name": "Mutation",  
    "arn": "arn:aws:appsync:us-west-2:107289374856:apis/  
abcdefghijklmnopqrstuvwxyz/types/Mutation",  
    "format": "SDL"  
  }  
}
```

CDK

Tip

在使用 CDK 之前，我們建議您檢閱 CDK 的 [官方文件](https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws_appsync-readme.html) 以及 CDK AWS AppSync 參考。
https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws_appsync-readme.html

下列步驟只會顯示用來新增特定資源的程式碼片段一般範例。這並非生產程式碼中的可行解決方案。我們也假設您已經有運作中的應用程式。

您需要將查詢和結構描述根新增至 `.graphql` 檔案。我們的範例看起來像以下範例，但您想要將其取代為實際的結構描述程式碼：

```
schema {  
  query: Query  
  mutation: Mutation  
}  
  
type Obj_Type_1 {  
  id: ID!  
  title: String  
  date: AWSDateTime  
}  
  
type Query {  
  getObj: [Obj_Type_1]  
}  
  
type Mutation {  
  addObj(id: ID!, title: String, date: AWSDateTime): Obj_Type_1  
}
```

Note

更新結構描述根是選用的。我們在此範例中新增了此範例做為最佳實務。若要使用對 GraphQL API 所做的變更，您必須重新部署應用程式。

選用考量 - 使用列舉做為狀態

此時，您知道如何建立基本結構描述。不過，您可以新增許多項目來增加結構描述的功能。應用程式中找到的常見情況是使用列舉做為狀態。您可以使用列舉來強制在呼叫時選擇一組值中的特定值。這對您知道不會長時間大幅改變的實物來說是好事。假設說，我們可以新增列舉，在回應中傳回狀態碼或字串。

舉例來說，假設我們製作的社交媒體應用程式正在後端儲存使用者的貼文資料。我們的結構描述包含代表個別文章資料的Post類型：

```
type Post {  
  id: ID!  
  title: String  
  date: AWSDateTime  
  poststatus: PostStatus  
}
```

我們的 Post 將包含唯一的 id、貼文 title date、貼文，以及一個名為 的列舉PostStatus，代表應用程式處理貼文時的狀態。對於我們的操作，我們將有一個查詢，會傳回所有貼文資料：

```
type Query {  
  getPosts: [Post]  
}
```

我們也會有一個變動，將文章新增至資料來源：

```
type Mutation {  
  addPost(id: ID!, title: String, date: AWSDateTime, poststatus: PostStatus): Post  
}
```

查看我們的結構描述，PostStatus列舉可能有數種狀態。我們可能會希望名為 success（已成功處理貼文）、pending（正在處理貼文）和 error（無法處理貼文）的三種基本狀態。若要新增列舉，我們可以執行下列動作：

```
enum PostStatus {  
  success  
  pending  
  error  
}
```

完整的結構描述可能如下所示：

```
schema {  
  query: Query  
  mutation: Mutation  
}  
  
type Post {
```

```

    id: ID!
    title: String
    date: AWSDateTime
    poststatus: PostStatus
  }

  type Mutation {
    addPost(id: ID!, title: String, date: AWSDateTime, poststatus: PostStatus): Post
  }

  type Query {
    getPosts: [Post]
  }

  enum PostStatus {
    success
    pending
    error
  }

```

如果使用者Post在應用程式中新增，則會呼叫 `addPost` 操作來處理該資料。當連接至 的解析程式 `addPost` 處理資料時，它會持續更新，`poststatus` 並顯示 操作的狀態。查詢時，`Post` 將包含資料的最終狀態。請記住，我們只會描述我們希望資料在結構描述中如何運作。我們正在假設許多有關解析程式的實作（將實作實際的商業邏輯來處理資料以滿足請求）。

選用考量 - 訂閱

In AWS AppSync 訂閱會叫用為變動的回應。您可利用結構描述之中的 `Subscription` 類型及 `@aws_subscribe()` 指令進行設定，表示哪些變動用於叫用一項以上的訂閱。如需設定訂閱的詳細資訊，請參閱[即時資料](#)。

選用考量 - 關係和分頁

假設您在 DynamoDB 資料表中 `Posts` 存放了 100 萬個資料，而且您想要傳回其中一些資料。不過，上述提供的範例查詢只會傳回所有文章。您不會想要在每次提出請求時擷取所有這些項目。反之，您會想要透過它們[進行分頁](#)。請對結構描述進行下列變更：

- 在 `getPosts` 欄位中，新增兩個輸入引數：`nextToken`（疊代運算）和 `limit`（疊代限制）。
- 新增包含 `Posts`（擷取 `Post` 物件清單）和 `nextToken`（疊代運算）欄位的新 `PostIterator` 類型。
- 變更，`getPosts` 使其傳回 `PostIterator`，而不是 `Post` 物件清單。

```
schema {  
  query: Query  
  mutation: Mutation  
}  
  
type Post {  
  id: ID!  
  title: String  
  date: AWSDateTime  
  poststatus: PostStatus  
}  
  
type Mutation {  
  addPost(id: ID!, title: String, date: AWSDateTime, poststatus: PostStatus): Post  
}  
  
type Query {  
  getPosts(limit: Int, nextToken: String): PostIterator  
}  
  
enum PostStatus {  
  success  
  pending  
  error  
}  
  
type PostIterator {  
  posts: [Post]  
  nextToken: String  
}
```

`PostIterator` 類型可讓您傳回 `Post` 物件清單的一部分，以及 `nextToken` 用於取得下一個部分的。在 `PostIterator`，有一個項目清單 `Post([Post])`，以分頁字元 () 傳回 `nextToken`。在 AWS AppSync，這會透過解析程式連線至 Amazon DynamoDB，並自動產生為加密字元。這會將 `limit` 引數的值轉換成 `maxResults` 參數，並將 `nextToken` 引數轉換成 `exclusiveStartKey` 參數。如需 AWS AppSync 主控台內的範例和內建範本範例，請參閱[解析程式參考 \(JavaScript\)](#)。

連接資料來源 in AWS AppSync

資料來源是您 AWS 帳戶中 GraphQL APIs 可與之互動的資源。AWS AppSync 支援多種資料來源 AWS Lambda，例如 Amazon DynamoDB、關聯式資料庫 (Amazon Aurora Serverless)、Amazon OpenSearch Service 和 HTTP 端點。An AWS AppSync API 可以設定為與多個資料來源互動，讓您能

夠在單一位置彙總資料。AWS AppSync 可以使用您帳戶中的現有 AWS 資源，或從結構描述定義代表您佈建 DynamoDB 資料表。

下一節將示範如何將資料來源連接至 GraphQL API。

資料來源的類型

現在您已在 AWS AppSync 主控台中建立結構描述，您可以將資料來源連接至該結構描述。當您最初建立 API 時，可以選擇在建立預先定義的結構描述期間佈建 Amazon DynamoDB 資料表。不過，我們不會在本節中涵蓋該選項。您可以在[啟動結構描述](#)區段中查看此範例。

反之，我們將查看 AWS AppSync 支援的所有資料來源。為應用程式挑選正確的解決方案有許多因素。以下各節將為每個資料來源提供一些額外的內容。如需資料來源的一般資訊，請參閱[資料來源](#)。

Amazon DynamoDB

Amazon DynamoDB 是可擴展應用程式 AWS 的主要儲存解決方案之一。DynamoDB 的核心元件是資料表，只是資料集合。您通常會根據 Book 或等實體建立資料表 Author。資料表項目資訊會儲存為項目，這些項目是每個項目唯一的欄位群組。完整項目代表資料庫中的資料列/記錄。例如，Book 項目的項目可能包含 title 和 author 及其值。title 和等個別欄位 author 稱為屬性，與關聯式資料庫中的資料欄值類似。

您可以猜測，資料表將用於存放應用程式中的資料。AWS AppSync 可讓您將 DynamoDB 資料表連接至 GraphQL API 以操作資料。從前端 Web 和行動部落格取得此[使用案例](#)。此應用程式可讓使用者註冊社交媒體應用程式。使用者可以加入群組，並上傳廣播給其他訂閱該群組的使用者的文章。其應用程式會在 DynamoDB 中存放使用者、文章和使用群組資訊。GraphQL API（由管理 AWS AppSync）會與 DynamoDB 資料表連接。當使用者在系統將反映在前端的系統中進行變更時，GraphQL API 會擷取這些變更，並將其即時廣播給其他使用者。

AWS Lambda

Lambda 是一種事件驅動服務，可自動建置必要的資源，以執程式碼作為事件的回應。Lambda 使用函數，這些函數是包含程式碼、相依性和組態的群組陳述式，用於執行資源。函數會在偵測到觸發條件時自動執行，這是一組叫用函數的活動。觸發可以是發出 API 呼叫的應用程式、您帳戶中啟動資源 AWS 的服務等。觸發時，函數會處理事件，這些事件是包含要修改之資料的 JSON 文件。

Lambda 非常適合執程式碼，無需佈建資源即可執行。從前端 Web 和行動部落格取得此[使用案例](#)。此使用案例與 DynamoDB 區段中展示的使用案例略有相似。在此應用程式中，GraphQL API 負責定義物件的操作，例如新增文章（變動）和擷取該資料（查詢）。為了實作其操作的功能（例如

`getPost ( id: String ! ) : Post`、`getPostsByAuthor ( author: String ! ) : [ Post ]`), 他們使用 Lambda 函數來處理傳入請求。在選項 2：AWS AppSync 搭配 Lambda 解析程式下，他們使用 AWS AppSync 服務來維護其結構描述，並將 Lambda 資料來源連結至其中一個操作。呼叫操作時，Lambda 會與 Amazon RDS Proxy 連接，以在資料庫上執行商業邏輯。

Amazon RDS

Amazon RDS 可讓您快速建置和設定關聯式資料庫。在 Amazon RDS 中，您將建立一般資料庫執行個體，做為雲端中隔離的資料庫環境。在此執行個體中，您將使用 資料庫引擎，這是實際的 RDBMS 軟體 (PostgreSQL、MySQL 等)。此服務透過使用 AWS 基礎設施提供可擴展性、修補和加密等安全服務，以及降低部署的管理成本，來卸載大部分後端工作。

從 Lambda 區段取得相同的[使用案例](#)。在選項 3：AWS AppSync 使用 Amazon RDS 解析程式下，顯示的另一個選項是直接將 中的 GraphQL API 連結至 AWS AppSync Amazon RDS。他們使用[資料 API](#)，將資料庫與 GraphQL API 建立關聯。解析程式會連接到欄位（通常是查詢、變動或訂閱），並實作存取資料庫所需的 SQL 陳述式。當用戶端發出呼叫 欄位的請求時，解析程式會執行陳述式並傳回回應。

Amazon EventBridge

在 EventBridge 中，您將建立事件匯流排，這是從您連接的服務或應用程式 (事件來源) 接收事件，並根據一組規則處理事件的管道。事件是執行環境中的一些狀態變更，而規則則是事件的一組篩選條件。規則遵循事件模式，或事件狀態變更的中繼資料 (id、區域、帳戶號碼、ARN (s) 等)。當事件符合事件模式時，EventBridge 會將整個管道的事件傳送至目的地服務 (目標)，並觸發規則中指定的動作。

EventBridge 非常適合將變更狀態的操作路由到其他一些服務。從前端 Web 和行動部落格取得此[使用案例](#)。此範例描述的電子商務解決方案有數個團隊維護不同的服務。這些服務之一會在前端交付的每個步驟（下單、進行中、運送、交付等）為客戶提供訂單更新。不過，管理此服務的前端團隊無法直接存取排序系統資料，因為這是由個別後端團隊所維護。後端團隊的排序系統也被描述為黑色方塊，因此很難收集有關他們建構資料的方式的資訊。不過，後端團隊確實設定了一個系統，透過 EventBridge 管理的事件匯流排發佈訂單資料。為了存取來自事件匯流排的資料並將其路由到前端，前端團隊建立了指向其 GraphQL API 的新目標 AWS AppSync。他們也建立了規則，只傳送與訂單更新相關的資料。進行更新時，來自事件匯流排的資料會傳送至 GraphQL API。API 中的結構描述會處理資料，然後將其傳遞至前端。

無資料來源

如果您不打算使用資料來源，您可以將其設定為 none。雖然 none 資料來源仍明確分類為資料來源，但 不是儲存媒體。一般而言，解析程式會在某個時間點調用一或多個資料來源來處理請求。不過，在

某些情況下，您可能不需要操作資料來源。將資料來源設定為 `none` 將執行請求、略過資料調用步驟，然後執行回應。

從 EventBridge 區段取得相同的[使用案例](#)。在結構描述中，變動會處理狀態更新，然後將其傳送給訂閱者。回顧解析程式的運作方式，通常至少有一個資料來源調用。不過，此案例中的資料已由事件匯流排自動傳送。這表示不需要變動來執行資料來源調用；訂單狀態可以直接在本機處理。變動設定為 `none`，可做為沒有資料來源調用的傳遞值。然後，結構描述會填入傳送給訂閱者的資料。

OpenSearch

Amazon OpenSearch Service 是一套工具，可實作全文搜尋、資料視覺化和記錄。您可以使用此服務來查詢您上傳的結構化資料。

在此服務中，您將建立 OpenSearch 的執行個體。這些稱為節點。在節點中，您將新增至少一個索引。從概念上來看，索引有點像關聯式資料庫中的資料表。（不過，OpenSearch 不符合 ACID 規範，因此不應以此方式使用）。您將使用上傳到 OpenSearch 服務的資料填入索引。上傳資料時，資料將會以索引中存在的一或多個碎片編製索引。碎片就像索引的分割區，其中包含一些資料，可以與其他碎片分開查詢。上傳後，您的資料將建構為稱為文件的 JSON 檔案。然後，您可以查詢節點以取得文件中的資料。

HTTP 端點

您可以使用 HTTP 端點做為資料來源。AWS AppSync 可以傳送請求到端點，其中包含參數和承載等相關資訊。HTTP 回應將公開給解析程式，該解析程式會在完成其（操作）操作後傳回最終回應。

新增資料來源

如果您建立了資料來源，您可以將其連結至 AWS AppSync 服務，更具體地說是 API。

Console

1. 登入 AWS Management Console 並開啟 [AppSync 主控台](#)。
 - a. 在儀表板中選擇您的 API。
 - b. 在側邊欄中，選擇資料來源。
2. 選擇 Create data source (建立資料來源)。
 - a. 為您的資料來源命名。您也可以提供描述，但這是選用的。
 - b. 選擇您的資料來源類型。
 - c. 對於 DynamoDB，您必須選擇您的區域，然後選擇區域中的資料表。您可以選擇建立新的一般資料表角色或匯入資料表的現有角色，以指定與資料表的互動規則。您可以啟用[版](#)

本控制，當多個用戶端同時嘗試更新資料時，它可以為每個請求自動建立資料版本。版本控制用於保留和維護多個資料變體，用於偵測和解決衝突。您也可以啟用自動產生結構描述，這會取得資料來源，並產生在結構描述中存取它所需的一些 CRUDList、和 Query 操作。

對於 OpenSearch，您必須選擇您的區域，然後選擇區域中的網域（叢集）。您可以選擇建立新的一般資料表角色或匯入資料表的現有角色，以指定與網域的互動規則。

對於 Lambda，您必須選擇您的區域，然後選擇區域中 Lambda 函數的 ARN。您可以選擇建立新的一般資料表角色或匯入資料表的現有角色，以指定與 Lambda 函數的互動規則。

針對 HTTP，您必須輸入 HTTP 端點。

對於 EventBridge，您必須選擇您的區域，然後選擇區域中的事件匯流排。您可以選擇建立新的一般資料表角色或匯入資料表的現有角色，以指定與事件匯流排的互動規則。

對於 RDS，您必須選擇您的區域，然後選擇秘密存放區（使用者名稱和密碼）、資料庫名稱和結構描述。

對於無，您將新增沒有實際資料來源的資料來源。這是用於在本機處理解析程式，而不是透過實際的資料來源。

 Note

如果您要匯入現有的角色，他們需要信任政策。如需詳細資訊，請參閱 [IAM 信任政策](#)。

3. 選擇建立。

 Note

或者，如果您要建立 DynamoDB 資料來源，您可以前往主控台結構描述頁面，選擇頁面頂端的建立資源，然後填寫預先定義的模型以轉換為資料表。在此選項中，您將填寫或匯入基本類型、設定包含分割區索引鍵的基本資料表資料，以及檢閱結構描述變更。

CLI

- 執行 `create-data-source` 命令來建立資料來源。

您需要為此特定命令輸入幾個參數：

1. 您 API `api-id` 的。
2. 資料表 `name` 的。
3. 資料來源 `type` 的。視您選擇的資料來源類型而定，您可能需要輸入 `service-role-arn` 和 `-config` 標籤。

範例命令可能如下所示：

```
aws appsync create-data-source --api-id abcdefghijklmnopqrstuvwxyz
--name data_source_name --type data_source_type --service-role-arn
arn:aws:iam::107289374856:role/role_name --[data_source_type]-config {params}
```

CDK

Tip

在使用 CDK 之前，我們建議您檢閱 CDK 的 [官方文件](https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws_appsync-readme.html) 以及 CDK AWS AppSync 參考。

https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws_appsync-readme.html

下列步驟只會顯示用來新增特定資源之程式碼片段的一般範例。這並非您生產程式碼中的工作解決方案。我們也假設您已經有運作中的應用程式。

若要新增特定資料來源，您需要將 建構新增至堆疊檔案。您可以在這裡找到資料來源類型的清單：

- [DynamoDbDataSource](#)
- [EventBridgeDataSource](#)
- [HttpDataSource](#)
- [LambdaDataSource](#)
- [NoneDataSource](#)
- [OpenSearchDataSource](#)
- [RdsDataSource](#)

1. 一般而言，您可能需要將匯入指令新增至您正在使用的服務。例如，它可能會遵循以下表單：

```
import * as x from 'x'; # import wildcard as the 'x' keyword from 'x-service'
import {a, b, ...} from 'c'; # import {specific constructs} from 'c-service'
```

例如，以下是您可以匯入 AWS AppSync 和 DynamoDB 服務的方式：

```
import * as appsync from 'aws-cdk-lib/aws-appsync';
import * as dynamodb from 'aws-cdk-lib/aws-dynamodb';
```

2. 某些像 RDS 的服務在建立資料來源（例如，VPC 建立、角色和存取憑證）之前，需要在堆疊檔案中進行一些額外的設定。如需詳細資訊，請參閱相關 CDK 頁面中的範例。
3. 對於大多數資料來源，特別是 AWS 服務，您將在堆疊檔案中建立新的資料來源執行個體。一般而言，這看起來會如下所示：

```
const add_data_source_func = new service_scope.resource_name(scope: Construct,
  id: string, props: data_source_props);
```

例如，以下是 Amazon DynamoDB 資料表範例：

```
const add_ddb_table = new dynamodb.Table(this, 'Table_ID', {
  partitionKey: {
    name: 'id',
    type: dynamodb.AttributeType.STRING,
  },
  sortKey: {
    name: 'id',
    type: dynamodb.AttributeType.STRING,
  },
  tableClass: dynamodb.TableClass.STANDARD,
});
```

Note

大多數資料來源至少會有一個必要的 prop (將沒有?符號表示)。請參閱 CDK 文件，了解需要哪些道具。

4. 接著，您需要將資料來源連結至 GraphQL API。建議的方法是在為管道解析程式建立函數時新增它。例如，以下程式碼片段是掃描 DynamoDB 資料表中所有元素的函數：

```
const add_func = new appsync.AppsyncFunction(this, 'func_ID', {
  name: 'func_name_in_console',
  add_api,
  dataSource: add_api.addDynamoDbDataSource('data_source_name_in_console',
  add_ddb_table),
  code: appsync.Code.fromInline(`
    export function request(ctx) {
      return { operation: 'Scan' };
    }

    export function response(ctx) {
      return ctx.result.items;
    }
  `),
  runtime: appsync.FunctionRuntime.JS_1_0_0,
});
```

在dataSource道具中，您可以呼叫 GraphQL API (add_api)，並使用其中一個內建方法 (addDynamoDbDataSource) 建立資料表與 GraphQL API 之間的關聯。引數是將存在於 AWS AppSync 主控台 (data_source_name_in_console在此範例中為) 和資料表方法 () 的此連結名稱add_ddb_table。當您開始建立解析程式時，將在下一節中顯示有關此主題的更多資訊。

有替代方法來連結資料來源。您可以在技術上api新增至資料表函數中的道具清單。例如，以下是步驟 3 的程式碼片段，但具有包含 api GraphQL API 的 props：

```
const add_api = new appsync.GraphqlApi(this, 'API_ID', {
  ...
});

const add_ddb_table = new dynamodb.Table(this, 'Table_ID', {
  ...

  api: add_api
});
```

或者，您可以分別呼叫GraphqlApi建構：

```
const add_api = new appsync.GraphqlApi(this, 'API_ID', {
```



```

    ...
  });

  const add_ddb_table = new dynamodb.Table(this, 'Table_ID', {
    ...
  });

  const link_data_source =
    add_api.addDynamoDbDataSource('data_source_name_in_console', add_ddb_table);

```

我們建議只在函數的道具中建立關聯。否則，您必須在 AWS AppSync 主控台中手動將解析程式函數連結至資料來源（如果您想要繼續使用主控台值 `data_source_name_in_console`），或在函數中以其他名稱建立個別的關聯，例如 `data_source_name_in_console_2`。這是因為道具處理資訊的方式受到限制。

Note

您必須重新部署應用程式，才能查看您的變更。

IAM 信任政策

如果您為資料來源使用現有的 IAM 角色，則需要授予該角色適當的許可，以對 AWS 資源執行操作，例如在 Amazon DynamoDB 資料表 `PutItem` 上執行操作。您也需要修改該角色的信任政策，以允許 AWS AppSync 將其用於資源存取，如下列範例政策所示：

JSON

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "appsync.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}

```


您也可以將條件新增至信任政策，以視需要限制對資料來源的存取。目前，SourceArn 和 SourceAccount 金鑰可用於這些條件。例如，下列政策會將對資料來源的存取限制在帳戶 123456789012：

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "appsync.amazonaws.com"
      },
      "Action": "sts:AssumeRole",
      "Condition": {
        "StringEquals": {
          "aws:SourceAccount": "123456789012"
        }
      }
    }
  ]
}
```

或者，您可以使用下列政策 abcdefghijklmnopq，將資料來源的存取限制為特定 API，例如：

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "appsync.amazonaws.com"
      },
      "Action": "sts:AssumeRole",
      "Condition": {
        "ArnEquals": {
```

```

        "aws:SourceArn": "arn:aws:appsync:us-west-2:123456789012:apis/
        abcdefghijklmnopq"
    }
}
]
}

```

您可以使用下列政策us-east-1，限制從特定區域存取 all AWS AppSync APIs，例如：

JSON

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "appsync.amazonaws.com"
      },
      "Action": "sts:AssumeRole",
      "Condition": {
        "ArnEquals": {
          "aws:SourceArn": "arn:aws:appsync:us-east-1:123456789012:apis/*"
        }
      }
    }
  ]
}

```

在下一節 ([設定解析程式](#)) 中，我們將新增解析程式商業邏輯，並將其連接到結構描述中的欄位，以處理資料來源中的資料。

如需角色政策組態的詳細資訊，請參閱《IAM 使用者指南》中的[修改角色](#)。

如需 AWS Lambda 解析程式 for AWS AppSync 跨帳戶存取的詳細資訊，請參閱[建置跨帳戶 AWS Lambda 解析程式 for AWS AppSync](#)。

在 中設定解析程式 AWS AppSync

在前幾節中，您已了解如何建立 GraphQL 結構描述和資料來源，然後在 AWS AppSync 服務中將這些結構描述和資料來源連結在一起。在您的結構描述中，您可能已在查詢和變動中建立一或多個欄位（操作）。雖然結構描述描述了操作從資料來源請求的資料類型，但它從未實作這些操作在資料方面的行為。

操作的行為一律會在解析程式中實作，這會連結到執行操作的欄位。如需解析程式如何正常運作的詳細資訊，請參閱[解析程式](#)頁面。

在 中 AWS AppSync，您的解析程式會繫結至執行時間，這是解析程式執行的環境。執行時間會指定您的解析程式將寫入的語言。目前支援兩種執行時間：APPSYNC_JS (JavaScript) 和 Apache Velocity 範本語言 (VTL)。

實作解析程式時，會遵循一般結構：

- 在步驟之前：當用戶端提出請求時，所使用結構描述欄位的解析程式（通常是您的查詢、變動、訂閱）會傳遞請求資料。解析程式會開始處理使用步驟前處理常式的請求資料，這允許在資料通過解析程式之前執行一些預先處理操作。
- Function(s)：在步驟執行之前，請求會傳遞至函數清單。清單中的第一個函數將針對資料來源執行。函數是解析程式程式碼的子集，其中包含自己的請求和回應處理常式。請求處理常式會取得請求資料，並對資料來源執行操作。回應處理常式會先處理資料來源的回應，再將其傳回清單。如果有多個函數，請求資料將傳送至清單中要執行的下一個函數。清單中的函數將依開發人員定義的順序序列執行。執行所有函數後，最終結果會傳遞至步驟後。
- 後步驟：後步驟是處理常式函數，可讓您在最終函數的回應上執行一些最終操作，然後再將其傳遞至 GraphQL 回應。

此流程是管道解析程式的範例。兩個執行時間都支援管道解析程式。不過，這是管道解析程式可以執行的操作的簡化說明。此外，我們只描述一個可能的解析程式組態。如需支援的解析程式組態的詳細資訊，請參閱 APPSYNC_JS 的 [JavaScript 解析程式概觀](#)或 VTL 的 [解析程式映射範本概觀](#)。

如您所見，解析程式是模組化的。為了讓解析程式的元件正常運作，它們必須能夠對等其他元件的執行狀態。從[解析程式](#)區段中，您知道解析程式中的每個元件都可以作為一組引數 (args、context等) 傳遞有關執行狀態的重要資訊。在 中 AWS AppSync，這由 嚴格處理context。這是要解析之欄位資訊的容器。這可以包含來自傳遞引數、結果、授權資料、標頭資料等的所有內容。如需內容的詳細資訊，請參閱 APPSYNC_JS 的 [Resolver 內容物件參考](#)或 VTL 的 [Resolver 映射範本內容參考](#)。

內容並非您可用來實作解析程式的唯一工具。AWS AppSync 支援各種公用程式，可用於產生價值、處理錯誤、剖析、轉換等。您可以在[此處](#)查看 APPSYNC_JS 的公用程式清單，或是 [VTL](#) 的公用程式清單。

在下列各節中，您將了解如何在 GraphQL API 中設定解析程式。

主題

- [建立基本查詢 \(JavaScript\)](#)
- [建立基本查詢 \(VTL\)](#)

建立基本查詢 (JavaScript)

GraphQL 解析程式將類型結構描述中的欄位連接到資料來源。解析程式是滿足請求的機制。

Resolvers in AWS AppSync 使用 JavaScript 將 GraphQL 表達式轉換為資料來源可以使用的格式。或者，映射範本可以用 [Apache Velocity 範本語言 \(VTL\)](#) 撰寫，將 GraphQL 表達式轉換為資料來源可以使用的格式。

本節說明如何使用 JavaScript 設定解析程式。[Resolver 教學課程 \(JavaScript\)](#) 區段提供如何使用 JavaScript 實作解析程式的深入教學課程。[Resolver 參考 \(JavaScript\)](#) 區段提供可搭配 JavaScript 解析程式使用的公用程式操作說明。

我們建議您先遵循本指南，再嘗試使用上述任何教學課程。

在本節中，我們將逐步解說如何建立和設定查詢和變動的解析程式。

Note

本指南假設您已建立結構描述，並且至少有一個查詢或變動。如果您要尋找訂閱（即時資料），請參閱[本指南](#)。

在本節中，我們將提供一些設定解析程式的一般步驟，以及使用下列結構描述的範例：

```
// schema.graphql file

input CreatePostInput {
  title: String
  date: AWSDateTime
}
```

```
type Post {
  id: ID!
  title: String
  date: AWSDateTime
}

type Mutation {
  createPost(input: CreatePostInput!): Post
}

type Query {
  getPost: [Post]
}
```

建立基本查詢解析程式

本節將說明如何建立基本查詢解析程式。

Console

1. 登入 AWS Management Console 並開啟 [AppSync 主控台](#)。
 - a. 在 APIs 儀表板中，選擇您的 GraphQL API。
 - b. 在側邊欄中，選擇結構描述。
2. 輸入結構描述和資料來源的詳細資訊。如需詳細資訊，請參閱[設計您的結構描述](#)和[連接資料來源](#)章節。
3. 在結構描述編輯器旁邊，有一個稱為解析程式的視窗。此方塊包含結構描述視窗中定義的類型和欄位清單。您可以將解析程式連接至欄位。您很可能將解析程式連接到您的欄位操作。在本節中，我們將介紹簡單的查詢組態。在查詢類型下，選擇查詢欄位旁的連接。
4. 在連接解析程式頁面的解析程式類型下，您可以選擇管道或單位解析程式。如需這些類型的詳細資訊，請參閱[解析程式](#)。本指南將使用 pipeline resolvers。

Tip

建立管道解析程式時，您的資料來源會連接到管道函數 (s)。函數會在您建立管道解析程式本身後建立，因此此頁面中沒有設定它的選項。如果您使用的是單位解析程式，則資料來源會直接繫結至解析程式，因此您會在此頁面中設定。

針對解析程式執行時間，選擇 APPSYNC_JS 以啟用 JavaScript 執行時間。

5. 您可以為此 API 啟用[快取](#)。我們建議您立即關閉此功能。選擇建立。
6. 在編輯解析程式頁面上，有一個稱為解析程式碼的程式碼編輯器，可讓您實作解析程式處理常式和回應的邏輯（步驟前後）。如需詳細資訊，請參閱[JavaScript 解析程式概觀](#)。

 Note

在我們的範例中，我們只會將請求保留空白，並將回應集設為從[內容](#)傳回最後一個資料來源結果：

```
import {util} from '@aws-appsync/utils';

export function request(ctx) {
  return {};
}

export function response(ctx) {
  return ctx.prev.result;
}
```

在本節下方，有一個名為 Functions 的資料表。函數可讓您實作可在多個解析程式間重複使用的程式碼。您可以視需要將原始程式碼儲存為新增至解析程式的函數，而不是持續重寫或複製程式碼。

函數構成管道操作清單的大量內容。在解析程式中使用多個函數時，您可以設定函數的順序，它們將依序執行。它們會在請求函數執行後和回應函數開始之前執行。

若要新增函數，請在函數下選擇新增函數，然後選擇建立新函數。或者，您可能會看到建立函數按鈕來選擇。

- a. 選擇資料來源。這將是解析程式對其執行動作的資料來源。

 Note

在我們的範例中，我們正在連接的解析程式getPost，它會透過擷取Post物件id。假設我們已為此結構描述設定 DynamoDB 資料表。其分割區索引鍵設定為id，且為空白。

- b. 輸入 Function name。

- c. 在函數程式碼下，您需要實作函數的行為。這可能令人困惑，但每個函數都有自己的本機請求和回應處理常式。請求會執行，然後進行資料來源調用來處理請求，然後由回應處理常式處理資料來源回應。結果會存放在[內容](#)物件中。之後，清單中的下一個函數會執行，如果是最後一個函數，則會傳遞至步驟後回應處理常式。

 Note

在我們的範例中，我們會將解析程式連接至 `getPost`，從資料來源取得 `Post` 物件清單。我們的請求函數會從我們的資料表請求資料，資料表會將其回應傳遞至內容 (ctx)，然後回應會在內容中傳回結果。AWS AppSync 的強度在於其與其他 AWS 服務的互連性。由於我們使用 `DynamoDB`，因此我們有[一套操作](#)來簡化這類操作。我們也有一些其他資料來源類型的樣板範例。

我們的程式碼如下所示：

```
import { util } from '@aws-appsync/utils';

/**
 * Performs a scan on the dynamodb data source
 */
export function request(ctx) {
  return { operation: 'Scan' };
}

/**
 * return a list of scanned post items
 */
export function response(ctx) {
  return ctx.result.items;
}
```

在此步驟中，我們新增了兩個函數：

- `request`：請求處理常式會對資料來源執行擷取操作。引數包含內容物件 (ctx) 或一些資料，可供執行特定操作的所有解析程式使用。例如，它可能包含授權資料、要解析的欄位名稱等。傳回陳述式會執行 [Scan](#) 操作（如需範例，請參閱[此處](#)）。由於我們使用 `DynamoDB`，因此我們可以使用該服務的一些操作。掃描會執行資料表中所有項目的基本擷取。此操作的結果會存放在內容物件中做為 `result` 容器，再傳遞給回應處理常式。在管道中的回應之前 `request` 執行。

- `response`：傳回輸出的回應處理常式`request`。引數是更新的內容物件，而傳回陳述式是`ctx.prev.result`。在本指南的這個階段，您可能不熟悉此值。`ctx`是指內容物件。`prev`是指管道中的先前操作，也就是我們的`request`。`result` 包含解析程式在管道中移動時的結果(s)。如果您將所有操作放在一起，`ctx.prev.result` 會傳回上次執行的操作結果，也就是請求處理常式。

d. 完成後選擇建立。

7. 返回解析程式畫面的函數下，選擇新增函數下拉式清單，並將函數新增至函數清單。
8. 選擇儲存以更新解析程式。

CLI

新增您的 函數

- 使用 [create-function](#) 命令為您的管道解析程式建立 函數。

您需要為此特定命令輸入幾個參數：

1. 您 API `api-id` 的。
2. AWS AppSync 主控台中 `name` 函數的。
3. `data-source-name`、或 函數將使用的資料來源名稱。它必須已在 AWS AppSync 服務中建立並連結至您的 GraphQL API。
4. 函數的 `runtime`、或 環境和語言。對於 JavaScript，名稱必須為 `APPSYNC_JS`，執行期為 `1.0.0`。
5. 函數的 `code`、或 請求和回應處理常式。雖然您可以手動輸入，但將其新增至 `.txt` 檔案（或類似格式），然後將其做為引數傳入會更為容易。

Note

我們的查詢程式碼將位於傳入為引數的檔案中：

```
import { util } from '@aws-appsync/utils';

/**
 * Performs a scan on the dynamodb data source
 */
export function request(ctx) {
```



```
    return { operation: 'Scan' };  
  }  
  
  /**  
   * return a list of scanned post items  
   */  
  export function response(ctx) {  
    return ctx.result.items;  
  }  
}
```

範例命令可能如下所示：

```
aws appsync create-function \  
--api-id abcdefghijklmnopqrstuvwxyz \  
--name get_posts_func_1 \  
--data-source-name table-for-posts \  
--runtime name=APPSYNC_JS,runtimeVersion=1.0.0 \  
--code file://~/path/to/file/{filename}.{fileType}
```

輸出會在 CLI 中傳回。範例如下：

```
{  
  "functionConfiguration": {  
    "functionId": "ejglgvmcabdn7lx75ref4qeig4",  
    "functionArn": "arn:aws:appsync:us-west-2:107289374856:apis/  
abcdefghijklmnopqrstuvwxyz/functions/ejglgvmcabdn7lx75ref4qeig4",  
    "name": "get_posts_func_1",  
    "dataSourceName": "table-for-posts",  
    "maxBatchSize": 0,  
    "runtime": {  
      "name": "APPSYNC_JS",  
      "runtimeVersion": "1.0.0"  
    },  
    "code": "Code output goes here"  
  }  
}
```

 Note

請務必在 `functionId` 某處記錄，因為這將用於將函數連接至解析程式。

建立您的解析程式

- Query 執行 `create-resolver` 命令來建立 的管道函數。

您需要為此特定命令輸入幾個參數：

1. 您 API `api-id` 的。
2. `type-name` 或結構描述中的特殊物件類型（查詢、變動、訂閱）。
3. 在您欲連接解析程式之特殊物件類型內的 `field-name` 或欄位操作。
4. `kind`，指定單位或管道解析程式。將此設定為 `PIPELINE` 以啟用管道函數。
5. 要連接到解析程式的 `pipeline-config`（或）函數。請確定您知道函數 `functionId` 的值。列出的順序很重要。
6. `runtime`，也就是 `APPSYNC_JS`(JavaScript)。 `runtimeVersion` 目前為 `1.0.0`。
7. `code`，其中包含步驟處理常式前後的。

 Note

我們的查詢程式碼將位於傳入為引數的檔案中：

```
import { util } from '@aws-appsync/utils';

/**
 * Sends a request to `put` an item in the DynamoDB data source
 */
export function request(ctx) {
  const { id, ...values } = ctx.args;
  return {
    operation: 'PutItem',
    key: util.dynamodb.toMapValues({ id }),
    attributeValues: util.dynamodb.toMapValues(values),
  };
}
```

```
/**
 * returns the result of the `put` operation
 */
export function response(ctx) {
  return ctx.result;
}
```

範例命令可能如下所示：

```
aws appsync create-resolver \
--api-id abcdefghijklmnopqrstuvwxyz \
--type-name Query \
--field-name getPost \
--kind PIPELINE \
--pipeline-config functions=ejglgvmcabdn7lx75ref4qeig4 \
--runtime name=APPSYNC_JS,runtimeVersion=1.0.0 \
--code file:///path/to/file/{filename}.{fileType}
```

輸出會在 CLI 中傳回。範例如下：

```
{
  "resolver": {
    "typeName": "Mutation",
    "fieldName": "getPost",
    "resolverArn": "arn:aws:appsync:us-west-2:107289374856:apis/
abcdefghijklmnopqrstuvwxyz/types/Mutation/resolvers/getPost",
    "kind": "PIPELINE",
    "pipelineConfig": {
      "functions": [
        "ejglgvmcabdn7lx75ref4qeig4"
      ]
    },
    "maxBatchSize": 0,
    "runtime": {
      "name": "APPSYNC_JS",
      "runtimeVersion": "1.0.0"
    },
    "code": "Code output goes here"
  }
}
```

CDK

 Tip

在使用 CDK 之前，建議您檢閱 CDK 的 [官方文件](https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws_appsync-readme.html) 以及 CDK AWS AppSync 參考。 https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws_appsync-readme.html

下列步驟只會顯示用來新增特定資源之程式碼片段的一般範例。這並非您生產程式碼中的工作解決方案。我們也假設您已有運作中的應用程式。

基本應用程式將需要下列項目：

1. 服務匯入指令
2. 結構描述程式碼
3. 資料來源產生器
4. 函數程式碼
5. 解析程式碼

從 [設計結構描述](#) 和 [連接資料來源](#) 區段中，我們知道堆疊檔案將包含格式的匯入指令：

```
import * as x from 'x'; # import wildcard as the 'x' keyword from 'x-service'  
import {a, b, ...} from 'c'; # import {specific constructs} from 'c-service'
```

 Note

在先前的章節中，我們只說明了如何匯入 AWS AppSync 建構。在真實程式碼中，您必須匯入更多服務，才能執行應用程式。在我們的範例中，如果我們要建立非常簡單的 CDK 應用程式，我們至少會匯入 AWS AppSync 服務與資料來源，也就是 DynamoDB 資料表。我們也需要匯入一些額外的建構，才能部署應用程式：

```
import * as cdk from 'aws-cdk-lib';  
import * as appsync from 'aws-cdk-lib/aws-appsync';  
import * as dynamodb from 'aws-cdk-lib/aws-dynamodb';  
import { Construct } from 'constructs';
```

若要摘要下列各項：

- `import * as cdk from 'aws-cdk-lib';` : 這可讓您定義 CDK 應用程式和建構，例如堆疊。它還包含一些實用的公用程式函數，例如操作中繼資料。如果您熟悉此匯入指令，但想知道為何此處未使用 `cdk` 核心程式庫，請參閱[遷移](#)頁面。
- `import * as appsync from 'aws-cdk-lib/aws-appsync';` : 這會匯入[AWS AppSync 服務](#)。
- `import * as dynamodb from 'aws-cdk-lib/aws-dynamodb';` : 這會匯入[DynamoDB 服務](#)。
- `import { Construct } from 'constructs';` : 我們需要此項目來定義根[建構](#)。

匯入的類型取決於您呼叫的服務。我們建議您查看 CDK 文件以取得範例。頁面頂端的結構描述會是 CDK 應用程式中的個別檔案，做為 `.graphql` 檔案。在堆疊檔案中，我們可以使用下列格式將其與新的 GraphQL 建立關聯：

```
const add_api = new appsync.GraphqlApi(this, 'graphql-example', {
  name: 'my-first-api',
  schema: appsync.SchemaFile.fromAsset(path.join(__dirname, 'schema.graphql')),
});
```

Note

在範圍 `add_api` 中，我們使用 `new` 關鍵字後面接著 `new appsync.GraphqlApi` 來新增 GraphQL API。我們的範圍是 `this`，CFN id 是 `graphql-example`，而我們的道具是 `my-first-api`（主控台中 API 的名稱）和 `schema.graphql`（結構描述檔案的絕對路徑）。

若要新增資料來源，您必須先將資料來源新增至堆疊。然後，您需要使用來源特定方法將其與 GraphQL API 建立關聯。當您建立解析程式函數時，就會發生關聯。同時，讓我們使用建立 DynamoDB 資料表的範例 `dynamodb.Table`：

```
const add_ddb_table = new dynamodb.Table(this, 'posts-table', {
  partitionKey: {
    name: 'id',
    type: dynamodb.AttributeType.STRING,
  },
});
```

```
});
```

Note

如果我們在範例中使用此值，則會新增 CFN ID 為 `posts-table` 且分割區索引鍵為 的新 DynamoDB 資料表 `id (S)`。

接下來，我們需要在堆疊檔案中實作解析程式。以下是掃描 DynamoDB 資料表中所有項目的簡單查詢範例：

```
const add_func = new appsync.AppsyncFunction(this, 'func-get-posts', {
  name: 'get_posts_func_1',
  add_api,
  dataSource: add_api.addDynamoDbDataSource('table-for-posts', add_ddb_table),
  code: appsync.Code.fromInline(`
    export function request(ctx) {
      return { operation: 'Scan' };
    }

    export function response(ctx) {
      return ctx.result.items;
    }
  `),
  runtime: appsync.FunctionRuntime.JS_1_0_0,
});

new appsync.Resolver(this, 'pipeline-resolver-get-posts', {
  add_api,
  typeName: 'Query',
  fieldName: 'getPost',
  code: appsync.Code.fromInline(`
    export function request(ctx) {
      return {};
    }

    export function response(ctx) {
      return ctx.prev.result;
    }
  `),
  runtime: appsync.FunctionRuntime.JS_1_0_0,
  pipelineConfig: [add_func],
});
```

```
});
```

Note

首先，我們建立了名為 `add_func` 的函數。此建立順序可能看起來有點反直覺，但您必須在管道解析程式中建立函數，才能建立解析程式本身。函數的格式如下：

```
AppsyncFunction(scope: Construct, id: string, props: AppsyncFunctionProps)
```

我們的範圍是 `this`，我們的 CFN ID 是 `func-get-posts`，而我們的道具包含實際的函數詳細資訊。在道具內，我們包括：

- 將出現在 AWS AppSync 主控台的 `name` 函數的 (`get_posts_func_1`)。
- 我們先前建立的 GraphQL API (`add_api`)。
- 資料來源；這是我們將資料來源連結至 GraphQL API 值，然後將其連接至函數的點。我們採用我們建立的資料表 (`add_ddb_table`)，並使用其中一種 `GraphqlApi` 方法 (`add_api`) 將其連接到 GraphQL API (`addDynamoDbDataSource`)。ID 值 (`table-for-posts`) 是主控台中 AWS AppSync 資料來源的名稱。如需來源特定方法的清單，請參閱下列頁面：
 - [DynamoDbDataSource](#)
 - [EventBridgeDataSource](#)
 - [HttpDataSource](#)
 - [LambdaDataSource](#)
 - [NoneDataSource](#)
 - [OpenSearchDataSource](#)
 - [RdsDataSource](#)
- 程式碼包含函數的請求和回應處理常式，這是簡單的掃描和傳回。
- 執行時間指定要使用 `APPSYNC_JS` 執行時間 1.0.0 版。請注意，目前這是 `APPSYNC_JS` 唯一可用的版本。

接下來，我們需要將 函數連接至管道解析程式。我們建立解析程式的格式如下：

```
Resolver(scope: Construct, id: string, props: ResolverProps)
```

我們的範圍是 `this`，我們的 CFN ID 是 `pipeline-resolver-get-posts`，而我們的道具包含實際的函數詳細資訊。在道具中，我們包括：

- 我們先前建立的 GraphQL API (`add_api`)。
- 特殊物件類型名稱；這是查詢操作，因此我們只會新增值 `Query`。
- 欄位名稱 (`getPost`) 是 `Query` 類型下結構描述中的欄位名稱。
- 此程式碼包含處理常式前後的。在函數執行其操作之後，我們的範例只會傳回內容中的任何結果。
- 執行時間指定要使用 `APPSYNC_JS` 執行時間 1.0.0 版。請注意，目前這是 `APPSYNC_JS` 唯一可用的版本。
- 管道組態包含我們所建立函數的參考 (`add_func`)。

為了摘要此範例中發生的情況，您看到了實作請求和回應處理常式的 AWS AppSync 函數。函數負責與您的資料來源互動。請求處理常式將 `Scan` 操作傳送到 AWS AppSync，指示它針對 DynamoDB 資料來源執行的操作。回應處理常式傳回項目清單 (`ctx.result.items`)。然後項目清單會自動映射至 `Post GraphQL` 類型。

建立基本變動解析程式

本節將說明如何建立基本的變動解析程式。

Console

1. 登入 AWS Management Console 並開啟 [AppSync 主控台](#)。
 - a. 在 APIs 儀表板中，選擇您的 GraphQL API。
 - b. 在側邊欄中，選擇結構描述。
2. 在解析程式區段和變動類型下，選擇欄位旁的連接。

Note

在我們的範例中，我們正在連接的解析程式 `createPost`，將 `Post` 物件新增至我們的資料表。假設我們使用上一節的相同 DynamoDB 資料表。其分割區索引鍵設定為 `id`，且為空白。

3. 在連接解析程式頁面的解析程式類型下，選擇 pipeline resolvers。提醒您，您可以[在這裡](#)找到有關解析程式的詳細資訊。針對解析程式執行時間，選擇 APPSYNC_JS 以啟用 JavaScript 執行時間。
4. 您可以為此 API 啟用[快取](#)。我們建議您立即關閉此功能。選擇建立。
5. 選擇新增函數，然後選擇建立新函數。或者，您可能會看到建立函數按鈕來選擇。
 - a. 選擇您的資料來源。這應該是您使用變動操作資料的來源。
 - b. 輸入 Function name。
 - c. 在函數程式碼下，您需要實作函數的行為。這是一個變動，因此請求最好會對調用的資料來源執行一些狀態變更操作。結果將由回應函數處理。

 Note

createPost 正在新增或 "putting" 資料表 Post 中的新，並將我們的參數做為資料。我們可以新增如下內容：

```
import { util } from '@aws-appsync/utils';

/**
 * Sends a request to `put` an item in the DynamoDB data source
 */
export function request(ctx) {
  return {
    operation: 'PutItem',
    key: util.dynamodb.toMapValues({id: util.autoId()}),
    attributeValues: util.dynamodb.toMapValues(ctx.args.input),
  };
}

/**
 * returns the result of the `put` operation
 */
export function response(ctx) {
  return ctx.result;
}
```

在此步驟中，我們也新增了 request 和 response 函數：

- request：請求處理常式接受內容做為引數。請求處理常式傳回陳述式會執行 [PutItem](#) 命令，這是內建的 DynamoDB 操作（如需範例，請參閱[此處](#)或[此](#)

處)。PutItem 命令會採用分割區key值（由自動產生util.autoid()）並從內容引數輸入 attributes（這些是我們將在請求中傳遞的值），將Post物件新增至 DynamoDB 資料表。key 是 id，而 attributes 是 date 和 title 欄位引數。它們都是透過 [util.dynamodb.toMapValues](#) 協助程式預先格式化，以使用 DynamoDB 資料表。

- response：回應接受更新的內容，並傳回請求處理常式的結果。

d. 完成後，請選擇建立。

6. 返回解析程式畫面的函數下，選擇新增函數下拉式清單，並將函數新增至函數清單。
7. 選擇儲存以更新解析程式。

CLI

新增您的 函數

- 使用 [create-function](#) 命令為您的管道解析程式建立 函數。

您需要為此特定命令輸入幾個參數：

1. 您 API api-id 的。
2. AWS AppSync 主控台中 name 函數的。
3. data-source-name、或 函數將使用的資料來源名稱。它必須已在 AWS AppSync 服務中建立並連結至您的 GraphQL API。
4. 函數的 runtime、或 環境和語言。對於 JavaScript，名稱必須為 APPSYNC_JS，執行期為 1.0.0。
5. 函數的 code、或 請求和回應處理常式。雖然您可以手動輸入，但將其新增至 .txt 檔案（或類似格式），然後將其做為引數傳入會更為容易。

Note

我們的查詢程式碼將位於傳入為引數的檔案中：

```
import { util } from '@aws-appsync/utils';

/**
 * Sends a request to `put` an item in the DynamoDB data source
 */
export function request(ctx) {
```

```

    return {
      operation: 'PutItem',
      key: util.dynamodb.toMapValues({id: util.autoId()}),
      attributeValues: util.dynamodb.toMapValues(ctx.args.input),
    };
  }

  /**
   * returns the result of the `put` operation
   */
  export function response(ctx) {
    return ctx.result;
  }

```

範例命令可能如下所示：

```

aws appsync create-function \
--api-id abcdefghijklmnopqrstuvwxyz \
--name add_posts_func_1 \
--data-source-name table-for-posts \
--runtime name=APPSYNC_JS,runtimeVersion=1.0.0 \
--code file:///path/to/file/{filename}.{fileType}

```

輸出會在 CLI 中傳回。範例如下：

```

{
  "functionConfiguration": {
    "functionId": "vulcmbfcxffiram63psb4ddua",
    "functionArn": "arn:aws:appsync:us-west-2:107289374856:apis/
abcdefghijklmnopqrstuvwxyz/functions/vulcmbfcxffiram63psb4ddua",
    "name": "add_posts_func_1",
    "dataSourceName": "table-for-posts",
    "maxBatchSize": 0,
    "runtime": {
      "name": "APPSYNC_JS",
      "runtimeVersion": "1.0.0"
    },
    "code": "Code output foes here"
  }
}

```

 Note

請務必在 `functionId` 某處記錄，因為這將用於將函數連接至解析程式。

建立您的解析程式

- 執行 `create-resolver` 命令 Mutation，為 建立管道函數。

您需要為此特定命令輸入幾個參數：

1. 您 API `api-id` 的。
2. `type-name` 結構描述中的 或特殊物件類型（查詢、變動、訂閱）。
3. 在您欲連接解析程式之特殊物件類型內的 `field-name` 或 欄位操作。
4. `kind`，指定單位或管道解析程式。將此設定為 `PIPELINE` 以啟用管道函數。
5. 要連接到解析程式的 `pipeline-config`（或）函數。請確定您知道函數 `functionId` 的值。列出的順序很重要。
6. `runtime`，也就是 `APPSYNC_JS`(JavaScript)。 `runtimeVersion` 目前為 `1.0.0`。
7. `code`，其中包含步驟前後的。

 Note

我們的查詢程式碼將位於傳入為引數的檔案中：

```
import { util } from '@aws-appsync/utils';

/**
 * Sends a request to `put` an item in the DynamoDB data source
 */
export function request(ctx) {
  const { id, ...values } = ctx.args;
  return {
    operation: 'PutItem',
    key: util.dynamodb.toMapValues({ id }),
    attributeValues: util.dynamodb.toMapValues(values),
  };
}
```

```
/**
 * returns the result of the `put` operation
 */
export function response(ctx) {
  return ctx.result;
}
```

範例命令可能如下所示：

```
aws appsync create-resolver \
--api-id abcdefghijklmnopqrstuvwxyz \
--type-name Mutation \
--field-name createPost \
--kind PIPELINE \
--pipeline-config functions=vulcmbfcxffiram63psb4ddua \
--runtime name=APPSYNC_JS,runtimeVersion=1.0.0 \
--code file:///path/to/file/{filename}.{fileType}
```

輸出會在 CLI 中傳回。範例如下：

```
{
  "resolver": {
    "typeName": "Mutation",
    "fieldName": "createPost",
    "resolverArn": "arn:aws:appsync:us-west-2:107289374856:apis/
abcdefghijklmnopqrstuvwxyz/types/Mutation/resolvers/createPost",
    "kind": "PIPELINE",
    "pipelineConfig": {
      "functions": [
        "vulcmbfcxffiram63psb4ddua"
      ]
    },
    "maxBatchSize": 0,
    "runtime": {
      "name": "APPSYNC_JS",
      "runtimeVersion": "1.0.0"
    },
    "code": "Code output goes here"
  }
}
```

CDK

 Tip

在使用 CDK 之前，建議您檢閱 CDK 的[官方文件](https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws_appsync-readme.html)以及 CDK AWS AppSync參考。https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws_appsync-readme.html

下列步驟只會顯示用來新增特定資源之程式碼片段的一般範例。這不是生產程式碼中的工作解決方案。我們也假設您已有運作中的應用程式。

- 若要進行變動，假設您在同一個專案中，您可以將它新增至堆疊檔案，例如查詢。以下是變動的修改函數和解析程式，Post會將新的 新增至資料表：

```
const add_func_2 = new appsync.AppsyncFunction(this, 'func-add-post', {
  name: 'add_posts_func_1',
  add_api,
  dataSource: add_api.addDynamoDbDataSource('table-for-posts-2', add_ddb_table),
  code: appsync.Code.fromInline(`
    export function request(ctx) {
      return {
        operation: 'PutItem',
        key: util.dynamodb.toMapValues({id: util.autoId()}),
        attributeValues: util.dynamodb.toMapValues(ctx.args.input),
      };
    }

    export function response(ctx) {
      return ctx.result;
    }
  `),
  runtime: appsync.FunctionRuntime.JS_1_0_0,
});

new appsync.Resolver(this, 'pipeline-resolver-create-posts', {
  add_api,
  typeName: 'Mutation',
  fieldName: 'createPost',
  code: appsync.Code.fromInline(`
    export function request(ctx) {
      return {};
    }
  `)
```

```
    export function response(ctx) {
      return ctx.prev.result;
    }
  },
  runtime: appsync.FunctionRuntime.JS_1_0_0,
  pipelineConfig: [add_func_2],
});
```

Note

由於此變動和查詢結構類似，我們只會說明我們為了進行變動所做的變更。在函數中，我們將 CFN ID 變更為 func-add-post，並將名稱變更為 add_posts_func_1，以反映我們將 Posts 新增至資料表的事實。在資料來源中，我們在 AWS AppSync 主控台中與資料表 (add_ddb_table) 建立新的關聯，table-for-posts-2 因為 addDynamoDbDataSource 方法需要它。請記住，這個新的關聯仍然使用我們先前建立的相同資料表，但我們現在在 AWS AppSync 主控台中有兩個與其連線：一個用於查詢，table-for-posts 另一個用於變動 table-for-posts-2。程式碼已變更為新增，Post 方法是自動產生其 id 值，並接受用戶端對其餘欄位的輸入。在解析程式中，我們將 ID 值變更為 pipeline-resolver-create-posts，以反映我們將 Posts 新增至資料表的事實。為了反映結構描述中的變動，類型名稱已變更為 Mutation，而名為 createPost。管道組態已設定為我們的新變動函數 add_func_2。

為了摘要此範例中發生的情況，AWS AppSync 會自動將 createPost 欄位中定義的引數從 GraphQL 結構描述轉換為 DynamoDB 操作。此範例使用金鑰在 DynamoDB 中存放記錄 id，該金鑰是使用我們的 `util.autoId()` 協助程式自動建立的。您從 AWS AppSync 主控台提出的請求或以其他方式傳遞到內容引數 (`ctx.args.input`) 的所有其他欄位都會儲存為資料表的屬性。金鑰和屬性都會使用 `util.dynamodb.toMapValues(values)` 協助程式自動對應至相容的 DynamoDB 格式。

AWS AppSync 也支援用於編輯解析程式的測試和偵錯工作流程。您可以使用模擬 context 物件，在叫用範本之前查看範本的轉換值。或者，您可以在執行查詢時以互動方式檢視對資料來源的完整請求。如需詳細資訊，請參閱[測試和偵錯解析程式 \(JavaScript\)](#) 和[監控和記錄](#)。

進階解析程式

如果您遵循[設計結構描述](#)中的選用分頁區段，您仍然需要將解析程式新增至您的請求，才能使用分頁。我們的範例使用稱為 `getPosts` 的查詢分頁，一次只傳回請求的一部分物件。解析程式在該欄位上的程式碼可能如下所示：

```
/**
 * Performs a scan on the dynamodb data source
 */
export function request(ctx) {
  const { limit = 20, nextToken } = ctx.args;
  return { operation: 'Scan', limit, nextToken };
}

/**
 * @returns the result of the `put` operation
 */
export function response(ctx) {
  const { items: posts = [], nextToken } = ctx.result;
  return { posts, nextToken };
}
```

在請求中，我們會傳入請求的內容。我們的 `limit` 是 `20`，這表示我們在第一個查詢 `Posts` 中傳回最多 20 個。我們的 `nextToken` 游標會固定為資料來源中的第一個 `Post` 項目。這些會傳遞給 `args`。然後，請求會執行從第一個 `Post` 到掃描限制數量的掃描。資料來源會將結果存放在內容中，該內容會傳遞給回應。回應會傳回 `Posts` 其擷取的，然後將 `nextToken` 設定為限制之後 `Post` 的項目。下一個請求會傳送到以執行完全相同的物件，但在第一個查詢後立即從位移開始。請記住，這些類型的請求會循序完成，而不是平行完成。

測試和偵錯解析程式 in AWS AppSync (JavaScript)

AWS AppSync 會對資料來源在 GraphQL 欄位上執行解析程式。使用管道解析程式時，函數會與您的資料來源互動。如 [JavaScript 解析程式概觀](#) 中所述，函數會使用以 JavaScript 撰寫並在 `APPSYNC_JS` 執行時間執行的請求和回應處理常式，與資料來源通訊。這可讓您在與資料來源通訊之前和之後提供自訂邏輯和條件。

為了協助開發人員寫入、測試和偵錯這些解析程式，AWS AppSync 主控台也提供工具來建立 GraphQL 請求和回應，並將模擬資料縮減至個別欄位解析程式。此外，您可以在 AWS AppSync 主控台中執行查詢、變動和訂閱，並查看來自 Amazon CloudWatch 的整個請求的詳細日誌串流。這包括來自資料來源的結果。

使用模擬資料進行測試

叫用 GraphQL 解析程式時，它包含具有請求相關資訊的 context 物件。其中包括用戶端引數、身分資訊，以及父 GraphQL 欄位的資料。它也會存放資料來源的結果，可用於回應處理常式。如需此結構和程式設計時要使用之可用協助程式公用程式的詳細資訊，請參閱[解析程式內容物件參考](#)。

寫入或編輯解析程式函數時，您可以將模擬或測試內容物件傳遞至主控台編輯器。這可讓您查看請求和回應處理常式如何評估，而不會實際針對資料來源執行。例如您可傳送測試 `firstname: Shaggy` 引數，了解該引數在範本程式碼之中使用 `ctx.args.firstname` 時如何進行評估。您也可以測試任何公用程式協助程式的評估，例如 `util.autoId()` 或 `util.time.nowISO8601()`。

測試解析程式

此範例將使用 AWS AppSync 主控台來測試解析程式。

1. 登入 AWS Management Console 並開啟 [AppSync 主控台](#)。
 - a. 在 APIs 儀表板中，選擇您的 GraphQL API。
 - b. 在 Sidebar 中，選擇函數。
2. 選擇現有的 函數。
3. 在更新函數頁面頂端，選擇選取測試內容，然後選擇建立新內容。
4. 選取範例內容物件，或在下面的設定測試內容視窗中手動填入 JSON。
5. 輸入文字內容名稱。
6. 選擇 Save (儲存) 按鈕。
7. 若要使用此項模擬的內容物件評估解析程式，選擇 Run Test (執行測試) 按鈕。

如需更實際的範例，假設您有一個應用程式存放的 GraphQL 類型 `Dog`，該類型使用物件的自動 ID 產生，並將它們存放在 Amazon DynamoDB 中。您也想要從 GraphQL 變動的引數寫入一些值，並僅允許特定使用者查看回應。下列程式碼片段顯示結構描述的外觀：

```
type Dog {
  breed: String
  color: String
}

type Mutation {
  addDog(firstname: String, age: Int): Dog
```

```
}
```

您可以撰寫 an AWS AppSync 函數，並將其新增至您的addDog解析程式來處理變動。若要測試您的 AWS AppSync 函數，您可以填入內容物件，如下列範例所示。下列引數來自 name 及 age，以及將 username 填入 identity 物件之中的用戶端：

```
{
  "arguments" : {
    "firstname": "Shaggy",
    "age": 4
  },
  "source" : {},
  "result" : {
    "breed" : "Miniature Schnauzer",
    "color" : "black_grey"
  },
  "identity": {
    "sub" : "uuid",
    "issuer" : " https://cognito-idp.{region}.amazonaws.com/{userPoolId}",
    "username" : "Nadia",
    "claims" : { },
    "sourceIp" :[  "x.x.x.x" ],
    "defaultAuthStrategy" : "ALLOW"
  }
}
```

您可以使用下列程式碼測試您的 AWS AppSync 函數：

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
  return {
    operation: 'PutItem',
    key: util.dynamodb.toMapValues({ id: util.autoId() }),
    attributeValues: util.dynamodb.toMapValues(ctx.args),
  };
}

export function response(ctx) {
  if (ctx.identity.username === 'Nadia') {
    console.log("This request is allowed")
    return ctx.result;
  }
}
```

```
}  
  util.unauthorized();  
}
```

評估的請求和回應處理常式具有來自測試內容物件的資料，以及來自產生的值`util.autoId()`。此外，若您將 `username` 變更為 `Nadia` 以外的值，將不會傳回結果，因為授權檢查將會失敗。如需精細存取控制的詳細資訊，請參閱[授權使用案例](#)。

使用 AWS AppSync 的 APIs

您可以使用 `EvaluateCode` API 命令，以模擬資料遠端測試程式碼。若要開始使用命令，請確定您已將 `appsync:evaluateMappingCode` 許可新增至政策。例如：

JSON

```
{  
  "Version": "2012-10-17",  
  "Statement": [  
    {  
      "Effect": "Allow",  
      "Action": "appsync:evaluateCode",  
      "Resource": "arn:aws:appsync:<region>:<account>:*"  
    }  
  ]  
}
```

您可以使用 [AWS CLI](#) 或 [AWS SDKs](#) 來利用命令。例如，從上一節取得結構描述及其 AWS AppSync Dog 函數請求和回應處理常式。在本機工作站上使用 CLI，將程式碼儲存到名為 `code.js` 的檔案，然後將 `context` 物件儲存到名為 `context.json` 的檔案。從您的 shell 執行下列命令：

```
$ aws appsync evaluate-code \  
  --code file://code.js \  
  --function response \  
  --context file://context.json \  
  --runtime name=APPSYNC_JS,runtimeVersion=1.0.0
```

回應包含 `evaluationResult` 其中包含處理常式傳回的承載。它還包含物件，該 `logs` 物件會保留您的處理常式在評估期間產生的日誌清單。這可讓您輕鬆地偵錯程式碼執行，並查看評估的相關資訊，以協助故障診斷。例如：

```
{
  "evaluationResult": "{\\\"breed\\\":\\\"Miniature Schnauzer\\\",\\\"color\\\":\\\"black_grey\\\"}",
  "logs": [
    "INFO - code.js:13:5: \\\"This request is allowed\\\""
  ]
}
```

`evaluationResult` 可以剖析為 JSON，它提供：

```
{
  "breed": "Miniature Schnauzer",
  "color": "black_grey"
}
```

使用 SDK，您可以輕鬆整合來自您最愛測試套件的測試，以驗證處理常式的行為。我們建議您使用 [Jest 測試架構](#) 建立測試，但任何測試套件都可以運作。下列程式碼片段顯示假設驗證執行。請注意，我們預期評估回應是有效的 JSON，因此我們使用 `從字串回應JSON.parse` 擷取 JSON：

```
const AWS = require('aws-sdk')
const fs = require('fs')
const client = new AWS.AppSync({ region: 'us-east-2' })
const runtime = {name:'APPSYNC_JS',runtimeVersion:'1.0.0'}

test('request correctly calls DynamoDB', async () => {
  const code = fs.readFileSync('./code.js', 'utf8')
  const context = fs.readFileSync('./context.json', 'utf8')
  const contextJSON = JSON.parse(context)

  const response = await client.evaluateCode({ code, context, runtime, function:
'request' }).promise()
  const result = JSON.parse(response.evaluationResult)

  expect(result.key.id.S).toBeDefined()
  expect(result.attributeValues.firstname.S).toEqual(contextJSON.arguments.firstname)
})
```

這會產生下列結果：

```
Ran all test suites.
> jest
```

```
PASS ./index.test.js
# request correctly calls DynamoDB (543 ms)
Test Suites: 1 passed, 1 total
Tests: 1 passed, 1 total
Snapshots: 0 totalTime: 1.511 s, estimated 2 s
```

偵錯即時查詢

端對端 end-to-end 測試和記錄無法用來偵錯生產應用程式。AWS AppSync 可讓您使用 Amazon CloudWatch 記錄錯誤和完整請求詳細資訊。此外，您可以使用 AWS AppSync 主控台測試 GraphQL 查詢、變動和訂閱，以及將每個請求的即時串流日誌資料傳回查詢編輯器以進行即時偵錯。針對訂閱，日誌會顯示連線時間資訊。

若要執行此操作，您需要事先啟用 Amazon CloudWatch logs，如[監控和記錄](#)中所述。接著，在 AWS AppSync 主控台中，選擇查詢索引標籤，然後輸入有效的 GraphQL 查詢。在右下方區段中，按一下並拖曳日誌視窗以開啟日誌檢視。在頁面頂端，選擇執行箭頭圖示來執行您的 GraphQL 查詢。稍後，您操作的完整請求和回應日誌會串流到本節，您可以在 主控台中檢視它們。

在 AWS AppSync (JavaScript) 中設定和使用管道解析程式

AWS AppSync 會在 GraphQL 欄位上執行解析程式。在某些情況下，應用程式需要執行多個操作，才能解析單一 GraphQL 欄位。透過管道解析程式，開發人員現在可以編寫稱為 函數的操作，並依序執行這些操作。在像是以需要先執行授權檢查才能擷取欄位資料的情況中，就很適合使用管道解析程式。

如需 JavaScript 管道解析程式架構的詳細資訊，請參閱 [JavaScript 解析程式概觀](#)。

步驟 1：建立管道解析程式

在 AWS AppSync 主控台中，前往結構描述頁面。

儲存下列結構描述：

```
schema {
  query: Query
  mutation: Mutation
}

type Mutation {
  signUp(input: Signup): User
}

type Query {
```

```
    getUser(id: ID!): User
  }

  input Signup {
    username: String!
    email: String!
  }

  type User {
    id: ID!
    username: String
    email: AWSEmail
  }
```

我們將將管道解析程式連接到 變動類型的 signUp 欄位。在右側的變動類型中，選擇signUp變動欄位旁的連接。將解析程式設定為 pipeline resolver和APPSYNC_JS執行時間，然後建立解析程式。

我們的管道解析程式會註冊使用者，此註冊的第一步是驗證電子郵件地址輸入，然後在系統中儲存此位使用者。我們會將電子郵件驗證封裝在 validateEmail 函數內，並將使用者儲存在 saveUser 函數內。validateEmail 函數會先執行，而當電子郵件驗證有效時，saveUser 函數就會接著執行。

執行流程如下：

1. Mutation.signUp 解析程式請求處理常式
2. validateEmail 函數
3. saveUser 函數
4. Mutation.signUp 解析程式回應處理常式

由於我們可能會在 API 上的其他解析程式中重複使用 validateEmail 函數，因此我們希望避免存取，ctx.args因為這些函數會從一個 GraphQL 欄位變更為另一個欄位。反之，我們可以使用 ctx.stash 來存放 signUp(input: Signup) 輸入欄位引數所傳遞的電子郵件屬性。

透過取代您的請求和回應函數來更新您的解析程式程式碼：

```
export function request(ctx) {
  ctx.stash.email = ctx.args.input.email
  return {}
}

export function response(ctx) {
```

```
    return ctx.prev.result;
  }
```

選擇建立或儲存以更新解析程式。

步驟 2：建立 函數

在管道解析程式頁面的函數區段中，按一下新增函數，然後按一下建立新函數。您也可以建立函數而不經過解析程式頁面；若要這樣做，請在 AWS AppSync 主控台中前往函數頁面。選擇 Create function (建立函數) 按鈕。讓我們來建立可檢查電子郵件是否有效且來源是特定網域的函數。如果電子郵件無效，則該函數會引發錯誤。否則，它會轉送任何獲予的任何輸入。

請確定您已建立 NONE 類型的資料來源。在資料來源名稱清單中選擇此資料來源。在函數名稱中輸入 validateEmail。在函數程式碼區域中，使用此程式碼片段覆寫所有項目：

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
  const { email } = ctx.stash;
  const valid = util.matches(
    '^[a-zA-Z0-9_.+-]+@(?:([a-zA-Z0-9-]+\\.)?[a-zA-Z]+\\.)(myvaliddomain)\\.com',
    email
  );
  if (!valid) {
    util.error(`"${email}" is not a valid email.`);
  }

  return { payload: { email } };
}

export function response(ctx) {
  return ctx.result;
}
```

檢閱您的輸入，然後選擇建立。我們已建立我們 validateEmail 函數。重複這些步驟，以使用下列程式碼建立 saveUser 函數（為了簡化，我們使用 NONE 資料來源，並假設使用者在函數執行後已儲存在系統中。）：

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
```

```
    return ctx.prev.result;
  }

  export function response(ctx) {
    ctx.result.id = util.autoId();
    return ctx.result;
  }
```

我們剛建立了 `saveUser` 函數。

步驟 3：將函數新增至管道解析程式

我們的函數應該會自動新增至我們剛建立的管道解析程式。如果不是這種情況，或者您透過函數頁面建立函數，您可以按一下 `signUp` 解析程式頁面上的新增函數來連接函數。將 `validateEmail` 和 `saveUser` 函數新增至解析程式。`validateEmail` 函數應該放在 `saveUser` 函數之前。當您新增更多函數時，您可以使用上下移動選項來重組函數的執行順序。檢閱您的變更，然後選擇儲存。

步驟 4：執行查詢

在 AWS AppSync 主控台中，前往查詢頁面。在 Explorer 中，請確定您使用的是變動。如果不是，請在下拉式清單 Mutation 中選擇，然後選擇 +。輸入下列查詢：

```
mutation {
  signUp(input: {email: "nadia@myvaliddomain.com", username: "nadia"}) {
    id
    username
  }
}
```

這應該會傳回如下內容：

```
{
  "data": {
    "signUp": {
      "id": "256b6cc2-4694-46f4-a55e-8cb14cc5d7fc",
      "username": "nadia"
    }
  }
}
```

我們已使用管道解析程式成功註冊我們的使用者，並完成輸入電子郵件的驗證。

建立基本查詢 (VTL)

Note

我們現在主要支援 APPSYNC_JS 執行期及其文件。請考慮[在此處](#)使用 APPSYNC_JS 執行期及其指南。

GraphQL 解析程式將類型結構描述中的欄位連接到資料來源。解析程式是滿足請求的機制。AWS AppSync 可以從結構描述自動建立和連接解析程式，或從現有資料表建立結構描述並連接解析程式，而無需撰寫任何程式碼。

Resolvers in AWS AppSync 使用 JavaScript 將 GraphQL 表達式轉換為資料來源可以使用的格式。或者，映射範本可以用 [Apache Velocity 範本語言 \(VTL\)](#) 撰寫，將 GraphQL 表達式轉換為資料來源可以使用的格式。

本節將說明如何使用 VTL 設定解析程式。您可以在[解析程式映射範本程式設計指南中找到編寫解析程式的入門教學風格程式設計指南](#)，以及可在[解析程式映射範本內容參考](#)中找到程式設計時使用的協助程式公用程式。AWS AppSync 也有內建測試和偵錯流程，可讓您從頭開始編輯或撰寫時使用。如需詳細資訊，請參閱[測試和偵錯解析程式](#)。

我們建議您先遵循本指南，再嘗試使用任何上述教學課程。

在本節中，我們將逐步解說如何建立解析程式、為變動新增解析程式，以及使用進階組態。

建立您的第一個解析程式

遵循先前章節的範例，第一個步驟是為您的 Query 類型建立解析程式。

Console

1. 登入 AWS Management Console 並開啟 [AppSync 主控台](#)。
 - a. 在 APIs 儀表板中，選擇您的 GraphQL API。
 - b. 在側邊欄中，選擇結構描述。
2. 在頁面右側，有一個稱為解析程式的視窗。此方塊包含頁面左側結構描述視窗中定義的類型和欄位清單。您可以將解析程式連接至欄位。例如，在查詢類型下，選擇 getTodos 欄位旁的連接。
3. 在建立解析程式頁面上，選擇您在[連接資料來源指南中建立的資料來源](#)。在設定映射範本視窗中，您可以使用右側的下拉式清單選擇一般請求和回應映射範本，或自行撰寫。

Note

將請求映射範本與回應映射範本配對稱為單位解析程式。單位解析程式通常用於執行轉換操作；我們建議僅將它們用於具有少量資料來源的單一操作。對於更複雜的操作，我們建議使用管道解析程式，其可以依序使用多個資料來源執行多個操作。

如需請求與回應映射範本之間差異的詳細資訊，請參閱[單位解析程式](#)。

如需使用管道解析程式的詳細資訊，請參閱[管道解析程式](#)。

- 對於常見的使用案例，AWS AppSync 主控台具有內建範本，可用來從資料來源取得項目（例如，所有項目查詢、個別查詢等）。例如，在設計getTodos沒有分頁的[結構](#)描述的簡單版本上，列出項目的請求映射範本如下所示：

```
{
  "version" : "2017-02-28",
  "operation" : "Scan"
}
```

- 您一律需要回應映射範本來隨附請求。主控台為清單提供具有下列傳遞值的預設值：

```
$util.toJson($ctx.result.items)
```

在這個範例中，項目清單的 context 物件（別名為 \$ctx）為 \$context.result.items 格式。如果您的 GraphQL 操作傳回單一項目，則會是 \$context.result。AWS AppSync 提供常見操作的協助程式函數，例如先前列出的 \$util.toJson 函數，以正確格式化回應。如需函數的完整清單，請參閱[解析程式映射範本公用程式參考](#)。

- 選擇儲存解析程式。

API

- 呼叫 [CreateResolver](#) API 來建立解析程式物件。
- 您可以呼叫 [UpdateResolver](#) API 來修改解析程式的欄位。

CLI

- 執行 [create-resolver](#) 命令來建立解析程式。

您需要為此特定命令輸入 6 個參數：

1. 您 API `api-id` 的。
2. 您想要在結構描述中修改 `type-name` 的類型。在主控台範例中，這是 `Query`。
3. 您要在類型中修改的欄位 `field-name` 的。在主控台範例中，這是 `getTodos`。
4. 您在連接資料來源指南中建立 `data-source-name` 的資料來源的。 <https://docs.aws.amazon.com/appsync/latest/devguide/attaching-a-data-source.html>
5. `request-mapping-template`，這是請求的內文。在主控台範例中，這是：

```
{
  "version" : "2017-02-28",
  "operation" : "Scan"
}
```

6. `response-mapping-template`，這是回應的內文。在主控台範例中，這是：

```
$util.toJson($ctx.result.items)
```

範例命令可能如下所示：

```
aws appsync create-resolver --api-id abcdefghijklmnopqrstuvwxyz --type-name
Query --field-name getTodos --data-source-name TodoTable --request-mapping-
template "{ \"version\" : \"2017-02-28\", \"operation\" : \"Scan\", }" --response-
mapping-template "\"$util.toJson('$ctx.result.items')"
```

輸出會在 CLI 中傳回。範例如下：

```
{
  "resolver": {
    "kind": "UNIT",
    "dataSourceName": "TodoTable",
    "requestMappingTemplate": "{ version : 2017-02-28, operation : Scan, }",
    "resolverArn": "arn:aws:appsync:us-west-2:107289374856:apis/
abcdefghijklmnopqrstuvwxyz/types/Query/resolvers/getTodos",
    "typeName": "Query",
    "fieldName": "getTodos",
    "responseMappingTemplate": "$util.toJson($ctx.result.items)"
  }
}
```

- 若要修改解析程式的欄位和/或映射範本，請執行 [update-resolver](#) 命令。

除了 `api-id` 參數之外，`create-resolver` 命令中使用的參數將由 `update-resolver` 命令的新值覆寫。

新增變動的解析程式

下一個步驟是為您的 `Mutation` 類型建立解析程式。

Console

- 登入 AWS Management Console 並開啟 [AppSync 主控台](#)。
 - 在 APIs 儀表中，選擇您的 GraphQL API。
 - 在側邊欄中，選擇結構描述。
- 在變動類型下，選擇 `addTodo` 欄位旁的連接。
- 在建立解析程式頁面上，選擇您在 [連接資料來源指南中建立的資料來源](#)。
- 在設定映射範本視窗中，您將需要修改請求範本，因為這是您要將新項目新增至 DynamoDB 的變動。請使用下列要求映射範本：

```
{
  "version" : "2017-02-28",
  "operation" : "PutItem",
  "key" : {
    "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
  },
  "attributeValues" : $util.dynamodb.toMapValuesJson($ctx.args)
}
```

- AWS AppSync 會自動將 `addTodo` 欄位中定義的引數從 GraphQL 結構描述轉換為 DynamoDB 操作。上一個範例使用 索引鍵將記錄存放在 DynamoDB 中 `id`，該索引鍵會從變動引數傳遞為 `$ctx.args.id`。您傳遞的所有其他欄位都會使用 自動映射至 DynamoDB 屬性 `$util.dynamodb.toMapValuesJson($ctx.args)`。

在此解析程式中，使用下列的回應映射範本：

```
$util.toJson($ctx.result)
```

AWS AppSync 也支援用於編輯解析程式的測試和偵錯工作流程。您可以使用模擬 context 物件，先查看範本轉換後的值，然後再叫用。或者，您可以在執行查詢時以互動方式檢視對資料來源的完整要求執行。如需詳細資訊，請參閱[測試和偵錯解析程式](#)以及[監控和記錄](#)。

6. 選擇儲存解析程式。

API

您也可以使用[建立第一個解析程式](#)區段中的命令和本節中的參數詳細資訊，以 APIs 執行此操作。

CLI

您也可以使用[建立第一個解析程式](#)區段中的命令和本節中的參數詳細資訊，在 CLI 中執行此操作。

此時，如果您未使用進階解析程式，您可以開始使用 GraphQL API，如[使用 API](#)中所述。

進階解析程式

如果您遵循進階區段，並在[設計結構描述以執行分頁掃描中建置範例結構描述](#)，請改用下列請求範本做為 getTodos 欄位：

```
{
  "version" : "2017-02-28",
  "operation" : "Scan",
  "limit": $util.defaultIfNull(${ctx.args.limit}, 20),
  "nextToken": $util.toJson($util.defaultIfNullOrBlank(${ctx.args.nextToken}, null))
}
```

對此分頁使用案例而言，回應映射不只是傳遞，因為其中必須包含游標 (讓用戶端了解接下來從哪個頁面開始) 和結果集。映射範本如下所示：

```
{
  "todos": $util.toJson($context.result.items),
  "nextToken": $util.toJson($context.result.nextToken)
}
```

前述回應映射範本的欄位，應符合 TodoConnection 類型之中定義的欄位。

對於您擁有 Comments 資料表且正在解析 Todo 類型 (傳回類型的 [Comment]) 註解欄位的關係，您可以使用對第二個資料表執行查詢的映射範本。若要這樣做，您必須已經建立 Comments 資料表的資料來源，如[連接資料來源](#)中所述。

Note

我們針對第二個資料表使用查詢操作，僅供說明之用。您可以改為對 DynamoDB 使用其他操作。此外，您可以從其他資料來源提取資料，例如 AWS Lambda 或 Amazon OpenSearch Service，因為關係是由 GraphQL 結構描述控制。

Console

1. 登入 AWS Management Console 並開啟 [AppSync 主控台](#)。
 - a. 在 APIs 儀表板中，選擇您的 GraphQL API。
 - b. 在側邊欄中，選擇結構描述。
2. 在待辦事項類型下，選擇 comments 欄位旁的連接。
3. 在建立解析程式頁面上，選擇您的評論資料表資料來源。快速入門指南中註解資料表的預設名稱為 AppSyncCommentTable，但可能會因您提供的名稱而有所不同。
4. 將下列程式碼片段新增至您的請求映射範本：

```
{
  "version": "2017-02-28",
  "operation": "Query",
  "index": "todoid-index",
  "query": {
    "expression": "todoid = :todoid",
    "expressionValues": {
      ":todoid": {
        "S": $util.toJson($context.source.id)
      }
    }
  }
}
```

5. context.source 參照目前正在解析之欄位的父物件。在這個範例中，source.id 代表個別 Todo 物件，這個物件會在稍後用於查詢表達式。

您可使用傳遞回應映射範本，如下所示：

```
$util.toJson($ctx.result.items)
```

6. 選擇儲存解析程式。

7. 最後，回到 主控台的結構描述頁面，將解析程式連接到 addComment 欄位，並指定 Comments 資料表的資料來源。在這種情況下，要求映射範本是簡單的 PutItem，其中具有註解為引數的特定 todoid，但您需要使用 \$utils.autoId() 公用程式來為註解建立唯一的排序索引鍵，如下所示：

```
{
  "version": "2017-02-28",
  "operation": "PutItem",
  "key": {
    "todoid": { "S": $util.toJson($context.arguments.todoid) },
    "commentid": { "S": "$util.autoId()" }
  },
  "attributeValues" : $util.dynamodb.toMapValuesJson($ctx.args)
}
```

使用傳遞回應範本，如下所示：

```
$util.toJson($ctx.result)
```

API

您也可以使用[建立第一個解析程式](#)區段中的命令和本節中的參數詳細資訊，以 APIs 執行此操作。

CLI

您也可以使用[建立第一個解析程式](#)區段中的命令和本節中的參數詳細資訊，在 CLI 中執行此操作。

使用直接 Lambda 解析程式 (VTL) 停用 VTL 映射範本

Note

我們現在主要支援 APPSYNC_JS 執行期及其文件。請考慮[在此處](#)使用 APPSYNC_JS 執行期及其指南。

使用直接 Lambda 解析程式時，您可以避免使用 VTL 映射範本 AWS Lambda。AWS AppSync 可以為您的 Lambda 函數提供預設承載，以及從 Lambda 函數對 GraphQL 類型的回應進行預設轉譯。您可以選擇提供請求範本、回應範本，或由 AWS AppSync 處理。

若要進一步了解 AWS AppSync 提供的預設請求承載和回應轉譯，請參閱 [Direct Lambda 解析程式參考](#)。如需設定 AWS Lambda 資料來源和設定 IAM 信任政策的詳細資訊，請參閱[連接資料來源](#)。

設定直接 Lambda 解析程式

下列各節將說明如何連接 Lambda 資料來源，並將 Lambda 解析程式新增至您的欄位。

新增 Lambda 資料來源

您必須先新增 Lambda 資料來源，才能啟用直接 Lambda 解析程式。

Console

1. 登入 AWS Management Console 並開啟 [AppSync 主控台](#)。
 - a. 在 APIs 儀表板中，選擇您的 GraphQL API。
 - b. 在側邊欄中，選擇資料來源。
2. 選擇 Create data source (建立資料來源)。
 - a. 針對資料來源名稱，輸入資料來源的名稱，例如 **myFunction**。
 - b. 針對資料來源類型，選擇 AWS Lambda 函數。
 - c. 針對區域，選擇適當的區域。
 - d. 對於函數 ARN，從下拉式清單中選擇 Lambda 函數。您可以搜尋函數名稱，或手動輸入您要使用的函數 ARN。
 - e. 建立新的 IAM 角色（建議）或選擇具有 IAM `lambda:invokeFunction` 許可的現有角色。現有角色需要信任政策，如[連接資料來源](#)一節所述。

以下是 IAM 政策範例，具有對資源執行操作所需的許可：

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [ "lambda:invokeFunction" ],
      "Resource": [
        "arn:aws:lambda:us-west-2:123456789012:function:myFunction",

```



```

        "arn:aws:lambda:us-
west-2:123456789012:function:myFunction:*"
    ]
}
}

```

3. 選擇建立按鈕。

CLI

1. 執行 [create-data-source](#) 命令來建立資料來源物件。

您需要為此特定命令輸入 4 個參數：

1. 您 API `api-id` 的。
2. 資料來源 `name` 的。在主控台範例中，這是資料來源名稱。
3. 資料來源 `type` 的。在主控台範例中，這是 AWS Lambda 函數。
4. `lambda-config`，這是主控台範例中的函數 ARN。

Note

還有其他參數，例如 必須設定 `Region`，但通常會預設為您的 CLI 組態值。

範例命令可能如下所示：

```

aws appsync create-data-source --api-id abcdefghijklmnopqrstuvwxyz
--name myFunction --type AWS_LAMBDA --lambda-config
lambdaFunctionArn=arn:aws:lambda:us-west-2:102847592837:function:appsync-
lambda-example

```

輸出會在 CLI 中傳回。範例如下：

```

{
  "dataSource": {
    "dataSourceArn": "arn:aws:appsync:us-west-2:102847592837:apis/
abcdefghijklmnopqrstuvwxyz/datasources/myFunction",
    "type": "AWS_LAMBDA",

```

```
    "name": "myFunction",
    "lambdaConfig": {
      "lambdaFunctionArn": "arn:aws:lambda:us-
west-2:102847592837:function:appsync-lambda-example"
    }
  }
}
```

- 若要修改資料來源的屬性，請執行 [update-data-source](#) 命令。

除了 `api-id` 參數之外，`create-data-source` 命令中使用的參數將由 `update-data-source` 命令中的新值覆寫。

啟用直接 Lambda 解析程式

建立 Lambda 資料來源並設定適當的 IAM 角色以允許 AWS AppSync 叫用函數之後，您可以將其連結至解析程式或管道函數。

Console

- 登入 AWS Management Console 並開啟 [AppSync 主控台](#)。
 - 在 APIs 儀表板中，選擇您的 GraphQL API。
 - 在側邊欄中，選擇結構描述。
- 在解析程式視窗中，選擇欄位或操作，然後選取連接按鈕。
- 在建立新的解析程式頁面中，從下拉式清單中選擇 Lambda 函數。
- 為了利用直接 Lambda 解析程式，請確認在設定映射範本區段中已停用請求和回應映射範本。
- 選擇儲存解析程式按鈕。

CLI

- 執行 [create-resolver](#) 命令來建立解析程式。

您需要為此特定命令輸入 6 個參數：

- 您 API `api-id` 的。
- 結構描述中 `type-name` 類型的。
- 結構描述中 欄位 `field-name` 的。

4. `data-source-name` 或您的 Lambda 函數名稱。

5. `request-mapping-template`，這是請求的內文。在主控台範例中，已停用：

```
" "
```

6. `response-mapping-template`，這是回應的內文。在主控台範例中，這也已停用：

```
" "
```

範例命令可能如下所示：

```
aws appsync create-resolver --api-id abcdefghijklmnopqrstuvwxyz --type-name
Subscription --field-name onCreateTodo --data-source-name LambdaTest --request-
mapping-template " " --response-mapping-template " "
```

輸出會在 CLI 中傳回。範例如下：

```
{
  "resolver": {
    "resolverArn": "arn:aws:appsync:us-west-2:102847592837:apis/
abcdefghijklmnopqrstuvwxyz/types/Subscription/resolvers/onCreateTodo",
    "typeName": "Subscription",
    "kind": "UNIT",
    "fieldName": "onCreateTodo",
    "dataSourceName": "LambdaTest"
  }
}
```

當您停用映射範本時，中會發生幾個額外的行為 AWS AppSync：

- 透過停用映射範本，您會向 發出訊號 AWS AppSync，表示您接受 [Direct Lambda 解析程式參考](#) 中指定的預設資料轉譯。
- 透過停用請求映射範本，您的 Lambda 資料來源會收到包含整個[內容](#)物件的承載。
- 透過停用回應映射範本，您的 Lambda 調用結果將根據請求映射範本的版本或請求映射範本是否也停用而翻譯。

測試和偵錯解析程式 in AWS AppSync (VTL)

Note

我們現在主要支援 APPSYNC_JS 執行期及其文件。請考慮[在此處](#)使用 APPSYNC_JS 執行期及其指南。

AWS AppSync 會對資料來源在 GraphQL 欄位上執行解析程式。如[解析程式映射範本概觀](#)中所述，解析程式會使用範本語言與資料來源通訊。這可讓您自訂行為，並在與資料來源通訊前後套用邏輯和條件。如需編寫解析程式的入門教學型程式設計指南，請參閱[解析程式映射範本程式設計指南](#)。

為了協助開發人員寫入、測試和偵錯這些解析程式，AWS AppSync 主控台也提供工具來建立 GraphQL 請求和回應，並將模擬資料縮減至個別欄位解析程式。此外，您可以在 AWS AppSync 主控台中執行查詢、變動和訂閱，並從 Amazon CloudWatch 查看整個請求的詳細日誌串流。其中包括資料來源的結果。

使用模擬資料進行測試

叫用 GraphQL 解析程式時，會包含包含請求相關資訊的 context 物件。其中包括用戶端引數、身分資訊，以及父 GraphQL 欄位的資料。它還包含來自資料來源的結果，可用於回應範本。如需詳細資訊來了解此項架構，以及程式設計時可用的各種協助公用程式，請參閱[解析程式映射範本內容參考](#)。

寫入或編輯解析程式時，您可以將模擬或測試內容物件傳遞至主控台編輯器。這可讓您在沒有實際依據資料來源執行的情況下，了解這兩者如何請求及回應範本評估。例如您可傳送測試 `firstname: Shaggy` 引數，了解該引數在範本程式碼之中使用 `$ctx.args.firstname` 時如何進行評估。您也可以測試任何公用程式協助程式的評估，例如 `$util.autoId()` 或 `util.time.nowISO8601()`。

測試解析程式

此範例將使用 AWS AppSync 主控台來測試解析程式。

1. 登入 AWS Management Console 並開啟 [AppSync 主控台](#)。
 - a. 在 APIs 儀表板中，選擇您的 GraphQL API。
 - b. 在側邊欄中，選擇結構描述。
2. 如果您尚未這麼做，請在 類型下和 欄位旁，選擇連接以新增您的解析程式。

如需如何建置完整解析程式的詳細資訊，請參閱[設定解析程式](#)。

否則，請選取已在 欄位中的解析程式。

3. 在編輯解析程式頁面頂端，選擇選取測試內容，然後選擇建立新內容。
4. 選取範例內容物件，或在下方的執行內容視窗中手動填入 JSON。
5. 在文字內容名稱中輸入 。
6. 選擇 Save (儲存) 按鈕。
7. 在編輯解析程式頁面頂端，選擇執行測試。

如需更實際的範例，假設您有一個應用程式存放的 GraphQL 類型Dog是使用物件的自動 ID 產生，並將其存放在 Amazon DynamoDB 中。您也希望寫入 GraphQL 變動引數的值，並且只讓特定使用者看見回應。結構描述看起來類似如下：

```
type Dog {
  breed: String
  color: String
}

type Mutation {
  addDog(firstname: String, age: Int): Dog
}
```

當您為addDog變動新增解析程式時，您可以填入內容物件，如下列範例所示。下列引數來自 name 及 age，以及將 username 填入 identity 物件之中的用戶端：

```
{
  "arguments" : {
    "firstname": "Shaggy",
    "age": 4
  },
  "source" : {},
  "result" : {
    "breed" : "Miniature Schnauzer",
    "color" : "black_grey"
  },
  "identity": {
    "sub" : "uuid",
    "issuer" : " https://cognito-idp.{region}.amazonaws.com/{userPoolId}",
    "username" : "Nadia",
    "claims" : { },
  }
}
```

```

    "sourceIp" : [ "x.x.x.x" ],
    "defaultAuthStrategy" : "ALLOW"
  }
}
```

您可利用下列要求及回應映射範本對此進行測試：

請求範本

```

{
  "version" : "2017-02-28",
  "operation" : "PutItem",
  "key" : {
    "id" : { "S" : "$util.autoId()" }
  },
  "attributeValues" : $util.dynamodb.toMapValuesJson($ctx.args)
}
```

回應範本

```

#if ($context.identity.username == "Nadia")
  $util.toJson($ctx.result)
#else
  $util.unauthorized()
#end
```

經過評估的範本擁有測試內容物件的資料，以及 `$util.autoId()` 產生的值。此外，若您將 `username` 變更為 Nadia 以外的值，將不會傳回結果，因為授權檢查將會失敗。如需精細存取控制的詳細資訊，請參閱[授權使用案例](#)。

使用 AWS AppSync 的 APIs 測試映射範本

您可以使用 `EvaluateMappingTemplate` API 命令，以模擬資料遠端測試映射範本。若要開始使用命令，請確定您已將 `appsync:evaluateMappingTemplate` 許可新增至政策。例如：

JSON

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
```

```

    "Effect": "Allow",
    "Action": "appsync:evaluateMappingTemplate",
    "Resource": "arn:aws:appsync:<region>:<account>:*"
  }
]
}

```

您可以使用 [AWS CLI](#) 或 [AWS SDKs](#) 來利用 命令。例如，從上一節取得 Dog 結構描述及其請求/回應映射範本。在本機工作站上使用 CLI，將請求範本儲存到名為 `request.vtl` 的檔案，然後將 context 物件儲存到名為 `context.json` 的檔案。從您的 shell 執行下列命令：

```
aws appsync evaluate-mapping-template --template file://request.vtl --context file://context.json
```

命令會傳回下列回應：

```

{
  "evaluationResult": "{\n    \"version\" : \"2017-02-28\",\n    \"operation\" : \"PutItem\",\n    \"key\" : {\n      \"id\" : { \"S\" : \"afcb4c85-49f8-40de-8f2b-248949176456\" }\n    },\n    \"attributeValues\" : {\n      \"firstname\": {\"S\": \"Shaggy\"},\n      \"age\": {\"N\": 4}\n    }\n  }"
}

```

`evaluationResult` 包含使用提供的 測試您提供的範本的結果 context。您也可以使用 AWS SDKs 測試範本。以下是使用適用於 JavaScript V2 的 AWS SDK 的範例：

```

const AWS = require('aws-sdk')
const client = new AWS.AppSync({ region: 'us-east-2' })

const template = fs.readFileSync('./request.vtl', 'utf8')
const context = fs.readFileSync('./context.json', 'utf8')

client
  .evaluateMappingTemplate({ template, context })
  .promise()
  .then((data) => console.log(data))

```

使用 SDK，您可以輕鬆整合來自您最愛測試套件的測試，以驗證範本的行為。我們建議您使用 [Jest 測試架構](#) 建立測試，但任何測試套件都可以運作。下列程式碼片段顯示假設性驗證執行。請注意，我們預期評估回應是有效的 JSON，因此我們使用 `從字串回應JSON.parse` 擷取 JSON：

```
const AWS = require('aws-sdk')
const fs = require('fs')
const client = new AWS.AppSync({ region: 'us-east-2' })

test('request correctly calls DynamoDB', async () => {
  const template = fs.readFileSync('./request.vtl', 'utf8')
  const context = fs.readFileSync('./context.json', 'utf8')
  const contextJSON = JSON.parse(context)

  const response = await client.evaluateMappingTemplate({ template,
    context }).promise()
  const result = JSON.parse(response.evaluationResult)

  expect(result.key.id.S).toBeDefined()
  expect(result.attributeValues.firstname.S).toEqual(contextJSON.arguments.firstname)
})
```

這會產生下列結果：

```
Ran all test suites.
> jest

PASS ./index.test.js
# request correctly calls DynamoDB (543 ms)

Test Suites: 1 passed, 1 total
Tests: 1 passed, 1 total
Snapshots: 0 total
Time: 1.511 s, estimated 2 s
```

偵錯即時查詢

end-to-end測試和記錄無法用來偵錯生產應用程式。AWS AppSync 可讓您使用 Amazon CloudWatch 記錄錯誤和完整請求詳細資訊。此外，您可以使用 AWS AppSync 主控台來測試 GraphQL 查詢、變動和訂閱，以及將每個請求的即時串流日誌資料傳回查詢編輯器以進行即時偵錯。針對訂閱，日誌會顯示連線時間資訊。

若要執行此操作，您需要事先啟用 Amazon CloudWatch logs，如[監控和記錄](#)中所述。接著，在 AWS AppSync 主控台中，選擇查詢索引標籤，然後輸入有效的 GraphQL 查詢。在右下方區段中，按一下並拖曳日誌視窗以開啟日誌檢視。在頁面頂端，選擇執行箭頭圖示來執行您的 GraphQL 查詢。幾分鐘後，操作的完整請求及回應日誌，將串流至此區段，然後您可在主控台中檢視。

設定和使用管道解析程式 in AWS AppSync (VTL)

Note

我們現在主要支援 APPSYNC_JS 執行期及其文件。請考慮[在此處](#)使用 APPSYNC_JS 執行期及其指南。

AWS AppSync 會在 GraphQL 欄位上執行解析程式。在某些情況下，應用程式需要執行多個操作，才能解析單一 GraphQL 欄位。透過管道解析程式，開發人員現在可以編寫稱為 Functions 的操作，並依序執行這些操作。在像是以需要先執行授權檢查才能擷取欄位資料的情況中，就很適合使用管道解析程式。

管道解析程式包含 Before 映射範本和 After 映射範本，以及一份函數清單。每個函數都有一個請求和回應映射範本，它會針對資料來源執行。由於管道解析程式是將執行委派到一份函數清單，所以不會連結到任何資料來源。單位解析程式和函數是對資料來源執行操作的基礎。如需詳細資訊，請參閱[解析程式映射範本概觀](#)。

步驟 1：建立管道解析程式

在 AWS AppSync 主控台中，前往結構描述頁面。

儲存下列結構描述：

```
schema {
  query: Query
  mutation: Mutation
}

type Mutation {
  signUp(input: Signup): User
}

type Query {
  getUser(id: ID!): User
}

input Signup {
  username: String!
  email: String!
}
```

```
type User {  
  id: ID!  
  username: String  
  email: AWSEmail  
}
```

我們將將管道解析程式連接到 Mutation 類型的 signUp 欄位。在右側的變動類型中，選擇signUp變動欄位旁的連接。在建立解析程式頁面上，按一下動作，然後按一下更新執行時間。選擇 Pipeline Resolver，然後選擇 VTL，然後選擇更新。頁面現在應會顯示三個區段：映射前範本文字區域、函數區段，以及映射後範本文字區域。

我們的管道解析程式會註冊使用者，此註冊的第一步是驗證電子郵件地址輸入，然後在系統中儲存此位使用者。我們將在 validateEmail 函數中封裝該電子郵件驗證，並在 saveUser 函數中封裝使用者儲存步驟。validateEmail 函數會先執行，而當電子郵件驗證有效時，saveUser 函數就會接著執行。

執行流程將如下所示：

1. Mutation.signUp 解析程式要求映射範本
2. validateEmail 函數
3. saveUser 函數
4. Mutation.signUp 解析程式回應映射範本

由於我們可能會在 API 上的其他解析程式中重複使用 validateEmail 函數，因此我們希望避免存取，\$ctx.args因為這些函數會從一個 GraphQL 欄位變更為另一個欄位。反之，我們可以使用 \$ctx.stash 來存放 signUp(input: Signup) 輸入欄位引數所傳遞的電子郵件屬性。

映射範本之前：

```
## store email input field into a generic email key  
$util.qr($ctx.stash.put("email", $ctx.args.input.email))  
{}
```

主控台提供預設的 AFTER 映射範本，我們將使用：

```
$util.toJson($ctx.result)
```

選擇建立或儲存以更新解析程式。

步驟 2：建立 函數

在管道解析程式頁面的函數區段中，按一下新增函數，然後按一下建立新函數。您也可以在不經過解析程式頁面的情況下建立函數；若要這樣做，請在 AWS AppSync 主控台中前往函數頁面。選擇 **Create function** (建立函數) 按鈕。讓我們來建立可檢查電子郵件是否有效且來源是特定網域的函數。如果電子郵件無效，則該函數會引發錯誤。否則，它會轉送任何獲予的任何輸入。

在新函數頁面上，選擇動作，然後選擇更新執行時間。選擇 VTL，然後選擇更新。請確定您已建立 **NONE** 類型的資料來源。在資料來源名稱清單中選擇此資料來源。在函數名稱中輸入 `validateEmail`。在函數程式碼區域中，使用此程式碼片段覆寫所有項目：

```
#set($valid = $util.matches("[a-zA-Z0-9_+-.]+@((?:[a-zA-Z0-9-]+\.)?[a-zA-Z]+\.)?(myvaliddomain)\.com", $ctx.stash.email))
#if (!$valid)
    $util.error("$ctx.stash.email is not a valid email.")
#end
{
    "payload": { "email": $util.toJson($ctx.stash.email) }
}
```

將此貼到回應映射範本中：

```
$util.toJson($ctx.result)
```

檢閱您的變更，然後選擇建立。我們已建立我們 `validateEmail` 函數。重複這些步驟，使用下列請求和回應映射範本建立 `saveUser` 函數（為了簡化起見，我們使用 **NONE** 資料來源，並在函數執行後假設使用者已儲存在系統中。）：

要求映射範本：

```
## $ctx.prev.result contains the signup input values. We could have also
## used $ctx.args.input.
{
    "payload": $util.toJson($ctx.prev.result)
}
```

回應映射範本：

```
## an id is required so let's add a unique random identifier to the output
```

```
$util.qr($ctx.result.put("id", $util.autoId()))
$util.toJson($ctx.result)
```

我們剛建立了 `saveUser` 函數。

步驟 3：將函數新增至管道解析程式

我們的函數應該會自動新增至我們剛建立的管道解析程式。如果不是這種情況，或者您透過函數頁面建立函數，您可以在解析程式頁面上按一下新增函數來連接它們。同時將 `validateEmail` 和 `saveUser` 函數新增至解析程式。`validateEmail` 函數應該放在 `saveUser` 函數之前。當您新增更多函數時，您可以使用上下移動選項來重組函數的執行順序。檢閱您的變更，然後選擇儲存。

步驟 4：執行查詢

在 AWS AppSync 主控台中，前往查詢頁面。在瀏覽器中，請確定您使用的是變動。如果不是，請在下拉式清單 `Mutation` 中選擇，然後選擇 `+`。輸入下列查詢：

```
mutation {
  signUp(input: {
    email: "nadia@myvaliddomain.com"
    username: "nadia"
  }) {
    id
    email
  }
}
```

這應該會傳回如下內容：

```
{
  "data": {
    "signUp": {
      "id": "256b6cc2-4694-46f4-a55e-8cb14cc5d7fc",
      "email": "nadia@myvaliddomain.com"
    }
  }
}
```

我們已使用管道解析程式成功註冊我們的使用者，並完成輸入電子郵件的驗證。若要取得更多著重管道解析程式的全面教學課程，您可以移至[教學課程：管道解析程式](#)

搭配 AWS CDK 使用 AWS AppSync API

Tip

在使用 CDK 之前，建議您檢閱 CDK 的[官方文件](https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws_appsync-readme.html)以及 CDK AWS AppSync 參考。https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws_appsync-readme.html
我們也建議確保您的 [AWS CLI](#) 和 [NPM](#) 安裝可在您的系統上運作。

在本節中，我們將建立簡單的 CDK 應用程式，可從 DynamoDB 資料表新增和擷取項目。這是使用[設計結構描述](#)、[連接資料來源](#)和[設定解析程式 \(JavaScript\)](#) 區段中的一些程式碼的快速入門範例。

設定 CDK 專案

Warning

這些步驟可能不是完全準確的，具體取決於您的環境。我們假設您的系統已安裝必要的公用程式、與服務的界面方式 AWS，以及適當的組態。

第一步是安裝 AWS CDK。在 CLI 中，您可以輸入下列命令：

```
npm install -g aws-cdk
```

接著，您需要建立專案目錄，然後導覽至該目錄。建立和導覽至目錄的一組命令範例如下：

```
mkdir example-cdk-app  
cd example-cdk-app
```

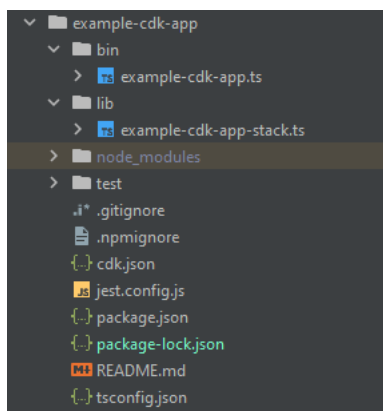
接著，您需要建立應用程式。我們的服務主要使用 TypeScript。在專案目錄中，輸入下列命令：

```
cdk init app --language typescript
```

當您執行此操作時，將會安裝 CDK 應用程式及其初始化檔案：

```
Initializing a new git repository...
hint: Using 'master' as the name for the initial branch. This default branch name
hint: is subject to change. To configure the initial branch name to use in all
hint: of your new repositories, which will suppress this warning, call:
hint:
hint:   git config --global init.defaultBranch <name>
hint:
hint: Names commonly chosen instead of 'master' are 'main', 'trunk' and
hint: 'development'. The just-created branch can be renamed via this command:
hint:
hint:   git branch -m <name>
Executing npm install...
✔ All done!
```

您的專案結構可能如下所示：



您會注意到我們有一些重要的目錄：

- **bin**：初始 bin 檔案將建立應用程式。我們不會在本指南中接觸此項目。
- **lib**：lib 目錄包含您的堆疊檔案。您可以將堆疊檔案視為個別的執行單位。建構將位於我們的堆疊檔案中。基本上，這些是服務的資源，會在應用程式部署 AWS CloudFormation 時於其中啟動。這就是我們大部分編碼都會發生的地方。
- **node_modules**：此目錄由 NPM 建立，並包含您使用 npm 命令安裝的所有套件相依性。

我們的初始堆疊檔案可能包含如下內容：

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
// import * as sqs from 'aws-cdk-lib/aws-sqs';

export class ExampleCdkAppStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);
```

```
// The code that defines your stack goes here

// example resource
// const queue = new sqs.Queue(this, 'ExampleCdkAppQueue', {
//   visibilityTimeout: cdk.Duration.seconds(300)
// });
}
```

這是在我們的應用程式中建立堆疊的樣板程式碼。我們在此範例中的大部分程式碼都會進入此類別的範圍內。

若要驗證您的堆疊檔案是否在應用程式中，請在應用程式的目錄中，在終端機中執行下列命令：

```
cdk ls
```

堆疊清單應該會出現。如果沒有，則您可能需要再次執行這些步驟，或查看官方文件以取得協助。

如果您想要在部署之前建置程式碼變更，您可以隨時在終端機中執行下列命令：

```
npm run build
```

此外，若要在部署之前查看變更：

```
cdk diff
```

將程式碼新增至堆疊檔案之前，我們會執行引導。引導允許我們在應用程式部署之前佈建 CDK 的資源。如需此程序的詳細資訊，請參閱[此處](#)。若要建立引導，命令為：

```
cdk bootstrap aws://ACCOUNT-NUMBER/REGION
```

Tip

此步驟需要您帳戶中的數個 IAM 許可。如果您沒有引導，您的引導會被拒絕。如果發生這種情況，您可能必須刪除由引導造成的不完整資源，例如它產生的 S3 儲存貯體。

Bootstrap 將啟動數個資源。最終訊息看起來會如下所示：

```
❗ Bootstrapping environment [redacted]
Trusted accounts for deployment: (none)
Trusted accounts for lookup: (none)
Using default execution policy of 'arn:aws:iam::aws:policy/AdministratorAccess'. Pass '--cloudformation-execution-policies' to customize.
CDKToolkit: creating CloudFormation changeset...
✅ Environment [redacted] bootstrapped.
```

每個區域的每個帳戶都會完成一次，因此您不必經常這麼做。引導的主要資源是 AWS CloudFormation 堆疊和 Amazon S3 儲存貯體。

Amazon S3 儲存貯體用於存放授予執行部署所需許可的檔案和 IAM 角色。所需的資源是在 AWS CloudFormation 堆疊中定義，稱為引導堆疊，通常命名為 CDKToolkit。與任何 AWS CloudFormation 堆疊一樣，它在部署後會出現在 AWS CloudFormation 主控台中：

Stacks (10)

Filter by stack name Filter status Active View nested

	Stack name	Status	Created time	Description
☐	CDKToolkit	CREATE_COMPLETE	2023-07-30 21:20:19 UTC-0700	This stack includes resources needed to deploy AWS CDK apps into this environment

對於儲存貯體，也可以說出相同的：

Name	AWS Region	Access	Creation date
cdk-l[redacted]-assets-[redacted]-us-west-2	US West (Oregon) us-west-2	Bucket and objects not public	July 30, 2023, 21:20:29 (UTC-07:00)

若要匯入堆疊檔案中所需的服務，我們可以使用以下命令：

```
npm install aws-cdk-lib # V2 command
```

Tip

如果 V2 發生問題，您可以使用 V1 命令安裝個別程式庫：

```
npm install @aws-cdk/aws-appsync @aws-cdk/aws-dynamodb
```

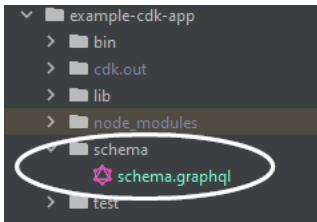
我們不建議這麼做，因為 V1 已棄用。

實作 CDK 專案 - 結構描述

我們現在可以開始實作程式碼。首先，我們必須建立結構描述。您可以直接在應用程式中建立 `.graphql` 檔案：


```
mkdir schema
touch schema.graphql
```

在我們的範例中，我們包含了一個名為 `schema` 的最上層目錄，`schema` 其中包含我們的 `schema.graphql`：



在我們的結構描述中，讓我們包含一個簡單的範例：

```
input CreatePostInput {
  title: String!
  content: String!
}

type Post {
  id: ID!
  title: String!
  content: String!
}

type Mutation {
  createPost(input: CreatePostInput!): Post!
}

type Query {
  getPost(id: ID!): Post!
}
```

回到堆疊檔案中，我們需要確保定義了下列匯入指令：

```
import * as cdk from 'aws-cdk-lib';
import * as appsync from 'aws-cdk-lib/aws-appsync';
import * as dynamodb from 'aws-cdk-lib/aws-dynamodb';
import { Construct } from 'constructs';
```

在類別中，我們將新增程式碼以建立 GraphQL API，並將其連接到我們的 `schema.graphql` 檔案：

```
export class ExampleCdkAppStack extends cdk.Stack {
```

```

constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // makes a GraphQL API
    const api = new appsync.GraphqlApi(this, 'post-apis', {
        name: 'api-to-process-posts',
        schema: appsync.SchemaFile.fromAsset('schema/schema.graphql'),
    });
}
}

```

我們也會新增一些程式碼來列印 GraphQL URL、API 金鑰和區域：

```

export class ExampleCdkAppStack extends cdk.Stack {
    constructor(scope: Construct, id: string, props?: cdk.StackProps) {
        super(scope, id, props);

        // Makes a GraphQL API construct
        const api = new appsync.GraphqlApi(this, 'post-apis', {
            name: 'api-to-process-posts',
            schema: appsync.SchemaFile.fromAsset('schema/schema.graphql'),
        });

        // Prints out URL
        new cdk.CfnOutput(this, "GraphQLAPIURL", {
            value: api.graphqlUrl
        });

        // Prints out the AppSync GraphQL API key to the terminal
        new cdk.CfnOutput(this, "GraphQLAPIKey", {
            value: api.apiKey || ''
        });

        // Prints out the stack region to the terminal
        new cdk.CfnOutput(this, "Stack Region", {
            value: this.region
        });
    }
}

```

此時，我們將再次使用部署應用程式：

```
cdk deploy
```

這是結果：

```
ExampleCdkAppStack: deploying... [1/1]
ExampleCdkAppStack: creating CloudFormation changeset...

✅ ExampleCdkAppStack

🌟 Deployment time: 16.13s

Outputs:
ExampleCdkAppStack.GraphQLAPIKey = [REDACTED]
ExampleCdkAppStack.GraphQLAPIURL = https://[REDACTED]/graphql
ExampleCdkAppStack.StackRegion = us-west-2
Stack ARN:
arn:aws:cloudformation:[REDACTED]

🌟 Total time: 22s
```

顯示我們的範例成功，但讓我們檢查 AWS AppSync 主控台以確認：



顯示我們的 API 已建立。現在，我們將檢查連接到 API 的結構描述：

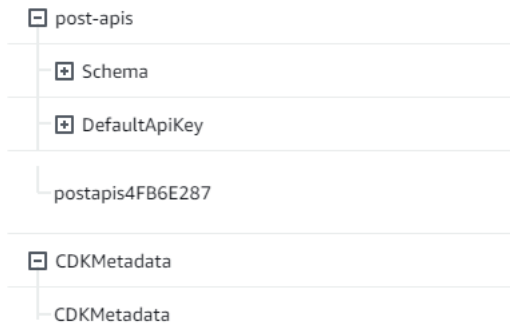
```
Schema

1 input CreatePostInput {
2   title: String
3   date: AWSDateTime
4 }
5
6 type Post {
7   id: ID!
8   title: String
9   date: AWSDateTime
10 }
11
12 type Mutation {
13   createPost(input: CreatePostInput!): Post
14 }
15
16 type Query {
17   getPost: [Post]
18 }
```

這似乎符合我們的結構描述程式碼，因此成功。從中繼資料觀點確認這一點的另一種方法是查看 AWS CloudFormation 堆疊：

○ ExampleCdkAppStack	🟢 UPDATE_COMPLETE	2023-07-30 22:13:31 UTC-0700	-
○ CDKToolkit	🟢 CREATE_COMPLETE	2023-07-30 21:20:19 UTC-0700	This stack includes resources needed to deploy AWS CDK apps into this environment

當我們部署 CDK 應用程式時，它會逐步 AWS CloudFormation 啟動引導等資源。我們應用程式內的每個堆疊都會以 1 : 1 與 AWS CloudFormation 堆疊進行映射。如果您返回堆疊程式碼，堆疊名稱會從類別名稱中擷取 ExampleCdkAppStack。您可以在 GraphQL API 建構中查看其建立的資源，這也符合我們的命名慣例：



實作 CDK 專案 - 資料來源

接下來，我們需要新增資料來源。我們的範例將使用 DynamoDB 資料表。在堆疊類別中，我們會新增一些程式碼來建立新的資料表：

```

export class ExampleCdkAppStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // Makes a GraphQL API construct
    const api = new appsync.GraphqlApi(this, 'post-apis', {
      name: 'api-to-process-posts',
      schema: appsync.SchemaFile.fromAsset('schema/schema.graphql'),
    });

    //creates a DDB table
    const add_ddb_table = new dynamodb.Table(this, 'posts-table', {
      partitionKey: {
        name: 'id',
        type: dynamodb.AttributeType.STRING,
      },
    });

    // Prints out URL
    new cdk.CfnOutput(this, "GraphQLAPIURL", {

```

```

        value: api.graphqlUrl
    });

    // Prints out the AppSync GraphQL API key to the terminal
    new cdk.CfnOutput(this, "GraphQLAPIKey", {
        value: api.apiKey || ''
    });

    // Prints out the stack region to the terminal
    new cdk.CfnOutput(this, "Stack Region", {
        value: this.region
    });
}
}

```

此時，讓我們再次部署：

```
cdk deploy
```

我們應該檢查 DynamoDB 主控台是否有新的資料表：

☐	ExampleCdkAppStack-poststable	Active	id (S)	-	0		Provisioned (5)	Provisioned (5)	0 bytes	Standard

我們的堆疊名稱正確，且資料表名稱符合我們的程式碼。如果我們再次檢查 AWS CloudFormation 堆疊，現在將會看到新的資料表：

Logical ID
☒ post-apis
☒ posts-table
poststable6CB5A2E6
☒ CDKMetadata

實作 CDK 專案 - Resolver

此範例將使用兩個解析程式：一個用於查詢資料表，另一個用於新增資料表。由於我們使用管道解析程式，因此我們需要宣告兩個管道解析程式，每個都具有一個函數。在查詢中，我們將新增下列程式碼：

```

export class ExampleCdkAppStack extends cdk.Stack {
    constructor(scope: Construct, id: string, props?: cdk.StackProps) {
        super(scope, id, props);
    }
}

```

```
// Makes a GraphQL API construct
const api = new appsync.GraphqlApi(this, 'post-apis', {
  name: 'api-to-process-posts',
  schema: appsync.SchemaFile.fromAsset('schema/schema.graphql'),
});

//creates a DDB table
const add_ddb_table = new dynamodb.Table(this, 'posts-table', {
  partitionKey: {
    name: 'id',
    type: dynamodb.AttributeType.STRING,
  },
});

// Creates a function for query
const add_func = new appsync.AppsyncFunction(this, 'func-get-post', {
  name: 'get_posts_func_1',
  api,
  dataSource: api.addDynamoDbDataSource('table-for-posts', add_ddb_table),
  code: appsync.Code.fromInline(`
    export function request(ctx) {
      return { operation: 'Scan' };
    }

    export function response(ctx) {
      return ctx.result.items;
    }
  `),
  runtime: appsync.FunctionRuntime.JS_1_0_0,
});

// Creates a function for mutation
const add_func_2 = new appsync.AppsyncFunction(this, 'func-add-post', {
  name: 'add_posts_func_1',
  api,
  dataSource: api.addDynamoDbDataSource('table-for-posts-2', add_ddb_table),
  code: appsync.Code.fromInline(`
    export function request(ctx) {
      return {
        operation: 'PutItem',
        key: util.dynamodb.toMapValues({id: util.autoId()}),
        attributeValues: util.dynamodb.toMapValues(ctx.args.input),
      };
    }
  `),
});
```

```

        export function response(ctx) {
            return ctx.result;
        }
    `),
    runtime: appsync.FunctionRuntime.JS_1_0_0,
});

// Adds a pipeline resolver with the get function
new appsync.Resolver(this, 'pipeline-resolver-get-posts', {
    api,
    typeName: 'Query',
    fieldName: 'getPost',
    code: appsync.Code.fromInline(`
        export function request(ctx) {
            return {};
        }

        export function response(ctx) {
            return ctx.prev.result;
        }
    `),
    runtime: appsync.FunctionRuntime.JS_1_0_0,
    pipelineConfig: [add_func],
});

// Adds a pipeline resolver with the create function
new appsync.Resolver(this, 'pipeline-resolver-create-posts', {
    api,
    typeName: 'Mutation',
    fieldName: 'createPost',
    code: appsync.Code.fromInline(`
        export function request(ctx) {
            return {};
        }

        export function response(ctx) {
            return ctx.prev.result;
        }
    `),
    runtime: appsync.FunctionRuntime.JS_1_0_0,
    pipelineConfig: [add_func_2],
});

```

```
// Prints out URL
new cdk.CfnOutput(this, "GraphQLAPIURL", {
  value: api.graphqlUrl
});

// Prints out the AppSync GraphQL API key to the terminal
new cdk.CfnOutput(this, "GraphQLAPIKey", {
  value: api.apiKey || ''
});

// Prints out the stack region to the terminal
new cdk.CfnOutput(this, "Stack Region", {
  value: this.region
});
}
}
```

在此程式碼片段中，我們新增了一個名為的管道解析程式，`pipeline-resolver-create-posts`並`func-add-post`附加了一個名為的函數。這是將Posts新增至資料表的程式碼。另一個管道解析程式使用名為的函數`pipeline-resolver-get-posts`呼叫`func-get-post`，該函數會擷取Posts新增至資料表的。

我們將部署此項目，以將其新增至 AWS AppSync 服務：

```
cdk deploy
```

讓我們檢查 AWS AppSync 主控台，查看它們是否已連接到我們的 GraphQL API：

Mutation	
Field	Resolver
<code>createPost(...): Post</code>	 Pipeline
Query	
Field	Resolver
<code>getPost: [Post]</code>	 Pipeline

它看起來是正確的。在程式碼中，這兩個解析程式都連接到我們製作的 GraphQL API（由解析程式和函數中存在的 `props api` 值表示）。在 GraphQL API 中，我們附加解析程式的欄位也指定在 `props` 中（由每個解析程式中的 `typename` 和 `fieldname` props 定義）。

讓我們看看解析程式的內容從開始是否正確 `pipeline-resolver-get-posts`：

The screenshot displays the AWS AppSync console interface. The top section, titled 'Resolver code', shows the JavaScript code for the 'request' and 'response' functions. The 'request' function is empty, while the 'response' function returns `ctx.prev.result`. A blue arrow points to the `response` function. Below the code editor, the status bar indicates 'APPSYNC_JS Ln 1, Col 1' with zero errors and warnings. The bottom section, titled 'Functions', provides a description of functions and a search bar. A list of functions is shown, with 'add_posts_func_1' highlighted. A blue arrow points to this function. Below the function name, there is a description field (currently empty) and a 'Function code' section marked as 'read-only'.

```
1  
2 export function request(ctx) {  
3   return {};  
4 }  
5  
6 export function response(ctx) {  
7   return ctx.prev.result;  
8 }  
9
```

APPSYNC_JS Ln 1, Col 1 Errors: 0 Warnings: 0

Functions

Each function is executed in sequence and can execute a single operation against a data source.

add_posts_func_1 [Edit](#)
Description
-
▶ **Function code** read-only

前後處理常式符合我們的 `code` 道具值。我們也可以看到名為 `add_posts_func_1` 的函數，其符合我們在解析程式中連接的函數名稱。

讓我們看看該函數的程式碼內容：

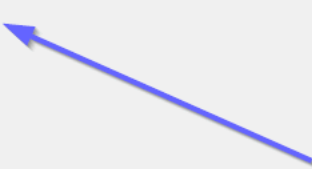
add_posts_func_1 Edit

Description

-

▼ **Function code** read-only

```
1
2   export function request(ctx) {
3     return {
4       operation: 'PutItem',
5       key: util.dynamodb.toMapValues({id: util.autoId()}),
6       attributeValues: util.dynamodb.toMapValues(ctx.args.input),
7     };
8   }
9
10  export function response(ctx) {
11    return ctx.result;
12  }
13
```



這符合add_posts_func_1函數的code道具。我們的查詢已成功上傳，因此我們檢查查詢：

▼ Resolver code

```
1
2 export function request(ctx) {
3   return {};
4 }
5
6 export function response(ctx) {
7   return ctx.prev.result;
8 }
9
```

APPSYNC_JS Ln 1, Col 1  Errors: 0  Warnings: 0

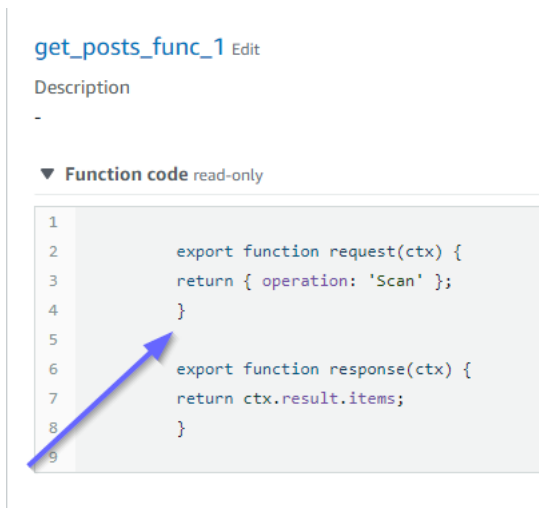
Functions
Each function is executed in sequence and can execute a single operation against a data source.

[get_posts_func_1](#) Edit

Description
-

► Function code read-only

這些也符合程式碼。如果我們查看 `get_posts_func_1`：



一切似乎都就位。為了從中繼資料的角度進行確認，我們可以再次檢查 中的 AWS CloudFormation 堆疊：

Logical ID
☐ post-apis
☐ posts-table
☐ func-get-post
☐ func-add-post
☐ pipeline-resolver-get-posts
☐ pipeline-resolver-create-posts
☐ CDKMetadata

現在，我們需要執行一些請求來測試此程式碼。

實作 CDK 專案 - 請求

為了在 AWS AppSync 主控台中測試我們的應用程式，我們提出了一個查詢和一個變動：

```

1 query MyQuery {
2   getPost {
3     id
4     date
5     title
6   }
7 }
8
9 mutation MyMutation {
10  createPost(input: {date: "1970-01-01T12:30:00.000Z", title: "first post"}) {
11    date
12    id
13    title
14  }
15 }
16

```

MyMutation 包含具有引數 1970-01-01T12:30:00.000Z 和 createPost 的操作 first post。它會傳回我們傳入 date 和 title 的，以及自動產生的 id 值。執行變動會產生結果：

```

{
  "data": {
    "createPost": {
      "date": "1970-01-01T12:30:00.000Z",
      "id": "4dc1c2dd-0aa3-4055-9eca-7c140062ada2",
      "title": "first post"
    }
  }
}

```

如果我們快速檢查 DynamoDB 資料表，我們可以在掃描時看到資料表中的項目：

☐	id (String)	date	title
☐	9f62c4dd-49d5-48d5-b835-143284c72fe0	1970-01-01T12:30:00.000Z	first post

返回 AWS AppSync 主控台，如果我們執行查詢來擷取此 Post，我們會得到下列結果：

```

{
  "data": {
    "getPost": [
      {
        "id": "9f62c4dd-49d5-48d5-b835-143284c72fe0",
        "date": "1970-01-01T12:30:00.000Z",
        "title": "first post"
      }
    ]
  }
}

```

```
}  
}
```

在 中使用即時資料應用程式的訂閱 AWS AppSync

Important

自 2025 年 3 月 13 日起，您可以使用 AWS AppSync Events 建置由 WebSockets 支援的即時 PubSub API。如需詳細資訊，請參閱 AppSync Events 開發人員指南中的[透過 WebSocket 發佈事件](#)。AWS AppSync

AWS AppSync 可讓您利用訂閱來實作即時應用程式更新、推送通知等。當用戶端叫用 GraphQL 訂閱操作時，AWS AppSync 會自動建立和維護安全的 WebSocket 連線。應用程式接著可以從資料來源即時將資料分發給訂閱者，同時 AWS AppSync 會持續管理應用程式的連線和擴展需求。以下各節將向您展示 in AWS AppSync 訂閱的運作方式。

GraphQL 結構描述訂閱指令

訂閱 in AWS AppSync 被調用為對變動的回應。這表示您可以在變動上指定 GraphQL 結構描述指令，讓任何資料來源 in AWS AppSync 即時。

AWS Amplify 用戶端程式庫會自動處理訂閱連線管理。這些程式庫使用純 WebSockets 做為用戶端和服務之間的網路通訊協定。

Note

若要控制訂閱的連線時間授權，您可以使用 AWS Identity and Access Management (IAM) AWS Lambda、Amazon Cognito 身分集區或 Amazon Cognito 使用者集區進行欄位層級授權。對於訂閱的精細存取控制，您可以將解析程式連接至訂閱欄位，並使用發起人身分和 AWS AppSync 資料來源執行邏輯。如需詳細資訊，請參閱[設定授權和身分驗證來保護 GraphQL APIs](#)。

會從變動觸發訂閱且會將變動選擇設定傳送給訂閱者。

以下範例示範如何使用 GraphQL 訂閱。它不會指定資料來源，因為資料來源可以是 Lambda、Amazon DynamoDB 或 Amazon OpenSearch Service。

若要開始使用訂閱，您必須將訂閱進入點新增至結構描述，如下所示：

```
schema {  
  query: Query  
  mutation: Mutation  
  subscription: Subscription  
}
```

假設您有一個部落格文章網站，以及您想要訂閱新部落格和變更現有的部落格。若要執行此操作，請將以下 Subscription 定義新增至結構描述：

```
type Subscription {  
  addedPost: Post  
  updatedPost: Post  
  deletedPost: Post  
}
```

假設您有以下變動：

```
type Mutation {  
  addPost(id: ID! author: String! title: String content: String url: String): Post!  
  updatePost(id: ID! author: String! title: String content: String url: String ups:  
    Int! downs: Int! expectedVersion: Int!): Post!  
  deletePost(id: ID!): Post!  
}
```

您可以透過為您希望接收通知的每個訂閱新增 @aws_subscribe(mutations: ["mutation_field_1", "mutation_field_2"]) 指示 (如下所示) 讓這些欄位變得即時：

```
type Subscription {  
  addedPost: Post  
  @aws_subscribe(mutations: ["addPost"])  
  updatedPost: Post  
  @aws_subscribe(mutations: ["updatePost"])  
  deletedPost: Post  
  @aws_subscribe(mutations: ["deletePost"])  
}
```

由於 @aws_subscribe(mutations: ["", ..., ""]) 採用一系列的變動輸入，因此您可以指定多個變動來啟動訂閱。如果您是從用戶端進行訂閱，您的 GraphQL 查詢可能如下所示：

```
subscription NewPostSub {
  addedPost {
    __typename
    version
    title
    content
    author
    url
  }
}
```

用戶端連線和工具需要此訂閱查詢。

使用純 WebSockets 用戶端時，每個用戶端都會完成選取集篩選，因為每個用戶端都可以定義自己的選取集。在此情況下，訂閱選項集必須是變動選項集的子集。例如，連結至變動 `addPost(...){id author title url version}` 的訂閱 `addedPost{author title}` 僅接收文章的作者和標題。而沒有接收其他欄位。但是，如果變動缺少其選項集中的作者，則訂閱者將取得作者欄位的 `null` 值 (或者如果作者欄位在結構描述中被定義為必要/非 `null`，則會出現錯誤)。

使用純 WebSockets 時，訂閱選擇集至關重要。如果訂閱中未明確定義欄位，則 AWS AppSync 不會傳回 欄位。

在上述範例中，訂閱沒有引數。假設您的結構描述如下所示：

```
type Subscription {
  updatedPost(id:ID! author:String): Post
  @aws_subscribe(mutations: ["updatePost"])
}
```

在此情況下，您的用戶端定義訂閱如下：

```
subscription UpdatedPostSub {
  updatedPost(id:"XYZ", author:"ABC") {
    title
    content
  }
}
```

在結構描述 `subscription` 欄位的傳回類型必須與對應變動欄位的傳回類型相符。在上述範例中，這是以 `addPost` 類型形式傳回做為 `addedPost` 和 `Post` 而顯示。

若要在用戶端上設定訂閱，請參閱 [使用 Amplify 用戶端建置用戶端應用程式](#)。

使用訂閱引數

使用 GraphQL 訂閱的重要部分是了解何時及如何使用引數。您可以進行細微的變更，以修改通知用戶端發生變動的方式和時間。若要執行此作業，請參閱快速入門章節中建立 "Todos" 的範例結構描述。對於此範例結構描述，定義了下列變動：

```
type Mutation {  
  createTodo(input: CreateTodoInput!): Todo  
  updateTodo(input: UpdateTodoInput!): Todo  
  deleteTodo(input: DeleteTodoInput!): Todo  
}
```

在預設範例中，用戶端可以使用 `TodoonUpdateTodosubscription` 沒有引數的 訂閱任何 的更新：

```
subscription OnUpdateTodo {  
  onUpdateTodo {  
    description  
    id  
    name  
    when  
  }  
}
```

您可以使用 引數 `subscription` 來篩選您的。例如，若要只在更新 `todo` 具有特定 ID 的 `subscription` 時觸發，請指定 ID 值：

```
subscription OnUpdateTodo {  
  onUpdateTodo(id: "a-todo-id") {  
    description  
    id  
    name  
    when  
  }  
}
```

您也可以傳遞多個引數。例如，以下 `subscription` 示範如何在特定位置和時間收到任何 `Todo` 更新的通知：

```
subscription todosAtHome {
```

```
onUpdateTodo(when: "tomorrow", where: "at home") {  
  description  
  id  
  name  
  when  
  where  
}  
}
```

請注意，所有引數都是選用的。如果您未在 `where` 中指定任何引數 `subscription`，則會訂閱應用程式中發生的所有 `Todo` 更新。不過，您可以更新 `subscription` 的欄位定義，以要求 `ID` 引數。這將強制特定的回應，`todo` 而不是所有 `todo`：

```
onUpdateTodo(  
  id: ID!,  
  name: String,  
  when: String,  
  where: String,  
  description: String  
): Todo
```

引數 Null 值具有意義

在 AWS AppSync 中進行訂閱查詢時，`null` 引數值會篩選結果，與完全省略引數不同。

讓我們回到可建立待辦事項的待辦事項 API 範例。請參閱快速入門章節中的範例結構描述。

讓我們修改結構描述，在 `Todo` 類型中包含描述擁有者身分的新 `owner` 欄位。欄位不是必要 `owner` 欄位，只能在上設定 `UpdateTodoInput`。請參閱下列簡化版本的結構描述：

```
type Todo {  
  id: ID!  
  name: String!  
  when: String!  
  where: String!  
  description: String!  
  owner: String  
}  
  
input CreateTodoInput {  
  name: String!  
  when: String!
```

```

    where: String!
    description: String!
  }

input UpdateTodoInput {
  id: ID!
  name: String
  when: String
  where: String
  description: String
  owner: String
}

type Subscription {
  onUpdateTodo(
    id: ID,
    name: String,
    when: String,
    where: String,
    description: String
  ): Todo @aws_subscribe(mutations: ["updateTodo"])
}

```

下列訂閱會傳回所有Todo更新：

```

subscription MySubscription {
  onUpdateTodo {
    description
    id
    name
    when
    where
  }
}

```

如果您修改上述訂閱以新增欄位引數 `owner: null`，您現在會詢問不同的問題。此訂閱現在會註冊用戶端，以收到尚未提供擁有者的所有Todo更新通知。

```

subscription MySubscription {
  onUpdateTodo(owner: null) {
    description
    id
    name
  }
}

```

```
    when
    where
  }
}
```

Note

自 2022 年 1 月 1 日起，MQTT over WebSockets 不再提供做為 GraphQL 訂閱 AWS AppSync APIs 通訊協定。Pure WebSockets 是中唯一支援的通訊協定 AWS AppSync。以 AWS AppSync SDK 或 Amplify 程式庫為基礎的用戶端，在 2019 年 11 月之後發行，預設會自動使用純 WebSockets 將用戶端升級至最新版本可讓用戶端使用 AWS AppSync 純 WebSockets 引擎。

Pure WebSockets 隨附更大的承載大小 (240 KB)、更廣泛的用戶端選項，以及改善的 CloudWatch 指標。如需使用純 WebSocket 用戶端的詳細資訊，請參閱 [建置即時 WebSocket 用戶端 in AWS AppSync](#)。

在中建立由無伺服器 WebSockets 提供支援的一般 pub/sub APIs AWS AppSync

Important

自 2025 年 3 月 13 日起，您可以使用 AWS AppSync Events 建置由 WebSockets 支援的即時 PubSub API。如需詳細資訊，請參閱《AppSync Events 開發人員指南》中的 [透過 WebSocket 發佈事件](#)。AWS AppSync

有些應用程式只需要簡單的 WebSocket APIs 其中用戶端會接聽特定的頻道或主題。沒有特定形狀或強烈類型需求的一般 JSON 資料可以推送到以純且簡單的發佈訂閱 (pub/sub) 模式收聽這些頻道之一的用戶端。

Use AWS AppSync 透過在 APIs 後端和用戶端上自動產生 GraphQL 程式碼，在幾分鐘內實作幾乎沒有 GraphQL 知識的簡單 pub/sub WebSocket API。GraphQL

建立和設定 pub-sub APIs

若要開始使用，請執行下列動作：

1. 登入 AWS Management Console 並開啟 [AppSync 主控台](#)。

- 在儀表板上，選擇 Create API (建立 API)。
2. 在下一個畫面中，選擇建立即時 API，然後選擇下一步。
 3. 為您的 pub/sub API 輸入易記的名稱。
 4. 您可以啟用[私有 API](#) 功能，但我們建議您暫時將其關閉。選擇下一步。
 5. 您可以選擇使用 WebSockets 自動產生運作中的 pub/sub API。目前也建議關閉此功能。選擇下一步。
 6. 選擇建立 API，然後等待幾分鐘。將在您的帳戶中建立新的預先設定 AWS AppSync pub/sub API AWS。

API 使用 AWS AppSync 的內建本機解析程式（如需使用本機解析程式的詳細資訊，請參閱[AWS AppSync 開發人員指南](#)》中的[教學課程：本機解析程式](#)）來管理多個臨時 pub/sub 通道和 WebSocket 連線，這些連線只會根據頻道名稱自動將資料交付和篩選至訂閱的用戶端。使用 API 金鑰授權 API 呼叫。

部署 API 之後，您會收到幾個額外的步驟來產生用戶端程式碼，並將其與您的用戶端應用程式整合。如需如何快速整合用戶端的範例，本指南將使用簡單的 React Web 應用程式。

1. 首先在本機電腦上使用 [NPM](#) 建立樣板 React 應用程式：

```
$ npx create-react-app mypubsub-app
$ cd mypubsub-app
```

Note

此範例使用 [Amplify 程式庫](#) 將用戶端連線至後端 API。不過，您不需要在本機建立 Amplify CLI 專案。雖然 React 是此範例中選擇的用戶端，但 Amplify 程式庫也支援 iOS、Android 和 Flutter 用戶端，在這些不同的執行時間中提供相同的功能。支援的 Amplify 用戶端提供簡單的抽象概念，以幾行程式碼與 AWS AppSync GraphQL API 後端互動，包括內建 WebSocket 功能，完全相容於 [AWS AppSync 即時 WebSocket 通訊協定](#)：

```
$ npm install @aws-amplify/api
```

2. 在 AWS AppSync 主控台中，選取 JavaScript，然後選取下載以下載單一檔案，其中包含 API 組態詳細資訊和產生的 GraphQL 操作程式碼。
3. 將下載的檔案複製到 React 專案中的 /src 資料夾。

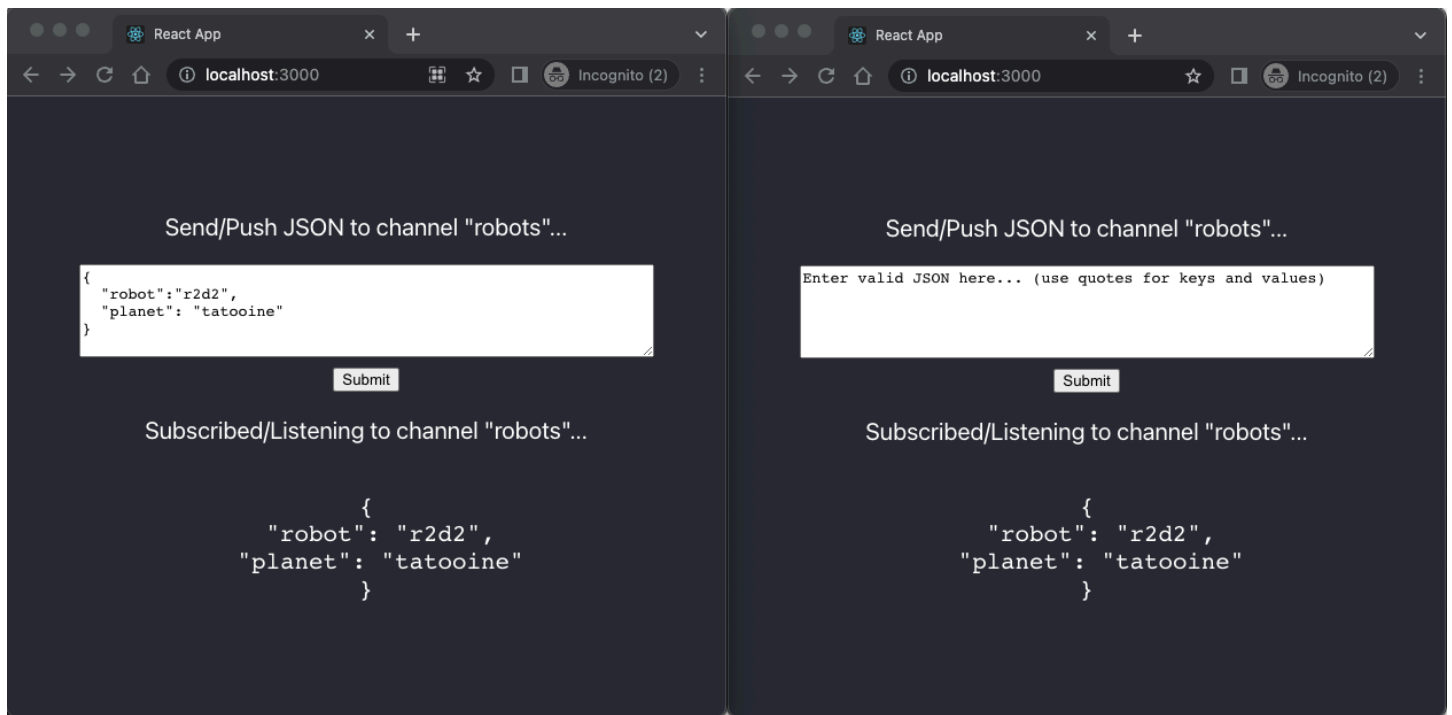
- 接著，將現有樣板src/App.js檔案的內容取代為 主控台中可用的範例用戶端程式碼。
- 使用下列命令在本機啟動應用程式：

```
$ npm start
```

- 若要測試傳送和接收即時資料，請開啟兩個瀏覽器視窗並存取 `localhost#3000`。範例應用程式設定為將一般 JSON 資料傳送至名為####硬式編碼頻道。
- 在其中一個瀏覽器視窗中，在文字方塊中輸入下列 JSON Blob，然後按一下提交：

```
{
  "robot": "r2d2",
  "planet": "tatooine"
}
```

這兩個瀏覽器執行個體都會訂閱####頻道，並即時接收發佈的資料，顯示在 Web 應用程式底部：



系統會自動產生所有必要的 GraphQL API 程式碼，包括結構描述、解析程式和操作，以啟用一般 pub/sub 使用案例。在後端，資料會發佈至具有 GraphQL 變動的 AWS AppSync 即時端點，如下所示：

```
mutation PublishData {
  publish(data: "{\"msg\": \"hello world!\"}", name: "channel") {
    data
  }
}
```

```
    name
  }
}
```

訂閱者會使用相關的 GraphQL 訂閱存取傳送至特定臨時頻道的已發佈資料：

```
subscription SubscribeToData {
  subscribe(name:"channel") {
    name
    data
  }
}
```

在現有應用程式中實作 pub-sub APIs

如果您只需要在現有應用程式中實作即時功能，這個一般 pub/sub API 組態可以輕鬆整合到任何應用程式或 API 技術。雖然使用單一 API 端點與 GraphQL 在單一網路呼叫中安全地存取、操作和合併來自一或多個資料來源的資料，但不需要從頭轉換或重建現有的 REST 型應用程式，以利用 AWS AppSync 的即時功能。例如，您可以在單獨的 API 端點中擁有現有的 CRUD 工作負載，用戶端只會將訊息或事件從現有應用程式傳送到一般 pub/sub API，以用於即時和 pub/sub 目的。

在 中定義增強型訂閱篩選條件 AWS AppSync

Important

自 2025 年 3 月 13 日起，您可以使用 AWS AppSync Events 建置由 WebSockets 支援的即時 PubSub API。如需詳細資訊，請參閱《AppSync Events 開發人員指南》中的[透過 WebSocket 發佈事件](#)。AWS AppSync

在 中 AWS AppSync，您可以使用支援其他邏輯運算子的篩選條件，直接在 GraphQL API 訂閱解析程式中定義和啟用後端資料篩選的商業邏輯。您可以設定這些篩選條件，與用戶端中訂閱查詢上定義的訂閱引數不同。如需使用訂閱引數的詳細資訊，請參閱[使用訂閱引數](#)。如需運算子清單，請參閱[AWS AppSync 解析程式映射範本公用程式參考](#)。

基於本文件的目的，我們會將即時資料篩選分為下列類別：

- 基本篩選 - 根據訂閱查詢中的用戶端定義引數進行篩選。
- 增強型篩選 - 根據 AWS AppSync 服務後端中集中定義的邏輯進行篩選。

下列各節說明如何設定增強型訂閱篩選條件，並顯示其實際用途。

在 GraphQL 結構描述中定義訂閱

若要使用增強型訂閱篩選條件，請在 GraphQL 結構描述中定義訂閱，然後使用篩選延伸來定義增強型篩選條件。若要說明增強型訂閱篩選的運作方式 AWS AppSync，請使用下列 GraphQL 結構描述，其定義票證管理系統 API，例如：

```
type Ticket {
  id: ID
  createdAt: AWSDateTime
  content: String
  severity: Int
  priority: Priority
  category: String
  group: String
  status: String
}

type Mutation {
  createTicket(input: TicketInput): Ticket
}

type Query {
  getTicket(id: ID!): Ticket
}

type Subscription {
  onSpecialTicketCreated: Ticket @aws_subscribe(mutations: ["createTicket"])
  onGroupTicketCreated(group: String!): Ticket @aws_subscribe(mutations:
    ["createTicket"])
}

enum Priority {
  none
  lowest
  low
  medium
  high
  highest
}
```



```
}

input TicketInput {
  content: String
  severity: Int
  priority: Priority
  category: String
  group: String
}
```

假設您為 API 建立NONE資料來源，然後使用此資料來源將解析程式連接到createTicket變動。您的處理常式可能如下所示：

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
  return {
    payload: {
      id: util.autoId(),
      createdAt: util.time.nowISO8601(),
      status: 'pending',
      ...ctx.args.input,
    },
  };
}

export function response(ctx) {
  return ctx.result;
}
```

Note

增強型篩選條件會在指定訂閱的 GraphQL 解析程式處理常式中啟用。如需詳細資訊，請參閱[解析程式參考](#)。

若要實作增強型篩選條件的行為，您必須使用 `extensions.setSubscriptionFilter()` 函數來定義篩選條件表達式，根據訂閱用戶端可能感興趣的 GraphQL 變動所發佈的資料進行評估。如需篩選延伸模組的詳細資訊，請參閱[延伸模組](#)。

下一節說明如何使用篩選延伸模組來實作增強型篩選條件。

使用篩選延伸項目建立增強型訂閱篩選條件

增強型篩選條件會以 JSON 寫入訂閱解析程式的回應處理常式中。篩選條件可以分組在稱為的清單中filterGroup。篩選條件是使用至少一個規則來定義，每個規則都有欄位、運算子和值。讓我們為定義新的解析程式onSpecialTicketCreated，以設定增強型篩選條件。您可以在使用 AND 邏輯評估的篩選條件中設定多個規則，而篩選群組中的多個篩選條件則使用 OR 邏輯評估：

```
import { util, extensions } from '@aws-appsync/utils';

export function request(ctx) {
  // simply return null for the payload
  return { payload: null };
}

export function response(ctx) {
  const filter = {
    or: [
      { severity: { ge: 7 }, priority: { in: ['high', 'medium'] } },
      { category: { eq: 'security' }, group: { in: ['admin', 'operators'] } },
    ],
  };
  extensions.setSubscriptionFilter(util.transform.toSubscriptionFilter(filter));

  // important: return null in the response
  return null;
}
```

根據上述範例中定義的篩選條件，如果票證是以下列方式建立，則會自動將重要票證推送至訂閱的 API 用戶端：

- priority 關卡high或 medium

AND

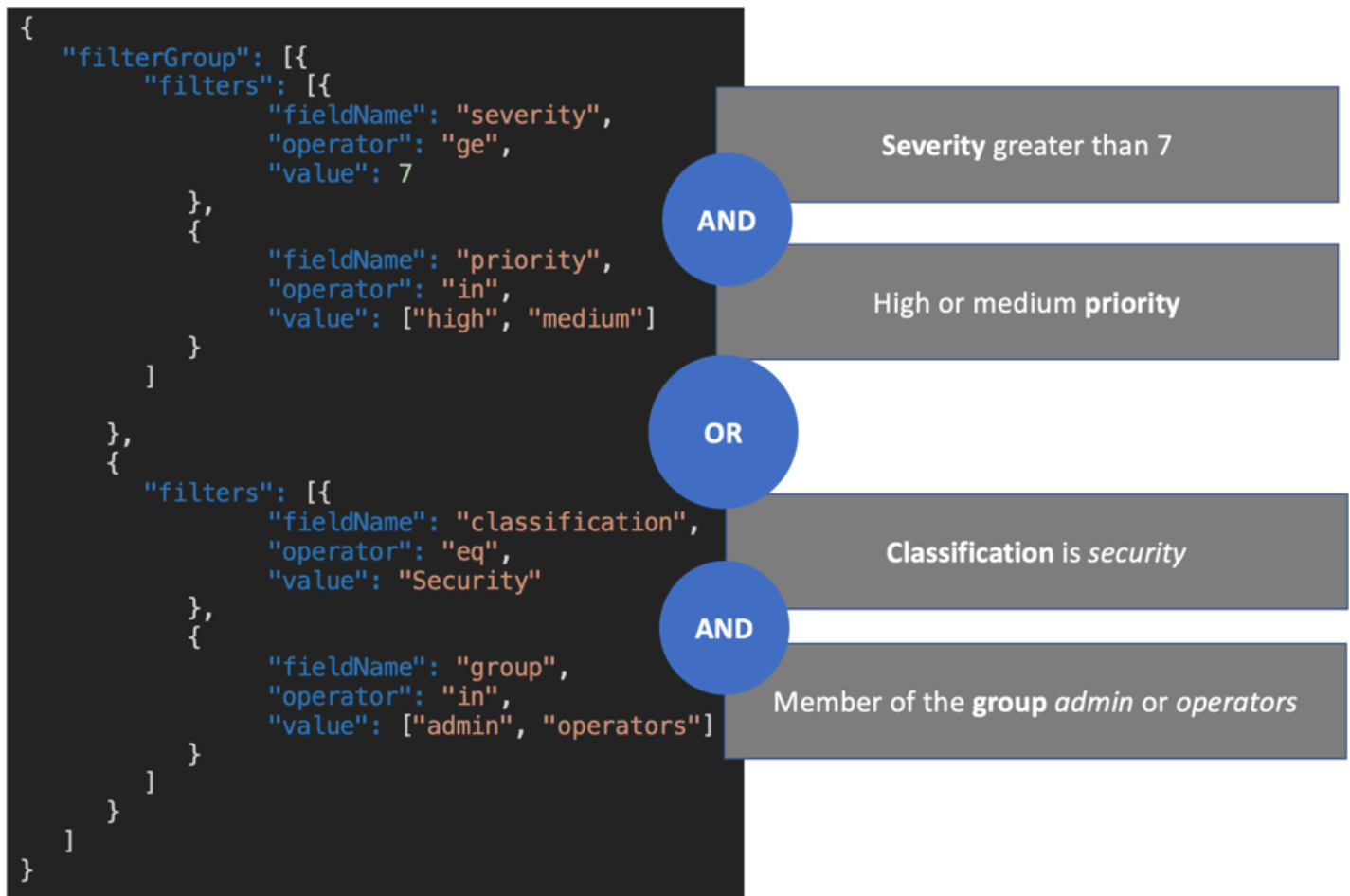
- severity 層級大於或等於 7(ge)

或

- classification 票證設定為 Security

AND

- group 指派設定為 admin或 operators



訂閱解析程式中定義的篩選條件（增強篩選）優先於僅根據訂閱引數進行篩選（基本篩選）。如需使用訂閱引數的詳細資訊，請參閱[使用訂閱引數](#)。

如果在訂閱的 GraphQL 結構描述中定義並需要引數，則只有在引數在解析程式的 `extensions.setSubscriptionFilter()` 方法中定義為規則時，才會根據指定的引數進行篩選。不過，如果訂閱解析程式中沒有 `extensions` 篩選方法，則用戶端中定義的引數只會用於基本篩選。您無法同時使用基本篩選和增強型篩選。

您可以使用訂閱篩選條件延伸邏輯中的 [context 變數](#) 來存取請求的相關內容資訊。例如，使用 Amazon Cognito 使用者集區、OIDC 或 Lambda 自訂授權方進行授權時，您可以在建立訂閱 `context.identity` 時擷取 中使用者的相關資訊。您可以使用該資訊，根據使用者的身分建立篩選條件。

現在假設您想要實作 的增強型篩選條件行為 `onGroupTicketCreated`。 `onGroupTicketCreated` 訂閱需要強制 `group` 名稱做為引數。建立時，票證會自動指派 `pending` 狀態。您可以設定訂閱篩選條件，只接收屬於所提供群組的新建立票證：

```
import { util, extensions } from '@aws-appsync/utils';

export function request(ctx) {
  // simplify return null for the payload
  return { payload: null };
}

export function response(ctx) {
  const filter = { group: { eq: ctx.args.group }, status: { eq: 'pending' } };
  extensions.setSubscriptionFilter(util.transform.toSubscriptionFilter(filter));

  return null;
}
```

使用如下範例中的變動發佈資料時：

```
mutation CreateTicket {
  createTicket(input: {priority: medium, severity: 2, group: "aws"}) {
    id
    priority
    severity
    status
    group
    createdAt
  }
}
```

一旦建立具有createTicket變動的票證，訂閱的用戶端就會監聽透過 WebSockets 自動推送的資料：

```
subscription OnGroup {
  onGroupTicketCreated(group: "aws") {
    category
    status
    severity
    priority
    id
    group
    createdAt
    content
  }
}
```

無需引數即可訂閱用戶端，因為篩選邏輯是以增強型篩選在 AWS AppSync 服務中實作，可簡化用戶端程式碼。只有在符合定義的篩選條件時，用戶端才會接收資料。

定義巢狀結構描述欄位的增強型篩選條件

您可以使用增強型訂閱篩選來篩選巢狀結構描述欄位。假設我們修改了上一節的結構描述，以包含位置和地址類型：

```
type Ticket {
  id: ID
  createdAt: AWSDateTime
  content: String
  severity: Int
  priority: Priority
  category: String
  group: String
  status: String
  location: ProblemLocation
}

type Mutation {
  createTicket(input: TicketInput): Ticket
}

type Query {
  getTicket(id: ID!): Ticket
}

type Subscription {
  onSpecialTicketCreated: Ticket @aws_subscribe(mutations: ["createTicket"])
  onGroupTicketCreated(group: String!): Ticket @aws_subscribe(mutations:
    ["createTicket"])
}

type ProblemLocation {
  address: Address
}

type Address {
  country: String
}

enum Priority {
```

```

  none
  lowest
  low
  medium
  high
  highest
}

input TicketInput {
  content: String!
  severity: Int!
  priority: Priority!
  category: String!
  group: String!
  location: AWSJSON!
}
```

透過此結構描述，您可以使用 . 分隔符號來表示巢狀化。下列範例會為 下的巢狀結構描述欄位新增篩選條件規則 `location.address.country`。如果票證的地址設定為 `USA`，則會觸發訂閱 `USA`：

```

import { util, extensions } from '@aws-appsync/utils';

export const request = (ctx) => ({ payload: null });

export function response(ctx) {
  const filter = {
    or: [
      { severity: { ge: 7 }, priority: { in: ['high', 'medium'] } },
      { category: { eq: 'security' }, group: { in: ['admin', 'operators'] } },
      { 'location.address.country': { eq: 'USA' } },
    ],
  };
  extensions.setSubscriptionFilter(util.transform.toSubscriptionFilter(filter));
  return null;
}
```

在上述範例中，`location` 代表巢狀層級一，`address` 代表巢狀層級二，`country` 代表巢狀層級三，所有這些都由 . 分隔符號分隔。

您可以使用 `createTicket` 變動來測試此訂閱：

```

mutation CreateTicketInUSA {
  createTicket(input: {location: "{\"address\":{\"country\":\"USA\"}}"}) {
    category
  }
}
```

```

    content
    createdAt
    group
    id
    location {
      address {
        country
      }
    }
    priority
    severity
    status
  }
}
```

從用戶端定義增強型篩選條件

您可以在 GraphQL 中使用基本篩選搭配[訂閱引數](#)。在訂閱查詢中進行呼叫的用戶端會定義引數的值。在具有extensions篩選條件的 AWS AppSync 訂閱解析程式中啟用增強型篩選條件時，解析程式中定義的後端篩選條件會優先且優先。

使用訂閱中的filter引數設定動態、用戶端定義的增強型篩選條件。設定這些篩選條件時，您必須更新 GraphQL 結構描述以反映新引數：

```

...
type Subscription {
  onSpecialTicketCreated(filter: String): Ticket
    @aws_subscribe(mutations: ["createTicket"])
}
...
```

然後，用戶端可以傳送訂閱查詢，如下列範例所示：

```

subscription onSpecialTicketCreated($filter: String) {
  onSpecialTicketCreated(filter: $filter) {
    id
    group
    description
    priority
    severity
  }
}
```

您可以設定查詢變數，如下列範例所示：

```
{"filter" : "{\"severity\":{\"le\":2}}"}}
```

`util.transform.toSubscriptionFilter()` 解析程式公用程式可在訂閱回應映射範本中實作，以套用每個用戶端訂閱引數中定義的篩選條件：

```
import { util, extensions } from '@aws-appsync/utils';

export function request(ctx) {
  // simplify return null for the payload
  return { payload: null };
}

export function response(ctx) {
  const filter = ctx.args.filter;
  extensions.setSubscriptionFilter(util.transform.toSubscriptionFilter(filter));
  return null;
}
```

透過此策略，用戶端可以定義自己的篩選條件，使用增強型篩選邏輯和其他運算子。當指定的用戶端在安全的 WebSocket 連線中調用訂閱查詢時，會指派篩選條件。如需增強型篩選之轉換公用程式的詳細資訊，包括 `filter` 查詢變數承載的格式，請參閱 [JavaScript 解析程式概觀](#)。

其他增強型篩選限制

以下是幾個使用案例，其中對增強型篩選條件施加了額外的限制：

- 增強型篩選條件不支援篩選最上層物件清單。在此使用案例中，增強型訂閱會忽略來自變動的已發佈資料。
- AWS AppSync 最多支援五個巢狀層級。將忽略巢狀層級 5 以上結構描述欄位的篩選條件。使用以下 GraphQL 回應。允許中的 `continent` 欄位，`venue.address.country.metadata.continent` 因為它是層級 5 巢狀。不過，`financial.venue.address.country.metadata.capital.financial` 是第六層巢狀，因此篩選條件無法運作：

```
{
  "data": {
    "onCreateFilterEvent": {
      "venue": {
        "address": {
```



```
        "country": {
          "metadata": {
            "capital": {
              "financial": "New York"
            },
            "continent" : "North America"
          }
        },
        "state": "WA"
      },
      "builtYear": 2023
    },
    "private": false,
  }
}
```

在 中使用篩選條件取消訂閱 WebSocket 連線 AWS AppSync

Important

自 2025 年 3 月 13 日起，您可以使用 AWS AppSync Events 建置由 WebSockets 支援的即時 PubSub API。如需詳細資訊，請參閱《AppSync Events 開發人員指南》中的[透過 WebSocket 發佈事件](#)。AWS AppSync

在 中 AWS AppSync，您可以強制取消訂閱，並根據特定篩選邏輯從連線用戶端關閉（驗證）WebSocket 連線。這在授權相關案例中非常有用，例如當您從群組中移除使用者時。

訂閱失效是為了回應變動中定義的承載。我們建議您將用於使訂閱連線失效的變動視為 API 中的管理操作，並限制其對管理員使用者、群組或後端服務的使用，藉此相應地限定其範圍許可。例如，使用結構描述授權指令，例如 `@aws_auth(cognito_groups: ["Administrators"])` 或 `@aws_iam`。如需詳細資訊，請參閱[使用其他授權模式](#)。

失效篩選條件使用與[增強型訂閱篩選條件](#)相同的語法和邏輯。使用下列公用程式定義這些篩選條件：

- `extensions.invalidateSubscriptions()` – 在 GraphQL 解析程式的變動回應處理常式中定義。
- `extensions.setSubscriptionInvalidationFilter()` – 在連結至變動之訂閱的 GraphQL 解析程式回應處理常式中定義。

如需失效篩選延伸模組的詳細資訊，請參閱 [JavaScript 解析程式概觀](#)。

使用訂閱失效

若要查看訂閱失效的運作方式 AWS AppSync，請使用下列 GraphQL 結構描述：

```
type User {
  userId: ID!
  groupId: ID!
}

type Group {
  groupId: ID!
  name: String!
  members: [ID!]!
}

type GroupMessage {
  userId: ID!
  groupId: ID!
  message: String!
}

type Mutation {
  createGroupMessage(userId: ID!, groupId : ID!, message: String!): GroupMessage
  removeUserFromGroup(userId: ID!, groupId : ID!) : User @aws_iam
}

type Subscription {
  onGroupMessageCreated(userId: ID!, groupId : ID!): GroupMessage
    @aws_subscribe(mutations: ["createGroupMessage"])
}

type Query {
  none: String
}
```

在 `removeUserFromGroup` 變動解析程式碼中定義失效篩選條件：

```
import { extensions } from '@aws-appsync/utils';

export function request(ctx) {
  return { payload: null };
}
```

```

}

export function response(ctx) {
  const { userId, groupId } = ctx.args;
  extensions.invalidateSubscriptions({
    subscriptionField: 'onGroupMessageCreated',
    payload: { userId, groupId },
  });
  return { userId, groupId };
}

```

調用變動時，payload 物件中定義的資料會用來取消訂閱中定義的訂閱subscriptionField。失效篩選條件也會在onGroupMessageCreated訂閱的回應映射範本中定義。

如果extensions.invalidateSubscriptions()承載包含的 ID 符合篩選條件中定義的訂閱用戶端 IDs，則會取消訂閱對應的訂閱。此外，WebSocket 連線已關閉。定義訂閱的onGroupMessageCreated訂閱解析程式碼：

```

import { util, extensions } from '@aws-appsync/utils';

export function request(ctx) {
  // simplify return null for the payload
  return { payload: null };
}

export function response(ctx) {
  const filter = { groupId: { eq: ctx.args.groupId } };
  extensions.setSubscriptionFilter(util.transform.toSubscriptionFilter(filter));

  const invalidation = { groupId: { eq: ctx.args.groupId }, userId: { eq:
    ctx.args.userId } };
  extensions.setSubscriptionInvalidationFilter(util.transform.toSubscriptionFilter(invalidation));

  return null;
}

```

請注意，訂閱回應處理常式可以同時定義訂閱篩選條件和失效篩選條件。

例如，假設用戶端 A *group-1* 使用以下訂閱請求，將 ID 為 的新使用者訂閱*user-1*至 ID 為 的群組：

```
onGroupMessageCreated(userId : "user-1", groupId: :"group-1") {...}
```

AWS AppSync 會執行訂閱解析程式，其會產生訂閱和失效篩選條件，如上所述onGroupMessageCreated回應映射範本所定義。對於用戶端 A，訂閱篩選條件僅允許將資料傳送至 *group-1*，並同時為 *user-1*和 定義失效篩選條件*group-1*。

現在假設用戶端 B *group-2*使用以下訂閱請求，將 ID 為 的使用者訂閱*user-2*至 ID 為 的群組：

```
onGroupMessageCreated(userId : "user-2", groupId : "group-2") {...}
```

AWS AppSync 會執行訂閱解析程式，產生訂閱和失效篩選條件。對於用戶端 B，訂閱篩選條件僅允許將資料傳送至 *group-2*，並同時為 *user-2*和 定義失效篩選條件*group-2*。

接著，假設具有 ID 的新群組訊息*message-1*是使用變動請求建立，如下列範例所示：

```
createGroupMessage(id: "message-1", groupId :  
    "group-1", message: "test message"){...}
```

符合已定義篩選條件的訂閱用戶端會透過 WebSockets 自動接收下列資料承載：

```
{  
  "data": {  
    "onGroupMessageCreated": {  
      "id": "message-1",  
      "groupId": "group-1",  
      "message": "test message",  
    }  
  }  
}
```

用戶端 A 會收到訊息，因為篩選條件符合定義的訂閱篩選條件。不過，用戶端 B 不會收到訊息，因為使用者不是 的一部分*group-1*。此外，請求不符合訂閱解析程式中定義的訂閱篩選條件。

最後，假設*user-1*使用*group-1*下列變動請求從 中移除：

```
removeUserFromGroup(userId: "user-1", groupId : "group-1"){...}
```

變動會啟動其extensions.invalidateSubscriptions()解析程式回應處理常式程式碼中定義的訂閱失效。AWS AppSync 然後取消訂閱用戶端 A 並關閉其 WebSocket 連線。用戶端 B 不受影響，因為變動中定義的失效承載與其使用者或群組不相符。

當連線 AWS AppSync 失效時，用戶端會收到確認已取消訂閱的訊息：

```
{  
  "message": "Subscription complete."  
}
```

在訂閱失效篩選條件中使用內容變數

如同增強型訂閱篩選條件，您可以在訂閱失效篩選條件延伸中使用 [context變數](#) 來存取特定資料。

例如，您可以將電子郵件地址設定為變動中的失效承載，然後將其與使用 Amazon Cognito 使用者集區或 OpenID Connect 授權之訂閱使用者的電子郵件屬性或宣告進行比對。extensions.setSubscriptionInvalidationFilter() 訂閱失效器中定義的失效篩選條件會檢查變動extensions.invalidateSubscriptions()承載設定的電子郵件地址是否與 中使用者 JWT 字串擷取的電子郵件地址相符context.identity.claims.email，進而啟動失效。

建置即時 WebSocket 用戶端 in AWS AppSync

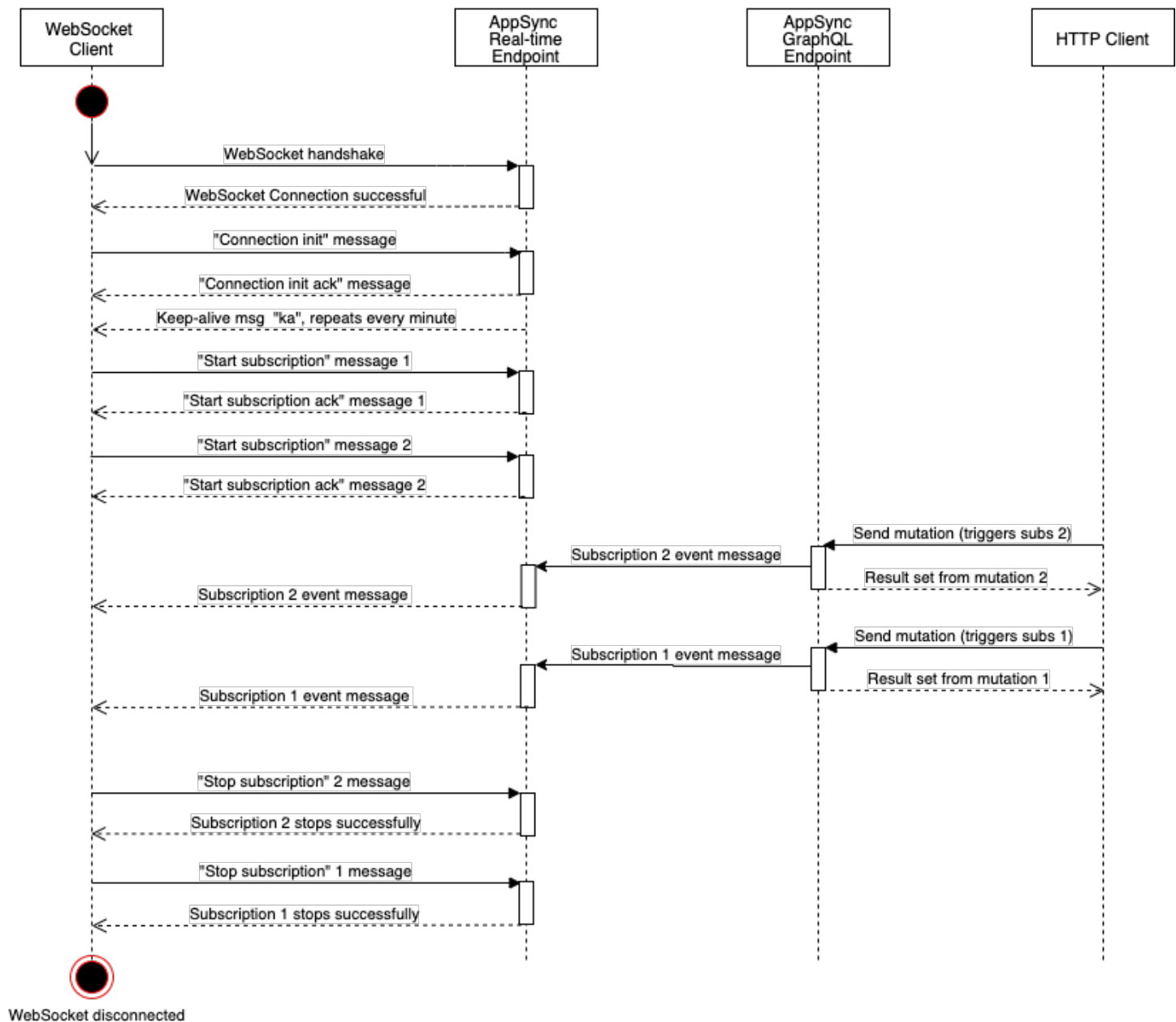
Important

自 2025 年 3 月 13 日起，您可以使用 AWS AppSync Events 建置由 WebSockets 支援的即時 PubSub API。如需詳細資訊，請參閱《AppSync Events 開發人員指南》中的[透過 WebSocket 發佈](#)事件。AWS AppSync

AWS AppSync 的即時 WebSocket 用戶端可透過多步驟程序啟用 GraphQL 訂閱。用戶端會先與 AWS AppSync 即時端點建立 WebSocket 連線、傳送連線初始化訊息，並等待確認。成功連線後，用戶端會透過傳送具有唯一 IDs 和 GraphQL 查詢的開始訊息來註冊訂閱。AWS AppSync 會以確認訊息確認成功訂閱。然後，用戶端會接聽由對應變動觸發的訂閱事件。為了維持連線，AWS AppSync 會定期傳送保持連線訊息。完成後，用戶端會透過傳送停止訊息來取消註冊訂閱。此系統在單一 WebSocket 連線上支援多個訂閱，並容納各種授權模式，包括 API 金鑰、Amazon Cognito 使用者集區、IAM 和 Lambda。

GraphQL 訂閱的即時 WebSocket 用戶端實作

下列序列圖和步驟顯示 WebSocket 用戶端、HTTP 用戶端和 AWS AppSync 之間的即時訂閱工作流程。



1. 用戶端會與 AWS AppSync 即時端點建立 WebSocket 連線。如果存在網路錯誤，用戶端應該執行抖動的指數輪詢。如需詳細資訊，請參閱 AWS 架構部落格上的[指數退避和抖動](#)。
2. (選用) 成功建立 WebSocket 連線後，用戶端會傳送訊息 `connection_init`。
3. 如果 `connection_init` 已傳送，用戶端會等待來自 AWS AppSync `connection_ack` 的訊息。此訊息包含 `connectionTimeoutMs` 參數，這是 "ka"(keep-alive) 訊息的最大等待時間，以毫秒為單位。
4. AWS AppSync 會定期 "ka" 傳送訊息。用戶端會追蹤收到每則 "ka" 訊息的時間。如果用戶端未在 `connectionTimeoutMs` 幾毫秒內收到 "ka" 訊息，用戶端應該關閉連線。

5. 用戶端會透過傳送訂閱訊息來註冊 `start` 訂閱。單一 WebSocket 連線支援多個訂閱，即使它們處於不同的授權模式。
6. 用戶端等待 for AWS AppSync `start_ack` 傳送訊息以確認訂閱成功。如果發生錯誤，AWS AppSync 會傳回 `"type": "error"` 訊息。
7. 用戶端會接聽訂閱事件，這些事件會在呼叫對應的變動後傳送。查詢和變動通常會透過 `https://` 傳送至 AWS AppSync GraphQL 端點。訂閱會使用安全的 WebSocket () 流經 AWS AppSync 即時端點 `wss://`。
8. 用戶端會透過傳送訂閱訊息來取消註冊 `stop` 訂閱。
9. 取消註冊所有訂閱並檢查沒有透過 WebSocket 傳輸的訊息後，用戶端可以斷開與 WebSocket 的連線。

建立 WebSocket 連線的交握詳情

若要使用 AWS AppSync 連接和啟動成功的交握，WebSocket 用戶端需要下列項目：

- The AWS AppSync 即時端點
- 標頭 – 包含與 AWS AppSync 端點和授權相關的資訊。AWS AppSync 支援下列三種提供標頭的方法：
 - 透過查詢字串的標頭
 - 標頭資訊編碼為 base64 字串，衍生自字串化 JSON 物件。此 JSON 物件包含與 AWS AppSync 端點和授權相關的詳細資訊。JSON 物件的內容會根據授權模式而有所不同。
 - 透過 的標頭 `Sec-WebSocket-Protocol`
 - 來自字串化 JSON 物件的 base64Url-encoded 字串，其中包含與 AWS AppSync 端點和授權相關的資訊，會做為 `Sec-WebSocket-Protocol` 標頭中的通訊協定傳遞。JSON 物件的內容會根據授權模式而有所不同。
 - 透過標準 HTTP 標頭的標頭：
 - 標頭可以在連線請求中做為標準 HTTP 標頭傳遞，類似於 GraphQL 查詢和變動至 AWS AppSync 的標頭傳遞方式。不過，私有 API 連線請求不支援透過標準 HTTP 標頭傳遞標頭。
- payload – Base64-encoded 字串 payload。只有在使用查詢字串提供標頭時，才需要承載

透過這些要求，WebSocket 用戶端可以連接到 URL，其中包含具有查詢字串的即時端點，使用 `graphql-ws` 做為 WebSocket 通訊協定。

從 GraphQL 端點探索 即時端點

在通訊協定和網域中，AWS AppSync GraphQL 端點和 AWS AppSync 即時端點略有不同。您可以使用 AWS Command Line Interface (AWS CLI) 命令 擷取 GraphQL 端點 `aws appsync get-graphql-api`。

AWS AppSync GraphQL 端點：

```
https://example1234567890000.appsync-api.us-east-1.amazonaws.com/graphql
```

AWS AppSync 即時端點：

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql
```

應用程式可以使用任何 HTTP 用戶端連線至 AWS AppSync GraphQL 端點 (`https://`) 進行查詢和變動。應用程式可以使用訂閱的任何 WebSocket 用戶端連線到 AWS AppSync 即時端點 (`wss://`)。

使用自訂網域名稱，您可以使用單一網域與兩個端點互動。例如，如果您將 `api.example.com` 設定為自訂網域，則可以使用這些 URLs 與 GraphQL 和即時端點互動：

AWS AppSync 自訂網域 GraphQL 端點：

```
https://api.example.com/graphql
```

AWS AppSync 自訂網域即時端點：

```
wss://api.example.com/graphql/realtime
```

以 AWS AppSync API 授權模式為基礎的標頭參數格式

連線查詢字串中使用的 `header` 物件格式會根據 AWS AppSync API 授權模式而有所不同。物件中的 `host` 欄位是指 AWS AppSync GraphQL 端點，即使對即時端點進行 `wss://` 呼叫，也會用來驗證連線。若要啟動交握並建立授權連線，`payload` 應該是空的 JSON 物件。只有在標頭透過查詢字串傳遞時，才需要承載。

下列各節示範每個授權模式的標頭格式。

API 金鑰

API 金鑰標頭

標頭內容

- "host": <string> : AWS AppSync GraphQL 端點的主機或您的自訂網域名稱。
- "x-api-key": <string> : 為 AWS AppSync API 設定的 API 金鑰。

範例

```
{
  "host": "example1234567890000.appsync-api.us-east-1.amazonaws.com",
  "x-api-key": "da2-12345678901234567890123456"
}
```

透過查詢字串的標頭

首先，包含 host 和的 JSON 物件 x-api-key 會轉換為字串。接下來，此字串會使用 base64 編碼進行編碼。產生的 base64 編碼字串會新增為名為 WebSocket URL header 的查詢參數，以建立與 AWS AppSync 即時端點的連線。產生的請求 URL 採用下列形式：

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql?
header=eyJ0b3N0IjoieXhhbXBsZTEyMzQ1Njc4MDAwMDAuYXBwc3luYy1hcGkudXMtZWZzdC0xLmFtYXpvbmF3cy5jb20i
```

請務必注意，除了 base64 編碼的標頭物件之外，空的 JSON 物件 {} 也是 base64 編碼的，並包含在 WebSocket URL payload 中做為名為 的個別查詢參數。

透過 的標頭 Sec-WebSocket-Protocol

包含 host 和的 JSON 物件 x-api-key 會轉換為字串，然後使用 base64Url 編碼進行編碼。產生的 base64Url-encoded 字串字首為 header-。在與 AWS AppSync 即時端點建立 WebSocket 連線時，除了 Sec-WebSocket-Protocol 標頭 graphql-ws 中的 之外，此字首字串也會用作新的子通訊協定。

產生的請求 URL 採用下列形式：

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql
```

Sec-WebSocket-Protocol 標頭包含下列值：

```
"sec-websocket-protocol" : ["graphql-ws", "header-
ewogICAgImhvc3QiOiJleGFtcGx1MTIzNDU2Nzg5MDAwMC5hcHBzeW5jLWFWaS51cy1lYXN0LTEuYW1hem9uYXdzLmNvbSI
```

透過標準 HTTP 標頭的標頭

在此方法中，在與 AWS AppSync 即時端點建立 WebSocket 連線時，會使用標準 HTTP 標頭傳輸主機和 API 金鑰資訊。產生的請求 URL 採用下列形式：

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql
```

請求標頭會包含下列項目：

```
"sec-websocket-protocol" : ["graphql-ws"]
"host": "example1234567890000.appsync-api.us-east-1.amazonaws.com",
"x-api-key": "da2-12345678901234567890123456"
```

Amazon Cognito 使用者集區和 OpenID 連線 (OIDC)

Amazon Cognito 和 OIDCheader

標頭內容：

- "Authorization": <string> : JWT ID 字符。標頭可以使用[承載結構](#)描述。
- "host": <string> : AWS AppSync GraphQL 端點的主機或您的自訂網域名稱。

範例：

```
{
  "Authorization": "eyJEXAMPLEiJjbG5xb3A5eW5MK09QYXIrMTJHWEFLSXBiU5WNHhsQjEXAMPLEnM2WldvPSIsImFsZS9zZWEE2DJH7sH0l2zxYi7f-SmEGoh2AD8emxQRYajByz-rE4Jh0Q0ymN2Ys-ZIkMpVBTPgu-TMWdy0HhDumUj20P82yeZ3wLZAtr_gM4LzjXUXmI_K2yGjuXfXTaa1mvQEBG0mQfVd7SfwXB-jcv4RYVi6j25qgow9Ew52ufurPqaK-3WAKG32KpV8J4-Wejq8t0c-yA7sb8EnB551b7TU93uKRiVVK3E55Nk5ADPoam_WYE45i3s5qVAP_-InW75NUo0CGTsS8YWMfb6ecHYJ-1j-bzA27zaT9VjctXn9byNFZmEXAMPLExw",
  "host": "example1234567890000.appsync-api.us-east-1.amazonaws.com"
}
```

透過查詢字串的標頭

首先，包含 host 和的 JSON 物件 Authorization 會轉換為字串。接下來，此字串會使用 base64 編碼進行編碼。產生的 base64 編碼字串會新增為名為 WebSocket URL header 的查詢參數，以建立與 AWS AppSync 即時端點的連線。產生的請求 URL 採用下列形式：

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql?
header=eyJBdXRob3JpemF0aW9uIjoizXlKcmFXUWlPaUpqYkc1eGIZQTVlVzVNSzA5UVlYSXJNVEpIV0VGTFNYQm1lVTVX
```

請務必注意，除了 base64 編碼的標頭物件之外，空的 JSON 物件 {} 也是 base64 編碼的，並包含在 WebSocket URL payload 中做為名為 `header` 的個別查詢參數。

透過的標頭 **Sec-WebSocket-Protocol**

包含 `host` 和的 JSON 物件 `Authorization` 會轉換為字串，然後使用 `base64Url` 編碼進行編碼。產生的 `base64Url-encoded` 字串字首為 `header-`。在與 AWS AppSync 即時端點建立 WebSocket 連線時，除了 `Sec-WebSocket-Protocol` 標頭 `graphql-ws` 中的之外，此字首字串也會用作新的子通訊協定。

產生的請求 URL 採用下列形式：

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql
```

`Sec-WebSocket-Protocol` 標頭包含下列值：

```
"sec-websocket-protocol" : ["graphql-ws", "header-
ewogICAgImhvc3QiOiJleGFtcGx1MTIzNDU2Nzg5MDAwMC5hcHBzeW5jLWFwaS51cy1lYXN0LTEuYW1hem9uYXdzLmNvbSI
```

透過標準 HTTP 標頭的標頭

在此方法中，在建立與 AWS AppSync 即時端點的 WebSocket 連線時，會使用標準 HTTP 標頭傳輸主機和授權資訊。產生的請求 URL 採用下列形式：

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql
```

請求標頭會包含下列項目：

```
"sec-websocket-protocol" : ["graphql-ws"]
"Authorization": "eyJEXAMPLEiJjbG5xb3A5eW5MK09QYXIrMTJHWEFLSXBiU5WNHhsQjEXAMPLEnM2WldvPSIsImFsZy
zEE2DJH7sH0l2zxYi7f-SmEGoh2AD8emxQRYajByz-rE4Jh0Q0ymN2Ys-ZIkMpVBTPgu-
TMWDyOHhDumUj20P82yeZ3wLZAtr_gM4LzjXUXmI_K2yGjuXfXTaa1mvQEBG0mQfVd7SfwXB-
jcv4RYVi6j25qgow9Ew52ufurPqaK-3WAKG32KpV8J4-Wejq8t0c-
yA7sb8EnB551b7TU93uKRiVVK3E55Nk5ADPoam_WYE45i3s5qVAP_-InW75NUo0CGTsS8YWMfb6ecHYJ-1j-
bzA27zaT9VjctXn9byNFZmEXAMPLExw",
"host": "example1234567890000.appsync-api.us-east-1.amazonaws.com"
```

IAM

IAM 標頭

標頭內容

- "accept": "application/json, text/javascript" : 常數 <string> 參數。
- "content-encoding": "amz-1.0" : 常數 <string> 參數。
- "content-type": "application/json; charset=UTF-8" : 常數 <string> 參數。
- "host": <string> : 這是 AWS AppSync GraphQL 端點的主機。
- "x-amz-date": <string> : 時間戳記必須採用 UTC 和下列 ISO 8601 格式 : YYYYMMDD'T'HHMMSS'Z'。例如 , 20150830T123600Z 是有效的時間戳記。不包含時間戳記的毫秒數。如需詳細資訊 , 請參閱 [中的 Signature 第 4 版中的處理日期](#) AWS 一般參考。
- "X-Amz-Security-Token": <string> : AWS 工作階段字符 , 使用臨時安全登入資料時需要。如需詳細資訊 , 請參閱《IAM 使用者指南》中的[搭配使用暫時憑證與 AWS 資源](#)。
- "Authorization": <string> : AWS AppSync 端點的簽章第 4 版 (SigV4) 簽署資訊。如需簽署程序的詳細資訊 , 請參閱《》中的[任務 4 : 將簽章新增至 HTTP 請求](#) AWS 一般參考。

SigV4 簽署 HTTP 請求包含正式 URL , 這是 /connect 附加的 AWS AppSync GraphQL 端點。服務端點 AWS 區域與您使用 AWS AppSync API 的區域相同 , 且服務名稱為 'appsync'。要簽署的 HTTP 請求如下 :

```
{
  url: "https://example1234567890000.appsync-api.us-east-1.amazonaws.com/graphql/connect",
  data: "{}",
  method: "POST",
  headers: {
    "accept": "application/json, text/javascript",
    "content-encoding": "amz-1.0",
    "content-type": "application/json; charset=UTF-8",
  }
}
```

範例

```
{
  "accept": "application/json, text/javascript",
```

```

"content-encoding": "amz-1.0",
"content-type": "application/json; charset=UTF-8",
"host": "example1234567890000.appsync-api.us-east-1.amazonaws.com",
"x-amz-date": "20200401T001010Z",
"X-Amz-Security-Token":
"AgEXAMPLEZ2luX2VjEAoaDmFwLXNvdXR0ZWFEEXAMPLEcwrQIgAh97Cljq7w0PL8KsxP3YtDuyC/9hAj8PhJ7Fvf38SgoC
+
+pEagWCveZUjKE0zyUhBEXAMPLEjj//////////8BEXAMPLEx0Dk2NDgyNzg1NSIMo1mWnpESWUoYw4BkKqEFSrm3DXuL8
+ZbVc4JKjDP4vUCKNR6Le9C9pZp9Psw0NoFy3vLBUDAXEXAMPLE0VG8feXfiEEA+1khgFK/
wEtW+9zF7NaMMSe07wN2gG2tH0eKMEXAMPLEQX+sMbytQo8iepP9PZ0z1ZsSFb/
dP5Q8hk6YEXAMPLEYcKZsTkDAq2uKFQ8mYUVA9EtQnNRiFLEY83aKvG/tqLWNnG1SNVx7SMcfovkFDqQamm
+88y10wwAEYK7qcocX6Z7G6GaYuIfGpaX2MCCELeQvZ+8WxEg0nIfz7GYvsYNjLZSaRnV4G
+ILY1F0QNW64S9Nvj
+BwDg3ht2CrNvpwjVY1j9U3nmxE0UG5ne83LL5hhqMpm25kmL7enVgw2kQzmU2id4IKu0C/
WaoDRu02F5zE63vJbxN8AYs7338+4B4HBb6BZ60Ugg96Q15RA41/
gIqxaVPxyTpDfTU5GfSLxocdYeniqqpFMtZG2n9d0u7GsQNCfKNCg3qDZm4tDo8tZbuym0a2VcF2E5hFEgXBa
+XLJCfXi/770qAEjP0x7Qdk3B43p8KG/BaioP5RsV8zBGvH1zAgyPha2rN70/
tT13yrmPd5QYefwzexjKrV4mWiuRg8NTHYSZJUaeyCwTom80VFUJXG
+GYTUyv5W22aBcnoRGiCiKEYTL0kgXecdKFTHmcIAejQ9Welr0a196Kq87w5KNMCKcCGFnwBNFLmfmbpNqT6rUBxxs3X5nt
aox0FtHX21eF6qIGT8j1z+l2opU+ggwUgkhUUgCH2TfqBj+MLMVvpvgqJsPKt582caFKArIFIv0
+9QupxLnEH2hz04TMTfnU6bQC6z1buVe7h
+t0Lnh1YPFsLQ88anib/7TTC8k9DsBTq0ASE8R2GbSEsm09qbbMwgEaYU0KtGeyQsSJdhSk6XxXThrWL9EnwBCXDkICMqd
+WgtPtK00weDlCaRs3R2qXcbNgVhleMk4IwnF8D1695AenU1LwHj0JLkCjxgNFiWAFEPH9aEXAMPLExA==",
  "Authorization": "AWS4-HMAC-SHA256 Credential=XXXXXXXXXXXXXXXXXXXX/20200401/
us-east-1/appsync/aws4_request, SignedHeaders=accept;content-
encoding;content-type;host;x-amz-date;x-amz-security-token,
  Signature=83EXAMPLEbcc1fe3ee69f75cd5ebbf4cb4f150e4f99cec869f149c5EXAMPLEdc"
}

```

透過查詢字串的標頭

首先，包含 host(AWS AppSync GraphQL 端點) 和其他授權標頭的 JSON 物件會轉換為字串。接下來，此字串會使用 base64 編碼進行編碼。產生的 base64 編碼字串會新增至 WebSocket URL，做為名為 `header` 的查詢參數。產生的請求 URL 採用下列形式：

```

wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql?
header=eyJBdXRob3JpemF0aW9uIjoizXlKcmFXUWlPaUpqYkc1eGIzQTVlVzVNSzA5UVlYSXJNVEpIV0VGTFNYQm1lVTVX

```

請務必注意，除了 base64 編碼的標頭物件之外，空的 JSON 物件 `{}` 也是 base64 編碼的，並包含在 WebSocket URL payload 中做為名為 `header` 的個別查詢參數。

透過 `Sec-WebSocket-Protocol` 的標頭

包含 和其他授權標頭host的 JSON 物件會轉換為字串，然後使用 base64Url 編碼進行編碼。產生的 base64Url-encoded字串字首為 header-。在與 AWS AppSync 即時端點建立 WebSocket 連線時，除了 Sec-WebSocket-Protocol標頭graphql-ws中的 之外，此字首字串也會用作新的子通訊協定。

產生的請求 URL 採用下列形式：

wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql

Sec-WebSocket-Protocol 標頭包含下列值：

```
"sec-websocket-protocol" : ["graphql-ws", "header-ew0KICAiYWNjZXB0IjogImFwcGxpY2F0aw9uL2pzbz24sIHRleHQvamF2YXNjcmlwdCIsDQogICJjb250ZW50LWVuY29kaW5
```

透過標準 HTTP 標頭的標頭

在此方法中，在與 AWS AppSync 即時端點建立 WebSocket 連線時，會使用標準 HTTP 標頭傳輸主機和其他授權資訊。產生的請求 URL 採用下列形式：

wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql

請求標頭會包含下列項目：

```
"sec-websocket-protocol": ["graphql-ws"]  
"accept": "application/json, text/javascript",  
"content-encoding": "amz-1.0",  
"content-type": "application/json; charset=UTF-8",  
"host": "example1234567890000.apps-sync-api.us-east-1.amazonaws.com",  
"x-amz-date": "20200401T001010Z",  
"X-Amz-Security-Token":  
    "AgEXAMPLEEZ2luX2VjEAoaDmFwLXNvdXRoZWFEEXAMPLEEcwRQIgAh97Cljq7w0PL8KsxP3YtDuyc/9hAj8PhJ7Fvf38SgoC  
+  
+pEagWCveZUjkEn0zyUhBEXAMPLEjj//////////8BEXAMPLEExODk2NDgyNzg1NSIMo1mWnpESWUoYw4BkKqEF5rm3DXuL8  
+ZbVc4JKjDP4vUCKNR6Le9C9pZp9PsW0NoFy3vLBUDAXEXAMPLEEOVG8feXfiEEA+1khgFK/  
wEtWR+9zf7NaMMSe07wN2gG2tH0eKMEXAMPLEQX+sMbytQo8iepP9PZ0z1ZsSFb/  
dP5Q8hk6YEXAMPLEYcKZsTkDAq2uKFQ8mYUVA9EtQnNRiFLEY83aKvG/tqLWNnG1SNvx7SMcfovkFDqQamm  
+88y10wwAEYK7qcocex6Z7GGcaYuIfGpaX2MCCELEqvZ+8WxEgOnIfz7GYvsYNjLZSaRnV4G  
+ILY1F0QNw64S9Nvj  
+BwDg3ht2CrNvpwjVYlj9U3nmxE0UG5ne83LL5hhqMpm25kmL7enVgw2kQzmU2id4IKu0C/  
WaoDRuO2F5ze63vJbxN8AYs7338+4B4HBb6BZ60Ugg96Q15RA41/  
gIqxaVPxyTpDfTU5GfSLxocdYeniqqPFmtZG2n9d0u7GsQNcFkNcG3qDZm4tDo8tZbuym0a2VcF2E5hFEgXBa  
+XLJCfXi/770qAEjp0x7Qdk3B43p8KG/Baiop5RsV8zBGvH1zAqyPha2rN70/
```

```
tT13yrmPd5QYefwzexjKrV4mWIuRg8NTHYSZJUaeyCwTom80VFUJXG
+GYTUyv5W22aBcnoRGiCiKEYTL0kgXecdKFTHmcIAejQ9We1r0a196Kq87w5KNMckcCGFnwBNFLmfmbpNqT6rUBxss3X5nt
aox0FtHX21eF6qIGT8j1z+l2opU+ggwUgkhUUgCH2TfqBj+MLMVVvpqgJsPKt582caFKArIFiv0
+9QupxLnEH2hz04TMTfnU6bQC6z1buVe7h
+t0Lnh1YPFsLQ88anib/7TTC8k9DsBTq0ASe8R2GbSEsm09qbbMwgEaYUh0KtGeyQsSJdhSk6XxXThrWL9EnwBCXDkICMqd
+WgtPtK00weDlCaRs3R2qXcbNgVhleMk4IWnF8D1695AenU1LwHj0JLkCjxgNFfiWAFEPH9aEXAMPLExA==" ,
"Authorization": "AWS4-HMAC-SHA256 Credential=XXXXXXXXXXXXXXXXXXXX/20200401/
us-east-1/appsycn/aws4_request, SignedHeaders=accept;content-
encoding;content-type;host;x-amz-date;x-amz-security-token,
Signature=83EXAMPLEebcc1fe3ee69f75cd5ebbf4cb4f150e4f99cec869f149c5EXAMPLEdc"
```

若要使用自訂網域簽署請求：

```
{
  url: "https://api.example.com/graphql/connect",
  data: "{}",
  method: "POST",
  headers: {
    "accept": "application/json, text/javascript",
    "content-encoding": "amz-1.0",
    "content-type": "application/json; charset=UTF-8",
  }
}
```

範例

```
{
  "accept": "application/json, text/javascript",
  "content-encoding": "amz-1.0",
  "content-type": "application/json; charset=UTF-8",
  "host": "api.example.com",
  "x-amz-date": "20200401T001010Z",
  "X-Amz-Security-Token":
    "AgEXAMPLEZ21uX2VjEAoaDmFwLXNvdXR0ZWFEXAMPLECwRQIgAh97Cljq7w0PL8KsXP3YtDuyC/9hAj8PhJ7Fvf38SgoC
    +
    +pEagWCveZUjKE0zyUhBEXAMPLEjj/////////8BEXAMPLEx0Dk2NDgyNzg1NSIMo1mWnpESWUoYw4BkKqEFS1m3DXuL8
    +ZbVc4JKjDP4vUCKNR6Le9C9pZp9PsW0NoFy3vLBUDAXEXAMPLE0VG8feXfiEEA+1khgFK/
    wEtW+9zF7NaMMmSe07wN2gG2tH0eKMEXAMPLEEQX+sMbytQo8iepP9PZ0z1ZsSFb/
    dP5Q8hk6YEXAMPLEYcKZsTkDAq2uKFQ8mYUVA9EtQnNRiFLEY83aKvG/tqLWNnG1SNVx7SMcfovkFDqQamm
    +88y10wwAEYK7qcocX6Z7GGcaYuIfGpaX2MCCELeQvZ+8WxEg0nIfz7GYvsYNjLZSaRnV4G
    +ILY1F0QNW64S9Nvj
    +BwDg3ht2CrNvpwjVY1j9U3nmxE0UG5ne83LL5hhqMpm25kmL7enVgw2kQzmU2id4IKu0C/
    WaoDRu02F5zE63vJbxN8AYs7338+4B4HBb6BZ60Ugg96Q15RA41/
```

```
gIqxaVPxyTpDfTU5GfSLxocdYeniqqpFMtZG2n9d0u7GsQNcFkNcG3qDZm4tDo8tZbuym0a2VcF2E5hFEgXBa
+XLJCfXi/770qAEjP0x7Qdk3B43p8KG/BaioP5RsV8zBGvH1zAgyPha2rN70/
tT13yrmPd5QYefwzexjKrV4mWIuRg8NTHYSZJUaeyCwTom80VFUJXG
+GYTUyv5W22aBcnoRGiCiKEYTL0kgXecdKFTHmcIAejQ9We1r0a196Kq87w5KNMckcCGFnwBNFLmfnbpNqT6rUBxxs3X5nt
aox0FtHX21eF6qIGT8j1z+l2opU+ggwUgkhUUgCH2TfqBj+MLMVVvpgqJsPKt582caFKArIFiv0
+9QupxLnEH2hz04TMTfnU6bQC6z1buVe7h
+t0Lnh1YPFsLQ88anib/7TTC8k9DsBTq0ASe8R2GbSEsm09qbbMwgEaYUh0KtGeyQsSJdhSk6XxXThrWL9EnwBCXDkICMq
+WgtPtK00weDlCaRs3R2qXcbNgVhleMk4IwnF8D1695AenU1LwHj0JLkCjxgNFfiWAFEPH9aEXAMPLExA=="
  "Authorization": "AWS4-HMAC-SHA256 Credential=XXXXXXXXXXXXXXXXXXXX/20200401/
us-east-1/appsync/aws4_request, SignedHeaders=accept;content-
encoding;content-type;host;x-amz-date;x-amz-security-token,
Signature=83EXAMPLEebcc1fe3ee69f75cd5ebbf4cb4f150e4f99cec869f149c5EXAMPLEdc"
}
```

使用查詢字串請求 URL

```
wss://api.example.com/graphql?
header=eyJEXAMPLEHQiOiJhcHBsaWNhdGlvbi9qc29uLCB0ZXh0L2phdmFEXAMPLEQiLCJjb250ZW50LWVuY29kaW5nIjoE
```

Note

一個 WebSocket 連線可以有多個訂閱（即使使用不同的身分驗證模式）。其中一種實作方式是為第一個訂閱建立 WebSocket 連線，然後在最後一個訂閱取消註冊時將其關閉。您可以等待幾秒鐘再關閉 WebSocket 連線來最佳化此作業，以防應用程式在最後一個訂閱取消註冊後立即訂閱。對於行動應用程式範例，從一個畫面變更為另一個畫面時，在卸載事件時，會停止訂閱，而在掛載事件時，則會啟動不同的訂閱。

Lambda 授權

Lambda 授權標頭

標頭內容

- "Authorization": <string> : 以傳遞的值 authorizationToken。
- "host": <string> : AWS AppSync GraphQL 端點的主機或您的自訂網域名稱。

範例

```
{
```



```
{
  "Authorization": "M0UzQzM1MkQtMkI0Ni000TZCLUI1NkQtMUM0MTQ0QjVBRTczCkI1REEzRTIxLTk5NzItNDJENi1BQ
  "host": "example1234567890000.apps-sync-api.us-east-1.amazonaws.com"
}
```

透過查詢字串的標頭

首先，包含 `host` 和的 JSON 物件 `Authorization` 會轉換為字串。接下來，此字串會使用 `base64` 編碼進行編碼。產生的 `base64` 編碼字串會新增為名為 `WebSocket URL header` 的查詢參數，以建立與 AWS AppSync 即時端點的連線。產生的請求 URL 採用下列形式：

```
wss://example1234567890000.apps-sync-realtime-api.us-east-1.amazonaws.com/graphql?
header=eyJBdXRob3JpemF0aW9uIjoiZXlKcmFXUWlPaUpqYkc1eGIzQTV1VzVNSzA5UV1YSXJNVEpIV0VGTFNYQm11VTVX
```

請務必注意，除了 `base64` 編碼的標頭物件之外，空的 JSON 物件 `{}` 也是 `base64` 編碼的，並包含在 `WebSocket URL payload` 中做為名為 的個別查詢參數。

透過 的標頭 `Sec-WebSocket-Protocol`

包含 `host` 和的 JSON 物件 `Authorization` 會轉換為字串，然後使用 `base64Url` 編碼進行編碼。產生的 `base64Url-encoded` 字串字首為 `header-`。在與 AWS AppSync 即時端點建立 `WebSocket` 連線時，除了 `Sec-WebSocket-Protocol` 標頭 `graphql-ws` 中的 之外，此字首字串也會用作新的子通訊協定。

產生的請求 URL 採用下列形式：

```
wss://example1234567890000.apps-sync-realtime-api.us-east-1.amazonaws.com/graphql
```

`Sec-WebSocket-Protocol` 標頭包含下列值：

```
"sec-websocket-protocol" : ["graphql-ws", "header-
ewogICAgImhvc3QiOiJleGFtcGx1MTIzNDU2Nzg5MDAwMC5hcHBzeW5jLWFWaS51cy11YXN0LTEuYW1hem9uYXdzLmNvbSI
```

透過標準 HTTP 標頭的標頭

在此方法中，在建立與 AWS AppSync 即時端點的 `WebSocket` 連線時，會使用標準 HTTP 標頭傳輸主機和授權資訊。產生的請求 URL 採用下列形式：

```
wss://example1234567890000.apps-sync-realtime-api.us-east-1.amazonaws.com/graphql
```

請求標頭會包含下列項目：

```
"sec-websocket-protocol" : ["graphql-ws"]
"Authorization": "eyJEXAMPLEiJjbG5xb3A5eW5MK09QYXIrMTJHWEFLSXBieU5WNHhsQjEXAMPLEnM2WldvPSIsImFsZy9zEE2DJH7sH012zxYi7f-SmEGoh2AD8emxQRYajByz-rE4Jh0Q0ymN2Ys-ZIkMpVBTPgu-TMWdy0HhDUMUj20P82yeZ3w1ZAttr_gM4LzjXUXmI_K2yGjuXfXTaa1mvQEBG0mQfVd7SfwXB-jcv4RYVi6j25qgow9Ew52ufurPqaK-3WAKG32KpV8J4-Wejq8t0c-yA7sb8EnB551b7TU93uKRiVVK3E55Nk5ADPoam_WYE45i3s5qVAP_-InW75NUo0CGTsS8YWMfb6ecHYJ-1j-bzA27zaT9VjctXn9byNFZmEXAMPLExw",
"host": "example1234567890000.appsync-api.us-east-1.amazonaws.com"
```

即時 WebSocket 操作

使用 AWS AppSync 啟動成功的 WebSocket 交握後，用戶端必須傳送後續訊息，以連線至不同操作的 AWS AppSync。這些訊息需要下列資料：

- **type**：操作類型。
- **id**：訂閱的唯一識別符。我們建議您使用 UUID 來達到此目的。
- **payload**：相關聯的承載，取決於操作類型。

type 欄位是唯一的必要欄位；**id**和 **payload** 欄位為選用。

事件順序

若要成功啟動、建立、註冊和處理訂閱請求，用戶端必須逐步執行下列順序：

1. 初始化連線 (**connection_init**)
2. 連線確認 (**connection_ack**)
3. 訂閱註冊 (**start**)
4. 訂閱確認 (**start_ack**)
5. 正在處理訂閱 (**data**)
6. 取消註冊訂閱 (**stop**)

連線初始化訊息

(選用) 成功交握後，用戶端可以傳送訊息 **connection_init**，開始與 AWS AppSync 即時端點通訊。訊息是透過將 JSON 物件字串化來取得的字串，如下所示：

```
{ "type": "connection_init" }
```

連線確認訊息

傳送 `connection_init` 訊息後，用戶端必須等待 `connection_ack` 訊息。接收前傳送的所有訊息 `connection_ack` 都會遭到忽略。該訊息應該如下所示：

```
{
  "type": "connection_ack",
  "payload": {
    // Time in milliseconds waiting for ka message before the client should terminate
    // the WebSocket connection
    "connectionTimeoutMs": 300000
  }
}
```

保持啟用訊息

除了連線確認訊息之外，用戶端還會定期接收保持連線訊息。如果用戶端未在連線逾時期間內收到持續連線訊息，用戶端應關閉連線。AWS AppSync 會持續傳送這些訊息和服務已註冊的訂閱，直到其自動關閉連線為止 (24 小時後)。保持連線訊息是活動訊號，不需要用戶端來確認它們。

```
{ "type": "ka" }
```

訂閱註冊訊息

用戶端收到 `connection_ack` 訊息後，用戶端可以將訂閱註冊訊息傳送至 AWS AppSync。此類型的訊息是字串化的 JSON 物件，其中包含下列欄位：

- `"id": <string>`：訂閱的 ID。此 ID 對於每個訂閱必須是唯一的，否則伺服器會傳回錯誤，指出訂閱 ID 重複。
- `"type": "start"`：常數 `<string>` 參數。
- `"payload": <Object>`：包含訂閱相關資訊的物件。
 - `"data": <string>`：包含 GraphQL 查詢和變數的字串化 JSON 物件。
 - `"query": <string>`：GraphQL 操作。
 - `"variables": <Object>`：包含查詢變數的物件。
 - `"extensions": <Object>`：包含授權物件的物件。

- "authorization": <Object> : 包含授權所需欄位的物件。

訂閱註冊的授權物件

以 [AWS AppSync API 授權模式為基礎的標頭參數格式](#) 區段中的相同規則適用於授權物件。唯一的例外是 IAM，其中 SigV4 簽章資訊略有不同。如需詳細資訊，請參閱 IAM 範例。

使用 Amazon Cognito 使用者集區的範例：

```
{
  "id": "ee849ef0-cf23-4cb8-9fcb-152ae4fd1e69",
  "payload": {
    "data": "{\"query\":\"subscription onCreateMessage {\\n onCreateMessage {\\n __typename\\n message\\n }\\n }\", \"variables\":{}}",
    "extensions": {
      "authorization": {
        "Authorization":
"eyJEXAMPLEiJjbG5xb3A5eW5MK09QYXIrMTJEXAMPLEBieU5WNHhsQjhPVW9YMNm2WldvPSIsImFsZyI6I1EXAMPLEEn0.e
qTCtrYeboUJ4luRSTPXaNewNeEXAMPLE14C6sfg05t00f0MpiUwj9k19gtNCCMqoSsjtQoUweFnH4JYa5EXAMPLEVx0yQEQ
RWvW7yQU3sNQRLEXAMPLEcd0yufBiCYs3dfQxTTdvR1B6Wz6CD781fNeKqfzzUn2beMoup2h6EXAMPLE4ow8cUPUPvG0DzR
        "host": "example1234567890000.appsync-api.us-east-1.amazonaws.com"
      }
    }
  },
  "type": "start"
}
```

使用 IAM 的範例：

```
{
  "id": "eEXAMPLE-cf23-1234-5678-152EXAMPLE69",
  "payload": {
    "data": "{\"query\":\"subscription onCreateMessage {\\n onCreateMessage {\\n __typename\\n message\\n }\\n }\", \"variables\":{}}",
    "extensions": {
      "authorization": {
        "accept": "application/json, text/javascript",
        "content-type": "application/json; charset=UTF-8",
        "X-Amz-Security-Token":
"AgEXAMPLEZ2luX2VjEAoaDmFwLXNvdXR0ZWFEEXAMPLECwRQIgAh97C1jq7w0PL8KsxP3YtDuyC/9hAj8PhJ7Fvf38SgoC
+
+pEagWCveZUjKE0zyUhBEXAMPLEjj//////////8BEXAMPLEx0Dk2NDgyNzg1NSIMo1mWnpESWUoYw4BkKqEFSrm3DXuL8

```

```
+ZbVc4JKjDP4vUCKNR6Le9C9pZp9PsW0NoFy3vLBudAXEXAMPLE0VG8feXfiEEA+1khgFK/
wEtwR+9zF7NaMMmSe07wN2gG2tH0eKMEXAMPLEQX+sMbytQo8iepP9PZ0z1ZsSFb/
dP5Q8hk6YEXAMPLEYcKZsTkDAq2uKFQ8mYUVA9EtQnNRiFLEY83aKvG/tqLWNnG1SNVx7SMcfovkFDqQamm
+88y10wwAEYK7qcoceX6Z7G6GaYuIfGpaX2MCCELeQvZ+8WxEg0nIfz7GYvsYNjLZSaRnV4G
+ILY1F0QNW64S9Nvj
+BwDg3ht2CrNvpwjVYlJ9U3nmxE0UG5ne83LL5hhqMpm25kmL7enVgw2kQzmU2id4IKu0C/
WaoDRu02F5zE63vJbxN8AYs7338+4B4HBb6BZ60Ugg96Q15RA41/
gIqxaVPxyTpDfTU5GfSLxocdYeniqqpFMtZG2n9d0u7GsQNCfKNCg3qDZm4tDo8tZbuym0a2VcF2E5hFEgXBa
+XLJCfXi/770qAEjP0x7Qdk3B43p8KG/BaioP5RsV8zBGvH1zAgyPha2rN70/
tT13yrmPd5QYefwzexjKrV4mWIuRg8NTHYSZJUaeyCwTom80VFUJXG
+GYTUyv5W22aBcnoRGiCiKEYTL0kgXecdKFTHmcIAejQ9Welr0a196Kq87w5KNMCKcCGFnwBNFLmfmbpNqT6rUBxss3X5nt
aox0FtHX21eF6qIGT8j1z+l2opU+ggwUgkhUUgCH2TfqBj+MLMVvpvgqJsPKt582caFKArIFiv0
+9QupxLnEH2hz04TMTfnU6bQC6z1buVe7h
+t0Lnh1YPFsLQ88anib/7TTC8k9DsBTq0ASe8R2GbSEsm09qbbMwgEaYUh0KtGeyQsSJdhSk6XxXThrWL9EnwBCXDkICMqd
+WgtPtK00weDlCaRs3R2qXcbNgVhleMk4IwnF8D1695AenU1LwHj0JLkCjxgNFfiWAFEPH9aEXAMPLExA=="
    "Authorization": "AWS4-HMAC-SHA256 Credential=XXXXXXXXXXXXXXXXXXXX/20200401/
us-east-1/appsycn/aws4_request, SignedHeaders=accept;content-
encoding;content-type;host;x-amz-date;x-amz-security-token,
Signature=b90131a61a7c4318e1c35ead5dbfdeb46339a7585bbdbecceaff51f4022eb1fd",
    "content-encoding": "amz-1.0",
    "host": "example1234567890000.appsycn-api.us-east-1.amazonaws.com",
    "x-amz-date": "20200401T001010Z"
  }
},
"type": "start"
}
```

使用自訂網域名稱的範例：

```
{
  "id": "key-cf23-4cb8-9fcb-152ae4fd1e69",
  "payload": {
    "data": "{\"query\":\"subscription onCreateMessage {\n\n onCreateMessage {\n\n\n__typename\n message\n }\n\n }\", \"variables\": {}}",
    "extensions": {
      "authorization": {
        "x-api-key": "da2-12345678901234567890123456",
        "host": "api.example.com"
      }
    }
  },
  "type": "start"
}
```

```
}
```

SigV4 簽章不需要/connect附加至 URL，而且 JSON 字串化 GraphQL 操作會取代 data。以下是 SigV4 簽章請求的範例：

```
{
  url: "https://example1234567890000.appsync-api.us-east-1.amazonaws.com/graphql",
  data: "{\"query\":\"subscription onCreateMessage {\n onCreateMessage {\n __typename\n message\n }\n }\",\"variables\":{}}",
  method: "POST",
  headers: {
    "accept": "application/json, text/javascript",
    "content-encoding": "amz-1.0",
    "content-type": "application/json; charset=UTF-8",
  }
}
```

訂閱確認訊息

傳送訂閱開始訊息後，用戶端應等待 AWS AppSync 傳送訊息start_ack。start_ack 訊息指出訂閱成功。

訂閱確認範例：

```
{
  "type": "start_ack",
  "id": "eEXAMPLE-cf23-1234-5678-152EXAMPLE69"
}
```

錯誤訊息

如果連線初始化或訂閱註冊失敗，或訂閱從伺服器結束，伺服器會傳送錯誤訊息給用戶端。如果在連線初始化期間發生錯誤，伺服器將會關閉連線。

- "type": "error"：常數 <string> 參數。
- "id": <string>：相應已註冊訂閱的 ID，如果相關的話。
- "payload" <Object>：包含對應錯誤資訊的物件。

範例：

```
{
  "type": "error",
  "payload": {
    "errors": [
      {
        "errorType": "LimitExceededError",
        "message": "Rate limit exceeded"
      }
    ]
  }
}
```

處理資料訊息

當用戶端提交變動時，AWS AppSync 會識別對其感興趣的所有訂閱者，並使用訂閱操作中對應的訂閱傳送訊息 "type": "data" 給每個 id "start" 訂閱者。預期用戶端會追蹤 id 其傳送的訂閱，以便在接收資料訊息時，用戶端可以比對對應的訂閱。

- "type": "data" : 常數 <string> 參數。
- "id": <string> : 對應已註冊訂閱的 ID。
- "payload" <Object> : 包含訂閱資訊的物件。

範例：

```
{
  "type": "data",
  "id": "ee849ef0-cf23-4cb8-9fcb-152ae4fd1e69",
  "payload": {
    "data": {
      "onCreateMessage": {
        "__typename": "Message",
        "message": "test"
      }
    }
  }
}
```

訂閱取消註冊訊息

當應用程式想要停止監聽訂閱事件時，用戶端應該使用下列字串化 JSON 物件傳送訊息：

- "type": "stop" : 常數 <string> 參數。
- "id": <string> : 要取消註冊的訂閱 ID。

範例：

```
{
  "type": "stop",
  "id": "ee849ef0-cf23-4cb8-9fcb-152ae4fd1e69"
}
```

AWS AppSync 會使用下列字串化 JSON 物件傳回確認訊息：

- "type": "complete" : 常數 <string> 參數。
- "id": <string> : 未註冊訂閱的 ID。

用戶端收到確認訊息後，就不會再收到此特定訂閱的訊息。

範例：

```
{
  "type": "complete",
  "id": "eEXAMPLE-cf23-1234-5678-152EXAMPLE69"
}
```

斷開 WebSocket 的連線

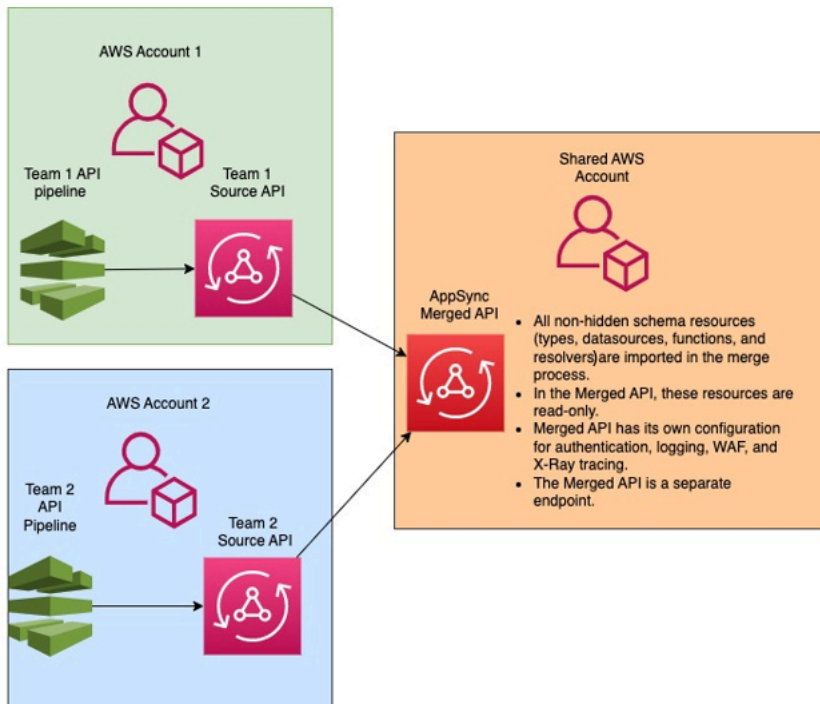
在中斷連線之前，為了避免資料遺失，用戶端應具有必要的邏輯，以檢查目前沒有透過 WebSocket 連線進行的操作。從 WebSocket 中斷連線之前，所有訂閱都應該取消註冊。

在 中合併 APIs AWS AppSync

隨著 GraphQL 的使用在組織內擴展，API ease-of-use API 開發速度之間可能會出現取捨。一方面，組織採用 AWS AppSync 和 GraphQL 來簡化應用程式開發，為開發人員提供靈活的 API，他們可以使用單一網路呼叫安全地存取、操作和結合來自一或多個資料網域的資料。另一方面，組織內負責將不同資料網域合併到單一 GraphQL API 端點的團隊，可能需要能夠獨立建立、管理和部署 API 更新，以增加其開發速度。

為了解決這種壓力，AWS AppSync 合併 APIs 功能允許來自不同資料網域的團隊獨立建立和部署 AWS AppSync APIs (例如 GraphQL 結構描述、解析程式、資料來源和函數)，然後可以合併為單一

合併 API。這可讓組織維持簡單易用的跨網域 API，以及讓不同團隊能夠快速獨立進行 API 更新的方式。



使用合併 APIs，組織可以將多個獨立來源 AWS AppSync APIs 的資源匯入單一 AWS AppSync 合併 API 端點。若要這樣做，AWS AppSync 可讓您建立 AWS AppSync 來源 APIs 的清單，然後將與來源 APIs 相關聯的所有中繼資料，包括結構描述、類型、資料來源、解析程式和函數，合併到新的 AWS AppSync 合併 API。

在合併期間，可能會因為來源 API 資料內容中的不一致而發生合併衝突，例如在合併多個結構描述時類型命名衝突。對於來源 APIs 衝突的簡單使用案例，不需要修改來源 API 結構描述。產生的合併 API 只會從原始 AWS AppSync APIs 匯入所有類型、解析程式、資料來源和函數。對於發生衝突的複雜使用案例，使用者/團隊必須透過各種方式解決衝突。為使用者 AWS AppSync 提供數種工具和範例，可以減少合併衝突。

在 中設定的後續合併 AWS AppSync 會將來源 APIs 中所做的變更傳播到相關聯的合併 API。

合併 APIs 和聯合

GraphQL 社群中有許多解決方案和模式，可用於合併 GraphQL 結構描述並透過共用圖形實現團隊協作。AWS AppSync 合併 APIs 採用結構描述合成的建置時間方法，其中來源 APIs 會合併為單獨的合併 API。另一種方法是將多個來源 APIs 或子圖形的執行時間路由器分層。在此方法中，路由器會接收請

求、參考其維護為中繼資料的合併結構描述、建構請求計劃，然後將請求元素分佈至其基礎子圖/伺服器。下表會將 AWS AppSync 合併 API 建置時間方法與路由器型執行時間方法與 GraphQL 結構描述合成進行比較：

Feature	AppSync Merged API	Router-based solutions
Sub-graphs managed independently	Yes	Yes
Sub-graphs addressable independently	Yes	Yes
Automated schema composition	Yes	Yes
Automated conflict detection	Yes	Yes
Conflict resolution via schema directives	Yes	Yes
Supported sub-graph servers	AWS AppSync*	Varies
Network complexity	Single, merged API means no extra network hops.	Multi-layer architecture requires query planning and delegation, sub-query parsing and serialization/deserialization, and reference resolvers in sub-graphs to perform joins.
Observability support	Built-in monitoring, logging, and tracing. A single, Merged API server means simplified debugging.	Build-your-own observability across router and all associated sub-graph servers. Complex debugging across distributed system.
Authorization support	Built in support for multiple authorization modes.	Build-your-own authorization rules.

Cross account security	Built-in support for cross-AWS cloud account associations.	Build-your-own security model.
Subscriptions support	Yes	No

* AWS AppSync 合併 APIs 只能與 AWS AppSync 來源 APIs 建立關聯。如果您需要跨 AWS AppSync 和非 AWS AppSync 子圖形結構描述合成的支援，您可以將一或多個 AWS AppSync GraphQL 和/或合併 APIs 連接到以路由器為基礎的解決方案。例如，請參閱參考部落格，以使用具有 Apollo Federation v2：[Apollo GraphQL Federation AWS AppSync](#) 的路由器架構將 AWS AppSync APIs 新增為子圖。

主題

- [合併 API 衝突解決](#)
- [設定結構描述](#)
- [設定授權模式](#)
- [設定執行角色](#)
- [使用 設定跨帳戶合併 APIs AWS RAM](#)
- [合併](#)
- [合併 APIs 的其他支援](#)
- [合併 API 限制](#)
- [建立合併 APIs](#)

合併 API 衝突解決

發生合併衝突時，AWS AppSync 會提供使用者數個工具和範例，以協助對問題進行故障診斷（疑難排解）。

合併 API 結構描述指令

AWS AppSync 引進數個 GraphQL 指令，可用於減少或解決來源 APIs 之間的衝突：

- **@canonical**：此指令會設定具有類似名稱和資料的類型/欄位優先順序。如果兩個或多個來源 APIs 具有相同的 GraphQL 類型或欄位，則其中一個 APIs 可以將其類型或欄位標註為正式，這會在合併期間排定優先順序。合併時，會忽略其他來源 APIs 中未以此指令標註的衝突類型/欄位。

- `@hidden`：此指令會封裝特定類型/欄位，以將其從合併程序中移除。團隊可能想要移除或隱藏來源 API 中的特定類型或操作，因此只有內部用戶端可以存取特定類型資料。附加此指令後，類型或欄位不會合併到合併 API。
- `@renamed`：此指令會變更類型/欄位的名稱，以減少命名衝突。在某些情況下，不同的 APIs 具有相同的類型或欄位名稱。不過，它們都需要在合併的結構描述中可用。將它們全部包含在合併 API 中的簡單方法是將欄位重新命名為類似但不同的欄位。

若要顯示公用程式結構描述指令，請考慮下列範例：

在此範例中，假設我們想要合併兩個來源 APIs。我們獲得兩個結構描述來建立和擷取文章（例如評論區段或社交媒體文章）。假設類型和欄位非常類似，在合併操作期間發生衝突的機率很高。以下程式碼片段顯示每個結構描述的類型和欄位。

第一個檔案名為 `Source1.graphql`，是 GraphQL 結構描述，允許使用者 `Posts` 使用 `putPost` 變動建立。每個 `Post` 都包含標題和 ID。ID 用於參考 `User` 或張貼者的資訊（電子郵件和地址）`Message`，以及或承載（內容）。`User` 類型會以 `@canonical` 標籤標註。

```
# This snippet represents a file called Source1.graphql

type Mutation {
  putPost(id: ID!, title: String!): Post
}

type Post {
  id: ID!
  title: String!
}

type Message {
  id: ID!
  content: String
}

type User @canonical {
  id: ID!
  email: String!
  address: String!
}

type Query {
  singlePost(id: ID!): Post
}
```

```

    getMessage(id: ID!): Message
  }

```

第二個檔案名為 `Source2.graphql`，是 GraphQL 結構描述，其作用與 `Source1.graphql` 非常相似。不過請注意，每種類型的欄位都不同。合併這兩個結構描述時，會因為這些差異而發生合併衝突。

另請注意 `Source2.graphql` 如何也包含數個指令來減少這些衝突。`Post` 類型會以 `@hidden` 標籤標註，以在合併操作期間混淆自己。`Message` 類型會以 `@renamed` 標籤標註，以便在命名與其他 `Message` 類型衝突 `ChatMessage` 時將類型名稱修改為。

```

# This snippet represents a file called Source2.graphql

type Post @hidden {
  id: ID!
  title: String!
  internalSecret: String!
}

type Message @renamed(to: "ChatMessage") {
  id: ID!
  chatId: ID!
  from: User!
  to: User!
}

# Stub user so that we can link the canonical definition from Source1
type User {
  id: ID!
}

type Query {
  getPost(id: ID!): Post
  getMessage(id: ID!): Message @renamed(to: "getChatMessage")
}

```

合併發生時，結果會產生 `MergedSchema.graphql` 檔案：

```

# This snippet represents a file called MergedSchema.graphql

type Mutation {
  putPost(id: ID!, title: String!): Post
}

```

```
# Post from Source2 was hidden so only uses the Source1 definition.
type Post {
  id: ID!
  title: String!
}

# Renamed from Message to resolve the conflict
type ChatMessage {
  id: ID!
  chatId: ID!
  from: User!
  to: User!
}

type Message {
  id: ID!
  content: String
}

# Canonical definition from Source1
type User {
  id: ID!
  email: String!
  address: String!
}

type Query {
  singlePost(id: ID!): Post
  getMessage(id: ID!): Message

  # Renamed from getMessage
  getChatMessage(id: ID!): ChatMessage
}
```

合併中發生了幾件事：

- 由於 `@canonical` 註釋，Source1.graphql 的 User 類型優先於 User Source2.graphql 的。
- Source1.graphql 的 Message 類型包含在合併中。Source1 不過，Message 來自 Source2.graphql 的有命名衝突。由於其 `@renamed` 註釋，它也包含在合併中，但具有替代名稱 ChatMessage。

- 包含 Source1.graphql 的 Post 類型，但不包含 Source2.graphql 的 Post 類型。一般而言，此類型會發生衝突，但由於 Source2.graphql 的 Post 類型具有 @hidden 註釋，因此其資料會混淆，且不包含在合併中。這不會造成任何衝突。
- 已更新 Query 類型，以包含兩個檔案的內容。不過，GetChatMessage 由於 指令，一個 GetMessage 查詢已重新命名為 。這解決了兩個具有相同名稱的查詢之間的命名衝突。

也有未將任何指令新增至衝突類型的情況。在這裡，合併類型將包含該類型所有來源定義中所有欄位的聯集。例如，請考量下列範例：

此結構描述稱為 Source1.graphql，允許建立和擷取 Posts。組態類似於先前的範例，但資訊較少。

```
# This snippet represents a file called Source1.graphql

type Mutation {
  putPost(id: ID!, title: String!): Post
}

type Post {
  id: ID!
  title: String!
}

type Query {
  getPost(id: ID!): Post
}
```

此結構描述稱為 Source2.graphql，允許建立和擷取 Reviews（例如電影評分或餐廳評論）。Reviews 與相同 ID 值 Post 的相關聯。它們一起包含完整檢閱文章的標題、文章 ID 和承載訊息。

合併時，這兩種 Post 類型之間會發生衝突。由於沒有註釋可解決此問題，預設行為是對衝突的類型執行聯集操作。

```
# This snippet represents a file called Source2.graphql

type Mutation {
  putReview(id: ID!, postId: ID!, comment: String!): Review
}

type Post {
  id: ID!
```

```
    reviews: [Review]
  }

  type Review {
    id: ID!
    postId: ID!
    comment: String!
  }

  type Query {
    getReview(id: ID!): Review
  }
```

合併發生時，結果會產生 `MergedSchema.graphql` 檔案：

```
# This snippet represents a file called MergedSchema.graphql

type Mutation {
  putReview(id: ID!, postId: ID!, comment: String!): Review
  putPost(id: ID!, title: String!): Post
}

type Post {
  id: ID!
  title: String!
  reviews: [Review]
}

type Review {
  id: ID!
  postId: ID!
  comment: String!
}

type Query {
  getPost(id: ID!): Post
  getReview(id: ID!): Review
}
```

合併中發生了幾件事：

- `Mutation` 類型沒有衝突且已合併。

- Post 類型欄位透過聯集操作合併。請注意這兩個 之間的聯集如何產生單一 id、title 和單一 reviews。
- Review 類型沒有衝突且已合併。
- Query 類型沒有衝突且已合併。

在共用類型上管理解析程式

在上述範例中，請考慮 Source1.graphql 已在 上設定單位解析程式的情況 Query.getPost，該解析程式使用名為 的 DynamoDB 資料來源 PostDataSource。此解析程式會傳回 id 和 title Post 類型的。現在，請考慮 Source2.graphql 已在 上設定管道解析程式 Post.reviews，其會執行兩個 函數。Function1 已連接 None 資料來源以執行自訂授權檢查。Function2 已連接 DynamoDB 資料來源以查詢 reviews 資料表。

```
query GetPostQuery {
  getPost(id: "1") {
    id,
    title,
    reviews
  }
}
```

當用戶端對合併 API 端點執行上述查詢時，AWS AppSync 服務會先 Query.getPost 從 執行的單位解析程式 Source1，呼叫 PostDataSource 並從 DynamoDB 傳回資料。然後，它會執行 Post.reviews 管道解析程式，其中 會 Function1 執行自訂授權邏輯，並 Function2 傳回在 id 中找到的評論 \$context.source。服務會將請求視為單一 GraphQL 執行來處理，而此簡單請求只需要單一請求字符。

管理共用類型的解析程式衝突

在下列情況下，我們也會在 上實作解析程式，Query.getPost 以便一次提供超過 中欄位解析程式的多個欄位 Source2。Source1.graphql 可能如下所示：

```
# This snippet represents a file called Source1.graphql

type Post {
  id: ID!
  title: String!
  date: AWSDateTime!
}
```

```
type Query {  
  getPost(id: ID!): Post  
}
```

Source2.graphql 可能如下所示：

```
# This snippet represents a file called Source2.graphql  
  
type Post {  
  id: ID!  
  content: String!  
  contentHash: String!  
  author: String!  
}  
  
type Query {  
  getPost(id: ID!): Post  
}
```

嘗試合併這兩個結構描述會產生合併錯誤，因為 AWS AppSync 合併 APIs 不允許多個來源解析程式連接到相同的欄位。為了解決此衝突，您可以實作需要 Source2.graphql 新增個別類型的欄位解析程式模式，以定義其從 Post 類型擁有的欄位。在下列範例中，我們新增名為 `PostInfo` 的類型，其中包含由 Source2.graphql 解析的內容和作者欄位。Source1.graphql 將實作連接至 `Query.getPost` 的解析程式，而 Source2.graphql 現在會將解析程式連接至 `Post.postInfo`，以確保可成功擷取所有資料：

```
type Post {  
  id: ID!  
  postInfo: PostInfo  
}  
  
type PostInfo {  
  content: String!  
  contentHash: String!  
  author: String!  
}  
  
type Query {  
  getPost(id: ID!): Post  
}
```

雖然解決這類衝突需要重寫來源 API 結構描述，而且可能還需要用戶端變更其查詢，但這種方法的優點是合併解析程式的擁有權在來源團隊之間仍然很明確。

設定結構描述

雙方負責設定結構描述以建立合併 API：

- 合併 API 擁有者 - 合併 API 擁有者必須設定合併 API 的授權邏輯和進階設定，例如記錄、追蹤、快取和 WAF 支援。
- 關聯的來源 API 擁有者 - 關聯的 API 擁有者必須設定組成合併 API 的結構描述、解析程式和資料來源。

由於合併 API 的結構描述是從相關聯來源 APIs，因此僅供讀取。這表示必須在來源 APIs 中啟動結構描述的變更。在 AWS AppSync 主控台中，您可以使用結構描述視窗上方的下拉式清單，在合併結構描述與合併 APIs 中包含的來源 API 的個別結構描述之間切換。

設定授權模式

有多種授權模式可用於保護您的合併 API。若要進一步了解 中的授權模式 AWS AppSync，請參閱[授權和身分驗證](#)。

下列授權模式可用於合併 APIs：

- API 金鑰：最簡單的授權策略。所有請求都必須在 `x-api-key` 請求標頭下包含 API 金鑰。過期的 API 金鑰會在過期日期後保留 60 天。
- AWS Identity and Access Management (IAM)：IAM 授權策略會授權所有已簽署 `sigv4` 的請求。AWS
- Amazon Cognito 使用者集區：透過 Amazon Cognito 使用者集區授權您的使用者，以實現更精細的控制。
- AWS Lambda 授權方：一種無伺服器函數，可讓您使用自訂邏輯來驗證和授權對 AWS AppSync API 的存取。
- OpenID Connect：此授權類型會強制執行 OIDC 相容服務提供的 OpenID Connect (OIDC) 字符。您的應用程式可以運用由 OIDC 提供者定義的使用者與權限來控制存取權限。

合併 API 的授權模式是由合併 API 擁有者設定。在合併操作時，合併 API 必須包含在來源 API 上設定的主要授權模式，做為其本身的主要授權模式或次要授權模式。否則，它會不相容，合併操作會失敗並

產生衝突。在來源 APIs 中使用多重驗證指令時，合併程序能夠將這些指令自動合併到統一的端點。如果來源 API 的主要授權模式不符合合併 API 的主要授權模式，則會自動新增這些身分驗證指令，以確保來源 API 中類型的授權模式一致。

設定執行角色

建立合併 API 時，您需要定義服務角色。AWS 服務角色是 AWS 服務用來代表您執行任務的 AWS Identity and Access Management (IAM) 角色。

在這種情況下，您的合併 API 必須執行解析程式，才能從來源 APIs 中設定的資料來源存取資料。此所需的服務角色是 `mergedApiExecutionRole`，而且必須具有明確存取權，才能透過 IAM 許可在合併 APIs 包含的來源 API `appsync:SourceGraphQL` 上執行請求。在執行 GraphQL 請求期間，AWS AppSync 服務將擔任此服務角色，並授權該角色執行 `appsync:SourceGraphQL` 動作。

AWS AppSync 支援允許或拒絕請求中特定最上層欄位的此許可，例如 IAM 授權模式對 IAM APIs 的運作方式。對於 non-top-level 欄位，AWS AppSync 會要求您定義來源 API ARN 本身的許可。為了限制對合併 API 中特定 non-top-level 欄位的存取，我們建議您在 Lambda 中實作自訂邏輯，或使用 `@hidden` 指令從合併 API 隱藏來源 API 欄位。如果您想要允許角色在來源 API 內執行所有資料操作，您可以新增以下政策。請注意，第一個資源項目允許存取所有最上層欄位，第二個項目涵蓋在來源 API 資源本身上授權的子解析程式：

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [{
    "Effect": "Allow",
    "Action": [ "appsync:SourceGraphQL" ],
    "Resource": [
      "arn:aws:appsync:us-west-2:123456789012:apis/YourSourceGraphQLApiId/*",
      "arn:aws:appsync:us-west-2:123456789012:apis/YourSourceGraphQLApiId" ]
  }]
}
```

如果只想將存取權限制在特定最上層欄位，您可以使用如下所示的政策：

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [{
    "Effect": "Allow",
    "Action": [ "appsync:SourceGraphQL" ],
    "Resource": [
      "arn:aws:appsync:us-west-2:123456789012:apis/YourSourceGraphQLApiId/
types/Query/fields/<Field-1>",
      "arn:aws:appsync:us-west-2:123456789012:apis/
YourSourceGraphQLApiId" ]
    }]
}
```

您也可以使用 AWS AppSync 主控台 API 建立精靈來產生服務角色，以允許合併 API 存取在與合併 API 位於相同帳戶中 APIs 中設定的資源。如果您的來源 APIs 與合併的 API 不在同一帳戶中，您必須先使用 AWS Resource Access Manager () 共用資源 AWS RAM。

使用 設定跨帳戶合併 APIs AWS RAM

當您建立合併 API 時，您可以選擇將來源 APIs 與透過 AWS Resource Access Manager () 共用的其他帳戶建立關聯 AWS RAM。AWS RAM 可協助您跨 AWS 帳戶、組織或組織單位 (OUs) 以及 IAM 角色和使用者安全地共用資源。

AWS AppSync 與 整合，AWS RAM 以支援從單一合併 APIs 跨多個帳戶設定和存取來源 API。AWS RAM 可讓您建立資源共用或資源容器，以及將針對每個資源共用的許可集。您可以將 AWS AppSync APIs 新增至資源共用 AWS RAM。在資源共用中，AWS AppSync 提供三種不同的許可集，可與 RAM 中的 AWS AppSync API 相關聯：

1. **AWSRAMPermissionAppSyncSourceApiOperationAccess**：AWS RAM 如果未指定其他許可，則在 中共用 AWS AppSync API 時新增的預設許可集。此許可集用於與合併 AWS AppSync API 擁有者共用來源 API。此許可集包含來源 API `appsync:AssociateMergedGraphQLApi` 上的許可，以及在執行時間存取來源 API 資源所需的 `appsync:SourceGraphQL` 許可。
2. **AWSRAMPermissionAppSyncMergedApiOperationAccess**：與來源 API 擁有者共用合併 API 時，應設定此許可集。此許可集可讓來源 API 設定合併 API，包括將目標主體擁有的任何來源 APIs 與合併 API 建立關聯，以及讀取和更新合併 API 的來源 API 關聯。

3. `AWSRAMPermissionAppSyncAllowSourceGraphQLAccess`：此許可集允許 `appsync:SourceGraphQL` 許可與 AWS AppSync API 搭配使用。它旨在用於與合併 API 擁有者共用來源 API。與來源 API 操作存取的預設許可集相反，此許可集僅包含執行時間許可 `appsync:SourceGraphQL`。如果使用者選擇將合併 API 操作存取權分享給來源 API 擁有者，他們也需要從來源 API 分享此許可給合併 API 擁有者，才能透過合併 API 端點取得執行時間存取權。

AWS AppSync 也支援客戶受管許可。當其中一個提供的 AWS 受管許可無法運作時，您可以建立自己的客戶受管許可。客戶受管許可是您撰寫和維護的受管許可，方法是精確指定可在使用共用資源的條件下執行哪些動作 AWS RAM。AWS AppSync 可讓您在建立自己的許可時，從下列動作中進行選擇：

1. `appsync:AssociateSourceGraphqlApi`
2. `appsync:AssociateMergedGraphqlApi`
3. `appsync:GetSourceApiAssociation`
4. `appsync:UpdateSourceApiAssociation`
5. `appsync:StartSchemaMerge`
6. `appsync:ListTypesByAssociation`
7. `appsync:SourceGraphQL`

一旦您在 中正確共用來源 API 或合併 API，AWS RAM 並且在必要時已接受資源共用邀請，當您在合併 API 上建立或更新來源 API 關聯時，就會在 AWS AppSync 主控台中顯示。您也可以透過呼叫 提供 `ListGraphqlApis` 的操作 AWS AppSync 並使用 `OTHER_ACCOUNTS` 擁有者篩選條件，列出使用 AWS RAM 與您的帳戶共用的所有 AWS AppSync APIs，無論許可設定為何。

 Note

透過 共用 AWS RAM 需要 中的發起人 AWS RAM 具有許可，才能對正在共用的任何 API 執行 `appsync:PutResourcePolicy` 動作。

合併

管理合併

合併 APIs 旨在支援團隊在統一 AWS AppSync 端點上的協作。團隊可以在後端獨立發展自己的隔離來源 GraphQL APIs，同時 AWS AppSync 服務會管理資源與單一合併 API 端點的整合，以減少協作中的摩擦並縮短開發前置時間。

自動合併

與 AWS AppSync 合併 APIs 相關聯的來源 API 可以設定為在對來源 API 進行任何變更後自動合併（自動合併）到合併 API。這可確保來源 API 的變更一律傳播到背景的合併 API 端點。來源 API 結構描述中的任何變更都會在合併 API 中更新，只要它不會與合併 API 中的現有定義發生合併衝突。如果來源 API 中的更新正在更新解析程式、資料來源或函數，也會更新匯入的資源。當引入無法自動解決（自動解決）的新衝突時，合併 API 結構描述更新會因為合併操作期間不支援的衝突而被拒絕。錯誤訊息可在主控台中針對狀態為 的每個來源 API 關聯使用 `MERGE_FAILED`。您也可以使用 AWS SDK 或使用 CLI 呼叫指定來源 API AWS 關聯的 `GetSourceApiAssociation` 操作來檢查錯誤訊息，如下所示：

```
aws appsync get-source-api-association --merged-api-identifier <Merged API ARN> --
association-id <SourceApiAssociation id>
```

這將產生以下格式的結果：

```
{
  "sourceApiAssociation": {
    "associationId": "<association id>",
    "associationArn": "<association arn>",
    "sourceApiId": "<source api id>",
    "sourceApiArn": "<source api arn>",
    "mergedApiArn": "<merged api arn>",
    "mergedApiId": "<merged api id>",
    "sourceApiAssociationConfig": {
      "mergeType": "MANUAL_MERGE"
    },
    "sourceApiAssociationStatus": "MERGE_FAILED",
    "sourceApiAssociationStatusDetail": "Unable to resolve conflict on object with
name title: Merging is not supported for fields with different types."
  }
}
```

手動合併

來源 API 的預設設定是手動合併。若要合併自上次更新合併 API 以來在來源 APIs 中發生的任何變更，來源 API 擁有者可以從 AWS AppSync 主控台或透過 AWS SDK 和 CLI AWS 中可用的 `StartSchemaMerge` 操作叫用手動合併。

合併 APIs 的其他支援

設定訂閱

與以路由器為基礎的 GraphQL 結構描述合成方法不同，AWS AppSync 合併 APIs 為 GraphQL 訂閱提供內建支援。您關聯來源 APIs 中定義的所有訂閱操作都會在合併 API 中自動合併和運作，無需修改。若要進一步了解如何透過無伺服器 WebSockets 連線 AWS AppSync 支援訂閱，請參閱 [即時資料](#)。

設定可觀測性

AWS AppSync 合併 APIs 透過 [Amazon CloudWatch](#) 提供內建記錄、監控和指標。AWS AppSync 也提供透過追蹤的內建支援 [AWS X-Ray](#)。

設定自訂網域

AWS AppSync 合併 APIs 提供內建支援，讓您將自訂網域與合併 API 的 [GraphQL 和即時端點](#) 搭配使用。

設定快取

AWS AppSync 合併 APIs 提供內建支援，可選擇性地快取請求層級和/或解析程式層級的回應，以及回應壓縮。若要進一步了解，請參閱 [快取和壓縮](#)。

設定私有 APIs

AWS AppSync 合併 APIs 提供私有 APIs 的內建支援，可將對合併 API 的 GraphQL 和即時端點的存取限制為來自 [您可以設定的 VPC 端點的](#) 流量。

設定防火牆規則

AWS AppSync 合併 APIs 提供的內建支援 AWS WAF，可讓您透過定義 [Web 應用程式防火牆規則](#) 來保護 APIs。

設定稽核日誌

AWS AppSync 合併 APIs 提供 AWS CloudTrail 的內建支援，可讓您 [設定和管理稽核日誌](#)。

合併 API 限制

開發合併 APIs 時，請注意下列規則：

1. 合併 API 不能是另一個合併 API 的來源 API。
2. 來源 API 無法與多個合併 API 建立關聯。
3. 合併 API 結構描述文件的預設大小限制為 10 MB。
4. 可與合併 APIs 相關聯的來源 API 預設數量為 10。不過，如果您在合併 API 中需要超過 10 APIs，您可以請求提高限制。

建立合併 APIs

在主控台中建立合併 API

1. 登入 AWS Management Console 並開啟 [AWS AppSync 主控台](#)。
 - 在儀表板中，選擇建立 API。
 2. 選擇合併 API，然後選擇下一步。
 3. 在指定 API 詳細資訊頁面中，輸入下列資訊：
 - a. 在 API 詳細資訊下，輸入下列資訊：
 - i. 指定合併 API 的 API 名稱。此欄位是標記 GraphQL API 的方式，可方便地將其與其他 GraphQL APIs 區分開來。
 - ii. 指定聯絡人詳細資訊。此欄位是選用的，可將名稱或群組連接至 GraphQL API。它不會連結到其他資源或由其他資源產生，運作方式很類似 API 名稱欄位。
 - b. 在服務角色下，您必須將 IAM 執行角色連接至合併的 API，讓 AWS AppSync 可以在執行時間安全地匯入和使用您的資源。您可以選擇建立和使用新的服務角色，這可讓您指定 AWS AppSync 將使用的政策和資源。您也可以匯入現有的 IAM 角色，方法是選擇使用現有的服務角色，然後從下拉式清單中選取角色。
 - c. 在私有 API 組態下，您可以選擇啟用私有 API 功能。請注意，此選項無法在建立合併的 API 之後變更。如需私有 APIs 的詳細資訊，請參閱 [使用 AWS AppSync 私有 APIs](#)。
- 完成後，請選擇下一步。
4. 接著，您必須新增將用作合併 APIs 基礎的 GraphQL API。在選取來源 APIs 頁面中，輸入下列資訊：

- a. 在 AWS 帳戶資料表的 APIs 中，選擇新增來源 APIs。在 GraphQL APIs 清單中，每個項目都會包含下列資料：
 - i. 名稱：GraphQL API 的 API 名稱欄位。
 - ii. API ID：GraphQL API 的唯一 ID 值。
 - iii. 主要身分驗證模式：GraphQL API 的預設授權模式。如需 中授權模式的詳細資訊 AWS AppSync，請參閱[授權和身分驗證](#)。
 - iv. 額外身分驗證模式：GraphQL API 中設定的次要授權模式。
 - v. 透過選取 APIs 名稱欄位旁的核取方塊，選擇您將在合併 API 中使用的 API。之後，選擇新增來源 APIs。選取的 GraphQL APIs 會顯示在您 AWS 帳戶資料表 APIs 中。
- b. 在來自其他 AWS 帳戶的 APIs 表格中，選擇新增來源 APIs。此清單中的 GraphQL APIs 來自透過 AWS Resource Access Manager () 將資源分享給您的其他帳戶 AWS RAM。在此資料表中選取 GraphQL APIs 的程序與上一節的程序相同。如需透過 共用資源的詳細資訊 AWS RAM，請參閱[什麼是 AWS Resource Access Manager ?](#)。

完成後，請選擇下一步。

- c. 新增您的主要身分驗證模式。如需詳細資訊，請參閱[授權和身分驗證](#)。選擇下一步。
- d. 檢閱您的輸入，然後選擇建立 API。

使用 RDS 自我檢查建置 GraphQL APIs

AWS AppSync 的自我檢查公用程式可以從資料庫資料表探索模型，並提議 GraphQL 類型。AWS AppSync 主控台的建立 API 精靈可以立即從 Aurora MySQL 或 PostgreSQL 資料庫產生 API。它會自動建立類型和 JavaScript 解析程式來讀取和寫入資料。

AWS AppSync 透過 Amazon RDS Data API 提供與 Amazon Aurora 資料庫的直接整合。Amazon RDS Data API 提供安全 HTTP 端點，AWS AppSync 可連線至 以執行 SQL 陳述式，而不需要持久性資料庫連線。您可以使用此項目，為 Aurora 上的 MySQL 和 PostgreSQL 工作負載建立關聯式資料庫 API。

使用 為關聯式資料庫建置 API AWS AppSync 有幾個優點：

- 您的資料庫不會直接公開給用戶端，將存取點與資料庫本身解耦。
- 您可以建置專為不同應用程式需求 APIs 量身打造的 API，無需前端自訂商業邏輯。這符合 Backend-For-Frontend (BFF) 模式。

- 您可以使用各種授權模式在 AWS AppSync layer 實作授權和存取控制，以控制存取。不需要額外的運算資源即可連線至資料庫，例如託管 Web 伺服器或代理連線。
- 您可以透過訂閱新增即時功能，並透過 AppSync 進行的資料變動會自動推送至連線的用戶端。
- 用戶端可以透過 HTTPS 使用 443 等常見連接埠連線至 API。

AWS AppSync 讓從現有關聯式資料庫建置 APIs 變得簡單。其自我檢查公用程式可以從資料庫資料表探索模型，並提議 GraphQL 類型。主控台的建立 API AWS AppSync 精靈可以立即從 Aurora MySQL 或 PostgreSQL 資料庫產生 API。它會自動建立類型和 JavaScript 解析程式來讀取和寫入資料。

AWS AppSync 提供整合的 JavaScript 公用程式，可簡化在解析程式中寫入 SQL 陳述式的程序。您可以使用 AWS AppSync 的 sql 標籤範本搭配動態值，或 rds 模組公用程式以程式設計方式建置陳述式。如需詳細資訊，請參閱 [RDS 資料來源的解析程式函數參考](#) 和 [內建模組](#)。

使用自我檢查功能（主控台）

如需詳細的教學課程和入門指南，請參閱 [教學課程：Aurora PostgreSQL Serverless with Data API](#)。

AWS AppSync 主控台可讓您從使用資料 API 設定的現有 Aurora 資料庫，在幾分鐘內建立 AWS AppSync GraphQL API。這會根據您的資料庫組態快速產生操作結構描述。您可以依原狀使用 API 或建置以新增功能。

1. 登入 AWS Management Console 並開啟 [AppSync 主控台](#)。
 - 在儀表板上，選擇 Create API (建立 API)。
2. 在 API 選項下，選擇 GraphQL APIs，從 Amazon Aurora 叢集開始，然後選擇下一步。
 - a. 輸入 API 名稱。這將用作 主控台中 API 的識別符。
 - b. 如需聯絡詳細資訊，您可以輸入聯絡點來識別 API 的管理員。此為選用欄位。
 - c. 在私有 API 組態下，您可以啟用私有 API 功能。私有 API 只能從設定的 VPC 端點 (VPCE) 存取。如需詳細資訊，請參閱 [私有 APIs](#)。

不建議在此範例中啟用此功能。檢閱您的輸入後，請選擇下一步。

3. 在資料庫頁面中，選擇選取資料庫。
 - a. 您需要從叢集中選擇資料庫。第一步是選擇叢集所在的區域。
 - b. 從下拉式清單中選擇 Aurora 叢集。請注意，您必須先建立並 [啟用](#) 對應的資料 API，才能使用資源。

- c. 接著，您必須將資料庫的登入資料新增至服務。這主要是使用 完成 AWS Secrets Manager。選擇您秘密所在的區域。如需如何擷取秘密資訊的詳細資訊，請參閱[尋找秘密](#)或[擷取秘密](#)。
 - d. 從下拉式清單中新增您的秘密。請注意，使用者必須具有資料庫的[讀取許可](#)。
4. 選擇匯入。

AWS AppSync 將開始導入您的資料庫、探索資料表、資料欄、主索引鍵和索引。它會檢查 GraphQL API 中是否支援探索到的資料表。請注意，若要支援建立新的資料列，資料表需要主索引鍵，可以使用多個 column. AWS AppSync maps 資料表資料欄來輸入欄位，如下所示：

資料類型	欄位類型
VARCHAR	String
CHAR	String
BINARY	String
VARBINARY	String
TINYBLOB	String
TINYTEXT	String
TEXT	String
BLOB	String
MEDIUMTEXT	String
MEDIUMBLOB	String
LONGTEXT	String
LOB	String
BOOL	Boolean
BOOLEAN	Boolean
BIT	Int

TINYINT	Int
SMALLINT	Int
MEDIUMINT	Int
INT	Int
INTEGER	Int
BIGINT	Int
YEAR	Int
FLOAT	Float
DOUBLE	Float
DECIMAL	Float
DEC	Float
NUMERIC	Float
DATE	AWSDate
TIMESTAMP	String
DATETIME	String
TIME	AWSTime
JSON	AWSJson
ENUM	ENUM

5. 資料表探索完成後，資料庫區段將填入您的資訊。在新的資料庫資料表區段中，資料表中的資料可能已填入並轉換為結構描述的類型。如果您沒有看到某些必要資料，您可以選擇新增資料表、按一下出現的模態中這些類型的核取方塊，然後選擇新增來檢查資料。

若要從資料庫資料表區段中移除類型，請按一下您要移除的類型旁的核取方塊，然後選擇移除。如果您想要稍後再次新增，移除的類型將放置在新增資料表模態中。

請注意，AWS AppSync 使用資料表名稱做為類型名稱，但您可以重新命名它們 - 例如，將##等複數資料表名稱變更為類型名稱##。若要重新命名資料庫資料表區段中的類型，請按一下您要重新命名之類型的核取方塊，然後按一下類型名稱欄中的鉛筆圖示。

若要根據您的選擇預覽結構描述的內容，請選擇預覽結構描述。請注意，此結構描述不能為空，因此您至少必須有一個資料表轉換為類型。此外，此結構描述的大小不能超過 1 MB。

- 在服務角色下，選擇是否要特別為此匯入建立新的服務角色，或使用現有的角色。

6. 選擇 Next (下一步)。
7. 接著，選擇是否建立唯讀 API (僅限查詢) 或用於讀取和寫入資料的 API (具有查詢和變動)。後者也支援由變動觸發的即時訂閱。
8. 選擇 Next (下一步)。
9. 檢閱您的選擇，然後選擇建立 API。AWS AppSync 會建立 API，並將解析程式連接至查詢和變動。產生的 API 可完全運作，並可視需要擴充。

使用自我檢查功能 (API)

您可以使用 `StartDataSourceIntrospection` 自我檢查 API，以程式設計方式探索資料庫中的模型。如需 命令的詳細資訊，請參閱使用 [StartDataSourceIntrospection](#) API。

若要使用 `StartDataSourceIntrospection`，請提供您的 Aurora 叢集 Amazon Resource Name (ARN)、資料庫名稱和 AWS Secrets Manager 秘密 ARN。命令會啟動自我檢查程序。您可以使用 `GetDataSourceIntrospection` 命令擷取結果。您可以指定命令是否應傳回所探索模型的儲存定義語言 (SDL) 字串。這對於直接從探索到的模型產生 SDL 結構描述定義非常有用。

例如，如果您有下列簡單 Todos 資料表的資料定義語言 (DDL) 陳述式：

```
create table if not exists public.todos
(
  id serial constraint todos_pk primary key,
  description text,
  due timestamp,
  "createdAt" timestamp default now()
);
```

您使用以下項目開始自我檢查。

```
aws appsync start-data-source-introspection \
```

```
--rds-data-api-config resourceArn=<cluster-arn>,secretArn=<secret-arn>,databaseName=database
```

接著，使用 `GetDataSourceIntrospection` 命令來擷取結果。

```
aws appsync get-data-source-introspection \  
  --introspection-id a1234567-8910-abcd-efgh-identifier \  
  --include-models-sdl
```

這會傳回下列結果。

```
{  
  "introspectionId": "a1234567-8910-abcd-efgh-identifier",  
  "introspectionStatus": "SUCCESS",  
  "introspectionStatusDetail": null,  
  "introspectionResult": {  
    "models": [  
      {  
        "name": "todos",  
        "fields": [  
          {  
            "name": "description",  
            "type": {  
              "kind": "Scalar",  
              "name": "String",  
              "type": null,  
              "values": null  
            },  
            "length": 0  
          },  
          {  
            "name": "due",  
            "type": {  
              "kind": "Scalar",  
              "name": "AWSDateTime",  
              "type": null,  
              "values": null  
            },  
            "length": 0  
          },  
          {  
            "name": "id",  
            "type": {
```

```

        "kind": "NonNull",
        "name": null,
        "type": {
            "kind": "Scalar",
            "name": "Int",
            "type": null,
            "values": null
        },
        "values": null
    },
    "length": 0
},
{
    "name": "createdAt",
    "type": {
        "kind": "Scalar",
        "name": "AWSDateTime",
        "type": null,
        "values": null
    },
    "length": 0
}
],
"primaryKey": {
    "name": "PRIMARY_KEY",
    "fields": [
        "id"
    ]
},
"indexes": [],
"sdl": "type todos {\n  description: String\n  due: AWSDateTime\n  id: Int!\n  createdAt: AWSDateTime\n}"
},
"nextToken": null
}
}

```


使用 Amplify 用戶端建置用戶端應用程式

您可以使用任何 AWS AppSync GraphQL 用戶端連線至您的 GraphQL API，但我們強烈建議您使用 Amplify v6 用戶端。Amplify 不僅為您的 GraphQL API 自動產生強型用戶端 SDKs，也支援用戶端應用程式中的即時資料和增強型 GraphQL 查詢功能。對於 Web 應用程式，Amplify 可以產生 JavaScript 用戶端。對於以跨平台或行動環境為目標的人，Amplify 可滿足 Android、iOS 和 React Native 的需求。若要深入了解這些平台的用戶端程式碼產生，請參閱 Amplify [文件](#)。以下是使用 JavaScript React 應用程式開始旅程的指南：

Note

您需要安裝和設定 [npm](#) 和 [Amazon CLI](#)，才能開始使用。如果您使用的是 Amplify v6 用戶端，[請遵循本指南](#)。

開始使用：

1. 在本機電腦上，導覽至專案的目錄。使用下列命令安裝 Amplify 程式庫：

```
npm install aws-amplify
```

2. 下載您的組態檔案，並將其放在您的專案資料夾中。您的組態檔案通常包含 config 變數，其中已定義一些設定（端點、區域、授權模式等）。例如，它看起來可能像這樣：

```
const config = {
  API: {
    GraphQL: {
      endpoint: 'https://abcdefghijklmnopqrstuvwxyz.appsync-api.us-west-2.amazonaws.com/graphql',
      region: 'us-west-2',
      defaultAuthMode: 'apiKey',
      apiKey: ''
    }
  }
};

export default config;
```

3. 在您的程式碼中，匯入 Amplify Library 和您的組態以設定 Amplify：

```
import { Amplify } from 'aws-amplify';
import config from './aws-exports.js';

Amplify.configure(config);
```

或者，使用 API 組態中的程式碼片段直接設定 Amplify：

```
import { Amplify } from 'aws-amplify';

Amplify.configure({
  API: {
    GraphQL: {
      endpoint: 'https://abcdefghijklmnopqrstuvwxyz.appsync-api.us-west-2.amazonaws.com/graphql',
      region: 'us-west-2',
      defaultAuthMode: 'apiKey',
      apiKey: ''
    }
  }
});
```

4. 使用 Amplify 工具鏈，您可以選擇根據您的結構描述自動產生操作，這可節省您手動編寫指令碼的工作量。在應用程式的根目錄中，使用下列 CLI 命令：

```
npx @aws-amplify/cli codegen add --apiId <id goes here> --region <region goes here>
```

這將下載 API 的結構描述，並依預設將用戶端協助程式程式碼產生到 `src/graphql` 資料夾。每次 API 部署後，您可以重新執行下列命令，以產生更新的 GraphQL 陳述式和類型：

```
npx @aws-amplify/cli codegen
```

5. 您現在可以為 Android、Swift、Flutter 和 JavaScript DataStore 產生模型。使用下列命令下載您的結構描述：

```
aws appsync get-introspection-schema --api-id <id goes here> --region <region goes here> --format SDL schema.graphql
```

然後，從應用程式的根目錄執行下列命令：

```
npx @aws-amplify/cli codegen models \  
--model-schema schema.graphql \  
--target [android|ios|flutter|javascript|typescript] \  
--output-dir ./
```

適用於 AWS AppSync 的 JavaScript 解析程式教學課程

AWS AppSync 會使用資料來源和解析程式來翻譯 GraphQL 請求，並從 AWS 資源擷取資訊。AWS AppSync 支援使用特定資料來源類型的自動佈建和連線。AWS AppSync 也支援 Amazon DynamoDB、AWS Lambda、關聯式資料庫 (Amazon Aurora Serverless)、Amazon OpenSearch Service 和 HTTP 端點做為資料來源。您可以將 GraphQL API 與現有的 AWS 資源搭配使用，或從頭建置資料來源和解析程式。以下章節旨在以教學的形式闡明一些較常見的 GraphQL 使用案例。

主題

- [使用 DynamoDB JavaScript 解析程式建立簡單的貼文應用程式](#)
- [在 中 使用 AWS Lambda 解析程式 AWS AppSync](#)
- [使用本機解析程式 in AWS AppSync](#)
- [在 中 結合 GraphQL 解析程式 AWS AppSync](#)
- [在 AWS AppSync 中使用 Amazon OpenSearch Service 解析程式](#)
- [在 AWS AppSync 中執行 DynamoDB 交易](#)
- [在 AWS AppSync 中使用 DynamoDB 批次操作](#)
- [使用 HTTP 解析程式 in AWS AppSync](#)
- [在 中 使用 Aurora PostgreSQL 搭配資料 API AWS AppSync](#)

使用 DynamoDB JavaScript 解析程式建立簡單的貼文應用程式

在本教學課程中，您將 Amazon DynamoDB 資料表匯入 AWS AppSync，並使用 JavaScript JavaScript 管道解析程式來連接它們，以建置功能完整的 GraphQL API，供您在自己的應用程式中使用。

您將使用 AWS AppSync 主控台來佈建 Amazon DynamoDB 資源、建立解析程式，並將它們連接到資料來源。您也可以透過 GraphQL 陳述式讀取和寫入 Amazon DynamoDB 資料庫，並訂閱即時資料。

您必須完成特定步驟，才能將 GraphQL 陳述式翻譯為 Amazon DynamoDB 操作，以及將回應翻譯回 GraphQL。本教學課程概述透過幾個真實世界案例和資料存取模式的組態程序。

建立 GraphQL API

在 中 建立 GraphQL API AWS AppSync

1. 開啟 AppSync 主控台，然後選擇建立 API。
2. 選取從頭開始設計，然後選擇下一步。
3. 為您的 API 命名 PostTutorialAPI，然後選擇下一步。跳到檢閱頁面，同時將其餘選項設定為其預設值，然後選擇 Create。

AWS AppSync 主控台會為您建立新的 GraphQL API。透過停用，它會使用 API 金鑰身分驗證模式。您可以使用主控台來設定其他 GraphQL API 和對其執行查詢，以進行此教學的其他部分。

定義基本文章 API

現在您已擁有 GraphQL API，您可以設定基本結構描述，以允許基本建立、擷取和刪除貼文資料。

將資料新增至您的結構描述

1. 在您的 API 中，選擇結構描述索引標籤。
2. 我們將建立結構描述，定義 Post 類型和 操作 addPost 來新增和取得 Post 物件。在結構描述窗格中，使用下列程式碼取代內容：

```
schema {
  query: Query
  mutation: Mutation
}

type Query {
  getPost(id: ID!): Post
}

type Mutation {
  addPost(
    id: ID!
    author: String!
    title: String!
    content: String!
    url: String!
  ): Post!
}

type Post {
  id: ID!
  author: String
  title: String
```

```
content: String
url: String
ups: Int!
downs: Int!
version: Int!
}
```

3. 選擇 Save Schema (儲存結構描述)。

設定 Amazon DynamoDB 資料表

AWS AppSync 主控台可協助佈建所需的 AWS 資源，將您自己的資源存放在 Amazon DynamoDB 資料表中。在此步驟中，您將建立 Amazon DynamoDB 資料表來存放您的文章。您也將設定[次要索引](#)，以供我們稍後使用。

建立 Amazon DynamoDB 資料表

1. 在結構描述頁面上，選擇建立資源。
2. 選擇使用現有類型，然後選擇Post類型。
3. 在其他索引區段中，選擇新增索引。
4. 為索引命名 author-index。
5. 將 Primary key 設定為 author，將 Sort金鑰設定為 None。
6. 停用 自動產生 GraphQL。在此範例中，我們將自行建立解析程式。
7. 選擇 Create (建立)。

您現在有一個名為 的新資料來源PostTable，您可以透過在側邊索引標籤中瀏覽資料來源來查看。您將使用此資料來源將查詢和變動連結至 Amazon DynamoDB 資料表。

設定 addPost 解析程式 (Amazon DynamoDB PutItem)

現在 AWS AppSync，您知道 Amazon DynamoDB 資料表，您可以透過定義解析程式將其連結至個別查詢和變動。您建立的第一個解析程式是使用 JavaScript 的addPost管道解析程式，可讓您在 Amazon DynamoDB 資料表中建立文章。管道解析程式具有下列元件：

- 在 GraphQL 結構描述中要附加解析程式的位置。在這種情況下，您會對 createPost 類型上的 Mutation 欄位設定解析程式。當發起人呼叫變動時，將會叫用此解析程式{ addPost(...) {...} }。

- 用於此解析程式的資料來源。在此情況下，您想要使用先前定義的 DynamoDB 資料來源，以便您可以將項目新增至 `post-table-for-tutorial` DynamoDB 資料表。
- 請求處理常式。請求處理常式是一種函數，可處理來自發起人的傳入請求，並將其轉換為指示 AWS AppSync，讓對 DynamoDB 執行。
- 回應處理常式。回應處理常式的任務是處理來自 DynamoDB 的回應，並將其翻譯回 GraphQL 預期的內容。如果 DynamoDB 中的資料形狀與 GraphQL 中的 `Post` 類型不同，此功能會很有用，但如果他們的形狀一樣，您只需傳遞資料。

設定您的解析程式

1. 在 API 中，選擇結構描述索引標籤。
2. 在解析程式窗格中，尋找 `Mutation` 類型下的 `addPost` 欄位，然後選擇連接。
3. 選擇您的資料來源，然後選擇建立。
4. 在您的程式碼編輯器中，將程式碼取代為此程式碼片段：

```
import { util } from '@aws-appsync/utils'
import * as ddb from '@aws-appsync/utils/dynamodb'

export function request(ctx) {
  const item = { ...ctx.arguments, ups: 1, downs: 0, version: 1 }
  const key = { id: ctx.args.id ?? util.autoId() }
  return ddb.put({ key, item })
}

export function response(ctx) {
  return ctx.result
}
```

5. 選擇 **Save (儲存)**。

Note

在此程式碼中，您可以使用 DynamoDB 模組 `utils`，讓您輕鬆地建立 DynamoDB 請求。

AWS AppSync 隨附名為 `util.autoId()` 的自動 ID 產生公用程式，用於產生新文章的 ID。如果您未指定 ID，則公用程式會自動為您產生 ID。

```
const key = { id: ctx.args.id ?? util.autoId() }
```

如需 JavaScript 可用公用程式的詳細資訊，請參閱[解析程式和函數的 JavaScript 執行期功能](#)。

呼叫 API 以新增文章

現在已設定解析程式，AWS AppSync 可以將傳入addPost變動轉譯為 Amazon DynamoDB PutItem操作。您現在可以執行變動以將項目放置到資料表中。

執行 操作

1. 在 API 中，選擇查詢索引標籤。
2. 在查詢窗格中，新增下列變動：

```
mutation addPost {  
  addPost(  
    id: 123,  
    author: "AUTHORNAME"  
    title: "Our first post!"  
    content: "This is our first post."  
    url: "https://aws.amazon.com/appsync/"  
  ) {  
    id  
    author  
    title  
    content  
    url  
    ups  
    downs  
    version  
  }  
}
```

3. 選擇執行（橘色播放按鈕），然後選擇 addPost。新建立的文章結果應該會出現在查詢窗格右側的結果窗格中。其看起來與下列類似：

```
{  
  "data": {  
    "addPost": {  
      "id": "123",  
      "author": "AUTHORNAME",  
      "title": "Our first post!",  
    }  
  }  
}
```



```

    "content": "This is our first post.",
    "url": "https://aws.amazon.com/appsync/",
    "ups": 1,
    "downs": 0,
    "version": 1
  }
}

```

下列說明顯示發生了什麼：

1. AWS AppSync 收到addPost變動請求。
2. AWS AppSync 會執行解析程式的請求處理常式。ddb.put 函數會建立如下所示的PutItem請求：

```

{
  operation: 'PutItem',
  key: { id: { S: '123' } },
  attributeValues: {
    downs: { N: 0 },
    author: { S: 'AUTHORNAME' },
    ups: { N: 1 },
    title: { S: 'Our first post!' },
    version: { N: 1 },
    content: { S: 'This is our first post.' },
    url: { S: 'https://aws.amazon.com/appsync/' }
  }
}

```

3. AWS AppSync 使用此值來產生和執行 Amazon DynamoDB PutItem請求。
4. AWS AppSync 取得PutItem請求的結果並將其轉換回 GraphQL 類型。

```

{
  "id" : "123",
  "author": "AUTHORNAME",
  "title": "Our first post!",
  "content": "This is our first post.",
  "url": "https://aws.amazon.com/appsync/",
  "ups" : 1,
  "downs" : 0,
  "version" : 1
}

```

5. 回應處理常式會立即傳回結果 (return ctx.result)。
6. 最終結果會顯示在 GraphQL 回應中。

設定 getPost 解析程式 (Amazon DynamoDB GetItem)

現在，您可以將資料新增至 Amazon DynamoDB 資料表，您需要設定 getPost 查詢，以便從資料表擷取該資料。若要執行此作業，您可以設定另一個解析程式。

新增您的解析程式

1. 在您的 API 中，選擇結構描述索引標籤。
2. 在右側的解析程式窗格中，尋找 Query 類型的 getPost 欄位，然後選擇連接。
3. 選擇您的資料來源，然後選擇建立。
4. 在程式碼編輯器中，使用此程式碼片段取代程式碼：

```
import * as ddb from '@aws-appsync/utils/dynamodb'

export function request(ctx) {
  return ddb.get({ key: { id: ctx.args.id } })
}

export const response = (ctx) => ctx.result
```

5. 儲存您的解析程式。

Note

在此解析程式中，我們使用回應處理常式的箭頭函數表達式。

呼叫 API 以取得文章

現在已設定解析程式，AWS AppSync 知道如何將傳入 getPost 查詢轉譯為 Amazon DynamoDB GetItem 操作。您現在可以執行查詢，擷取您稍早建立的貼文。

執行查詢

1. 在您的 API 中，選擇查詢索引標籤。

2. 在查詢窗格中，新增下列程式碼，並使用您在建立文章後複製的 ID：

```
query getPost {  
  getPost(id: "123") {  
    id  
    author  
    title  
    content  
    url  
    ups  
    downs  
    version  
  }  
}
```

3. 選擇執行（橘色播放按鈕），然後選擇 `getPost`。新建立的文章結果應該會出現在查詢窗格右側的結果窗格中。
4. 從 Amazon DynamoDB 擷取的文章應該會出現在查詢窗格右側的結果窗格中。其看起來與下列類似：

```
{  
  "data": {  
    "getPost": {  
      "id": "123",  
      "author": "AUTHORNAME",  
      "title": "Our first post!",  
      "content": "This is our first post.",  
      "url": "https://aws.amazon.com/appsync/",  
      "ups": 1,  
      "downs": 0,  
      "version": 1  
    }  
  }  
}
```

或者，請採用下列範例：

```
query getPost {  
  getPost(id: "123") {  
    id  
    author
```

```
    title
  }
}
```

如果您的 `getPost` 查詢只需要 `id`、`author` 和 `title`，您可以變更請求函數以使用投影表達式來僅指定您想要從 DynamoDB 資料表傳輸的屬性，以避免從 DynamoDB 傳輸不必要的資料 AWS AppSync。例如，請求函數可能看起來像下面的程式碼片段：

```
import * as ddb from '@aws-appsync/utils/dynamodb'

export function request(ctx) {
  return ddb.get({
    key: { id: ctx.args.id },
    projection: ['author', 'id', 'title'],
  })
}

export const response = (ctx) => ctx.result
```

您也可以搭配使用 [selectionSetList](#) `getPost` 來代表 expression：

```
import * as ddb from '@aws-appsync/utils/dynamodb'

export function request(ctx) {
  const projection = ctx.info.selectionSetList.map((field) => field.replace('/', '.'))
  return ddb.get({ key: { id: ctx.args.id }, projection })
}

export const response = (ctx) => ctx.result
```

建立 updatePost 變動 (Amazon DynamoDB UpdateItem)

到目前為止，您可以在 Amazon DynamoDB 中建立和擷取 Post 物件。接著，您將設定新的變動來更新物件。與需要指定所有欄位的 `addPost` 變動相比，此變動只允許您指定要變更的欄位。它還引入了一個新的 `expectedVersion` 引數，允許您指定要修改的版本。您將設定條件，以確保您正在修改物件的最新版本。您將使用 `UpdateItem` Amazon DynamoDB [https : //operation.sc](https://operation.sc)

更新解析程式

1. 在您的 API 中，選擇結構描述索引標籤。

2. 修改 Schema (結構描述) 窗格中的 Mutation 類型來新增 updatePost 變動，如下所示：

```
type Mutation {  
  updatePost(  
    id: ID!,  
    author: String!,  
    title: String!,  
    content: String!,  
    url: String!,  
    expectedVersion: Int!  
  ): Post  
  
  addPost(  
    id: ID!  
    author: String!  
    title: String!  
    content: String!  
    url: String!  
  ): Post!  
}
```

3. 選擇 Save Schema (儲存結構描述)。

4. 在右側的解析程式窗格中，尋找Mutation類型上新建立updatePost的欄位，然後選擇連接。使用以下程式碼片段建立新的解析程式：

```
import { util } from '@aws-appsync/utils';  
import * as ddb from '@aws-appsync/utils/dynamodb';  
  
export function request(ctx) {  
  const { id, expectedVersion, ...rest } = ctx.args;  
  const values = Object.entries(rest).reduce((obj, [key, value]) => {  
    obj[key] = value ?? ddb.operations.remove();  
    return obj;  
  }, {});  
  
  return ddb.update({  
    key: { id },  
    condition: { version: { eq: expectedVersion } },  
    update: { ...values, version: ddb.operations.increment(1) },  
  });  
}  
  
export function response(ctx) {
```

```
const { error, result } = ctx;
if (error) {
  util.appendError(error.message, error.type);
}
return result;
```

5. 儲存您所做的任何變更。

此解析程式使用 `ddb.update` 來建立 Amazon DynamoDB `UpdateItem` 請求。您只是要求 Amazon DynamoDB 更新特定屬性，而不是撰寫整個項目。這是使用 Amazon DynamoDB 更新表達式來完成。

`ddb.update` 函數會採用金鑰和更新物件做為引數。然後，檢查傳入引數的值。將值設定為 `null`，請使用 DynamoDB `remove` 操作來表示該值應從 DynamoDB 項目中移除。

也有新的 `condition` 區段。條件表達式可讓您在執行操作之前，根據已在 Amazon DynamoDB 中的物件狀態，告知 AWS AppSync 和 Amazon DynamoDB 請求是否應成功。在此情況下，只有在 Amazon DynamoDB 中目前項目 `version` 的欄位完全符合 `expectedVersion` 引數時，您才希望 `UpdateItem` 請求成功。當項目更新時，我們希望增加的值 `version`。這很容易與操作函數搭配使用 `increment`。

如需條件表達式的詳細資訊，請參閱 [條件表達式](#) 文件。

如需 `UpdateItem` 請求的詳細資訊，請參閱 [UpdateItem](#) 文件和 [DynamoDB 模組](#) 文件。

如需有關如何撰寫更新表達式的詳細資訊，請參閱 [DynamoDB UpdateExpressions](#) 文件。

呼叫 API 以更新文章

讓我們嘗試使用新的解析程式更新 `Post` 物件。

更新物件

1. 在您的 API 中，選擇查詢索引標籤。
2. 在查詢窗格中，新增下列變動。您還需要將 `id` 引數更新為您稍早記下的值：

```
mutation updatePost {
  updatePost(
    id:123
    title: "An empty story"
    content: null
    expectedVersion: 1
```

```
) {  
  id  
  author  
  title  
  content  
  url  
  ups  
  downs  
  version  
}  
}
```

3. 選擇執行（橘色播放按鈕），然後選擇 `updatePost`。
4. Amazon DynamoDB 中更新的文章應該會出現在查詢窗格右側的結果窗格中。其看起來與下列類似：

```
{  
  "data": {  
    "updatePost": {  
      "id": "123",  
      "author": "A new author",  
      "title": "An empty story",  
      "content": null,  
      "url": "https://aws.amazon.com/appsync/",  
      "ups": 1,  
      "downs": 0,  
      "version": 2  
    }  
  }  
}
```

在此請求中，您要求 AWS AppSync 和 Amazon DynamoDB 僅更新 `title` 和 `content` 欄位。所有其他欄位都是單獨保留的（不是遞增 `version` 欄位）。您可以將 `title` 屬性設定為新值，並從文章中移除 `content` 屬性。`author`、`url`、`ups` 和 `downs` 欄位維持原狀。嘗試再次執行變動請求，同時完全離開請求。您應該會看到類似以下的回應：

```
{  
  "data": {  
    "updatePost": null  
  },  
  "errors": [  
    {  
      "message": "Cannot update item with null title",  
      "locations": [  
        {  
          "line": 1,  
          "column": 10  
        }  
      ],  
      "path": [  
        "updatePost"  
      ]  
    }  
  ]  
}
```

```

{
  "path": [
    "updatePost"
  ],
  "data": null,
  "errorType": "DynamoDB:ConditionalCheckFailedException",
  "errorInfo": null,
  "locations": [
    {
      "line": 2,
      "column": 3,
      "sourceName": null
    }
  ],
  "message": "The conditional request failed (Service: DynamoDb, Status Code: 400, Request ID: 1RR3QN5F35CS8IV5VR40Q09NNBVV4KQNS05AEMVJF66Q9ASUAAJG)"
}
]
}

```

請求失敗，因為條件表達式評估為 false：

1. 第一次執行請求時，Amazon DynamoDB 中文章的 version 欄位值為 1，符合 expectedVersion 引數。請求成功，這表示 version 欄位已在 Amazon DynamoDB 中遞增至 2。
2. 第二次執行請求時，Amazon DynamoDB 中文章的 version 欄位值為 2，這不符合 expectedVersion 引數。

此模式通常稱為樂觀鎖定。

建立投票變動 (Amazon DynamoDB UpdateItem)

Post 類型包含 ups 和 downs 欄位，以啟用上標和下標的記錄。不過，目前 API 不會讓我們對它們執行任何動作。讓我們新增變動，讓我們向上和向下移動文章。

新增您的變動

1. 在 API 中，選擇結構描述索引標籤。
2. 在結構描述窗格中，修改 Mutation 類型並新增 DIRECTION 列舉以新增新的投票變動：

```
type Mutation {
```



```

vote(id: ID!, direction: DIRECTION!): Post
updatePost(
  id: ID!,
  author: String!,
  title: String!,
  content: String!,
  url: String!,
  expectedVersion: Int!
): Post
addPost(
  id: ID!,
  author: String!,
  title: String!,
  content: String!,
  url: String!
): Post!
}

enum DIRECTION {
  UP
  DOWN
}

```

3. 選擇 Save Schema (儲存結構描述)。
4. 在右側的解析程式窗格中，尋找 Mutation 類型上新建立 vote 的欄位，然後選擇連接。透過建立程式碼並以下列程式碼片段取代，建立新的解析程式：

```

import * as ddb from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
  const field = ctx.args.direction === 'UP' ? 'ups' : 'downs';
  return ddb.update({
    key: { id: ctx.args.id },
    update: {
      [field]: ddb.operations.increment(1),
      version: ddb.operations.increment(1),
    },
  });
}

export const response = (ctx) => ctx.result;

```

5. 儲存您所做的任何變更。

呼叫 API 以向上或向下移動文章

現在已設定新的解析程式，AWS AppSync 知道如何將傳入upvotePost或downvote變動轉譯為Amazon DynamoDB UpdateItem操作。您現在可以執行變動，對您之前建立的文章表達贊同或不贊同。

執行您的變動

1. 在您的 API 中，選擇查詢索引標籤。
2. 在查詢窗格中，新增下列變動。您還需要將id引數更新為您稍早記下的值：

```
mutation votePost {  
  vote(id:123, direction: UP) {  
    id  
    author  
    title  
    content  
    url  
    ups  
    downs  
    version  
  }  
}
```

3. 選擇執行（橘色播放按鈕），然後選擇 votePost。
4. Amazon DynamoDB 中更新的文章應該會出現在查詢窗格右側的結果窗格中。其看起來與下列類似：

```
{  
  "data": {  
    "vote": {  
      "id": "123",  
      "author": "A new author",  
      "title": "An empty story",  
      "content": null,  
      "url": "https://aws.amazon.com/appsync/",  
      "ups": 6,  
      "downs": 0,  
      "version": 4  
    }  
  }  
}
```

```
}

```

5. 選擇再執行幾次。每次執行查詢1時，您應該會看到 ups 和 version 欄位遞增。
6. 變更查詢以使用不同的 呼叫它DIRECTION。

```
mutation votePost {
  vote(id:123, direction: DOWN) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

7. 選擇執行（橘色播放按鈕），然後選擇 votePost。

這次，您應該會看到 1 downs 和 version 欄位在每次執行查詢時遞增。

設定 deletePost 解析程式 (Amazon DynamoDB DeleteItem)

接著，您會想要建立變動來刪除文章。您將使用 Amazon DynamoDB DeleteItem 操作來執行此操作。

新增您的變動

1. 在您的結構描述中，選擇結構描述索引標籤。
2. 在結構描述窗格中，修改 Mutation 類型以新增新的deletePost變動：

```
type Mutation {
  deletePost(id: ID!, expectedVersion: Int): Post
  vote(id: ID!, direction: DIRECTION!): Post
  updatePost(
    id: ID!,
    author: String,
    title: String,
    content: String,
    url: String,
```

```

        expectedVersion: Int!
      ): Post
    addPost(
      id: ID!
      author: String!,
      title: String!,
      content: String!,
      url: String!
    ): Post!
  }

```

3. 這次，您將 `expectedVersion` 欄位設為選用。接著，選擇儲存結構描述。
4. 在右側的解析程式窗格中，尋找 `Mutation` 類型中新建立 `delete` 的欄位，然後選擇連接。使用下列程式碼建立新的解析程式：

```

import { util } from '@aws-appsync/utils'

import { util } from '@aws-appsync/utils';
import * as ddb from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
  let condition = null;
  if (ctx.args.expectedVersion) {
    condition = {
      or: [
        { id: { attributeExists: false } },
        { version: { eq: ctx.args.expectedVersion } },
      ],
    };
  }
  return ddb.remove({ key: { id: ctx.args.id }, condition });
}

export function response(ctx) {
  const { error, result } = ctx;
  if (error) {
    util.appendError(error.message, error.type);
  }
  return result;
}

```

 Note

`expectedVersion` 引數是選用的引數。如果發起人在請求中設定`expectedVersion`引數，請求處理常式會新增條件，只有在項目已刪除或 Amazon DynamoDB 中的文章`version`屬性完全符合時，才會允許`DeleteItem`請求成功`expectedVersion`。如果省略，將不會在 `DeleteItem` 要求中指定條件表達式。無論值為 `version`或項目是否存在於 Amazon DynamoDB 中，它都會成功。
即使您要刪除項目，如果項目尚未刪除，您也可以傳回已刪除的項目。

如需`DeleteItem`請求的詳細資訊，請參閱 [DeleteItem](#) 文件。

呼叫 API 以刪除文章

現在已設定解析程式，AWS AppSync 知道如何將傳入`delete`變動轉譯為 Amazon DynamoDB `DeleteItem`操作。您現在可以執行變動以在資料表中刪除項目。

執行您的變動

1. 在 API 中，選擇查詢索引標籤。
2. 在查詢窗格中，新增下列變動。您還需要將`id`引數更新為您稍早記下的值：

```
mutation deletePost {  
  deletePost(id:123) {  
    id  
    author  
    title  
    content  
    url  
    ups  
    downs  
    version  
  }  
}
```

3. 選擇執行（橘色播放按鈕），然後選擇 `deletePost`。
4. 文章會從 Amazon DynamoDB 刪除。請注意，會 AWS AppSync 傳回從 Amazon DynamoDB 刪除之項目的值，該值應該會出現在查詢窗格右側的結果窗格中。其看起來與下列類似：

```
{
  "data": {
    "deletePost": {
      "id": "123",
      "author": "A new author",
      "title": "An empty story",
      "content": null,
      "url": "https://aws.amazon.com/appsync/",
      "ups": 6,
      "downs": 4,
      "version": 12
    }
  }
}
```

5. 只有當此呼叫deletePost是從 Amazon DynamoDB 實際刪除時，才會傳回值。再次選擇執行。

6. 呼叫仍然成功，但不會傳回任何值：

```
{
  "data": {
    "deletePost": null
  }
}
```

7. 現在，讓我們嘗試刪除文章，但這次指定 expectedValue。首先，您需要建立新的文章，因為您剛剛刪除了到目前為止一直使用的文章。

8. 在查詢窗格中，新增下列變動：

```
mutation addPost {
  addPost(
    id:123
    author: "AUTHORNAME"
    title: "Our second post!"
    content: "A new post."
    url: "https://aws.amazon.com/appsync/"
  ) {
    id
    author
    title
    content
    url
  }
}
```

```

    ups
    downs
    version
  }
}

```

9. 選擇執行（橘色播放按鈕），然後選擇 `addPost`。

10 新建立的文章結果應該會出現在查詢窗格右側的結果窗格中。記錄新建立物件 `id` 的，因為您只需一點時間就需要它。其看起來與下列類似：

```

{
  "data": {
    "addPost": {
      "id": "123",
      "author": "AUTHORNAME",
      "title": "Our second post!",
      "content": "A new post.",
      "url": "https://aws.amazon.com/appsync/",
      "ups": 1,
      "downs": 0,
      "version": 1
    }
  }
}

```

11 現在，讓我們嘗試刪除具有 `expectedVersion` 非法值的文章。在查詢窗格中，新增下列變動。您還需要將 `id` 引數更新為您稍早記下的值：

```

mutation deletePost {
  deletePost(
    id:123
    expectedVersion: 9999
  ) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}

```

```
}

```

12 選擇執行（橘色播放按鈕），然後選擇 deletePost。會傳回下列結果：

```
{
  "data": {
    "deletePost": null
  },
  "errors": [
    {
      "path": [
        "deletePost"
      ],
      "data": null,
      "errorType": "DynamoDB:ConditionalCheckFailedException",
      "errorInfo": null,
      "locations": [
        {
          "line": 2,
          "column": 3,
          "sourceName": null
        }
      ],
      "message": "The conditional request failed (Service: DynamoDb, Status Code: 400, Request ID: 70830037M1FTFRK038A4CI9H43VV4KQNS05AEMVJF66Q9ASUAAJG)"
    }
  ]
}
```

13 請求失敗，因為條件表達式評估為 false。Amazon DynamoDB 中文章 version 的值與引數中 expectedValue 指定的 不相符。會在 GraphQL 回應 data 區段中 errors 欄位傳回物件的目前值。重試要求，但更正 expectedVersion：

```
mutation deletePost {
  deletePost(
    id:123
    expectedVersion: 1
  ) {
    id
    author
    title
    content
    url
  }
}
```



```

    ups
    downs
    version
  }
}
```

14 選擇執行（橘色播放按鈕），然後選擇 `deletePost`。

這次請求成功，並傳回從 Amazon DynamoDB 刪除的值：

```

{
  "data": {
    "deletePost": {
      "id": "123",
      "author": "AUTHORNAME",
      "title": "Our second post!",
      "content": "A new post.",
      "url": "https://aws.amazon.com/appsync/",
      "ups": 1,
      "downs": 0,
      "version": 1
    }
  }
}
```

15 再次選擇執行。呼叫仍然成功，但這次不會傳回任何值，因為文章已在 Amazon DynamoDB 中刪除。

```
{ "data": { "deletePost": null } }
```

設定 allPost 解析程式 (Amazon DynamoDB Scan)

到目前為止，只有在您知道要查看的每個文章id的時，API 才有用。讓我們新增新的解析程式，傳回資料表中的所有文章。

新增您的變動

1. 在 API 中，選擇結構描述索引標籤。
2. 修改 Schema (結構描述) 窗格中的 Query 類型以新增 allPost 查詢，如下所示：

```
type Query {
```

```
allPost(limit: Int, nextToken: String): PaginatedPosts!  
getPost(id: ID): Post  
}
```

3. 新增 PaginationPosts 類型：

```
type PaginatedPosts {  
  posts: [Post!]!  
  nextToken: String  
}
```

4. 選擇 Save Schema (儲存結構描述)。

5. 在右側的解析程式窗格中，尋找Query類型中新建立allPost的欄位，然後選擇連接。使用下列程式碼建立新的解析程式：

```
import * as ddb from '@aws-appsync/utils/dynamodb';  
  
export function request(ctx) {  
  const { limit = 20, nextToken } = ctx.arguments;  
  return ddb.scan({ limit, nextToken });  
}  
  
export function response(ctx) {  
  const { items: posts = [], nextToken } = ctx.result;  
  return { posts, nextToken };  
}
```

此解析程式的請求處理常式預期有兩個選用引數：

- limit - 指定在單一呼叫中傳回的項目數量上限。
- nextToken - 用來擷取下一組結果（我們將顯示的值稍後nextToken來自何處）。

6. 儲存對解析程式所做的任何變更。

如需Scan請求的詳細資訊，請參閱[掃描](#)參考文件。

呼叫 API 以掃描所有文章

現在已設定解析程式，AWS AppSync 知道如何將傳入allPost查詢轉譯為 Amazon DynamoDB Scan操作。您現在可以掃描資料表來擷取所有文章。在您可以嘗試之前，您必須將一些資料填入資料表，因為您已刪除目前為止所使用的項目。

新增和查詢資料

1. 在您的 API 中，選擇查詢索引標籤。
2. 在查詢窗格中，新增下列變動：

```
mutation addPost {  
  post1: addPost(id:1 author: "AUTHORNAME" title: "A series of posts, Volume 1"  
    content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }  
  post2: addPost(id:2 author: "AUTHORNAME" title: "A series of posts, Volume 2"  
    content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }  
  post3: addPost(id:3 author: "AUTHORNAME" title: "A series of posts, Volume 3"  
    content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }  
  post4: addPost(id:4 author: "AUTHORNAME" title: "A series of posts, Volume 4"  
    content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }  
  post5: addPost(id:5 author: "AUTHORNAME" title: "A series of posts, Volume 5"  
    content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }  
  post6: addPost(id:6 author: "AUTHORNAME" title: "A series of posts, Volume 6"  
    content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }  
  post7: addPost(id:7 author: "AUTHORNAME" title: "A series of posts, Volume 7"  
    content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }  
  post8: addPost(id:8 author: "AUTHORNAME" title: "A series of posts, Volume 8"  
    content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }  
  post9: addPost(id:9 author: "AUTHORNAME" title: "A series of posts, Volume 9"  
    content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }  
}
```

3. 選擇執行（橘色播放按鈕）。
4. 現在，讓我們來掃描資料表，一次會傳回五個結果。在查詢窗格中，新增下列查詢：

```
query allPost {  
  allPost(limit: 5) {  
    posts {  
      id  
      title  
    }  
    nextToken  
  }  
}
```

5. 選擇執行（橘色播放按鈕），然後選擇 allPost。

前五篇文章應該會出現在查詢窗格右側的結果窗格中。其看起來與下列類似：

```
{
  "data": {
    "allPost": {
      "posts": [
        {
          "id": "5",
          "title": "A series of posts, Volume 5"
        },
        {
          "id": "1",
          "title": "A series of posts, Volume 1"
        },
        {
          "id": "6",
          "title": "A series of posts, Volume 6"
        },
        {
          "id": "9",
          "title": "A series of posts, Volume 9"
        },
        {
          "id": "7",
          "title": "A series of posts, Volume 7"
        }
      ],
      "nextToken": "<token>"
    }
  }
}
```

6. 您收到五個結果nextToken，以及可用來取得下一組結果的。更新 allPost 查詢以包括來自之前結果組的 nextToken：

```
query allPost {
  allPost(
    limit: 5
    nextToken: "<token>"
  ) {
    posts {
      id
      author
    }
  }
}
```

```
    nextToken
  }
}
```

7. 選擇執行（橘色播放按鈕），然後選擇 allPost。

其餘四篇文章應該會出現在查詢窗格右側的結果窗格中。這組結果nextToken中沒有，因為您已翻頁瀏覽所有九篇文章，但沒有任何剩餘文章。其看起來與下列類似：

```
{
  "data": {
    "allPost": {
      "posts": [
        {
          "id": "2",
          "title": "A series of posts, Volume 2"
        },
        {
          "id": "3",
          "title": "A series of posts, Volume 3"
        },
        {
          "id": "4",
          "title": "A series of posts, Volume 4"
        },
        {
          "id": "8",
          "title": "A series of posts, Volume 8"
        }
      ],
      "nextToken": null
    }
  }
}
```

設定 allPostsByAuthor 解析程式 (Amazon DynamoDB 查詢)

除了掃描 Amazon DynamoDB 的所有文章之外，您也可以查詢 Amazon DynamoDB 以擷取特定作者建立的文章。您先前建立的 Amazon DynamoDB 資料表已有GlobalSecondaryIndex稱為的author-index，您可以搭配 Amazon DynamoDB Query操作使用，以擷取特定作者建立的所有文章。

新增查詢

1. 在您的 API 中，選擇結構描述索引標籤。
2. 修改 Schema (結構描述) 窗格中的 Query 類型以新增 allPostsByAuthor 查詢，如下所示：

```
type Query {  
  allPostsByAuthor(author: String!, limit: Int, nextToken: String): PaginatedPosts!  
  allPost(limit: Int, nextToken: String): PaginatedPosts!  
  getPost(id: ID): Post  
}
```

請注意，這使用與allPost查詢相同的PaginatedPosts類型。

3. 選擇 Save Schema (儲存結構描述)。
4. 在右側的解析程式窗格中，尋找 Query 類型上新建立allPostsByAuthor的欄位，然後選擇連接。使用以下程式碼片段建立解析程式：

```
import * as ddb from '@aws-appsync/utils/dynamodb';  
  
export function request(ctx) {  
  const { limit = 20, nextToken, author } = ctx.arguments;  
  return ddb.query({  
    index: 'author-index',  
    query: { author: { eq: author } },  
    limit,  
    nextToken,  
  });  
}  
  
export function response(ctx) {  
  const { items: posts = [], nextToken } = ctx.result;  
  return { posts, nextToken };  
}
```

如同allPost解析程式，此解析程式有兩個選用引數：

- limit - 指定在單一呼叫中傳回的項目數量上限。
- nextToken - 擷取下一組結果 (的值nextToken可從先前的呼叫取得) 。

5. 儲存對解析程式所做的任何變更。

如需Query請求的詳細資訊，請參閱[查詢](#)參考文件。

呼叫 API 以依作者查詢所有文章

現在已設定解析程式，AWS AppSync 知道如何針對author-index索引將傳入allPostsByAuthor的變動轉換為 DynamoDB Query操作。您現在可以查詢資料表以擷取特定作者的所有文章。

不過，在此之前，讓我們將更多文章填入資料表，因為到目前為止，每篇文章都有相同的作者。

新增資料和查詢

1. 在 API 中，選擇查詢索引標籤。
2. 在查詢窗格中，新增下列變動：

```
mutation addPost {  
  post1: addPost(id:10 author: "Nadia" title: "The cutest dog in the world" content:  
    "So cute. So very, very cute." url: "https://aws.amazon.com/appsync/" ) { author,  
    title }  
  post2: addPost(id:11 author: "Nadia" title: "Did you know...?" content: "AppSync  
    works offline?" url: "https://aws.amazon.com/appsync/" ) { author, title }  
  post3: addPost(id:12 author: "Steve" title: "I like GraphQL" content: "It's great"  
    url: "https://aws.amazon.com/appsync/" ) { author, title }  
}
```

3. 選擇執行（橘色播放按鈕），然後選擇 addPost。
4. 現在，讓我們查詢資料表，傳回 Nadia 撰寫的所有文章。在查詢窗格中，新增下列查詢：

```
query allPostsByAuthor {  
  allPostsByAuthor(author: "Nadia") {  
    posts {  
      id  
      title  
    }  
    nextToken  
  }  
}
```

5. 選擇執行（橘色播放按鈕），然後選擇 allPostsByAuthor。撰寫的所有文章Nadia都應出現在查詢窗格右側的結果窗格中。其看起來與下列類似：

```
{
```

```

"data": {
  "allPostsByAuthor": {
    "posts": [
      {
        "id": "10",
        "title": "The cutest dog in the world"
      },
      {
        "id": "11",
        "title": "Did you know...?"
      }
    ],
    "nextToken": null
  }
}

```

6. 分頁適用於 Query，如同它適用於 Scan。例如，讓我們查詢 AUTHORNAME 的所有文章，一次取得五篇。
7. 在查詢窗格中，新增下列查詢：

```

query allPostsByAuthor {
  allPostsByAuthor(
    author: "AUTHORNAME"
    limit: 5
  ) {
    posts {
      id
      title
    }
    nextToken
  }
}

```

8. 選擇執行（橘色播放按鈕），然後選擇 allPostsByAuthor。撰寫的所有文章AUTHORNAME都應出現在查詢窗格右側的結果窗格中。其看起來與下列類似：

```

{
  "data": {
    "allPostsByAuthor": {
      "posts": [
        {
          "id": "6",

```



```

        "title": "A series of posts, Volume 6"
      },
      {
        "id": "4",
        "title": "A series of posts, Volume 4"
      },
      {
        "id": "2",
        "title": "A series of posts, Volume 2"
      },
      {
        "id": "7",
        "title": "A series of posts, Volume 7"
      },
      {
        "id": "1",
        "title": "A series of posts, Volume 1"
      }
    ],
    "nextToken": "<token>"
  }
}
}

```

9. 使用之前查詢傳回的值更新 nextToken 引數，如下所示：

```

query allPostsByAuthor {
  allPostsByAuthor(
    author: "AUTHORNAME"
    limit: 5
    nextToken: "<token>"
  ) {
    posts {
      id
      title
    }
    nextToken
  }
}

```

10 選擇執行（橘色播放按鈕），然後選擇 allPostsByAuthor。撰寫的其餘文章AUTHORNAME應該會出現在查詢窗格右側的結果窗格中。其看起來與下列類似：

```
{
  "data": {
    "allPostsByAuthor": {
      "posts": [
        {
          "id": "8",
          "title": "A series of posts, Volume 8"
        },
        {
          "id": "5",
          "title": "A series of posts, Volume 5"
        },
        {
          "id": "3",
          "title": "A series of posts, Volume 3"
        },
        {
          "id": "9",
          "title": "A series of posts, Volume 9"
        }
      ],
      "nextToken": null
    }
  }
}
```

使用 集合

到目前為止，Post 類型是平面索引鍵/值物件。您也可以使用解析程式建立複雜物件的模型，例如集合、清單和映射。讓我們將 Post 類型更新為包含標籤。文章可以有零個或多個標籤，這些標籤以字串集的形式存放在 DynamoDB 中。您也會設定一些變動來新增和移除標籤，以及新查詢以掃描含特定標籤的文章。

設定您的資料

1. 在您的 API 中，選擇結構描述索引標籤。
2. 修改 Schema (結構描述) 窗格中的 Post 類型以新增 tags 欄位，如下所示：

```
type Post {
  id: ID!
```

```
author: String
title: String
content: String
url: String
ups: Int!
downs: Int!
version: Int!
tags: [String!]
}
```

3. 修改 Schema (結構描述) 窗格中的 Query 類型以新增 allPostsByTag 查詢，如下所示：

```
type Query {
  allPostsByTag(tag: String!, limit: Int, nextToken: String): PaginatedPosts!
  allPostsByAuthor(author: String!, limit: Int, nextToken: String): PaginatedPosts!
  allPost(limit: Int, nextToken: String): PaginatedPosts!
  getPost(id: ID!): Post
}
```

4. 修改 Schema (結構描述) 窗格中的 Mutation 類型以新增 addTag 和 removeTag 變動，如下所示：

```
type Mutation {
  addTag(id: ID!, tag: String!): Post
  removeTag(id: ID!, tag: String!): Post
  deletePost(id: ID!, expectedVersion: Int): Post
  upvotePost(id: ID!): Post
  downvotePost(id: ID!): Post
  updatePost(
    id: ID!,
    author: String,
    title: String,
    content: String,
    url: String,
    expectedVersion: Int!
  ): Post
  addPost(
    author: String!,
    title: String!,
    content: String!,
    url: String!
  ): Post!
}
```

5. 選擇 Save Schema (儲存結構描述)。
6. 在右側的解析程式窗格中，尋找Query類型上新建立allPostsByTag的欄位，然後選擇連接。使用以下程式碼片段建立您的解析程式：

```
import * as ddb from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
  const { limit = 20, nextToken, tag } = ctx.arguments;
  return ddb.scan({ limit, nextToken, filter: { tags: { contains: tag } } });
}

export function response(ctx) {
  const { items: posts = [], nextToken } = ctx.result;
  return { posts, nextToken };
}
```

7. 儲存您對解析程式所做的任何變更。
8. 現在，addTag使用下面的程式碼片段對 Mutation 欄位執行相同的操作：

Note

雖然 DynamoDB utils 目前不支援集合操作，但您仍可自行建置請求來與集合互動。

```
import { util } from '@aws-appsync/utils'

export function request(ctx) {
  const { id, tag } = ctx.arguments
  const expressionValues = util.dynamodb.toMapValues({ ':plusOne': 1 })
  expressionValues[':tags'] = util.dynamodb.toStringSet([tag])

  return {
    operation: 'UpdateItem',
    key: util.dynamodb.toMapValues({ id }),
    update: {
      expression: `ADD tags :tags, version :plusOne`,
      expressionValues,
    },
  }
}
```

```
export const response = (ctx) => ctx.result
```

9. 儲存對解析程式所做的任何變更。

10.removeTag 使用下面的程式碼片段，再對 Mutation 欄位重複一次：

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
  const { id, tag } = ctx.arguments;
  const expressionValues = util.dynamodb.toMapValues({ ':plusOne': 1 });
  expressionValues[':tags'] = util.dynamodb.toStringSet([tag]);

  return {
    operation: 'UpdateItem',
    key: util.dynamodb.toMapValues({ id }),
    update: {
      expression: `DELETE tags :tags ADD version :plusOne`,
      expressionValues,
    },
  };
}

export const response = (ctx) => ctx.result
```

11.儲存對解析程式所做的任何變更。

呼叫 API 來使用標籤

現在您已設定解析程式，AWS AppSync 知道如何將傳入的 removeTag、addTag 和 allPostsByTag 請求轉換為 DynamoDB UpdateItem 和 Scan 操作。為了嘗試看看，讓我們選擇您先前建立的其中一篇文章。例如，讓我們使用 Nadia 撰寫的一篇文章。

使用標籤

1. 在您的 API 中，選擇查詢索引標籤。
2. 在查詢窗格中，新增下列查詢：

```
query allPostsByAuthor {
  allPostsByAuthor(
    author: "Nadia"
  ) {
```

```
posts {  
  id  
  title  
}  
nextToken  
}
```

3. 選擇執行（橘色播放按鈕），然後選擇 `allPostsByAuthor`。
4. Nadia 的所有文章都應出現在查詢窗格右側的結果窗格中。其看起來與下列類似：

```
{  
  "data": {  
    "allPostsByAuthor": {  
      "posts": [  
        {  
          "id": "10",  
          "title": "The cutest dog in the world"  
        },  
        {  
          "id": "11",  
          "title": "Did you known...?"  
        }  
      ],  
      "nextToken": null  
    }  
  }  
}
```

5. 讓我們使用標題為 全世界最可愛的狗。記錄它 `id`，因為稍後會用到它。現在，讓我們嘗試新增 `dog` 標籤。
6. 在查詢窗格中，新增下列變動。您也必須將 `id` 引數更新為您先前記下的值。

```
mutation addTag {  
  addTag(id:10 tag: "dog") {  
    id  
    title  
    tags  
  }  
}
```

7. 選擇執行（橘色播放按鈕），然後選擇 `addTag`。文章會以新標籤更新：

```
{
  "data": {
    "addTag": {
      "id": "10",
      "title": "The cutest dog in the world",
      "tags": [
        "dog"
      ]
    }
  }
}
```

8. 您可以新增更多標籤。更新變動，將tag引數變更為 puppy：

```
mutation addTag {
  addTag(id:10 tag: "puppy") {
    id
    title
    tags
  }
}
```

9. 選擇執行（橘色播放按鈕），然後選擇 addTag。文章會以新標籤更新：

```
{
  "data": {
    "addTag": {
      "id": "10",
      "title": "The cutest dog in the world",
      "tags": [
        "dog",
        "puppy"
      ]
    }
  }
}
```

10 您也可以刪除標籤。在查詢窗格中，新增下列變動。您還需要將id引數更新為您稍早記下的值：

```
mutation removeTag {
  removeTag(id:10 tag: "puppy") {
    id
  }
}
```

```
    title
    tags
  }
}
```

11 選擇執行（橘色播放按鈕），然後選擇 `removeTag`。貼文將會更新，且 `puppy` 標籤將會刪除。

```
{
  "data": {
    "addTag": {
      "id": "10",
      "title": "The cutest dog in the world",
      "tags": [
        "dog"
      ]
    }
  }
}
```

12 您也可以搜尋具有標籤的所有文章。在查詢窗格中，新增下列查詢：

```
query allPostsByTag {
  allPostsByTag(tag: "dog") {
    posts {
      id
      title
      tags
    }
    nextToken
  }
}
```

13 選擇執行（橘色播放按鈕），然後選擇 `allPostsByTag`。具有 `dog` 標籤的所有文章將會傳回，如下所示：

```
{
  "data": {
    "allPostsByTag": {
      "posts": [
        {
          "id": "10",
          "title": "The cutest dog in the world",
          "tags": [
```



```
        "dog",
        "puppy"
      ]
    }
  ],
  "nextToken": null
}
}
```

結論

在本教學課程中，您已建置 API，可讓您使用 AWS AppSync 和 GraphQL 在 DynamoDB 中操作 Post 物件。

若要清除，您可以從主控台刪除 AWS AppSync GraphQL API。

若要刪除與 DynamoDB 資料表相關聯的角色，請在資料來源資料表中選取您的資料來源，然後按一下編輯。在建立或使用現有角色下記下角色的值。前往 IAM 主控台以刪除角色。

若要刪除 DynamoDB 資料表，請按一下資料來源清單中的資料表名稱。這會帶您前往 DynamoDB 主控台，您可以在其中刪除資料表。

在 中使用 AWS Lambda 解析程式 AWS AppSync

您可以使用 AWS Lambda with AWS AppSync 來解析任何 GraphQL 欄位。例如，GraphQL 查詢可能會將呼叫傳送至 Amazon Relational Database Service (Amazon RDS) 執行個體，而 GraphQL 變動可能會寫入 Amazon Kinesis 串流。在本節中，我們將示範如何撰寫 Lambda 函數，以根據 GraphQL 欄位操作的調用執行商業邏輯。

建立 Lambda 函數

下列範例顯示寫入 Node.js (執行時間：Node.js 18.x) 的 Lambda 函數，在部落格文章應用程式中對部落格文章執行不同的操作。請注意，程式碼應該儲存在副檔名為 .mis 的檔案名稱中。

```
export const handler = async (event) => {
  console.log('Received event {}'.format(JSON.stringify(event, 3)))

  const posts = {
```

```

1: { id: '1', title: 'First book', author: 'Author1', url: 'https://amazon.com/',
  content: 'SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT
AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1', ups: '100', downs: '10', },
  2: { id: '2', title: 'Second book', author: 'Author2', url: 'https://amazon.com',
  content: 'SAMPLE TEXT AUTHOR 2 SAMPLE TEXT AUTHOR 2 SAMPLE TEXT', ups: '100', downs:
'10', },
  3: { id: '3', title: 'Third book', author: 'Author3', url: null, content: null,
ups: null, downs: null },
  4: { id: '4', title: 'Fourth book', author: 'Author4', url: 'https://
www.amazon.com/', content: 'SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT
AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT
AUTHOR 4 SAMPLE TEXT AUTHOR 4', ups: '1000', downs: '0', },
  5: { id: '5', title: 'Fifth book', author: 'Author5', url: 'https://
www.amazon.com/', content: 'SAMPLE TEXT AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT
AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT', ups: '50', downs: '0', },
}

const relatedPosts = {
1: [posts['4']],
  2: [posts['3'], posts['5']],
  3: [posts['2'], posts['1']],
  4: [posts['2'], posts['1']],
  5: [],
}

console.log('Got an Invoke Request.')
let result
switch (event.field) {
case 'getPost':
  return posts[event.arguments.id]
case 'allPosts':
  return Object.values(posts)
case 'addPost':
  // return the arguments back
return event.arguments
case 'addPostErrorWithData':
  result = posts[event.arguments.id]
  // attached additional error information to the post
  result.errorMessage = 'Error with the mutation, data has changed'
  result.errorType = 'MUTATION_ERROR'
return result
case 'relatedPosts':
  return relatedPosts[event.source.id]
default:

```

```
        throw new Error('Unknown field, unable to resolve ' + event.field)
    }
}
```

此 Lambda 函數會依 ID 擷取文章、新增文章、擷取文章清單，以及擷取特定文章的相關文章。

Note

Lambda 函數使用 上的 switch 陳述式 `event.field` 來判斷目前正在解析哪個欄位。

使用 AWS 管理主控台建立此 Lambda 函數。

設定 Lambda 的資料來源

建立 Lambda 函數後，在 AWS AppSync 主控台中導覽至 GraphQL API，然後選擇資料來源索引標籤。

選擇建立資料來源，輸入易記的資料來源名稱（例如 **Lambda**），然後針對資料來源類型選擇 AWS Lambda 函數。針對區域，選擇與函數相同的區域。針對函數 ARN，選擇 Lambda 函數的 Amazon Resource Name (ARN)。

選擇 Lambda 函數後，您可以建立新的 AWS Identity and Access Management (IAM) 角色（AWS AppSync 會為其指派適當的許可），或選擇具有下列內嵌政策的現有角色：

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "lambda:InvokeFunction"
      ],
      "Resource": "arn:aws:lambda:REGION:ACCOUNTNUMBER:function/
LAMBDA_FUNCTION"
    }
  ]
}
```

您還必須為 IAM 角色設定與 AWS AppSync 的信任關係，如下所示：

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "appsync.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

建立 GraphQL 結構描述

現在資料來源已連接至 Lambda 函數，請建立 GraphQL 結構描述。

從 AWS AppSync 主控台的結構描述編輯器，請確定您的結構描述符合下列結構描述：

```
schema {
  query: Query
  mutation: Mutation
}
type Query {
  getPost(id:ID!): Post
  allPosts: [Post]
}
type Mutation {
  addPost(id: ID!, author: String!, title: String, content: String, url: String):
  Post!
}
type Post {
  id: ID!
  author: String!
  title: String
  content: String
  url: String
}
```

```
    ups: Int
    downs: Int
    relatedPosts: [Post]
  }
```

設定解析程式

現在您已註冊 Lambda 資料來源和有效的 GraphQL 結構描述，您可以使用解析程式將 GraphQL 欄位連線至 Lambda 資料來源。

您將建立使用 AWS AppSync JavaScript (APPSYNC_JS) 執行時間並與 Lambda 函數互動的解析程式。若要進一步了解使用 JavaScript 編寫 AWS AppSync 解析程式和函數，請參閱[解析程式和函數的 JavaScript 執行期功能](#)。

如需 Lambda 映射範本的詳細資訊，請參閱 [Lambda 的 JavaScript 解析程式函數參考](#)。

在此步驟中，您將解析程式連接至下列欄位的 Lambda 函數：getPost(id:ID!)：Post、addPost(id: ID!, author: String!, title: String, content: String, url: String): Post!、allPosts: [Post]和 Post.relatedPosts: [Post]。從 AWS AppSync 主控台的結構描述編輯器，在解析程式窗格中，選擇 getPost(id:ID!): Post 欄位旁的連接。選擇您的 Lambda 資料來源。接著，提供下列程式碼：

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
  const {source, args} = ctx
  return {
    operation: 'Invoke',
    payload: { field: ctx.info.fieldName, arguments: args, source },
  };
}

export function response(ctx) {
  return ctx.result;
}
```

此解析程式程式碼會在叫用來源物件時，將欄位名稱、引數清單和內容傳遞至 Lambda 函數。選擇儲存。

您已成功連接第一個解析程式。針對其餘欄位重複此操作。

測試 GraphQL API

現在，您的 Lambda 函式已經連接到 GraphQL 解析程式，您可以使用主控台或用戶端應用程式執行一些變動和查詢。

在 AWS AppSync 主控台的左側，選擇查詢，然後貼上下列程式碼：

addPost Mutation

```
mutation AddPost {  
  addPost(  
    id: 6  
    author: "Author6"  
    title: "Sixth book"  
    url: "https://www.amazon.com/"  
    content: "This is the book is a tutorial for using GraphQL with AWS AppSync."  
  ) {  
    id  
    author  
    title  
    content  
    url  
    ups  
    downs  
  }  
}
```

getPost Query

```
query GetPost {  
  getPost(id: "2") {  
    id  
    author  
    title  
    content  
    url  
    ups  
    downs  
  }  
}
```

allPosts Query

```
query AllPosts {  
  allPosts {  
    id  
    author  
    title  
    content  
    url  
    ups  
    downs  
    relatedPosts {  
      id  
      title  
    }  
  }  
}
```

傳回錯誤

任何指定的欄位解析度都可能導致錯誤。透過 AWS AppSync，您可以從下列來源引發錯誤：

- 解析程式回應處理常式
- Lambda 函數

從解析程式回應處理常式

若要引發刻意錯誤，您可以使用 `util.error` 公用程式方法。它需要 引數 `errorMessage`、`errorType` 和選用 `data` 值。發生錯誤時，`data` 可將額外的資料傳回給用戶端。`data` 物件會新增到 GraphQL 最後回應中的 `errors`。

下列範例顯示如何在 `Post.relatedPosts: [Post]` 解析程式回應處理常式中使用它。

```
// the Post.relatedPosts response handler  
export function response(ctx) {  
  util.error("Failed to fetch relatedPosts", "LambdaFailure", ctx.result)  
  return ctx.result;  
}
```

這會產生類似下列的 GraphQL 回應：

```
{
  "data": {
    "allPosts": [
      {
        "id": "2",
        "title": "Second book",
        "relatedPosts": null
      },
      ...
    ]
  },
  "errors": [
    {
      "path": [
        "allPosts",
        0,
        "relatedPosts"
      ],
      "errorType": "LambdaFailure",
      "locations": [
        {
          "line": 5,
          "column": 5
        }
      ],
      "message": "Failed to fetch relatedPosts",
      "data": [
        {
          "id": "2",
          "title": "Second book"
        },
        {
          "id": "1",
          "title": "First book"
        }
      ]
    }
  ]
}
```

其中的 `allPosts[0].relatedPosts` 為 `null`，因為錯誤以及 `errorMessage`、`errorType` 和 `data` 出現在 `data.errors[0]` 物件中。

從 Lambda 函式

AWS AppSync 也了解 Lambda 函數擲回的錯誤。Lambda 程式設計模型可讓您引發已處理的錯誤。如果 Lambda 函數擲回錯誤，AWS AppSync 無法解析目前的欄位。回應中只會設定從 Lambda 傳回的錯誤訊息。目前，您無法透過從 Lambda 函數引發錯誤，將任何無關的資料傳遞回用戶端。

Note

如果您的 Lambda 函數引發未處理的錯誤，AWS AppSync 會使用 Lambda 設定的錯誤訊息。

下列 Lambda 函式會引發錯誤：

```
export const handler = async (event) => {
  console.log('Received event {}'.format(JSON.stringify(event, 3)))
  throw new Error('I always fail.')
}
```

在您的回應處理常式中收到錯誤。您可以使用 `util.appendError` 將錯誤附加至回應，以將其傳回 GraphQL 回應。若要這樣做，請將 your AWS AppSync 函數回應處理常式變更為：

```
// the lambdaInvoke response handler
export function response(ctx) {
  const { error, result } = ctx;
  if (error) {
    util.appendError(error.message, error.type, result);
  }
  return result;
}
```

這會傳回類似下列的 GraphQL 回應：

```
{
  "data": {
    "allPosts": null
  },
  "errors": [
    {
      "path": [
        "allPosts"
      ]
    }
  ]
}
```

```
    ],
    "data": null,
    "errorType": "Lambda:Unhandled",
    "errorInfo": null,
    "locations": [
      {
        "line": 2,
        "column": 3,
        "sourceName": null
      }
    ],
    "message": "I fail. always"
  }
]
```

進階使用案例：批次處理

此範例中的 Lambda 函數有一個 `relatedPosts` 欄位，可傳回指定文章的相關文章清單。在範例查詢中，來自 Lambda 函數 `allPosts` 的欄位調用會傳回五個文章。因為我們指定了也要 `relatedPosts` 解析每個傳回的文章，所以會叫用 `relatedPosts` 欄位操作五次。

```
query {
  allPosts { // 1 Lambda invocation - yields 5 Posts
    id
    author
    title
    content
    url
    ups
    downs
    relatedPosts { // 5 Lambda invocations - each yields 5 posts
      id
      title
    }
  }
}
```

雖然這在此特定範例中可能聽起來並不重要，但這種複合式過度擷取可能會快速破壞應用程式。

如果您要在同一查詢中，再次擷取傳回之相關 Posts 的 `relatedPosts`，那麼叫用的次數將大幅增加。

```

query {
  allPosts {    // 1 Lambda invocation - yields 5 Posts
    id
    author
    title
    content
    url
    ups
    downs
    relatedPosts {    // 5 Lambda invocations - each yield 5 posts = 5 x 5 Posts
      id
      title
      relatedPosts {    // 5 x 5 Lambda invocations - each yield 5 posts = 25 x 5
        id
        title
        author
      }
    }
  }
}

```

在此相對簡單的查詢中，AWS AppSync 會叫用 Lambda 函數 $1 + 5 + 25 = 31$ 次。

這是一個相當常見的挑戰，通常稱為 N+1 問題 (在此例中， $N = 5$)，它可能會增加應用程式的延遲和成本。

解決此問題的方法之一，是將類似的欄位解析程式請求一起批次處理。在此範例中，它可以改為解析指定批次文章的相關文章清單，而不是讓 Lambda 函數解析單一指定文章的相關文章清單。

為了示範這一點，讓我們更新的解析程式 `relatedPosts` 來處理批次處理。

```

import { util } from '@aws-appsync/utils';

export function request(ctx) {
  const {source, args} = ctx
  return {
    operation: ctx.info.fieldName === 'relatedPosts' ? 'BatchInvoke' : 'Invoke',
    payload: { field: ctx.info.fieldName, arguments: args, source },
  };
}

export function response(ctx) {

```

```
const { error, result } = ctx;
if (error) {
  util.appendError(error.message, error.type, result);
}
return result;
}
```

現在 `fieldName`，當正在解決的 為 `InvokeBatchInvoke` 時，程式碼會將操作從 變更為 `relatedPosts`。現在，在設定批次處理區段中的 函數上啟用批次處理。將批次大小上限設定為 5。選擇儲存。

透過此變更，解析 時 `relatedPosts`，Lambda 函數會收到下列項目做為輸入：

```
[
  {
    "field": "relatedPosts",
    "source": {
      "id": 1
    }
  },
  {
    "field": "relatedPosts",
    "source": {
      "id": 2
    }
  },
  ...
]
```

在請求中指定 `BatchInvoke` 時，Lambda 函數會收到請求清單，並傳回結果清單。

具體而言，結果清單必須符合請求承載項目的大小和順序，以便 AWS AppSync 可以相應地符合結果。

在此批次範例中，Lambda 函數會傳回一批結果，如下所示：

```
[
  [{"id": "2", "title": "Second book"}, {"id": "3", "title": "Third book"}], //
  relatedPosts for id=1
  [{"id": "3", "title": "Third book"}] //
  relatedPosts for id=2
]
```

您可以更新 Lambda 程式碼來處理 的批次處理relatedPosts：

```
export const handler = async (event) => {
  console.log('Received event {}'.format(JSON.stringify(event, 3)))
  //throw new Error('I fail. always')

  const posts = {
    1: { id: '1', title: 'First book', author: 'Author1', url: 'https://amazon.com/',
      content: 'SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT
      AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1', ups: '100', downs: '10', },
    2: { id: '2', title: 'Second book', author: 'Author2', url: 'https://amazon.com',
      content: 'SAMPLE TEXT AUTHOR 2 SAMPLE TEXT AUTHOR 2 SAMPLE TEXT', ups: '100', downs:
      '10', },
    3: { id: '3', title: 'Third book', author: 'Author3', url: null, content: null,
      ups: null, downs: null },
    4: { id: '4', title: 'Fourth book', author: 'Author4', url: 'https://
      www.amazon.com/', content: 'SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT
      AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT
      AUTHOR 4 SAMPLE TEXT AUTHOR 4', ups: '1000', downs: '0', },
    5: { id: '5', title: 'Fifth book', author: 'Author5', url: 'https://
      www.amazon.com/', content: 'SAMPLE TEXT AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT
      AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT', ups: '50', downs: '0', },
  }

  const relatedPosts = {
    1: [posts['4']],
    2: [posts['3'], posts['5']],
    3: [posts['2'], posts['1']],
    4: [posts['2'], posts['1']],
    5: [],
  }

  if (!event.field && event.length){
    console.log(`Got a BatchInvoke Request. The payload has ${event.length} items to
    resolve.`);
    return event.map(e => relatedPosts[e.source.id])
  }

  console.log('Got an Invoke Request.')
  let result
  switch (event.field) {
    case 'getPost':
      return posts[event.arguments.id]
    case 'allPosts':
```

```

    return Object.values(posts)
  case 'addPost':
    // return the arguments back
    return event.arguments
  case 'addPostErrorWithData':
    result = posts[event.arguments.id]
    // attached additional error information to the post
    result.errorMessage = 'Error with the mutation, data has changed'
    result.errorType = 'MUTATION_ERROR'
    return result
  case 'relatedPosts':
    return relatedPosts[event.source.id]
  default:
    throw new Error('Unknown field, unable to resolve ' + event.field)
}
}

```

傳回個別錯誤

上述範例顯示，您可以從 Lambda 函數傳回單一錯誤，或從回應處理常式引發錯誤。對於批次調用，從 Lambda 函數引發錯誤會將整個批次標記為失敗。對於發生無法復原錯誤的特定案例，例如與資料存放區的連線失敗，這可能是可接受的。不過，如果批次中的某些項目成功，而其他項目失敗，則有可能同時傳回錯誤和有效資料。由於 AWS AppSync 需要批次回應來列出符合批次原始大小的元素，因此您必須定義資料結構，以區分有效資料與錯誤。

例如，如果 Lambda 函數預期會傳回一批相關的文章，您可以選擇傳回 Response 物件清單，其中每個物件都有選用的資料、errorMessage 和 errorType 欄位。如果出現 errorMessage 欄位，表示發生錯誤。

下列程式碼示範如何更新 Lambda 函數：

```

export const handler = async (event) => {
  console.log('Received event {}'.format(JSON.stringify(event, 3)))
  // throw new Error('I fail. always')
  const posts = {
    1: { id: '1', title: 'First book', author: 'Author1', url: 'https://amazon.com/',
      content: 'SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1', ups: '100', downs: '10', },
    2: { id: '2', title: 'Second book', author: 'Author2', url: 'https://amazon.com/',
      content: 'SAMPLE TEXT AUTHOR 2 SAMPLE TEXT AUTHOR 2 SAMPLE TEXT', ups: '100', downs: '10', },
  }
}

```

```

    3: { id: '3', title: 'Third book', author: 'Author3', url: null, content: null,
      ups: null, downs: null },
    4: { id: '4', title: 'Fourth book', author: 'Author4', url: 'https://
www.amazon.com/', content: 'SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT
AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT
AUTHOR 4 SAMPLE TEXT AUTHOR 4', ups: '1000', downs: '0', },
    5: { id: '5', title: 'Fifth book', author: 'Author5', url: 'https://
www.amazon.com/', content: 'SAMPLE TEXT AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT
AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT', ups: '50', downs: '0', },
  }

  const relatedPosts = {
1: [posts['4']],
    2: [posts['3'], posts['5']],
    3: [posts['2'], posts['1']],
    4: [posts['2'], posts['1']],
    5: [],
  }

  if (!event.field && event.length){
console.log(`Got a BatchInvoke Request. The payload has ${event.length} items to
  resolve.`);
    return event.map(e => {
// return an error for post 2
    if (e.source.id === '2') {
return { 'data': null, 'errorMessage': 'Error Happened', 'errorType': 'ERROR' }
      }
      return {data: relatedPosts[e.source.id]}
    })
  }

  console.log('Got an Invoke Request.')
  let result
  switch (event.field) {
case 'getPost':
    return posts[event.arguments.id]
case 'allPosts':
    return Object.values(posts)
case 'addPost':
    // return the arguments back
return event.arguments
    case 'addPostErrorWithData':
      result = posts[event.arguments.id]
      // attached additional error information to the post

```

```

    result.errorMessage = 'Error with the mutation, data has changed'
    result.errorType = 'MUTATION_ERROR'
  }
  return result
  case 'relatedPosts':
    return relatedPosts[event.source.id]
  default:
    throw new Error('Unknown field, unable to resolve ' + event.field)
}
}

```

更新relatedPosts解析程式碼：

```

import { util } from '@aws-appsync/utils';

export function request(ctx) {
  const {source, args} = ctx
  return {
    operation: ctx.info.fieldName === 'relatedPosts' ? 'BatchInvoke' : 'Invoke',
    payload: { field: ctx.info.fieldName, arguments: args, source },
  };
}

export function response(ctx) {
  const { error, result } = ctx;
  if (error) {
    util.appendError(error.message, error.type, result);
  } else if (result.errorMessage) {
    util.appendError(result.errorMessage, result.errorType, result.data)
  } else if (ctx.info.fieldName === 'relatedPosts') {
    return result.data
  } else {
    return result
  }
}

```

回應處理常式現在會檢查Invoke操作時 Lambda 函數傳回的錯誤、檢查BatchInvoke操作個別項目傳回的錯誤，最後檢查 fieldName。對於 relatedPosts，函數會傳回 result.data。對於所有其他欄位，函數只會傳回 result。例如，請參閱下列查詢：

```

query AllPosts {
  allPosts {
    id
  }
}

```



```
    title
    content
    url
    ups
    downs
    relatedPosts {
      id
    }
    author
  }
}
```

此查詢會傳回類似下列的 GraphQL 回應：

```
{
  "data": {
    "allPosts": [
      {
        "id": "1",
        "relatedPosts": [
          {
            "id": "4"
          }
        ]
      },
      {
        "id": "2",
        "relatedPosts": null
      },
      {
        "id": "3",
        "relatedPosts": [
          {
            "id": "2"
          },
          {
            "id": "1"
          }
        ]
      },
      {
        "id": "4",
        "relatedPosts": [
```

```

        {
          "id": "2"
        },
        {
          "id": "1"
        }
      ]
    },
    {
      "id": "5",
      "relatedPosts": []
    }
  ]
},
"errors": [
  {
    "path": [
      "allPosts",
      1,
      "relatedPosts"
    ],
    "data": null,
    "errorType": "ERROR",
    "errorInfo": null,
    "locations": [
      {
        "line": 4,
        "column": 5,
        "sourceName": null
      }
    ],
    "message": "Error Happened"
  }
]
}

```

設定批次大小上限

若要在解析程式上設定批次大小上限，請在 AWS Command Line Interface () 中使用下列命令AWS CLI：

```
$ aws appsync create-resolver --api-id <api-id> --type-name Query --field-name
relatedPosts \
```

```
--code "<code-goes-here>" \  
--runtime name=APPSYNC_JS,runtimeVersion=1.0.0 \  
--data-source-name "<lambda-datasource>" \  
--max-batch-size X
```

Note

提供請求映射範本時，您必須使用 BatchInvoke 操作來使用批次處理。

使用本機解析程式 in AWS AppSync

AWS AppSync 可讓您使用支援的資料來源 (AWS Lambda、Amazon DynamoDB 或 Amazon OpenSearch Service) 來執行各種操作。不過，在某些情況下，可能不需要呼叫支援的資料來源。

這就是本機解析程式的實用之處。本機解析程式只會將請求處理常式的結果轉送到回應處理常式，而不是呼叫遠端資料來源。欄位解析不會 leave AWS AppSync。

本機解析程式在許多情況下都很有用。最常見的使用案例是發佈通知而不觸發資料來源呼叫。為了示範此使用案例，讓我們建置一個 pub/sub 應用程式，使用者可以在其中發佈和訂閱訊息。此範例使用訂閱，因此，如果您不熟悉訂閱，可以遵循[即時資料](#)教學課程。

建立 pub/sub 應用程式

首先，選擇從頭開始設計選項，並在建立 GraphQL API 時設定選用的詳細資訊，以建立空白的 GraphQL API。

在我們的 pub/sub 應用程式中，用戶端可以訂閱和發佈訊息。每個已發佈的訊息都包含名稱和資料。將此新增至結構描述：

```
type Channel {  
  name: String!  
  data: AWSJSON!  
}  
  
type Mutation {  
  publish(name: String!, data: AWSJSON!): Channel  
}  
  
type Query {
```

```
  getChannel: Channel
}

type Subscription {
  subscribe(name: String!): Channel
  @aws_subscribe(mutations: ["publish"])
}
```

接下來，讓我們將解析程式連接到 `Mutation.publish` 欄位。在結構描述窗格旁的解析程式窗格中，尋找 `Mutation` 類型，然後尋找 `publish(...): Channel` 欄位，然後按一下連接。

建立無資料來源並將其命名為 `PageDataSource`。將它連接到您的解析程式。

使用下列程式碼片段新增您的解析程式實作：

```
export function request(ctx) {
  return { payload: ctx.args };
}

export function response(ctx) {
  return ctx.result;
}
```

請確定您建立解析程式並儲存所做的變更。

傳送和訂閱訊息

若要讓用戶端接收訊息，他們必須先訂閱收件匣。

在查詢窗格中，執行 `SubscribeToData` 訂閱：

```
subscription SubscribeToData {
  subscribe(name:"channel") {
    name
    data
  }
}
```

每當叫用 `publish` 變動時，訂閱者都會收到訊息，但只有在訊息傳送到 `channel` 訂閱時才會收到訊息。讓我們在查詢窗格中試試看。當您的訂閱仍在主控台中執行時，請開啟另一個主控台，並在查詢窗格中執行下列請求：

Note

在此範例中，我們使用有效的 JSON 字串。

```
mutation PublishData {
  publish(data: "{\"msg\": \"hello world!\"}", name: "channel") {
    data
    name
  }
}
```

結果看起來像這樣：

```
{
  "data": {
    "publish": {
      "data": "{\"msg\": \"hello world!\"}",
      "name": "channel"
    }
  }
}
```

我們剛示範如何使用本機解析程式，方法是發佈訊息並接收訊息，而不離開 the AWS AppSync 服務。

在 中結合 GraphQL 解析程式 AWS AppSync

GraphQL 結構描述中的解析程式與欄位具有一對一的關係，以及高度的彈性。由於資料來源是在獨立於結構描述的解析程式上設定，因此您可以透過不同的資料來源來解析或操作 GraphQL 類型，讓您混合並比對結構描述，以最符合您的需求。

下列案例示範如何在結構描述中混合和比對資料來源。開始之前，您應該先熟悉如何設定 Amazon DynamoDB 和 AWS Lambda Amazon OpenSearch Service 的資料來源和解析程式。

範例結構描述

下列結構描述的類型為 Post，每個都有三個 Query 和 Mutation 操作：

```
type Post {
```

```
    id: ID!
    author: String!
    title: String
    content: String
    url: String
    ups: Int
    downs: Int
    version: Int!
  }

  type Query {
    allPost: [Post]
    getPost(id: ID!): Post
    searchPosts: [Post]
  }

  type Mutation {
    addPost(
      id: ID!,
      author: String!,
      title: String,
      content: String,
      url: String
    ): Post
    updatePost(
      id: ID!,
      author: String!,
      title: String,
      content: String,
      url: String,
      ups: Int!,
      downs: Int!,
      expectedVersion: Int!
    ): Post
    deletePost(id: ID!): Post
  }
```

在此範例中，您總共會有六個解析程式，每個解析程式都需要資料來源。解決此問題的一種方法是將這些關聯到稱為 `Posts` 的單一 Amazon DynamoDB 資料表，其中 `AllPost` 欄位執行掃描，`searchPosts` 欄位執行查詢（請參閱 [DynamoDB 的 JavaScript 解析程式函數參考](#)）。不過，您並不限於 Amazon DynamoDB；Lambda 或 OpenSearch Service 等不同資料來源都存在，以符合您的業務需求。

透過解析程式變更資料

您可能需要從 AWS AppSync 資料來源未直接支援的第三方資料庫傳回結果。您可能還必須對資料執行複雜的修改，才能傳回 API 用戶端 (API 用戶端)。這可能是由於資料類型的格式不正確，例如用戶端的時間戳記差異，或處理回溯相容性問題。在這種情況下，將 AWS Lambda 函數作為資料來源連接到您的 AWS AppSync API 是適當的解決方案。為了說明起見，在下列範例中，AWS Lambda 函數會操作從第三方資料存放區擷取的資料：

```
export const handler = (event, context, callback) => {  
  // fetch data  
  const result = fetcher()  
  
  // apply complex business logic  
  const data = transform(result)  
  
  // return to AppSync  
  return data  
};
```

這是一個非常有效的 Lambda 函式，可以連結至 GraphQL 結構描述中的 AllPost 欄位，如此，任何傳回所有結果的查詢，都會取得支持票和不支持票票數的隨機數量。

DynamoDB 和 OpenSearch Service

對於某些應用程式，您可以對 DynamoDB 執行變動或簡單查詢，並具有背景程序將文件傳輸至 OpenSearch Service。您可以直接將 searchPosts 解析程式連接至 OpenSearch Service 資料來源，並使用 GraphQL 查詢傳回搜尋結果（來自源自 DynamoDB 的資料）。這在將進階搜尋操作新增至您的應用程式時非常強大，例如關鍵字、模糊字詞配對，甚至是地理空間查詢。可以透過 ETL 程序從 DynamoDB 傳輸資料，或者，您可以使用 Lambda 從 DynamoDB 串流。

若要開始使用這些特定資料來源，請參閱我們的 [DynamoDB](#) 和 [Lambda](#) 教學課程。

例如，使用先前教學課程中的結構描述，下列變動會將項目新增至 DynamoDB：

```
mutation addPost {  
  addPost(  
    id: 123  
    author: "Nadia"  
    title: "Our first post!"  
    content: "This is our first post."  
  )  
}
```

```
url: "https://aws.amazon.com/appsync/"
) {
  id
  author
  title
  content
  url
  ups
  downs
  version
}
}
```

這會將資料寫入 DynamoDB，然後透過 Lambda 將資料串流至 Amazon OpenSearch Service，然後您可以使用它依不同的欄位搜尋文章。例如，由於資料位於 Amazon OpenSearch Service 中，因此您可以使用自由格式文字搜尋作者或內容欄位，即使是空格，如下所示：

```
query searchName{
  searchAuthor(name:"  Nadia  "){
    id
    title
    content
  }
}

----- or -----

query searchContent{
  searchContent(text:"test"){
    id
    title
    content
  }
}
```

由於資料是直接寫入 DynamoDB，因此您仍然可以使用 `allPost{...}` 和 `getPost{...}` 查詢，針對資料表執行有效的清單或項目查詢操作。此堆疊會針對 DynamoDB 串流使用下列範例程式碼：

Note

此 Python 程式碼是範例，並非用於生產程式碼。


```
import boto3
import requests
from requests_aws4auth import AWS4Auth

region = '' # e.g. us-east-1
service = 'es'
credentials = boto3.Session().get_credentials()
awsauth = AWS4Auth(credentials.access_key, credentials.secret_key, region, service,
    session_token=credentials.token)

host = '' # the OpenSearch Service domain, e.g. https://search-mydomain.us-
west-1.es.amazonaws.com
index = 'lambda-index'
datatype = '_doc'
url = host + '/' + index + '/' + datatype + '/'

headers = { "Content-Type": "application/json" }

def handler(event, context):
    count = 0
    for record in event['Records']:
        # Get the primary key for use as the OpenSearch ID
        id = record['dynamodb']['Keys']['id']['S']

        if record['eventName'] == 'REMOVE':
            r = requests.delete(url + id, auth=awsauth)
        else:
            document = record['dynamodb']['NewImage']
            r = requests.put(url + id, auth=awsauth, json=document, headers=headers)
        count += 1
    return str(count) + ' records processed.'
```

然後，您可以使用 DynamoDB 串流將此項目連接至主索引鍵為的 DynamoDB 資料表id，而 DynamoDB 來源的任何變更都會串流到您的 OpenSearch Service 網域。如需進行這項設定的詳細資訊，請參閱 [DynamoDB 串流文件](#)。

在 AWS AppSync 中使用 Amazon OpenSearch Service 解析程式

AWS AppSync 支援從您在自己的 AWS 帳戶中佈建的網域使用 Amazon OpenSearch Service，前提是這些網域不存在於 VPC 中。在佈建網域之後，您可以使用資料來源來連線到這些網域，此時您可以

在結構描述中設定解析程式，來進行 GraphQL 操作，例如查詢、變動和訂閱。本教學課程將會逐步說明一些常見的範例。

如需詳細資訊，請參閱 [OpenSearch 的 JavaScript 解析程式函數參考](#)。

建立新的 OpenSearch Service 網域

若要開始使用本教學課程，您需要現有的 OpenSearch Service 網域。如果尚未擁有，您可以使用下列範例。請注意，建立 OpenSearch Service 網域最多可能需要 15 分鐘，然後才能繼續將其與 an AWS AppSync 資料來源整合。

```
aws cloudformation create-stack --stack-name AppSyncOpenSearch \
--template-url https://s3.us-west-2.amazonaws.com/awsappsync/resources/elasticsearch/
ESResolverCFTemplate.yaml \
--parameters ParameterKey=OSDomainName,ParameterValue=ddtestdomain
ParameterKey=Tier,ParameterValue=development \
--capabilities CAPABILITY_NAMED_IAM
```

您可以在帳戶中 AWS 的 US-West-2（奧勒岡）區域中啟動下列 AWS CloudFormation 堆疊：

Launch Stack 

設定 OpenSearch Service 的資料來源

建立 OpenSearch Service 網域後，導覽至 your AWS AppSync GraphQL API，然後選擇資料來源索引標籤。選擇建立資料來源，然後輸入資料來源的易記名稱，例如「**oss**」。然後，為資料來源類型選擇 Amazon OpenSearch 網域，選擇適當的區域，您應該會看到列出的 OpenSearch Service 網域。選取後，您可以建立新的角色，而 AWS AppSync 會指派角色適當的許可，或者您可以選擇具有下列內嵌政策的現有角色：

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "Stmt1234234",
      "Effect": "Allow",
      "Action": [
```

```

        "es:ESHttpDelete",
        "es:ESHttpHead",
        "es:ESHttpGet",
        "es:ESHttpPost",
        "es:ESHttpPut"
    ],
    "Resource": [
        "arn:aws:es:REGION:ACCOUNTNUMBER:domain/democluster/*"
    ]
}
]
}

```

您還需要為該角色設定與 AWS AppSync 的信任關係：

JSON

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "appsync.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}

```

此外，OpenSearch Service 網域有自己的存取政策，您可以透過 Amazon OpenSearch Service 主控台進行修改。您必須為 OpenSearch Service 網域新增類似以下的政策，以及適當的動作和資源。請注意，主體將是 AWS AppSync 資料來源角色，如果您讓該主控台建立，可以在 IAM 主控台中找到該角色。

JSON

```

{
  "Version": "2012-10-17",

```

```

    "Statement": [
      {
        "Effect": "Allow",
        "Principal": {
          "AWS": "arn:aws:iam::ACCOUNTNUMBER:role/service-role/
APPSYNC_DATASOURCE_ROLE"
        },
        "Action": [
          "es:ESHttpDelete",
          "es:ESHttpHead",
          "es:ESHttpGet",
          "es:ESHttpPost",
          "es:ESHttpPut"
        ],
        "Resource": "arn:aws:es:REGION:ACCOUNTNUMBER:domain/DOMAIN_NAME/*"
      }
    ]
  }
}

```

連接解析程式

現在資料來源已連線至 OpenSearch Service 網域，您可以使用解析程式將其連線至 GraphQL 結構描述，如下列範例所示：

```

type Query {
  getPost(id: ID!): Post
  allPosts: [Post]
}

type Mutation {
  addPost(id: ID!, author: String, title: String, url: String, ups: Int, downs: Int,
content: String): AWSJSON
}

type Post {
  id: ID!
  author: String
  title: String
  url: String
  ups: Int
  downs: Int
  content: String
}

```

```
}

```

請注意，有使用者定義的 `Post` 類型與 `id` 欄位。在下列範例中，我們假設有一個程序（可以自動化）可將此類型放入您的 OpenSearch Service 網域，這會映射到索引 `/post/_docpost` 所在的路徑根目錄。在此根路徑中，您可以執行個別文件搜尋、使用 `進行萬用字元搜尋/id/post*`，或使用的路徑進行多文件搜尋 `/post/_search`。例如，如果您有另一種名為的類型 `User`，您可以在名為的新索引下編製文件索引 `user`，然後使用路徑執行搜尋 `/user/_search`。

從 AWS AppSync Posts 主控台的結構描述編輯器中，修改先前的結構描述以包含 `searchPosts` 查詢：

```
type Query {
  getPost(id: ID!): Post
  allPosts: [Post]
  searchPosts: [Post]
}
```

儲存結構描述。在解析程式窗格中，尋找 `searchPosts` 並選擇連接。選擇您的 OpenSearch Service 資料來源並儲存解析程式。使用以下程式碼片段更新解析程式的程式碼：

```
import { util } from '@aws-appsync/utils'

/**
 * Searches for documents by using an input term
 * @param {import('@aws-appsync/utils').Context} ctx the context
 * @returns {*} the request
 */
export function request(ctx) {
  return {
    operation: 'GET',
    path: `/post/_search`,
    params: { body: { from: 0, size: 50 } },
  }
}

/**
 * Returns the fetched items
 * @param {import('@aws-appsync/utils').Context} ctx the context
 * @returns {*} the result
 */
export function response(ctx) {
  if (ctx.error) {
```

```
    util.error(ctx.error.message, ctx.error.type)
  }
  return ctx.result.hits.hits.map((hit) => hit._source)
}
```

這會假設上述結構描述的文件已在 OpenSearch Service 的 post 欄位下編製索引。如果您以不同的方式建構資料，您將需要相應地更新。

修改您的搜尋

上述解析程式請求處理常式會針對所有記錄執行簡單的查詢。假設您想要根據特定的作者來進行搜尋。此外，假設您希望該作者成為 GraphQL 查詢中定義的引數。在 AWS AppSync 主控台的結構描述編輯器中，新增allPostsByAuthor查詢：

```
type Query {
  getPost(id: ID!): Post
  allPosts: [Post]
  allPostsByAuthor(author: String!): [Post]
  searchPosts: [Post]
}
```

在解析程式窗格中，尋找allPostsByAuthor並選擇連接。選擇 OpenSearch Service 資料來源，並使用下列程式碼：

```
import { util } from '@aws-appsync/utils'

/**
 * Searches for documents by `author`
 * @param {import('@aws-appsync/utils').Context} ctx the context
 * @returns {*} the request
 */
export function request(ctx) {
  return {
    operation: 'GET',
    path: '/post/_search',
    params: {
      body: {
        from: 0,
        size: 50,
        query: { match: { author: ctx.args.author } },
      },
    },
  },
}
```

```

    }
  }

  /**
   * Returns the fetched items
   * @param {import('@aws-appsync/utils').Context} ctx the context
   * @returns {*} the result
   */
  export function response(ctx) {
    if (ctx.error) {
      util.error(ctx.error.message, ctx.error.type)
    }
    return ctx.result.hits.hits.map((hit) => hit._source)
  }

```

請注意，body 會填入 author 欄位的查詢詞語，此欄位會做為引數，從用戶端直接傳來。或者，您可以使用預先填入的資訊，例如標準文字。

將資料新增至 OpenSearch Service

由於 GraphQL 變動，您可能想要將資料新增至 OpenSearch Service 網域。這是一個具有搜尋和其他用途的強大機制。由於您可以使用 GraphQL 訂閱[讓資料即時](#)，因此它可以做為通知用戶端 OpenSearch Service 網域中資料更新的機制。

返回 AWS AppSync 主控台中的結構描述頁面，然後選取附加以進行 `addPost()` 變動。再次選取 OpenSearch Service 資料來源，並使用下列程式碼：

```

import { util } from '@aws-appsync/utils'

/**
 * Searches for documents by `author`
 * @param {import('@aws-appsync/utils').Context} ctx the context
 * @returns {*} the request
 */
export function request(ctx) {
  return {
    operation: 'PUT',
    path: `/post/_doc/${ctx.args.id}`,
    params: { body: ctx.args },
  }
}

```

```

/**
 * Returns the inserted post
 * @param {import('@aws-appsync/utils').Context} ctx the context
 * @returns {*} the result
 */
export function response(ctx) {
  if (ctx.error) {
    util.error(ctx.error.message, ctx.error.type)
  }
  return ctx.result
}

```

如同之前，這是資料結構的範例。如果您有不同的欄位名稱或索引，則需要更新 path 和 body。此範例也會示範如何使用 context.arguments，也可以在請求處理常式 ctx.args 中寫入為。

擷取單一文件

最後，如果您想要在結構描述中使用 getPost(id:ID) 查詢來傳回個別文件，請在 AWS AppSync 主控台的結構描述編輯器中尋找此查詢，然後選擇連接。再次選取 OpenSearch Service 資料來源，並使用下列程式碼：

```

import { util } from '@aws-appsync/utils'

/**
 * Searches for documents by `author`
 * @param {import('@aws-appsync/utils').Context} ctx the context
 * @returns {*} the request
 */
export function request(ctx) {
  return {
    operation: 'GET',
    path: `/post/_doc/${ctx.args.id}`,
  }
}

/**
 * Returns the post
 * @param {import('@aws-appsync/utils').Context} ctx the context
 * @returns {*} the result
 */
export function response(ctx) {
  if (ctx.error) {
    util.error(ctx.error.message, ctx.error.type)
  }
}

```



```
}  
return ctx.result._source  
}
```

執行查詢和變動

您現在應該能夠對 OpenSearch Service 網域執行 GraphQL 操作。導覽至 AWS AppSync 主控台的查詢索引標籤，並新增記錄：

```
mutation AddPost {  
  addPost (  
    id:"12345"  
    author: "Fred"  
    title: "My first book"  
    content: "This will be fun to write!"  
    url: "publisher website",  
    ups: 100,  
    downs:20  
  )  
}
```

您會在右側看到變動的結果。同樣地，您現在可以針對 OpenSearch Service 網域執行searchPosts查詢：

```
query search {  
  searchPosts {  
    id  
    title  
    author  
    content  
  }  
}
```

最佳實務

- OpenSearch Service 應該用於查詢資料，而不是做為您的主要資料庫。您可能想要將 OpenSearch Service 與 Amazon DynamoDB 搭配使用，如[合併 GraphQL 解析程式](#)中所述。
- 只有在允許 AWS AppSync 服務角色存取叢集時，才允許 存取您的網域。
- 您可以在開發時少量使用最低成本的叢集，然後在進入正式生產時，轉移到具有高可用性 (HA) 的較大叢集。

在 AWS AppSync 中執行 DynamoDB 交易

AWS AppSync 支援在單一區域中的一或多個資料表中使用 Amazon DynamoDB 交易操作。支援的操作包括 `TransactGetItems` 和 `TransactWriteItems`。透過使用這些 in AWS AppSync 功能，您可以執行下列任務：

- 在單一查詢中傳遞金鑰清單，並從資料表傳回結果
- 從單一查詢中的一或多個資料表讀取記錄
- 以all-or-nothing方式將交易中的記錄寫入一或多個資料表
- 滿足某些條件時執行交易

許可

如同其他解析程式，您需要建立資料來源 in AWS AppSync，並建立角色或使用現有的角色。由於交易操作在 DynamoDB 資料表上需要不同的許可，因此您需要授予已設定的角色讀取或寫入動作的許可：

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "dynamodb:DeleteItem",
        "dynamodb:GetItem",
        "dynamodb:PutItem",
        "dynamodb:Query",
        "dynamodb:Scan",
        "dynamodb:UpdateItem"
      ],
      "Effect": "Allow",
      "Resource": [
        "arn:aws:dynamodb:region:accountId:table/TABLENAME",
        "arn:aws:dynamodb:region:accountId:table/TABLENAME/*"
      ]
    }
  ]
}
```

 Note

角色會繫結至 in AWS AppSync 中的資料來源，而 欄位上的解析程式會針對資料來源叫用。設定為針對 DynamoDB 擷取的資料來源只有一個指定的資料表，以保持組態簡單。因此，在單一解析程式中針對多個資料表執行交易操作時 (這是一項更為進階的工作)，您必須授予權限給該資料來源上的角色，以存取解析程式將會與其互動的所有資料表。這會在上述 IAM 政策的 Resource (資源) 欄位中完成。針對資料表的交易呼叫組態是在解析程式程式碼中完成，如下所述。

資料來源

為簡單起見，我們將針對本教學課程中使用的所有解析程式，使用相同的資料來源。

我們將有兩個名為 `savingAccounts` 和 `checkingAccounts` 的資料表，兩個資料表都使用 `accountNumber` 做為分割區索引鍵，而使用 `transactionId` 做為分割區索引鍵的 `transactionHistory` 資料表。您可以使用以下 CLI 命令來建立資料表。請務必將 `region` 取代為您的區域。

使用 CLI

```
aws dynamodb create-table --table-name savingAccounts \
  --attribute-definitions AttributeName=accountNumber,AttributeType=S \
  --key-schema AttributeName=accountNumber,KeyType=HASH \
  --provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5 \
  --table-class STANDARD --region region

aws dynamodb create-table --table-name checkingAccounts \
  --attribute-definitions AttributeName=accountNumber,AttributeType=S \
  --key-schema AttributeName=accountNumber,KeyType=HASH \
  --provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5 \
  --table-class STANDARD --region region

aws dynamodb create-table --table-name transactionHistory \
  --attribute-definitions AttributeName=transactionId,AttributeType=S \
  --key-schema AttributeName=transactionId,KeyType=HASH \
  --provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5 \
  --table-class STANDARD --region region
```

在 AWS AppSync 主控台的資料來源中，建立新的 DynamoDB 資料來源，並將其命名為 `TransactTutorial`。選取 `savingAccounts` 做為資料表（雖然使用交易時特定資料表並不重要）。選擇

以建立新的角色和資料來源。您可以檢閱資料來源組態，以查看產生的角色名稱。在 IAM 主控台中，您可以新增內嵌政策，允許資料來源與所有資料表互動。

將 `region` 和 `accountID` 為您的區域和帳戶 ID：

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "dynamodb:DeleteItem",
        "dynamodb:GetItem",
        "dynamodb:PutItem",
        "dynamodb:Query",
        "dynamodb:Scan",
        "dynamodb:UpdateItem"
      ],
      "Effect": "Allow",
      "Resource": [
        "arn:aws:dynamodb:region:accountId:table/savingAccounts",
        "arn:aws:dynamodb:region:accountId:table/savingAccounts/*",
        "arn:aws:dynamodb:region:accountId:table/checkingAccounts",
        "arn:aws:dynamodb:region:accountId:table/checkingAccounts/*",
        "arn:aws:dynamodb:region:accountId:table/transactionHistory",
        "arn:aws:dynamodb:region:accountId:table/transactionHistory/*"
      ]
    }
  ]
}
```

交易

在此範例中，內容是傳統的銀行交易，我們將使用 `TransactWriteItems` 以進行下列操作：

- 從存款帳戶轉帳至支票帳戶
- 為每筆交易產生新的交易記錄

然後，我們會使用 `TransactGetItems` 擷取存款帳戶和支票帳戶中的詳細資料。

 Warning

TransactWriteItems 與衝突偵測和解決搭配使用時，不支援。必須停用這些設定，以防止可能的錯誤。

定義 GraphQL 結構描述，如下所示：

```
type SavingAccount {
  accountNumber: String!
  username: String
  balance: Float
}

type CheckingAccount {
  accountNumber: String!
  username: String
  balance: Float
}

type TransactionHistory {
  transactionId: ID!
  from: String
  to: String
  amount: Float
}

type TransactionResult {
  savingAccounts: [SavingAccount]
  checkingAccounts: [CheckingAccount]
  transactionHistory: [TransactionHistory]
}

input SavingAccountInput {
  accountNumber: String!
  username: String
  balance: Float
}

input CheckingAccountInput {
  accountNumber: String!
  username: String
  balance: Float
}
```

```

}

input TransactionInput {
  savingAccountNumber: String!
  checkingAccountNumber: String!
  amount: Float!
}

type Query {
  getAccounts(savingAccountNumbers: [String], checkingAccountNumbers: [String]):
  TransactionResult
}

type Mutation {
  populateAccounts(savingAccounts: [SavingAccountInput], checkingAccounts:
  [CheckingAccountInput]): TransactionResult
  transferMoney(transactions: [TransactionInput]): TransactionResult
}

```

TransactWriteItems - 填入帳戶

為了在帳戶之間轉帳，我們需要在表格中填入詳細資料。我們將使用 GraphQL 操作 `Mutation.populateAccounts` 來執行此作業。

在結構描述區段中，按一下 `Mutation.populateAccounts` 操作旁的連接。選擇 `TransactTutorial` 資料來源，然後選擇建立。

現在請使用下列程式碼：

```

import { util } from '@aws-appsync/utils'

export function request(ctx) {
  const { savingAccounts, checkingAccounts } = ctx.args

  const savings = savingAccounts.map(({ accountNumber, ...rest }) => {
    return {
      table: 'savingAccounts',
      operation: 'PutItem',
      key: util.dynamodb.toMapValues({ accountNumber }),
      attributeValues: util.dynamodb.toMapValues(rest),
    }
  })
}

```

```

const checkings = checkingAccounts.map(({ accountNumber, ...rest }) => {
  return {
    table: 'checkingAccounts',
    operation: 'PutItem',
    key: util.dynamodb.toMapValues({ accountNumber }),
    attributeValues: util.dynamodb.toMapValues(rest),
  }
})
return {
  version: '2018-05-29',
  operation: 'TransactWriteItems',
  transactItems: [...savings, ...checkings],
}
}

export function response(ctx) {
  if (ctx.error) {
    util.error(ctx.error.message, ctx.error.type, null, ctx.result.cancellationReasons)
  }
  const { savingAccounts: sInput, checkingAccounts: cInput } = ctx.args
  const keys = ctx.result.keys
  const savingAccounts = sInput.map((_, i) => keys[i])
  const sLength = sInput.length
  const checkingAccounts = cInput.map((_, i) => keys[sLength + i])
  return { savingAccounts, checkingAccounts }
}

```

儲存解析程式並導覽至 AWS AppSync 主控台的查詢區段以填入帳戶。

執行下列的變動：

```

mutation populateAccounts {
  populateAccounts (
    savingAccounts: [
      {accountNumber: "1", username: "Tom", balance: 100},
      {accountNumber: "2", username: "Amy", balance: 90},
      {accountNumber: "3", username: "Lily", balance: 80},
    ]
    checkingAccounts: [
      {accountNumber: "1", username: "Tom", balance: 70},
      {accountNumber: "2", username: "Amy", balance: 60},
      {accountNumber: "3", username: "Lily", balance: 50},
    ]) {
    savingAccounts {

```

```

    accountNumber
  }
  checkingAccounts {
    accountNumber
  }
}
}

```

我們在一個變動中填入三個儲存帳戶和三個檢查帳戶。

使用 DynamoDB 主控台來驗證資料是否同時顯示在 savingAccounts 和 checkingAccounts 資料表中。

TransactWriteItems - 轉帳

使用下列程式碼將解析程式連接至transferMoney變動。對於每次轉移，我們需要檢查和節省帳戶的成功修改器，而且我們需要追蹤交易中的轉移。

```

import { util } from '@aws-appsync/utils'

export function request(ctx) {
  const transactions = ctx.args.transactions

  const savings = []
  const checkings = []
  const history = []
  transactions.forEach((t) => {
    const { savingAccountNumber, checkingAccountNumber, amount } = t
    savings.push({
      table: 'savingAccounts',
      operation: 'UpdateItem',
      key: util.dynamodb.toMapValues({ accountNumber: savingAccountNumber }),
      update: {
        expression: 'SET balance = balance - :amount',
        expressionValues: util.dynamodb.toMapValues({ ':amount': amount }),
      },
    })
    checkings.push({
      table: 'checkingAccounts',
      operation: 'UpdateItem',
      key: util.dynamodb.toMapValues({ accountNumber: checkingAccountNumber }),
      update: {
        expression: 'SET balance = balance + :amount',
        expressionValues: util.dynamodb.toMapValues({ ':amount': amount }),
      },
    })
  })
}

```



```

    },
  })
  history.push({
    table: 'transactionHistory',
    operation: 'PutItem',
    key: util.dynamodb.toMapValues({ transactionId: util.autoId() }),
    attributeValues: util.dynamodb.toMapValues({
      from: savingAccountNumber,
      to: checkingAccountNumber,
      amount,
    }),
  })
})

return {
  version: '2018-05-29',
  operation: 'TransactWriteItems',
  transactItems: [...savings, ...checkings, ...history],
}
}

export function response(ctx) {
  if (ctx.error) {
    util.error(ctx.error.message, ctx.error.type, null, ctx.result.cancellationReasons)
  }
  const tInput = ctx.args.transactions
  const tLength = tInput.length
  const keys = ctx.result.keys
  const savingAccounts = tInput.map((_, i) => keys[tLength * 0 + i])
  const checkingAccounts = tInput.map((_, i) => keys[tLength * 1 + i])
  const transactionHistory = tInput.map((_, i) => keys[tLength * 2 + i])
  return { savingAccounts, checkingAccounts, transactionHistory }
}

```

現在，導覽至 AWS AppSync 主控台的查詢區段，並執行 transferMoney 變動，如下所示：

```

mutation write {
  transferMoney(
    transactions: [
      {savingAccountNumber: "1", checkingAccountNumber: "1", amount: 7.5},
      {savingAccountNumber: "2", checkingAccountNumber: "2", amount: 6.0},
      {savingAccountNumber: "3", checkingAccountNumber: "3", amount: 3.3}
    ]) {

```

```

    savingAccounts {
      accountNumber
    }
    checkingAccounts {
      accountNumber
    }
    transactionHistory {
      transactionId
    }
  }
}

```

我們以一個變動傳送三個銀行交易。使用 DynamoDB 主控台來驗證資料是否顯示在 savingAccounts、checkingAccounts 和 transactionHistory 資料表中。

TransactGetItems - 擷取帳戶

為了從單一交易請求中的節省和檢查帳戶擷取詳細資訊，我們會將解析程式連接到結構描述上的 Query.getAccounts GraphQL 操作。選取連接，挑選在教學課程開始時建立的相同 TransactTutorial 資料來源。使用下列程式碼：

```

import { util } from '@aws-appsync/utils'

export function request(ctx) {
  const { savingAccountNumbers, checkingAccountNumbers } = ctx.args

  const savings = savingAccountNumbers.map((accountNumber) => {
    return { table: 'savingAccounts', key: util.dynamodb.toMapValues({ accountNumber }) }
  })
  const checkings = checkingAccountNumbers.map((accountNumber) => {
    return { table: 'checkingAccounts', key:
util.dynamodb.toMapValues({ accountNumber }) }
  })
  return {
    version: '2018-05-29',
    operation: 'TransactGetItems',
    transactItems: [...savings, ...checkings],
  }
}

export function response(ctx) {
  if (ctx.error) {
    util.error(ctx.error.message, ctx.error.type, null, ctx.result.cancellationReasons)
  }
}

```

```
}

const { savingAccountNumbers: sInput, checkingAccountNumbers: cInput } = ctx.args
const items = ctx.result.items
const savingAccounts = sInput.map((_, i) => items[i])
const sLength = sInput.length
const checkingAccounts = cInput.map((_, i) => items[sLength + i])
return { savingAccounts, checkingAccounts }
}
```

儲存解析程式並導覽至 AWS AppSync 主控台的查詢區段。若要擷取節省和檢查帳戶，請執行下列查詢：

```
query getAccounts {
  getAccounts(
    savingAccountNumbers: ["1", "2", "3"],
    checkingAccountNumbers: ["1", "2"]
  ) {
    savingAccounts {
      accountNumber
      username
      balance
    }
    checkingAccounts {
      accountNumber
      username
      balance
    }
  }
}
```

我們已成功使用 AWS AppSync 示範 DynamoDB 交易的使用方式。

在 AWS AppSync 中使用 DynamoDB 批次操作

AWS AppSync 支援在單一區域中的一或多個資料表中使用 Amazon DynamoDB 批次操作。支援的操作包括 BatchGetItem、BatchPutItem 和 BatchDeleteItem。透過使用這些 in AWS AppSync 功能，您可以執行下列任務：

- 在單一查詢中傳遞金鑰清單，並從資料表傳回結果
- 從單一查詢中的一或多個資料表讀取記錄

- 大量寫入記錄至一或多個資料表
- 有條件地在可能有關聯的多個資料表中寫入或刪除記錄

批次操作 in AWS AppSync 與非批次操作有兩個主要差異：

- 資料來源角色必須具有解析程式將存取的所有資料表的許可。
- 解析程式的資料表規格是請求物件的一部分。

單一資料表批次

Warning

BatchPutItem 與衝突偵測和解決搭配使用時，不支援 和 BatchDeleteItem。必須停用這些設定，以防止可能的錯誤。

若要開始使用，讓我們建立新的 GraphQL API。在 AWS AppSync 主控台中，選擇從頭開始建立 API、GraphQL APIs 和設計。為您的 API 命名 BatchTutorial API，選擇下一步，然後在指定 GraphQL 資源步驟中，選擇稍後建立 GraphQL 資源，然後按一下下一步。檢閱您的詳細資訊並建立 API。前往結構描述頁面並貼上下列結構描述，請注意，對於查詢，我們將傳遞 IDs 清單：

```
type Post {
  id: ID!
  title: String
}

input PostInput {
  id: ID!
  title: String
}

type Query {
  batchGet(ids: [ID]): [Post]
}

type Mutation {
  batchAdd(posts: [PostInput]): [Post]
  batchDelete(ids: [ID]): [Post]
}
```

儲存您的結構描述，然後選擇頁面頂端的建立資源。選擇使用現有類型，然後選取Post類型。為您的資料表命名 Posts。確定主索引鍵設定為 id，取消選取自動產生 GraphQL（您將提供自己的程式碼），然後選取建立。為了協助您開始使用，AWS AppSync 會建立新的 DynamoDB 資料表，以及連接至具有適當角色之資料表的資料來源。不過，您仍然需要將一些許可新增至角色。前往資料來源頁面，然後選擇新的資料來源。在選取現有角色下，您會注意到已自動為資料表建立角色。請記下角色（看起來像 appsync-ds-ddb-aaabbbccdd-Posts），然後前往 IAM 主控台 (<https://console.aws.amazon.com/iam/> : //)。在 IAM 主控台中，選擇角色，然後從資料表中選擇您的角色。在您的角色中，在許可政策下，按一下政策旁的「+」（應該具有與角色名稱類似的名稱）。當政策出現時，選擇可摺疊頂端的編輯。您需要將批次許可新增至政策，特別是 dynamodb:BatchGetItem 和 dynamodb:BatchWriteItem。這看起來會像這樣：

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "dynamodb:DeleteItem",
        "dynamodb:GetItem",
        "dynamodb:PutItem",
        "dynamodb:Query",
        "dynamodb:Scan",
        "dynamodb:UpdateItem",
        "dynamodb:BatchWriteItem",
        "dynamodb:BatchGetItem"
      ],
      "Resource": [
        "arn:aws:dynamodb:...",
        "arn:aws:dynamodb:..."
      ]
    }
  ]
}
```

選擇下一步，然後選擇儲存變更。您的政策現在應該允許批次處理。

返回 AWS AppSync 主控台，前往結構描述頁面，然後選取 Mutation.batchAdd 欄位旁的連接。使用 Posts 資料表做為資料來源來建立您的解析程式。在程式碼編輯器中，將處理常式取代為下面

的程式碼片段。此程式碼片段會自動取得 GraphQL input `PostInput` 類型中的每個項目，並建置 `BatchPutItem` 操作所需的映射：

```
import { util } from "@aws-appsync/utils";

export function request(ctx) {
  return {
    operation: "BatchPutItem",
    tables: {
      Posts: ctx.args.posts.map((post) => util.dynamodb.toMapValues(post)),
    },
  };
}

export function response(ctx) {
  if (ctx.error) {
    util.error(ctx.error.message, ctx.error.type);
  }
  return ctx.result.data.Posts;
}
```

導覽至 AWS AppSync 主控台的查詢頁面，並執行下列 `batchAdd` 變動：

```
mutation add {
  batchAdd(posts:[{
    id: 1 title: "Running in the Park"},{
    id: 2 title: "Playing fetch"
  }]){
    id
    title
  }
}
```

您應該會在畫面上看到列印的結果；這可以透過檢閱 DynamoDB 主控台來掃描寫入 `Posts` 資料表的值來驗證。

接下來，重複連接解析程式的程序，但對於使用 `Posts` 資料表做為資料來源 `Query.batchGet` 的欄位。使用下列程式碼取代處理常式。這會自動擷取 GraphQL `ids: []` 類型的每個項目，並建置 `BatchGetItem` 作業所需的對應圖：

```
import { util } from "@aws-appsync/utils";
```

```
export function request(ctx) {
  return {
    operation: "BatchGetItem",
    tables: {
      Posts: {
        keys: ctx.args.ids.map((id) => util.dynamodb.toMapValues({ id })),
        consistentRead: true,
      },
    },
  };
}

export function response(ctx) {
  if (ctx.error) {
    util.error(ctx.error.message, ctx.error.type);
  }
  return ctx.result.data.Posts;
}
```

現在，返回 AWS AppSync 主控台的查詢頁面，並執行下列batchGet查詢：

```
query get {
  batchGet(ids:[1,2,3]){
    id
    title
  }
}
```

這應該會針對您先前所新增的兩個 id 值，傳回其結果。請注意，傳回null的值id值為 3。這是因為您的Posts資料表中尚無該值的記錄。另請注意，AWS AppSync 會以與傳遞給查詢的金鑰相同的順序傳回結果，這是 AWS AppSync 代表您執行的額外功能。因此，如果您切換到 batchGet(ids:[1,3,2])，您會看到訂單已變更。您也會知道哪個 id 傳回 null 值。

最後，使用Posts資料表做為資料來源，將另一個解析程式連接到 Mutation.batchDelete 欄位。使用下列程式碼取代處理常式。這會自動擷取 GraphQL ids:[] 類型的每個項目，並建置 BatchGetItem 作業所需的對應圖：

```
import { util } from "@aws-appsync/utils";

export function request(ctx) {
  return {
    operation: "BatchDeleteItem",
```

```
    tables: {
      Posts: ctx.args.ids.map((id) => util.dynamodb.toMapValues({ id })),
    },
  };
}

export function response(ctx) {
  if (ctx.error) {
    util.error(ctx.error.message, ctx.error.type);
  }
  return ctx.result.data.Posts;
}
```

現在，返回 AWS AppSync 主控台的查詢頁面，並執行下列 `batchDelete` 變動：

```
mutation delete {
  batchDelete(ids:[1,2]){ id }
}
```

`id 1` 和 `2` 的記錄現在應已刪除。如果重新執行先前的 `batchGet()` 查詢，這些應該會傳回 `null`。

多資料表批次

Warning

`BatchPutItem` 與衝突偵測和解決搭配使用時，不支援 `BatchDeleteItem`。必須停用這些設定，以防止可能的錯誤。

AWS AppSync 也可讓您跨資料表執行批次操作。來試試建置更複雜的應用程式。假設我們正在建置寵物運作狀態應用程式，其中感應器會報告寵物的位置和身體溫度。感應器是由電池供電，而且每隔幾分鐘就會試著連線到網路。當感應器建立連線時，它會將其讀數傳送至我們的 AWS AppSync API。觸發條件接著就會分析資料，然後將儀表板呈現給寵物的主人。讓我們著重介紹感應器與後端資料存放區之間的互動。

在 AWS AppSync 主控台中，選擇從頭開始建立 API、GraphQL APIs 和設計。為您的 API 命名 `MultiBatchTutorial` API，選擇下一步，然後在指定 GraphQL 資源步驟中，選擇稍後建立 GraphQL 資源，然後按一下下一步。檢閱您的詳細資訊並建立 API。前往結構描述頁面，貼上並儲存下列結構描述：

```
type Mutation {
```



```
# Register a batch of readings
recordReadings(tempReadings: [TemperatureReadingInput], locReadings:
[LocationReadingInput]): RecordResult
# Delete a batch of readings
deleteReadings(tempReadings: [TemperatureReadingInput], locReadings:
[LocationReadingInput]): RecordResult
}

type Query {
  # Retrieve all possible readings recorded by a sensor at a specific time
  getReadings(sensorId: ID!, timestamp: String!): [SensorReading]
}

type RecordResult {
  temperatureReadings: [TemperatureReading]
  locationReadings: [LocationReading]
}

interface SensorReading {
  sensorId: ID!
  timestamp: String!
}

# Sensor reading representing the sensor temperature (in Fahrenheit)
type TemperatureReading implements SensorReading {
  sensorId: ID!
  timestamp: String!
  value: Float
}

# Sensor reading representing the sensor location (lat,long)
type LocationReading implements SensorReading {
  sensorId: ID!
  timestamp: String!
  lat: Float
  long: Float
}

input TemperatureReadingInput {
  sensorId: ID!
  timestamp: String
  value: Float
}
```

```
input LocationReadingInput {  
  sensorId: ID!  
  timestamp: String  
  lat: Float  
  long: Float  
}
```

我們需要建立兩個 DynamoDB 資料表：

- locationReadings 將存放感應器位置讀數。
- temperatureReadings 將存放感應器溫度讀數。

兩個資料表將共用相同的主索引鍵結構：sensorId (String) 作為分割區索引鍵，timestamp (String) 作為排序索引鍵。

選擇頁面頂端的建立資源。選擇使用現有類型，然後選取locationReadings類型。為您的資料表命名 locationReadings。確定主索引鍵設定為 sensorId，排序索引鍵設定為 timestamp。取消選取自動產生 GraphQL（您將提供自己的程式碼），然後選取建立。temperatureReadings 使用 temperatureReadings 作為類型和資料表名稱，重複此程序。使用與上述相同的金鑰。

您的新資料表將包含自動產生的角色。您仍然需要將一些許可新增至這些角色。前往資料來源頁面，然後選擇 locationReadings。在選取現有角色下，您可以看到角色。請記下角色（看起來像 appsync-ds-ddb-aaabbbcccd-dd-locationReadings），然後前往 IAM 主控台 (<https://console.aws.amazon.com/iam/> : //)。在 IAM 主控台中，選擇角色，然後從資料表中選擇您的角色。在您的角色中，在許可政策下，按一下政策旁的「+」（應該具有與角色名稱類似的名稱）。當政策出現時，選擇可摺疊頂端的編輯。您需要將許可新增至此政策。這看起來會像這樣：

JSON

```
{  
  "Version": "2012-10-17",  
  "Statement": [  
    {  
      "Effect": "Allow",  
      "Action": [  
        "dynamodb:DeleteItem",  
        "dynamodb:GetItem",  
        "dynamodb:PutItem",  
        "dynamodb:Query",
```

```

        "dynamodb:Scan",
        "dynamodb:UpdateItem",
        "dynamodb:BatchGetItem",
        "dynamodb:BatchWriteItem"
    ],
    "Resource": [
        "arn:aws:dynamodb:region:account:table/locationReadings",
        "arn:aws:dynamodb:region:account:table/locationReadings/*",
        "arn:aws:dynamodb:region:account:table/temperatureReadings",
        "arn:aws:dynamodb:region:account:table/temperatureReadings/*"
    ]
}
]
}

```

選擇下一步，然後選擇儲存變更。使用上述相同的政策程式碼片段，為temperatureReadings資料來源重複此程序。

BatchPutItem - 記錄感應器讀數

感應器必須能夠在連線到網際網路之後傳送其讀數。感應器將使用 GraphQL 欄位 `Mutation.recordReadings` 這個 API 來執行此項動作。我們需要將解析程式新增至此欄位。

在 AWS AppSync 主控台的結構描述頁面中，選取 `Mutation.recordReadings` 欄位旁的連接。在下一個畫面上，使用 `locationReadings` 資料表做為資料來源來建立您的解析程式。

建立解析程式之後，請在編輯器中將處理常式取代為下列程式碼。BatchPutItem 此操作允許我們指定多個資料表：

```

import { util } from '@aws-appsync/utils'

export function request(ctx) {
  const { locReadings, tempReadings } = ctx.args
  const locationReadings = locReadings.map((loc) => util.dynamodb.toMapValues(loc))
  const temperatureReadings = tempReadings.map((tmp) => util.dynamodb.toMapValues(tmp))

  return {
    operation: 'BatchPutItem',
    tables: {
      locationReadings,
      temperatureReadings,
    },
  },
}

```

```
}  
}  
  
export function response(ctx) {  
  if (ctx.error) {  
    util.appendError(ctx.error.message, ctx.error.type)  
  }  
  return ctx.result.data  
}
```

進行批次操作時，呼叫可能會同時傳回錯誤和結果。在這種情況中，我們可以隨意進行一些額外的錯誤處理。

Note

的使用 `utils.appendError()` 類似於 `util.error()`，主要區別在於它不會中斷請求或回應處理常式的評估。反之，它會發出欄位發生錯誤的訊號，但允許評估處理常式，並將資料傳回給發起人。當您的應用程式需要傳回部分結果 `utils.appendError()` 時，建議您使用。

儲存解析程式並導覽至 AWS AppSync 主控台中的查詢頁面。我們現在可以傳送一些感應器讀數。

執行下列的變動：

```
mutation sendReadings {  
  recordReadings(  
    tempReadings: [  
      {sensorId: 1, value: 85.5, timestamp: "2018-02-01T17:21:05.000+08:00"},  
      {sensorId: 1, value: 85.7, timestamp: "2018-02-01T17:21:06.000+08:00"},  
      {sensorId: 1, value: 85.8, timestamp: "2018-02-01T17:21:07.000+08:00"},  
      {sensorId: 1, value: 84.2, timestamp: "2018-02-01T17:21:08.000+08:00"},  
      {sensorId: 1, value: 81.5, timestamp: "2018-02-01T17:21:09.000+08:00"}  
    ]  
    locReadings: [  
      {sensorId: 1, lat: 47.615063, long: -122.333551, timestamp:  
"2018-02-01T17:21:05.000+08:00"},  
      {sensorId: 1, lat: 47.615163, long: -122.333552, timestamp:  
"2018-02-01T17:21:06.000+08:00"},  
      {sensorId: 1, lat: 47.615263, long: -122.333553, timestamp:  
"2018-02-01T17:21:07.000+08:00"},  
      {sensorId: 1, lat: 47.615363, long: -122.333554, timestamp:  
"2018-02-01T17:21:08.000+08:00"},  
    ]  
  )  
}
```

```

      {sensorId: 1, lat: 47.615463, long: -122.333555, timestamp:
"2018-02-01T17:21:09.000+08:00"}
    }) {
      locationReadings {
        sensorId
        timestamp
        lat
        long
      }
      temperatureReadings {
        sensorId
        timestamp
        value
      }
    }
  }
}
```

我們在一個變動中傳送十個感應器讀數，並將讀數分割為兩個資料表。使用 DynamoDB 主控台來驗證資料是否同時顯示在 `locationReadings` 和 `temperatureReadings` 資料表中。

BatchDeleteItem - 刪除感應器讀數

同樣地，我們也需要能夠刪除感應器讀數的批次。讓我們使用 `Mutation.deleteReadings` GraphQL 欄位來進行這項動作。在 AWS AppSync 主控台的結構描述頁面中，選取 `Mutation.deleteReadings` 欄位旁的連接。在下一個畫面上，使用 `locationReadings` 資料表做為資料來源來建立您的解析程式。

建立解析程式之後，請將程式碼編輯器中的處理常式取代為下方的程式碼片段。在此解析程式中，我們使用協助程式函數映射器，從提供的輸入 `timestamp` 中擷取 `sensorId` 和。

```

import { util } from '@aws-appsync/utils'

export function request(ctx) {
  const { locReadings, tempReadings } = ctx.args
  const mapper = ({ sensorId, timestamp }) => util.dynamodb.toMapValues({ sensorId,
timestamp })

  return {
    operation: 'BatchDeleteItem',
    tables: {
      locationReadings: locReadings.map(mapper),
      temperatureReadings: tempReadings.map(mapper),
    }
  }
}
```

```
    },  
  }  
}  
  
export function response(ctx) {  
  if (ctx.error) {  
    util.appendError(ctx.error.message, ctx.error.type)  
  }  
  return ctx.result.data  
}
```

儲存解析程式並導覽至 AWS AppSync 主控台其中的查詢頁面。現在，讓我們刪除幾個感應器讀數。

執行下列的變動：

```
mutation deleteReadings {  
  # Let's delete the first two readings we recorded  
  deleteReadings(  
    tempReadings: [{sensorId: 1, timestamp: "2018-02-01T17:21:05.000+08:00"}]  
    locReadings: [{sensorId: 1, timestamp: "2018-02-01T17:21:05.000+08:00"}]) {  
    locationReadings {  
      sensorId  
      timestamp  
      lat  
      long  
    }  
    temperatureReadings {  
      sensorId  
      timestamp  
      value  
    }  
  }  
}
```

Note

不同於 DeleteItem 操作，回應中未傳回完整刪除的項目。只會傳回已傳遞的索引鍵。若要進一步了解，請參閱 [DynamoDB 的 JavaScript 解析程式函數參考中的 BatchDeleteItem](#)。

透過 DynamoDB 主控台驗證這兩個讀數是否已從 locationReadings 和 temperatureReadings 資料表中刪除。

BatchGetItem - 擷取讀數

我們應用程式的另一個常見操作是在特定時間點擷取感應器的讀數。試試將解析程式連接到結構描述中的 `Query.getReadings` GraphQL 欄位。在 AWS AppSync 主控台的結構描述頁面中，選取 `Query.getReadings` 欄位旁的連接。在下一個畫面上，使用 `locationReadings` 資料表做為資料來源來建立您的解析程式。

讓我們使用下列程式碼：

```
import { util } from '@aws-appsync/utils'

export function request(ctx) {
  const keys = [util.dynamodb.toMapValues(ctx.args)]
  const consistentRead = true
  return {
    operation: 'BatchGetItem',
    tables: {
      locationReadings: { keys, consistentRead },
      temperatureReadings: { keys, consistentRead },
    },
  }
}

export function response(ctx) {
  if (ctx.error) {
    util.appendError(ctx.error.message, ctx.error.type)
  }
  const { locationReadings: locs, temperatureReadings: temps } = ctx.result.data

  return [
    ...locs.map((l) => ({ ...l, __typename: 'LocationReading' })),
    ...temps.map((t) => ({ ...t, __typename: 'TemperatureReading' })),
  ]
}
```

儲存解析程式並導覽至 AWS AppSync 主控台內的查詢頁面。現在，讓我們擷取感應器讀數。

執行下列的查詢：

```
query getReadingsForSensorAndTime {
  # Let's retrieve the very first two readings
  getReadings(sensorId: 1, timestamp: "2018-02-01T17:21:06.000+08:00") {
```

```
    sensorId
    timestamp
    ...on TemperatureReading {
      value
    }
    ...on LocationReading {
      lat
      long
    }
  }
}
```

我們已成功示範如何使用 AWS AppSync 進行 DynamoDB 批次操作。

錯誤處理

In AWS AppSync 中，資料來源操作有時可能會傳回部分結果。我們將會使用部分結果這個詞彙，來表示操作的輸出包含一些資料和錯誤的情況。由於錯誤處理本質上是應用程式特定的，AWS AppSync 讓您有機會處理回應處理常式中的錯誤。如果發生解析程式呼叫錯誤，在文字內容中會出現 `ctx.error`。呼叫錯誤一律包含訊息和類型，可做為 `ctx.error.message` 和 `ctx.error.type` 屬性存取。在回應處理常式中，您可以透過三種方式處理部分結果：

1. 只需傳回資料，即可省略呼叫錯誤。
2. 透過停止處理常式評估來引發錯誤（使用 `util.error(...)`），這不會傳回任何資料。
3. 附加錯誤（使用 `util.appendError(...)`）並傳回資料。

讓我們使用 DynamoDB 批次操作來示範上述三個點的每一個。

DynamoDB 批次操作

使用 DynamoDB 批次操作時，批次作業就有可能部分完成。也就是說，請求的項目或索引鍵可以有一些尚未處理完成。如果 AWS AppSync 無法完成批次，未處理的項目將在內容上設定調用錯誤。

我們將會使用 `Query.getReadings` 欄位組態（取自於本教學課程先前區段所說明的 `BatchGetItem` 操作）來建置錯誤處理功能。這次，讓我們假設在執行 `Query.getReadings` 欄位時，`temperatureReadings` DynamoDB 資料表用盡了佈建的輸送量。DynamoDB 在第二次嘗試 by AWS AppSync `ProvisionedThroughputExceededException` 期間引發，以處理批次中剩餘的元素。

下列 JSON 代表 DynamoDB 批次調用之後但在呼叫回應處理常式之前序列化的內容：


```
{
  "arguments": {
    "sensorId": "1",
    "timestamp": "2018-02-01T17:21:05.000+08:00"
  },
  "source": null,
  "result": {
    "data": {
      "temperatureReadings": [
        null
      ],
      "locationReadings": [
        {
          "lat": 47.615063,
          "long": -122.333551,
          "sensorId": "1",
          "timestamp": "2018-02-01T17:21:05.000+08:00"
        }
      ]
    },
    "unprocessedKeys": {
      "temperatureReadings": [
        {
          "sensorId": "1",
          "timestamp": "2018-02-01T17:21:05.000+08:00"
        }
      ],
      "locationReadings": []
    }
  },
  "error": {
    "type": "DynamoDB:ProvisionedThroughputExceededException",
    "message": "You exceeded your maximum allowed provisioned throughput for a table or for one or more global secondary indexes. (...)"
  },
  "outErrors": []
}
```

對於內容的幾個注意事項：

- 調用錯誤已在 `ctx.error` by AWS AppSync 的內容上設定，且錯誤類型已設定為 `DynamoDB:ProvisionedThroughputExceededException`。

- `ctx.result.data` 即使發生錯誤，也會在下映射每個資料表的結果。
- 未處理的金鑰可在取得 `ctx.result.data.unprocessedKeys`。在這裡，AWS AppSync 無法使用金鑰 (`sensorId: 1`、時間戳記: `2018-02-01T17:21:05.000+08:00`) 擷取項目，因為資料表輸送量不足。

Note

如果是 `BatchPutItem`，則為 `ctx.result.data.unprocessedItems`。如果是 `BatchDeleteItem`，則為 `ctx.result.data.unprocessedKeys`。

讓我們用三種不同的方式來處理這項錯誤。

1. 抑制呼叫錯誤

傳回資料而不處理呼叫錯誤，可有效地抑制錯誤，讓指定 GraphQL 欄位的結果一律成功。

我們編寫的程式碼很熟悉，只著重於結果資料。

回應處理常式

```
export function response(ctx) {  
  return ctx.result.data  
}
```

GraphQL 回應

```
{  
  "data": {  
    "getReadings": [  
      {  
        "sensorId": "1",  
        "timestamp": "2018-02-01T17:21:05.000+08:00",  
        "lat": 47.615063,  
        "long": -122.333551  
      },  
      {  
        "sensorId": "1",  
        "timestamp": "2018-02-01T17:21:05.000+08:00",  
        "value": 85.5  
      }  
    ]  
  }  
}
```

```
    ]  
  }  
}
```

錯誤回應中不會加入任何錯誤，因為只處理了資料。

2. 引發錯誤以中止回應處理常式執行

從用戶端的角度來看，當部分失敗應視為完全失敗時，您可以中止回應處理常式執行，以防止傳回資料。`util.error(...)` 公用程式方法可確實完成這項動作。

回應處理常式程式碼

```
export function response(ctx) {  
  if (ctx.error) {  
    util.error(ctx.error.message, ctx.error.type, null,  
      ctx.result.data.unprocessedKeys);  
  }  
  return ctx.result.data;  
}
```

GraphQL 回應

```
{  
  "data": {  
    "getReadings": null  
  },  
  "errors": [  
    {  
      "path": [  
        "getReadings"  
      ],  
      "data": null,  
      "errorType": "DynamoDB:ProvisionedThroughputExceededException",  
      "errorInfo": {  
        "temperatureReadings": [  
          {  
            "sensorId": "1",  
            "timestamp": "2018-02-01T17:21:05.000+08:00"  
          }  
        ],  
        "locationReadings": []  
      }  
    }  
  ],  
}
```

```

    "locations": [
      {
        "line": 58,
        "column": 3
      }
    ],
    "message": "You exceeded your maximum allowed provisioned throughput for a table
or for one or more global secondary indexes. (...)"
  }
]
}

```

即使 DynamoDB 批次作業可能已經傳回一些結果，我們還是選擇丟出錯誤，如此 `getReadings` GraphQL 欄位為 `null`，而錯誤也會加入到 GraphQL 回應的 `errors` (錯誤) 區塊。

3. 附加錯誤，以同時傳回資料和錯誤

在某些情況中，為提供更好的使用者體驗，應用程式可以傳回部分結果，並通知其用戶端有未處理的項目。用戶端可以決定是否要重試，或是轉譯錯誤後傳回給最終使用者。`util.appendError(...)` 是公用程式方法，可讓應用程式設計工具在內容上附加錯誤，而不會干擾回應處理常式的評估，以啟用此行為。評估回應處理常式後，AWS AppSync 會將任何內容錯誤附加至 GraphQL 回應的錯誤區塊，以處理這些錯誤。

回應處理常式程式碼

```

export function response(ctx) {
  if (ctx.error) {
    util.appendError(ctx.error.message, ctx.error.type, null,
ctx.result.data.unprocessedKeys);
  }
  return ctx.result.data;
}

```

我們轉送 GraphQL 回應錯誤區塊內的調用錯誤和 `unprocessedKeys` 元素。`getReadings` 欄位也會傳回 `locationReadings` 資料表中的部分資料，如下方回應所示。

GraphQL 回應

```

{
  "data": {
    "getReadings": [
      null,

```

```
{
  {
    "sensorId": "1",
    "timestamp": "2018-02-01T17:21:05.000+08:00",
    "value": 85.5
  }
],
},
"errors": [
  {
    "path": [
      "getReadings"
    ],
    "data": null,
    "errorType": "DynamoDB:ProvisionedThroughputExceededException",
    "errorInfo": {
      "temperatureReadings": [
        {
          "sensorId": "1",
          "timestamp": "2018-02-01T17:21:05.000+08:00"
        }
      ],
      "locationReadings": []
    },
    "locations": [
      {
        "line": 58,
        "column": 3
      }
    ],
    "message": "You exceeded your maximum allowed provisioned throughput for a table
or for one or more global secondary indexes. (...)"
  }
]
}
```

使用 HTTP 解析程式 in AWS AppSync

AWS AppSync 可讓您使用支援的資料來源（即 Amazon DynamoDB、AWS Lambda、Amazon OpenSearch Service 或 Amazon Aurora）來執行各種操作，以及任何任意 HTTP 端點來解析 GraphQL 欄位。在您的 HTTP 端點可供使用之後，您可以使用資料來源連線到這些端點。然後，您可以在結構描述中設定解析程式以執行 GraphQL 操作，例如查詢、變動和訂閱。本教學將逐步說明一些常見的範例。

在本教學課程中，您將 REST API（使用 Amazon API Gateway 和 Lambda 建立）與 AWS AppSync GraphQL 端點搭配使用。

建立 REST API

您可以使用下列 AWS CloudFormation 範本來設定適用於本教學課程的 REST 端點：



AWS CloudFormation 堆疊會執行下列步驟：

1. 設定 Lambda 函數，其中包含您微服務的商業邏輯。
2. 使用下列endpoint/method/content類型組合設定 API Gateway REST API：

API 資源路徑	HTTP 方法	已支援的內容類型
/v1/users	POST	application/json
/v1/users	GET	application/json
/v1/users/1	GET	application/json
/v1/users/1	PUT	application/json
/v1/users/1	DELETE	application/json

建立 GraphQL API

若要在 AWS AppSync 中建立 GraphQL API：

1. 開啟 AWS AppSync 主控台，然後選擇建立 API。
2. 選擇 GraphQL APIs，然後選擇從頭開始設計。選擇下一步。
3. 針對 API 名稱，輸入 UserData。選擇下一步。
4. 選擇 Create GraphQL resources later。選擇下一步。
5. 檢閱您的輸入，然後選擇建立 API。

AWS AppSync 主控台會使用 API 金鑰身分驗證模式為您建立新的 GraphQL API。您可以使用 主控台 進一步設定 GraphQL API 並執行請求。

建立 GraphQL 結構描述

現在您有一個 GraphQL API，讓我們來建立 GraphQL 結構描述吧。在 AWS AppSync 主控台的結構描述編輯器中，使用下列程式碼片段：

```
type Mutation {
  addUser(userInput: UserInput!): User
  deleteUser(id: ID!): User
}

type Query {
  getUser(id: ID): User
  listUser: [User!]!
}

type User {
  id: ID!
  username: String!
  firstname: String
  lastname: String
  phone: String
  email: String
}

input UserInput {
  id: ID!
  username: String!
  firstname: String
  lastname: String
  phone: String
  email: String
}
```

設定 HTTP 資料來源

若要設定您的 HTTP 資料來源，請執行以下作業：

1. 在 your AWS AppSync GraphQL API 的資料來源頁面中，選擇建立資料來源。
2. 輸入資料來源的名稱，例如 HTTP_Example。

3. 在資料來源類型中，選擇 HTTP 端點。
4. 將端點設定為教學課程開始時建立的 API Gateway 端點。如果您導覽至 Lambda 主控台並在應用程式下尋找應用程式，您可以找到堆疊產生的端點。在應用程式的設定中，您應該會看到 API 端點，該端點將是您的端點 AWS AppSync。請確定您不包含階段名稱做為端點的一部分。例如，如果您的端點是 `https://aaabbbcccd.execute-api.us-east-1.amazonaws.com/v1`，您會在 中輸入 `https://aaabbbcccd.execute-api.us-east-1.amazonaws.com`。

Note

目前，AWS AppSync 僅支援公有端點。

如需 AWS AppSync 服務所辨識認證授權機構的詳細資訊，請參閱 [AWS AppSync HTTPS 端點的 所辨識的憑證授權機構 \(CA\)](#)。

設定解析程式

在此步驟中，您將將 HTTP 資料來源連線至 `getUser` 和 `addUser` 查詢。

若要設定 `getUser` 解析程式：

1. 在 your AWS AppSync GraphQL API 中，選擇結構描述索引標籤。
2. 在結構描述編輯器右側，在解析程式窗格中和查詢類型下，尋找 `getUser` 欄位，然後選擇連接。
3. 將解析程式類型保留為 `Unit` 並將執行時間保留為 `APPSYNC_JS`。
4. 在資料來源名稱中，選擇您先前建立的 HTTP 端點。
5. 選擇建立。
6. 在解析程式程式碼編輯器中，新增下列程式碼片段做為您的請求處理常式：

```
import { util } from '@aws-appsync/utils'

export function request(ctx) {
  return {
    version: '2018-05-29',
    method: 'GET',
    params: {
      headers: {
        'Content-Type': 'application/json',
      },
    },
  },
}
```



```
},  
resourcePath: `/v1/users/${ctx.args.id}`,  
}  
}
```

7. 新增下列程式碼片段做為您的回應處理常式：

```
export function response(ctx) {  
  const { statusCode, body } = ctx.result  
  // if response is 200, return the response  
  if (statusCode === 200) {  
    return JSON.parse(body)  
  }  
  // if response is not 200, append the response to error block.  
  util.appendError(body, statusCode)  
}
```

8. 選擇 Query (查詢) 標籤，然後執行以下查詢：

```
query GetUser{  
  getUser(id:1){  
    id  
    username  
  }  
}
```

這應該會傳回以下回應：

```
{  
  "data": {  
    "getUser": {  
      "id": "1",  
      "username": "nadia"  
    }  
  }  
}
```

若要設定addUser解析程式：

1. 選擇 Schema (結構描述) 標籤。
2. 在結構描述編輯器右側，在解析程式窗格中和查詢類型下，尋找 addUser 欄位，然後選擇連接。

3. 將解析程式類型保留為 `Unit` 並將執行時間保留為 `APPSYNC_JS`。
4. 在資料來源名稱中，選擇您先前建立的 HTTP 端點。
5. 選擇建立。
6. 在解析程式程式碼編輯器中，新增下列程式碼片段做為您的請求處理常式：

```
export function request(ctx) {
  return {
    "version": "2018-05-29",
    "method": "POST",
    "resourcePath": "/v1/users",
    "params": {
      "headers": {
        "Content-Type": "application/json"
      },
      "body": ctx.args.userInput
    }
  }
}
```

7. 新增下列程式碼片段做為您的回應處理常式：

```
export function response(ctx) {
  if(ctx.error) {
    return util.error(ctx.error.message, ctx.error.type)
  }
  if (ctx.result.statusCode == 200) {
    return ctx.result.body
  } else {
    return util.appendError(ctx.result.body, "ctx.result.statusCode")
  }
}
```

8. 選擇 Query (查詢) 標籤，然後執行以下查詢：

```
mutation addUser{
  addUser(userInput:{
    id:"2",
    username:"shaggy"
  }){
    id
    username
  }
}
```

```
}
```

如果您再次執行getUser查詢，應傳回下列回應：

```
{
  "data": {
    "getUser": {
      "id": "2",
      "username": "shaggy"
    }
  }
}
```

叫用 AWS 服務

您可以使用 HTTP 解析程式來設定 AWS 服務的 GraphQL API 介面。對的 HTTP 請求 AWS 必須使用 [Signature 第 4 版程序](#) 簽署，以便 AWS 可以識別傳送這些請求的人員。當您將 IAM 角色與 HTTP 資料來源建立關聯時，AWS AppSync 會代表您計算簽章。

您提供兩個額外的元件，以使用 HTTP 解析程式叫用 AWS 服務：

- 具有呼叫 AWS 服務 APIs 許可的 IAM 角色
- 資料來源中的簽署組態

例如，如果您想要使用 HTTP 解析程式呼叫 [ListGraphqlApis 操作](#)，請先[建立 AppSync 擔任並連接下列政策的 IAM 角色](#)：AWS AppSync

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "appsync:ListGraphqlApis"
      ],
      "Effect": "Allow",
      "Resource": "*"
    }
  ]
}
```

```
    ]
  }
}
```

接著，建立 HTTP 資料來源 for AWS AppSync。在此範例中，您在美國西部（奧勒岡）區域呼叫 AWS AppSync。在名為 `http.json` 的檔案中設定以下 HTTP 組態，包含簽署區域和服務名稱：

```
{
  "endpoint": "https://appsync.us-west-2.amazonaws.com/",
  "authorizationConfig": {
    "authorizationType": "AWS_IAM",
    "awsIamConfig": {
      "signingRegion": "us-west-2",
      "signingServiceName": "appsync"
    }
  }
}
```

然後，使用 AWS CLI 建立具有關聯角色的資料來源，如下所示：

```
aws appsync create-data-source --api-id <API-ID> \
                               --name AWSAppSync \
                               --type HTTP \
                               --http-config file:///http.json \
                               --service-role-arn <ROLE-ARN>
```

當您將解析程式連接至結構描述中的欄位時，請使用下列請求映射範本來 call AWS AppSync：

```
{
  "version": "2018-05-29",
  "method": "GET",
  "resourcePath": "/v1/apis"
}
```

當您為此資料來源執行 GraphQL 查詢時，AWS AppSync 會使用您提供的角色簽署請求，並在請求中包含簽章。查詢會傳回您帳戶中該 AWS 區域中的 AWS AppSync GraphQL APIs 清單。

在 中使用 Aurora PostgreSQL 搭配資料 API AWS AppSync

了解如何使用 將您的 GraphQL API 連線至 Aurora PostgreSQL 資料庫 AWS AppSync。此整合可讓您透過 GraphQL 操作執行 SQL 查詢和變動，以建置可擴展的資料驅動型應用程式。AWS AppSync 提供資料來源，以針對使用資料 API 啟用的 Amazon Aurora 叢集執行 SQL 陳述式。您可以使用 AWS AppSync 解析程式，針對具有 GraphQL 查詢、變動和訂閱的資料 API 執行 SQL 陳述式。

開始本教學課程之前，您應該對 AWS 服務和 GraphQL 概念有基本的熟悉。

Note

此教學會使用 US-EAST-1 區域。

主題

- [設定您的 Aurora PostgreSQL 資料庫](#)
- [建立資料庫和資料表](#)
- [建立 GraphQL 結構描述](#)
- [RDS 的解析程式](#)
- [刪除叢集](#)

設定您的 Aurora PostgreSQL 資料庫

將 Amazon RDS 資料來源新增至 之前 AWS AppSync，請執行下列動作。

1. 在 Aurora Serverless v2 叢集上啟用資料 API。
2. 使用 設定秘密 AWS Secrets Manager
3. 使用下列 AWS CLI 命令建立叢集。

```
aws rds create-db-cluster \  
    --db-cluster-identifier appsync-tutorial \  
    --engine aurora-postgresql \  
    --engine-version 16.6 \  
    --serverless-v2-scaling-configuration MinCapacity=0,MaxCapacity=1 \  
    --master-username USERNAME \  
    --master-user-password COMPLEX_PASSWORD \  
    --enable-http-endpoint
```

這會傳回叢集的 ARN。建立叢集之後，您必須使用下列 AWS CLI 命令新增 Serverless v2 執行個體。

```
aws rds create-db-instance \  
  --db-cluster-identifier appsync-tutorial \  
  --db-instance-identifier appsync-tutorial-instance-1 \  
  --db-instance-class db.serverless \  
  --engine aurora-postgresql
```

Note

這些端點需要一些時間才能啟用。您可以在叢集的連線與安全索引標籤中的 RDS 主控台中檢查其狀態。

使用以下 AWS CLI 命令檢查叢集狀態。

```
aws rds describe-db-clusters \  
  --db-cluster-identifier appsync-tutorial \  
  --query "DBClusters[0].Status"
```

使用 USERNAME 和 COMPLEX_PASSWORD 上一個步驟中的，透過 AWS Secrets Manager 主控台或 AWS CLI 使用輸入檔案建立秘密，如下所示：

```
{  
  "username": "USERNAME",  
  "password": "COMPLEX_PASSWORD"  
}
```

將此做為參數傳遞至 AWS CLI：

```
aws secretsmanager create-secret \  
  --name appsync-tutorial-rds-secret \  
  --secret-string file://creds.json
```

這會傳回秘密的 ARN。記下 Aurora Serverless v2 叢集的 ARN 和 Secret，以供稍後在 AWS AppSync 主控台中建立資料來源時使用。

建立資料庫和資料表

首先，建立名為 `TESTDB` 的資料庫。在 PostgreSQL 中，資料庫是存放資料表和其他 SQL 物件的容器。驗證您的 Aurora Serverless v2 叢集已正確設定，再將其新增至您的 AWS AppSync API。首先，使用 `--sql` 參數建立 `TESTDB` 資料庫，如下所示。

```
aws rds-data execute-statement \  
  --resource-arn "arn:aws:rds:us-east-1:111122223333 ISN:cluster:appsync-tutorial" \  
  --secret-arn "arn:aws:secretsmanager:us-east-1:111122223333 ISN:secret:appsync-  
tutorial-rds-secret" \  
  --sql "create DATABASE \"testdb\"" \  
  --database "postgres"
```

如果執行時沒有任何錯誤，請使用 `create table` 命令新增兩個資料表：

```
aws rds-data execute-statement \  
  --resource-arn "arn:aws:rds:us-east-1:111122223333 ISN:cluster:appsync-tutorial" \  
  --secret-arn "arn:aws:secretsmanager:us-east-1:111122223333 ISN:secret:appsync-  
tutorial-rds-secret" \  
  --database "testdb" \  
  --sql 'create table public.todos (id serial constraint todos_pk primary key,  
description text not null, due date not null, "createdAt" timestamp default now());'  
  
aws rds-data execute-statement \  
  --resource-arn "arn:aws:rds:us-east-1:111122223333 ISN:cluster:appsync-tutorial" \  
  --secret-arn "arn:aws:secretsmanager:us-east-1:111122223333 ISN:secret:appsync-  
tutorial-rds-secret" \  
  --database "testdb" \  
  --sql 'create table public.tasks (id serial constraint tasks_pk primary key,  
description varchar, "todoId" integer not null constraint tasks_todos_id_fk references  
public.todos);'
```

如果成功，請在 API 中將叢集新增為資料來源。

建立 GraphQL 結構描述

現在您的 Aurora Serverless v2 Data API 正在執行已設定的資料表，我們將建立 GraphQL 結構描述。您可以使用 API 建立精靈從現有資料庫匯入資料表組態，以快速建立 API。

若要開始：

1. 在 AWS AppSync 主控台中，選擇建立 API，然後從 Amazon Aurora 叢集開始。
2. 指定 API 詳細資訊，例如 API 名稱，然後選取您的資料庫以產生 API。
3. 選擇您的資料庫。如有需要，請更新 區域，然後選擇您的 Aurora 叢集和 TESTDB 資料庫。
4. 選擇您的秘密，然後選擇匯入。
5. 一旦發現資料表，請更新類型名稱。Todos 將 變更為 Todo，將 Tasks 變更為 Task。

6. 選擇預覽結構描述來預覽產生的結構描述。您的結構描述看起來會像這樣：

```
type Todo {
  id: Int!
  description: String!
  due: AWSDate!
  createdAt: String
}

type Task {
  id: Int!
  todoId: Int!
  description: String
}
```

7. 對於角色，您可以 AWS AppSync 建立新的角色，或使用類似下列的政策來建立角色：

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "rds-data:ExecuteStatement",
      ],
      "Resource": [
        "arn:aws:rds:us-east-1:111122223333 ISN:cluster:appsync-tutorial",
        "arn:aws:rds:us-east-1:111122223333 ISN:cluster:appsync-tutorial:*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "secretsmanager:GetSecretValue"
      ],
      "Resource": [
        "arn:aws:secretsmanager:us-east-1:111122223333
ISN:secret:your:secret:arn:appsync-tutorial-rds-secret",
        "arn:aws:secretsmanager:us-east-1:111122223333
ISN:secret:your:secret:arn:appsync-tutorial-rds-secret:*"
      ]
    }
  ]
}
```



```
}
```

請注意，您授予角色存取權的政策中有兩個陳述式。第一個資源是您的 Aurora 叢集，第二個資源是您的 AWS Secrets Manager ARN。

選擇下一步，檢閱組態詳細資訊，然後選擇建立 API。您現在擁有可完全操作的 API。您可以在結構描述頁面上檢閱 API 的完整詳細資訊。

RDS 的解析程式

API 建立流程會自動建立解析程式，以與我們的類型互動。如果您查看結構描述頁面，您會找到一些混沌的解析程式。

- `todo` 透過 `Mutation.createTodo` 欄位建立。
- `todo` 透過 `Mutation.updateTodo` 欄位更新。
- `todo` 透過 `Mutation.deleteTodo` 欄位刪除。
- `todo` 透過 `Query.getTodo` 欄位取得單一。
- `todos` 透過 `Query.listTodos` 欄位列出所有。

您將找到為 `Task` 類型連接的類似欄位和解析程式。讓我們進一步了解一些解析程式。

`Mutation.createTodo`

從 AWS AppSync 主控台的結構描述編輯器右側，選擇 `testdb` 旁的 `createTodo(...): Todo`。解析程式程式碼使用 `rds` 模組的 `insert` 函數，動態建立將資料新增至 `todos` 資料表的插入陳述式。由於我們使用 `Postgres`，因此我們可以利用 `returning` 陳述式來取回插入的資料。

更新下列解析程式，以正確指定 `due` 欄位的 `DATE` 類型。

```
import { util } from '@aws-appsync/utils';
import { insert, createPgStatement, toJsonObject, typeHint } from '@aws-appsync/utils/rds';

export function request(ctx) {
  const { input } = ctx.args;
  // if a due date is provided, cast is as `DATE`
  if (input.due) {
    input.due = typeHint.DATE(input.due)
  }
}
```

```

    const insertStatement = insert({
      table: 'todos',
      values: input,
      returning: '*',
    });
    return createPgStatement(insertStatement)
  }

  export function response(ctx) {
    const { error, result } = ctx;
    if (error) {
      return util.appendError(
        error.message,
        error.type,
        result
      )
    }
    return toJsonObject(result)[0][0]
  }

```

儲存解析程式。類型提示會將輸入物件中的 `due` 正確標記為 `DATE` 類型。這可讓 Postgres 引擎正確解釋值。接著，更新您的結構描述，`id` 從 `CreateTodo` 輸入中移除。由於 Postgres 資料庫可以傳回產生的 ID，因此您可以依賴它來建立，並以單一請求傳回結果，如下所示。

```

input CreateTodoInput {
  due: AWSDate!
  createdAt: String
  description: String!
}

```

進行變更並更新您的結構描述。前往查詢編輯器，將項目新增至資料庫，如下所示。

```

mutation CreateTodo {
  createTodo(input: {description: "Hello World!", due: "2023-12-31"}) {
    id
    due
    description
    createdAt
  }
}

```

您會取得下列結果。

```
{
  "data": {
    "createTodo": {
      "id": 1,
      "due": "2023-12-31",
      "description": "Hello World!",
      "createdAt": "2023-11-14 20:47:11.875428"
    }
  }
}
```

Query.listTodos

從主控台的結構描述編輯器右側，選擇 `testdb` 旁的 `listTodos(id: ID!): Todo`。請求處理常式會使用選取公用程式函數，在執行時間動態建置請求。

```
export function request(ctx) {
  const { filter = {}, limit = 100, nextToken } = ctx.args;
  const offset = nextToken ? +util.base64Decode(nextToken) : 0;
  const statement = select({
    table: 'todos',
    columns: '*',
    limit,
    offset,
    where: filter,
  });
  return createPgStatement(statement)
}
```

我們希望 `todos` 根據 `due` 日期進行篩選。讓我們更新解析程式，將 `due` 值轉換為 `DATE`。更新匯入和請求處理常式的清單，如下所示。

```
import { util } from '@aws-appsync/utils';
import * as rds from '@aws-appsync/utils/rds';

export function request(ctx) {
  const { filter: where = {}, limit = 100, nextToken } = ctx.args;
  const offset = nextToken ? +util.base64Decode(nextToken) : 0;

  // if `due` is used in a filter, CAST the values to DATE.
  if (where.due) {
    Object.entries(where.due).forEach(([k, v]) => {
```

```

    if (k === 'between') {
      where.due[k] = v.map((d) => rds.typeHint.DATE(d));
    } else {
      where.due[k] = rds.typeHint.DATE(v);
    }
  });
}

const statement = rds.select({
  table: 'todos',
  columns: '*',
  limit,
  offset,
  where,
});
return rds.createPgStatement(statement);
}

export function response(ctx) {
  const {
    args: { limit = 100, nextToken },
    error,
    result,
  } = ctx;
  if (error) {
    return util.appendError(error.message, error.type, result);
  }
  const offset = nextToken ? +util.base64Decode(nextToken) : 0;
  const items = rds.toJsonObject(result)[0];
  const endOfResults = items?.length < limit;
  const token = endOfResults ? null : util.base64Encode(`${offset + limit}`);
  return { items, nextToken: token };
}

```

在查詢編輯器中執行下列動作。

```

query LIST {
  listTodos(limit: 10, filter: {due: {between: ["2021-01-01", "2025-01-02"]}}) {
    items {
      id
      due
      description
    }
  }
}

```

```
}
}
```

Mutation.updateTodo

您也可以update使用 Todo。從查詢編輯器，讓我們更新第一個 id Todo項目1。

```
mutation UPDATE {
  updateTodo(input: {id: 1, description: "edits"}) {
    description
    due
    id
  }
}
```

請注意，您必須指定要更新之項目id的。您也可以指定條件，只更新符合特定條件的項目。例如，如果描述以 開頭，我們可能只想要編輯項目edits，如下所示。

```
mutation UPDATE {
  updateTodo(input: {id: 1, description: "edits: make a change", condition:
    {description: {beginsWith: "edits"}}}) {
    description
    due
    id
  }
}
```

就像我們處理 create和 list操作的方式一樣，我們可以更新解析程式，將 due 欄位轉換為 DATE。將這些變更儲存至 updateTodo，如下所示。

```
import { util } from '@aws-appsync/utils';
import * as rds from '@aws-appsync/utils/rds';

export function request(ctx) {
  const { input: { id, ...values }, condition = {}, } = ctx.args;
  const where = { ...condition, id: { eq: id } };

  // if `due` is used in a condition, CAST the values to DATE.
  if (condition.due) {
    Object.entries(condition.due).forEach(([k, v]) => {
      if (k === 'between') {
```

```

        condition.due[k] = v.map((d) => rds.typeHint.DATE(d));
    } else {
        condition.due[k] = rds.typeHint.DATE(v);
    }
});
}

// if a due date is provided, cast is as `DATE`
if (values.due) {
    values.due = rds.typeHint.DATE(values.due);
}

const updateStatement = rds.update({
    table: 'todos',
    values,
    where,
    returning: '*',
});
return rds.createPgStatement(updateStatement);
}

export function response(ctx) {
    const { error, result } = ctx;
    if (error) {
        return util.appendError(error.message, error.type, result);
    }
    return rds.toJsonObject(result)[0][0];
}

```

現在請嘗試使用 條件進行更新：

```

mutation UPDATE {
  updateTodo(
    input: {
      id: 1, description: "edits: make a change", due: "2023-12-12",
      condition: {
        description: {beginsWith: "edits"}, due: {ge: "2023-11-08"}}
    }
  ) {
    description
    due
    id
  }
}

```

Mutation.deleteTodo

您可以Todo具有 deleteTodo 變動的 delete 。這的運作方式與updateTodo變動類似，而且您必須指定您要刪除之項目id的，如下所示。

```
mutation DELETE {  
  deleteTodo(input: {id: 1}) {  
    description  
    due  
    id  
  }  
}
```

撰寫自訂查詢

我們已使用rds模組公用程式來建立 SQL 陳述式。我們也可以撰寫自己的自訂靜態陳述式來與資料庫互動。首先，更新結構描述以從CreateTask輸入中移除 id 欄位。

```
input CreateTaskInput {  
  todoId: Int!  
  description: String  
}
```

接著，建立幾個任務。任務與 有外部金鑰關係Todo，如下所示。

```
mutation TASKS {  
  a: createTask(input: {todoId: 2, description: "my first sub task"}) { id }  
  b:createTask(input: {todoId: 2, description: "another sub task"}) { id }  
  c: createTask(input: {todoId: 2, description: "a final sub task"}) { id }  
}
```

在名為 的Query類型中建立新的欄位getTodoAndTasks，如下所示。

```
getTodoAndTasks(id: Int!): Todo
```

將tasks欄位新增至 Todo類型，如下所示。

```
type Todo {  
  due: AWSDate!  
  id: Int!
```

```

    createdAt: String
    description: String!
    tasks: TaskConnection
  }

```

儲存結構描述。從主控台的結構描述編輯器右側，選擇連接的解析程式 `getTodosAndTasks(id: Int!): Todo`。選擇您的 Amazon RDS 資料來源。使用下列程式碼更新您的解析程式。

```

import { sql, createPgStatement, toJsonObject } from '@aws-appsync/utils/rds';

export function request(ctx) {
  return createPgStatement(
    sql`SELECT * from todos where id = ${ctx.args.id}`,
    sql`SELECT * from tasks where "todoId" = ${ctx.args.id}`);
}

export function response(ctx) {
  const result = toJsonObject(ctx.result);
  const todo = result[0][0];
  if (!todo) {
    return null;
  }
  todo.tasks = { items: result[1] };
  return todo;
}

```

在此程式碼中，我們使用 `sql` 標籤範本來撰寫 SQL 陳述式，以便在執行時間安全地將動態值傳遞給。一次最多 `createPgStatement` 可能需要兩個 SQL 請求。我們使用它為 傳送一個查詢 `todo`，為 傳送另一個查詢 `tasks`。您可以針對該事項，使用 `JOIN` 陳述式或任何其他方法完成此操作。該想法能夠撰寫您自己的 SQL 陳述式來實作您的商業邏輯。若要在查詢編輯器中使用查詢，請執行下列動作。

```

query TodoAndTasks {
  getTodosAndTasks(id: 2) {
    id
    due
    description
    tasks {
      items {
        id
        description
      }
    }
  }
}

```



```
}  
}
```

刪除叢集

Important

刪除叢集是永久的。在執行此動作之前，請徹底檢閱您的專案。

若要刪除叢集：

```
$ aws rds delete-db-cluster \  
  --db-cluster-identifier appsync-tutorial \  
  --skip-final-snapshot
```

的 VTL 解析程式教學課程 AWS AppSync

Note

我們現在主要支援 APPSYNC_JS 執行期及其文件。請考慮[在此處](#)使用 APPSYNC_JS 執行期及其指南。

AWS AppSync 會使用資料來源和解析程式來翻譯 GraphQL 請求，並從 AWS 資源擷取資訊。AWS AppSync 支援使用特定資料來源類型的自動佈建和連線。AWS AppSync 也支援 Amazon DynamoDB、AWS Lambda、關聯式資料庫 (Amazon Aurora Serverless)、Amazon OpenSearch Service 和 HTTP 端點做為資料來源。您可以搭配現有的 AWS 資源使用 GraphQL API，或從頭建置資料來源和解析程式。以下章節旨在以教學的形式闡明一些較常見的 GraphQL 使用案例。

AWS AppSync 針對解析程式使用以 Apache Velocity 範本語言 (VTL) 編寫的映射範本。如需使用映射範本的詳細資訊，請參閱[解析程式映射範本參考](#)。如需使用 VTL 的詳細資訊，請參閱[解析程式映射範本程式設計指南](#)。

AWS AppSync 支援從 GraphQL 結構描述自動佈建 DynamoDB 資料表，如從結構描述佈建（選用）和啟動範例結構描述中所述。您也可以從現有將建立結構描述和連接解析程式的 DynamoDB 資料表匯入。這在從 Amazon DynamoDB 匯入中概述（選用）。

主題

- [使用 DynamoDB 解析程式建立簡單的貼文應用程式](#)
- [在 中使用 AWS Lambda 解析程式 AWS AppSync](#)
- [使用 Amazon OpenSearch Service 解析程式 in AWS AppSync](#)
- [使用本機解析程式 in AWS AppSync](#)
- [在 中結合 GraphQL 解析程式 AWS AppSync](#)
- [在 AWS AppSync 中使用 DynamoDB 批次操作](#)
- [在 AWS AppSync 中執行 DynamoDB 交易](#)
- [使用 HTTP 解析程式 in AWS AppSync](#)
- [搭配 使用 Aurora Serverless v2 AWS AppSync](#)
- [在 AWS AppSync 中使用管道解析程式](#)
- [在 中的版本控制資料來源上使用 Delta Sync 操作 AWS AppSync](#)

使用 DynamoDB 解析程式建立簡單的貼文應用程式

Note

我們現在主要支援 APPSYNC_JS 執行期及其文件。請考慮[在此處](#)使用 APPSYNC_JS 執行期及其指南。

本教學課程說明如何將自己的 Amazon DynamoDB 資料表帶到 AWS AppSync，並將其連接到 GraphQL API。

您可以讓 AWS AppSync 代表您佈建 DynamoDB 資源。或者，如果您願意，您可以透過建立資料來源和解析程式來將現有的資料表連接到 GraphQL 結構描述。在這兩種情況下，您將能夠透過 GraphQL 陳述式讀取和寫入 DynamoDB 資料庫，以及訂閱即時資料。

有您必須完成的特定的組態步驟，才能讓 GraphQL 陳述式轉譯到 DynamoDB 操作，以及讓回應轉譯回 GraphQL。本教學課程概述透過幾個真實世界案例和資料存取模式的組態程序。

設定 DynamoDB 資料表

若要開始本教學課程，您必須先遵循下列步驟來佈建 AWS 資源。

1. 在 CLI 中使用下列 AWS CloudFormation 範本佈建 AWS 資源：

```
aws cloudformation create-stack \  
  --stack-name AWSAppSyncTutorialForAmazonDynamoDB \  
  --template-url https://s3.us-west-2.amazonaws.com/awsappsync/resources/  
dynamodb/AmazonDynamoDBCFTemplate.yaml \  
  --capabilities CAPABILITY_NAMED_IAM
```

或者，您可以在帳戶中 AWS 的美國西部 2（奧勒岡）區域啟動下列 AWS CloudFormation 堆疊。

Launch Stack 

這將會建立下列項目：

- 稱為 AppSyncTutorial-Post 的 DynamoDB 資料表，可存放 Post 資料。
- IAM 角色和關聯的 IAM 受管政策，以允許 AWS AppSync 與 Post 資料表互動。

2. 要查看有關堆疊和建立的資源的詳細資訊，請執行以下 CLI 命令：

```
aws cloudformation describe-stacks --stack-name AWSAppSyncTutorialForAmazonDynamoDB
```

3. 之後若要刪除資源，您可以執行以下項目：

```
aws cloudformation delete-stack --stack-name AWSAppSyncTutorialForAmazonDynamoDB
```

建立 GraphQL API

若要在 AWS AppSync 中建立 GraphQL API：

1. 登入 AWS Management Console 並開啟 [AppSync 主控台](#)。
 - 在 APIs 儀表板中，選擇建立 API。
2. 在自訂您的 API 或從 Amazon DynamoDB 匯入視窗下，選擇從頭建置。
 - 選擇相同視窗右側的開始。
3. 在 API 名稱欄位中，將 API 的名稱設定為 `AWSAppSyncTutorial`。
4. 選擇建立。

AWS AppSync 主控台會使用 API 金鑰身分驗證模式為您建立新的 GraphQL API。您可以使用主控台來設定其他 GraphQL API 和對其執行查詢，以進行此教學的其他部分。

定義基本文章 API

現在您已建立 an AWS AppSync GraphQL API，您可以設定基本結構描述，以允許基本建立、擷取和刪除文章資料。

1. 登入 AWS Management Console 並開啟 [AppSync 主控台](#)。
 - 在 APIs 儀表板中，選擇您剛建立的 API。
2. 在側邊欄中，選擇結構描述。
 - 在結構描述窗格中，將內容取代為下列程式碼：

```
schema {  
  query: Query
```

```
    mutation: Mutation
  }

  type Query {
    getPost(id: ID!): Post
  }

  type Mutation {
    addPost(
      id: ID!
      author: String!
      title: String!
      content: String!
      url: String!
    ): Post!
  }

  type Post {
    id: ID!
    author: String
    title: String
    content: String
    url: String
    ups: Int!
    downs: Int!
    version: Int!
  }
```

3. 選擇儲存。

這個結構描述會定義 Post 類型和操作以新增和取得 Post 物件。

設定 DynamoDB 資料表的資料來源

接著，將結構描述中定義的查詢和變動連結至 AppSyncTutorial-PostDynamoDB 資料表。

首先，AWS AppSync 需要注意您的資料表。您可以在 AWS AppSync 中設定資料來源來執行此操作：

1. 登入 AWS Management Console 並開啟 [AppSync 主控台](#)。
 - a. 在 APIs 儀表板中，選擇您的 GraphQL API。

- b. 在側邊欄中，選擇資料來源。
2. 選擇 Create data source (建立資料來源)。
 - a. 對於資料來源名稱，在 中輸入 PostDynamoDBTable。
 - b. 針對資料來源類型，選擇 Amazon DynamoDB 資料表。
 - c. 針對區域，選擇 US-WEST-2。
 - d. 針對資料表名稱，選擇 AppSyncTutorial-Post DynamoDB 資料表。
 - e. 建立新的 IAM 角色（建議）或選擇具有 IAM `lambda:invokeFunction` 許可的現有角色。現有角色需要信任政策，如[連接資料來源](#)一節所述。

以下是 IAM 政策範例，具有對資源執行操作所需的許可：

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [ "lambda:invokeFunction" ],
      "Resource": [
        "arn:aws:lambda:us-west-2:123456789012:function:myFunction",
        "arn:aws:lambda:us-west-2:123456789012:function:myFunction:*"
      ]
    }
  ]
}
```

3. 選擇建立。

設定 addPost 解析程式 (DynamoDB PutItem)

After AWS AppSync 知道 DynamoDB 資料表，您可以透過定義解析程式將其連結至個別查詢和變動。您建立的第一個解析程式是 addPost 解析程式，可讓您在 AppSyncTutorial-Post DynamoDB 資料表中建立文章。

解析程式具有下列元件：

- 在 GraphQL 結構描述中要附加解析程式的位置。在這種情況下，您會對 addPost 類型上的 Mutation 欄位設定解析程式。發起人呼叫 mutation { addPost(...){...} } 時會叫用此解析程式。
- 用於此解析程式的資料來源。在這種情況下，您想要使用先前定義的 PostDynamoDBTable 資料來源，如此您可以將項目新增到 AppSyncTutorial-Post DynamoDB 資料表。
- 請求映射範本。請求映射範本的目的是從發起人取得傳入請求，並將其轉換為指示 for AWS AppSync 以對 DynamoDB 執行。
- 回應映射範本。回應映射範本的任務即是從 DynamoDB 取得回應，並將其轉譯為 GraphQL 期望的項目。如果 DynamoDB 中的資料形狀與 GraphQL 中的 Post 類型不同，此功能會很有用，但如果他們的形狀一樣，您只需傳遞資料。

設定解析程式：

1. 登入 AWS Management Console 並開啟 [AppSync 主控台](#)。
 - a. 在 APIs 儀表板中，選擇您的 GraphQL API。
 - b. 在側邊欄中，選擇資料來源。
2. 選擇 Create data source (建立資料來源)。
 - a. 對於資料來源名稱，在 中輸入 PostDynamoDBTable。
 - b. 針對資料來源類型，選擇 Amazon DynamoDB 資料表。
 - c. 針對區域，選擇 US-WEST-2。
 - d. 針對資料表名稱，選擇 AppSyncTutorial-Post DynamoDB 資料表。
 - e. 建立新的 IAM 角色（建議）或選擇具有 IAM lambda:invokeFunction 許可的現有角色。現有角色需要信任政策，如[連接資料來源](#)一節所述。

以下是 IAM 政策範例，具有對資源執行操作所需的許可：

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [ "lambda:invokeFunction" ],
```

```

      "Resource": [
        "arn:aws:lambda:us-
west-2:123456789012:function:myFunction",
        "arn:aws:lambda:us-
west-2:123456789012:function:myFunction:*"
      ]
    }
  ]
}

```

3. 選擇建立。
4. 選擇 Schema (結構描述) 標籤。
5. 在右側的資料類型窗格中，尋找變動類型的 addPost 欄位，然後選擇連接。
6. 在動作功能表中，選擇更新執行時間，然後選擇單位解析程式 (僅限 VTL)。
7. 在 Data source name (資料來源名稱) 中，選擇 PostDynamoDBTable。
8. 將以下內容貼到 Configure the request mapping template (設定請求映射範本) 區段：

```

{
  "version" : "2017-02-28",
  "operation" : "PutItem",
  "key" : {
    "id" : $util.dynamodb.toDynamoDBJson($context.arguments.id)
  },
  "attributeValues" : {
    "author" : $util.dynamodb.toDynamoDBJson($context.arguments.author),
    "title" : $util.dynamodb.toDynamoDBJson($context.arguments.title),
    "content" : $util.dynamodb.toDynamoDBJson($context.arguments.content),
    "url" : $util.dynamodb.toDynamoDBJson($context.arguments.url),
    "ups" : { "N" : 1 },
    "downs" : { "N" : 0 },
    "version" : { "N" : 1 }
  }
}

```

注意：所有金鑰和屬性值皆有指定的類型。例如，您將 author 欄位設為 { "S" : "\${context.arguments.author}" }。S 部分指出 to AWS AppSync 和 DynamoDB 的值將是字串值。會將實際值填入 author 引數。同樣地，version 欄位是號碼欄位，因為它會將 N 用於類型。最後，您也可以初始化 ups、downs 和 version 欄位。

在本教學課程中，您已指定 GraphQL ID! 類型，其會將插入 DynamoDB 的新項目編製索引，做為 client arguments. AWS AppSync comes 的一部分，其中包含名為 的自動 ID 產生公用程式 `$utils.autoId()`，您也可以以 的形式使用 `"id" : { "S" : "${$utils.autoId()}" }`。然後，您可以直接將 `id: ID!` 移出 `addPost()` 的結構描述定義，該 ID 便會自動插入。您不會在本教學課程中使用此技術，但您應該將其視為寫入 DynamoDB 資料表時的最佳實務。

如需映射範本的詳細資訊，請參閱[解析程式映射範本概觀](#)參考文件。如需 `GetItem` 要求映射的詳細資訊，請參閱 [GetItem](#) 參考文件。如需有關類型的詳細資訊，請參閱[類型系統 \(要求映射\)](#) 參考文件。

9. 將以下內容貼到 Configure the response mapping template (設定回應映射範本) 區段：

```
$utils.toJson($context.result)
```

注意：由於 `AppSyncTutorial-Post` 資料表中的資料形狀正好與 GraphQL 中的 `Post` 類型形狀相符，回應映射範本只需直接傳遞結果。另請注意，此教學課程中的所有範例會使用相同的回應映射範本，因此您只需建立一個檔案。

10. 選擇儲存。

呼叫 API 以新增文章

現在已設定解析程式，AWS AppSync 可以將傳入 `addPost` 變動轉譯為 DynamoDB `PutItem` 操作。您現在可以執行變動以將項目放置到資料表中。

- 選擇 Queries (查詢) 標籤。
- 將以下變動貼到 Queries (查詢) 窗格：

```
mutation addPost {  
  addPost(  
    id: 123  
    author: "AUTHORNAME"  
    title: "Our first post!"  
    content: "This is our first post."  
    url: "https://aws.amazon.com/appsync/"  
  ) {  
    id  
    author  
    title  
  }  
}
```

```

    content
    url
    ups
    downs
    version
  }
}
```

- 選擇 Execute query (執行查詢) (橘色播放按鈕)。
- 新建立文章的結果應該會出現在查詢窗格右側的結果窗格中。其看起來與下列類似：

```

{
  "data": {
    "addPost": {
      "id": "123",
      "author": "AUTHORNAME",
      "title": "Our first post!",
      "content": "This is our first post.",
      "url": "https://aws.amazon.com/appsync/",
      "ups": 1,
      "downs": 0,
      "version": 1
    }
  }
}
```

以下是發生的情況：

- AWS AppSync 收到addPost變動請求。
- AWS AppSync 接受了請求和請求映射範本，並產生了請求映射文件。此看起來如下：

```

{
  "version" : "2017-02-28",
  "operation" : "PutItem",
  "key" : {
    "id" : { "S" : "123" }
  },
  "attributeValues" : {
    "author": { "S" : "AUTHORNAME" },
    "title": { "S" : "Our first post!" },
    "content": { "S" : "This is our first post." },
  }
}
```

```

    "url": { "S" : "https://aws.amazon.com/appsync/" },
    "ups" : { "N" : 1 },
    "downs" : { "N" : 0 },
    "version" : { "N" : 1 }
  }
}

```

- AWS AppSync 使用請求映射文件來產生和執行 DynamoDBPutItem 請求。
- AWS AppSync 取得PutItem請求的結果，並將其轉換回 GraphQL 類型。

```

{
  "id" : "123",
  "author": "AUTHORNAME",
  "title": "Our first post!",
  "content": "This is our first post.",
  "url": "https://aws.amazon.com/appsync/",
  "ups" : 1,
  "downs" : 0,
  "version" : 1
}

```

- 透過回應映射文件來進行傳遞，這可讓其以原狀進行傳遞。
- 傳回在 GraphQL 回應中新建立的物件。

設定 getPost 解析程式 (DynamoDB GetItem)

現在您可以將資料新增至 AppSyncTutorial-PostDynamoDB 資料表，您需要設定getPost查詢，以便從AppSyncTutorial-Post資料表擷取該資料。若要執行此作業，您可以設定另一個解析程式。

- 選擇 Schema (結構描述) 標籤。
- 在右側的資料類型窗格中，尋找查詢類型的 getPost 欄位，然後選擇連接。
- 在動作功能表中，選擇更新執行時間，然後選擇單位解析程式（僅限 VTL）。
- 在 Data source name (資料來源名稱) 中，選擇 PostDynamoDBTable。
- 將以下內容貼到 Configure the request mapping template (設定請求映射範本) 區段：

```

{
  "version" : "2017-02-28",
  "operation" : "GetItem",
  "key" : {
    "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
  }
}

```

```
}  
}
```

- 將以下內容貼到 Configure the response mapping template (設定回應映射範本) 區段：

```
$utils.toJson($context.result)
```

- 選擇儲存。

呼叫 API 以取得文章

現在已設定解析程式，AWS AppSync 知道如何將傳入getPost查詢轉譯為 DynamoDBGetItem 操作。您現在可以執行查詢，擷取您稍早建立的貼文。

- 選擇 Queries (查詢) 標籤。
- 在 Queries (查詢) 窗格中，貼入下面內容：

```
query getPost {  
  getPost(id:123) {  
    id  
    author  
    title  
    content  
    url  
    ups  
    downs  
    version  
  }  
}
```

- 選擇 Execute query (執行查詢) (橘色播放按鈕)。
- 從 DynamoDB 擷取的文章應該會出現在查詢窗格右側的結果窗格中。其看起來與下列類似：

```
{  
  "data": {  
    "getPost": {  
      "id": "123",  
      "author": "AUTHORNAME",  
      "title": "Our first post!",  
      "content": "This is our first post.",  
      "url": "https://aws.amazon.com/appsync/",  
    }  
  }  
}
```

```
    "ups": 1,
    "downs": 0,
    "version": 1
  }
}
```

以下是發生的情況：

- AWS AppSync 收到getPost查詢請求。
- AWS AppSync 接受了請求和請求映射範本，並產生了請求映射文件。此看起來如下：

```
{
  "version" : "2017-02-28",
  "operation" : "GetItem",
  "key" : {
    "id" : { "S" : "123" }
  }
}
```

- AWS AppSync 使用請求映射文件來產生和執行 DynamoDB GetItem 請求。
- AWS AppSync 取得GetItem請求的結果，並將其轉換回 GraphQL 類型。

```
{
  "id" : "123",
  "author": "AUTHORNAME",
  "title": "Our first post!",
  "content": "This is our first post.",
  "url": "https://aws.amazon.com/appsync/",
  "ups" : 1,
  "downs" : 0,
  "version" : 1
}
```

- 透過回應映射文件來進行傳遞，這可讓其以原狀進行傳遞。
- 在回應中傳回擷取的物件。

或者，請採用下列範例：

```
query getPost {
```

```
getPost(id:123) {  
  id  
  author  
  title  
}
```

如果您的 `getPost` 查詢只需要 `id`、`author` 和 `title`，您可以將請求映射範本變更為使用投影表達式，以僅指定您想要從 DynamoDB 資料表傳輸的屬性，以避免從 DynamoDB 傳輸不必要的資料 AWS AppSync。例如，請求映射範本可能看起來像下面的程式碼片段：

```
{  
  "version" : "2017-02-28",  
  "operation" : "GetItem",  
  "key" : {  
    "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)  
  },  
  "projection" : {  
    "expression" : "#author, id, title",  
    "expressionNames" : { "#author" : "author"}  
  }  
}
```

建立 updatePost Mutation (DynamoDB UpdateItem)

到目前為止，您可以在 DynamoDB 中建立和擷取 Post 物件。接下來，您會設定新的變動，讓我們能夠更新物件。您可以使用 UpdateItem DynamoDB 操作來執行此作業。

- 選擇 Schema (結構描述) 標籤。
- 修改 Schema (結構描述) 窗格中的 Mutation 類型來新增 updatePost 變動，如下所示：

```
type Mutation {  
  updatePost(  
    id: ID!,  
    author: String!,  
    title: String!,  
    content: String!,  
    url: String!  
  ): Post  
  addPost(  
    author: String!
```

```

        title: String!
        content: String!
        url: String!
    ): Post!
}

```

- 選擇儲存。
- 在右側的資料類型窗格中，尋找變動類型上新建立的 updatePost 欄位，然後選擇連接。
- 在動作功能表中，選擇更新執行時間，然後選擇單位解析程式（僅限 VTL）。
- 在 Data source name (資料來源名稱) 中，選擇 PostDynamoDBTable。
- 將以下內容貼到 Configure the request mapping template (設定請求映射範本) 區段：

```

{
  "version" : "2017-02-28",
  "operation" : "UpdateItem",
  "key" : {
    "id" : $util.dynamodb.toDynamoDBJson($context.arguments.id)
  },
  "update" : {
    "expression" : "SET author = :author, title = :title, content = :content,
#url = :url ADD version :one",
    "expressionNames": {
      "#url" : "url"
    },
    "expressionValues": {
      ":author" : $util.dynamodb.toDynamoDBJson($context.arguments.author),
      ":title" : $util.dynamodb.toDynamoDBJson($context.arguments.title),
      ":content" : $util.dynamodb.toDynamoDBJson($context.arguments.content),
      ":url" : $util.dynamodb.toDynamoDBJson($context.arguments.url),
      ":one" : { "N": 1 }
    }
  }
}

```

注意：此解析程式正在使用 DynamoDB UpdateItem，這與 PutItem 操作明顯不同。您只是要求 DynamoDB 更新特定屬性，而不是撰寫整個項目。這是使用 DynamoDB Update Expressions 完成。表達式本身會在 expression 區段中的 update 欄位中指定。其要求設定 author、title、content 以及 url 屬性，然後遞增 version 欄位。要使用的值不會出現在表達式本身；表達式的預留位置名稱開頭為冒號，其會在 expressionValues 欄位中加以定義。最

後，DynamoDB 具有無法出現在 中的預留單字expression。例如，由於 url 是保留字，因此若要更新 url 欄位，您可以使用名稱預留位置，並在 expressionNames 欄位中定義他們。

如需 UpdateItem 請求映射的詳細資訊，請參閱 [UpdateItem](#) 參考文件。如需有關如何撰寫更新表達式的詳細資訊，請參閱 [DynamoDB UpdateExpressions](#) 文件。

- 將以下內容貼到 Configure the response mapping template (設定回應映射範本) 區段：

```
$utils.toJson($context.result)
```

呼叫 API 以更新文章

現在已設定解析程式，AWS AppSync 知道如何將傳入update變動轉換為 DynamoDBUpdate 操作。您現在可以執行變動，以更新您之前撰寫的項目。

- 選擇 Queries (查詢) 標籤。
- 將以下變動貼到 Queries (查詢) 窗格。您也必須將 id 引數更新為您先前記下的值。

```
mutation updatePost {  
  updatePost(  
    id:"123"  
    author: "A new author"  
    title: "An updated author!"  
    content: "Now with updated content!"  
    url: "https://aws.amazon.com/appsync/"  
  ) {  
    id  
    author  
    title  
    content  
    url  
    ups  
    downs  
    version  
  }  
}
```

- 選擇 Execute query (執行查詢) (橘色播放按鈕)。
- DynamoDB 中更新的文章應該會出現在查詢窗格右側的結果窗格中。其看起來與下列類似：

```
{
```



```
"data": {
  "updatePost": {
    "id": "123",
    "author": "A new author",
    "title": "An updated author!",
    "content": "Now with updated content!",
    "url": "https://aws.amazon.com/appsync/",
    "ups": 1,
    "downs": 0,
    "version": 2
  }
}
```

在此範例中，ups和downs欄位並未修改，因為請求映射範本並未ask AWS AppSync 和 DynamoDB 對這些欄位執行任何動作。此外，version欄位以1遞增，因為您要求 AWS AppSync 和 DynamoDB 將1新增至version欄位。

修改 updatePost 解析程式 (DynamoDB UpdateItem)

這對於 updatePost 變動而言是一個好的開始，但有兩個主要的問題：

- 如果您只想更新單一欄位，您必須更新所有欄位。
- 如果兩個人同時修改物件，您便可能會失去資訊。

為了解決這些問題，您將會修改 updatePost 變動，使其僅修改請求中指定的引數，然後將條件新增到 UpdateItem 操作。

1. 選擇 Schema (結構描述) 標籤。
2. 在 Schema (結構描述) 窗格中，修改 Mutation 類型中的 updatePost 欄位，以便從 author、title、content 及 url 引數中移除驚嘆號，並請確保 id 欄位維持原狀。這會使它們變成可選引數。此外，新增必要 expectedVersion 引數。

```
type Mutation {
  updatePost(
    id: ID!,
    author: String,
    title: String,
    content: String,
```

```

        url: String,
        expectedVersion: Int!
    ): Post
    addPost(
        author: String!
        title: String!
        content: String!
        url: String!
    ): Post!
}

```

3. 選擇儲存。
4. 在右側的資料類型窗格中，尋找變動類型的 `updatePost` 欄位。
5. 選擇 `PostDynamoDBTable` 以開啟現有的解析程式。
6. 在 `Configure the request mapping template` (設定要求映射範本) 中修改要求映射範本，如下所示：

```

{
  "version" : "2017-02-28",
  "operation" : "UpdateItem",
  "key" : {
    "id" : $util.dynamodb.toDynamoDBJson($context.arguments.id)
  },

  ## Set up some space to keep track of things you're updating **
  #set( $expNames = {} )
  #set( $expValues = {} )
  #set( $expSet = {} )
  #set( $expAdd = {} )
  #set( $expRemove = [] )

  ## Increment "version" by 1 **
  ${expAdd.put("version", ":one")}
  ${expValues.put(":one", { "N" : 1 })}

  ## Iterate through each argument, skipping "id" and "expectedVersion" **
  #foreach( $entry in $context.arguments.entrySet() )
    #if( $entry.key != "id" && $entry.key != "expectedVersion" )
      #if( (!$entry.value) && ("${entry.value}" == "") )
        ## If the argument is set to "null", then remove that attribute from
        the item in DynamoDB **

        #set( $discard = ${expRemove.add("#${entry.key}")} )

```

```

        ${expNames.put("#${entry.key}", "${entry.key}")}
    #else
        ## Otherwise set (or update) the attribute on the item in DynamoDB **

        ${expSet.put("#${entry.key}", ":${entry.key}")}
        ${expNames.put("#${entry.key}", "${entry.key}")}
        ${expValues.put(":${entry.key}", { "S" : "${entry.value}" })}
    #end
#end
#end

## Start building the update expression, starting with attributes you're going to
SET **
#set( $expression = "" )
#if( !${expSet.isEmpty()} )
    #set( $expression = "SET" )
    #foreach( $entry in $expSet.entrySet() )
        #set( $expression = "${expression} ${entry.key} = ${entry.value}" )
        #if ( $foreach.hasNext )
            #set( $expression = "${expression}," )
        #end
    #end
#end
#end

## Continue building the update expression, adding attributes you're going to ADD
**
#if( !${expAdd.isEmpty()} )
    #set( $expression = "${expression} ADD" )
    #foreach( $entry in $expAdd.entrySet() )
        #set( $expression = "${expression} ${entry.key} ${entry.value}" )
        #if ( $foreach.hasNext )
            #set( $expression = "${expression}," )
        #end
    #end
#end
#end

## Continue building the update expression, adding attributes you're going to
REMOVE **
#if( !${expRemove.isEmpty()} )
    #set( $expression = "${expression} REMOVE" )

    #foreach( $entry in $expRemove )
        #set( $expression = "${expression} ${entry}" )
        #if ( $foreach.hasNext )

```

```

        #set( $expression = "${expression}," )
    #end
#end
#end

## Finally, write the update expression into the document, along with any
expressionNames and expressionValues **
"update" : {
    "expression" : "${expression}"
    #if( !${expNames.isEmpty()} )
        , "expressionNames" : $utils.toJson($expNames)
    #end
    #if( !${expValues.isEmpty()} )
        , "expressionValues" : $utils.toJson($expValues)
    #end
},

"condition" : {
    "expression" : "version = :expectedVersion",
    "expressionValues" : {
        ":expectedVersion" :
$util.dynamodb.toDynamoDBJson($context.arguments.expectedVersion)
    }
}
}

```

7. 選擇儲存。

此範本是更為複雜的範例。它示範映射範本的強大功能和靈活性。它會循環使用所有引數，但會略過 `id` 和 `expectedVersion`。如果引數設為某個項目，則其 ask AWS AppSync 和 DynamoDB 會更新 DynamoDB 中物件上的該屬性。如果屬性設定為 `null`，則其 asked AWS AppSync 和 DynamoDB 會從後置物件中移除該屬性。若未指定引數，它將會保留該屬性。它會增量 `version` 欄位。

另外，還有新的 `condition` 區段。條件表達式可讓您 tell AWS AppSync 和 DynamoDB，根據在執行操作之前已在 DynamoDB 中的物件狀態，請求是否應成功。在此情況下，只有在 DynamoDB 中目前項目 `version` 的欄位完全符合 `expectedVersion` 引數時，您才希望 `UpdateItem` 請求成功。

如需條件表達式的詳細資訊，請參閱[條件表達式](#)參考文件。

呼叫 API 以更新文章

讓我們嘗試使用新解析程式來更新 Post 物件：

- 選擇 Queries (查詢) 標籤。
- 將以下變動貼到 Queries (查詢) 窗格。您也必須將 id 引數更新為您先前記下的值。

```
mutation updatePost {  
  updatePost(  
    id:123  
    title: "An empty story"  
    content: null  
    expectedVersion: 2  
  ) {  
    id  
    author  
    title  
    content  
    url  
    ups  
    downs  
    version  
  }  
}
```

- 選擇 Execute query (執行查詢) (橘色播放按鈕)。
- DynamoDB 中更新的文章應該會出現在查詢窗格右側的結果窗格中。其看起來與下列類似：

```
{  
  "data": {  
    "updatePost": {  
      "id": "123",  
      "author": "A new author",  
      "title": "An empty story",  
      "content": null,  
      "url": "https://aws.amazon.com/appsync/",  
      "ups": 1,  
      "downs": 0,  
      "version": 3  
    }  
  }  
}
```

在此請求中，您要求 AWS AppSync 和 DynamoDB 僅更新 title 和 content 欄位。它會將所有其他欄位單獨保留 (除了遞增 version 欄位)。您將 title 屬性設定為新的值，並從文章中移除 content 屬性。author、url、ups 和 downs 欄位維持原狀。

嘗試再次執行變動要求，讓要求維持原狀。您應該會看到類似以下的回應：

```
{
  "data": {
    "updatePost": null
  },
  "errors": [
    {
      "path": [
        "updatePost"
      ],
      "data": {
        "id": "123",
        "author": "A new author",
        "title": "An empty story",
        "content": null,
        "url": "https://aws.amazon.com/appsync/",
        "ups": 1,
        "downs": 0,
        "version": 3
      },
      "errorType": "DynamoDB:ConditionalCheckFailedException",
      "locations": [
        {
          "line": 2,
          "column": 3
        }
      ],
      "message": "The conditional request failed (Service: AmazonDynamoDBv2; Status Code: 400; Error Code: ConditionalCheckFailedException; Request ID: ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZ)"
    }
  ]
}
```

條件表達式評估為「false」，因此要求失敗：

- 第一次執行請求時，DynamoDB 中文文章的 version 欄位值為 2，符合 expectedVersion 引數。請求成功，這表示 version 欄位已在 DynamoDB 中遞增至 3。

- 第二次執行請求時，DynamoDB 中文章的 `version` 欄位值為 3，這不符合 `expectedVersion` 數。

此模式通常稱為樂觀鎖定。

an AWS AppSync DynamoDB 解析程式的一項功能是傳回 DynamoDB 中貼文物件的目前值。您可以在 GraphQL 回應的 `data` 區段 `errors` 欄位中找到此值。您的應用程式可以使用這些資訊來決定繼續執行的方式。在這種情況下，您可以看到 DynamoDB 中物件 `version` 的欄位設定為 3，因此您只要將 `expectedVersion` 引數更新為 3，請求就會再次成功。

如需有關處理條件檢查錯誤的詳細資訊，請參閱[條件表達式](#)映射範本參考文件。

建立 upvotePost 和 downvotePost 變動 (DynamoDB UpdateItem)

Post 類型具有 `ups` 和 `downs` 欄位，可記錄贊同數和不贊同數，但是到目前為止，API 不會讓我們使用它們來執行任何動作。讓我們新增一些變動來對文章表示贊同與不贊同。

- 選擇 Schema (結構描述) 標籤。
- 修改 Schema (結構描述) 窗格中的 Mutation 類型以新增 `upvotePost` 和 `downvotePost` 變動，如下所示：

```
type Mutation {
  upvotePost(id: ID!): Post
  downvotePost(id: ID!): Post
  updatePost(
    id: ID!,
    author: String,
    title: String,
    content: String,
    url: String,
    expectedVersion: Int!
  ): Post
  addPost(
    author: String!,
    title: String!,
    content: String!,
    url: String!
  ): Post!
}
```

- 選擇儲存。

- 在右側的資料類型窗格中，尋找變動類型上新建立的 upvotePost 欄位，然後選擇連接。
- 在動作功能表中，選擇更新執行時間，然後選擇單位解析程式（僅限 VTL）。
- 在 Data source name (資料來源名稱) 中，選擇 PostDynamoDBTable。
- 將以下內容貼到 Configure the request mapping template (設定請求映射範本) 區段：

```
{
  "version" : "2017-02-28",
  "operation" : "UpdateItem",
  "key" : {
    "id" : $util.dynamodb.toDynamoDBJson($context.arguments.id)
  },
  "update" : {
    "expression" : "ADD ups :plusOne, version :plusOne",
    "expressionValues" : {
      ":plusOne" : { "N" : 1 }
    }
  }
}
```

- 將以下內容貼到 Configure the response mapping template (設定回應映射範本) 區段：

```
$utils.toJson($context.result)
```

- 選擇儲存。
- 在右側的資料類型窗格中，尋找變動類型上新建立downvotePost的欄位，然後選擇連接。
- 在 Data source name (資料來源名稱) 中，選擇 PostDynamoDBTable。
- 將以下內容貼到 Configure the request mapping template (設定請求映射範本) 區段：

```
{
  "version" : "2017-02-28",
  "operation" : "UpdateItem",
  "key" : {
    "id" : $util.dynamodb.toDynamoDBJson($context.arguments.id)
  },
  "update" : {
    "expression" : "ADD downs :plusOne, version :plusOne",
    "expressionValues" : {
      ":plusOne" : { "N" : 1 }
    }
  }
}
```



```
}
```

- 將以下內容貼到 Configure the response mapping template (設定回應映射範本) 區段：

```
$utils.toJson($context.result)
```

- 選擇儲存。

呼叫 API 來對文章表達贊同或不贊同

現在已設定新的解析程式，AWS AppSync 知道如何將傳入 `upvotePost` 或 `downvote` 變動轉換為 DynamoDB `UpdateItem` 操作。您現在可以執行變動，對您之前建立的文章表達贊同或不贊同。

- 選擇 Queries (查詢) 標籤。
- 將以下變動貼到 Queries (查詢) 窗格。您也必須將 `id` 引數更新為您先前記下的值。

```
mutation votePost {  
  upvotePost(id:123) {  
    id  
    author  
    title  
    content  
    url  
    ups  
    downs  
    version  
  }  
}
```

- 選擇 Execute query (執行查詢) (橘色播放按鈕)。
- 文章會在 DynamoDB 中更新，並應出現在查詢窗格右側的結果窗格中。其看起來與下列類似：

```
{  
  "data": {  
    "upvotePost": {  
      "id": "123",  
      "author": "A new author",  
      "title": "An empty story",  
      "content": null,  
      "url": "https://aws.amazon.com/appsync/",  
      "ups": 6,  
    }  
  }  
}
```

```
      "downs": 0,  
      "version": 4  
    }  
  }  
}
```

- 再選擇幾次 Execute query (執行查詢)。您應查看在每次執行查詢時以 1 為單位而遞增的 ups 和 version 欄位。
- 變更查詢以呼叫 downvotePost 變動，如下所示：

```
mutation votePost {  
  downvotePost(id:123) {  
    id  
    author  
    title  
    content  
    url  
    ups  
    downs  
    version  
  }  
}
```

- 選擇 Execute query (執行查詢) (橘色播放按鈕)。此時，您應查看在每次執行查詢時以 1 為單位而遞增的 downs 和 version 欄位。

```
{  
  "data": {  
    "downvotePost": {  
      "id": "123",  
      "author": "A new author",  
      "title": "An empty story",  
      "content": null,  
      "url": "https://aws.amazon.com/appsync/",  
      "ups": 6,  
      "downs": 4,  
      "version": 12  
    }  
  }  
}
```

設定 deletePost Resolver (DynamoDB DeleteItem)

您要設定的下一個變動是刪除貼文。您將使用 DeleteItem DynamoDB 操作來執行此操作。

- 選擇 Schema (結構描述) 標籤。
- 修改 Schema (結構描述) 窗格中的 Mutation 類型來新增 deletePost 變動，如下所示：

```
type Mutation {  
  deletePost(id: ID!, expectedVersion: Int!): Post  
  upvotePost(id: ID!): Post  
  downvotePost(id: ID!): Post  
  updatePost(  
    id: ID!,  
    author: String,  
    title: String,  
    content: String,  
    url: String,  
    expectedVersion: Int!  
  ): Post  
  addPost(  
    author: String!,  
    title: String!,  
    content: String!,  
    url: String!  
  ): Post!  
}
```

這次您將 expectedVersion 欄位設為選用，稍後在您新增要求映射範本時，將對此進行說明。

- 選擇儲存。
- 在右側的資料類型窗格中，尋找變動類型上新建立的刪除欄位，然後選擇連接。
- 在動作功能表中，選擇更新執行時間，然後選擇單位解析程式（僅限 VTL）。
- 在 Data source name (資料來源名稱) 中，選擇 PostDynamoDBTable。
- 將以下內容貼到 Configure the request mapping template (設定請求映射範本) 區段：

```
{  
  "version" : "2017-02-28",  
  "operation" : "DeleteItem",  
  "key": {  
    "id": $util.dynamodb.toDynamoDBJson($context.arguments.id)  
  }  
}
```

```

    #if( $context.arguments.containsKey("expectedVersion") )
      , "condition" : {
        "expression"      : "attribute_not_exists(id) OR version
= :expectedVersion",
        "expressionValues" : {
          ":expectedVersion" :
$util.dynamodb.toDynamoDBJson($context.arguments.expectedVersion)
        }
      }
    #end
  }
}

```

注意：expectedVersion 引數為可選引數。如果發起人在請求中設定expectedVersion引數，則範本會新增條件，只有在項目已刪除或 DynamoDB 中的文章version屬性完全符合時，才會允許DeleteItem請求成功expectedVersion。如果省略，將不會在 DeleteItem 要求中指定條件表達式。無論 的值為何version，或 DynamoDB 中是否存在該項目，它都會成功。

- 將以下內容貼到 Configure the response mapping template (設定回應映射範本) 區段：

```
$utils.toJson($context.result)
```

注意：即使您正在刪除某個項目，如果該項目尚未刪除，您仍然可以傳回已刪除的項目。

- 選擇儲存。

如需有關 DeleteItem 要求映射的詳細資訊，請參閱 [DeleteItem](#) 參考文件。

呼叫 API 以刪除文章

現在已設定解析程式，AWS AppSync 知道如何將傳入delete變動轉換為 DynamoDBDeleteItem 操作。您現在可以執行變動以在資料表中刪除項目。

- 選擇 Queries (查詢) 標籤。
- 將以下變動貼到 Queries (查詢) 窗格。您也必須將 id 引數更新為您先前記下的值。

```

mutation deletePost {
  deletePost(id:123) {
    id
    author
    title
    content
  }
}

```

```
    url
    ups
    downs
    version
  }
}
```

- 選擇 Execute query (執行查詢) (橘色播放按鈕)。
- 文章會從 DynamoDB 中刪除。請注意，AWS AppSync 會傳回從 DynamoDB 刪除的項目值，該值應該會出現在查詢窗格右側的結果窗格中。其看起來與下列類似：

```
{
  "data": {
    "deletePost": {
      "id": "123",
      "author": "A new author",
      "title": "An empty story",
      "content": null,
      "url": "https://aws.amazon.com/appsync/",
      "ups": 6,
      "downs": 4,
      "version": 12
    }
  }
}
```

如果對 deletePost 的呼叫實際上是從 DynamoDB 遭到刪除的呼叫，則只會傳回值。

- 再次選擇 Execute query (執行查詢)。
- 此呼叫仍會成功，但不會傳回任何值。

```
{
  "data": {
    "deletePost": null
  }
}
```

現在讓我們嘗試刪除文章，但這次是指定 expectedValue。首先，您需要建立新文章，因為您刪除的正好是您到目前為止所用的文章。

- 將以下變動貼到 Queries (查詢) 窗格：

```
mutation addPost {  
  addPost(  
    id:123  
    author: "AUTHORNAME"  
    title: "Our second post!"  
    content: "A new post."  
    url: "https://aws.amazon.com/appsync/"  
  ) {  
    id  
    author  
    title  
    content  
    url  
    ups  
    downs  
    version  
  }  
}
```

- 選擇 Execute query (執行查詢) (橘色播放按鈕)。
- 新建立文章的結果應該會出現在查詢窗格右側的結果窗格中。請記下新建立的物件的 id，因為您很快就會用到它。其看起來與下列類似：

```
{  
  "data": {  
    "addPost": {  
      "id": "123",  
      "author": "AUTHORNAME",  
      "title": "Our second post!",  
      "content": "A new post.",  
      "url": "https://aws.amazon.com/appsync/",  
      "ups": 1,  
      "downs": 0,  
      "version": 1  
    }  
  }  
}
```

現在讓我們嘗試刪除該文章，但我們將錯誤的值放入 expectedVersion：

- 將以下變動貼到 Queries (查詢) 窗格。您也必須將 `id` 引數更新為您先前記下的值。

```
mutation deletePost {
  deletePost(
    id:123
    expectedVersion: 9999
  ) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

- 選擇 Execute query (執行查詢) (橘色播放按鈕)。

```
{
  "data": {
    "deletePost": null
  },
  "errors": [
    {
      "path": [
        "deletePost"
      ],
      "data": {
        "id": "123",
        "author": "AUTHORNAME",
        "title": "Our second post!",
        "content": "A new post.",
        "url": "https://aws.amazon.com/appsync/",
        "ups": 1,
        "downs": 0,
        "version": 1
      },
      "errorType": "DynamoDB:ConditionalCheckFailedException",
      "locations": [
        {
          "line": 2,
```

```

      "column": 3
    }
  ],
  "message": "The conditional request failed (Service: AmazonDynamoDBv2;
Status Code: 400; Error Code: ConditionalCheckFailedException; Request ID:
ABCDEFGHIJKLMN0PQRSTUVWXYZABCDEFGHIJKLMN0PQRSTUVWXYZ)"
}
]
}

```

請求失敗，因為條件表達式評估為 false：DynamoDB 中文章 version 的值不符合引數中 expectedValue 指定的。會在 GraphQL 回應 data 區段中 errors 欄位傳回物件的目前值。

- 重試要求，但更正 expectedVersion：

```

mutation deletePost {
  deletePost(
    id:123
    expectedVersion: 1
  ) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}

```

- 選擇 Execute query (執行查詢) (橘色播放按鈕)。
- 這次請求成功，並傳回從 DynamoDB 刪除的值：

```

{
  "data": {
    "deletePost": {
      "id": "123",
      "author": "AUTHORNAME",
      "title": "Our second post!",
      "content": "A new post.",
      "url": "https://aws.amazon.com/appsync/",
      "ups": 1,

```



```

      "downs": 0,
      "version": 1
    }
  }
}
```

- 再次選擇 Execute query (執行查詢)。
- 呼叫仍然成功，但這次不會傳回任何值，因為文章已在 DynamoDB 中刪除。

```

{
  "data": {
    "deletePost": null
  }
}
```

設定 allPost 解析程式 (DynamoDB Scan)

到目前為止，如果您知道要尋找之每個文章的 id，API 才有用。讓我們新增新的解析程式，傳回資料表中的所有文章。

- 選擇 Schema (結構描述) 標籤。
- 修改 Schema (結構描述) 窗格中的 Query 類型以新增 allPost 查詢，如下所示：

```

type Query {
  allPost(count: Int, nextToken: String): PaginatedPosts!
  getPost(id: ID): Post
}
```

- 新增 PaginationPosts 類型：

```

type PaginatedPosts {
  posts: [Post!]!
  nextToken: String
}
```

- 選擇儲存。
- 在右側的資料類型窗格中，尋找查詢類型上新建立allPost欄位，然後選擇連接。
- 在動作功能表中，選擇更新執行時間，然後選擇單位解析程式（僅限 VTL）。
- 在 Data source name (資料來源名稱) 中，選擇 PostDynamoDBTable。

- 將以下內容貼到 Configure the request mapping template (設定請求映射範本) 區段：

```
{
  "version" : "2017-02-28",
  "operation" : "Scan"
  #if( ${context.arguments.count} )
    , "limit": $util.toJson($context.arguments.count)
  #end
  #if( ${context.arguments.nextToken} )
    , "nextToken": $util.toJson($context.arguments.nextToken)
  #end
}
```

此解析程式有兩個選用引數：count 會指定單一呼叫中傳回的項目上限數，nextToken 可用來擷取下一組結果 (稍後您將顯示 nextToken 值來自何處)。

- 將以下內容貼到 Configure the response mapping template (設定回應映射範本) 區段：

```
{
  "posts": $utils.toJson($context.result.items)
  #if( ${context.result.nextToken} )
    , "nextToken": $util.toJson($context.result.nextToken)
  #end
}
```

注意：到目前為止，此回應映射範本與所有其他範本皆不同。allPost 查詢結果是 PaginatedPosts，其中包含文章清單和分頁字符。此物件的形狀與從 the AWS AppSync DynamoDB Resolver 傳回的物件不同：文章清單在 the AWS AppSync DynamoDB Resolver 結果 items 中稱為，但在 posts 中稱為 PaginatedPosts。

- 選擇儲存。

如需有關 Scan 要求映射的詳細資訊，請參閱[掃描](#)參考文件。

呼叫 API 以掃描所有文章

現在已設定解析程式，AWS AppSync 知道如何將傳入 allPost 查詢轉譯為 DynamoDB Scan 操作。您現在可以掃描資料表來擷取所有文章。

在您可以嘗試之前，您必須將一些資料填入資料表，因為您已刪除目前為止所使用的項目。

- 選擇 Queries (查詢) 標籤。

- 將以下變動貼到 Queries (查詢) 窗格：

```
mutation addPost {  
  post1: addPost(id:1 author: "AUTHORNAME" title: "A series of posts, Volume 1"  
    content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }  
  post2: addPost(id:2 author: "AUTHORNAME" title: "A series of posts, Volume 2"  
    content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }  
  post3: addPost(id:3 author: "AUTHORNAME" title: "A series of posts, Volume 3"  
    content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }  
  post4: addPost(id:4 author: "AUTHORNAME" title: "A series of posts, Volume 4"  
    content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }  
  post5: addPost(id:5 author: "AUTHORNAME" title: "A series of posts, Volume 5"  
    content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }  
  post6: addPost(id:6 author: "AUTHORNAME" title: "A series of posts, Volume 6"  
    content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }  
  post7: addPost(id:7 author: "AUTHORNAME" title: "A series of posts, Volume 7"  
    content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }  
  post8: addPost(id:8 author: "AUTHORNAME" title: "A series of posts, Volume 8"  
    content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }  
  post9: addPost(id:9 author: "AUTHORNAME" title: "A series of posts, Volume 9"  
    content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }  
}
```

- 選擇 Execute query (執行查詢) (橘色播放按鈕)。

現在，讓我們來掃描資料表，一次會傳回五個結果。

- 將以下查詢貼到 Queries (查詢) 窗格中：

```
query allPost {  
  allPost(count: 5) {  
    posts {  
      id  
      title  
    }  
    nextToken  
  }  
}
```

- 選擇 Execute query (執行查詢) (橘色播放按鈕)。
- 前五篇文章應該會出現在查詢窗格右側的結果窗格中。其看起來與下列類似：

```
{
  "data": {
    "allPost": {
      "posts": [
        {
          "id": "5",
          "title": "A series of posts, Volume 5"
        },
        {
          "id": "1",
          "title": "A series of posts, Volume 1"
        },
        {
          "id": "6",
          "title": "A series of posts, Volume 6"
        },
        {
          "id": "9",
          "title": "A series of posts, Volume 9"
        },
        {
          "id": "7",
          "title": "A series of posts, Volume 7"
        }
      ],
      "nextToken":
        "eyJ2ZXJzaW9uIjoxLCJ0b2t1biI6IkFRSUNBSGo4eHR0RG0xWXhUa1F0cEhXMEp1R3B0M1B3eTh0SmRvcG9ad2RHYjI"
    }
  }
}
```

您取得五個結果以及 `nextToken`，您可以用它來取得下一組結果。

- 更新 `allPost` 查詢以包括來自之前結果組的 `nextToken`：

```
query allPost {
  allPost(
    count: 5
    nextToken:
      "eyJ2ZXJzaW9uIjoxLCJ0b2t1biI6IkFRSUNBSGo4eHR0RG0xWXhUa1F0cEhXMEp1R3B0M1B3eTh0SmRvcG9ad2RHYjI"
  ) {
```

```
    posts {
      id
      author
    }
    nextToken
  }
}
```

- 選擇 Execute query (執行查詢) (橘色播放按鈕)。
- 剩餘的四篇文章應該會出現在查詢窗格右側的結果窗格中。在這組結果中沒有 nextToken，因為您已翻遍了所有九篇文章，已經沒有剩餘的文章。其看起來與下列類似：

```
{
  "data": {
    "allPost": {
      "posts": [
        {
          "id": "2",
          "title": "A series of posts, Volume 2"
        },
        {
          "id": "3",
          "title": "A series of posts, Volume 3"
        },
        {
          "id": "4",
          "title": "A series of posts, Volume 4"
        },
        {
          "id": "8",
          "title": "A series of posts, Volume 8"
        }
      ],
      "nextToken": null
    }
  }
}
```

設定 allPostsByAuthor Resolver (DynamoDB 查詢)

除了掃描 DynamoDB 的所有文章之外，您也可以查詢 DynamoDB 以擷取特定作者建立的文章。您先前建立的 DynamoDB 資料表已有 GlobalSecondaryIndex 稱為的 author-index，您可以使用 DynamoDBQuery 操作來擷取特定作者建立的所有文章。

- 選擇 Schema (結構描述) 標籤。
- 修改 Schema (結構描述) 窗格中的 Query 類型以新增 allPostsByAuthor 查詢，如下所示：

```
type Query {  
  allPostsByAuthor(author: String!, count: Int, nextToken: String): PaginatedPosts!  
  allPost(count: Int, nextToken: String): PaginatedPosts!  
  getPost(id: ID): Post  
}
```

注意：這會使用與您在 allPost 查詢中使用的相同 PaginatedPosts 類型。

- 選擇儲存。
- 在右側的資料類型窗格中，尋找查詢類型上新建立的 allPostsByAuthor 欄位，然後選擇連接。
- 在動作功能表中，選擇更新執行時間，然後選擇單位解析程式（僅限 VTL）。
- 在 Data source name (資料來源名稱) 中，選擇 PostDynamoDBTable。
- 將以下內容貼到 Configure the request mapping template (設定請求映射範本) 區段：

```
{  
  "version" : "2017-02-28",  
  "operation" : "Query",  
  "index" : "author-index",  
  "query" : {  
    "expression": "author = :author",  
    "expressionValues" : {  
      ":author" : $util.dynamodb.toDynamoDBJson($context.arguments.author)  
    }  
  }  
  #if( ${context.arguments.count} )  
    , "limit": $util.toJson($context.arguments.count)  
  #end  
  #if( ${context.arguments.nextToken} )  
    , "nextToken": "${context.arguments.nextToken}"  
  #end  
}
```

如同 `allPost` 解析程式，此解析程式有兩個可選引數：`count`，其會指定項目的上限數，以在單一呼叫中傳回，以及 `nextToken`，其可用來擷取接下來的結果組 (您可從先前的呼叫中取得 `nextToken` 值)。

- 將以下內容貼到 `Configure the response mapping template (設定回應映射範本)` 區段：

```
{
  "posts": $utils.toJson($context.result.items)
  #if( ${context.result.nextToken} )
    , "nextToken": $util.toJson($context.result.nextToken)
  #end
}
```

注意：這是與您在 `allPost` 解析程式中所使用的相同回應映射範本。

- 選擇儲存。

如需有關 Query 要求映射的詳細資訊，請參閱[查詢](#)參考文件。

呼叫 API 來查詢某個作者的所有文章

現在已設定解析程式，AWS AppSync 知道如何根據 `author-index` 索引將傳入 `allPostsByAuthor` 變動轉換為 `DynamoDBQuery` 操作。您現在可以查詢資料表以擷取特定作者的所有文章。

在這麼做之前，讓我們再將一些文章填入資料表，因為目前為止每篇文章的作者都一樣。

- 選擇 `Queries (查詢)` 標籤。
- 將以下變動貼到 `Queries (查詢)` 窗格：

```
mutation addPost {
  post1: addPost(id:10 author: "Nadia" title: "The cutest dog in the world" content:
    "So cute. So very, very cute." url: "https://aws.amazon.com/appsync/" ) { author,
    title }
  post2: addPost(id:11 author: "Nadia" title: "Did you know...?" content: "AppSync
    works offline?" url: "https://aws.amazon.com/appsync/" ) { author, title }
  post3: addPost(id:12 author: "Steve" title: "I like GraphQL" content: "It's great"
    url: "https://aws.amazon.com/appsync/" ) { author, title }
}
```

- 選擇 `Execute query (執行查詢)` (橘色播放按鈕)。

現在，讓我們查詢資料表，傳回 Nadia 撰寫的所有文章。

- 將以下查詢貼到 Queries (查詢) 窗格中：

```
query allPostsByAuthor {
  allPostsByAuthor(author: "Nadia") {
    posts {
      id
      title
    }
    nextToken
  }
}
```

- 選擇 Execute query (執行查詢) (橘色播放按鈕)。
- Nadia 撰寫的所有文章應該會出現在查詢窗格右側的結果窗格中。其看起來與下列類似：

```
{
  "data": {
    "allPostsByAuthor": {
      "posts": [
        {
          "id": "10",
          "title": "The cutest dog in the world"
        },
        {
          "id": "11",
          "title": "Did you know...?"
        }
      ],
      "nextToken": null
    }
  }
}
```

分頁適用於 Query，如同它適用於 Scan。例如，讓我們查詢 AUTHORNAME 的所有文章，一次取得五篇。

- 將以下查詢貼到 Queries (查詢) 窗格中：

```
query allPostsByAuthor {
```



```

allPostsByAuthor(
  author: "AUTHORNAME"
  count: 5
) {
  posts {
    id
    title
  }
  nextToken
}
}

```

- 選擇 Execute query (執行查詢) (橘色播放按鈕)。
- AUTHORNAME 撰寫的所有文章應該會出現在查詢窗格右側的結果窗格中。其看起來與下列類似：

```

{
  "data": {
    "allPostsByAuthor": {
      "posts": [
        {
          "id": "6",
          "title": "A series of posts, Volume 6"
        },
        {
          "id": "4",
          "title": "A series of posts, Volume 4"
        },
        {
          "id": "2",
          "title": "A series of posts, Volume 2"
        },
        {
          "id": "7",
          "title": "A series of posts, Volume 7"
        },
        {
          "id": "1",
          "title": "A series of posts, Volume 1"
        }
      ],
      "nextToken":
"eyJ2ZXJzaW9uIjozLCJ0b2t1biI6IkFRSUNBSGo4eHR0RG0xWXhUa1F0cEhXMEp1R3B0M1B3eTh0SmRvcG9ad2RHYjI
    }
  }
}

```

```
}
}
```

- 使用之前查詢傳回的值更新 nextToken 引數，如下所示：

```
query allPostsByAuthor {
  allPostsByAuthor(
    author: "AUTHORNAME"
    count: 5
    nextToken:
"eyJ2ZXJzaW9uIjozLCJ0b2t1biI6IkFRSUNBSGo4eHR0RG0xWXhUa1F0cEhXMEp1R3B0M1B3eTh0SmRvcG9ad2RHYjI
  ) {
    posts {
      id
      title
    }
    nextToken
  }
}
```

- 選擇 Execute query (執行查詢) (橘色播放按鈕)。
- AUTHORNAME 撰寫的其餘文章應該會出現在查詢窗格右側的結果窗格中。其看起來與下列類似：

```
{
  "data": {
    "allPostsByAuthor": {
      "posts": [
        {
          "id": "8",
          "title": "A series of posts, Volume 8"
        },
        {
          "id": "5",
          "title": "A series of posts, Volume 5"
        },
        {
          "id": "3",
          "title": "A series of posts, Volume 3"
        },
        {
          "id": "9",
          "title": "A series of posts, Volume 9"
        }
      ]
    }
  }
}
```

```
    ],  
    "nextToken": null  
  }  
}  
}
```

使用集

到目前為止，Post 類型為統一金鑰/值物件。您也可以使用 AWS AppSyncDynamoDB 解析程式建立複雜物件的模型，例如集合、清單和映射。

讓我們將 Post 類型更新為包含標籤。文章可以有 0 個或多個標籤，這些標籤會以字串集的形式存放在 DynamoDB 中。您也會設定一些變動來新增和移除標籤，以及新查詢以掃描含特定標籤的文章。

- 選擇 Schema (結構描述) 標籤。
- 修改 Schema (結構描述) 窗格中的 Post 類型以新增 tags 欄位，如下所示：

```
type Post {  
  id: ID!  
  author: String  
  title: String  
  content: String  
  url: String  
  ups: Int!  
  downs: Int!  
  version: Int!  
  tags: [String!]  
}
```

- 修改 Schema (結構描述) 窗格中的 Query 類型以新增 allPostsByTag 查詢，如下所示：

```
type Query {  
  allPostsByTag(tag: String!, count: Int, nextToken: String): PaginatedPosts!  
  allPostsByAuthor(author: String!, count: Int, nextToken: String): PaginatedPosts!  
  allPost(count: Int, nextToken: String): PaginatedPosts!  
  getPost(id: ID): Post  
}
```

- 修改 Schema (結構描述) 窗格中的 Mutation 類型以新增 addTag 和 removeTag 變動，如下所示：

```

type Mutation {
  addTag(id: ID!, tag: String!): Post
  removeTag(id: ID!, tag: String!): Post
  deletePost(id: ID!, expectedVersion: Int): Post
  upvotePost(id: ID!): Post
  downvotePost(id: ID!): Post
  updatePost(
    id: ID!,
    author: String,
    title: String,
    content: String,
    url: String,
    expectedVersion: Int!
  ): Post
  addPost(
    author: String!,
    title: String!,
    content: String!,
    url: String!
  ): Post!
}

```

- 選擇儲存。
- 在右側的資料類型窗格中，尋找查詢類型上新建立的 allPostsByTag 欄位，然後選擇連接。
- 在 Data source name (資料來源名稱) 中，選擇 PostDynamoDBTable。
- 將以下內容貼到 Configure the request mapping template (設定請求映射範本) 區段：

```

{
  "version" : "2017-02-28",
  "operation" : "Scan",
  "filter": {
    "expression": "contains (tags, :tag)",
    "expressionValues": {
      ":tag": $util.dynamodb.toDynamoDBJson($context.arguments.tag)
    }
  }
}
#if( ${context.arguments.count} )
  , "limit": $util.toJson($context.arguments.count)
#end
#if( ${context.arguments.nextToken} )
  , "nextToken": $util.toJson($context.arguments.nextToken)

```

```
#end
}
```

- 將以下內容貼到 Configure the response mapping template (設定回應映射範本) 區段：

```
{
  "posts": $utils.toJson($context.result.items)
  #if( ${context.result.nextToken} )
    , "nextToken": $util.toJson($context.result.nextToken)
  #end
}
```

- 選擇儲存。
- 在右側的資料類型窗格中，尋找變動類型上新建立的 addTag 欄位，然後選擇連接。
- 在 Data source name (資料來源名稱) 中，選擇 PostDynamoDBTable。
- 將以下內容貼到 Configure the request mapping template (設定請求映射範本) 區段：

```
{
  "version" : "2017-02-28",
  "operation" : "UpdateItem",
  "key" : {
    "id" : $util.dynamodb.toDynamoDBJson($context.arguments.id)
  },
  "update" : {
    "expression" : "ADD tags :tags, version :plusOne",
    "expressionValues" : {
      ":tags" : { "SS": [ $util.toJson($context.arguments.tag) ] },
      ":plusOne" : { "N" : 1 }
    }
  }
}
```

- 將以下內容貼到 Configure the response mapping template (設定回應映射範本) 區段：

```
$utils.toJson($context.result)
```

- 選擇儲存。
- 在右側的資料類型窗格中，尋找變動類型上新建立的 removeTag 欄位，然後選擇連接。
- 在 Data source name (資料來源名稱) 中，選擇 PostDynamoDBTable。
- 將以下內容貼到 Configure the request mapping template (設定請求映射範本) 區段：

```
{
  "version" : "2017-02-28",
  "operation" : "UpdateItem",
  "key" : {
    "id" : $util.dynamodb.toDynamoDBJson($context.arguments.id)
  },
  "update" : {
    "expression" : "DELETE tags :tags ADD version :plusOne",
    "expressionValues" : {
      ":tags" : { "SS": [ $util.toJson($context.arguments.tag) ] },
      ":plusOne" : { "N" : 1 }
    }
  }
}
```

- 將以下內容貼到 Configure the response mapping template (設定回應映射範本) 區段：

```
$utils.toJson($context.result)
```

- 選擇儲存。

呼叫 API 來使用標籤

現在您已設定解析程式，AWS AppSync 知道如何將傳入的 `removeTag`、`addTag` 和 `allPostsByTag` 請求轉譯為 `DynamoDBUpdateItem` 和 `Scan` 操作。

為了嘗試看看，讓我們選擇您先前建立的其中一篇文章。例如，讓我們使用 Nadia 撰寫的一篇文章。

- 選擇 Queries (查詢) 標籤。
- 將以下查詢貼到 Queries (查詢) 窗格中：

```
query allPostsByAuthor {
  allPostsByAuthor(
    author: "Nadia"
  ) {
    posts {
      id
      title
    }
    nextToken
  }
}
```

```
}
```

- 選擇 Execute query (執行查詢) (橘色播放按鈕)。
- Nadia 的所有文章應該會出現在查詢窗格右側的結果窗格中。其看起來與下列類似：

```
{
  "data": {
    "allPostsByAuthor": {
      "posts": [
        {
          "id": "10",
          "title": "The cutest dog in the world"
        },
        {
          "id": "11",
          "title": "Did you known...?"
        }
      ],
      "nextToken": null
    }
  }
}
```

- 我們使用標題為 "The cutest dog in the world" 的文章。請記下其 id，因為您稍後將會用到它。

現在，讓我們嘗試新增 dog 標籤。

- 將以下變動貼到 Queries (查詢) 窗格。您也必須將 id 引數更新為您先前記下的值。

```
mutation addTag {
  addTag(id:10 tag: "dog") {
    id
    title
    tags
  }
}
```

- 選擇 Execute query (執行查詢) (橘色播放按鈕)。
- 使用新標籤更新的文章。

```
{
  "data": {
    "addTag": {
      "id": "10",
      "title": "The cutest dog in the world",
      "tags": [
        "dog"
      ]
    }
  }
}
```

您可以新增更多的標籤，如下所示：

- 更新變動以將 tag 引數變更為 puppy。

```
mutation addTag {
  addTag(id:10 tag: "puppy") {
    id
    title
    tags
  }
}
```

- 選擇 Execute query (執行查詢) (橘色播放按鈕)。
- 使用新標籤更新的文章。

```
{
  "data": {
    "addTag": {
      "id": "10",
      "title": "The cutest dog in the world",
      "tags": [
        "dog",
        "puppy"
      ]
    }
  }
}
```


您也可以刪除標籤：

- 將以下變動貼到 Queries (查詢) 窗格。您也必須將 `id` 引數更新為您先前記下的值。

```
mutation removeTag {
  removeTag(id:10 tag: "puppy") {
    id
    title
    tags
  }
}
```

- 選擇 Execute query (執行查詢) (橘色播放按鈕)。
- 貼文將會更新，且 puppy 標籤將會刪除。

```
{
  "data": {
    "addTag": {
      "id": "10",
      "title": "The cutest dog in the world",
      "tags": [
        "dog"
      ]
    }
  }
}
```

您也可以搜尋所有包含標籤的貼文：

- 將以下查詢貼到 Queries (查詢) 窗格中：

```
query allPostsByTag {
  allPostsByTag(tag: "dog") {
    posts {
      id
      title
      tags
    }
    nextToken
  }
}
```

- 選擇 Execute query (執行查詢) (橘色播放按鈕)。
- 具有 dog 標籤的所有文章將會傳回，如下所示：

```
{
  "data": {
    "allPostsByTag": {
      "posts": [
        {
          "id": "10",
          "title": "The cutest dog in the world",
          "tags": [
            "dog",
            "puppy"
          ]
        }
      ],
      "nextToken": null
    }
  }
}
```

使用清單和映射

除了使用 DynamoDB 集之外，您也可以使用 DynamoDB 清單和映射，在單一物件中建立複雜資料的模型。

讓我們新增功能以將評論新增到文章。這將建模為 DynamoDB 中物件上的映射 Post 物件清單。

注意：在實際應用程式中，您會在它們自己的資料表中塑造評論。在本教學課程中，您將只是將它們新增至 Post 資料表。

- 選擇 Schema (結構描述) 標籤。
- 修改 Schema (結構描述) 窗格中新增 Comment 類型，如下所示：

```
type Comment {
  author: String!
  comment: String!
}
```

- 修改 Schema (結構描述) 窗格中的 Post 類型以新增 comments 欄位，如下所示：

```
type Post {  
  id: ID!  
  author: String  
  title: String  
  content: String  
  url: String  
  ups: Int!  
  downs: Int!  
  version: Int!  
  tags: [String!]  
  comments: [Comment!]  
}
```

- 修改 Schema (結構描述) 窗格中的 Mutation 類型來新增 addComment 變動，如下所示：

```
type Mutation {  
  addComment(id: ID!, author: String!, comment: String!): Post  
  addTag(id: ID!, tag: String!): Post  
  removeTag(id: ID!, tag: String!): Post  
  deletePost(id: ID!, expectedVersion: Int): Post  
  upvotePost(id: ID!): Post  
  downvotePost(id: ID!): Post  
  updatePost(  
    id: ID!,  
    author: String,  
    title: String,  
    content: String,  
    url: String,  
    expectedVersion: Int!  
  ): Post  
  addPost(  
    author: String!,  
    title: String!,  
    content: String!,  
    url: String!  
  ): Post!  
}
```

- 選擇儲存。
- 在右側的資料類型窗格中，尋找變動類型上新建立的 addComment 欄位，然後選擇連接。
- 在 Data source name (資料來源名稱) 中，選擇 PostDynamoDBTable。

- 將以下內容貼到 Configure the request mapping template (設定請求映射範本) 區段：

```
{
  "version" : "2017-02-28",
  "operation" : "UpdateItem",
  "key" : {
    "id" : $util.dynamodb.toDynamoDBJson($context.arguments.id)
  },
  "update" : {
    "expression" : "SET comments =
list_append(if_not_exists(comments, :emptyList), :newComment) ADD version :plusOne",
    "expressionValues" : {
      ":emptyList": { "L" : [] },
      ":newComment" : { "L" : [
        { "M": {
          "author": $util.dynamodb.toDynamoDBJson($context.arguments.author),
          "comment": $util.dynamodb.toDynamoDBJson($context.arguments.comment)
        }
      }
    ] },
    ":plusOne" : $util.dynamodb.toDynamoDBJson(1)
  }
}
```

這個更新表達式將會將包含新評論的清單附加到現有的 `comments` 清單。如果清單不存在，則會建立一個。

- 將以下內容貼到 Configure the response mapping template (設定回應映射範本) 區段：

```
$utils.toJson($context.result)
```

- 選擇儲存。

呼叫 API 以新增評論

現在您已設定解析程式，AWS AppSync 知道如何將傳入 `addComment` 的請求轉換為 `DynamoDBUpdateItem` 操作。

讓我們透過將評論新增至您已新增標籤的相同文章，以嘗試此操作。

- 選擇 Queries (查詢) 標籤。

- 將以下查詢貼到 Queries (查詢) 窗格中：

```
mutation addComment {
  addComment(
    id:10
    author: "Steve"
    comment: "Such a cute dog."
  ) {
    id
    comments {
      author
      comment
    }
  }
}
```

- 選擇 Execute query (執行查詢) (橘色播放按鈕)。
- Nadia 的所有文章應該會出現在查詢窗格右側的結果窗格中。其看起來與下列類似：

```
{
  "data": {
    "addComment": {
      "id": "10",
      "comments": [
        {
          "author": "Steve",
          "comment": "Such a cute dog."
        }
      ]
    }
  }
}
```

如果您多次執行要求，系統會將多個評論附加到清單。

結論

在本教學課程中，您已建置 API，讓我們可以使用 AWS AppSync 和 GraphQL 在 DynamoDB 中操作 Post 物件。如需詳細資訊，請參閱[解析程式映射範本參考](#)。

若要清除，您可以從主控台將 AppSync GraphQL API 刪除。

若要刪除您為此教學課程建立的 DynamoDB 資料表和 IAM 角色，您可以執行下列動作來刪除AWSAppSyncTutorialForAmazonDynamoDB堆疊，或造訪 AWS CloudFormation 主控台並刪除堆疊：

```
aws cloudformation delete-stack \  
  --stack-name AWSAppSyncTutorialForAmazonDynamoDB
```

在 中使用 AWS Lambda 解析程式 AWS AppSync

Note

我們現在主要支援 APPSYNC_JS 執行期及其文件。請考慮[在此處](#)使用 APPSYNC_JS 執行期及其指南。

您可以使用 AWS Lambda with AWS AppSync 來解析任何 GraphQL 欄位。例如，GraphQL 查詢可能會將呼叫傳送至 Amazon Relational Database Service (Amazon RDS) 執行個體，而 GraphQL 變動可能會寫入 Amazon Kinesis 串流。在本節中，我們將示範如何撰寫 Lambda 函數，根據 GraphQL 欄位操作的調用執行商業邏輯。

建立 Lambda 函數

下列範例顯示寫入的 Lambda 函數Node.js，該函數會在部落格文章應用程式中對部落格文章執行不同的操作。

```
exports.handler = (event, context, callback) => {  
  console.log("Received event {}", JSON.stringify(event, 3));  
  var posts = {  
    "1": {"id": "1", "title": "First book", "author": "Author1", "url": "https://  
amazon.com/", "content": "SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR  
1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1", "ups": "100",  
"downs": "10"},  
    "2": {"id": "2", "title": "Second book", "author": "Author2", "url": "https://  
amazon.com", "content": "SAMPLE TEXT AUTHOR 2 SAMPLE TEXT AUTHOR 2 SAMPLE TEXT", "ups":  
"100", "downs": "10"},  
    "3": {"id": "3", "title": "Third book", "author": "Author3", "url": null,  
"content": null, "ups": null, "downs": null },  
    "4": {"id": "4", "title": "Fourth book", "author": "Author4", "url": "https://  
www.amazon.com/", "content": "SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT
```

```

AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT
AUTHOR 4 SAMPLE TEXT AUTHOR 4", "ups": "1000", "downs": "0"},
    "5": {"id": "5", "title": "Fifth book", "author": "Author5", "url": "https://
www.amazon.com/", "content": "SAMPLE TEXT AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT
AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT", "ups": "50", "downs": "0"} }];

var relatedPosts = {
    "1": [posts['4']],
    "2": [posts['3'], posts['5']],
    "3": [posts['2'], posts['1']],
    "4": [posts['2'], posts['1']],
    "5": []
};

console.log("Got an Invoke Request.");
switch(event.field) {
    case "getPost":
        var id = event.arguments.id;
        callback(null, posts[id]);
        break;
    case "allPosts":
        var values = [];
        for(var d in posts){
            values.push(posts[d]);
        }
        callback(null, values);
        break;
    case "addPost":
        // return the arguments back
        callback(null, event.arguments);
        break;
    case "addPostErrorWithData":
        var id = event.arguments.id;
        var result = posts[id];
        // attached additional error information to the post
        result.errorMessage = 'Error with the mutation, data has changed';
        result.errorType = 'MUTATION_ERROR';
        callback(null, result);
        break;
    case "relatedPosts":
        var id = event.source.id;
        callback(null, relatedPosts[id]);
        break;
    default:

```

```
        callback("Unknown field, unable to resolve" + event.field, null);
        break;
    }
};
```

此 Lambda 函數會依 ID 擷取文章、新增文章、擷取文章清單，以及擷取特定文章的相關文章。

注意：Lambda 函數使用上的 switch 陳述式 `event.field` 來判斷目前正在解析哪個欄位。

使用 AWS 管理主控台或 AWS CloudFormation 堆疊建立此 Lambda 函數。若要從 CloudFormation 堆疊建立函數，您可以使用下列 AWS Command Line Interface (AWS CLI) 命令：

```
aws cloudformation create-stack --stack-name AppSyncLambdaExample \
--template-url https://s3.us-west-2.amazonaws.com/awsappsync/resources/lambda/
LambdaCFTemplate.yaml \
--capabilities CAPABILITY_NAMED_IAM
```

您也可以從這裡在美國西部（奧勒岡）AWS 區域中啟動 AWS CloudFormation 堆疊：

A yellow button with a blue border and a play icon, labeled "Launch Stack".

設定 Lambda 的資料來源

建立 Lambda 函數後，在 AWS AppSync 主控台中導覽至 GraphQL API，然後選擇資料來源索引標籤。

選擇建立資料來源，輸入易記的資料來源名稱（例如 **Lambda**），然後針對資料來源類型選擇 AWS Lambda 函數。針對區域，選擇與函數相同的區域。（如果您從提供的 CloudFormation 堆疊建立函數，則函數可能位於 US-WEST-2。）針對函數 ARN，選擇 Lambda 函數的 Amazon Resource Name (ARN)。

選擇 Lambda 函數後，您可以建立新的 AWS Identity and Access Management (IAM) 角色（AWS AppSync 會為其指派適當的許可），或選擇具有下列內嵌政策的現有角色：

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
```



```

    {
      "Effect": "Allow",
      "Action": [
        "lambda:InvokeFunction"
      ],
      "Resource": "arn:aws:lambda:REGION:ACCOUNTNUMBER:function/
LAMBDA_FUNCTION"
    }
  ]
}

```

您還必須為 IAM 角色設定與 AWS AppSync 的信任關係，如下所示：

JSON

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "appsync.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}

```

建立 GraphQL 結構描述

現在資料來源已連接至 Lambda 函數，請建立 GraphQL 結構描述。

從 AWS AppSync 主控台的結構描述編輯器，請確定您的結構描述符合下列結構描述：

```

schema {
  query: Query
  mutation: Mutation
}

type Query {

```

```
    getPost(id:ID!): Post
    allPosts: [Post]
  }

  type Mutation {
    addPost(id: ID!, author: String!, title: String, content: String, url: String):
    Post!
  }

  type Post {
    id: ID!
    author: String!
    title: String
    content: String
    url: String
    ups: Int
    downs: Int
    relatedPosts: [Post]
  }
```

設定解析程式

現在您已註冊 Lambda 資料來源和有效的 GraphQL 結構描述，您可以使用解析程式將 GraphQL 欄位連線至 Lambda 資料來源。

若要建立解析程式，您需要映射範本。若要進一步了解映射範本，請參閱 [Resolver Mapping Template Overview](#)。

如需 Lambda 映射範本的詳細資訊，請參閱 [Resolver mapping template reference for Lambda](#)。

在此步驟中，您將解析程式連接至下列欄位的 Lambda 函數：getPost(id:ID!): Post、addPost(id: ID!, author: String!, title: String, content: String, url: String): Post!、allPosts: [Post]和 Post.relatedPosts: [Post]。

從 AWS AppSync 主控台的結構描述編輯器的右側，選擇連接解析程式getPost(id:ID!): Post。

然後，在動作功能表中，選擇更新執行時間，然後選擇單位解析程式（僅限 VTL）。

之後，選擇您的 Lambda 資料來源。在 request mapping template (請求映射範本) 區段中，選擇 Invoke And Forward Arguments (叫用並轉發引數)。

修改 payload 物件以新增欄位名稱。您的範本看起來應該如下所示：

```
{
  "version": "2017-02-28",
  "operation": "Invoke",
  "payload": {
    "field": "getPost",
    "arguments": $utils.toJson($context.arguments)
  }
}
```

在 response mapping template (回應映射範本) 區段中，選擇 Return Lambda Result (傳回 Lambda 結果)。

在此案例中，使用原本的基礎範本。它看起來應該如下所示：

```
$utils.toJson($context.result)
```

選擇儲存。您已成功連接第一個解析程式。對剩餘欄位重複此操作，如下所示：

`addPost(id: ID!, author: String!, title: String, content: String, url: String): Post!` 請求映射範本：

```
{
  "version": "2017-02-28",
  "operation": "Invoke",
  "payload": {
    "field": "addPost",
    "arguments": $utils.toJson($context.arguments)
  }
}
```

`addPost(id: ID!, author: String!, title: String, content: String, url: String): Post!` 回應映射範本：

```
$utils.toJson($context.result)
```

`allPosts: [Post]` 請求映射範本：

```
{
  "version": "2017-02-28",
  "operation": "Invoke",
```

```
"payload": {
  "field": "allPosts"
}
```

allPosts: [Post] 回應映射範本：

```
$utils.toJson($context.result)
```

Post.relatedPosts: [Post] 請求映射範本：

```
{
  "version": "2017-02-28",
  "operation": "Invoke",
  "payload": {
    "field": "relatedPosts",
    "source": $utils.toJson($context.source)
  }
}
```

Post.relatedPosts: [Post] 回應映射範本：

```
$utils.toJson($context.result)
```

測試 GraphQL API

現在，您的 Lambda 函式已經連接到 GraphQL 解析程式，您可以使用主控台或用戶端應用程式執行一些變動和查詢。

在 AWS AppSync 主控台的左側，選擇查詢，然後貼上下列程式碼：

addPost Mutation

```
mutation addPost {
  addPost(
    id: 6
    author: "Author6"
    title: "Sixth book"
    url: "https://www.amazon.com/"
    content: "This is the book is a tutorial for using GraphQL with AWS AppSync."
  )
}
```

```
    ) {  
      id  
      author  
      title  
      content  
      url  
      ups  
      downs  
    }  
  }  
}
```

getPost Query

```
query getPost {  
  getPost(id: "2") {  
    id  
    author  
    title  
    content  
    url  
    ups  
    downs  
  }  
}
```

allPosts Query

```
query allPosts {  
  allPosts {  
    id  
    author  
    title  
    content  
    url  
    ups  
    downs  
    relatedPosts {  
      id  
      title  
    }  
  }  
}
```

傳回錯誤

任何指定的欄位解析都可能導致錯誤。透過 AWS AppSync，您可以從下列來源引發錯誤：

- 請求或回應映射範本
- Lambda 函數

從映射範本

若要引發刻意錯誤，您可以使用 Velocity 範本語言 (VTL) 範本中的 `$utils.error` 協助程式方法。它使用 `errorMessage`、`errorType`，及選用的 `data` 值做為引數。發生錯誤時，`data` 可將額外的資料傳回給用戶端。`data` 物件會新增到 GraphQL 最後回應中的 `errors`。

以下範例說明如何在 `Post.relatedPosts: [Post]` 回應映射範本中使用它：

```
$utils.error("Failed to fetch relatedPosts", "LambdaFailure", $context.result)
```

這會產生類似下列的 GraphQL 回應：

```
{
  "data": {
    "allPosts": [
      {
        "id": "2",
        "title": "Second book",
        "relatedPosts": null
      },
      ...
    ]
  },
  "errors": [
    {
      "path": [
        "allPosts",
        0,
        "relatedPosts"
      ],
      "errorType": "LambdaFailure",
      "locations": [
        {
          "line": 5,
```

```

        "column": 5
      }
    ],
    "message": "Failed to fetch relatedPosts",
    "data": [
      {
        "id": "2",
        "title": "Second book"
      },
      {
        "id": "1",
        "title": "First book"
      }
    ]
  }
]
}

```

其中的 `allPosts[0].relatedPosts` 為 `null`，因為錯誤以及 `errorMessage`、`errorType` 和 `data` 出現在 `data.errors[0]` 物件中。

從 Lambda 函式

AWS AppSync 也了解 Lambda 函數擲回的錯誤。Lambda 程式設計模型可讓您引發已處理的錯誤。如果 Lambda 函數擲回錯誤，AWS AppSync 無法解析目前的欄位。回應中只會設定從 Lambda 傳回的錯誤訊息。目前，您無法透過從 Lambda 函數引發錯誤，將任何無關的資料傳遞回用戶端。

注意：如果您的 Lambda 函數引發未處理的錯誤，AWS AppSync 會使用 Lambda 設定的錯誤訊息。

下列 Lambda 函式會引發錯誤：

```

exports.handler = (event, context, callback) => {
  console.log("Received event {}", JSON.stringify(event, 3));
  callback("I fail. Always.");
};

```

這會傳回類似下列的 GraphQL 回應：

```

{
  "data": {
    "allPosts": [
      {
        "id": "2",

```

```

        "title": "Second book",
        "relatedPosts": null
      },
      ...
    ]
  },
  "errors": [
    {
      "path": [
        "allPosts",
        0,
        "relatedPosts"
      ],
      "errorType": "Lambda:Handled",
      "locations": [
        {
          "line": 5,
          "column": 5
        }
      ],
      "message": "I fail. Always."
    }
  ]
}

```

進階使用案例：批次處理

此範例中的 Lambda 函數具有 `relatedPosts` 欄位，可傳回指定文章的相關文章清單。在範例查詢中，來自 Lambda 函數 `allPosts` 的欄位調用會傳回五個文章。因為我們指定了也要 `relatedPosts` 解析每個傳回的文章，所以會叫用 `relatedPosts` 欄位操作五次。

```

query allPosts {
  allPosts { // 1 Lambda invocation - yields 5 Posts
    id
    author
    title
    content
    url
    ups
    downs
    relatedPosts { // 5 Lambda invocations - each yields 5 posts
      id
      title
    }
  }
}

```



```

    }
  }
}
```

雖然這在此特定範例中可能聽起來並不重要，但這種複合式過度擷取可能會快速破壞應用程式。

如果您要在同一查詢中，再次擷取傳回之相關 Posts 的 `relatedPosts`，那麼叫用的次數將大幅增加。

```

query allPosts {
  allPosts { // 1 Lambda invocation - yields 5 Posts
    id
    author
    title
    content
    url
    ups
    downs
    relatedPosts { // 5 Lambda invocations - each yield 5 posts = 5 x 5 Posts
      id
      title
      relatedPosts { // 5 x 5 Lambda invocations - each yield 5 posts = 25 x 5
        id
        title
        author
      }
    }
  }
}
```

在此相對簡單的查詢中，AWS AppSync 會叫用 Lambda 函數 $1 + 5 + 25 = 31$ 次。

這是一個相當常見的挑戰，通常稱為 $N+1$ 問題 (在此例中， $N = 5$)，它可能會增加應用程式的延遲和成本。

解決此問題的方法之一，是將類似的欄位解析程式請求一起批次處理。在此範例中，它可以改為解析指定批次文章的相關文章清單，而不是讓 Lambda 函數解析單一指定文章的相關文章清單。

為了示範，讓我們將 `Post.relatedPosts: [Post]` 解析程式切換為啟用批次的解析程式。

在 AWS AppSync 主控台的右側，選擇現有的 `Post.relatedPosts: [Post]` 解析程式。將請求映射範本變更為以下內容：

```
{
  "version": "2017-02-28",
  "operation": "BatchInvoke",
  "payload": {
    "field": "relatedPosts",
    "source": $utils.toJson($context.source)
  }
}
```

只有 operation 欄位從 Invoke 變更為 BatchInvoke。承載欄位現在會成為範本中指定之任何項目的陣列。在此範例中，Lambda 函數會收到下列項目做為輸入：

```
[
  {
    "field": "relatedPosts",
    "source": {
      "id": 1
    }
  },
  {
    "field": "relatedPosts",
    "source": {
      "id": 2
    }
  },
  ...
]
```

在請求映射範本中指定 BatchInvoke 時，Lambda 函數會收到請求清單，並傳回結果清單。

具體而言，結果清單必須符合請求承載項目的大小和順序，以便 AWS AppSync 可以相應地符合結果。

在此批次範例中，Lambda 函數會傳回一批結果，如下所示：

```
[
  [{ "id": "2", "title": "Second book" }, { "id": "3", "title": "Third book" } ], //
  relatedPosts for id=1
  [{ "id": "3", "title": "Third book" } ]
  // relatedPosts for id=2
]
```

Node.js 中的下列 Lambda 函數示範 Post.relatedPosts 欄位的此批次處理功能，如下所示：

```
exports.handler = (event, context, callback) => {
  console.log("Received event {}", JSON.stringify(event, 3));
  var posts = {
    "1": {"id": "1", "title": "First book", "author": "Author1", "url": "https://amazon.com/", "content": "SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1", "ups": "100", "downs": "10"},
    "2": {"id": "2", "title": "Second book", "author": "Author2", "url": "https://amazon.com", "content": "SAMPLE TEXT AUTHOR 2 SAMPLE TEXT AUTHOR 2 SAMPLE TEXT", "ups": "100", "downs": "10"},
    "3": {"id": "3", "title": "Third book", "author": "Author3", "url": null, "content": null, "ups": null, "downs": null },
    "4": {"id": "4", "title": "Fourth book", "author": "Author4", "url": "https://www.amazon.com/", "content": "SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4", "ups": "1000", "downs": "0"},
    "5": {"id": "5", "title": "Fifth book", "author": "Author5", "url": "https://www.amazon.com/", "content": "SAMPLE TEXT AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT", "ups": "50", "downs": "0"} };

  var relatedPosts = {
    "1": [posts['4']],
    "2": [posts['3'], posts['5']],
    "3": [posts['2'], posts['1']],
    "4": [posts['2'], posts['1']],
    "5": []
  };

  console.log("Got a BatchInvoke Request. The payload has %d items to resolve.", event.length);
  // event is now an array
  var field = event[0].field;
  switch(field) {
    case "relatedPosts":
      var results = [];
      // the response MUST contain the same number
      // of entries as the payload array
      for (var i=0; i< event.length; i++) {
        console.log("post {}", JSON.stringify(event[i].source));
        results.push(relatedPosts[event[i].source.id]);
      }
      console.log("results {}", JSON.stringify(results));
  }
}
```

```

        callback(null, results);
        break;
    default:
        callback("Unknown field, unable to resolve" + field, null);
        break;
    }
};

```

傳回個別錯誤

先前的範例顯示，您可以從 Lambda 函數傳回單一錯誤，或從映射範本引發錯誤。對於批次調用，從 Lambda 函數引發錯誤會將整個批次標記為失敗。這對於發生無法復原錯誤的特定案例可能是可接受的，例如與資料存放區的連線失敗。不過，如果批次中的某些項目成功，而其他項目失敗，則有可能同時傳回錯誤和有效資料。Because AWS AppSync 需要批次回應來列出符合批次原始大小的元素，您必須定義資料結構，以區分有效資料與錯誤。

例如，如果 Lambda 函數預期會傳回一批相關的文章，您可以選擇傳回 Response 物件清單，其中每個物件都有選用的資料、errorMessage 和 errorType 欄位。如果出現 errorMessage 欄位，表示發生錯誤。

下列程式碼示範如何更新 Lambda 函數：

```

exports.handler = (event, context, callback) => {
    console.log("Received event {}", JSON.stringify(event, 3));
    var posts = {
        "1": {"id": "1", "title": "First book", "author": "Author1", "url": "https://amazon.com/", "content": "SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1", "ups": "100", "downs": "10"},
        "2": {"id": "2", "title": "Second book", "author": "Author2", "url": "https://amazon.com", "content": "SAMPLE TEXT AUTHOR 2 SAMPLE TEXT AUTHOR 2 SAMPLE TEXT", "ups": "100", "downs": "10"},
        "3": {"id": "3", "title": "Third book", "author": "Author3", "url": null, "content": null, "ups": null, "downs": null },
        "4": {"id": "4", "title": "Fourth book", "author": "Author4", "url": "https://www.amazon.com/", "content": "SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4", "ups": "1000", "downs": "0"},
        "5": {"id": "5", "title": "Fifth book", "author": "Author5", "url": "https://www.amazon.com/", "content": "SAMPLE TEXT AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT", "ups": "50", "downs": "0"} };
};

```

```

    var relatedPosts = {
      "1": [posts['4']],
      "2": [posts['3'], posts['5']],
      "3": [posts['2'], posts['1']],
      "4": [posts['2'], posts['1']],
      "5": []
    };

    console.log("Got a BatchInvoke Request. The payload has %d items to resolve.",
event.length);
    // event is now an array
    var field = event[0].field;
    switch(field) {
      case "relatedPosts":
        var results = [];
        results.push({ 'data': relatedPosts['1'] });
        results.push({ 'data': relatedPosts['2'] });
        results.push({ 'data': null, 'errorMessage': 'Error Happened', 'errorType':
'ERROR' });
        results.push(null);
        results.push({ 'data': relatedPosts['3'], 'errorMessage': 'Error Happened
with last result', 'errorType': 'ERROR' });
        callback(null, results);
        break;
      default:
        callback("Unknown field, unable to resolve" + field, null);
        break;
    }
  };
};

```

在此範例中，下列回應映射範本會剖析 Lambda 函數的每個項目，並引發任何發生的錯誤：

```

#if( $context.result && $context.result.errorMessage )
  $utils.error($context.result.errorMessage, $context.result.errorType,
    $context.result.data)
#else
  $utils.toJson($context.result.data)
#end

```

此範例會傳回類似下列的 GraphQL 回應：

```

{
  "data": {

```

```
"allPosts": [
  {
    "id": "1",
    "relatedPostsPartialErrors": [
      {
        "id": "4",
        "title": "Fourth book"
      }
    ]
  },
  {
    "id": "2",
    "relatedPostsPartialErrors": [
      {
        "id": "3",
        "title": "Third book"
      },
      {
        "id": "5",
        "title": "Fifth book"
      }
    ]
  },
  {
    "id": "3",
    "relatedPostsPartialErrors": null
  },
  {
    "id": "4",
    "relatedPostsPartialErrors": null
  },
  {
    "id": "5",
    "relatedPostsPartialErrors": null
  }
],
"errors": [
  {
    "path": [
      "allPosts",
      2,
      "relatedPostsPartialErrors"
    ]
  }
],
```

```
    "errorType": "ERROR",
    "locations": [
      {
        "line": 4,
        "column": 9
      }
    ],
    "message": "Error Happened"
  },
  {
    "path": [
      "allPosts",
      4,
      "relatedPostsPartialErrors"
    ],
    "data": [
      {
        "id": "2",
        "title": "Second book"
      },
      {
        "id": "1",
        "title": "First book"
      }
    ],
    "errorType": "ERROR",
    "locations": [
      {
        "line": 4,
        "column": 9
      }
    ],
    "message": "Error Happened with last result"
  }
]
```

設定批次大小上限

根據預設，使用 `時BatchInvoke`，AWS AppSync 會以最多五個項目的批次傳送請求到您的 Lambda 函數。您可以設定 Lambda 解析程式的批次大小上限。

若要在解析程式上設定批次大小上限，請在 AWS Command Line Interface () 中使用下列命令AWS CLI：

```
$ aws appsync create-resolver --api-id <api-id> --type-name Query --field-name
relatedPosts \
--request-mapping-template "<template>" --response-mapping-template "<template>" --
data-source-name "<lambda-datasource>" \
--max-batch-size X
```

Note

提供請求映射範本時，您必須使用 BatchInvoke操作來使用批次處理。

您也可以使用下列命令來啟用和設定 Direct Lambda 解析程式上的批次處理：

```
$ aws appsync create-resolver --api-id <api-id> --type-name Query --field-name
relatedPosts \
--data-source-name "<lambda-datasource>" \
--max-batch-size X
```

使用 VTL 範本的批次大小組態上限

對於具有 VTL 請求中範本的 Lambda 解析程式，除非其已直接指定為 VTL 中的BatchInvoke操作，否則最大批次大小不會生效。同樣地，如果您執行最上層變動，則不會針對變動執行批次處理，因為 GraphQL 規格需要依序執行平行變動。

例如，採取下列變動：

```
type Mutation {
  putItem(input: Item): Item
  putItems(inputs: [Item]): [Item]
}
```

使用第一個變動，我們可以建立 10Items，如以下程式碼片段所示：

```
mutation MyMutation {
  v1: putItem($someItem1) {
    id,
    name
```



```
}
v2: putItem($someItem2) {
  id,
  name
}
v3: putItem($someItem3) {
  id,
  name
}
v4: putItem($someItem4) {
  id,
  name
}
v5: putItem($someItem5) {
  id,
  name
}
v6: putItem($someItem6) {
  id,
  name
}
v7: putItem($someItem7) {
  id,
  name
}
v8: putItem($someItem8) {
  id,
  name
}
v9: putItem($someItem9) {
  id,
  name
}
v10: putItem($someItem10) {
  id,
  name
}
}
```

在此範例中，即使 Lambda Resolver 中的批次大小上限設為 10，也不會在 10 的 Items 群組中批次處理。相反地，它們會根據 GraphQL 規格循序執行。

若要執行實際批次變動，您可以使用第二個變動遵循下列範例：

```
mutation MyMutation {  
  putItems([$someItem1, $someItem2, $someItem3,$someItem4, $someItem5, $someItem6,  
    $someItem7, $someItem8, $someItem9, $someItem10]) {  
    id,  
    name  
  }  
}
```

如需搭配 Direct Lambda 解析程式使用批次處理的詳細資訊，請參閱 [Direct Lambda 解析程式](#)。

使用 Amazon OpenSearch Service 解析程式 in AWS AppSync

Note

我們現在主要支援 APPSYNC_JS 執行期及其文件。請考慮[在此處](#)使用 APPSYNC_JS 執行期及其指南。

AWS AppSync 支援從您在自己的 AWS 帳戶中佈建的網域使用 Amazon OpenSearch Service，前提是這些網域不存在於 VPC 中。在佈建網域之後，您可以使用資料來源來連線到這些網域，此時您可以在結構描述中設定解析程式，來進行 GraphQL 操作，例如查詢、變動和訂閱。本教學課程將會逐步說明一些常見的範例。

如需詳細資訊，請參閱 [OpenSearch 的解析程式映射範本參考](#)。

一鍵設定

若要在已設定 Amazon OpenSearch Service 的 AWS AppSync 中自動設定 GraphQL 端點，您可以使用此 AWS CloudFormation 範本：

Launch Stack

AWS CloudFormation 部署完成後，您可以直接跳至[執行 GraphQL 查詢和變動](#)。

建立新的 OpenSearch Service 網域

若要開始使用本教學課程，您需要現有的 OpenSearch Service 網域。如果尚未擁有，您可以使用下列範例。請注意，建立 OpenSearch Service 網域最多可能需要 15 分鐘，然後才能繼續將其與 an AWS AppSync 資料來源整合。

```
aws cloudformation create-stack --stack-name AppSyncOpenSearch \  
--template-url https://s3.us-west-2.amazonaws.com/awsappsync/resources/elasticsearch/  
ESResolverCFTemplate.yaml \  
--parameters ParameterKey=OSDomainName,ParameterValue=ddtestdomain  
ParameterKey=Tier,ParameterValue=development \  
--capabilities CAPABILITY_NAMED_IAM
```

您可以在 AWS 帳戶中的美國西部 2（奧勒岡）區域啟動下列 AWS CloudFormation 堆疊：

Launch Stack 

設定 OpenSearch Service 的資料來源

建立 OpenSearch Service 網域後，導覽至 your AWS AppSync GraphQL API，然後選擇資料來源索引標籤。選擇新增並輸入資料來源的易記名稱，例如「oss」。然後選擇資料來源類型的 Amazon OpenSearch 網域，選擇適當的區域，您應該會看到列出的 OpenSearch Service 網域。選取後，您可以建立新的角色，而 AWS AppSync 會指派適當的角色許可，或者您可以選擇具有下列內嵌政策的現有角色：

JSON

```
{  
  "Version": "2012-10-17",  
  "Statement": [  
    {  
      "Sid": "Stmt1234234",  
      "Effect": "Allow",  
      "Action": [  
        "es:ESHttpDelete",  
        "es:ESHttpHead",  
        "es:ESHttpGet",  
        "es:ESHttpPost",  
        "es:ESHttpPut"  
      ],  
      "Resource": [  
        "arn:aws:es:REGION:ACCOUNTNUMBER:domain/democluster/*"  
      ]  
    }  
  ]  
}
```

您還需要為該角色設定與 AWS AppSync 的信任關係：

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "appsync.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

此外，OpenSearch Service 網域具有自己的存取政策，您可以透過 Amazon OpenSearch Service 主控台進行修改。您需要新增類似下列的政策，以及 OpenSearch Service 網域的適當動作和資源。請注意，主體將是 AppSync 資料來源角色，如果您讓主控台建立此角色，可以在 IAM 主控台中找到。

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws:iam::ACCOUNTNUMBER:role/service-role/
APPSYNC_DATASOURCE_ROLE"
      },
      "Action": [
        "es:ESHttpDelete",
        "es:ESHttpHead",
        "es:ESHttpGet",
        "es:ESHttpPost",
        "es:ESHttpPut"
      ],
      "Resource": "arn:aws:es:REGION:ACCOUNTNUMBER:domain/DOMAIN_NAME/*"
    }
  ]
}
```

```

    }
  ]
}

```

連結解析程式

現在資料來源已連接至 OpenSearch Service 網域，您可以使用解析程式將其連接至 GraphQL 結構描述，如下列範例所示：

```

schema {
  query: Query
  mutation: Mutation
}

type Query {
  getPost(id: ID!): Post
  allPosts: [Post]
}

type Mutation {
  addPost(id: ID!, author: String, title: String, url: String, ups: Int, downs: Int,
content: String): AWSJSON
}

type Post {
  id: ID!
  author: String
  title: String
  url: String
  ups: Int
  downs: Int
  content: String
}
...

```

請注意，有使用者定義的 Post 類型與 id 欄位。在下列範例中，我們假設有一個程序（可自動化）可將此類型放入您的 OpenSearch Service 網域，這會對應至路徑根 /post/_doc，其中 post 是索引。在此根路徑中，您可以執行個別文件搜尋、使用 進行萬用字元搜尋 /id/post*，或使用的路徑進行多文件搜尋 /post/_search。例如，如果您有另一種名為 的類型 User，您可以在名為 的新索引下為文件編製索引 user，然後使用路徑 執行搜尋 /user/_search。

從 AWS AppSync 主控台的結構描述編輯器中，修改先前的Posts結構描述以包含searchPosts查詢：

```
type Query {  
  getPost(id: ID!): Post  
  allPosts: [Post]  
  searchPosts: [Post]  
}
```

儲存結構描述。在右側的 searchPosts 中，選擇 Attach resolver (附加解析程式)。在動作功能表中，選擇更新執行時間，然後選擇單位解析程式（僅限 VTL）。然後，選擇您的 OpenSearch Service 資料來源。在 request mapping template (請求映射範本) 區段中，選取 Query posts (查詢文章) 的下拉式清單，來取得基本範本。將 path 修改為 /post/_search。它看起來應該如下所示：

```
{  
  "version": "2017-02-28",  
  "operation": "GET",  
  "path": "/post/_search",  
  "params": {  
    "headers": {},  
    "queryString": {},  
    "body": {  
      "from": 0,  
      "size": 50  
    }  
  }  
}
```

這會假設上述結構描述的文件已在 OpenSearch Service 的 post 欄位下編製索引。如果您建立的資料結構不同，將會需要進行對應的更新。

在回應映射範本區段下，如果您想要從 OpenSearch Service 查詢取得資料結果並轉譯為 GraphQL，則需要指定適當的 _source 篩選條件。使用下列的範本：

```
[  
  #foreach($entry in $context.result.hits.hits)  
  #if( $velocityCount > 1 ) , #end  
  $utils.toJson($entry.get("_source"))  
  #end  
]
```

修改您的搜尋

前述的請求映射範本，會針對所有記錄執行簡單的查詢。假設您想要根據特定的作者來進行搜尋。此外，假設您希望該位作者成為 GraphQL 查詢中所定義的引數。在 AWS AppSync 主控台的結構描述編輯器中，新增 `allPostsByAuthor` 查詢：

```
type Query {
  getPost(id: ID!): Post
  allPosts: [Post]
  allPostsByAuthor(author: String!): [Post]
  searchPosts: [Post]
}
```

現在選擇連接解析程式，然後選取 OpenSearch Service 資料來源，但在回應映射範本中使用下列範例：

```
{
  "version": "2017-02-28",
  "operation": "GET",
  "path": "/post/_search",
  "params": {
    "headers": {},
    "queryString": {},
    "body": {
      "from": 0,
      "size": 50,
      "query": {
        "match": {
          "author": $util.toJson($context.arguments.author)
        }
      }
    }
  }
}
```

請注意，`body` 會填入 `author` 欄位的查詢詞語，此欄位會做為引數，從用戶端直接傳來。您可以選擇性地預先填入資訊 (例如標準的文字內容)，或甚至使用其他[公用程式](#)。

如果您正在使用此解析程式，請使用與先前範例相同的資訊，來填寫回應映射範本。

將資料新增至 OpenSearch Service

由於 GraphQL 變動，您可能想要將資料新增至 OpenSearch Service 網域。這是一個具有搜尋和其他用途的強大機制。由於您可以使用 GraphQL 訂閱[讓資料即時](#)，因此它可做為通知用戶端 OpenSearch Service 網域中資料更新的機制。

返回 AWS AppSync 主控台中的結構描述頁面，並為 `addPost()` 變動選取連接解析程式。再次選取 OpenSearch Service Posts 資料來源，並針對結構描述使用以下回應映射範本：

```
{
  "version": "2017-02-28",
  "operation": "PUT",
  "path": $util.toJson("/post/_doc/$context.arguments.id"),
  "params": {
    "headers": {},
    "queryString": {},
    "body": {
      "id": $util.toJson($context.arguments.id),
      "author": $util.toJson($context.arguments.author),
      "ups": $util.toJson($context.arguments.ups),
      "downs": $util.toJson($context.arguments.downs),
      "url": $util.toJson($context.arguments.url),
      "content": $util.toJson($context.arguments.content),
      "title": $util.toJson($context.arguments.title)
    }
  }
}
```

和之前相同，這是如何建構資料結構的一個範例。如果有不同的欄位名稱或索引，您需要適當地更新 `path` 與 `body`。此範例也示範了如何利用 `$context.arguments`，以 GraphQL 變動引數來填寫範本。

在繼續之前，請使用下列回應映射範本，這會傳回變動操作的結果或錯誤資訊做為輸出：

```
#if($context.error)
  $util.toJson($ctx.error)
#else
  $util.toJson($context.result)
#end
```


擷取單一文件

最後，如果您想要在結構描述中使用 `getPost(id:ID)` 查詢來傳回個別文件，請在 AWS AppSync 主控台的結構描述編輯器中尋找此查詢，然後選擇連接解析程式。再次選取 OpenSearch Service 資料來源，並使用下列映射範本：

```
{
  "version": "2017-02-28",
  "operation": "GET",
  "path": $util.toJson("post/_doc/$context.arguments.id"),
  "params": {
    "headers": {},
    "queryString": {},
    "body": {}
  }
}
```

由於上述的 `path` 使用 `id` 引數與空的內容，因此這會傳回空的單一文件。不過，因為現在傳回的是單一項目而非清單，您需要使用下列回應映射範本：

```
$utils.toJson($context.result.get("_source"))
```

執行查詢與變動

您現在應該能夠對 OpenSearch Service 網域執行 GraphQL 操作。導覽至 AWS AppSync 主控台的查詢索引標籤，並新增記錄：

```
mutation addPost {
  addPost (
    id: "12345"
    author: "Fred"
    title: "My first book"
    content: "This will be fun to write!"
    url: "publisher website",
    ups: 100,
    downs: 20
  )
}
```

您會在右側看到變動的結果。同樣地，您現在可以針對 OpenSearch Service 網域執行 `searchPosts` 查詢：

```
query searchPosts {  
  searchPosts {  
    id  
    title  
    author  
    content  
  }  
}
```

最佳實務

- OpenSearch Service 應該用於查詢資料，而不是做為您的主要資料庫。您可能想要將 OpenSearch Service 與 Amazon DynamoDB 搭配使用，如[合併 GraphQL 解析程式](#)中所述。
- 只有在允許 AWS AppSync 服務角色存取叢集時，才允許 存取您的網域。
- 您可以在開發時少量使用最低成本的叢集，然後在進入正式生產時，轉移到具有高可用性 (HA) 的較大叢集。

使用本機解析程式 in AWS AppSync

Note

我們現在主要支援 APPSYNC_JS 執行期及其文件。請考慮[在此處](#)使用 APPSYNC_JS 執行期及其指南。

AWS AppSync 可讓您使用支援的資料來源 (AWS Lambda、Amazon DynamoDB 或 Amazon OpenSearch Service) 來執行各種操作。不過，在某些情況下，可能不需要呼叫支援的資料來源。

這就是本機解析程式的實用之處。本機解析程式只是將請求映射範本的結果轉發到回應映射範本，而不是呼叫遠端資料來源。欄位解析度不會 leave AWS AppSync。

本機解析程式適用於多種使用案例。最常見的使用案例是發佈通知而不觸發資料來源呼叫。為了示範此使用案例，我們來建置一個分頁應用程式，讓使用者可以互相傳遞頁面。此範例使用訂閱，因此，如果您不熟悉訂閱，可以遵循[即時資料](#)教學課程。

建立分頁應用程式

在我們的分頁應用程式中，用戶端可以訂閱收件匣，並向其他用戶端傳送頁面。每個頁面皆包含訊息。結構描述如下：

```
schema {
  query: Query
  mutation: Mutation
  subscription: Subscription
}

type Subscription {
  inbox(to: String!): Page
  @aws_subscribe(mutations: ["page"])
}

type Mutation {
  page(body: String!, to: String!): Page!
}

type Page {
  from: String
  to: String!
  body: String!
  sentAt: String!
}

type Query {
  me: String
}
```

現在將解析程式附加到 `Mutation.page` 欄位。在 Schema (結構描述) 窗格中，按一下右側窗格上欄位定義旁邊的 `Attach Resolver` (附加解析程式)。建立類型為無的新資料來源，將其命名為 `PageDataSource`。

在請求映射範本部分，輸入：

```
{
  "version": "2017-02-28",
  "payload": {
    "body": $util.toJson($context.arguments.body),
    "from": $util.toJson($context.identity.username),
```

```
    "to": $util.toJson($context.arguments.to),
    "sentAt": "$util.time.nowISO8601()"
  }
}
```

在回應映射範本部分，則選取預設的 Forward the result (轉發結果)。儲存您的解析程式。您的應用程式現已就緒，可以開始傳遞頁面！

傳送和訂閱頁面

用戶端必須先訂閱收件匣，才能接收頁面。

現在就讓我們在 Queries (查詢) 窗格中執行 inbox 訂閱：

```
subscription Inbox {
  inbox(to: "Nadia") {
    body
    to
    from
    sentAt
  }
}
```

每當叫用Mutation.page變動時，Nadia 都會收到頁面。現在透過執行變動來叫用變動：

```
mutation Page {
  page(to: "Nadia", body: "Hello, World!") {
    body
    to
    from
    sentAt
  }
}
```

我們剛示範如何使用本機解析程式，方法是傳送頁面並接收頁面，而不使用 leaving AWS AppSync。

在 中結合 GraphQL 解析程式 AWS AppSync

Note

我們現在主要支援 APPSYNC_JS 執行期及其文件。請考慮[在此處](#)使用 APPSYNC_JS 執行期及其指南。

GraphQL 結構描述中的解析程式與欄位具有一對一的關係，以及高度的彈性。由於資料來源是在解析程式上與結構描述個別獨立設定，因此您可以透過不同的資料來源來解析或操縱 GraphQL 的類型，對結構描述進行混合和匹配，最大程度地滿足您的需求。

下列範例案例示範如何在結構描述中混合和比對資料來源。開始之前，建議您先熟悉設定資料來源和 Amazon DynamoDB 和 Amazon OpenSearch Service 的解析程式 AWS Lambda，如先前教學所述。

範例結構描述

下列結構描述的類型為 Post，已定義 3 個Query操作和 3 個Mutation操作：

```
type Post {
  id: ID!
  author: String!
  title: String
  content: String
  url: String
  ups: Int
  downs: Int
  version: Int!
}

type Query {
  allPost: [Post]
  getPost(id: ID!): Post
  searchPosts: [Post]
}

type Mutation {
  addPost(
    id: ID!,
    author: String!,
    title: String,
    content: String,
```

```

        url: String
      ): Post
    updatePost(
      id: ID!,
      author: String!,
      title: String,
      content: String,
      url: String,
      ups: Int!,
      downs: Int!,
      expectedVersion: Int!
    ): Post
    deletePost(id: ID!): Post
  }

```

在此範例中，總共要連結 6 個解析程式。其中一種可能的方法是讓所有這些都來自稱為的 Amazon DynamoDB 資料表 Posts，其中 AllPosts 會執行掃描並 searchPosts 執行查詢，如 [DynamoDB Resolver 映射範本參考](#) 中所述。不過，還有其他方法可以滿足您的業務需求，例如從 Lambda 或 OpenSearch Service 解析這些 GraphQL 查詢。

透過解析程式來修改資料

您可能需要將資料庫的結果傳回給用戶端，例如 DynamoDB（或 Amazon Aurora），其中某些屬性已變更。這可能是資料類型的格式所造成（例如用戶端上時間戳記的差異），或是為了處理回溯相容性的問題。基於說明目的，在下列範例中，AWS Lambda 函數會在每次叫用 GraphQL 解析程式時指派隨機數字，以操控部落格文章的向上和向下投票：

```

'use strict';
const doc = require('dynamodb-doc');
const dynamo = new doc.DynamoDB();

exports.handler = (event, context, callback) => {
  const payload = {
    TableName: 'Posts',
    Limit: 50,
    Select: 'ALL_ATTRIBUTES',
  };

  dynamo.scan(payload, (err, data) => {
    const result = { data: data.Items.map(item => {
      item.ups = parseInt(Math.random() * (50 - 10) + 10, 10);
      item.downs = parseInt(Math.random() * (20 - 0) + 0, 10);
    }) };
    callback(null, result);
  });
};

```

```
        return item;
    }) };
    callback(err, result.data);
  });
};
```

這是一個非常有效的 Lambda 函式，可以連結至 GraphQL 結構描述中的 `AllPosts` 欄位，如此，任何傳回所有結果的查詢，都會取得支持票和不支持票票數的隨機數量。

DynamoDB 和 OpenSearch Service

對於某些應用程式，您可以對 DynamoDB 執行變動或簡單查詢，並具有背景程序將文件傳輸至 OpenSearch Service。然後，您只需將 `searchPosts Resolver` 連接至 OpenSearch Service 資料來源，並使用 GraphQL 查詢傳回搜尋結果（來自源自 DynamoDB 的資料）。在為您的應用程式加入進階的搜尋操作時（例如關鍵字詞、模糊字詞比對或甚至地理空間查詢），這可以是非常強大的功能。可以透過 ETL 程序從 DynamoDB 傳輸資料，或者您也可以使用 Lambda 從 DynamoDB 串流。您可以使用您 AWS 帳戶中美國西部 2（奧勒岡）區域的下列 AWS CloudFormation 堆疊來啟動完整的範例：

Launch Stack 

此範例中的結構描述可讓您使用 DynamoDB 解析程式新增文章，如下所示：

```
mutation add {
  putPost(author:"Nadia"
    title:"My first post"
    content:"This is some test content"
    url:"https://aws.amazon.com/appsync/")
  {
    id
    title
  }
}
```

這會將資料寫入 DynamoDB，然後透過 Lambda 將資料串流至 Amazon OpenSearch Service，您可以依不同的欄位搜尋所有文章。例如，由於資料位於 Amazon OpenSearch Service 中，因此您可以使用自由格式文字搜尋作者或內容欄位，即使是空格，如下所示：

```
query searchName{
  searchAuthor(name:"  Nadia  "){
    id
  }
}
```

```
        title
        content
      }
    }
  }

query searchContent{
  searchContent(text:"test"){
    id
    title
    content
  }
}
```

由於資料是直接寫入 DynamoDB，因此您仍然可以使用 `allPosts{...}` 和 `singlePost{...}` 查詢，針對資料表執行有效的清單或項目查詢操作。此堆疊會針對 DynamoDB 串流使用下列範例程式碼：

注意：這些程式碼僅為示範。

```
var AWS = require('aws-sdk');
var path = require('path');
var stream = require('stream');

var esDomain = {
  endpoint: 'https://opensearch-domain-name.REGION.es.amazonaws.com',
  region: 'REGION',
  index: 'id',
  doctype: 'post'
};

var endpoint = new AWS.Endpoint(esDomain.endpoint)
var creds = new AWS.EnvironmentCredentials('AWS');

function postDocumentToES(doc, context) {
  var req = new AWS.HttpRequest(endpoint);

  req.method = 'POST';
  req.path = '/_bulk';
  req.region = esDomain.region;
  req.body = doc;
  req.headers['presigned-expires'] = false;
  req.headers['Host'] = endpoint.host;
```



```
// Sign the request (Sigv4)
var signer = new AWS.Signers.V4(req, 'es');
signer.addAuthorization(creds, new Date());

// Post document to ES
var send = new AWS.NodeHttpClient();
send.handleRequest(req, null, function (httpResp) {
  var body = '';
  httpResp.on('data', function (chunk) {
    body += chunk;
  });
  httpResp.on('end', function (chunk) {
    console.log('Successful', body);
    context.succeed();
  });
}, function (err) {
  console.log('Error: ' + err);
  context.fail();
});
}

exports.handler = (event, context, callback) => {
  console.log("event => " + JSON.stringify(event));
  var posts = '';

  for (var i = 0; i < event.Records.length; i++) {
    var eventName = event.Records[i].eventName;
    var actionType = '';
    var image;
    var noDoc = false;
    switch (eventName) {
      case 'INSERT':
        actionType = 'create';
        image = event.Records[i].dynamodb.NewImage;
        break;
      case 'MODIFY':
        actionType = 'update';
        image = event.Records[i].dynamodb.NewImage;
        break;
      case 'REMOVE':
        actionType = 'delete';
        image = event.Records[i].dynamodb.OldImage;
        noDoc = true;
        break;
    }
  }
}
```

```

    }

    if (typeof image !== "undefined") {
      var postData = {};
      for (var key in image) {
        if (image.hasOwnProperty(key)) {
          if (key === 'postId') {
            postData['id'] = image[key].S;
          } else {
            var val = image[key];
            if (val.hasOwnProperty('S')) {
              postData[key] = val.S;
            } else if (val.hasOwnProperty('N')) {
              postData[key] = val.N;
            }
          }
        }
      }
    }

    var action = {};
    action[actionType] = {};
    action[actionType]._index = 'id';
    action[actionType]._type = 'post';
    action[actionType]._id = postData['id'];
    posts += [
      JSON.stringify(action),
    ].concat(noDoc?[]:[JSON.stringify(postData)]).join('\n') + '\n';
  }
}
console.log('posts:',posts);
postDocumentToES(posts, context);
};

```

然後，您可以使用 DynamoDB 串流將此項目連接至主索引鍵為的 DynamoDB 資料表id，而 DynamoDB 來源的任何變更都會串流到您的 OpenSearch Service 網域。如需進行這項設定的詳細資訊，請參閱 [DynamoDB 串流文件](#)。

在 AWS AppSync 中使用 DynamoDB 批次操作

Note

我們現在主要支援 APPSYNC_JS 執行期及其文件。請考慮[在此處](#)使用 APPSYNC_JS 執行期及其指南。

AWS AppSync 支援在單一區域中的一或多個資料表中使用 Amazon DynamoDB 批次操作。支援的操作包括 BatchGetItem、BatchPutItem 和 BatchDeleteItem。透過在 AWS AppSync 中使用這些功能，您可以執行下列任務：

- 在單次查詢中傳遞索引鍵清單，並從資料表傳回結果
- 在單次查詢中讀取一個或多個資料表的記錄
- 將大量的記錄寫入一個或多個資料表
- 在可能具有關聯的多個資料表中，有條件地寫入或刪除記錄

搭配 DynamoDB in AWS AppSync 使用批次操作是一項進階技術，需要多加思考和了解您的後端操作和資料表結構。此外，批次操作 in AWS AppSync 與非批次操作有兩個主要差異：

- 對於解析程式將會存取的所有資料表，資料來源角色都必須具有權限。
- 解析程式資料表規格包含於映射範本中。

許可

與其他解析程式一樣，您需要建立資料來源 in AWS AppSync，並建立角色或使用現有的角色。由於批次操作在 DynamoDB 資料表上需要不同的許可，因此您需要為讀取或寫入動作授予設定的角色許可：

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "dynamodb:BatchGetItem",
        "dynamodb:BatchWriteItem"
      ]
    }
  ]
}
```

```

    ],
    "Effect": "Allow",
    "Resource": [
        "arn:aws:dynamodb:region:account:table/TABLENAME",
        "arn:aws:dynamodb:region:account:table/TABLENAME/*"
    ]
}
]
}

```

注意：角色會繫結至 in AWS AppSync 的資料來源，而 欄位上的解析程式會針對資料來源叫用。設定為針對 DynamoDB 擷取的資料來源只會指定一個資料表，以保持簡單的組態。因此，在單一解析程式中針對多個資料表執行批次作業時 (這是一項更為進階的工作)，您必須授予權限給該資料來源上的角色，以存取解析程式將會與其互動的所有資料表。這會在上述 IAM 政策的 Resource (資源) 欄位中完成。針對批次操作所要呼叫的資料表進行設定，這項動作是在解析程式範本中完成，我們將會在下列內容中說明此範本。

資料來源

為簡單起見，我們將針對本教學課程中使用的所有解析程式，使用相同的資料來源。在資料來源索引標籤上，建立新的 DynamoDB 資料來源，並將其命名為 BatchTutorial。資料表可以使用任何名稱，因為資料表名稱在批次作業中是指定為請求映射範本的一部分。我們會將資料表命名為 empty。

在本教學課程中，具備下列內嵌政策的任何角色皆可使用：

JSON

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "dynamodb:BatchGetItem",
        "dynamodb:BatchWriteItem"
      ],
      "Effect": "Allow",
      "Resource": [
        "arn:aws:dynamodb:region:account:table/Posts",
        "arn:aws:dynamodb:region:account:table/Posts/*",
        "arn:aws:dynamodb:region:account:table/locationReadings",

```

```

        "arn:aws:dynamodb:region:account:table/locationReadings/*",
        "arn:aws:dynamodb:region:account:table/temperatureReadings",
        "arn:aws:dynamodb:region:account:table/temperatureReadings/*"
    ]
}
}
}

```

單一資料表批次

Warning

BatchPutItem 與衝突偵測和解決搭配使用時，不支援 和 BatchDeleteItem。必須停用這些設定，以防止可能的錯誤。

例如，假設有一個名為 Posts 的資料表，您想要利用批次操作來新增和移除此資料表的項目。使用下列的結構描述，請注意，我們會將 ID 清單傳入查詢：

```

type Post {
  id: ID!
  title: String
}

input PostInput {
  id: ID!
  title: String
}

type Query {
  batchGet(ids: [ID]): [Post]
}

type Mutation {
  batchAdd(posts: [PostInput]): [Post]
  batchDelete(ids: [ID]): [Post]
}

schema {
  query: Query
  mutation: Mutation
}

```

```
}
```

使用下列請求映射範本將解析程式連接至 `batchAdd()` 欄位。這會自動擷取 GraphQL `input PostInput` 類型的每個項目，並建置 `BatchPutItem` 作業所需的對應圖：

```
#set($postsdata = [])
#foreach($item in ${ctx.args.posts})
    $util.qr($postsdata.add($util.dynamodb.toMapValues($item)))
#end

{
  "version" : "2018-05-29",
  "operation" : "BatchPutItem",
  "tables" : {
    "Posts": $utils.toJson($postsdata)
  }
}
```

在此情況中的回應映射範本只是簡單的直接傳遞，但是該資料表名稱會以 `..data.Posts` 的形式附加至內容物件，如下所示：

```
$util.toJson($ctx.result.data.Posts)
```

現在導覽至 AWS AppSync 主控台的查詢頁面，並執行下列 `batchAdd` 變動：

```
mutation add {
  batchAdd(posts:[{
    id: 1 title: "Running in the Park",{
    id: 2 title: "Playing fetch"
  }]){
    id
    title
  }
}
```

您應該會看到列印到螢幕的結果，並且可以透過 DynamoDB 主控台獨立驗證這兩個值都寫入貼文資料表。

接著，使用下列請求映射範本將解析程式連接至 `batchGet()` 欄位。這會自動擷取 GraphQL `ids:[]` 類型的每個項目，並建置 `BatchGetItem` 作業所需的對應圖：

```
#set($ids = [])
#foreach($id in ${ctx.args.ids})
    #set($map = {})
    $util.qr($map.put("id", $util.dynamodb.toString($id)))
    $util.qr($ids.add($map))
#end

{
  "version" : "2018-05-29",
  "operation" : "BatchGetItem",
  "tables" : {
    "Posts": {
      "keys": $util.toJson($ids),
      "consistentRead": true,
      "projection" : {
        "expression" : "#id, title",
        "expressionNames" : { "#id" : "id"}
      }
    }
  }
}
```

回應映射範本仍然只是簡單的直接傳遞，該資料表名稱也會再次地以 `..data.Posts` 的形式附加至內容物件：

```
$util.toJson($ctx.result.data.Posts)
```

現在返回 AWS AppSync 主控台的查詢頁面，並執行下列 `batchGet` 查詢：

```
query get {
  batchGet(ids:[1,2,3]){
    id
    title
  }
}
```

這應該會針對您先前所新增的兩個 `id` 值，傳回其結果。請注意，值為 `null` 的 `id` 傳回了 3 值。這是因為在 `Posts` 資料表中，尚未有任何記錄具有該值。另請注意，AWS AppSync 會以與傳入查詢的金鑰相同的順序傳回結果，這是 AWS AppSync 代表您執行的額外功能。因此，如果切換到 `batchGet(ids:[1,3,2])`，順序將會變更。您也會知道哪個 `id` 傳回 `null` 值。

最後，使用以下請求映射範本將解析程式連接到 `batchDelete()` 欄位。這會自動擷取 GraphQL `ids:[]` 類型的每個項目，並建置 `BatchGetItem` 作業所需的對應圖：

```
#set($ids = [])
#foreach($id in ${ctx.args.ids})
    #set($map = {})
    $util.qr($map.put("id", $util.dynamodb.toString($id)))
    $util.qr($ids.add($map))
#end

{
  "version" : "2018-05-29",
  "operation" : "BatchDeleteItem",
  "tables" : {
    "Posts": $util.toJson($ids)
  }
}
```

回應映射範本仍然只是簡單的直接傳遞，該資料表名稱也會再次地以 `..data.Posts` 的形式附加至內容物件：

```
$util.toJson($ctx.result.data.Posts)
```

現在返回 AWS AppSync 主控台的查詢頁面，並執行下列 `batchDelete` 變動：

```
mutation delete {
  batchDelete(ids:[1,2]){ id }
}
```

`id 1` 和 `2` 的記錄現在應已刪除。如果重新執行先前的 `batchGet()` 查詢，這些應該會傳回 `null`。

多重資料表批次

Warning

`BatchPutItem` 與衝突偵測和解決搭配使用時，不支援 `BatchDeleteItem`。必須停用這些設定，以防止可能的錯誤。

AWS AppSync 也可讓您跨資料表執行批次操作。來試試建置更複雜的應用程式。試想建置一個寵物健康狀態 (Pet Health) 應用程式，其中會有感應器呈報寵物的位置和體溫。感應器是由電池供電，而且每隔幾分鐘就會試著連線到網路。當感應器建立連線時，它會將其讀數傳送至 our AWS AppSync API。觸發條件接著就會分析資料，然後將儀表板呈現給寵物的主人。讓我們著重介紹感應器與後端資料存放區之間的互動。

作為先決條件，讓我們先建立兩個 DynamoDB 資料表；locationReadings 會儲存感應器位置讀數，而 temperatureReadings 會儲存感應器溫度讀數。這兩個資料表碰巧擁有相同的主索引鍵結構：sensorId (String) 為分割區索引鍵、timestamp (String) 為排序索引鍵。

讓我們使用下列 GraphQL 結構描述：

```
type Mutation {
  # Register a batch of readings
  recordReadings(tempReadings: [TemperatureReadingInput], locReadings:
[LocationReadingInput]): RecordResult
  # Delete a batch of readings
  deleteReadings(tempReadings: [TemperatureReadingInput], locReadings:
[LocationReadingInput]): RecordResult
}

type Query {
  # Retrieve all possible readings recorded by a sensor at a specific time
  getReadings(sensorId: ID!, timestamp: String!): [SensorReading]
}

type RecordResult {
  temperatureReadings: [TemperatureReading]
  locationReadings: [LocationReading]
}

interface SensorReading {
  sensorId: ID!
  timestamp: String!
}

# Sensor reading representing the sensor temperature (in Fahrenheit)
type TemperatureReading implements SensorReading {
  sensorId: ID!
  timestamp: String!
  value: Float
}
```

```
# Sensor reading representing the sensor location (lat,long)
type LocationReading implements SensorReading {
  sensorId: ID!
  timestamp: String!
  lat: Float
  long: Float
}

input TemperatureReadingInput {
  sensorId: ID!
  timestamp: String
  value: Float
}

input LocationReadingInput {
  sensorId: ID!
  timestamp: String
  lat: Float
  long: Float
}
```

BatchPutItem - 記錄感應器讀數

感應器必須能夠在連線到網際網路之後傳送其讀數。感應器將使用 GraphQL 欄位 `Mutation.recordReadings` 這個 API 來執行此項動作。讓我們連結解析程式，讓 API 能夠開始運作。

選取 `Mutation.recordReadings` 欄位旁的連接。在下一個畫面中，選取在教學課程開始時所建立的同一個 `BatchTutorial` 資料來源。

讓我們來新增下列請求映射範本：

請求映射範本

```
## Convert tempReadings arguments to DynamoDB objects
#set($tempReadings = [])
#foreach($reading in ${ctx.args.tempReadings})
  $util.qr($tempReadings.add($util.dynamodb.toMapValues($reading)))
#end

## Convert locReadings arguments to DynamoDB objects
#set($locReadings = [])
#foreach($reading in ${ctx.args.locReadings})
```

```

    $util.qr($locReadings.add($util.dynamodb.toMapValues($reading)))
#end

{
  "version" : "2018-05-29",
  "operation" : "BatchPutItem",
  "tables" : {
    "locationReadings": $utils.toJson($locReadings),
    "temperatureReadings": $utils.toJson($tempReadings)
  }
}

```

如您所見，BatchPutItem 操作可讓我們指定多個資料表。

讓我們使用下列回應映射範本。

回應映射範本

```

## If there was an error with the invocation
## there might have been partial results
#if($ctx.error)
  ## Append a GraphQL error for that field in the GraphQL response
  $utils.appendError($ctx.error.message, $ctx.error.message)
#end
## Also returns data for the field in the GraphQL response
$utils.toJson($ctx.result.data)

```

進行批次操作時，呼叫可能會同時傳回錯誤和結果。在這種情況中，我們可以隨意進行一些額外的錯誤處理。

注意：\$utils.appendError() 的使用類似於 \$util.error()，主要差別在於前者不會中斷映射範本的評估，而是在欄位出現錯誤時發出訊號，但允許範本的評估，進而將資料傳回給呼叫程式。如果應用程式需要傳回部分結果，建議您使用 \$utils.appendError()。

儲存解析程式並導覽至 AWS AppSync 主控台的查詢頁面。讓我們開始傳送一些感應器讀數吧！

執行下列的變動：

```

mutation sendReadings {
  recordReadings(
    tempReadings: [
      {sensorId: 1, value: 85.5, timestamp: "2018-02-01T17:21:05.000+08:00"},

```

```

    {sensorId: 1, value: 85.7, timestamp: "2018-02-01T17:21:06.000+08:00"},
    {sensorId: 1, value: 85.8, timestamp: "2018-02-01T17:21:07.000+08:00"},
    {sensorId: 1, value: 84.2, timestamp: "2018-02-01T17:21:08.000+08:00"},
    {sensorId: 1, value: 81.5, timestamp: "2018-02-01T17:21:09.000+08:00"}
  ]
  locReadings: [
    {sensorId: 1, lat: 47.615063, long: -122.333551, timestamp:
"2018-02-01T17:21:05.000+08:00"},
    {sensorId: 1, lat: 47.615163, long: -122.333552, timestamp:
"2018-02-01T17:21:06.000+08:00"}
    {sensorId: 1, lat: 47.615263, long: -122.333553, timestamp:
"2018-02-01T17:21:07.000+08:00"}
    {sensorId: 1, lat: 47.615363, long: -122.333554, timestamp:
"2018-02-01T17:21:08.000+08:00"}
    {sensorId: 1, lat: 47.615463, long: -122.333555, timestamp:
"2018-02-01T17:21:09.000+08:00"}
  ]) {
    locationReadings {
      sensorId
      timestamp
      lat
      long
    }
    temperatureReadings {
      sensorId
      timestamp
      value
    }
  }
}
```

我們會在一次的變動中傳送 10 個感應器讀數，讀數的資料會分置於兩個資料表。使用 DynamoDB 主控台來驗證資料是否同時顯示在 locationReadings 和 temperatureReadings 資料表中。

BatchDeleteItem - 刪除感應器讀數

同樣地，我們也需要刪除分批的感應器讀數。讓我們使用 Mutation.deleteReadings GraphQL 欄位來進行這項動作。選取 Mutation.recordReadings 欄位旁的連接。在下一個畫面中，選取在教學課程開始時所建立的同一個 BatchTutorial 資料來源。

讓我們使用下列請求映射範本。

請求映射範本

```

## Convert tempReadings arguments to DynamoDB primary keys
#set($tempReadings = [])
#foreach($reading in ${ctx.args.tempReadings})
    #set($pkey = {})
    $util.qr($pkey.put("sensorId", $reading.sensorId))
    $util.qr($pkey.put("timestamp", $reading.timestamp))
    $util.qr($tempReadings.add($util.dynamodb.toMapValues($pkey)))
#end

## Convert locReadings arguments to DynamoDB primary keys
#set($locReadings = [])
#foreach($reading in ${ctx.args.locReadings})
    #set($pkey = {})
    $util.qr($pkey.put("sensorId", $reading.sensorId))
    $util.qr($pkey.put("timestamp", $reading.timestamp))
    $util.qr($locReadings.add($util.dynamodb.toMapValues($pkey)))
#end

{
    "version" : "2018-05-29",
    "operation" : "BatchDeleteItem",
    "tables" : {
        "locationReadings": $utils.toJson($locReadings),
        "temperatureReadings": $utils.toJson($tempReadings)
    }
}

```

這個回應映射範本和我們用於 `Mutation.recordReadings` 的相同。

回應映射範本

```

## If there was an error with the invocation
## there might have been partial results
#if($ctx.error)
    ## Append a GraphQL error for that field in the GraphQL response
    $utils.appendError($ctx.error.message, $ctx.error.message)
#end
## Also return data for the field in the GraphQL response
$utils.toJson($ctx.result.data)

```

儲存解析程式並導覽至 AWS AppSync 主控台的查詢頁面。現在，讓我們刪除一些感應器讀數吧！

執行下列的變動：

```
mutation deleteReadings {
  # Let's delete the first two readings we recorded
  deleteReadings(
    tempReadings: [{sensorId: 1, timestamp: "2018-02-01T17:21:05.000+08:00"}]
    locReadings: [{sensorId: 1, timestamp: "2018-02-01T17:21:05.000+08:00"}]) {
    locationReadings {
      sensorId
      timestamp
      lat
      long
    }
    temperatureReadings {
      sensorId
      timestamp
      value
    }
  }
}
```

透過 DynamoDB 主控台驗證這兩個讀數已從 locationReadings 和 temperatureReadings 資料表中刪除。

BatchGetItem - 擷取讀數

我們寵物健康狀態 (Pet Health) 應用程式另一個常用的操作，就是在特定的時間點擷取感應器的讀數。試試將解析程式連接到結構描述中的 Query.getReadings GraphQL 欄位。選取 Attach (附加)，然後在下一個畫面上，選取在教學課程開始時所建立的同一個 BatchTutorial 資料來源。

讓我們來新增下列請求映射範本。

請求映射範本

```
## Build a single DynamoDB primary key,
## as both locationReadings and tempReadings tables
## share the same primary key structure
#set($pkey = {})
$util.qr($pkey.put("sensorId", $ctx.args.sensorId))
$util.qr($pkey.put("timestamp", $ctx.args.timestamp))

{
  "version" : "2018-05-29",
  "operation" : "BatchGetItem",
```

```

    "tables" : {
      "locationReadings": {
        "keys": [$util.dynamodb.toMapValuesJson($pkey)],
        "consistentRead": true
      },
      "temperatureReadings": {
        "keys": [$util.dynamodb.toMapValuesJson($pkey)],
        "consistentRead": true
      }
    }
  }
}

```

請注意，我們現在使用的是 BatchGetItem 操作。

我們的回應映射範本將會有些微的不同，因為我們選擇傳回 SensorReading 清單。讓我們將叫用結果映射到所需的結構。

回應映射範本

```

## Merge locationReadings and temperatureReadings
## into a single list
## __typename needed as schema uses an interface
#set($sensorReadings = [])

#foreach($locReading in $ctx.result.data.locationReadings)
  $util.qr($locReading.put("__typename", "LocationReading"))
  $util.qr($sensorReadings.add($locReading))
#end

#foreach($tempReading in $ctx.result.data.temperatureReadings)
  $util.qr($tempReading.put("__typename", "TemperatureReading"))
  $util.qr($sensorReadings.add($tempReading))
#end

$util.toJson($sensorReadings)

```

儲存解析程式並導覽至 AWS AppSync 主控台的查詢頁面。現在，讓我們擷取感應器讀數吧！

執行下列的查詢：

```

query getReadingsForSensorAndTime {
  # Let's retrieve the very first two readings
  getReadings(sensorId: 1, timestamp: "2018-02-01T17:21:06.000+08:00") {

```

```
    sensorId
    timestamp
    ...on TemperatureReading {
      value
    }
    ...on LocationReading {
      lat
      long
    }
  }
}
```

我們已成功示範如何使用 AWS AppSync 進行 DynamoDB 批次操作。

錯誤處理

In AWS AppSync 中，資料來源操作有時可能會傳回部分結果。我們將會使用部分結果這個詞彙，來表示操作的輸出包含一些資料和錯誤的情況。由於錯誤處理本質上是應用程式特定的，AWS AppSync 讓您有機會處理回應映射範本中的錯誤。如果發生解析程式呼叫錯誤，在文字內容中會出現 `$ctx.error`。呼叫錯誤一律包含訊息和類型，可做為 `$ctx.error.message` 和 `$ctx.error.type` 屬性存取。在回應映射範本呼叫期間，您可以用三種方式來處理部分結果：

1. 藉由只傳回資料來抑制呼叫錯誤
2. 藉由停止進行回應映射範本的評估 (這不會傳回任何資料)，來產生錯誤 (使用 `$util.error(...)`)。
3. 附加錯誤 (使用 `$util.appendError(...)`)，而且也傳回資料

讓我們透過 DynamoDB 的批次操作來示範上述三點！

DynamoDB 批次操作

使用 DynamoDB 批次操作時，批次作業就有可能部分完成。也就是說，請求的項目或索引鍵可以有一些尚未處理完成。If AWS AppSync 無法完成批次，未處理的項目將在內容上設定調用錯誤。

我們將會使用 `Query.getReadings` 欄位組態 (取自於本教學課程先前區段所說明的 `BatchGetItem` 操作) 來建置錯誤處理功能。這次，讓我們假設在執行 `Query.getReadings` 欄位時，`temperatureReadings` DynamoDB 資料表用盡了佈建的輸送量。DynamoDB 在第二次嘗試 by AWS AppSync 時引發 `ProvisionedThroughputExceededException`，以處理批次中剩餘的元素。

下列的 JSON 呈現了在 DynamoDB 批次呼叫之後、回應映射範本評估之前的序列化內容。


```
{
  "arguments": {
    "sensorId": "1",
    "timestamp": "2018-02-01T17:21:05.000+08:00"
  },
  "source": null,
  "result": {
    "data": {
      "temperatureReadings": [
        null
      ],
      "locationReadings": [
        {
          "lat": 47.615063,
          "long": -122.333551,
          "sensorId": "1",
          "timestamp": "2018-02-01T17:21:05.000+08:00"
        }
      ]
    },
    "unprocessedKeys": {
      "temperatureReadings": [
        {
          "sensorId": "1",
          "timestamp": "2018-02-01T17:21:05.000+08:00"
        }
      ],
      "locationReadings": []
    }
  },
  "error": {
    "type": "DynamoDB:ProvisionedThroughputExceededException",
    "message": "You exceeded your maximum allowed provisioned throughput for a table or for one or more global secondary indexes. (...)"
  },
  "outErrors": []
}
```

對於內容的幾個注意事項：

- 調用錯誤已在 `$ctx.error` by AWS AppSync 的內容上設定，錯誤類型已設定為 `DynamoDB:ProvisionedThroughputExceededException`。

- 即使出現錯誤，也會針對每個資料表對應結果 (\$ctx.result.data)
- 未處理的索引鍵可透過 \$ctx.result.data.unprocessedKeys 取得。在這裡，AWS AppSync 無法使用金鑰 (sensorId : 1、時間戳記 : 2018-02-01T17:21:05.000+08:00) 擷取項目，因為資料表輸送量不足。

注意：對於 BatchPutItem，則為 \$ctx.result.data.unprocessedItems。如果是 BatchDeleteItem，則為 \$ctx.result.data.unprocessedKeys。

讓我們用三種不同的方式來處理這項錯誤。

1. 抑制呼叫錯誤

傳回資料而不處理呼叫錯誤，可有效地抑制錯誤，讓指定 GraphQL 欄位的結果一律成功。

我們所編寫的回應映射範本，只熟悉和著重於結果的資料。

回應映射範本：

```
$util.toJson($ctx.result.data)
```

GraphQL 回應：

```
{
  "data": {
    "getReadings": [
      {
        "sensorId": "1",
        "timestamp": "2018-02-01T17:21:05.000+08:00",
        "lat": 47.615063,
        "long": -122.333551
      },
      {
        "sensorId": "1",
        "timestamp": "2018-02-01T17:21:05.000+08:00",
        "value": 85.5
      }
    ]
  }
}
```

錯誤回應中不會加入任何錯誤，因為只處理了資料。

2. 丟出錯誤來中止範本的執行

如果從用戶端的角度，部分故障應視為完全故障，則您可以中止範本的執行，來防止傳回資料。`$util.error(...)` 公用程式方法可確實完成這項動作。

回應映射範本：

```
## there was an error let's mark the entire field
## as failed and do not return any data back in the response
#if ($ctx.error)
    $util.error($ctx.error.message, $ctx.error.type, null,
    $ctx.result.data.unprocessedKeys)
#end

$util.toJson($ctx.result.data)
```

GraphQL 回應：

```
{
  "data": {
    "getReadings": null
  },
  "errors": [
    {
      "path": [
        "getReadings"
      ],
      "data": null,
      "errorType": "DynamoDB:ProvisionedThroughputExceededException",
      "errorInfo": {
        "temperatureReadings": [
          {
            "sensorId": "1",
            "timestamp": "2018-02-01T17:21:05.000+08:00"
          }
        ],
        "locationReadings": []
      },
      "locations": [
        {
          "line": 58,
          "column": 3
        }
      ]
    }
  ]
}
```

```
    ],
    "message": "You exceeded your maximum allowed provisioned throughput for a table
or for one or more global secondary indexes. (...)"
  }
]
}
```

即使 DynamoDB 批次作業可能已經傳回一些結果，我們還是選擇丟出錯誤，如此 `getReadings` GraphQL 欄位為 `null`，而錯誤也會加入到 GraphQL 回應的 `errors` (錯誤) 區塊。

3. 附加錯誤，以同時傳回資料和錯誤

在某些情況中，為提供更好的使用者體驗，應用程式可以傳回部分結果，並通知其用戶端有未處理的項目。用戶端可以決定是否要重試，或是轉譯錯誤後傳回給最終使用者。`$util.appendError(...)` 是一項公用程式方法，可讓應用程式的設計人員將錯誤附加於文字內容，但不會干擾到範本的評估，進而完成前述的動作。評估範本之後，AWS AppSync 會將任何內容錯誤附加至 GraphQL 回應的錯誤區塊，以處理這些錯誤。

回應映射範本：

```
#if ($ctx.error)
  ## pass the unprocessed keys back to the caller via the `errorInfo` field
  $util.appendError($ctx.error.message, $ctx.error.type, null,
    $ctx.result.data.unprocessedKeys)
#end

$util.toJson($ctx.result.data)
```

我們會同時轉傳呼叫錯誤，以及 GraphQL 回應錯誤區塊中的 `unprocessedKeys` (未處理的索引鍵) 項目。`getReadings` 欄位也會傳回來自 `locationReadings` 資料表的部分資料，如下列的回應中所示。

GraphQL 回應：

```
{
  "data": {
    "getReadings": [
      null,
      {
        "sensorId": "1",
        "timestamp": "2018-02-01T17:21:05.000+08:00",
        "value": 85.5
      }
    ]
  }
}
```

```
]
},
"errors": [
  {
    "path": [
      "getReadings"
    ],
    "data": null,
    "errorType": "DynamoDB:ProvisionedThroughputExceededException",
    "errorInfo": {
      "temperatureReadings": [
        {
          "sensorId": "1",
          "timestamp": "2018-02-01T17:21:05.000+08:00"
        }
      ],
      "locationReadings": []
    },
    "locations": [
      {
        "line": 58,
        "column": 3
      }
    ],
    "message": "You exceeded your maximum allowed provisioned throughput for a table or for one or more global secondary indexes. (...)"
  }
]
}
```

在 AWS AppSync 中執行 DynamoDB 交易

Note

我們現在主要支援 APPSYNC_JS 執行期及其文件。請考慮[在此處](#)使用 APPSYNC_JS 執行期及其指南。

AWS AppSync 支援在單一區域中的一或多個資料表中使用 Amazon DynamoDB 交易操作。支援的操作包括 TransactGetItems 和 TransactWriteItems。透過使用這些 in AWS AppSync 功能，您可以執行下列任務：

- 在單次查詢中傳遞索引鍵清單，並從資料表傳回結果
- 在單次查詢中讀取一個或多個資料表的記錄
- 以「全有或全無」的方式，將交易中的記錄寫入一個或多個資料表
- 符合某些條件時執行交易

許可

如同其他解析程式，您需要建立資料來源 in AWS AppSync，並建立角色或使用現有的角色。由於交易操作在 DynamoDB 資料表上需要不同的許可，因此您需要授予已設定的角色讀取或寫入動作的許可：

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "dynamodb:DeleteItem",
        "dynamodb:GetItem",
        "dynamodb:PutItem",
        "dynamodb:Query",
        "dynamodb:Scan",
        "dynamodb:UpdateItem"
      ],
      "Effect": "Allow",
      "Resource": [
        "arn:aws:dynamodb:region:accountId:table/TABLENAME",
        "arn:aws:dynamodb:region:accountId:table/TABLENAME/*"
      ]
    }
  ]
}
```

注意：角色會繫結至 in AWS AppSync 的資料來源，而 欄位上的解析程式會針對資料來源叫用。設定為針對 DynamoDB 擷取的資料來源只會指定一個資料表，以保持簡單的組態。因此，在單一解析程式中針對多個資料表執行交易操作時（這是一項更為進階的工作），您必須授予權限給該資料來源上的角色，以存取解析程式將會與其互動的所有資料表。這會在上述 IAM 政策的 Resource（資源）欄位中完成。針對資料表設定交易呼叫是在解析程式範本中完成，我們將會在下列內容中說明此範本。

資料來源

為簡單起見，我們將針對本教學課程中使用的所有解析程式，使用相同的資料來源。在資料來源索引標籤上，建立新的 DynamoDB 資料來源，並將其命名為 TransactTutorial。資料表可以使用任何名稱，因為資料表名稱在交易操作中是指定為請求映射範本的一部分。我們會將資料表命名為 empty。

我們將兩個資料表稱為 savingAccounts 和 checkingAccounts，兩者皆具有 accountNumber 做為分割區索引鍵，以及 transactionHistory 資料表，具有 transactionId 做為分割區索引鍵。

在本教學課程中，具備下列內嵌政策的任何角色皆可使用。將 region 和 accountId 更換為您的區域和帳戶 ID：

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "dynamodb:DeleteItem",
        "dynamodb:GetItem",
        "dynamodb:PutItem",
        "dynamodb:Query",
        "dynamodb:Scan",
        "dynamodb:UpdateItem"
      ],
      "Effect": "Allow",
      "Resource": [
        "arn:aws:dynamodb:region:accountId:table/savingAccounts",
        "arn:aws:dynamodb:region:accountId:table/savingAccounts/*",
        "arn:aws:dynamodb:region:accountId:table/checkingAccounts",
        "arn:aws:dynamodb:region:accountId:table/checkingAccounts/*",
        "arn:aws:dynamodb:region:accountId:table/transactionHistory",
        "arn:aws:dynamodb:region:accountId:table/transactionHistory/*"
      ]
    }
  ]
}
```

交易

在此範例中，內容是傳統的銀行交易，我們將使用 `TransactWriteItems` 以進行下列操作：

- 從存款帳戶轉帳至支票帳戶
- 為每筆交易產生新的交易記錄

然後，我們會使用 `TransactGetItems` 擷取存款帳戶和支票帳戶中的詳細資料。

Warning

`TransactWriteItems` 與衝突偵測和解決搭配使用時，不支援。必須停用這些設定，以防止可能的錯誤。

定義 GraphQL 結構描述，如下所示：

```
type SavingAccount {
  accountNumber: String!
  username: String
  balance: Float
}

type CheckingAccount {
  accountNumber: String!
  username: String
  balance: Float
}

type TransactionHistory {
  transactionId: ID!
  from: String
  to: String
  amount: Float
}

type TransactionResult {
  savingAccounts: [SavingAccount]
  checkingAccounts: [CheckingAccount]
  transactionHistory: [TransactionHistory]
}
```



```
input SavingAccountInput {
  accountNumber: String!
  username: String
  balance: Float
}

input CheckingAccountInput {
  accountNumber: String!
  username: String
  balance: Float
}

input TransactionInput {
  savingAccountNumber: String!
  checkingAccountNumber: String!
  amount: Float!
}

type Query {
  getAccounts(savingAccountNumbers: [String], checkingAccountNumbers: [String]):
  TransactionResult
}

type Mutation {
  populateAccounts(savingAccounts: [SavingAccountInput], checkingAccounts:
  [CheckingAccountInput]): TransactionResult
  transferMoney(transactions: [TransactionInput]): TransactionResult
}

schema {
  query: Query
  mutation: Mutation
}
```

TransactWriteItems - 填入帳戶

為了在帳戶之間轉帳，我們需要在表格中填入詳細資料。我們將使用 GraphQL 操作 `Mutation.populateAccounts` 來執行此作業。

在結構描述區段中，按一下 `Mutation.populateAccounts` 操作旁的連接。前往 VTL 單位解析程式，然後選擇相同的 `TransactTutorial` 資料來源。

現在使用下列請求映射範本：

請求映射範本

```
#set($savingAccountTransactPutItems = [])
#set($index = 0)
#foreach($savingAccount in ${ctx.args.savingAccounts})
    #set($keyMap = {})
    $util.qr($keyMap.put("accountNumber",
$util.dynamodb.toString($savingAccount.accountNumber)))
    #set($attributeValues = {})
    $util.qr($attributeValues.put("username",
$util.dynamodb.toString($savingAccount.username)))
    $util.qr($attributeValues.put("balance",
$util.dynamodb.toNumber($savingAccount.balance)))
    #set($index = $index + 1)
    #set($savingAccountTransactPutItem = {"table": "savingAccounts",
        "operation": "PutItem",
        "key": $keyMap,
        "attributeValues": $attributeValues})
    $util.qr($savingAccountTransactPutItems.add($savingAccountTransactPutItem))
#end

#set($checkingAccountTransactPutItems = [])
#set($index = 0)
#foreach($checkingAccount in ${ctx.args.checkingAccounts})
    #set($keyMap = {})
    $util.qr($keyMap.put("accountNumber",
$util.dynamodb.toString($checkingAccount.accountNumber)))
    #set($attributeValues = {})
    $util.qr($attributeValues.put("username",
$util.dynamodb.toString($checkingAccount.username)))
    $util.qr($attributeValues.put("balance",
$util.dynamodb.toNumber($checkingAccount.balance)))
    #set($index = $index + 1)
    #set($checkingAccountTransactPutItem = {"table": "checkingAccounts",
        "operation": "PutItem",
        "key": $keyMap,
        "attributeValues": $attributeValues})
    $util.qr($checkingAccountTransactPutItems.add($checkingAccountTransactPutItem))
#end

#set($transactItems = [])
$util.qr($transactItems.addAll($savingAccountTransactPutItems))
```

```
$util.qr($transactItems.addAll($checkingAccountTransactPutItems))

{
  "version" : "2018-05-29",
  "operation" : "TransactWriteItems",
  "transactItems" : $util.toJson($transactItems)
}
```

以及下列回應映射範本：

回應映射範本

```
#if ($ctx.error)
  $util.appendError($ctx.error.message, $ctx.error.type, null,
    $ctx.result.cancellationReasons)
#end

#set($savingAccounts = [])
#foreach($index in [0..2])
  $util.qr($savingAccounts.add(${ctx.result.keys[$index]}))
#end

#set($checkingAccounts = [])
#foreach($index in [3..5])
  $util.qr($checkingAccounts.add(${ctx.result.keys[$index]}))
#end

#set($transactionResult = {})
$util.qr($transactionResult.put('savingAccounts', $savingAccounts))
$util.qr($transactionResult.put('checkingAccounts', $checkingAccounts))

$util.toJson($transactionResult)
```

儲存解析程式並導覽至 AWS AppSync 主控台的查詢區段以填入帳戶。

執行下列的變動：

```
mutation populateAccounts {
  populateAccounts (
    savingAccounts: [
      {accountNumber: "1", username: "Tom", balance: 100},
      {accountNumber: "2", username: "Amy", balance: 90},
      {accountNumber: "3", username: "Lily", balance: 80},
```

```

]
checkingAccounts: [
  {accountNumber: "1", username: "Tom", balance: 70},
  {accountNumber: "2", username: "Amy", balance: 60},
  {accountNumber: "3", username: "Lily", balance: 50},
]) {
  savingAccounts {
    accountNumber
  }
  checkingAccounts {
    accountNumber
  }
}
}

```

我們在一個變動中填入 3 個存款帳戶和 3 個支票帳戶。

使用 DynamoDB 主控台來驗證資料是否同時顯示在 savingAccounts 和 checkingAccounts 資料表中。

TransactWriteItems - 轉帳

使用下列請求映射範本將解析程式連接至transferMoney變動。請注意，amounts、savingAccountNumbers 以及 checkingAccountNumbers 的值是相同的。

```

#set($amounts = [])
#foreach($transaction in ${ctx.args.transactions})
  #set($attributeValueMap = {})
  $util.qr($attributeValueMap.put(":amount",
  $util.dynamodb.toNumber($transaction.amount)))
  $util.qr($amounts.add($attributeValueMap))
#end

#set($savingAccountTransactUpdateItems = [])
#set($index = 0)
#foreach($transaction in ${ctx.args.transactions})
  #set($keyMap = {})
  $util.qr($keyMap.put("accountNumber",
  $util.dynamodb.toString($transaction.savingAccountNumber)))
  #set($update = {})
  $util.qr($update.put("expression", "SET balance = balance - :amount"))
  $util.qr($update.put("expressionValues", $amounts[$index]))
  #set($index = $index + 1)
  #set($savingAccountTransactUpdateItem = {"table": "savingAccounts",

```

```

        "operation": "UpdateItem",
        "key": $keyMap,
        "update": $update})
    $util.qr($savingAccountTransactUpdateItems.add($savingAccountTransactUpdateItem))
#end

#set($checkingAccountTransactUpdateItems = [])
#set($index = 0)
#foreach($transaction in ${ctx.args.transactions})
    #set($keyMap = {})
    $util.qr($keyMap.put("accountNumber",
$util.dynamodb.toString($transaction.checkingAccountNumber)))
    #set($update = {})
    $util.qr($update.put("expression", "SET balance = balance + :amount"))
    $util.qr($update.put("expressionValues", $amounts[$index]))
    #set($index = $index + 1)
    #set($checkingAccountTransactUpdateItem = {"table": "checkingAccounts",
        "operation": "UpdateItem",
        "key": $keyMap,
        "update": $update})

    $util.qr($checkingAccountTransactUpdateItems.add($checkingAccountTransactUpdateItem))
#end

#set($transactionHistoryTransactPutItems = [])
#foreach($transaction in ${ctx.args.transactions})
    #set($keyMap = {})
    $util.qr($keyMap.put("transactionId", $util.dynamodb.toString(${utils.autoId()})))
    #set($attributeValues = {})
    $util.qr($attributeValues.put("from",
$util.dynamodb.toString($transaction.savingAccountNumber)))
    $util.qr($attributeValues.put("to",
$util.dynamodb.toString($transaction.checkingAccountNumber)))
    $util.qr($attributeValues.put("amount",
$util.dynamodb.toNumber($transaction.amount)))
    #set($transactionHistoryTransactPutItem = {"table": "transactionHistory",
        "operation": "PutItem",
        "key": $keyMap,
        "attributeValues": $attributeValues})

    $util.qr($transactionHistoryTransactPutItems.add($transactionHistoryTransactPutItem))
#end

#set($transactItems = [])

```

```

$util.qr($transactItems.addAll($savingAccountTransactUpdateItems))
$util.qr($transactItems.addAll($checkingAccountTransactUpdateItems))
$util.qr($transactItems.addAll($transactionHistoryTransactPutItems))

{
  "version" : "2018-05-29",
  "operation" : "TransactWriteItems",
  "transactItems" : $util.toJson($transactItems)
}

```

我們將會在單一 TransactWriteItems 操作中有 3 筆銀行交易。使用下列回應映射範本：

```

#if ($ctx.error)
  $util.appendError($ctx.error.message, $ctx.error.type, null,
    $ctx.result.cancellationReasons)
#end

#set($savingAccounts = [])
#foreach($index in [0..2])
  $util.qr($savingAccounts.add(${ctx.result.keys[$index]}))
#end

#set($checkingAccounts = [])
#foreach($index in [3..5])
  $util.qr($checkingAccounts.add(${ctx.result.keys[$index]}))
#end

#set($transactionHistory = [])
#foreach($index in [6..8])
  $util.qr($transactionHistory.add(${ctx.result.keys[$index]}))
#end

#set($transactionResult = {})
$util.qr($transactionResult.put('savingAccounts', $savingAccounts))
$util.qr($transactionResult.put('checkingAccounts', $checkingAccounts))
$util.qr($transactionResult.put('transactionHistory', $transactionHistory))

$util.toJson($transactionResult)

```

現在導覽至 AWS AppSync 主控台的查詢區段，並執行 transferMoney 變動，如下所示：

```

mutation write {
  transferMoney(

```

```

    transactions: [
      {savingAccountNumber: "1", checkingAccountNumber: "1", amount: 7.5},
      {savingAccountNumber: "2", checkingAccountNumber: "2", amount: 6.0},
      {savingAccountNumber: "3", checkingAccountNumber: "3", amount: 3.3}
    ]) {
      savingAccounts {
        accountNumber
      }
      checkingAccounts {
        accountNumber
      }
      transactionHistory {
        transactionId
      }
    }
  }
}
```

我們已傳送單一變動中的 2 筆銀行交易。使用 DynamoDB 主控台來驗證資料是否顯示在 savingAccounts、checkingAccounts 和 transactionHistory 資料表中。

TransactGetItems - 擷取帳戶

為了擷取單一交易請求中存款帳戶和支票帳戶的詳細資料，我們會將解析程式附加至結構模式中的 Query.getAccount GraphQL 操作。選取連接，前往 VTL 單元解析程式，然後在下一個畫面選擇在教學課程開始時建立的相同 TransactTutorial 資料來源。設定範本，如下所示：

請求映射範本

```

#set($savingAccountsTransactGets = [])
#foreach($savingAccountNumber in ${ctx.args.savingAccountNumbers})
  #set($savingAccountKey = {})
  $util.qr($savingAccountKey.put("accountNumber",
  $util.dynamodb.toString($savingAccountNumber)))
  #set($savingAccountTransactGet = {"table": "savingAccounts", "key":
  $savingAccountKey})
  $util.qr($savingAccountsTransactGets.add($savingAccountTransactGet))
#end

#set($checkingAccountsTransactGets = [])
#foreach($checkingAccountNumber in ${ctx.args.checkingAccountNumbers})
  #set($checkingAccountKey = {})
  $util.qr($checkingAccountKey.put("accountNumber",
  $util.dynamodb.toString($checkingAccountNumber)))
```

```

    #set($checkingAccountTransactGet = {"table": "checkingAccounts", "key":
    $checkingAccountKey})
    $util.qr($checkingAccountsTransactGets.add($checkingAccountTransactGet))
#end

#set($transactItems = [])
$util.qr($transactItems.addAll($savingAccountsTransactGets))
$util.qr($transactItems.addAll($checkingAccountsTransactGets))

{
  "version" : "2018-05-29",
  "operation" : "TransactGetItems",
  "transactItems" : $util.toJson($transactItems)
}

```

回應映射範本

```

#if ($ctx.error)
    $util.appendError($ctx.error.message, $ctx.error.type, null,
    $ctx.result.cancellationReasons)
#end

#set($savingAccounts = [])
#foreach($index in [0..2])
    $util.qr($savingAccounts.add(${ctx.result.items[$index]}))
#end

#set($checkingAccounts = [])
#foreach($index in [3..4])
    $util.qr($checkingAccounts.add($ctx.result.items[$index]))
#end

#set($transactionResult = {})
$util.qr($transactionResult.put('savingAccounts', $savingAccounts))
$util.qr($transactionResult.put('checkingAccounts', $checkingAccounts))

$util.toJson($transactionResult)

```

儲存解析程式並導覽至 AWS AppSync 主控台的查詢區段。若要擷取存款帳戶和支票帳戶，請執行下列查詢：

```

query getAccounts {
  getAccounts(

```



```
savingAccountNumbers: ["1", "2", "3"],
checkingAccountNumbers: ["1", "2"]
) {
  savingAccounts {
    accountNumber
    username
    balance
  }
  checkingAccounts {
    accountNumber
    username
    balance
  }
}
```

我們已成功使用 AWS AppSync 示範 DynamoDB 交易的使用方式。

使用 HTTP 解析程式 in AWS AppSync

Note

我們現在主要支援 APPSYNC_JS 執行期及其文件。請考慮[在此處](#)使用 APPSYNC_JS 執行期及其指南。

AWS AppSync 可讓您使用支援的資料來源（即 Amazon DynamoDB AWS Lambda、Amazon OpenSearch Service 或 Amazon Aurora）來執行各種操作，以及任何任意 HTTP 端點來解析 GraphQL 欄位。在您的 HTTP 端點可供使用之後，您可以使用資料來源連線到這些端點。然後，您可以在結構描述中設定解析程式以執行 GraphQL 操作，例如查詢、變動和訂閱。本教學將逐步說明一些常見的範例。

在本教學課程中，您將 REST API（使用 Amazon API Gateway 和 Lambda 建立）與 AWS AppSync GraphQL 端點搭配使用。

一鍵設定

如果您想要自動設定 GraphQL 端點 in AWS AppSync，並設定 HTTP 端點（使用 Amazon API Gateway 和 Lambda），您可以使用下列 AWS CloudFormation 範本：

Launch Stack 

建立 REST API

您可以使用下列 AWS CloudFormation 範本來設定適用於本教學課程的 REST 端點：

Launch Stack 

AWS CloudFormation 堆疊會執行下列步驟：

1. 設定 Lambda 函數，其中包含您微服務的商業邏輯。
2. 使用下列endpoint/method/content類型組合設定 API Gateway REST API：

API 資源路徑	HTTP 方法	已支援的內容類型
/v1/users	POST	application/json
/v1/users	GET	application/json
/v1/users/1	GET	application/json
/v1/users/1	PUT	application/json
/v1/users/1	DELETE	application/json

建立 GraphQL API

若要在 AWS AppSync 中建立 GraphQL API：

- 開啟 AWS AppSync 主控台，然後選擇建立 API。
- 針對 API 名稱，輸入 UserData。
- 選擇 Custom schema (自訂結構描述)。
- 選擇建立。

AWS AppSync 主控台會使用 API 金鑰身分驗證模式為您建立新的 GraphQL API。您可以使用主控台來設定 GraphQL API 的其餘部分，並在本教學的其餘部分中對其執行查詢。

建立 GraphQL 結構描述

現在您有一個 GraphQL API，讓我們來建立 GraphQL 結構描述吧。從 AWS AppSync 主控台的結構描述編輯器中，請確定您的結構描述符合下列結構描述：

```
schema {
  query: Query
  mutation: Mutation
}

type Mutation {
  addUser(userInput: UserInput!): User
  deleteUser(id: ID!): User
}

type Query {
  getUser(id: ID): User
  listUser: [User!]!
}

type User {
  id: ID!
  username: String!
  firstname: String
  lastname: String
  phone: String
  email: String
}

input UserInput {
  id: ID!
  username: String!
  firstname: String
  lastname: String
  phone: String
  email: String
}
```

設定您的 HTTP 資料來源

若要設定您的 HTTP 資料來源，請執行以下作業：

- 在 DataSources 標籤中，選擇 New (新增)，然後輸入易記的資料來源名稱 (例如，HTTP)。
- 在 Data source type (資料來源類型) 中選擇 HTTP。
- 將端點設定為建立的 API Gateway 端點。請確定您沒有在端點中包含階段名稱。

注意：目前 AWS AppSync 僅支援公有端點。

注意：如需認證授權機構的詳細資訊，AWS AppSync 請參閱 [HTTPS 端點的認證授權機構 AWS AppSync \(CA\)](#)。

設定解析程式

在此步驟中，您會將 http 資料來源連線到 getUser 查詢。

設定解析程式：

- 選擇 Schema (結構描述) 標籤。
- 在查詢類型下右側的資料類型窗格中，尋找 getUser 欄位，然後選擇連接。
- 在 Data source name (資料來源名稱) 中，選擇 HTTP。
- 將以下內容貼到 Configure the request mapping template (設定請求映射範本) 區段：

```
{
  "version": "2018-05-29",
  "method": "GET",
  "params": {
    "headers": {
      "Content-Type": "application/json"
    }
  },
  "resourcePath": $util.toJson("/v1/users/${ctx.args.id}")
}
```

- 將以下內容貼到 Configure the response mapping template (設定回應映射範本) 區段：

```
## return the body
#if($ctx.result.statusCode == 200)
  ##if response is 200
  $ctx.result.body
```

```
#else
  ##if response is not 200, append the response to error block.
  $utils.appendError($ctx.result.body, "$ctx.result.statusCode")
#end
```

- 選擇 Query (查詢) 標籤，然後執行以下查詢：

```
query GetUser{
  getUser(id:1){
    id
    username
  }
}
```

這應該會傳回以下回應：

```
{
  "data": {
    "getUser": {
      "id": "1",
      "username": "nadia"
    }
  }
}
```

- 選擇 Schema (結構描述) 標籤。
- 在 Mutation 下右側的資料類型窗格中，尋找 addUser 欄位，然後選擇連接。
- 在 Data source name (資料來源名稱) 中，選擇 HTTP。
- 將以下內容貼到 Configure the request mapping template (設定請求映射範本) 區段：

```
{
  "version": "2018-05-29",
  "method": "POST",
  "resourcePath": "/v1/users",
  "params":{
    "headers":{
      "Content-Type": "application/json",
    },
    "body": $util.toJson($ctx.args.userInput)
```

```
}
}
```

- 將以下內容貼到 Configure the response mapping template (設定回應映射範本) 區段：

```
## Raise a GraphQL field error in case of a datasource invocation error
#if($ctx.error)
    $util.error($ctx.error.message, $ctx.error.type)
#end
## if the response status code is not 200, then return an error. Else return the body
**
#if($ctx.result.statusCode == 200)
    ## If response is 200, return the body.
    $ctx.result.body
#else
    ## If response is not 200, append the response to error block.
    $utils.appendError($ctx.result.body, "$ctx.result.statusCode")
#end
```

- 選擇 Query (查詢) 標籤，然後執行以下查詢：

```
mutation addUser{
  addUser(userInput:{
    id:"2",
    username:"shaggy"
  }){
    id
    username
  }
}
```

這應該會傳回以下回應：

```
{
  "data": {
    "getUser": {
      "id": "2",
      "username": "shaggy"
    }
  }
}
```

```
}
```

叫用 AWS 服務

您可以使用 HTTP 解析程式來設定 AWS 服務的 GraphQL API 介面。對的 HTTP 請求 AWS 必須使用 [Signature 第 4 版程序](#) 簽署，以便 AWS 可以識別傳送這些請求的人員。當您將 IAM 角色與 HTTP 資料來源建立關聯時，AWS AppSync 會代表您計算簽章。

您提供兩個額外的元件，以使用 HTTP 解析程式叫用 AWS 服務：

- 具有呼叫 AWS 服務 APIs 許可的 IAM 角色
- 資料來源中的簽署組態

例如，如果您想要使用 HTTP 解析程式呼叫 [ListGraphqlApis 操作](#)，請先 [建立 AppSync 擔任的 IAM 角色](#)，並連接下列政策：AWS AppSync

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "appsync:ListGraphqlApis"
      ],
      "Effect": "Allow",
      "Resource": "*"
    }
  ]
}
```

接著，建立 HTTP 資料來源 for AWS AppSync。在此範例中，您在美國西部（奧勒岡）區域呼叫 AWS AppSync。在名為 `http.json` 的檔案中設定以下 HTTP 組態，包含簽署區域和服務名稱：

```
{
  "endpoint": "https://appsync.us-west-2.amazonaws.com/",
  "authorizationConfig": {
    "authorizationType": "AWS_IAM",
    "awsIamConfig": {
```

```
        "signingRegion": "us-west-2",
        "signingServiceName": "appsync"
    }
}
```

然後，使用 AWS CLI 建立具有關聯角色的資料來源，如下所示：

```
aws appsync create-data-source --api-id <API-ID> \
                               --name AWSAppSync \
                               --type HTTP \
                               --http-config file:///http.json \
                               --service-role-arn <ROLE-ARN>
```

當您將解析程式連接至結構描述中的 欄位時，請使用下列請求映射範本來 call AWS AppSync：

```
{
  "version": "2018-05-29",
  "method": "GET",
  "resourcePath": "/v1/apis"
}
```

當您為此資料來源執行 GraphQL 查詢時，AWS AppSync 會使用您提供的角色簽署請求，並在請求中包含簽章。查詢會傳回您帳戶中該 AWS 區域中的 AWS AppSync GraphQL APIs 清單。

搭配 使用 Aurora Serverless v2 AWS AppSync

使用 將您的 GraphQL API 連接至 Aurora Serverless 資料庫 AWS AppSync。此整合可讓您透過 GraphQL 查詢、變動和訂閱執行 SQL 陳述式，讓您有彈性地與關聯式資料互動。

Note

此教學會使用 US-EAST-1 區域。

優勢

- GraphQL 與關聯式資料庫之間的無縫整合
- 能夠透過 GraphQL 介面執行 SQL 操作

- 搭配 Aurora Serverless v2 的無伺服器可擴展性
- 透過 AWS Secrets Manager 安全存取資料
- 透過輸入淨化防止 SQL 注入
- 靈活的查詢功能，包括篩選和範圍操作

常用案例

- 使用關聯式資料需求建置可擴展的應用程式
- 建立同時需要 GraphQL 彈性和 SQL 資料庫功能的 APIs
- 透過 GraphQL 變動和查詢管理資料操作
- 實作安全資料庫存取模式

在本教學課程中，您將學到以下內容。

- 設定 Aurora Serverless v2 叢集
- 啟用資料 API 功能
- 建立和設定資料庫結構
- 定義資料庫操作的 GraphQL 結構描述
- 實作查詢和變動的解析程式
- 透過適當的輸入清理來保護您的資料存取
- 透過 GraphQL 介面執行各種資料庫操作

主題

- [設定資料庫叢集](#)
- [啟用 Data API](#)
- [建立資料庫及資料表](#)
- [GraphQL 結構描述](#)
- [將您的 API 連線至資料庫操作](#)
- [透過 API 修改您的資料](#)
- [擷取您的資料](#)
- [保護您的資料存取](#)

設定資料庫叢集

將 Amazon RDS 資料來源新增至 之前 AWS AppSync，您必須先在 Aurora Serverless v2 叢集上啟用資料 API，並使用 設定秘密AWS Secrets Manager。您可以使用 建立 Aurora Serverless v2 叢集 AWS CLI：

```
aws rds create-db-cluster \  
  --db-cluster-identifier appsync-tutorial \  
  --engine aurora-mysql \  
  --engine-version 8.0 \  
  --serverless-v2-scaling-configuration MinCapacity=0,MaxCapacity=1 \  
  --master-username USERNAME \  
  --master-user-password COMPLEX_PASSWORD \  
  --enable-http-endpoint
```

這會傳回叢集的 ARN。

建立叢集之後，您必須使用下列命令新增 Aurora Serverless v2 執行個體。

```
aws rds create-db-instance \  
  --db-cluster-identifier appsync-tutorial \  
  --db-instance-identifier appsync-tutorial-instance-1 \  
  --db-instance-class db.serverless \  
  --engine aurora-mysql
```

Note

這些端點需要一些時間才能啟用。您可以在叢集的連線與安全索引標籤中的 Amazon RDS 主控台中檢查其狀態。您也可以使用下列 AWS CLI 命令來檢查叢集的狀態。

```
aws rds describe-db-clusters \  
  --db-cluster-identifier appsync-tutorial \  
  --query "DBClusters[0].Status"
```

您可以使用 AWS Secrets Manager 主控台或 AWS CLI 建立秘密，並使用 USERNAME 和 COMPLEX_PASSWORD 一個步驟中的輸入檔案，如下所示。

```
{  
  "username": "USERNAME",
```

```
"password": "COMPLEX_PASSWORD"
}
```

將此做為參數傳遞至 AWS CLI：

```
aws secretsmanager create-secret --name HttpRDSecret --secret-string file://creds.json
--region us-east-1
```

這會傳回秘密的 ARN。

記下 ARN (屬於 Aurora Serverless 叢集及 Secret) 以供日後建立資料來源時用於 AppSync 主控台。

啟用 Data API

您可以[依照 RDS 文件中的說明](#)，在您的叢集上啟用 Data API。Data API 必須先行啟用，才能新增為 AppSync 資料來源。

建立資料庫及資料表

啟用資料 API 後，您就可以確保它可與 中的 `aws rds-data execute-statement` 命令搭配使用 AWS CLI。這可確保您的 Aurora Serverless 叢集在經過正確設定後，才新增到您的 AppSync API。首先使用 `--sql` 參數建立名為 TESTDB 的資料庫，如下所示：

```
aws rds-data execute-statement --resource-arn "arn:aws:rds:us-east-1:123456789000:cluster:http-endpoint-test" \
--schema "mysql" --secret-arn "arn:aws:secretsmanager:us-east-1:123456789000:secret:testHttp2-AmNvc1" \
--region us-east-1 --sql "create DATABASE TESTDB"
```

如果這次執行沒有錯誤，則搭配 `create table` 命令新增資料表：

```
aws rds-data execute-statement --resource-arn "arn:aws:rds:us-east-1:123456789000:cluster:http-endpoint-test" \
--schema "mysql" --secret-arn "arn:aws:secretsmanager:us-east-1:123456789000:secret:testHttp2-AmNvc1" \
--region us-east-1 \
--sql "create table Pets(id varchar(200), type varchar(200), price float)" --database "TESTDB"
```

如果所有執行都沒有問題，您就可以繼續將該叢集新增到您的 AppSync API。

GraphQL 結構描述

現在您的 Aurora Serverless Data API 已經啟動並搭配資料表運作，我們會建立 GraphQL 結構描述，並連接可執行變動和訂閱的解析程式。在 AWS AppSync 主控台中建立新的 API 並導覽至結構描述頁面，然後輸入以下內容：

```
type Mutation {
  createPet(input: CreatePetInput!): Pet
  updatePet(input: UpdatePetInput!): Pet
  deletePet(input: DeletePetInput!): Pet
}

input CreatePetInput {
  type: PetType
  price: Float!
}

input UpdatePetInput {
  id: ID!
  type: PetType
  price: Float!
}

input DeletePetInput {
  id: ID!
}

type Pet {
  id: ID!
  type: PetType
  price: Float
}

enum PetType {
  dog
  cat
  fish
  bird
  gecko
}

type Query {
  getPet(id: ID!): Pet
```

```

    listPets: [Pet]
    listPetsByPriceRange(min: Float, max: Float): [Pet]
  }

  schema {
    query: Query
    mutation: Mutation
  }

```

儲存您的結構描述並瀏覽到 Data Sources (資料來源) 頁面，並建立新的資料來源。選取資料來源類型為 Relational database (關聯式資料庫)，並提供易記名稱。使用您在上一個步驟中所建立的資料庫名稱，以及您所建立叢集的叢集 ARN。處理角色時，您可以使用 AppSync 來建立新角色，或是搭配類似下面政策來建立一個角色：

JSON

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "rds-data:DeleteItems",
        "rds-data:ExecuteSql",
        "rds-data:ExecuteStatement",
        "rds-data:GetItems",
        "rds-data:InsertItems",
        "rds-data:UpdateItems"
      ],
      "Resource": [
        "arn:aws:rds:us-east-1:123456789012:cluster:mydbcluster",
        "arn:aws:rds:us-east-1:123456789012:cluster:mydbcluster:*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "secretsmanager:GetSecretValue"
      ],
      "Resource": [
        "arn:aws:secretsmanager:us-east-1:123456789012:secret:mysecret",
        "arn:aws:secretsmanager:us-east-1:123456789012:secret:mysecret:*"
      ]
    }
  ]
}

```

```

    ]
  }
}
}

```

請注意，在您授予角色存取權的這個政策中有 2 個陳述式。第一個資源是您的 Aurora Serverless 叢集，第二個資源是您的 AWS Secrets Manager ARN。您將需要先在 AppSync 資料來源組態中同時提供兩種 ARN 後，再按一下 Create (建立)。

將此做為 參數傳遞至 AWS CLI。

```

aws secretsmanager create-secret \
  --name HttpRDSecret \
  --secret-string file://creds.json \
  --region us-east-1

```

這會傳回秘密的 ARN。記下 Aurora Serverless 叢集的 ARN 和 Secret，以供稍後在 AWS AppSync 主控台中建立資料來源時使用。

建置您的資料庫結構

啟用資料 API 後，您就可以確保它可與 中的 `aws rds-data execute-statement` 命令搭配使用 AWS CLI。這將確保您的 Aurora Serverless v2 叢集在新增至 AWS AppSync API 之前已正確設定。首先，使用 `--sql` 參數建立名為 TESTDB 的資料庫，如下所示。

```

aws rds-data execute-statement \
  --resource-arn "arn:aws:rds:us-east-1:111122223333:cluster:appsync-tutorial" \
  --secret-arn "arn:aws:secretsmanager:us-east-1:111122223333:secret:appsync-tutorial-rds-secret" \
  --region us-east-1 \
  --sql "create DATABASE TESTDB"

```

如果執行時沒有錯誤，請使用下列建立資料表命令新增資料表。

```

aws rds-data execute-statement \
  --resource-arn "arn:aws:rds:us-east-1:111122223333:cluster:http-endpoint-test" \
  --secret-arn "arn:aws:secretsmanager:us-east-1:111122223333:secret:testHttp2-AmNvc1" \
  --region us-east-1 \

```

```
--sql "create table Pets(id varchar(200), type varchar(200), price float)" \  
--database "TESTDB"
```

設計您的 API 介面

Aurora Serverless v2 Data API 啟動並使用資料表執行後，請建立 GraphQL 結構描述，並連接解析程式以執行變動和訂閱。在 AWS AppSync 主控台中建立新的 API，並導覽至主控台內的結構描述頁面，然後輸入以下內容。

```
type Mutation {  
    createPet(input: CreatePetInput!): Pet  
    updatePet(input: UpdatePetInput!): Pet  
    deletePet(input: DeletePetInput!): Pet  
}  
  
input CreatePetInput {  
    type: PetType  
    price: Float!  
}  
  
input UpdatePetInput {  
    id: ID!  
    type: PetType  
    price: Float!  
}  
  
input DeletePetInput {  
    id: ID!  
}  
  
type Pet {  
    id: ID!  
    type: PetType  
    price: Float  
}  
  
enum PetType {  
    dog  
    cat  
    fish  
    bird  
    gecko  
}
```

```

type Query {
  getPet(id: ID!): Pet
  listPets: [Pet]
  listPetsByPriceRange(min: Float, max: Float): [Pet]
}

schema {
  query: Query
  mutation: Mutation
}

```

儲存您的結構描述並瀏覽到 Data Sources (資料來源) 頁面，並建立新的資料來源。選擇資料來源類型的關聯式資料庫，並提供易記的名稱。使用您在上一個步驟中所建立的資料庫名稱，以及您所建立叢集的叢集 ARN。對於角色，您可以 AWS AppSync 建立新的角色，或使用類似下列的政策來建立角色。

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "rds-data:DeleteItems",
        "rds-data:ExecuteSql",
        "rds-data:ExecuteStatement",
        "rds-data:GetItems",
        "rds-data:InsertItems",
        "rds-data:UpdateItems"
      ],
      "Resource": [
        "arn:aws:rds:us-east-1:111122223333:cluster:mysqlcluster",
        "arn:aws:rds:us-east-1:111122223333:cluster:mysqlcluster:*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "secretsmanager:GetSecretValue"
      ],
      "Resource": [
        "arn:aws:secretsmanager:us-east-1:111122223333:secret:mysecret",
        "arn:aws:secretsmanager:us-east-1:111122223333:secret:mysecret:*"
      ]
    }
  ]
}

```



```
    }
  ]
}
```

請注意，在您授予角色存取權的這個政策中有 2 個陳述式。第一個資源是您的 Aurora Serverless v2 叢集，第二個是您的 AWS Secrets Manager ARN。在按一下建立之前，您需要在 AWS AppSync 資料來源組態中提供這兩個 ARNs。

將您的 API 連線至資料庫操作

現在我們有有效的 GraphQL 結構描述和 RDS 資料來源，您可以將解析程式連接到結構描述的 GraphQL 欄位。我們的 API 提供下列功能：

1. 使用 `Mutation.createPet` 欄位建立寵物
2. 使用 `Mutation.updatePet` 欄位更新寵物
3. 使用 `Mutation.deletePet` 欄位刪除寵物
4. 透過 `Query.getPet` 欄位取得單一使用
5. 使用 `Query.listPets` 欄位列出所有
6. 使用 `Query.listPetsByPriceRange` 欄位列出價格範圍內的寵物

Mutation.createPet

從 AWS AppSync 主控台的結構描述編輯器，選擇 的連接解析程式 `createPet(input: CreatePetInput!): Pet`。選擇 RDS 資料來源。在 request mapping template (要求映射範本) 區段中，新增下列範本：

```
#set($id=$utils.autoId())
{
  "version": "2018-05-29",
  "statements": [
    "insert into Pets VALUES (:ID, :TYPE, :PRICE)",
    "select * from Pets WHERE id = :ID"
  ],
  "variableMap": {
    ":ID": "$ctx.args.input.id",
    ":TYPE": $util.toJson($ctx.args.input.type),
    ":PRICE": $util.toJson($ctx.args.input.price)
  }
}
```

系統會根據陳述式陣列中的順序依序執行 SQL 陳述式。結果將以相同順序傳回。由於這是一個變動，因此您將在插入後執行選取陳述式，以擷取遞交的值，以填入 GraphQL 回應映射範本。

在 response mapping template (回應映射範本) 區段中，新增下列範本：

```
$utils.toJson($utils.rds.toJsonObject($ctx.result)[1][0])
```

由於陳述式有兩個 SQL 查詢，所以我們需要使用下列程式碼，指定自資料庫傳回矩陣中的第二個結果：\$utils.rds.toJsonString(\$ctx.result))[1][0])。

Mutation.updatePet

從 AWS AppSync 主控台的結構描述編輯器中，選擇連接解析程式。updatePet(input: UpdatePetInput!)：Pet 選擇您的 RDS 資料來源。在請求映射範本區段中，新增下列範本。

```
{
  "version": "2018-05-29",
  "statements": [
    $util.toJson("update Pets set type=:TYPE, price=:PRICE WHERE id=:ID"),
    $util.toJson("select * from Pets WHERE id = :ID")
  ],
  "variableMap": {
    ":ID": "$ctx.args.input.id",
    ":TYPE": $util.toJson($ctx.args.input.type),
    ":PRICE": $util.toJson($ctx.args.input.price)
  }
}
```

在回應映射範本區段中，新增下列範本。

```
$utils.toJson($utils.rds.toJsonObject($ctx.result)[1][0])
```

Mutation.deletePet

從 AWS AppSync 主控台的結構描述編輯器中，選擇連接解析程式。deletePet(input: DeletePetInput!)：Pet 選擇您的 RDS 資料來源。在請求映射範本區段中，新增下列範本。

```
{
  "version": "2018-05-29",
  "statements": [
    $util.toJson("select * from Pets WHERE id=:ID"),
    $util.toJson("delete from Pets WHERE id=:ID")
  ]
}
```

```
    ],  
    "variableMap": {  
      ":ID": "$ctx.args.input.id"  
    }  
  }  
}
```

在回應映射範本區段中，新增下列範本。

```
$utils.toJson($utils.rds.toJsonObject($ctx.result)[0][0])
```

Query.getPet

現在已為您的結構描述建立變動，請連接三個查詢，展示如何取得個別項目、清單和套用 SQL 篩選。從 AWS AppSync 主控台的結構描述編輯器中，選擇連接 解析程式 `getPet(id: ID!): Pet`。選擇您的 RDS 資料來源。在請求映射範本區段中，新增下列範本。

```
{  
  "version": "2018-05-29",  
  "statements": [  
    $util.toJson("select * from Pets WHERE id=:ID")  
  ],  
  "variableMap": {  
    ":ID": "$ctx.args.id"  
  }  
}
```

在 response mapping template (回應映射範本) 區段中，新增下列範本：

```
$utils.toJson($utils.rds.toJsonObject($ctx.result)[0][0])
```

Query.listPets

從 AWS AppSync 主控台的结构描述編輯器，選擇 的連接解析程式 `getPet(id: ID!): Pet`。選擇您的 RDS 資料來源。在請求映射範本區段中，新增下列範本。

```
{  
  "version": "2018-05-29",  
  "statements": [  
    "select * from Pets"  
  ]  
}
```

在回應映射範本區段中，新增下列範本。

```
$utils.toJson($utils.rds.toJsonObject($ctx.result)[0])
```

Query.listPetsByPriceRange

從 AWS AppSync 主控台結構描述編輯器，選擇 的連接解析程式 `getPet(id: ID!): Pet`。選擇您的 RDS 資料來源。在請求映射範本區段中，新增下列範本。

```
{
  "version": "2018-05-29",
  "statements": [
    "select * from Pets where price > :MIN and price < :MAX"
  ],
  "variableMap": {
    ":MAX": $util.toJson($ctx.args.max),
    ":MIN": $util.toJson($ctx.args.min)
  }
}
```

在 response mapping template (回應映射範本) 區段中，新增下列範本：

```
$utils.toJson($utils.rds.toJsonObject($ctx.result)[0])
```

透過 API 修改您的資料

現在，您已運用 SQL 陳述式設定您的所有解析程式，並已將 GraphQL API 連接到您的 Serverless Aurora Data API，您可以開始執行變動和查詢了。在 AWS AppSync 主控台，選擇查詢索引標籤，然後輸入下列內容來建立寵物：

```
mutation add {
  createPet(input : { type:fish, price:10.0 }){
    id
    type
    price
  }
}
```

回應應該包含如下的 id、type 和 price：

```
{
  "data": {
    "createPet": {
      "id": "c6fedbbe-57ad-4da3-860a-ffe8d039882a",
      "type": "fish",
      "price": "10.0"
    }
  }
}
```

您可以執行 `updatePet` 變動，修改此項目：

```
mutation update {
  updatePet(input : {
    id: ID_PLACEHOLDER,
    type:bird,
    price:50.0
  }){
    id
    type
    price
  }
}
```

請注意，我們使用先前從 `createPet` 操作傳回的 ID。這將是您的記錄在解析程式運用 `$util.autoId()` 時的唯一值。您可以用類似的方式刪除記錄：

```
mutation delete {
  deletePet(input : {id:ID_PLACEHOLDER}){
    id
    type
    price
  }
}
```

運用第一個變動，搭配 `price` 的幾個不同值，建立一些記錄，然後執行一些查詢。

擷取您的資料

仍在主控台的查詢索引標籤中，請使用下列陳述式列出您已建立的所有記錄。

```
query allpets {
```

```
listPets {  
  id  
  type  
  price  
}  
}
```

透過下列 GraphQL 查詢，利用 Query.listPetsByPriceRange 映射範本 where price > :MIN and price < :MAX 中具有 的 SQL WHERE 述詞：

```
query petsByPriceRange {  
  listPetsByPriceRange(min:1, max:11) {  
    id  
    type  
    price  
  }  
}
```

您應該只會看到 price 超過 \$1 元或不到 \$10 元的記錄。最後，您可以執行查詢來擷取個別記錄，如下所示：

```
query onePet {  
  getPet(id:ID_PLACEHOLDER){  
    id  
    type  
    price  
  }  
}
```

保護您的資料存取

SQL Injection 是資料庫應用程式中的安全漏洞。當攻擊者透過使用者輸入欄位插入惡意 SQL 程式碼時，就會發生這種情況。這可以允許未經授權存取資料庫資料。建議您在處理所有使用者輸入之前，使用仔細驗證並清除，variableMap 以防止 SQL Injection 攻擊。如果未使用變數映射，您必須負責清理其 GraphQL 操作的引數。達成這個淨化的一種方法是先在要求映射範本中提供輸入特定驗證步驟，接著再對 Data API 執行 SQL 陳述式。讓我們來看看如何修改 listPetsByPriceRange 範例的請求映射範本。您可以執行以下操作，而不再只是依賴使用者輸入：

```
#set($validMaxPrice = $util.matches("\d{1,3}[,\\.]?(\d{1,2})?", $ctx.args.maxPrice))  
  
#set($validMinPrice = $util.matches("\d{1,3}[,\\.]?(\d{1,2})?", $ctx.args.minPrice))
```

```

#if (!$validMaxPrice || !$validMinPrice)
    $util.error("Provided price input is not valid.")
#end
{
    "version": "2018-05-29",
    "statements": [
        "select * from Pets where price > :MIN and price < :MAX"
    ],

    "variableMap": {
        ":MAX": $util.toJson($ctx.args.maxPrice),
        ":MIN": $util.toJson($ctx.args.minPrice)
    }
}

```

在對 Data API 執行解析程式時防堵惡意輸入的另外一種方法，就是同時使用預備的陳述式來搭配預存程序和參數化的輸入。例如，在 `listPets` 的解析程式中，定義下列執行 `select` 為預先準備陳述式的程序：

```

CREATE PROCEDURE listPets (IN type_param VARCHAR(200))
BEGIN
    PREPARE stmt FROM 'SELECT * FROM Pets where type=?';
    SET @type = type_param;
    EXECUTE stmt USING @type;
    DEALLOCATE PREPARE stmt;
END

```

在 Aurora Serverless v2 執行個體中建立此項目。

```

aws rds-data execute-statement --resource-arn "arn:aws:rds:us-east-1:xxxxxxxxxxxx:cluster:http-endpoint-test" \
--schema "mysql" --secret-arn "arn:aws:secretsmanager:us-east-1:xxxxxxxxxxxx:secret:httpendpoint-xxxxxx" \
--region us-east-1 --database "DB_NAME" \
--sql "CREATE PROCEDURE listPets (IN type_param VARCHAR(200)) BEGIN PREPARE stmt FROM 'SELECT * FROM Pets where type=?'; SET @type = type_param; EXECUTE stmt USING @type; DEALLOCATE PREPARE stmt; END"

```

針對 `listPets` 產生的解析程式碼變得簡單，因為我們現在只要呼叫預存程序。至少，任何字串輸入都應將單引號[逸出](#)。

```
#set ($validType = $util.isString($ctx.args.type) && !
$util.isNullOrBlank($ctx.args.type))
#if (!$validType)
    $util.error("Input for 'type' is not valid.", "ValidationError")
#end

{
  "version": "2018-05-29",
  "statements": [
    "CALL listPets(:type)"
  ]
  "variableMap": {
    ":type": $util.toJson($ctx.args.type.replace("'", "''))
  }
}
```

使用逸出字串

使用單引號在 SQL 陳述式中標記字串常值的開頭和結尾，例如。 'some string value'若要允許在字串中使用具有一或多個單引號字元 (') 的字串值，必須以兩個單引號 (') 取代每個字串值。例如，如果輸入字串是 Nadia's dog，則您會將其逸出為如下的 SQL 陳述式：

```
update Pets set type='Nadia''s dog' WHERE id='1'
```

在 AWS AppSync 中使用管道解析程式

Note

我們現在主要支援 APPSYNC_JS 執行期及其文件。請考慮[在此處](#)使用 APPSYNC_JS 執行期及其指南。

AWS AppSync 提供簡單的方法，透過單位解析程式將 GraphQL 欄位連接至單一資料來源。不過，執行單一操作可能還是不夠。管道解析程式的功能是對資料來源依序執行操作。在您的 API 中建立函數，並將其連接到管道解析程式。每個函數執行結果都會輸送到下一個函數，直到沒有要執行的函數。使用管道解析程式，您現在可以直接在 AWS AppSync 中建置更複雜的工作流程。在此教學課程中，您會建置一個簡單的圖片檢視應用程式，而使用者可以透過此應用程式來張貼圖片和檢視朋友所張貼的圖片。

一鍵設定

如果您想要自動設定 GraphQL 端點 in AWS AppSync，並設定所有解析程式和必要的 AWS 資源，您可以使用下列 AWS CloudFormation 範本：

A yellow button with a blue play icon and the text "Launch Stack".

這個堆疊會在您的帳戶中建立下列資源：

- IAM Role for AWS AppSync 存取您帳戶中的資源
- 2 個 DynamoDB 資料表
- 1 個 Amazon Cognito 使用者工具
- 2 個 Amazon Cognito 使用者集區群組
- 3 個 Amazon Cognito 使用者集區使用者
- 1 AWS AppSync API

在 AWS CloudFormation 堆疊建立程序結束時，您會收到三位建立的 Amazon Cognito 使用者各一封電子郵件。每封電子郵件都包含臨時密碼，您可用來以 Amazon Cognito 使用者身分登入 AWS AppSync 主控台。儲存這些密碼，以供教學課程其餘內容使用。

手動設定

如果您偏好透過 AWS AppSync 主控台手動進行 step-by-step 程序，請遵循下列設定程序。

設定您的非 AWS AppSync 資源

API 會與兩個 DynamoDB 資料表通訊：存放圖片的圖片資料表，以及存放使用者之間關係的好友資料表。API 已設定為使用 Amazon Cognito 使用者集區做為驗證類型。下列 AWS CloudFormation 堆疊會在帳戶中設定這些資源。

A yellow button with a blue play icon and the text "Launch Stack".

在 AWS CloudFormation 堆疊建立程序結束時，您會收到三位建立的 Amazon Cognito 使用者各一封電子郵件。每封電子郵件都包含臨時密碼，您可用來以 Amazon Cognito 使用者身分登入 AWS AppSync 主控台。儲存這些密碼，以供教學課程其餘內容使用。

建立 GraphQL API

若要建立 GraphQL API in AWS AppSync：

1. 開啟 AWS AppSync 主控台，然後選擇從頭開始，然後選擇開始。
2. 將 API 名稱設定為 AppSyncTutorial-PicturesViewer。
3. 選擇 Create (建立)。

AWS AppSync 主控台會使用 API 金鑰身分驗證模式為您建立新的 GraphQL API。您可以使用主控台來設定其他 GraphQL API 和對其執行查詢，以進行此教學的其他部分。

設定 GraphQL API

您需要使用您剛建立的 Amazon Cognito 使用者集區來設定 AWS AppSync API。

1. 選擇 Settings (設定) 標籤。
2. 在 Authorization Type (授權類型) 區段下，選擇 Amazon Cognito User Pool (Amazon Cognito 使用者集區)。
3. 在使用者集區組態下，為AWS 區域選擇 US-WEST-2。
4. 選擇 AppSyncTutorial-UserPool 使用者集區。
5. 選擇 DENY 做為預設動作。
6. 保留 AppId client regex (AppId 用戶端 regex) 欄位空白。
7. 選擇 Save (儲存)。

API 現在已設定為使用 Amazon Cognito 使用者集區做為驗證類型。

設定 DynamoDB 資料表的資料來源

建立 DynamoDB 資料表之後，在主控台中導覽至 your AWS AppSync GraphQL API，然後選擇資料來源索引標籤。現在，您將為您剛建立的每個 DynamoDB 資料表建立資料來源 in AWS AppSync。

1. 選擇 Data source (資料來源) 標籤。
2. 選擇 New (新建) 來建立新資料來源。
3. 輸入 PicturesDynamoDBTable 做為資料來源名稱。
4. 選擇 Amazon DynamoDB table (Amazon DynamoDB 資料表) 做為資料來源類型。

5. 選擇 US-WEST-2 做為區域。
6. 從資料表清單中，選擇 AppSyncTutorial-PicturesDynamoDBDynamoDB 資料表。
7. 在建立或使用現有角色區段中，選擇現有角色。
8. 從 CloudFormation 範本中，選擇剛建立的角色。如果您未變更 ResourceNamePrefix，角色的名稱應該就是 AppSyncTutorial-DynamoDBRole。
9. 選擇 Create (建立)。

針對好友資料表重複相同的程序，如果您在建立 CloudFormation 堆疊時未變更 ResourceNamePrefix 參數，則 DynamoDB 資料表的名稱應為 AppSyncTutorial-Friends。

建立 GraphQL 結構描述

現在，資料來源已連接至 DynamoDB 資料表，讓我們建立 GraphQL 結構描述。從 AWS AppSync 主控台下的結構描述編輯器，確保您的結構描述符合下列結構描述：

```
schema {
  query: Query
  mutation: Mutation
}

type Mutation {
  createPicture(input: CreatePictureInput!): Picture!
  @aws_auth(cognito_groups: ["Admins"])
  createFriendship(id: ID!, target: ID!): Boolean
  @aws_auth(cognito_groups: ["Admins"])
}

type Query {
  getPicturesByOwner(id: ID!): [Picture]
  @aws_auth(cognito_groups: ["Admins", "Viewers"])
}

type Picture {
  id: ID!
  owner: ID!
  src: String
}

input CreatePictureInput {
  owner: ID!
```

```
src: String!  
}
```

選擇 Save Schema (儲存結構描述) 來儲存結構描述。

有些結構描述欄位已標註 @aws_auth 指令。由於 API 預設動作設定設為 DENY，所以 API 會拒絕不是 @aws_auth 指令提及群組成員的所有使用者。如需如何保護 API 的詳細資訊，您可以讀取 [Security \(安全性\)](#) 頁面。在這種情況下，只有管理使用者有權存取 Mutation.createPicture 和 Mutation.createFriendship 欄位，而屬於 Admins (管理員) 或 Viewers (瀏覽者) 群組之成員的使用者可以存取 Query.getPicturesByOwner 欄位。其他所有使用者都無存取權。

設定解析程式

現在，您已備妥有效的 GraphQL 結構描述和兩個資料來源，您可以將解析程式連接到結構描述上的 GraphQL 欄位。API 提供下列功能：

- 透過 Mutation.createPicture 欄位來建立圖片
- 透過 Mutation.createFriendship 欄位來建立朋友關係
- 透過 Query.getPicture 欄位來擷取圖片

Mutation.createPicture

從 AWS AppSync 主控台結構描述編輯器，選擇 的連接解析程式 createPicture(input: CreatePictureInput!): Picture!。選擇 DynamoDBPicturesDynamoDBTable PicturesDynamoDBTable 資料來源。在 request mapping template (要求映射範本) 區段中，新增下列範本：

```
#set($id = $util.autoId())  
  
{  
  "version" : "2018-05-29",  
  
  "operation" : "PutItem",  
  
  "key" : {  
    "id" : $util.dynamodb.toDynamoDBJson($id),  
    "owner": $util.dynamodb.toDynamoDBJson($ctx.args.input.owner)  
  },  
  
  "attributeValues" : $util.dynamodb.toMapValuesJson($ctx.args.input)
```

```
}
```

在 response mapping template (回應映射範本) 區段中，新增下列範本：

```
#if($ctx.error)
    $util.error($ctx.error.message, $ctx.error.type)
#end
$util.toJson($ctx.result)
```

建立圖片功能完成。您會使用隨機產生的 UUID 做為圖片 ID，並使用 Cognito 使用者名稱做為擁有者，將圖片儲存到 Pictures (圖片) 資料表。

Mutation.createFriendship

從 AWS AppSync 主控台結構描述編輯器，選擇 的連接解析程式 createFriendship(id: ID!, target: ID!): Boolean。選擇 DynamoDBFriendsDynamoDBTable FriendsDynamoDBTable 資料來源。在 request mapping template (要求映射範本) 區段中，新增下列範本：

```
#set($userToFriendFriendship = { "userId" : "$ctx.args.id", "friendId":
  "$ctx.args.target" })
#set($friendToUserFriendship = { "userId" : "$ctx.args.target", "friendId":
  "$ctx.args.id" })
#set($friendsItems = [$util.dynamodb.toMapValues($userToFriendFriendship),
  $util.dynamodb.toMapValues($friendToUserFriendship)])

{
  "version" : "2018-05-29",
  "operation" : "BatchPutItem",
  "tables" : {
    ## Replace 'AppSyncTutorial-' default below with the ResourceNamePrefix you
    provided in the CloudFormation template
    "AppSyncTutorial-Friends": $util.toJson($friendsItems)
  }
}
```

重要：在 BatchPutItem 要求範本中，應該要顯示 DynamoDB 資料表的完整名稱。預設資料表名稱為 AppSyncTutorial-Friends。如果您使用的資料表名稱不正確，則當 AppSync 嘗試擔任指定的角色時，將會發生錯誤。

為了讓本教學課程方便進行，請依照朋友關係要求已經核准的前提繼續執行，並將朋友關係項目直接存入 AppSyncTutorialFriends 資料表。

很快地，您就能為每個朋友關係儲存兩個項目，做為雙向關係。如需代表 many-to-many 關係的 Amazon DynamoDB 最佳實務的詳細資訊，請參閱 [DynamoDB 最佳實務](#)。

在 response mapping template (回應映射範本) 區段中，新增下列範本：

```
#if($ctx.error)
    $util.error($ctx.error.message, $ctx.error.type)
#end
true
```

注意：請確定您的要求範本包含適當的資料表名稱。預設名稱為 AppSyncTutorial-Friends，但如果您變更 CloudFormation ResourceNamePrefix 參數，則資料表名稱可能就會不同。

Query.getPicturesByOwner

現在您已經備妥朋友關係和圖片，您還需要提供讓使用者檢視其朋友圖片的權限。為了滿足這項要求，您必須先檢查申請者是內容擁有者的朋友，最後提供圖片查詢功能。

由於此功能需要兩個資料來源操作，所以您要建立兩個函數。第一種函數 isFriend 會檢查申請者和擁有者是否為朋友。第二個函數 getPicturesByOwner 會擷取具有擁有者 ID 的要求圖片。讓我們來看看下列對於 Query.getPicturesByOwner 欄位執行提議解析程式的執行流程：

1. Before 映射範本：準備內容和欄位輸入引數。
2. isFriend 函數：檢查申請者是否為圖片的擁有者。如果沒有，它會在好友資料表上執行 DynamoDB GetItem 操作，以檢查申請者和擁有者使用者是否為好友。
3. getPicturesByOwner 函數：使用擁有者索引全域次要索引上的 DynamoDB 查詢操作，從圖片資料表擷取圖片。
4. After 映射範本：映射圖片結果，讓 DynamoDB 屬性正確映射到預期的 GraphQL 類型欄位。

首先，讓我們來建立下列函數。

isFriend 函數

1. 選擇 Functions (函數) 索引標籤。
2. 選擇 Create Function (建立函數) 來建立函數。
3. 輸入 FriendsDynamoDBTable 做為資料來源名稱。
4. 對於函數名稱，輸入 isFriend。
5. 在要求映射範本文字區域中，貼上下列範本：

```
#set($ownerId = $ctx.prev.result.owner)
#set($callerId = $ctx.prev.result.callerId)

## if the owner is the caller, no need to make the check
#if($ownerId == $callerId)
    #return($ctx.prev.result)
#end

{
    "version" : "2018-05-29",

    "operation" : "GetItem",

    "key" : {
        "userId" : $util.dynamodb.toDynamoDBJson($callerId),
        "friendId" : $util.dynamodb.toDynamoDBJson($ownerId)
    }
}
```

6. 在回應映射範本文字區域中，貼上下列範本：

```
#if($ctx.error)
    $util.error("Unable to retrieve friend mapping message: ${ctx.error.message}",
    $ctx.error.type)
#end

## if the users aren't friends
#if(!$ctx.result)
    $util.unauthorized()
#end

$util.toJson($ctx.prev.result)
```

7. 選擇 Create Function (建立函數)。

結果：您已建立 isFriend 函數。

getPicturesByOwner 函數

1. 選擇 Functions (函數) 索引標籤。
2. 選擇 Create Function (建立函數) 來建立函數。

3. 輸入 PicturesDynamoDBTable 做為資料來源名稱。
4. 對於函數名稱，輸入 getPicturesByOwner。
5. 在要求映射範本文字區域中，貼上下列範本：

```
{
  "version" : "2018-05-29",

  "operation" : "Query",

  "query" : {
    "expression": "#owner = :owner",
    "expressionNames": {
      "#owner" : "owner"
    },
    "expressionValues" : {
      ":owner" : $util.dynamodb.toDynamoDBJson($ctx.prev.result.owner)
    }
  },

  "index": "owner-index"
}
```

6. 在回應映射範本文字區域中，貼上下列範本：

```
#if($ctx.error)
  $util.error($ctx.error.message, $ctx.error.type)
#end

$util.toJson($ctx.result)
```

7. 選擇 Create Function (建立函數)。

結果：您已建立 getPicturesByOwner 函數。現在這些函數已經建立完成，請將管道解析程式連接到 Query.getPicturesByOwner 欄位。

從 AWS AppSync 主控台結構描述編輯器，選擇 的連接解析程式 Query.getPicturesByOwner(id: ID!): [Picture]。在接下來的頁面，選擇會出現在資料來源下拉式清單中的 Convert to pipeline resolver (轉換成管道解析程式) 連結。使用下列程序來使用 Before 映射範本：

```
#set($result = { "owner": $ctx.args.id, "callerId": $ctx.identity.username })
```



```
$util.toJson($result)
```

在 after mapping template (After 映射範本) 區段中，使用下列範本：

```
#foreach($picture in $ctx.result.items)
  ## prepend "src://" to picture.src property
  #set($picture['src'] = "src://${picture['src']}")
#end
$util.toJson($ctx.result.items)
```

選擇 Create Resolver (建立解析程式)。您已成功連接第一個管道解析程式。在相同頁面上，新增您之前建立的兩個函數。在函數區段中，選擇 Add A Function (新增函數)，然後選擇或輸入第一個函數的名稱，isFriend 依照相同程序來處理 getPicturesByOwner 函數，新增第二個函數。確定清單中第一個顯示的是 isFriend 函數，後面是 getPicturesByOwner 函數。您可以使用上下箭頭，重新安排管道中的函數執行順序。

現在，管道解析程式已經建立完成，而且您已連接函數，讓我們來測試新建立的 GraphQL API。

測試您的 GraphQL API

首先，您必須透過您先前建立的管理員使用者，執行少量變動來填入圖片和朋友關係。在 AWS AppSync 主控台的左側，選擇查詢索引標籤。

createPicture Mutation

1. In AWS AppSync 主控台，選擇查詢索引標籤。
2. 選擇 Login With User Pools (登入使用者集區)。
3. 在模式中，輸入隨著 CloudFormation 堆疊建立的 Cognito Sample Client ID，例如，37solo6mmhh7k4v63cqdfgdg5d。
4. 輸入您以參數傳入 CloudFormation 堆疊的使用者名稱。預設為 nadia。
5. 使用透過電子郵件寄送，且由您提供做為 CloudFormation 堆疊 (例如，UserPoolUserEmail) 參數的臨時密碼。
6. 選擇 Login (登入)。您現在應該會看到此按鈕已重新命名為 Logout nadia，或是您在建立 CloudFormation 堆疊 (也就是 UserPoolUsername) 時所選擇的使用者名稱。

讓我們傳送幾個 createPicture 變動來填入該圖片資料表。在主控台中，執行以下 GraphQL 查詢：

```
mutation {  
  createPicture(input:{  
    owner: "nadia"  
    src: "nadia.jpg"  
  }) {  
    id  
    owner  
    src  
  }  
}
```

回應看起來應如下所示：

```
{  
  "data": {  
    "createPicture": {  
      "id": "c6fedbbe-57ad-4da3-860a-ffe8d039882a",  
      "owner": "nadia",  
      "src": "nadia.jpg"  
    }  
  }  
}
```

讓我們新增更多圖片：

```
mutation {  
  createPicture(input:{  
    owner: "shaggy"  
    src: "shaggy.jpg"  
  }) {  
    id  
    owner  
    src  
  }  
}
```

```
mutation {  
  createPicture(input:{  
    owner: "rex"  
    src: "rex.jpg"  
  }) {
```

```
    id
    owner
    src
  }
}
```

您已透過管理使用者的身分，使用 nadia 新增三張圖片。

createFriendship 變動

讓我們來新增朋友關係項目。在主控台中，執行下列變動。

注意：您仍必須登入管理使用者 (預設管理使用者是 nadia)。

```
mutation {
  createFriendship(id: "nadia", target: "shaggy")
}
```

回應看起來應該會像這樣：

```
{
  "data": {
    "createFriendship": true
  }
}
```

nadia 和 shaggy 是朋友。rex 不是任何人的朋友。

getPicturesByOwner 查詢

第一步是使用 Cognito 使用者集區，以及本教學課程一開始所設定的登入資料，登入成為 nadia 使用者。登入為 nadia 時，擷取 shaggy 擁有的圖片。

```
query {
  getPicturesByOwner(id: "shaggy") {
    id
    owner
    src
  }
}
```

由於 nadia 和 shaggy 是朋友，所以查詢應該會傳回相對應的圖片。

```
{
  "data": {
    "getPicturesByOwner": [
      {
        "id": "05a16fba-cc29-41ee-a8d5-4e791f4f1079",
        "owner": "shaggy",
        "src": "src://shaggy.jpg"
      }
    ]
  }
}
```

同樣地，嘗試擷取自己圖片的 nadia 也能順利擷取。為了避免在這種情況下執行 isFriend GetItem 操作，管道解析程式已完成相關的最佳化處理。嘗試下列查詢：

```
query {
  getPicturesByOwner(id: "nadia") {
    id
    owner
    src
  }
}
```

如果您啟用登入您的 API (在 Settings 窗格)，則當偵錯層級設定為 ALL (全部)，並再次執行相同查詢時，就會傳回欄位執行的日誌。透過查看日誌，您可以判斷 isFriend 函數是否在請求映射範本階段提早傳回：

```
{
  "errors": [],
  "mappingTemplateType": "Request Mapping",
  "path": "[getPicturesByOwner]",
  "resolverArn": "arn:aws:appsync:us-west-2:XXXX:apis/XXXX/types/Query/fields/getPicturesByOwner",
  "functionArn": "arn:aws:appsync:us-west-2:XXXX:apis/XXXX/functions/o2f42p2jrfdl3dw7s6xub2csdfs",
  "functionName": "isFriend",
  "earlyReturnedValue": {
    "owner": "nadia",
    "callerId": "nadia"
  },
}
```

```
"context": {
  "arguments": {
    "id": "nadia"
  },
  "prev": {
    "result": {
      "owner": "nadia",
      "callerId": "nadia"
    }
  },
  "stash": {},
  "outErrors": []
},
"fieldInError": false
}
```

earlyReturnedValue 索引鍵代表由 #return 指令傳回的資料。

最後，即使 rex 是 Viewers Cognito UserPool 群組的成員，但因為 rex 不是任何人的朋友，所以他無法存取 shaggy 或 nadia 所擁有的任何圖片。如果您以 rex 登入主控台，並執行下列查詢時：

```
query {
  getPicturesByOwner(id: "nadia") {
    id
    owner
    src
  }
}
```

您會收到下列關於未經授權的錯誤：

```
{
  "data": {
    "getPicturesByOwner": null
  },
  "errors": [
    {
      "path": [
        "getPicturesByOwner"
      ],
      "data": null,
      "errorType": "Unauthorized",
      "errorInfo": null,
    }
  ]
}
```

```
{
  "locations": [
    {
      "line": 2,
      "column": 9,
      "sourceName": null
    }
  ],
  "message": "Not Authorized to access getPicturesByOwner on type Query"
}
```

這表示您已成功使用管道解析程式，實作複雜的授權程序。

在 中的版本控制資料來源上使用 Delta Sync 操作 AWS AppSync

Note

我們現在主要支援 APPSYNC_JS 執行期及其文件。請考慮[在此處](#)使用 APPSYNC_JS 執行期及其指南。

用戶端應用程式 in AWS AppSync 會將 GraphQL 回應快取至本機行動/網路應用程式中的磁碟，以存放資料。已建立版本的資料來源和 Sync 操作可讓客戶使用單一解析程式來執行同步處理程序。透過這些查詢，用戶端可以使用從可能包含大量記錄的一個 base 查詢結果補充其本機快取，然後僅接收從上次查詢後有所更動的資料 (delta 更新)。藉由允許用戶端透過初始請求和其他請求的遞增性更新來執行快取的基本填入，所以您可將用戶端應用程式的計算工作移到後端。對於經常在上線和離線狀態之間切換的用戶端應用程式而言，這種做法可讓效率大幅提升。

若要實作 Delta Sync，Sync 查詢會在已建立版本的資料來源上使用 Sync 操作。當 an AWS AppSync 變動變更版本控制資料來源中的項目時，該變更的記錄也會存放在 Delta 資料表中。您可以選擇將不同的 Delta 資料表（例如，每種類型一個，每個網域區域一個）用於其他版本控制的資料來源，或將單一 Delta 資料表用於您的 API。AWS AppSync 建議不要將單一 Delta 資料表用於多個 APIs，以避免主要金鑰發生衝突。

此外，Delta Sync 用戶端也可以透過引數接收訂閱，然後用戶端協調訂閱和重新連線，並在離線和線上轉換之間進行寫入。Delta Sync 會自動恢復訂閱來執行這個程序，包括指數退避、透過不同的網路錯誤案例進行抖動，以及將事件存放在佇列中。接著會先執行適當的 delta 或 base 查詢，再合併佇列中的任何事件，最後依照一般方式處理訂閱。

[Amplify 架構網站](#)提供關於用戶端組態選項的文件，包括 Amplify DataStore。這份文件概述如何設定已建立版本的 DynamoDB 資料來源和 Sync 操作，以便與 Delta Sync 用戶端搭配使用，提供最佳的資料存取權。

一鍵設定

若要自動設定 GraphQL 端點 in AWS AppSync，並設定所有解析程式和必要的 AWS 資源，請使用此 AWS CloudFormation 範本：



這個堆疊會在您的帳戶中建立下列資源：

- 2 個 DynamoDB 資料表 (Base 和 Delta)
- 1 AWS AppSync API 與 API 金鑰
- 1 個具有 DynamoDB 資料表政策的 IAM 角色

兩個資料表會用來將同步查詢分割成第二個資料表，並做為用戶端離線時遺漏事件的日誌。為了讓 delta 資料表的查詢維持效率，您可以視需要使用 [Amazon DynamoDB TTLs](#) 來自動備妥這些事件。TTL 時間可根據您對資料來源的需求進行設定 (您可能希望將其設為 1 小時、1 天等)。

結構描述

為了示範 Delta Sync，範例應用程式會在 DynamoDB 中建立由 Base 和 Delta 資料表支援的 Posts 結構描述。AWS AppSync 會自動將變動寫入兩個資料表。同步查詢會依適當情況從 Base 或 Delta 資料表提取記錄，並且定義一份訂閱，說明用戶端如何透過其重新連線邏輯來發揮變動功能。

```
input CreatePostInput {
  author: String!
  title: String!
  content: String!
  url: String
  ups: Int
  downs: Int
  _version: Int
}

interface Connection {
  nextToken: String
```

```
    startedAt: AWSTimestamp!
  }

  type Mutation {
    createPost(input: CreatePostInput!): Post
    updatePost(input: UpdatePostInput!): Post
    deletePost(input: DeletePostInput!): Post
  }

  type Post {
    id: ID!
    author: String!
    title: String!
    content: String!
    url: AWSURL
    ups: Int
    downs: Int
    _version: Int
    _deleted: Boolean
    _lastChangedAt: AWSTimestamp!
  }

  type PostConnection implements Connection {
    items: [Post!]!
    nextToken: String
    startedAt: AWSTimestamp!
  }

  type Query {
    getPost(id: ID!): Post
    syncPosts(limit: Int, nextToken: String, lastSync: AWSTimestamp): PostConnection!
  }

  type Subscription {
    onCreatePost: Post
      @aws_subscribe(mutations: ["createPost"])
    onUpdatePost: Post
      @aws_subscribe(mutations: ["updatePost"])
    onDeletePost: Post
      @aws_subscribe(mutations: ["deletePost"])
  }

  input DeletePostInput {
    id: ID!
```



```
    _version: Int!
  }

  input UpdatePostInput {
    id: ID!
    author: String
    title: String
    content: String
    url: String
    ups: Int
    downs: Int
    _version: Int!
  }

  schema {
    query: Query
    mutation: Mutation
    subscription: Subscription
  }
```

這個 GraphQL 結構描述雖然是標準程序，但在繼續之前，有幾個事項值得我們提出說明。首先，所有變動會先自動寫入 Base 資料表，然後寫入 Delta 資料表。Base 資料表是狀態實況的中心來源，而 Delta 資料表是日誌記錄。如果您未傳入 `lastSync: AWSTimestamp`，則 `syncPosts` 查詢會針對基礎資料表執行，並補充快取，以及定期執行，做為用戶端離線時間超過 Delta 資料表中設定之 TTL 時間的邊緣案例的全域補齊程序。如果您傳入 `lastSync: AWSTimestamp`，`syncPosts` 查詢會執行處理您的 Delta 資料表，並由用戶端用來擷取自其上次離線後有所變更的事件。Amplify 用戶端會自動傳遞 `lastSync: AWSTimestamp` 值，並依適當情況儲存到磁碟中。

Post 上的 `_deleted` 欄位用於 DELETE 操作。當用戶端離線且記錄已從 Base 資料表移除時，此屬性會通知用戶端執行同步處理，將記錄自其本機快取中移除。如果用戶端離線時間較長，而且在用戶端可以配合 Delta Sync 查詢來擷取這個值之前，該項目已先移除，則 Base 查詢 (可在用戶端中設定) 的全域更新事件將會執行，並從快取中移除該項目。此欄位會標記為選用項目，因為只有在執行同步查詢而結果出現已刪除的項目時，才會傳回值。

變動

對於所有變動，AWS AppSync 會在基礎資料表中執行標準 Create/Update/Delete 操作，也會自動在 Delta 資料表中記錄變更。您可以藉由在資料來源中修改 `DeltaSyncTableTTL` 值，以縮短或延長記錄保留時間。對於擁有高速資料的組織，應該會考慮縮短這段時間。或者，如果您的用戶端離線時間較長，最好設定較長的資料保留時間。

同步查詢

基本查詢是未指定lastSync值的 DynamoDB Sync 操作。這種做法適合許多組織，因為 Base 查詢只會在一開始時執行，且後續是定期執行。

Delta 查詢是指定lastSync值的 DynamoDB Sync 操作。每當離線用戶端恢復上線，就會執行 delta 查詢 (只要 base 查詢定期時間未觸發執行)。用戶端會自動追蹤上次成功執行查詢以同步資料的時間。

執行 delta 查詢時，查詢的解析程式會使用 ds_pk 和 ds_sk 僅查詢自上次用戶端執行同步以來變更的記錄。用戶端會存放適當的 GraphQL 回應。

如需有關執行同步查詢的詳細資訊，請參閱[同步操作文件](#)。

範例

首先，藉由呼叫 createPost 變動來建立一個項目：

```
mutation create {
  createPost(input: {author: "Nadia", title: "My First Post", content: "Hello World"})
  {
    id
    author
    title
    content
    _version
    _lastChangedAt
    _deleted
  }
}
```

此變動的傳回值將如下所示：

```
{
  "data": {
    "createPost": {
      "id": "81d36bbb-1579-4efe-92b8-2e3f679f628b",
      "author": "Nadia",
      "title": "My First Post",
      "content": "Hello World",
      "_version": 1,
      "_lastChangedAt": 1574469356331,
    }
  }
}
```

```
    "_deleted": null
  }
}
```

如果您檢查 Base 資料表的內容，您會看到一筆記錄，如下所示：

```
{
  "_lastChangedAt": {
    "N": "1574469356331"
  },
  "_version": {
    "N": "1"
  },
  "author": {
    "S": "Nadia"
  },
  "content": {
    "S": "Hello World"
  },
  "id": {
    "S": "81d36bbb-1579-4efe-92b8-2e3f679f628b"
  },
  "title": {
    "S": "My First Post"
  }
}
```

如果您檢查 Delta 資料表的內容，您將會看見一筆記錄，如下所示：

```
{
  "_lastChangedAt": {
    "N": "1574469356331"
  },
  "_ttl": {
    "N": "1574472956"
  },
  "_version": {
    "N": "1"
  },
  "author": {
    "S": "Nadia"
  },
}
```

```
"content": {
  "S": "Hello World"
},
"ds_pk": {
  "S": "AppSync-delta-sync-post:2019-11-23"
},
"ds_sk": {
  "S": "00:35:56.331:81d36bbb-1579-4efe-92b8-2e3f679f628b:1"
},
"id": {
  "S": "81d36bbb-1579-4efe-92b8-2e3f679f628b"
},
"title": {
  "S": "My First Post"
}
}
```

現在，我們可以模擬 Base 查詢，客戶端將使用以下 `syncPosts` 查詢來執行該查詢以填入其本機資料存放區：

```
query baseQuery {
  syncPosts(limit: 100, lastSync: null, nextToken: null) {
    items {
      id
      author
      title
      content
      _version
      _lastChangedAt
    }
    startedAt
    nextToken
  }
}
```

此 Base 查詢的傳回值將如下所示：

```
{
  "data": {
    "syncPosts": {
      "items": [
        {
          "id": "81d36bbb-1579-4efe-92b8-2e3f679f628b",
```

```

        "author": "Nadia",
        "title": "My First Post",
        "content": "Hello World",
        "_version": 1,
        "_lastChangedAt": 1574469356331
      }
    ],
    "startedAt": 1574469602238,
    "nextToken": null
  }
}
}

```

稍後，我們將儲存 `startedAt` 值，以模擬 Delta 查詢，但需要先對資料表進行變更。讓我們使用 `updatePost` 變動以修改現有的 Post：

```

mutation updatePost {
  updatePost(input: {id: "81d36bbb-1579-4efe-92b8-2e3f679f628b", _version: 1, title:
    "Actually this is my Second Post"}) {
    id
    author
    title
    content
    _version
    _lastChangedAt
    _deleted
  }
}

```

此變動的傳回值將如下所示：

```

{
  "data": {
    "updatePost": {
      "id": "81d36bbb-1579-4efe-92b8-2e3f679f628b",
      "author": "Nadia",
      "title": "Actually this is my Second Post",
      "content": "Hello World",
      "_version": 2,
      "_lastChangedAt": 1574469851417,
      "_deleted": null
    }
  }
}

```

```
}
```

如果您現在檢查 Base 資料表的內容，您應該會看見更新的項目：

```
{
  "_lastChangedAt": {
    "N": "1574469851417"
  },
  "_version": {
    "N": "2"
  },
  "author": {
    "S": "Nadia"
  },
  "content": {
    "S": "Hello World"
  },
  "id": {
    "S": "81d36bbb-1579-4efe-92b8-2e3f679f628b"
  },
  "title": {
    "S": "Actually this is my Second Post"
  }
}
```

如果您現在檢查 Delta 資料表的內容，您應該會看到兩筆記錄：

1. 建立項目時的記錄
2. 更新項目時的記錄。

新項目如下所示：

```
{
  "_lastChangedAt": {
    "N": "1574469851417"
  },
  "_ttl": {
    "N": "1574473451"
  },
  "_version": {
    "N": "2"
  },
}
```

```

"author": {
  "S": "Nadia"
},
"content": {
  "S": "Hello World"
},
"ds_pk": {
  "S": "AppSync-delta-sync-post:2019-11-23"
},
"ds_sk": {
  "S": "00:44:11.417:81d36bbb-1579-4efe-92b8-2e3f679f628b:2"
},
"id": {
  "S": "81d36bbb-1579-4efe-92b8-2e3f679f628b"
},
"title": {
  "S": "Actually this is my Second Post"
}
}

```

現在，我們可以模擬 Delta 查詢，以擷取用戶端離線時發生的修改。我們將使用基本查詢傳回 `startedAt` 的值來提出請求：

```

query delta {
  syncPosts(limit: 100, lastSync: 1574469602238, nextToken: null) {
    items {
      id
      author
      title
      content
      _version
    }
    startedAt
    nextToken
  }
}

```

此 Delta 查詢的傳回值將如下所示：

```

{
  "data": {
    "syncPosts": {
      "items": [

```

```
{
  "id": "81d36bbb-1579-4efe-92b8-2e3f679f628b",
  "author": "Nadia",
  "title": "Actually this is my Second Post",
  "content": "Hello World",
  "_version": 2
},
"startedAt": 1574470400808,
"nextToken": null
}
```


中的組態和設定 AWS AppSync

AWS AppSync 可讓您：

- 快取經常請求但不太可能從請求變更為請求的資料。這可以減少解析程式的負載。如需詳細資訊，請參閱[the section called “設定伺服器端快取和 API 承載壓縮”](#)。
- 要處理和避免多個用戶端之間衝突的 GraphQL 物件版本。如需詳細資訊，請參閱[the section called “DynamoDB 的版本控制、衝突偵測和同步操作”](#)。
- 使用自訂網域名稱來設定單一、易記的網域，適用於您的 GraphQL 和即時 APIs。如需詳細資訊，請參閱[設定自訂網域名稱](#)。
- 允許透過 VPC 存取 GraphQL APIs。如需詳細資訊，請參閱[使用 AWS AppSync 私有 APIs](#)。
- 透過 整合分享您的 GraphQL APIs AWS Resource Access Manager。如需詳細資訊，請參閱[共用您的 AWS AppSync APIs](#)。
- 啟用自我檢查，並設定每個查詢的查詢深度和解析程式限制。如需詳細資訊，請參閱[組態限制](#)。
- 使用環境變數來調整 AWS AppSync 解析程式和函數的行為，而不更新程式碼。如需詳細資訊，請參閱[在 中使用環境變數 AWS AppSync](#)。

此外，AWS AppSync 包含下列用於記錄、監控和追蹤的標準 AWS 工具：

- [登入 AWS CloudTrail](#)
- [透過 Amazon CloudWatch 進行監控](#)
- [使用 追蹤 AWS X-Ray](#)

設定伺服器端快取和 API 承載壓縮 in AWS AppSync

AWS AppSync 的伺服器端資料快取功能可在高速記憶體內快取中提供資料，進而提升效能並減少延遲。這可減少直接存取資料來源的需求。快取適用於單元和管道解析程式。

AWS AppSync 也可讓您壓縮 API 回應，以便更快速地載入和下載承載內容。這可能會降低應用程式的壓力，也可能降低您的資料傳輸費用。壓縮行為是可設定的，您可以自行決定設定。

如需在 AWS AppSync API 中定義伺服器端快取和壓縮所需行為的說明，請參閱本節。

執行個體類型

AWS AppSync 在與 AWS AppSync API 相同的 AWS 帳戶和 AWS 區域中託管 Amazon ElastiCache (Redis OSS) 執行個體。

下列 ElastiCache (Redis OSS) 執行個體類型可供使用：

小型

1 個 vCPU、1.5 GiB RAM、低到中等網路效能

中型

2 個 vCPU、3 GiB RAM、低到中等網路效能

大型

2 個 vCPU、12.3 GiB RAM、高達 10 GB 的網路效能

xlarge

4 vCPU、25.05 GiB RAM、高達 10 GB 的網路效能

2xlarge

8 個 vCPU、50.47 GiB RAM、高達 10 GB 的網路效能

4xlarge

16 vCPU、101.38 GiB RAM、高達 10 GB 的網路效能

8xlarge

32 vCPU、203.26 GiB RAM、10 Gigabit 網路效能（並非所有區域都提供）

12xlarge

48 vCPU、317.77 GiB RAM、10 GB 網路效能

Note

過去，您會指定特定的執行個體類型（例如 `t2.medium`）。截至 2020 年 7 月，這些舊版執行個體類型仍然可用，但其使用已棄用且不建議。我們建議您使用此處所述的一般執行個體類型。

快取行為

以下是與快取相關的行為：

無

沒有伺服器端快取。

完整要求快取

完整請求快取是一種個別快取解析程式執行結果的機制。透過此設定，會 AWS AppSync 快取請求期間叫用的所有解析程式的執行，並個別快取每個解析程式。每個解析程式的資料會從其資料來源擷取，並填入快取，直到存留時間 (TTL) 過期為止。對於後續 API 請求，會從快取傳回每個特定解析程式的結果。這表示除非 TTL 已過期，否則不會直接聯絡資料來源。AWS AppSync 會使用的內容，`context.arguments` 並將 `context.identity` 映射為每個解析程式的快取金鑰。

每個解析程式快取

使用此設定，每個解析程式都必須明確選擇加入，才能快取回應。您可以在解析程式上指定 TTL 和快取金鑰。您可以指定的快取金鑰是來自這些映射的頂層映射 `context.arguments`、`context.source` 和 `context.identity` 和/或字串欄位。TTL 值為必填，但快取索引鍵是可選的。如果您未指定任何快取金鑰，則預設值為 `context.arguments`、`context.source` 和 `context.identity` 映射的內容。

例如，您可以使用下列組合：

- `context.arguments` 和 `context.source`
- `context.arguments` 和 `context.identity.sub`
- `context.arguments.id` 或 `context.arguments.InputType.id`
- `context.source.id` 和 `context.identity.sub`
- `context.identity.claims.username`

當您僅指定 TTL 且不指定快取金鑰時，解析程式的行為會與完整請求快取相同。

操作層級快取

操作層級快取會將整個 GraphQL 查詢操作回應儲存為整體。啟用時，成功的查詢回應會快取到其 TTL 過期，最大可快取回應大小為 15 MB。對於具有相同快取金鑰的後續查詢請求，回應將直接從快取提供，而不會在 TTL 尚未過期時執行任何解析程式。

操作層級快取的快取金鑰是使用下列組合產生：

- 來自請求 JSON 承載的特定屬性：
 - query 字串
 - operationName 字串
 - variables 地圖
- context.identity 映射 (IAM 和 Amazon Cognito 請求context.identity.sourceIp的除外)
- context.request.headers 映射 (不包括下一節中列出的特定預留標頭)

請求所使用的授權類型也會影響快取金鑰。對於 IAM 授權的請求，快取金鑰還會包含允許和拒絕的資源清單。對於 Lambda 授權的請求，快取金鑰還會包含遭拒欄位的清單。

快取金鑰會考慮 中找到的所有請求標頭context.request.headers，但下列預留標頭除外，這些標頭通常對特定請求是唯一的：

- authorization
- cloudfront-forwarded-proto
- cloudfront-is-desktop-viewer
- cloudfront-is-mobile-viewer
- cloudfront-is-smarttv-viewer
- cloudfront-is-tablet-viewer
- cloudfront-viewer-asn
- cloudfront-viewer-country
- content-length
- host
- priority
- sec-ch-ua
- sec-ch-ua-mobile
- sec-ch-ua-platform
- viax-amz-cf-id
- x-amz-date
- x-amz-security-token
- x-amzn-appsync-is-vpce-request

- x-amzn-remote-ip
- x-amzn-requestid
- x-amzn-trace-id
- x-forwarded-for

快取存留時間

此設定定義在記憶體中存放快取項目的時間量。TTL 上限為 3,600 秒 (1 小時)，之後項目會自動刪除。

快取加密

當您使用 AWS AppSync 伺服器端資料快取功能時，靜態和傳輸中的加密一律會針對新快取啟用，且無法停用。

若要在現有 API 快取上啟用加密，請刪除快取，然後重新建立快取。

若要使快取項目失效，您可以使用 AWS AppSync 主控台或 () 進行排清快取 API AWS Command Line Interface 呼叫 AWS CLI。

如需詳細資訊，請參閱 AWS AppSync API 參考中的 [ApiCache](#) 資料類型。

快取移出

當您設定 AWS AppSync 的伺服器端快取時，您可以設定最大 TTL。此值定義快取項目儲存在記憶體中的時間量。在您必須從快取移除特定項目的情況下，您可以在解析程式的請求或回應中使用 AWS AppSync 的 `evictFromApiCache` 延伸公用程式。(例如，當您資料來源中的資料變更，且您的快取項目現在已過時。)若要從快取中移出項目，您必須知道其金鑰。因此，如果您必須以動態方式移出項目，建議您使用每個解析程式快取，並明確定義要用來將項目新增至快取的金鑰。

移出快取項目

若要從快取中移出項目，請使用 `evictFromApiCache` 延伸項目公用程式。指定類型名稱和欄位名稱，然後提供索引鍵值項目的物件，以建置您要移出的項目索引鍵。在物件中，每個金鑰代表快取解析程式 `cachingKey` 清單中所用 `context` 物件的有效項目。每個值都是用來建構金鑰值的實際值。您必須以與快取解析程式 `cachingKey` 清單中快取金鑰相同的順序，將項目放在物件中。

例如，請參閱下列結構描述：

```
type Note {
  id: ID!
  title: String
  content: String!
}

type Query {
  getNote(id: ID!): Note
}

type Mutation {
  updateNote(id: ID!, content: String!): Note
}
```

在此範例中，您可以啟用每個解析程式快取，然後針對`getNote`查詢啟用它。然後，您可以將快取金鑰設定為包含 `[context.arguments.id]`。

當您嘗試取得 `Note`，若要建置快取金鑰，AWS AppSync 會使用`getNote`查詢的 `id` 引數在其伺服器端快取中執行查詢。

當您更新 `Note`，必須移出特定備註的項目，以確保下一個請求從後端資料來源擷取它。若要這樣做，您必須建立請求處理常式。

下列範例顯示使用此方法處理移出的一個方法：

```
import { util, Context } from '@aws-appsync/utils';
import { update } from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
  extensions.evictFromApiCache('Query', 'getNote', { 'ctx.args.id': ctx.args.id });
  return update({ key: { id: ctx.args.id }, update: { context: ctx.args.content } });
}

export const response = (ctx) => ctx.result;
```

或者，您也可以在此回應處理常式中處理移出。

處理`updateNote`變動時，AWS AppSync 會嘗試移出項目。如果項目已成功清除，回應會在 `extensions` 物件中包含一個`apiCacheEntriesDeleted`值，顯示已刪除的項目數量：

```
"extensions": { "apiCacheEntriesDeleted": 1}
```

根據身分移出快取項目

您可以根據 context 物件中的多個值建立快取金鑰。

例如，採用下列結構描述，使用 Amazon Cognito 使用者集區做為預設身分驗證模式，並由 Amazon DynamoDB 資料來源提供支援：

```
type Note {
  id: ID! # a slug; e.g.: "my-first-note-on-graphql"
  title: String
  content: String!
}

type Query {
  getNote(id: ID!): Note
}

type Mutation {
  updateNote(id: ID!, content: String!): Note
}
```

Note 物件類型會儲存在 DynamoDB 資料表中。資料表具有複合索引鍵，其使用 Amazon Cognito 使用者名稱做為主索引鍵，並使用 id(Slug) Note 做為分割區索引鍵。這是多租用戶系統，可讓多個使用者託管和更新其私有 Note 物件，而這些物件永遠不會共用。

由於這是一個讀取密集型系統，getNote 查詢會使用每個解析程式的快取來快取，快取金鑰由組成 [context.identity.username, context.arguments.id]。Note 更新時，您可以移出該特定的項目 Note。您必須以解析程式 cachingKeys 清單中指定的相同順序，在物件中新增元件。

下列範例顯示：

```
import { util, Context } from '@aws-appsync/utils';
import { update } from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
  extensions.evictFromApiCache('Query', 'getNote', {
    'ctx.identity.username': ctx.identity.username,
    'ctx.args.id': ctx.args.id,
  });
  return update({ key: { id: ctx.args.id }, update: { context: ctx.args.content } });
}
```

```
export const response = (ctx) => ctx.result;
```

後端系統也可以更新 Note 並移出項目。例如，採用此變動：

```
type Mutation {  
  updateNoteFromBackend(id: ID!, content: String!, username: ID!): Note @aws_iam  
}
```

您可以移出項目，但將快取金鑰的元件新增至 `cachingKeys` 物件。

在下列範例中，移出發生在解析程式的回應中：

```
import { util, Context } from '@aws-appsync/utils';  
import { update } from '@aws-appsync/utils/dynamodb';  
  
export function request(ctx) {  
  return update({ key: { id: ctx.args.id }, update: { context: ctx.args.content } });  
}  
  
export function response(ctx) {  
  extensions.evictFromApiCache('Query', 'getNote', {  
    'ctx.identity.username': ctx.args.username,  
    'ctx.args.id': ctx.args.id,  
  });  
  return ctx.result;  
}
```

如果您的後端資料已在 AWS AppSync 外部更新，您可以透過呼叫使用 `NONE` 資料來源的變動，從快取中移出項目。

壓縮 API 回應

AWS AppSync 允許用戶端請求壓縮的承載。如果請求，會壓縮並傳回 API 回應，以回應表示偏好壓縮內容的請求。壓縮的 API 回應載入速度更快、內容下載速度更快，而且您的資料傳輸費用也可能降低。

Note

壓縮可在 2020 年 6 月 1 日之後建立的所有新 APIs 上使用。

AWS AppSync 會盡力 [壓縮](#) 物件。在極少數情況下，AWS AppSync 可能會根據各種因素略過壓縮，包括目前的容量。

AWS AppSync 可以將 GraphQL 查詢承載大小壓縮在 1,000 到 10,000,000 個位元組之間。若要啟用壓縮，用戶端必須傳送具有值的 `Accept-Encoding` 標頭 `gzip`。您可以透過檢查回應中的 `Content-Encoding` 標頭值來驗證壓縮 (`gzip`)。

根據預設，AWS AppSync 主控台中的查詢瀏覽器會自動設定請求中的標頭值。如果您執行的查詢有足夠的回應，可以使用瀏覽器開發人員工具來確認壓縮。

設定 GraphQL 和即時 APIs 自訂網域名稱

透過 AWS AppSync，您可以使用自訂網域名稱來設定單一且值得記憶的網域，同時適用於 GraphQL 和即時 APIs。

換句話說，您可以建立與帳戶中 AWS 的 AppSync APIs 端點 URLs。

當您設定 an AWS AppSync API 時，會佈建兩個端點：

AWS AppSync GraphQL 端點：

```
https://example1234567890000.apps-sync-api.us-east-1.amazonaws.com/graphql
```

AWS AppSync 即時端點：

```
wss://example1234567890000.apps-sync-realtime-api.us-east-1.amazonaws.com/graphql
```

使用自訂網域名稱，您可以使用單一網域與兩個端點互動。例如，如果您將 `api.example.com` 設定為自訂網域，則可以使用這些 URLs 與 GraphQL 和即時端點互動：

AWS AppSync 自訂網域 GraphQL 端點：

```
https://api.example.com/graphql
```

AWS AppSync 自訂網域即時端點：

```
wss://api.example.com/graphql/realtime
```

Note

AWS AppSync APIs 僅支援自訂網域名稱的 TLS 1.2 和 TLS 1.3。

註冊和設定網域名稱

若要設定 your AWS AppSync APIs 的自訂網域名稱，您必須有已註冊的網際網路網域名稱。您可以使用 Amazon Route 53 domain registration 或您選擇的第三方網域註冊商來註冊網際網路網域。如需 Route 53 的詳細資訊，請參閱 [《Amazon Route 53 開發人員指南》中的什麼是 Amazon Route 53？](#)。

API 的自訂網域名稱可以是子網域的名稱或已註冊網際網路網域的根網域（也稱為「區域頂點」）。建立自訂網域名稱 in AWS AppSync 之後，您必須建立或更新 DNS 供應商的資源記錄，以映射至您的 API 端點。如果沒有此映射，則自訂網域名稱繫結的 API 請求無法連線 AWS AppSync。

在 AWS AppSync 中建立自訂網域名稱

為 an AWS AppSync API 建立自訂網域名稱會設定 Amazon CloudFront 分佈。您必須設定 DNS 記錄，將自訂網域名稱對應至 CloudFront 分佈網域名稱。需要此映射才能透過映射的 CloudFront 分佈路由綁定自訂網域名稱 AWS AppSync 的 API 請求。您也必須提供自訂網域名稱的憑證。

若要設定自訂網域名稱或更新其憑證，您必須擁有更新 CloudFront 分佈的許可，並描述您計劃使用的 AWS Certificate Manager (ACM) 憑證。若要授予這些許可，請將下列 AWS Identity and Access Management (IAM) 政策陳述式連接至您帳戶中的 IAM 使用者、群組或角色：

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "AllowUpdateDistributionForAppSyncCustomDomainName",
      "Effect": "Allow",
      "Action": ["cloudfront:updateDistribution"],
      "Resource": ["*"]
    },
    {
      "Sid": "AllowDescribeCertificateForAppSyncCustomDomainName",
      "Effect": "Allow",
      "Action": "acm:DescribeCertificate",
      "Resource": "arn:aws:acm:<region>:<account-id>:certificate/<certificate-id>"
    }
  ]
}
```

AWS AppSync 透過利用 CloudFront 分佈上的伺服器名稱指示 (SNI) 來支援自訂網域名稱。如需在 CloudFront 分佈上使用自訂網域名稱的詳細資訊，包括所需的憑證格式和憑證金鑰長度上限，請參閱《Amazon CloudFront 開發人員指南》中的[搭配使用 HTTPS 與 CloudFront](#)。Amazon CloudFront

若要將自訂網域名稱設定為 API 的主機名稱，API 擁有者必須在涵蓋自訂網域名稱 us-east-1AWS 的區域（美國東部（維吉尼亞北部））中提供公有或匯入的 ACM 憑證。如需 ACM 的詳細資訊，請參閱 AWS Certificate Manager 《使用者指南》中的[什麼是 AWS Certificate Manager？](#)。

AWS AppSync 中的萬用字元自訂網域名稱

AWS AppSync 支援萬用字元自訂網域名稱。若要設定萬用字元自訂網域名稱，請指定萬用字元 (*) 做為自訂網域的第一個子網域。這代表根網域的所有可能子網域。例如，萬用字元自訂網域名稱 *.example.com 會產生子網域 b.example.com，例如 a.example.com、和 c.example.com。所有這些子網域都會路由至相同的網域。

若要使用萬用字元自訂網域名稱 in AWS AppSync，您必須提供 ACM 發行的憑證，其中包含可以保護相同網域中多個網站的萬用字元名稱。如需詳細資訊，請參閱 AWS Certificate Manager 《使用者指南》中的[ACM 憑證特性](#)。

中的 DynamoDB 資料來源版本控制、衝突偵測和同步操作 AWS AppSync

AWS AppSync 的 DynamoDB 進階資料管理功能利用三個主要功能：版本控制的資料來源、衝突偵測和解決，以及同步操作。這些工具支援強大、可擴展的應用程式，可有效處理分散式環境中的並行資料修改和同步。

版本控制的資料來源是 AWS AppSync 資料管理系統的基礎。此功能會自動增強具有版本控制中繼資料的 DynamoDB 項目、記錄 AWS AppSync 變動至 Delta 資料表所做的變更，並維護已刪除項目的 "tombstones"。開發人員可以設定已刪除項目的保留期和變更日誌，最佳化儲存，同時確保資料一致性。版本控制的資料來源可簡化衝突偵測和同步操作的實作，為進階資料處理提供堅實的基礎。

發生並行寫入時，衝突偵測和解決機制會保護資料完整性。AWS AppSync 提供三種策略：最佳化並行、自動合併和 Lambda 型解析。樂觀並行拒絕衝突的變動，允許用戶端使用更新的資料重試。自動合併會根據資料類型、合併清單、執行設定聯集，以及保留現有的純量值，自動解決衝突。以 Lambda 為基礎的解析度可為複雜的衝突案例啟用自訂邏輯。這些選項可讓開發人員靈活處理資料衝突，確保分散式系統的一致性。

同步操作可在用戶端應用程式中實現有效的資料擷取和更新。此功能可讓用戶端從 DynamoDB 資料表擷取所有結果，然後僅擷取自上次查詢後變更的資料。AWS AppSync 會根據提供的同步字符來判斷是否存取 Base 資料表或 Delta 資料表、最佳化效能並減少資料傳輸。

主題

- [在中版本控制 DynamoDB 資料來源 AWS AppSync](#)
- [中的衝突偵測和解決 AWS AppSync](#)
- [在 中的版本控制資料來源上使用 DynamoDB 同步操作 AWS AppSync](#)

在中版本控制 DynamoDB 資料來源 AWS AppSync

AWS AppSync 目前支援 DynamoDB 資料來源上的版本控制。衝突偵測、衝突解決和同步作業需要 Versioned 資料來源。當您在資料來源上啟用版本控制時，AWS AppSync 會自動：

- 使用物件版本控制中繼資料增強項目。
- 記錄對 Delta 資料表具有 AWS AppSync 變動的項目所做的變更。
- 針對可設定的時間量，在具有「tombstone」的 Base 資料表中，維護已刪除的項目。

版本控制的資料來源組態

當您在 DynamoDB 資料來源上啟用版本控制時，請指定以下欄位：

BaseTableTTL

在具有「tombstone」的 Base 資料表中，保留已刪除項目的分鐘數 - 指出已刪除項目的中繼資料欄位。如果您想要在刪除項目時立即移除項目，可以將此值設為 0。此欄位為必填。

DeltaSyncTableName

以 AWS AppSync 變動儲存項目變更的資料表名稱。此欄位為必填。

DeltaSyncTableTTL

保留 Delta 資料表中項目的分鐘數。此欄位為必填。

Delta 同步資料表記錄

AWS AppSync 目前支援使用 PutItem、UpdateItem 和 DeleteItem DynamoDB 操作進行變動的 Delta Sync Logging。

當 an AWS AppSync 變動變更版本控制資料來源中的項目時，該變更的記錄會存放在針對增量更新最佳化的 Delta 資料表中。您可以選擇針對其他版本控制的資料來源使用不同的 Delta 資料表（例如，每種類型一個，每個網域區域一個），或針對您的 API 使用單一 Delta 資料表。AWS AppSync 建議不要針對多個 APIs 使用單一 Delta 資料表，以避免主要金鑰發生衝突。

此資料表所需的結構描述如下所示：

ds_pk

用來做為分割區索引鍵的字串值。其建構方式是串連基本資料來源名稱和變更發生日期的 ISO 8601 格式（例如 Comments:2019-01-01）。

當來自 VTL 映射範本的 customPartitionKey 旗標設定為分割區索引鍵的資料欄名稱時（請參閱 AWS AppSync 開發人員指南中的 [DynamoDB 解析程式映射範本參考](#)），ds_pk 變更格式和字串的建構方式是將分割區索引鍵的值附加到基礎資料表的新記錄中。例如，如果 Base 資料表中的記錄具有分割區索引鍵值 1a 和排序索引鍵值 2b，則字串的新值將為：Comments:2019-01-01:1a。

ds_sk

用來做為排序索引鍵的字串值。其建構方式是串連變更發生時間的 ISO 8601 格式、項目的主索引鍵，以及項目的版本。這些欄位的組合保證 Delta 資料表中每個項目的唯一性（例如，對於的時間 09:30:00 和的 ID 1a 和的版本 2，這會是 09:30:00:1a:2）。

當來自 VTL 映射範本的 customPartitionKey 旗標設定為分割區索引鍵的資料欄名稱時（請參閱 AWS AppSync 開發人員指南中的 [DynamoDB 解析程式映射範本參考](#)），ds_sk 變更格式和字串的建構方式是將組合索引鍵的值取代為基礎資料表中排序索引鍵的值。使用上述範例，如果 Base 資料表中的記錄具有分割區索引鍵值 1a 和排序索引鍵值 2b，則字串的新值將為：09:30:00:2b:3。

_ttl

此數值用於存放應該從 Delta 資料表中移除項目時的時間戳記（以 epoch 秒為單位）。此值是透過將資料來源上設定的 DeltaSyncTableTTL 值加入發生變更的時刻所決定的。此欄位應該設定為 DynamoDB TTL 屬性。

設定為與 Base 資料表搭配使用的 IAM 角色必須也包含在 Delta 資料表上進行操作的許可。在此範例中，ChangeLog 會顯示名為 的基本資料表 Comments 和名為 的 Delta 資料表的許可政策：

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "dynamodb:DeleteItem",
        "dynamodb:GetItem",
        "dynamodb:PutItem",
        "dynamodb:Query",
        "dynamodb:Scan",
        "dynamodb:UpdateItem"
      ],
      "Resource": [
        "arn:aws:dynamodb:us-east-1:000000000000:table/Comments",
        "arn:aws:dynamodb:us-east-1:000000000000:table/Comments/*",
        "arn:aws:dynamodb:us-east-1:000000000000:table/ChangeLog",
        "arn:aws:dynamodb:us-east-1:000000000000:table/ChangeLog/*"
      ]
    }
  ]
}
```

版本控制的資料來源中繼資料

AWS AppSync 會代表您管理 Versioned 資料來源上的中繼資料欄位。自行修改這些欄位可能會導致應用程式發生錯誤或資料遺失。這些欄位包括：

`_version`

依序遞增的計數器，每當對項目進行變更時都會更新。

`_lastChangedAt`

此數值用於存放上次修改項目時的時間戳記 (以 epoch 毫秒為單位)。

`_deleted`

布林「tombstone」值，表示已刪除項目。應用程式可以使用此值，從本機資料存放區撤出已刪除的項目。

_ttl

此數值用於存放應該從基礎資料來源中移除項目時的時間戳記 (以 epoch 秒為單位)。

ds_pk

用來做為 Delta 資料表之分割區索引鍵的字串值。

ds_sk

用來做為 Delta 資料表之排序索引鍵的字串值。

gsi_ds_pk

產生的字串值屬性，以支援全域次要索引做為分割區索引鍵。只有在 VTL 映射範本中同時啟用 `customPartitionKey` 和 `populateIndexFields` 旗標時，才會包含它（請參閱AWS AppSync 開發人員指南》中的 [DynamoDB 解析程式映射範本參考](#)）。如果啟用，值將透過串連基礎資料來源名稱和變更發生日期的 ISO 8601 格式來建構（例如，如果基礎資料表名為 `Comments`，則此記錄將設定為 `Comments:2019-01-01`）。

gsi_ds_sk

產生的字串值屬性，支援全域次要索引做為排序索引鍵。只有在 VTL 映射範本中同時啟用 `customPartitionKey` 和 `populateIndexFields` 旗標時，才會包含它（請參閱AWS AppSync 開發人員指南》中的 [DynamoDB 解析程式映射範本參考](#)）。如果啟用，值的建構方式是串連發生變更的時間的 ISO 8601 格式、基礎資料表中項目的分割區索引鍵、基礎資料表中項目的排序索引鍵，以及項目的版本（例如 的時間09:30:00、分割區索引鍵值 1a、排序索引鍵值 2b和 版本 3，這會是 09:30:00:1a#2b:3）。

這些中繼資料欄位會影響基礎資料來源中項目的整體大小。AWS AppSync 建議在設計應用程式時為版本控制資料來源中繼資料保留 500 個位元組 + 最大主要金鑰大小的儲存體。若要在用戶端應用程式中使用此中繼資料，請在 GraphQL 類型和變動選取範圍中包含 `_version`、`_lastChangedAt` 和 `_deleted` 欄位。

中的衝突偵測和解決 AWS AppSync

當 AWS AppSync 發生並行寫入時，您可以設定衝突偵測和衝突解決策略，以適當處理更新。「衝突偵測」會判斷變動是否與資料來源中的實際寫入項目衝突。藉由將 `conflictDetection` 欄位之 `SyncConfig` 中的值設為 `VERSION`，即可啟用「衝突偵測」。

「衝突解決」是偵測到衝突時所採取的動作。這是透過在 `SyncConfig` 中設定「Conflict Handler (衝突處理常式)」欄位來決定的。有三種衝突解決策略：

- OPTIMISTIC_CONCURRENCY
- AUTOMERGE
- LAMBDA

在寫入操作期間，AWS AppSync 會自動遞增版本，且用戶端或設定為已啟用版本之資料來源的解析程式外部不應修改版本。這樣做會改變系統的一致性行為，並可能造成資料遺失。

樂觀並行

樂觀並行是 AWS AppSync 為版本控制資料來源提供的衝突解決策略。當衝突解析程式設為「開放式並行存取」時，如果偵測到傳入變動的版本與物件的實際版本不同，衝突處理常式將僅拒絕傳入請求。在 GraphQL 回應中，將提供最新版本伺服器上的現有項目。然後，預期用戶端在本機處理此衝突，並使用該項目的更新版本重試變動。

自動合併

自動合併提供開發人員一種簡單的方法來設定衝突解決策略，而不需撰寫用戶端邏輯以手動合併其他策略無法處理的衝突。自動合併在合併資料來解決衝突時，遵守嚴格的規則集。自動合併的原則是以 GraphQL 欄位的基礎資料類型為主。如下所示：

- 純量欄位上的衝突：GraphQL 純量或非集合的任何欄位（即 List、Set、Map）。拒絕純量欄位的傳入值，並選取伺服器中的現有值。
- List 的衝突：GraphQL 類型和資料庫類型為 List。串連傳入清單與伺服器中的現有清單。傳入變動的清單值將附加到伺服器列表的末尾。將保留重複的值。
- Set 的衝突：GraphQL 類型為 List，資料庫類型為 Set。使用傳入集合和伺服器中的現有集合套用集合聯集。這符合 Set 的屬性，表示沒有重複的項目。
- 當傳入變動將新欄位新增至項目，或針對值為 `null` 的欄位進行時，請將該欄位合併到現有項目。
- Map 的衝突：當資料庫中的基礎資料類型是 Map (即索引鍵-值文件) 時，會套用上述規則，因為其剖析和處理 Map 的每個屬性。

自動合併是為了自動偵測、合併和重試具有更新版本的請求而設計，讓用戶端無需手動合併任何衝突的資料。

為了展現自動合併如何處理純量類型衝突的範例。我們將使用以下記錄做為起點。

```
{
  "id" : 1,
```



```
"name" : "Nadia",
"jersey" : 5,
"_version" : 4
}
```

現在，由於用戶端尚未與伺服器同步，傳入變動可能試圖使用舊版本來更新該項目。如下所示：

```
{
  "id" : 1,
  "name" : "Nadia",
  "jersey" : 55,
  "_version" : 2
}
```

請注意，傳入請求中的過時版本 2。在此流程中，自動合併拒絕將 'jersey' 欄位更新為 '55'，並保留值 '5'，導致以下項目影像儲存在伺服器中。

```
{
  "id" : 1,
  "name" : "Nadia",
  "jersey" : 5,
  "_version" : 5 # version is incremented every time automerge performs a merge that is
  stored on the server.
}
```

假設如上所示的項目狀態為版本 5，現在假設傳入變動嘗試使用以下影像對項目進行變動：

```
{
  "id" : 1,
  "name" : "Shaggy",
  "jersey" : 5,
  "interests" : ["breakfast", "lunch", "dinner"] # underlying data type is a Set
  "points": [24, 30, 27] # underlying data type is a List
  "_version" : 3
}
```

傳入變動中有三個參考資訊。名稱 (純量) 已變更，但增加兩個新欄位，“interests” (Set) 和 “points” (List)。在這種情況下，因為版本不符，會偵測到衝突。自動合併遵循其屬性，並拒絕名稱變更，因為它是純量，而且是非衝突的欄位。這將導致儲存在伺服器中的項目顯示如下。

```
{
```

```
"id" : 1,
"name" : "Nadia",
"jersey" : 5,
"interests" : ["breakfast", "lunch", "dinner"] # underlying data type is a Set
"points": [24, 30, 27] # underlying data type is a List
"_version" : 6
}
```

現在，使用版本 6 的項目更新影像，現在假設傳入變動 (具有另一個不相符的版本) 嘗試將項目轉換為以下內容：

```
{
  "id" : 1,
  "name" : "Nadia",
  "jersey" : 5,
  "interests" : ["breakfast", "lunch", "brunch"] # underlying data type is a Set
  "points": [30, 35] # underlying data type is a List
  "_version" : 5
}
```

我們在這裡觀察到“interests”的傳入欄位具有伺服器中的一個重複值和兩個新值。在這種情況下，由於基礎資料類型是 Set，所以自動合併會將伺服器中的現有值與傳入請求中的值合併，並刪除任何重複項目。同樣地，“points”欄位也有衝突，有一個重複值和一個新值。但由於這裡的基礎資料類型是 List，自動合併會簡單地將傳入請求中的所有值附加到伺服器中已存在值的結尾。存放在伺服器上的產生合併影像顯示如下：

```
{
  "id" : 1,
  "name" : "Nadia",
  "jersey" : 5,
  "interests" : ["breakfast", "lunch", "dinner", "brunch"] # underlying data type is a Set
  "points": [24, 30, 27, 30, 35] # underlying data type is a List
  "_version" : 7
}
```

現在，假設存放在伺服器中的項目在版本 8 中如下所示。

```
{
  "id" : 1,
  "name" : "Nadia",
```

```
"jersey" : 5,
"interests" : ["breakfast", "lunch", "dinner", "brunch"] # underlying data type is a Set
"points": [24, 30, 27, 30, 35] # underlying data type is a List
"stats": {
  "ppg": "35.4",
  "apg": "6.3"
}
"_version" : 8
}
```

但傳入的請求嘗試使用以下影像來更新項目，再次發生版本不相符：

```
{
  "id" : 1,
  "name" : "Nadia",
  "stats": {
    "ppg": "25.7",
    "rpg": "6.9"
  }
  "_version" : 3
}
```

在這種情況下，我們可以看到伺服器中已存在的欄位遺失 (interests、points、jersey)。此外，已編輯映射 “stats” 中的 “ppg” 值，已新增新值 “rpg”，而且已省略 “apg”。自動合併會保留已省略的欄位 (注意：如果要移除欄位，必須使用相符的版本再次嘗試該請求)，以便不會遺失欄位。它也會將相同的規則套用至映射中的欄位，因此將拒絕對 “ppg” 進行的變更，而保留 “apg”，並新增欄位 “rpg”。存放在伺服器中的產生項目現在顯示如下：

```
{
  "id" : 1,
  "name" : "Nadia",
  "jersey" : 5,
  "interests" : ["breakfast", "lunch", "dinner", "brunch"] # underlying data type is a Set
  "points": [24, 30, 27, 30, 35] # underlying data type is a List
  "stats": {
    "ppg": "35.4",
    "apg": "6.3",
    "rpg": "6.9"
  }
  "_version" : 9
}
```

```
}
```

Lambdas

有多種 Lambda 解析策略可供選擇：

- **RESOLVE**：將現有項目取代為回應承載中提供的新項目。您一次只能對單一項目重試相同的操作。目前支援 DynamoDB PutItem 和 UpdateItem。
- **REJECT**：拒絕變動，並傳回 GraphQL 回應中現有項目的錯誤。目前支援 DynamoDB PutItem、UpdateItem 和 DeleteItem。
- **REMOVE**：移除現有項目。目前支援 DynamoDB DeleteItem。

Lambda 叫用請求

The AWS AppSync DynamoDB 解析程式會叫用 中指定的 Lambda 函數 LambdaConflictHandlerArn。它會使用資料來源上所設定的相同 service-role-arn。叫用承載的結構如下：

```
{
  "newItem": { ... },
  "existingItem": {... },
  "arguments": { ... },
  "resolver": { ... },
  "identity": { ... }
}
```

欄位定義如下：

newItem

預覽項目，如果變動成功。

existingItem

目前位於 DynamoDB 資料表中的項目。

arguments

來自 GraphQL 變動的引數。

resolver

AWS AppSync 解析程式的相關資訊。

identity

發起人的相關資訊。如果使用 API 金鑰存取，則此欄位會設為 Null。

承載範例：

```
{
  "newItem": {
    "id": "1",
    "author": "Jeff",
    "title": "Foo Bar",
    "rating": 5,
    "comments": ["hello world"],
  },
  "existingItem": {
    "id": "1",
    "author": "Foo",
    "rating": 5,
    "comments": ["old comment"]
  },
  "arguments": {
    "id": "1",
    "author": "Jeff",
    "title": "Foo Bar",
    "comments": ["hello world"]
  },
  "resolver": {
    "tableName": "post-table",
    "awsRegion": "us-west-2",
    "parentType": "Mutation",
    "field": "updatePost"
  },
  "identity": {
    "accountId": "123456789012",
    "sourceIp": "x.x.x.x",
    "username": "AIDAAAAAAAAAAAAAAAAAAAA",
    "userArn": "arn:aws:iam::123456789012:user/appsync"
  }
}
```

Lambda 叫用回應

適用於 PutItem 和 UpdateItem 衝突解決

RESOLVE 變動。回應必須採用下列格式。

```
{
  "action": "RESOLVE",
  "item": { ... }
}
```

item 欄位代表將用來取代基礎資料來源中現有項目的物件。如果包含在 item 中，則會忽略主索引鍵和同步中繼資料。

REJECT 變動。回應必須採用下列格式。

```
{
  "action": "REJECT"
}
```

適用於 DeleteItem 衝突解決

REMOVE 項目。回應必須採用下列格式。

```
{
  "action": "REMOVE"
}
```

REJECT 變動。回應必須採用下列格式。

```
{
  "action": "REJECT"
}
```

以下 Lambda 函數範例會檢查進行呼叫的對象和解析程式名稱。如果是由 jeffTheAdmin 產生，則會 REMOVE DeletePost 解析程式的物件，或針對「更新/放置」解析程式 RESOLVE 與新項目的衝突。如果不是，則變動為 REJECT。

```
exports.handler = async (event, context, callback) => {
  console.log("Event: " + JSON.stringify(event));

  // Business logic goes here.
  var response;
  if ( event.identity.user == "jeffTheAdmin" ) {
```

```
let resolver = event.resolver.field;

switch(resolver) {
  case "deletePost":
    response = {
      "action" : "REMOVE"
    }
    break;

  case "updatePost":
  case "createPost":
    response = {
      "action" : "RESOLVE",
      "item": event.newItem
    }
    break;
  default:
    response = { "action" : "REJECT" };
}
} else {
  response = { "action" : "REJECT" };
}

console.log("Response: " + JSON.stringify(response));
return response;
}
```

錯誤

以下是衝突解決程序期間可能發生的錯誤清單：

ConflictUnhandled

衝突偵測發現版本不相符，而且衝突處理常式拒絕變動。

範例：具有開放式並行存取衝突處理常式的衝突解決機制。或者，Lambda 衝突處理常式傳回 REJECT。

ConflictError

嘗試解決衝突時，發生內部錯誤。

範例：Lambda 衝突處理常式傳回格式錯誤的回應。或者，無法叫用 Lambda 衝突處理常式，因為找不到提供的資源 LambdaConflictHandlerArn。

MaxConflicts

已達到衝突解決的重試次數上限。

範例：同一個物件上有太多並行請求。解決衝突之前，另一個用戶端已將物件更新為新版本。

BadRequest

用戶端嘗試更新中繼資料欄位 (`_version`、`_ttl`、`_lastChangedAt`、`_deleted`)。

範例：用戶端嘗試更新具有更新變動 `_version` 的物件。

DeltaSyncWriteError

無法寫入差異同步記錄。

範例：變動成功，但嘗試寫入差異同步資料表時發生內部錯誤。

InternalFailure

發生內部錯誤。

UnsupportedOperation

不支援的操作 '`X`'。資料來源版本控制僅支援下列操作

(`TransactGetItems`、`PutItem`、`BatchGetItem`、`Scan`、`Query`、`GetItem`、`DeleteItem`、`UpdateItem`、`Sync`)

範例：在啟用衝突偵測/解決的情況下使用特定交易和批次操作。目前不支援這些操作。

CloudWatch Logs

如果 an AWS AppSync API 已啟用 CloudWatch Logs，並將記錄設定為欄位層級日誌 `enabled`，並將欄位層級日誌的日誌層級設定為 `ALL`，則 AWS AppSync 會將衝突偵測和解決資訊發出至日誌群組。如需日誌訊息格式的相關資訊，請參閱[衝突偵測與同步記錄的文件](#)。

在 中的版本控制資料來源上使用 DynamoDB 同步操作 AWS AppSync

版本控制的資料來源支援 `Sync` 操作，可讓您從 DynamoDB 資料表擷取所有結果，然後只接收自上次查詢後更改的資料（差異更新）。當 AWS AppSync 收到 `Sync` 操作的請求時，它會使用請求中指定的欄位來判斷是否應存取 Base 資料表或 Delta 資料表。

- 如果未指定 `lastSync` 欄位，則會在基礎資料表 `Scan` 上執行。

- 如果指定 `lastSync` 欄位，但值在 之前 `current moment - DeltaSyncTTL`，則會在基礎資料表 `Scan` 上執行。
- 如果指定 `lastSync` 欄位，且值在 或之後 `current moment - DeltaSyncTTL`，則會在 `Delta` 資料表 `Query` 上執行。

AWS AppSync 會將 `startedAt` 欄位傳回至所有 Sync 操作的回應映射範本。`startedAt` 欄位是開始進行 Sync 操作時，可以在本機存放並在另一個請求中使用的時間 (以 epoch 毫秒為單位)。如果請求中包含分頁字符，則該值將與請求針對第一頁結果傳回的值相同。

如需 Sync 映射範本的相關資訊，請參閱[映射範本參考](#)。

使用 CloudWatch 監控和記錄 GraphQL API 資料

您可以使用 CloudWatch 指標和 CloudWatch 日誌來記錄和偵錯 GraphQL API。這些工具可讓開發人員有效地監控效能、疑難排解問題，並最佳化其 GraphQL 操作。

CloudWatch 指標是一種工具，提供廣泛的指標來監控 API 效能和用量。這些指標分為兩個主要類別：

1. 一般 API 指標：這些指標包括 `4XXError`、`5XXError` 用於追蹤用戶端和伺服器錯誤、`Latency` 用於測量回應時間、`Requests` 用於監控總 API 呼叫，以及 `TokensConsumed` 用於追蹤資源用量的和。
2. 即時訂閱指標：這些指標著重於 WebSocket 連線和訂閱活動。其中包括連線請求、成功連線、訂閱註冊、訊息發佈以及作用中連線和訂閱的指標。

本指南也推出增強型指標，提供更精細的解析程式效能、資料來源互動和個別 GraphQL 操作資料。這些指標提供更深入的洞見，但會產生額外的成本。

CloudWatch Logs 是一種工具，可為您的 GraphQL APIs 啟用記錄功能。日誌可以設定為 API 的兩個層級：

1. 請求層級日誌：這些擷取整體請求資訊，包括 HTTP 標頭、GraphQL 查詢、操作摘要和訂閱註冊。
2. 欄位層級日誌：這些提供個別欄位解析的詳細資訊，包括請求和回應映射，以及每個欄位的追蹤資訊。

您可以設定記錄、解譯日誌項目，並使用日誌資料進行故障診斷和最佳化。AWS AppSync 提供各種日誌類型，可顯示查詢的執行、剖析、驗證和欄位解析資料。

設定和組態

若要在 GraphQL API 上開啟自動記錄，請使用 AWS AppSync 主控台。

1. 登入 AWS Management Console 並開啟 [AppSync 主控台](#)。
2. 在 APIs 頁面上，選擇 GraphQL API 的名稱。
3. 在 API 首頁的導覽窗格中，選擇設定。
4. 在 Logging (記錄)，執行以下項目：
 - a. 開啟啟用日誌。
 - b. 如需詳細的請求層級記錄，請選取包含詳細內容下的核取方塊。(選用)
 - c. 在欄位解析程式日誌層級下，選擇您偏好的欄位層級記錄層級 (無、錯誤、資訊、偵錯或全部)。(選用)
 - d. 在建立或使用現有角色下，選擇新角色以建立新的 AWS Identity and Access Management (IAM)，允許 AWS AppSync 將日誌寫入 CloudWatch。或者，選擇現有角色以選取 AWS 帳戶中現有 IAM 角色的 Amazon Resource Name (ARN)。
5. 選擇儲存。

手動 IAM 角色組態

如果您選擇使用現有的 IAM 角色，該角色必須授予 AWS AppSync 將日誌寫入 CloudWatch 所需的許可。若要手動設定，您必須提供服務角色 ARN，讓 AWS AppSync 可以在撰寫日誌時擔任該角色。

在 [IAM 主控台](#) 中，使用 `AWSAppSyncPushToCloudWatchLogsPolicy` 具有下列定義的名稱建立新的政策：

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "logs:CreateLogGroup",
        "logs:CreateLogStream",
        "logs:PutLogEvents"
      ]
    }
  ]
}
```

```
        "Resource": "*"
      }
    ]
  }
}
```

接著，建立名為 `AWSAppSyncPushToCloudWatchLogsRole` 的新角色，並將新建立的政策連接至角色。將此角色的信任關係編輯如下：

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "appsync.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

複製角色 ARN，並在設定 an AWS AppSync GraphQL API 的記錄時使用它。

CloudWatch 指標

您可以使用 CloudWatch 指標來監控並提供有關可能導致 HTTP 狀態碼或延遲之特定事件的提醒。會發出下列指標：

指標清單

4XXError

由於不正確的用戶端組態而使請求無效所產生的錯誤。一般而言，這些錯誤會在 GraphQL 處理之外的任何位置發生。例如，當請求包含不正確的 JSON 承載或不正確的查詢、服務受到調節，或授權設定設定錯誤時，可能會發生這些錯誤。

單位：計數。使用總和統計以取得這些錯誤的總次數。

5XXError

執行 GraphQL 查詢期間遇到的錯誤。例如，在針對空白或不正確的結構描述叫用查詢時，可能會發生這種情況。當 Amazon Cognito 使用者集區 ID 或 AWS 區域無效時，也可能發生這種情況。或者，如果 AWS AppSync 在處理請求期間遇到問題，也可能發生這種情況。

單位：計數。使用總和統計以取得這些錯誤的總次數。

Latency

當 AWS AppSync 從用戶端接收請求，以及傳回回應給用戶端之間的時間。這不包含回應到達最終裝置所發生的網路延遲。

單位：毫秒。使用平均統計資料以評估預期的延遲。

Requests

您帳戶中所有 APIs 已處理的請求（查詢 + 變動）數量，依區域排序。

單位：計數。在特定區域中處理的所有請求數量。

TokensConsumed

權杖Requests會根據 Request 消耗的資源量（使用的處理時間和記憶體）配置給。通常，每個都會Request使用一個字符。不過，Request消耗大量資源的 會視需要配置額外的字符。

單位：計數。配置給在特定區域中處理之請求的字符數量。

NetworkBandwidthOutAllowanceExceeded

Note

在 AWS AppSync 主控台的快取設定頁面上，快取運作狀態指標選項可讓您啟用此快取相關運作狀態指標。

由於輸送量超過彙總頻寬限制而捨棄的網路封包。這對於診斷快取組態中的瓶頸很有用。在 指標API_Id中指定，以記錄特定 API appsyncCacheNetworkBandwidthOutAllowanceExceeded 的資料。

單位：計數。超過 ID 所指定 API 頻寬限制後捨棄的封包數量。

EngineCPUUtilization

Note

在 AWS AppSync 主控台的快取設定頁面上，快取運作狀態指標選項可讓您啟用此快取相關運作狀態指標。

配置給 Redis OSS 程序的 CPU 使用率（百分比）。這對於診斷快取組態中的瓶頸很有用。在指標 `API_Id` 中指定，以記錄特定 API `appsyncCacheEngineCPUUtilization` 的資料。

單位：百分比。Redis OSS 程序目前針對 ID 指定的 API 使用的 CPU 百分比。

即時訂閱

所有指標都會在一個維度中發出：`GraphQLAPIId`。這意味著所有指標都與 GraphQL API ID 相結合。下列指標與透過純 WebSockets 的 GraphQL 訂閱相關：

指標清單

`ConnectRequests`

對提出的 WebSocket 連線請求數量 AWS AppSync，包括成功和失敗的嘗試。

單位：計數。使用總和統計資料來取得連線請求的總數。

`ConnectSuccess`

成功連線至 AWS AppSync 的 WebSocket 數目。您可以在沒有訂閱的情況下連線。

單位：計數。使用總和統計以取得這些成功連線的總次數。

`ConnectClientError`

由於用戶端錯誤，AWS AppSync 拒絕的 WebSocket 連線數目。這可能表示服務已調節，或授權設定設定錯誤。

單位：計數。使用總和統計以取得這些用戶端連線錯誤的總次數。

`ConnectServerError`

處理連線時源自 AWS AppSync 的錯誤數目。發生未預期的伺服器端問題時，通常會發生這種情況。

單位：計數。使用總和統計以取得這些伺服器端連線錯誤的總次數。

DisconnectSuccess

與 AWS AppSync 成功中斷 WebSocket 連線的次數。

單位：計數。使用總和統計以取得這些成功中斷連線的總次數。

DisconnectClientError

中斷 WebSocket 連線時，源自 AWS AppSync 的用戶端錯誤數目。

單位：計數。使用總和統計以取得這些中斷連線錯誤的總次數。

DisconnectServerError

中斷 WebSocket 連線時，源自 AWS AppSync 的伺服器錯誤數量。

單位：計數。使用總和統計以取得這些中斷連線錯誤的總次數。

SubscribeSuccess

透過 WebSocket 成功註冊至 AWS AppSync 的訂閱數量。沒有訂閱的連線是有可能的，但沒有連線的訂閱是不可能的。

單位：計數。使用總和統計以取得這些成功訂閱的總次數。

SubscribeClientError

由於用戶端錯誤而遭 AWS AppSync 拒絕的訂閱數量。當 JSON 承載不正確、服務受到調節或授權設定錯誤時，就會發生這種情況。

單位：計數。使用總和統計以取得這些用戶端訂閱錯誤的總次數。

SubscribeServerError

處理訂閱時源自 AWS AppSync 的錯誤數目。發生未預期的伺服器端問題時，通常會發生這種情況。

單位：計數。使用總和統計以取得這些伺服器端訂閱錯誤的總次數。

UnsubscribeSuccess

已成功處理的取消訂閱請求數量。

單位：計數。使用總和統計資料來取得成功取消訂閱請求的總次數。

UnsubscribeClientError

由於用戶端錯誤，AWS AppSync 拒絕的取消訂閱請求數量。

單位：計數。使用總和統計資料來取得用戶端取消訂閱請求錯誤的總發生次數。

UnsubscribeServerError

處理取消訂閱請求時，源自 AWS AppSync 的錯誤數目。發生未預期的伺服器端問題時，通常會發生這種情況。

單位：計數。使用總和統計資料來取得伺服器端取消訂閱請求錯誤的總發生次數。

PublishDataMessageSuccess

已成功發佈的訂閱事件訊息數目。

單位：計數。使用總和統計以取得已順利發佈之訂閱事件訊息的總計。

PublishDataMessageClientError

因為用戶端錯誤而無法發佈的訂閱事件訊息數目。

Unit：計數。使用總和統計以取得這些用戶端發佈訂閱事件錯誤的總次數。

PublishDataMessageServerError

發佈訂閱事件訊息時，源自 AWS AppSync 的錯誤數目。發生未預期的伺服器端問題時，通常會發生這種情況。

單位：計數。使用總和統計以取得這些伺服器端發佈訂閱事件錯誤的總次數。

PublishDataMessageSize

已發佈的訂閱事件訊息大小。

單位：位元組。

ActiveConnections

1 分鐘內從用戶端到 AWS AppSync 的並行 WebSocket 連線數。

單位：計數。使用總和統計來取得開啟的連線總數。

ActiveSubscriptions

在 1 分鐘內來自用戶端同時訂閱的數量。

單位：計數。使用總和統計來取得使用中的訂閱總數。

ConnectionDuration

連線保持開啟的時間。

單位：毫秒。使用平均統計資料來評估連線持續時間。

OutboundMessages

已成功發佈的計量訊息數量。一個計量訊息等於 5 kB 的交付資料。

單位：計數。使用總和統計資料來取得成功發佈的計量訊息總數。

InboundMessageSuccess

成功處理傳入訊息的數量。由變動叫用的每個訂閱類型都會產生一個傳入訊息。

單位：計數。使用總和統計資料來取得成功處理傳入訊息的總數。

InboundMessageError

由於 API 請求無效而無法處理的傳入訊息數量，例如超過 240 kB 訂閱承載大小限制。

單位：計數。使用總和統計資料來取得與 API 相關處理失敗的傳入訊息總數。

InboundMessageFailure

由於 錯誤而無法處理的傳入訊息數量 AWS AppSync。

單位：計數。使用總和統計資料來取得與處理失敗 AWS AppSync相關的傳入訊息總數。

InboundMessageDelayed

延遲傳入訊息的數量。違反傳入訊息速率配額或傳出訊息速率配額時，傳入訊息可能會延遲。

單位：計數。使用總和統計資料來取得已延遲的傳入訊息總數。

InboundMessageDropped

捨棄傳入訊息的數量。違反傳入訊息速率配額或傳出訊息速率配額時，可以捨棄傳入訊息。

單位：計數。使用總和統計資料來取得已捨棄的傳入訊息總數。

InvalidationSuccess

使用的變動成功使訂閱失效（取消訂閱）的數目
`$extensions.invalidateSubscriptions()`。

單位：計數。使用總和統計資料來擷取已成功取消訂閱的訂閱總數。

InvalidationRequestSuccess

已成功處理的失效請求數量。

單位：計數。使用總和統計資料來取得已成功處理的失效請求總數。

InvalidationRequestError

由於 API 請求無效而無法處理的失效請求數量。

單位：計數。使用總和統計資料來取得發生 API 相關處理失敗的失效請求總數。

InvalidationRequestFailure

由於 錯誤而無法處理的失效請求數量 AWS AppSync。

單位：計數。使用總和統計資料來取得發生 AWS AppSync 相關處理失敗的失效請求總數。

InvalidationRequestDropped

超過失效請求配額時捨棄的失效請求數量。

單位：計數。使用總和統計資料來取得捨棄失效請求的總數。

比較傳入和傳出訊息

執行變動時，會叫用具有該變動 `@aws_subscribe` 指令的訂閱欄位。每個訂閱調用都會產生一個傳入訊息。例如，如果兩個訂閱欄位在 `@aws_subscribe` 中指定相同的變動，則呼叫該變動時會產生兩個傳入訊息。

一個傳出訊息等於傳送至 WebSocket 用戶端的 5 kB 資料。例如，將 15 kB 的資料傳送至 10 個用戶端會產生 30 個傳出訊息 (15 kB * 10 個用戶端/每則訊息 5 kB = 30 則訊息)。

您可以請求增加傳入或傳出訊息的配額。如需詳細資訊，請參閱《AWS 一般參考指南》中的[AWS AppSync 端點和配額](#)，以及《Service Quotas 使用者指南》中[請求提高配額](#)的說明。

增強型指標

增強型指標會發出 API 用量和效能的精細資料，例如 AWS AppSync 請求和錯誤計數、延遲和快取命中/遺失。所有增強型指標資料都會傳送到您的 CloudWatch 帳戶，而且您可以設定要傳送的資料類型。

 Note

使用增強型指標時，會收取額外費用。如需詳細資訊，請參閱 Amazon CloudWatch 定價中的詳細監控定價方案。 [Amazon CloudWatch](#)

您可以在 AWS AppSync 主控台其中的各種設定頁面上找到這些指標。在 API 設定頁面上，增強型指標區段可讓您啟用或停用下列項目：

解析程式指標行為：這些選項控制如何收集解析程式的其他指標。您可以選擇啟用完整的請求解析程式指標（針對請求中的所有解析程式啟用指標）或每個解析程式指標（僅針對組態設定為啟用的解析程式啟用指標）。以下是可用的選項：

解析程式指標行為清單

GraphQL errors per resolver (GraphQLError)

每個解析程式發生的 GraphQL 錯誤數目。

指標維度：API_Id、Resolver

單位：計數。

Requests per resolver (Request)

請求期間發生的調用次數。這會依每個解析程式記錄。

指標維度：API_Id、Resolver

單位：計數。

Latency per resolver (Latency)

完成解析程式調用的時間。延遲的測量單位為毫秒，並以每個解析程式為基礎進行記錄。

指標維度：API_Id、Resolver

單位：毫秒。

Cache hits per resolver (CacheHit)

請求期間快取命中次數。只有在使用快取時才會發出。快取命中會依每個解析程式記錄。

指標維度：API_Id、Resolver

單位：計數。

Cache misses per resolver (CacheMiss)

請求期間快取遺失的次數。只有在使用快取時才會發出。快取遺漏會依每個解析程式記錄。

指標維度：API_Id、Resolver

單位：計數。

資料來源指標行為：這些選項控制如何收集資料來源的其他指標。您可以選擇啟用完整請求資料來源指標（針對請求中的所有資料來源啟用指標）或每個資料來源指標（僅針對組態設定為啟用的資料來源啟用指標）。以下是可用的選項：

資料來源指標行為清單

Requests per data source (Request)

在請求期間發生的調用次數。請求會依每個資料來源記錄。如果啟用完整請求，每個資料來源都會在 CloudWatch 中擁有自己的項目。

指標維度：API_Id、Datasource

單位：計數。

Latency per data source (Latency)

完成資料來源調用的時間。延遲是以每個資料來源為基礎進行記錄。

指標維度：API_Id、Datasource

單位：毫秒。

Errors per data source (GraphQLError)

資料來源調用期間發生的錯誤數目。

指標維度：API_Id、Datasource

單位：計數。

操作指標：啟用 GraphQL 操作層級指標。

操作指標行為清單

Requests per operation (Request)

呼叫指定 GraphQL 操作的次數。

指標維度：API_Id、Operation

單位：計數。

GraphQL errors per operation (GraphQLError)

在指定 GraphQL 操作期間發生的 GraphQL 錯誤數目。

指標維度：API_Id、Operation

單位：計數。

CloudWatch Logs

您可以在任何新的或現有的 GraphQL API 上設定兩種類型的登入：請求層級和欄位層級。

請求層級日誌

設定請求層級記錄 (包含詳細內容) 時，會記錄下列資訊：

- 使用的字符數量
- 請求和回應 HTTP 標頭
- 在請求中執行的 GraphQL 查詢
- 整體操作摘要
- 已註冊之新的和現有的 GraphQL 訂閱

欄位層級日誌

設定欄位層級記錄時，會記錄下列資訊：

- 使用每個欄位的來源和引數產生請求映射
- 每個欄位的轉換回應映射，其中包含解析該欄位的結果資料

- 追蹤每個欄位的資訊

如果您開啟記錄，AWS AppSync 會管理 CloudWatch Logs。此程序包含建立日誌群組和日誌串流，並以這些日誌向日誌串流報告。

當您開啟 GraphQL API 上的記錄並提出請求時，AWS AppSync 會在日誌群組下建立日誌群組和日誌串流。日誌群組的命名是遵循 `/aws/appsync/apis/{graphql_api_id}` 格式。在每個日誌群組中，日誌將進一步分為日誌串流。它們會依照記錄資料回報的上次事件時間進行排序。

每個日誌事件都會標記為該請求的 `x-amzn-RequestId`。這可協助您篩選 CloudWatch 中的日誌事件，以取得該請求的所有記錄資訊。您可以從每個 GraphQL AWS AppSync 請求的回應標頭取得 `RequestId`。

欄位層級記錄是用以下日誌層級設定的：

- 無 - 不會擷取欄位層級日誌。
- 錯誤 - 僅針對錯誤類別中的欄位記錄以下資訊：
 - 伺服器回應的錯誤區段
 - 欄位層級錯誤
 - 產生的請求/回應功能已解析錯誤的欄位
- 資訊 - 僅記錄資訊和錯誤類別中欄位的下列資訊：
 - 資訊層級訊息
 - 透過 `$util.log.info` 和 傳送的使用者訊息 `console.log`
 - 不會顯示欄位層級追蹤和映射日誌。
 - 如果包含詳細內容的欄位層級記錄設為 INFO 或更高版本，會 AWS AppSync 新增轉換的映射範本記錄訊息。這將包含新增至轉換的映射範本的任何資訊，或解析程式或函數執行的 JavaScript 程式碼的輸出，如果您打算將密碼或授權標頭等敏感資訊傳送到下游資料來源，且不希望 日誌中的資訊。
- 偵錯 - 僅針對偵錯、資訊和錯誤類別中的欄位記錄以下資訊：
 - 偵錯層級訊息
 - 透過 `$util.log.info`、`console.log`、和 `$util.log.debug` 傳送的使用者訊息 `console.debug`
 - 不會顯示欄位層級追蹤和映射日誌。
- 全部 - 記錄查詢中所有欄位的下列資訊：
 - 欄位層級追蹤資訊

- 為每個欄位解析的產生請求/回應函數

監控的優點

您可以使用記錄和指標來識別、故障診斷和最佳化您的 GraphQL 查詢。例如，這些將協助您使用查詢中每個欄位記錄的追蹤資訊以偵錯延遲問題。為了示範，假設您使用 GraphQL 查詢中巢狀的一或多個解析程式。CloudWatch Logs 中的範例欄位操作可能看起來類似以下內容：

```
{
  "path": [
    "singlePost",
    "authors",
    0,
    "name"
  ],
  "parentType": "Post",
  "returnType": "String!",
  "fieldName": "name",
  "startOffset": 416563350,
  "duration": 11247
}
```

這可能對應到 GraphQL 結構描述，類似如下：

```
type Post {
  id: ID!
  name: String!
  authors: [Author]
}

type Author {
  id: ID!
  name: String!
}

type Query {
  singlePost(id:ID!): Post
}
```

在上述日誌結果中，路徑會顯示從執行名為 `singlePost()` 的查詢所傳回資料中的單一項目 `singlePost()`。在此範例中，它代表第一個索引 (0) 的名稱欄位。startOffset 提供從 GraphQL 查詢操作開始的位移。持續時

間是解析欄位的總時間。這些值非常有用，可以診斷為何來自特定資料來源的資料的執行速度低於預期，或特定欄位是否降低了整個查詢的速度。例如，您可以選擇增加 Amazon DynamoDB 資料表的佈建輸送量，或從導致整體操作效能不佳的查詢中移除特定欄位。

自 2019 年 5 月 8 日起，AWS AppSync 會以全結構化 JSON 的形式產生日誌事件。這可協助您使用 CloudWatch Logs Insights 和 Amazon OpenSearch Service 等日誌分析服務，以了解 GraphQL 請求的效能，以及結構描述欄位的使用特性。例如，您可以輕鬆找出延遲時間很長，並且可能是效能問題根本原因的解析程式。您也可以找出結構描述中最常用和最不常用的欄位，並評估移除 GraphQL 欄位的影響。

衝突偵測和同步記錄

如果 an AWS AppSync API 已將欄位解析程式日誌層級設定為全部的 CloudWatch Logs 記錄，則 AWS AppSync 會向日誌群組發出衝突偵測和解決資訊。這可讓您深入了解 AWS AppSync API 如何回應衝突。為了協助您解譯回應，日誌中提供下列資訊：

指標清單

`conflictType`

詳述發生的衝突原因是版本不符或客戶提供的條件所致。

`conflictHandlerConfigured`

說明請求時在解析程式中設定的衝突處理常式。

`message`

提供如何偵測和解決衝突的相關資訊。

`syncAttempt`

在最終拒絕請求之前，伺服器為了同步資料而嘗試的次數。

`data`

如果設定的衝突處理常式是 Automerge，則會填入此欄位，以顯示每個欄位所 Automerge 採取的決策。提供的動作可能是：

- 拒絕 - 當 Automerge 拒絕傳入欄位值，以支持伺服器中的值。
- ADDED - 當 由於伺服器中沒有預先存在的值 Automerge 而新增傳入欄位時。
- 附加 - 當 將傳入值 Automerge 附加到伺服器中存在的清單值。

- MERGED - 當 Automerge 將傳入值合併到伺服器中存在之 Set 的值時。

使用字符計數來最佳化您的請求

耗用少於或等於 1,500 KB 秒記憶體和 vCPU 時間的請求會分配一個字符。資源耗用超過 1,500 KB-秒的請求會收到額外的字符。例如，如果請求耗用 3,350 KB-秒，AWS AppSync 會將三個字符（四捨五入至下一個整數值）分配給請求。根據預設，AWS AppSync 每秒最多會將 5,000 或 10,000 個請求權杖分配給您帳戶中 APIs，具體取決於其部署所在的 AWS 區域。如果您的 APIs 平均每秒使用兩個權杖，則每秒限制為 2,500 或 5,000 個請求。如果您需要的每秒字符數量超過分配的數量，您可以提交請求，以提高請求字符速率的預設配額。如需詳細資訊，請參閱 AWS 一般參考《指南》中的 [AWS AppSync 端點和配額](#)，以及 [《Service Quotas 使用者指南》中的請求提高配額](#)。Service Quotas

高每個請求字符計數可能表示有機會最佳化您的請求並改善 API 的效能。可以增加每個請求字符計數的因素包括：

- GraphQL 結構描述的大小和複雜性。
- 請求和回應映射範本的複雜性。
- 每個請求的解析程式叫用次數。
- 從解析程式傳回的資料量。
- 下游資料來源的延遲。
- 需要連續資料來源呼叫的結構描述和查詢設計（而不是平行或批次呼叫）。
- 記錄組態，特別是欄位層級和詳細日誌內容。

Note

除了 AWS AppSync 指標和日誌之外，用戶端還可以透過回應標頭 存取請求中消耗的字符數量 `x-amzn-appsync-TokensConsumed`。

日誌大小限制

根據預設，如果已啟用記錄，AWS AppSync 會為每個請求傳送最多 1 MB 的日誌。超過此大小的日誌將被截斷。若要減少日誌大小，請選擇欄位層級日誌的記錄 ERROR 層級並停用 VERBOSE 記錄，或者如果不需要，則完全停用欄位層級日誌。作為 ALL 日誌層級的替代方案，您可以使用增強型指標來取得特

定解析程式、資料來源或 GraphQL 操作的指標，或使用 AppSync 提供的記錄公用程式來僅記錄必要的資訊。

日誌類型參考

RequestSummary

- requestId：請求的唯一識別符。
- graphQLAPIId：提出請求的 GraphQL API ID。
- statusCode：HTTP 狀態碼回應。
- latency：請求的端對端延遲，以奈秒為單位，並且為整數。

```
{
  "logType": "RequestSummary",
  "requestId": "dbe87af3-c114-4b32-ae79-8af11f3f96f1",
  "graphQLAPIId": "pmo28inf75eepg63qxq4ekoeg4",
  "statusCode": 200,
  "latency": 242000000
}
```

ExecutionSummary

- requestId：請求的唯一識別符。
- graphQLAPIId：提出請求的 GraphQL API ID。
- startTime：請求的 GraphQL 處理開始時間戳記，格式為 RFC 3339。
- endTime：GraphQL 處理請求的結束時間戳記，以 RFC 3339 格式顯示。
- 持續時間：經過的總 GraphQL 處理時間，以奈秒為單位，以整數表示。
- version：ExecutionSummary 的結構描述版本。
- parsing：
 - startOffset：剖析的開始位移，以奈秒為單位，相對於調用，以整數表示。
 - duration：剖析所花費的時間，以奈秒為單位，並且為整數。
- validation：
 - startOffset：用於驗證的開始位移，以奈秒為單位，相對於調用，以整數表示。
 - duration：執行驗證所花費的時間，以奈秒為單位，並且為整數。

```
{
  "duration": 217406145,
  "logType": "ExecutionSummary",
  "requestId": "dbe87af3-c114-4b32-ae79-8af11f3f96f1",
  "startTime": "2019-01-01T06:06:18.956Z",
  "endTime": "2019-01-01T06:06:19.174Z",
  "parsing": {
    "startOffset": 49033,
    "duration": 34784
  },
  "version": 1,
  "validation": {
    "startOffset": 129048,
    "duration": 69126
  },
  "graphqlAPIId": "pmo28inf75eepg63qxq4ekoeg4"
}
```

追蹤

- `requestId`：請求的唯一識別符。
- `graphqlAPIId`：提出請求的 GraphQL API ID。
- `startOffset`：欄位解析度的開始位移，以奈秒為單位，相對於調用，以整數表示。
- `duration`：解析欄位所花費的時間，以奈秒為單位，並且為整數。
- `fieldName`：正在解析的欄位名稱。
- `parentType`：正在解析欄位的父類別。
- `returnType`：正在解析欄位的傳回類型。
- `path`：路徑區段的清單，從回應的根開始，並結束於正在解析的欄位。
- `resolverArn`：用於解析欄位的解析程式 ARN。在巢狀欄位上可能不存在。

```
{
  "duration": 216820346,
  "logType": "Tracing",
  "path": [
    "putItem"
  ],
  "fieldName": "putItem",
}
```

```

    "startOffset": 178156,
    "resolverArn": "arn:aws:appsync:us-east-1:111111111111:apis/
pmo28inf75eepg63qxq4ekoeg4/types/Mutation/fields/putItem",
    "requestId": "dbe87af3-c114-4b32-ae79-8af11f3f96f1",
    "parentType": "Mutation",
    "returnType": "Item",
    "graphqlAPIId": "pmo28inf75eepg63qxq4ekoeg4"
}

```

使用 CloudWatch Logs Insights 分析您的日誌

以下是您可以執行的查詢範例，您可以透過這些執行取得您 GraphQL 操作效能及運作狀態的可採取動作詳情。這些範例可在 CloudWatch Logs Insights 主控台中作為範例查詢使用。在 [CloudWatch 主控台](#) 中，選擇 Logs Insights，選取 GraphQL API 的 AWS AppSync 日誌群組，然後在範例 AWS AppSync 查詢下選擇查詢。

下列查詢會傳回前 10 個使用最多權杖的 GraphQL 請求：

```

filter @message like "Tokens Consumed"
| parse @message "* Tokens Consumed: *" as requestId, tokens
| sort tokens desc
| display requestId, tokens
| limit 10

```

以下查詢會傳回前 10 個延遲最長的解析程式：

```

fields resolverArn, duration
| filter logType = "Tracing"
| limit 10
| sort duration desc

```

以下查詢會傳回最常呼叫的解析程式：

```

fields ispresent(resolverArn) as isRes
| stats count() as invocationCount by resolverArn
| filter isRes and logType = "Tracing"
| limit 10
| sort invocationCount desc

```

以下查詢會傳回映射範本中錯誤最多的解析程式：

```
fields ispresent(resolverArn) as isRes
| stats count() as errorCount by resolverArn, logType
| filter isRes and (logType = "RequestMapping" or logType = "ResponseMapping") and
  fieldInError
| limit 10
| sort errorCount desc
```

以下查詢會傳回解析程式的延遲統計資料：

```
fields ispresent(resolverArn) as isRes
| stats min(duration), max(duration), avg(duration) as avg_dur by resolverArn
| filter isRes and logType = "Tracing"
| limit 10
| sort avg_dur desc
```

以下查詢會傳回欄位延遲統計資料：

```
stats min(duration), max(duration), avg(duration) as avg_dur
by concat(parentType, '/', fieldName) as fieldKey
| filter logType = "Tracing"
| limit 10
| sort avg_dur desc
```

CloudWatch Logs Insights 查詢的結果可匯出到 CloudWatch 儀表板。

使用 OpenSearch Service 分析您的日誌

您可以使用 Amazon OpenSearch Service 搜尋、分析和視覺化您的 AWS AppSync 日誌，以識別操作問題的效能瓶頸和根本原因。您可以找出延遲最長和錯誤最多的解析程式。此外，您可以使用 OpenSearch Dashboards 建立具有強大視覺化效果的儀表板。OpenSearch Dashboards 是 OpenSearch Service 中提供的開放原始碼資料視覺化和探索工具。使用 OpenSearch Dashboards，您可以持續監控 GraphQL 操作的效能和運作狀態。例如，您可以建立儀表板來視覺化 GraphQL 請求的 P90 延遲，並深入了解每個解析程式的 P90 延遲。

使用 OpenSearch Service 時，請使用「cwl*」做為篩選條件模式來搜尋 OpenSearch 索引。OpenSearch Service 會以「cwl-」為從 CloudWatch Logs 串流的日誌編製索引。若要將傳送至 OpenSearch Service 的其他 CloudWatch 日誌中的 differentiate AWS AppSync API 日誌，建議您將的其他篩選條件表達 `graphqlAPIID.keyword=YourGraphQLAPIID` 式新增至您的搜尋。

日誌格式遷移

AWS AppSync 在 2019 年 5 月 8 日當天或之後產生的日誌事件會格式化為全結構化 JSON。若要分析 2019 年 5 月 8 日之前的 GraphQL 請求，您可以使用 [GitHub 範例](#) 提供的指令碼，將較舊的日誌遷移至全結構化 JSON。若您需要使用 2019 年 5 月 8 日以前的日誌格式，請使用下列設定建立支援票證：將 Type (類型) 設為 Account Management (帳戶管理)，然後將 Category (類別) 設為 General Account Question (一般帳戶問題)。

您也可以使用 CloudWatch 中的 [指標篩選條件](#)，將日誌資料轉換為數值 CloudWatch 指標，以便在它們上繪製或設定警示。

使用 AWS X-Ray 追蹤 in AWS AppSync 中的請求

您可以使用 [AWS X-Ray](#) 來追蹤在 AWS AppSync 中執行的請求。您可以在可使用 X-Ray 的所有 AWS 區域中使用 X-Ray with AWS AppSync。X-Ray 為您提供整個 GraphQL 請求的詳細概觀。這可讓您分析 API 及其基礎解析程式和資料來源中的延遲。您可以使用 X-Ray 服務映射來檢視請求的延遲，包括與 X-Ray 整合的任何 AWS 服務。您也可以設定抽樣規則，以根據您指定的條件，告訴 X-Ray 要記錄哪些請求，以及以哪些抽樣率記錄。

如需在 X-Ray 中取樣的詳細資訊，請參閱 [在 AWS X-Ray 主控台中設定取樣規則](#)。

設定與組態

您可以透過 AWS AppSync 主控台為 GraphQL API 啟用 X-Ray 追蹤。

1. 登入 AWS AppSync 主控台。
2. 在導覽窗格中選擇 Settings (設定)。
3. 在 X-Ray 下，開啟 Enable X-Ray (啟用 X-Ray)。
4. 選擇 Save (儲存)。現在已為您的 API 啟用 X-Ray 追蹤。

如果您使用的是 AWS CLI 或 AWS CloudFormation，您也可以在建 new AWS AppSync API 時啟用 X-Ray 追蹤，或將 xrayEnabled 屬性設定為 true 來更新 existing AWS AppSync API。

為 an AWS AppSync API 啟用 X-Ray 追蹤時，會在您的帳戶中自動建立具有 AWS Identity and Access Management [適當許可的服務連結角色](#)。這可讓 AWS AppSync 以安全的方式將追蹤傳送至 X-Ray。

使用 X-Ray 追蹤您的 API

抽樣

透過使用抽樣規則，您可以控制在 AWS AppSync 中記錄的資料量，並可以即時修改抽樣行為，而無需修改或重新部署程式碼。例如，此規則使用 API ID 3n572shhcfokwhdnq1ogu59v6 對 GraphQL API 的請求進行取樣。

- 規則名稱 — test-sample
- 優先順序 — 10
- 儲槽大小 — 10
- 固定頻率 — 10
- 服務名稱 — *
- 服務類型 — AWS::AppSync::GraphQLAPI
- HTTP 方法 — *
- 資源 ARN — arn:aws:appsync:us-west-2:123456789012:apis/3n572shhcfokwhdnq1ogu59v6
- 主機 — *

了解追蹤

當您為 GraphQL API 啟用 X-Ray 追蹤時，您可以使用 X-Ray 追蹤詳細資訊頁面來檢查對 API 提出請求的詳細延遲資訊。以下範例顯示此特定請求的追蹤檢視以及服務對應。請求是對名為 postAPI 且具有 Post 類型的 API 提出，其資料包含在名為的 Amazon DynamoDB 資料表中 PostTable-Example。

下列追蹤影像對應下列 GraphQL 查詢：

```
query getPost {
  getPost(id: "1") {
    id
    title
  }
}
```

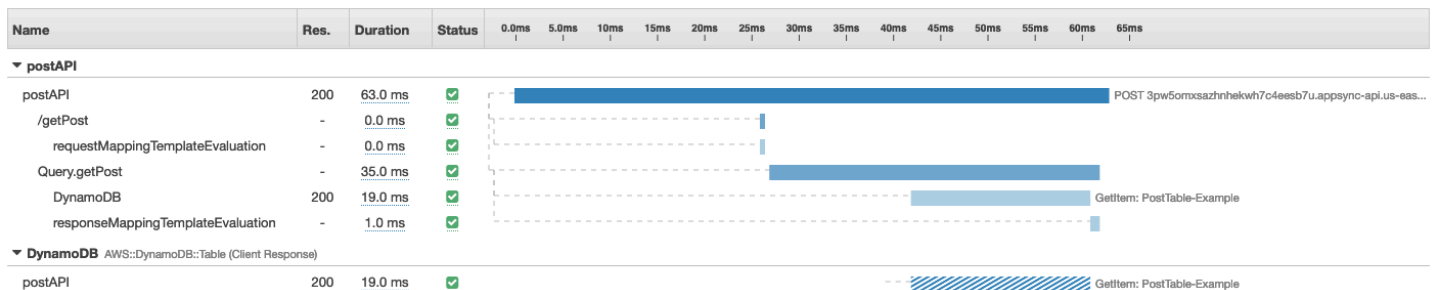
getPost 查詢的解析程式使用基礎 DynamoDB 資料來源。下列追蹤檢視顯示對 DynamoDB 的呼叫，以及查詢執行各個部分的延遲：

Traces > Details

Method	Response	Duration	Age	ID
POST	200	63.0 ms	12.1 sec (2020-01-27 02:45:05 UTC)	1-5e2e4eb1-0df8dba693373510ab7ae4c3

▼ Trace Map

Map legend ⓘ



- 在上述影像中，/getPost 表示要解析之元素的完整路徑。在這種情況下，因為 getPost 是根 Query 類型的欄位，它直接出現在路徑的根之後。
- requestMappingTemplateEvaluation 代表 AWS AppSync 在查詢中評估此元素的請求映射範本所花費的時間。
- Query.getPost 表示類型和欄位 (格式為 Type.field)。它可以包含多個子區段，取決於 API 的結構和要追蹤的請求而定。
 - DynamoDB 表示附加至此解析程式的資料來源。它包含網路呼叫 DynamoDB 以解析 欄位的延遲。
 - responseMappingTemplateEvaluation 代表 AWS AppSync 在查詢中評估此元素的回應映射範本所花費的時間。

當您在 X-Ray 中檢視追蹤時，您可以選擇子區段並探索詳細檢視，以取得有關 AWS AppSync 區段中子區段的其他內容和中繼資料資訊。

對於某些深度巢狀或複雜的查詢，請注意，交付至 X-Ray by AWS AppSync 的區段可以大於區段文件允許的大小上限，如[AWS X-Ray 區段文件](#)所定義。X-Ray 不會顯示超過限制的區段。

使用 記錄 AWS AppSync API 呼叫 AWS CloudTrail

AWS AppSync 已與 服務整合 AWS CloudTrail，此服務提供由使用者、角色 AWS 或服務在其中採取之動作的記錄 AWS AppSync。CloudTrail 會將 的所有 API 呼叫擷取 AWS AppSync 為事件。擷取的呼叫包括來自 AWS AppSync 主控台的呼叫，以及來自對 AWS AppSync APIs 的程式碼呼叫。您可以使用 CloudTrail 所收集的資訊來判斷提出的請求 AWS AppSync、提出請求者的 IP 地址、提出請求的人員、提出請求的時間，以及其他詳細資訊。

您可以建立線索，以將 CloudTrail 事件持續交付至 Amazon Simple Storage Service (Amazon S3) 儲存貯體，包括 的事件 AWS AppSync。如果您未設定線索，仍然可以在 CloudTrail 主控台中檢視最新的事件。

如需有關 CloudTrail 的相關資訊，請參閱 [AWS CloudTrail 使用者指南](#)。

AWS AppSync CloudTrail 中的資訊

當您建立 AWS 帳戶時，會在您的帳戶上啟用 CloudTrail。在事件歷史記錄中的 CloudTrail 主控台中，您可以檢視、搜尋和下載 AWS 帳戶中的最新事件。如需詳細資訊，請參閱《AWS CloudTrail 使用者指南》中的[使用 CloudTrail 事件歷史記錄檢視事件](#)。

若要不持續記錄您 AWS 帳戶中的事件，包括 的事件 AWS AppSync，請建立追蹤。根據預設，當您在主控台建立線索時，線索會套用到所有 AWS 區域。線索會記錄 AWS 分割區中所有區域的事件，並將日誌檔案交付至您指定的 Amazon S3 儲存貯體。此外，您可以設定其他 AWS 服務，以進一步分析和處理 CloudTrail 日誌中所收集的事件資料。如需詳細資訊，請參閱《AWS CloudTrail 使用者指南》中的下列主題：

- [為 AWS 您的帳戶建立線索](#)
- [AWS 服務與 CloudTrail 日誌的整合](#)
- [設定 CloudTrail 的 Amazon SNS 通知](#)
- [從多個區域接收 CloudTrail 日誌檔案](#)
- [從多個帳戶接收 CloudTrail 日誌檔案](#)

CloudTrail 會記錄所有 AWS AppSync API 操作。例如，對 `CreateGraphqlApi`、`CreateDataSource` 和 `ListResolvers` APIs 呼叫會在 CloudTrail 日誌檔案中產生項目。這些操作和其他操作都記錄在 [AWS AppSync API 參考](#) 中。

每一筆事件或日誌專案都會包含產生請求者的資訊。身分資訊可協助您判斷：

- 請求是使用根還是 AWS Identity and Access Management (IAM) 使用者登入資料提出。
- 提出該請求時，是否使用了特定角色或聯合身分使用者的暫時安全憑證。
- 請求是否由其他 AWS 服務提出。

如需詳細資訊，請參閱《AWS CloudTrail 使用者指南》中的 [CloudTrail userIdentity 元素](#)。

AWS AppSync CloudTrail 中的資料事件

[資料事件](#)提供在資源上或在資源中執行的資源操作的相關資訊 (例如，讀取或寫入 Amazon S3 物件)。這些也稱為資料平面操作。資料事件通常是大量資料的活動。根據預設，CloudTrail 不會記錄資料事件。CloudTrail 事件歷史記錄不會記錄資料事件。

資料事件需支付額外的費用。如需 CloudTrail 定價的詳細資訊，請參閱 [AWS CloudTrail 定價](#)。

您可以使用 CloudTrail 主控台 AWS CLI 或 CloudTrail API 操作 (這些操作包括查詢、變動和訂閱操作、將操作連接到您的即時 WebSocket 端點，而不是透過您的即時 WebSocket 端點傳送的訊息) 來記錄 `AWS::AppSync::GraphQLApi` 資源類型的資料事件。如需如何記錄資料事件的詳細資訊，請參閱《AWS CloudTrail 使用者指南》中的 [使用 AWS Management Console 記錄資料事件](#) 和 [使用 AWS Command Line Interface 記錄資料事件](#)。

下表列出您可以記錄資料事件 AWS AppSync 的資源類型。資料事件類型 (主控台) 欄顯示 CloudTrail 主控台中資料事件類型清單中選擇的值。resources.type 值欄會顯示值，您會在使用 AWS CLI 或 CloudTrail APIs 設定進階事件選取器時指定該 resources.type 值。記錄到 CloudTrail 的資料 API 資料行會針對資源類型顯示記錄到 CloudTrail 的 API 呼叫。

資料事件類型 (主控台)	resources.type 值	記錄到 CloudTrail 的資料 API
AppSync GraphQL	<code>AWS::AppSync::GraphQLApi</code>	GraphQL

您可以設定進階事件選取器來篩選 eventNames、readOnly 和 resources.ARN 欄位，以僅記錄對您重要的事件。如需這些欄位的詳細資訊，請參閱 AWS CloudTrail API 參考中的 [AdvancedFieldSelector](#)。

```
[
  {
    "name": "Only 1 AppSync API",
```

```
"fieldSelectors": [
  {
    "field": "eventCategory",
    "equals": [
      "Data"
    ]
  },
  {
    "field": "resources.type",
    "equals": [
      "AWS::AppSync::GraphQLApi"
    ]
  },
  {
    "field": "resources.ARN",
    "equals": [
      "arn:aws:appsync:us-east-1:111122223333:apis/YourGraphQLApiId"
    ]
  }
]
}
```

了解 AWS AppSync 日誌檔案項目

CloudTrail 會以包含一或多個日誌項目的日誌檔案的形式交付事件。事件代表來自任何來源的單一請求，並包含所請求操作、操作的日期和時間、請求參數等相關資訊。由於這些日誌檔案不是公有 API 呼叫的排序堆疊追蹤，因此不會以任何特定順序顯示。

Note

對於從 發出的日誌，requestID 不是授權的唯一 ID AWS AppSync。requestID 可由用戶端覆寫。因此，根據此資訊做出決策時，您應該謹慎。

下列範例 CloudTrail 日誌項目示範 CreateApiKey 操作。

```
{
  "Records": [{
    "eventVersion": "1.05",
    "userIdentity": {
      "type": "IAMUser",
```

```

    "principalId": "A1B2C3D4E5F6G7EXAMPLE",
    "arn": "arn:aws:iam::111122223333:user/Alice",
    "accountId": "111122223333",
    "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
    "userName": "diego_ramirez"
  },
  "eventTime": "2018-01-31T21:49:09Z",
  "eventSource": "appsync.amazonaws.com",
  "eventName": "CreateApiKey",
  "awsRegion": "us-west-2",
  "sourceIpAddress": "192.2.0.1",
  "userAgent": "aws-cli/1.11.72 Python/2.7.11 Darwin/16.7.0 botocore/1.5.35",
  "requestParameters": {
    "apiId": "a1b2c3d4e5f6g7h8i9jexample"
  },
  "responseElements": {
    "apiKey": {
      "id": "****",
      "expires": 1518037200000
    }
  },
  "requestID": "99999999-9999-9999-9999-999999999999",
  "eventID": "99999999-9999-9999-9999-999999999999",
  "readOnly": false,
  "eventType": "AwsApiCall",
  "recipientAccountId": "111122223333"
}
]
}
```

下列範例 CloudTrail 日誌項目示範 ListApiKeys 操作。

```

{
  "Records": [{
    "eventVersion": "1.05",
    "userIdentity": {
      "type": "IAMUser",
      "principalId": "A1B2C3D4E5F6G7EXAMPLE",
      "arn": "arn:aws:iam::111122223333:user/diego_ramirez",
      "accountId": "111122223333",
      "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
      "userName": "diego_ramirez"
    },
    ...
  ]
}
```

```

    "eventTime": "2018-01-31T21:49:09Z",
    "eventSource": "appsync.amazonaws.com",
    "eventName": "ListApiKeys",
    "awsRegion": "us-west-2",
    "sourceIPAddress": "192.2.0.1",
    "userAgent": "aws-cli/1.11.72 Python/2.7.11 Darwin/16.7.0 botocore/1.5.35",
    "requestParameters": {
      "apiId": "a1b2c3d4e5f6g7h8i9jexample"
    },
    "responseElements": {
      "apiKeys": [
        {
          "id": "****",
          "expires": 1517954400000
        },
        {
          "id": "****",
          "expires": 1518037200000
        }
      ]
    },
    "requestID": "99999999-9999-9999-9999-999999999999",
    "eventID": "99999999-9999-9999-9999-999999999999",
    "readOnly": false,
    "eventType": "AwsApiCall",
    "recipientAccountId": "111122223333"
  }
]
}

```

下列範例 CloudTrail 日誌項目示範 DeleteApiKey 操作。

```

{
  "Records": [{
    "eventVersion": "1.05",
    "userIdentity": {
      "type": "IAMUser",
      "principalId": "A1B2C3D4E5F6G7EXAMPLE",
      "arn": "arn:aws:iam::111122223333:user/diego_ramirez",
      "accountId": "111122223333",
      "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
      "userName": "diego_ramirez"
    },
    },
  ],
}

```

```

    "eventTime": "2018-01-31T21:49:09Z",
    "eventSource": "appsync.amazonaws.com",
    "eventName": "DeleteApiKey",
    "awsRegion": "us-west-2",
    "sourceIPAddress": "192.2.0.1",
    "userAgent": "aws-cli/1.11.72 Python/2.7.11 Darwin/16.7.0 botocore/1.5.35",
    "requestParameters": {
      "id": "****",
      "apiId": "a1b2c3d4e5f6g7h8i9jexample"
    },
    "responseElements": null,
    "requestID": "99999999-9999-9999-9999-999999999999",
    "eventID": "99999999-9999-9999-9999-999999999999",
    "readOnly": false,
    "eventType": "AwsApiCall",
    "recipientAccountId": "111122223333"
  }
]
}

```

下列範例 CloudTrail 日誌項目示範使用自訂 Lambda 函數授權方授權的成功 GraphQL 變動。

```

{
  "eventVersion": "1.10",
  "userIdentity": {
    "type": "Unknown"
  },
  "eventTime": "2024-11-06T15:42:30Z",
  "eventSource": "appsync.amazonaws.com",
  "eventName": "GraphQL",
  "awsRegion": "us-west-2",
  "sourceIPAddress": "15.248.1.214",
  "userAgent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:131.0)
  Gecko/20100101 Firefox/131.0",
  "requestParameters": null,
  "responseElements": null,
  "additionalEventData": {
    "operationName": "MyMutation",
    "authType": [
      "AWS_LAMBDA"
    ],
    "fieldAuthorizationResults": {
      "deniedFields": []
    }
  }
}

```

```

    }
  },
  "requestID": "c2d3768b-3446-40a1-bd95-8399fe776f96",
  "eventID": "21568be1-a1a8-4f43-b978-63cb4cc02a96",
  "readOnly": false,
  "resources": [
    {
      "accountId": "123456789012",
      "type": "AWS::AppSync::GraphQLApi",
      "ARN": "arn:aws:appsync:us-west-2:123456789012:apis/rxfqcxzi3nbvza2hsq4njqqq6u"
    }
  ],
  "eventType": "AwsApiCall",
  "managementEvent": false,
  "recipientAccountId": "123456789012",
  "eventCategory": "Data"
}

```

下列範例 CloudTrail 日誌項目示範使用自訂 Lambda 函數授權方授權的部分成功 GraphQL 操作。請注意指定拒絕欄位的 `fieldAuthorizationResults.deniedFields` 屬性。

```

{
  "eventVersion": "1.10",
  "userIdentity": {
    "type": "Unknown"
  },
  "eventTime": "2024-11-06T16:11:49Z",
  "eventSource": "appsync.amazonaws.com",
  "eventName": "GraphQL",
  "awsRegion": "us-west-2",
  "sourceIPAddress": "15.248.1.214",
  "userAgent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:131.0) Gecko/20100101 Firefox/131.0",
  "requestParameters": null,
  "responseElements": null,
  "additionalEventData": {
    "operationName": "MyMutation",
    "authType": [
      "AWS_LAMBDA"
    ],
    "fieldAuthorizationResults": {
      "deniedFields": [

```

```

    "arn:aws:appsync:us-west-2:123456789012:apis/rxfqcxzi3nbvza2hsq4njqqq6u/types/
Mutation/fields/createPost",
    "arn:aws:appsync:us-west-2:123456789012:apis/rxfqcxzi3nbvza2hsq4njqqq6u/types/
Subscription/fields/onCreatePost",
    "arn:aws:appsync:us-west-2:123456789012:apis/rxfqcxzi3nbvza2hsq4njqqq6u/types/
Post/fields/status"
  ]
},
"requestID": "ae817c4c-66ba-4f64-92a5-ba9c9c341dcd",
"eventID": "30109698-7605-476a-9dff-b7ed78d134dc",
"readOnly": false,
"resources": [
  {
    "accountId": "123456789012",
    "type": "AWS::AppSync::GraphQLApi",
    "ARN": "arn:aws:appsync:us-west-2:123456789012:apis/rxfqcxzi3nbvza2hsq4njqqq6u"
  }
],
"eventType": "AwsApiCall",
"managementEvent": false,
"recipientAccountId": "123456789012",
"eventCategory": "Data"
}

```

下列範例 CloudTrail 日誌項目示範失敗的 GraphQL 操作。

```

{
  "eventVersion": "1.10",
  "userIdentity": {
    "type": "Unknown"
  },
  "eventTime": "2024-11-06T15:51:11Z",
  "eventSource": "appsync.amazonaws.com",
  "eventName": "GraphQL",
  "awsRegion": "us-west-2",
  "sourceIPAddress": "15.248.1.214",
  "userAgent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:131.0) Gecko/20100101
Firefox/131.0",
  "errorCode": "AccessDenied",
  "errorMessage": "{\n  \"errors\" : [ {\n    \"errorType\" : \"UnauthorizedException\", \n
    \"message\" : \"You are not authorized to make this call.\" \n  } ]\n}",
  "requestParameters": null,

```

```

"responseElements": null,
"additionalEventData": {
  "operationName": "MyFullyDeniedLambdaMutation"
},
"requestID": "0bef3cf3-a48b-4de9-8b1f-038afb563516",
"eventID": "b738651f-4ec0-4548-8fec-200c6b42842b",
"readOnly": false,
"resources": [
  {
    "accountId": "123456789012",
    "type": "AWS::AppSync::GraphQLApi",
    "ARN": "arn:aws:appsync:us-west-2:123456789012:apis/rxfqcxi3nbvza2hsq4njqqq6u"
  }
],
"eventType": "AwsApiCall",
"managementEvent": false,
"recipientAccountId": "123456789012",
"eventCategory": "Data"
}

```

下列範例示範成功的 GraphQL 請求。

```

{
  "eventVersion": "1.10",
  "userIdentity": {
    "type": "AssumedRole",
    "principalId": "AIDACKCEVSQ6C2EXAMPLE:jane_doe",
    "arn": "arn:aws:sts::123456789012:assumed-role/admin/jane_doe",
    "accountId": "123456789012",
    "sessionContext": {
      "sessionIssuer": {
        "type": "Role",
        "principalId": "AIDACKCEVSQ6C2EXAMPLE",
        "arn": "arn:aws:iam::123456789012:role/admin",
        "accountId": "123456789012",
        "userName": "jane_doe"
      },
      "attributes": {
        "creationDate": "2024-11-06T15:40:09Z",
        "mfaAuthenticated": "false"
      }
    }
  }
},

```

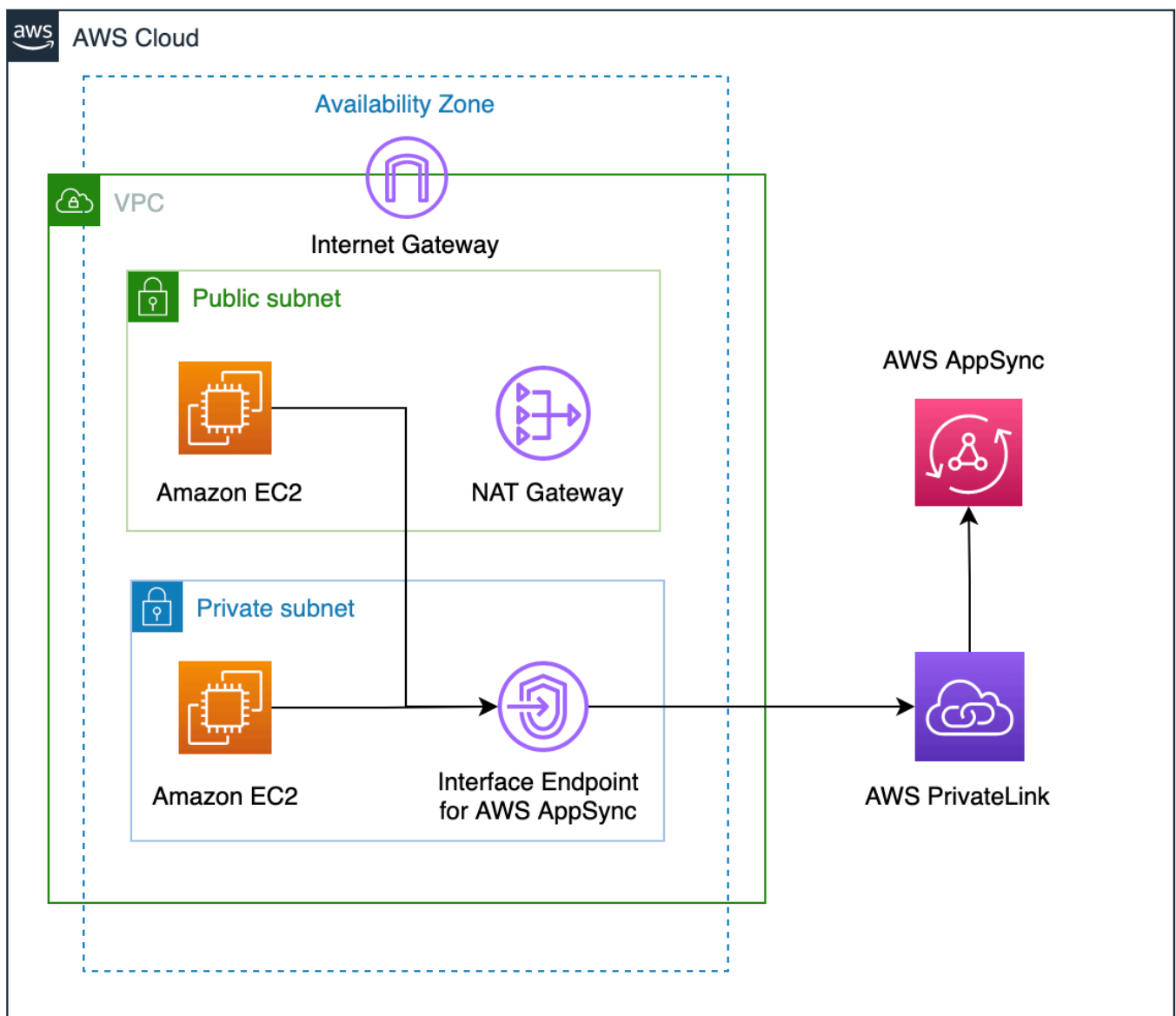


```
"eventTime": "2024-11-06T16:03:43Z",
"eventSource": "appsync.amazonaws.com",
"eventName": "GraphQL",
"awsRegion": "us-west-2",
"sourceIPAddress": "15.248.1.214",
"userAgent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:131.0) Gecko/20100101
Firefox/131.0",
"requestParameters": null,
"responseElements": null,
"additionalEventData": {
  "operationName": "IamFullSuccess",
  "authType": [
    "AWS_IAM"
  ],
  "fieldAuthorizationResults": {
    "allowedFields": [
      "arn:aws:appsync:us-west-2:123456789012:apis/rxfqcxzi3nbvza2hsq4njqqq6u/types/
Mutation/fields/createSecondPostAllowed"
    ],
    "deniedFields": []
  }
},
"requestID": "edc6bbbf-6bf2-40f5-820f-ef444f12e0c1",
"eventID": "524656a5-0925-4370-9e7e-08888e9c299f",
"readOnly": false,
"resources": [
  {
    "accountId": "123456789012",
    "type": "AWS::AppSync::GraphQLApi",
    "ARN": "arn:aws:appsync:us-west-2:123456789012:apis/rxfqcxzi3nbvza2hsq4njqqq6u"
  }
],
"eventType": "AwsApiCall",
"managementEvent": false,
"recipientAccountId": "123456789012",
"eventCategory": "Data"
}
```

使用 AWS AppSync 私有 APIs

如果您使用 Amazon Virtual Private Cloud (Amazon VPC)，您可以建立 AWS AppSync 私有 APIs，這些 APIs 只能從 VPC 存取。使用私有 API，您可以限制對內部應用程式的 API 存取，並連線到 GraphQL 和 Realtime 端點，而無需公開資料。

若要在 VPC 和服務之間建立私有連線 AWS AppSync，您必須建立[介面 VPC 端點](#)。介面端點採用，[AWS PrivateLink](#)可讓您在沒有網際網路閘道、NAT 裝置、VPN 連線或 AWS Direct Connect 連線的情況下，私下存取 AWS AppSync APIs。VPC 中的執行個體不需要公有 IP 地址，即能與 AWS AppSync API 通訊。VPC 與之間的流量 AWS AppSync 不會離開 AWS 網路。



啟用私有 API 功能之前，需要考慮一些其他因素：

- 在啟用私有 DNS 功能 AWS AppSync 的情況下設定的 VPC 介面端點，將導致 VPC 中的資源無法使用 AWS AppSync 產生的 APIs URL 叫用其他 AWS AppSync 公有 API。這是因為透過介面端點路由至公有 API 的請求，公有 APIs 不允許此請求。若要在此案例中叫用公有 APIs，建議您在公有 APIs 上設定自訂網域名稱，然後 VPC 中的資源可以使用這些名稱來叫用公有 API。
- 您的 AWS AppSync 私有 APIs 只能從您的 VPC 取得。AWS AppSync 主控台查詢編輯器只有在瀏覽器的網路組態可以將流量路由到 VPC（例如，透過 VPN 或的連線 AWS Direct Connect）時，才能連線到您的 API。
- 使用的 VPC 介面端點 AWS AppSync，您可以存取相同 AWS 帳戶和區域中的任何私有 API。若要進一步限制對私有 APIs 存取，您可以考慮下列選項：
 - 確保只有必要的管理員可以為其建立 VPC 端點介面 AWS AppSync。
 - 使用 VPC 端點自訂政策來限制可從 VPC 中的資源叫用哪些 APIs。
 - 對於 VPC 中的資源，我們建議您使用 IAM 授權來叫用 AWS AppSync APIs 方法是確保將資源授予 APIs 的縮小範圍角色。
- 建立或使用限制 IAM 主體的政策時，您必須將方法 `authorizationType` 的設定為 `AWS_IAM` 或 `NONE`。

建立 AWS AppSync 私有 APIs

下列步驟說明如何在 AWS AppSync 服務中建立私有 APIs。

Warning

您只能在建立 API 期間啟用私有 API 功能。建立 AWS AppSync API 或 AWS AppSync 私有 API 之後，就無法在 API 上修改此設定。

1. 登入 AWS Management Console 並開啟 [AppSync 主控台](#)。
 - 在儀表板上，選擇 Create API (建立 API)。
2. 選擇從頭開始設計 API，然後選擇下一步。
3. 在私有 API 區段中，選擇使用私有 API 功能。
4. 設定其餘選項，檢閱 API 的資料，然後選擇建立。

您必須先在 VPC AWS AppSync 中設定 的介面端點，才能使用 AWS AppSync 私有 API。請注意，私有 API 和 VPC 都必須位於相同的 AWS 帳戶和區域。

建立的介面端點 AWS AppSync

您可以使用 Amazon VPC AWS AppSync 主控台或 AWS Command Line Interface () 建立的介面端點 AWS CLI。如需詳細資訊，請參閱《Amazon VPC 使用者指南》中的[建立介面端點](#)。

Console

1. 登入 AWS Management Console 並開啟 Amazon VPC 主控台的[端點](#)頁面。
2. 選擇建立端點。
 - a. 在服務類別欄位中，確認已選取AWS 服務。
 - b. 在服務資料表中，選擇 `com.amazonaws.{region}.appsync-api`。確認類型資料欄值為 `Interface`。
 - c. 在 VPC 欄位中，選擇 VPC 及其子網路。
 - d. 若要啟用介面端點的私有 DNS 功能，請勾選啟用 DNS 名稱核取方塊。
 - e. 在安全群組欄位中，選擇一或多個安全群組。
3. 選擇建立端點。

CLI

使用 [create-vpc-endpoint](#) 命令，並指定 VPC ID、VPC 端點 (介面) 的類型、服務名稱、要使用端點的子網路，以及要與端點網路介面建立關聯的安全群組。例如：

```
$ aws ec2 create-vpc-endpoint --vpc-id vpc-ec43eb89 \
--vpc-endpoint-type Interface \
--service-name com.amazonaws.{region}.appsync-api \
--subnet-id subnet-abababab --security-group-id sg-1a2b3c4d
```

若要使用私有 DNS 選項，您必須設定 VPC 的 `enableDnsHostnames` 和 `enableDnsSupportattributes` 值。如需詳細資訊，請參閱 Amazon VPC 使用者指南中的[檢視並更新 VPC 的 DNS 支援](#)。如果您為介面端點啟用私有 DNS 功能，您可以使用其預設公有 DNS 端點，使用以下格式向 AWS AppSync API GraphQL 和即時端點提出請求：

```
https://{api_url_identifier}.appsync-api.{region}.amazonaws.com/graphql
```

如需服務端點的詳細資訊，請參閱《AWS 一般參考》中的[服務端點和配額](#)。

如需與介面端點進行服務互動的詳細資訊，請參閱《Amazon VPC 使用者指南》中的[透過介面端點存取服務](#)。

如需有關使用 AWS CloudFormation 建立和設定端點的資訊，請參閱AWS CloudFormation 使用者指南》中的 [AWS :: EC2 :: VPC Endpoint](#) 資源。

進階 範例

如果您為介面端點啟用私有 DNS 功能，您可以使用其預設公有 DNS 端點，使用以下格式向 AWS AppSync API GraphQL 和即時端點提出請求：

```
https://{api_url_identifier}.appsync-api.{region}.amazonaws.com/graphql
```

使用界面 VPC 端點公有 DNS 主機名稱，用於叫用 API 的基本 URL 將採用下列格式：

```
https://{vpc_endpoint_id}-{endpoint_dns_identifier}.appsync-api.
{region}.vpce.amazonaws.com/graphql
```

如果您已在 AZ 中部署端點，您也可以使用 AZ 特定的 DNS 主機名稱：

```
https://{vpc_endpoint_id}-{endpoint_dns_identifier}-{az_id}.appsync-api.
{region}.vpce.amazonaws.com/graphql.
```

使用 VPC 端點公有 DNS 名稱時，需要將 AWS AppSync API 端點主機名稱做為 Host 或 x-appsync-domain 標頭傳遞給請求。這些範例使用 TodoAPI 在[啟動範例結構描述指南中建立的](#)：

```
curl https://{vpc_endpoint_id}-{endpoint_dns_identifier}.appsync-api.
{region}.vpce.amazonaws.com/graphql \
-H "Content-Type:application/graphql" \
-H "x-api-key:da2-{xxxxxxxxxxxxxxxxxxxxxxxxxxxx}" \
-H "Host:{api_url_identifier}.appsync-api.{region}.amazonaws.com" \
-d '{"query":"mutation add($createtodoinput: CreateTodoInput!) {\n createTodo(input:
$createtodoinput) {\n id\n name\n where\n when\n description\n }\n}","variables":
{"createtodoinput":{"name":"My first GraphQL task","when":"Friday Night","where":"Day
1","description":"Learn more about GraphQL"}}}'
```

在下列範例中，我們將使用[啟動範例結構描述](#)指南中產生的 Todo 應用程式。為了測試範例 Todo API，我們將使用私有 DNS 來叫用 API。您可以使用您選擇的任何命令列工具；此範例使用 [curl](#) 傳送

查詢和變動，並使用 [wscat](#) 設定訂閱。若要模擬我們的範例，請將以下{ }命令中括號中的值取代為您 AWS 帳戶中的對應值。

測試變動操作 – **createTodo**請求

```
curl https://{api_url_identifier}.appsync-api.{region}.amazonaws.com/graphql \
-H "Content-Type:application/graphql" \
-H "x-api-key:da2-{xxxxxxxxxxxxxxxxxxxxxxxxxxxx}" \
-d '{"query":"mutation add($createtodoinput: CreateTodoInput!) {\n createTodo(input: $createtodoinput) {\n id\n name\n where\n when\n description\n }\n}", "variables": {"createtodoinput":{"name":"My first GraphQL task", "when":"Friday Night", "where":"Day 1", "description":"Learn more about GraphQL"}}}'
```

測試變動操作 – **createTodo**回應

```
{
  "data": {
    "createTodo": {
      "id": "<todo-id>",
      "name": "My first GraphQL task",
      "where": "Day 1",
      "when": "Friday Night",
      "description": "Learn more about GraphQL"
    }
  }
}
```

測試查詢操作 – **listTodos**請求

```
curl https://{api_url_identifier}.appsync-api.{region}.amazonaws.com/graphql \
-H "Content-Type:application/graphql" \
-H "x-api-key:da2-{xxxxxxxxxxxxxxxxxxxxxxxxxxxx}" \
-d '{"query":"query ListTodos {\n listTodos {\n items {\n description\n id\n name\n when\n where\n }\n }\n}", "variables":{"createtodoinput":{"name":"My first GraphQL task", "when":"Friday Night", "where":"Day 1", "description":"Learn more about GraphQL"}}}'
```

測試查詢操作 – **listTodos**請求

```
{
  "data": {
    "listTodos": {
```

```

    "items": [
      {
        "description": "Learn more about GraphQL",
        "id": "<todo-id>",
        "name": "My first GraphQL task",
        "when": "Friday night",
        "where": "Day 1"
      }
    ]
  }
}
}
}

```

測試訂閱操作 – 訂閱 `createTodo` 變動

若要在 中設定 GraphQL 訂閱 AWS AppSync，請參閱[建置即時 WebSocket 用戶端](#)。從 VPC 中的 Amazon EC2 執行個體，您可以使用 [wscat](#) 測試您的 AWS AppSync 私有 API 訂閱端點。以下範例使用 API KEY 進行授權。

```

$ header=`echo '{"host":"{api_url_identifier}.appsync-api.{region}.amazonaws.com","x-api-key":"da2-{xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx}"' | base64 | tr -d '\n'`
$ wscat -p 13 -s graphql-ws -c "wss://{api_url_identifier}.appsync-realtime-api.us-west-2.amazonaws.com/graphql?header=$header&payload=e30="
Connected (press CTRL+C to quit)
> {"type": "connection_init"}
< {"type": "connection_ack", "payload": {"connectionTimeoutMs": 300000}}
< {"type": "ka"}
> {"id": "f7a49717", "payload": {"data": {"query": "subscription onCreateTodo {onCreateTodo {description id name where when}}", "variables": {}, "extensions": {"authorization": {"x-api-key": "da2-{xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx}"}, "host": "{api_url_identifier}.appsync-api.{region}.amazonaws.com"}}, "type": "start"}
< {"id": "f7a49717", "type": "start_ack"}

```

或者，使用 VPC 端點網域名稱，同時確保在 `wscat` 命令中指定主機標頭以建立 WebSocket：

```

$ header=`echo '{"host":"{api_url_identifier}.appsync-api.{region}.amazonaws.com","x-api-key":"da2-{xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx}"' | base64 | tr -d '\n'`
$ wscat -p 13 -s graphql-ws -c "wss://{vpc_endpoint_id}-{endpoint_dns_identifier}.appsync-api.{region}.vpce.amazonaws.com/graphql?header=$header&payload=e30=" --header Host:{api_url_identifier}.appsync-realtime-api.us-west-2.amazonaws.com

```

```

Connected (press CTRL+C to quit)
> {"type": "connection_init"}
< {"type": "connection_ack", "payload": {"connectionTimeoutMs": 300000}}
< {"type": "ka"}
> {"id": "f7a49717", "payload": {"data": {"\query\": "\subscription
  onCreateTodo {onCreateTodo {description id priority title}}\",
\variables\": {}}, "extensions": {"authorization": {"x-api-key": "da2-
{xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx}", "host": "{api_url_identifier}.appsync-api.
{region}.amazonaws.com"}}}, "type": "start"}
< {"id": "f7a49717", "type": "start_ack"}

```

執行以下變動程式碼：

```

curl https://{api_url_identifier}.appsync-api.{region}.amazonaws.com/graphql \
-H "Content-Type: application/graphql" \
-H "x-api-key: da2-{xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx}" \
-d '{"query": "mutation add($createTodoInput: CreateTodoInput!) {\n createTodo(input:
  $createTodoInput) {\n id\n name\n where\n when\n description\n }\n}", "variables":
{"createTodoInput": {"name": "My first GraphQL task", "when": "Friday Night", "where": "Day
1", "description": "Learn more about GraphQL"}}}'

```

之後，會觸發訂閱，並顯示訊息通知，如下所示：

```

< {"id": "f7a49717", "type": "data", "payload": {"data": {"onCreateTodo": {"description": "Go
to the shops", "id": "169ce516-b7e8-4a6a-88c1-ab840184359f", "priority": 5, "title": "Go to
the shops"}}}}

```

使用 IAM 政策限制公有 API 建立

AWS AppSync 支援 IAM [Condition 陳述式](#) 以搭配私有 APIs 使用。visibility 欄位可以包含在 appsync:CreateGraphqlApi 操作的 IAM 政策陳述式中，以控制哪些 IAM 角色和使用者可以建立私有和公 APIs。這可讓 IAM 管理員定義 IAM 政策，僅允許使用者建立 Private GraphQL API。嘗試建立公有 API 的使用者會收到未經授權的訊息。

例如，IAM 管理員可以建立下列 IAM 政策陳述式，以允許建立私有 APIs：

```

{
  "Sid": "AllowPrivateAppSyncApis",
  "Effect": "Allow",
  "Action": "appsync:CreateGraphqlApi",
  "Resource": "*",

```



```
"Condition": {
  "ForAnyValue:StringEquals": {
    "appsync:Visibility": "PRIVATE"
  }
}
```

IAM 管理員也可以新增下列[服務控制政策](#)，以阻止 AWS 組織中的所有使用者建立私有 AWS AppSync APIs 以外的 APIs：

```
{
  "Sid": "BlockNonPrivateAppSyncApis",
  "Effect": "Deny",
  "Action": "appsync:CreateGraphqlApi",
  "Resource": "*",
  "Condition": {
    "ForAnyValue:StringNotEquals": {
      "appsync:Visibility": "PRIVATE"
    }
  }
}
```

Sharing AWS AppSync GraphQL APIs

AWS AppSync 與 AWS Resource Access Manager (AWS RAM) 整合以啟用資源共用。AWS RAM 是一項服務，可讓您與其他 AWS 帳戶 或透過 共用 AWS AppSync GraphQL APIs 上的調用動作（查詢、變動和訂閱操作，並將請求連接到您的即時 WebSocket 端點）AWS Organizations。您可以透過建立資源共享 AWS RAM，與 共用您擁有的資源。資源共享指定要共用的資源，以及共用它們的消費者。消費者可以包含下列項目。

- 中組織 AWS 帳戶 內部或外部的特定 AWS Organizations
- 中組織內部的組織單位 AWS Organizations
- 中的整個組織 AWS Organizations

如需 的詳細資訊 AWS RAM，請參閱[AWS Resource Access Manager 《使用者指南》](#)。

主題

- [shared AWS AppSync GraphQL APIs 的先決條件](#)
- [Share AWS AppSync GraphQL APIs](#)

- [停止共用 AWS AppSync GraphQL APIs](#)
- [跨帳戶事件](#)

shared AWS AppSync GraphQL APIs 的先決條件

Sharing AWS AppSync GraphQL APIs 具有下列先決條件。

- 若要共用 AWS AppSync GraphQL API，您必須在 中擁有它 AWS 帳戶。這表示必須在您的帳戶中配置或佈建 AWS AppSync GraphQL API。
- 若要與組織或 中的組織單位共用 AWS AppSync GraphQL API AWS Organizations，您必須啟用與 共用 AWS Organizations。如需詳細資訊，請參閱《AWS Resource Access Manager 使用者指南》中的[透過 AWS Organizations 啟用共用](#)。

Share AWS AppSync GraphQL APIs

若要共用 AWS AppSync GraphQL API，請先使用 建立資源共用 AWS Resource Access Manager。資源共用會指定要共用的資源、與其共用的消費者，以及主體可執行的動作。當您與其他人共用您擁有的 AWS AppSync GraphQL API 時 AWS 帳戶，您可以讓這些帳戶在您的 中呼叫該 AWS AppSync API AWS 帳戶。

如果您是 組織的一部分 AWS Organizations，且已啟用組織內的共用，則組織中的消費者會自動獲得共用資源的存取權。否則，消費者會收到加入資源共享的邀請，並在接受邀請後獲得共用資源的存取權。

共用考量

- 您只能共用 AWS AppSync GraphQL APIs，不能共用其他 API 類型，例如事件 APIs。
- 您只能共用具有 AWS_IAM 做為 APIs 上設定之授權模式之一的 AWS AppSync GraphQL API。

如果 從共用 AppSync GraphQL API 的授權模式清單中 AWS_IAM 移除，而資源共用可能仍然存在，則會呈現無效狀態。

- 您可以共用公有和私有 AWS AppSync GraphQL APIs。
- Private AWS AppSync GraphQL APIs 一律可以透過原始伺服器 VPC 中的 VPCs 端點存取 AWS 帳戶，而且支援所有授權模式，而不只是 AWS_IAM。
- 對於 shared AWS AppSync GraphQL APIs，僅針對 API 資源管理許可，不支援欄位和類型以及欄位資源的精細許可。當您共用 API 時，您會共用 API ARN 及其所有類型和欄位 ARNs。

使用 AWS RAM 主控台建立您擁有的資源共享

若要共用 AWS AppSync GraphQL API，請使用 AWS Resource Access Manager 使用者指南中[建立資源共用](#)所述的程序，並使用 RAM 許可名稱 `AWSRAMPermissionAppSyncGraphQLApiInvokeAccess`。

使用 AWS RAM 主控台建立和使用客戶受管許可來共用 private AWS AppSync GraphQL API

若要共用 private AWS AppSync GraphQL API，請使用 AWS Resource Access Manager 《使用者指南》中的建立[和使用客戶受管許可中所述的程序來建立客戶受管許可](#)。

例如，帳戶 A 的擁有者想要授予帳戶 B 中的主體許可，以存取透過 VPCE-B（帳戶 B 擁有的 VPC 端點）進行的呼叫的 private AWS AppSync GraphQL API (PrivateApiA)。在此情況下，帳戶 A 的擁有者需要建立 AWS RAM 客戶受管許可，如下所示。

```
{
  "Effect": "Allow",
  "Action": ["appsync:GraphQL"],
  "Condition": {
    "StringEqualsIgnoreCase": {
      "aws:SourceVpce": [
        "VPCE-B"
      ]
    }
  }
}
```

假設此新客戶受管 AWS RAM 許可名為 `private-api-A-access-via-vpce-b`。

若要 PrivateApiA 透過 啟用的跨帳戶存取 VPCE-B，客戶可以使用下列參數和先前範例中的客戶受管許可來建立 AWS RAM 資源共享。

- 資源類型： `appsync:Apis`
- 資源： `arn:aws:appsync:us-west-2:A:apis/PrivateApiA`
- 許可： `private-api-A-access-via-vpce-b`（客戶管理的許可）
- 委託人： `Account: B`

使用 建立您擁有的資源共享 AWS CLI

若要使用 共用 AWS AppSync GraphQL API AWS CLI，請使用 `create-resource-share` 命令搭配 `arn:aws:ram::aws:permission/AWSRAMPermissionAppSyncApiInvokeAccess` 做為 `--permission-arns` 切換的值。

如需 可用命令的完整清單 AWS RAM，請參閱 [AWS RAM CLI 參考](#)。

停止共用 AWS AppSync GraphQL APIs

若要停止共用您擁有的 AWS AppSync GraphQL APIs，您必須刪除資源共用或更新共用資源的主體。如需您要執行的動作，請參閱下列各節中的文件。

使用 AWS RAM 主控台停止共用您擁有的資源

請參閱 AWS Resource Access Manager 《使用者指南》中的[更新資源共享](#)。

使用 停止共用您擁有的資源 AWS CLI

使用 [disassociate-resource-share](#) 命令。

使用 AWS RAM 主控台刪除您擁有的資源共享

請參閱 AWS Resource Access Manager 《使用者指南》中的[刪除資源共享](#)。

使用 刪除您擁有的資源共享 AWS CLI

使用 [delete-resource-share](#) 命令。

如需 可用命令的完整清單 AWS RAM，請參閱 [AWS RAM CLI 參考](#)。

跨帳戶事件

您可以選擇加入記錄 AWS CloudTrail 資料事件，以監控和稽核跨帳戶 AWS AppSync GraphQL API DataPlane 活動。如需詳細資訊，請參閱《AWS CloudTrail 使用者指南》中的[記錄資料事件](#)。

使用 設定 GraphQL 執行複雜性、查詢深度和自我檢查 AWS AppSync

AWS AppSync 可讓您啟用或停用自我檢查功能，並將限制設定為單一查詢中的巢狀層級和解析程式數量。

使用自我檢查功能

Tip

如需 GraphQL 中自我檢查的詳細資訊，請參閱 [GraphQL 基礎網站上的這篇文章](#)。

根據預設，GraphQL 可讓您使用自我檢查來查詢結構描述本身，以探索其類型、欄位、查詢、變動、訂閱等。這是了解 GraphQL 服務如何塑造和處理資料的重要功能。不過，在處理自我檢查時，需要考量一些事項。您可能有一個使用案例，其會受益於停用自我檢查，例如，欄位名稱可能敏感或隱藏，或完整的 API 結構描述旨在為消費者保留未記錄的案例。在這些情況下，透過自我檢查發佈結構描述資料可能會導致刻意洩漏私有資料。

若要防止這種情況發生，您可以停用自我檢查。這將防止未經授權的對象在您的結構描述上使用自我檢查欄位。不過，請務必注意，自我檢查對於開發團隊了解其服務中的資料如何處理很有用。在內部，在生產程式碼中停用自我檢查作為額外的安全層時，保持啟用自我檢查可能會有所幫助。另一種處理方式是新增授權方法，該方法 AWS AppSync 也提供。如需詳細資訊，請參閱 [授權](#)。

AWS AppSync 可讓您在 API 層級啟用或停用自我檢查。若要啟用或停用自我檢查，請執行下列動作：

1. 登入 AWS Management Console 並開啟 [AppSync 主控台](#)。
2. 在 APIs 頁面上，選擇 GraphQL API 的名稱。
3. 在 API 首頁的導覽窗格中，選擇設定。
4. 在 API 組態中，選擇編輯。
5. 在自我檢查查詢下，執行下列動作：
 - 開啟或關閉啟用自我檢查查詢。
6. 選擇 Save (儲存)。

啟用自我檢查（預設行為）時，使用自我檢查系統將正常運作。例如，下圖顯示處理結構描述中所有可用類型的 `__schema` 欄位：



停用此功能時，驗證錯誤將改為出現在回應中：



設定查詢深度限制

有時候，您可能需要更精細地控制 API 在操作期間的運作方式。其中一個控制項是將限制新增至查詢可能處理的巢狀層級數量。根據預設，查詢能夠處理無限數量的巢狀層級。將查詢限制在指定的巢狀層級數量，可能會對專案的效能和彈性產生潛在影響。採取下列查詢：

```
query MyQuery {
  L1: nextLayer {
    L2: nextLayer {
      L3: nextLayer {
        L4: value
      }
    }
  }
}
```

```

    }
  }
}
}

```

您的專案可能會出於某些目的呼叫 來限制對 L1 或 L2 的查詢。根據預設，從 L1 到 的整個查詢 L4 都會處理，且無法控制。透過設定限制，您可以防止查詢存取超過指定層級的任何內容。

若要新增查詢深度限制，請執行下列動作：

1. 登入 AWS Management Console 並開啟 [AppSync 主控台](#)。
2. 在 APIs 頁面上，選擇 GraphQL API 的名稱。
3. 在 API 首頁的導覽窗格中，選擇設定。
4. 在 API 組態中，選擇編輯。
5. 在查詢深度下，執行下列動作：
 - a. 開啟或關閉啟用查詢深度。
 - b. 在最大深度中，設定深度限制。這可以介於 1 和 之間 75。
6. 選擇 Save (儲存)。

設定限制時，超過上限將導致 QueryDepthLimitReached 錯誤。例如，下圖顯示深度超過第三個 (L3) 和第四個 2 (L4) 層級限制的查詢：



請注意，在結構描述中，欄位仍然可以標記為 null 或不 null。如果不可為空的欄位收到 QueryDepthLimitReached 錯誤，該錯誤將擲回至第一個可為空的父欄位。

設定解析程式計數限制

您也可以控制每個查詢可以處理的解析程式數量。如同查詢深度，您可以設定此數量的限制。進行包含三個解析程式的下列查詢：

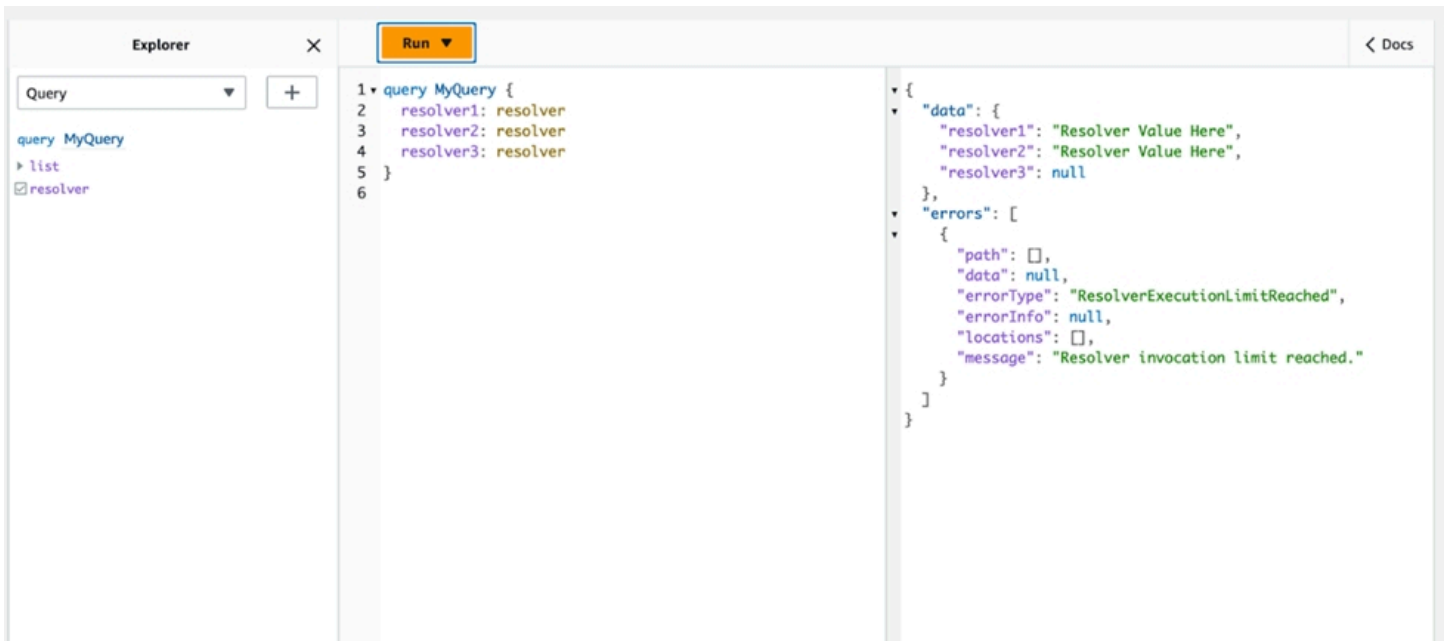
```
query MyQuery {  
  resolver1: resolver  
  resolver2: resolver  
  resolver3: resolver  
}
```

根據預設，每個查詢最多可以處理 10000 個解析程式。在上述範例中，resolver3將處理 resolver2、resolver1和。不過，您的專案可能會呼叫，限制每個查詢總共處理一或兩個解析程式。透過設定限制，您可以告訴查詢不要處理任何超過特定數字的解析程式，例如第一個 (resolver1) 或第二個 (resolver2) 解析程式。

若要新增解析程式計數限制，請執行下列動作：

1. 登入 AWS Management Console 並開啟 [AppSync 主控台](#)。
2. 在 APIs頁面上，選擇 GraphQL API 的名稱。
3. 在 API 首頁的導覽窗格中，選擇設定。
4. 在 API 組態中，選擇編輯。
5. 在解析程式計數限制下，執行下列動作：
 - a. 開啟啟用解析程式計數。
 - b. 在解析程式計數上限中，設定計數限制。這可以介於 1和 之間10000。
6. 選擇 Save (儲存)。

如同查詢深度限制，超過設定的解析程式限制會導致查詢結束，但其他解析程式ResolverExecutionLimitReached發生錯誤。在下圖中，解析程式計數限制為 2 的查詢會嘗試處理三個解析程式。由於限制，第三個解析程式會擲回錯誤且不會執行。



在 中使用環境變數 AWS AppSync

您可以使用環境變數來調整 AWS AppSync 解析程式和函數的行為，而無需更新程式碼。環境變數是與 API 組態一起存放的字串對，可供您的解析程式和函數在執行時間利用。它們特別適用於您必須參考只在初始設定期間可用，但在執行期間需要解析程式和函數使用的組態資料的情況。環境變數會公開程式碼中的組態資料，從而減少對這些值進行硬式編碼的需求。

Note

為了提高資料庫安全性，我們建議您使用 [Secrets Manager](#) 或 [AWS Systems Manager 參數存放區](#)，而非環境變數來存放登入資料或敏感資訊。若要利用此功能，請參閱[使用 AWS AppSync HTTP 資料來源叫用 AWS 服務](#)。

環境變數必須遵循數種行為和規則才能正常運作：

- JavaScript 解析程式/函數和 VTL 範本都支援環境變數。
- 不會在函數叫用之前評估環境變數。
- 環境變數僅支援字串值。
- 環境變數中的任何定義值都會視為字串常值，而不會展開。
- 變數評估最好應在函數程式碼中執行。

設定環境變數（主控台）

您可以建立變數並定義其鍵值對，以設定 AWS AppSync GraphQL API 的環境變數。您的解析程式和函數將使用環境變數的金鑰名稱，在執行時間擷取值。若要在 AWS AppSync 主控台中設定環境變數：

1. 登入 AWS Management Console 並開啟 [AppSync 主控台](#)。
2. 在 APIs 頁面上，選擇 GraphQL API 的名稱。
3. 在 API 首頁的導覽窗格中，選擇設定。
4. 在環境變數下，選擇新增環境變數。
5. 選擇 Add environment variable (新增環境變數)。
6. 輸入索引鍵和值。
7. 如有必要，請重複步驟 5 和 6 來新增更多索引鍵值。如果您需要移除索引鍵值，請選擇移除選項和要移除的索引鍵 (key)。
8. 選擇提交。

Tip

建立金鑰和值時，必須遵循幾個規則：

- 金鑰必須以字母開頭。
- 金鑰長度必須至少為兩個字元。
- 索引鍵只能包含字母、數字和底線字元 (_)。
- 值長度最多可達 512 個字元。
- 您可以在 GraphQL API 中設定最多 50 個鍵值對。

設定環境變數 (API)

若要使用 APIs 設定環境變數，您可以使用 `PutGraphqlApiEnvironmentVariables`。對應的 CLI 命令為 `put-graphql-api-environment-variables`。

若要使用 APIs 擷取環境變數，您可以使用 `GetGraphqlApiEnvironmentVariables`。對應的 CLI 命令為 `get-graphql-api-environment-variables`。

命令必須包含 API ID 和環境變數清單：

```
aws appsync put-graphql-api-environment-variables \  
  --api-id "<api-id>" \  
  --environment-variables '{"key1":"value1","key2":"value2", ...}'
```

下列範例 `abcdefghijklmnpqrstuvxyz` 會使用 `put-graphql-api-environment-variables` 命令，在 ID 為 `abcdefghijklmnpqrstuvxyz` 的 API 中設定兩個環境變數：

```
aws appsync put-graphql-api-environment-variables \  
  --api-id "abcdefghijklmnpqrstuvxyz" \  
  --environment-variables '{"USER_TABLE":"users_prod","DEBUG":"true"}'
```

請注意，當您使用 `put-graphql-api-environment-variables` 命令套用環境變數時，環境變數結構的內容會被覆寫；這表示現有的環境變數將會遺失。若要在新增環境變數時保留現有環境變數，請在請求中包含所有現有鍵值對以及新的鍵值對。使用上述範例，如果您想要新增 `"EMPTY":""`，您可以執行下列動作：

```
aws appsync put-graphql-api-environment-variables \  
  --api-id "abcdefghijklmnpqrstuvxyz" \  
  --environment-variables '{"USER_TABLE":"users_prod","DEBUG":"true", "EMPTY":""}'
```

若要擷取目前的組態，請使用 `get-graphql-api-environment-variables` 命令：

```
aws appsync get-graphql-api-environment-variables --api-id "<api-id>"
```

使用上述範例，您可以使用下列命令：

```
aws appsync get-graphql-api-environment-variables --api-id "abcdefghijklmnpqrstuvxyz"
```

結果會顯示環境變數清單及其索引鍵值：

```
{  
  "environmentVariables": {  
    "USER_TABLE": "users_prod",  
    "DEBUG": "true",  
    "EMPTY": ""  
  }  
}
```

設定環境變數 (CFN)

您可以使用以下範本來建立環境變數：

```
AWSTemplateFormatVersion: 2010-09-09
Resources:
  GraphQLApiWithEnvVariables:
    Type: "AWS::AppSync::GraphQLApi"
    Properties:
      Name: "MyApiWithEnvVars"
      AuthenticationType: "AWS_IAM"
      EnvironmentVariables:
        EnvKey1: "non-empty"
        EnvKey2: ""
```

環境變數和合併 APIs

來源 APIs 中定義的環境變數也可在您的合併 APIs 中使用。合併 APIs 中的環境變數為唯讀且無法更新。請注意，您的環境變數金鑰在所有來源 APIs 中必須是唯一的，才能讓您的合併成功；重複的金鑰一律會導致合併失敗。

擷取環境變數

若要擷取函數程式碼中的環境變數，請從解析程式和函數中的 `ctx.env` 物件擷取值。以下是此動作的一些範例。

Publishing to Amazon SNS

在此範例中，我們的 HTTP 解析程式會傳送訊息至 Amazon SNS 主題。只有在定義 GraphQL API 的堆疊且主題已部署之後，才會知道主題的 ARN。

```
/**
 * Sends a publish request to the SNS topic
 */
export function request(ctx) {
  const TOPIC_ARN = ctx.env.TOPIC_ARN;
  const { input: values } = ctx.args;
  // this custom function sends values to the SNS topic
  return publishToSNSRequest(TOPIC_ARN, values);
}
```

Transactions with DynamoDB

在此範例中，如果 API 部署用於預備或已在生產環境中，DynamoDB 資料表的名稱會不同。解析程式碼不需要變更。環境變數的值會根據 API 部署的位置更新。

```
import { util } from '@aws-appsync/utils';
export function request(ctx) {
  const { authorId, postId } = ctx.args;
  return {
    operation: 'TransactWriteItems',
    transactItems: [
      {
        table: ctx.env.POST_TABLE,
        operation: 'PutItem',
        key: util.dynamodb.toMapValues({ postId }),
        // rest of the configuration
      },
      {
        table: ctx.env.AUTHOR_TABLE,
        operation: 'UpdateItem',
        key: util.dynamodb.toMapValues({ authorId }),
        // rest of the configuration
      },
    ],
  };
}
```

設定授權和身分驗證來保護 GraphQL APIs

AWS AppSync 提供下列授權類型來保護 GraphQL APIs：API 金鑰、Lambda、IAM、OpenID Connect 和 Cognito 使用者集區。每個選項都提供不同的安全方法：

1. API 金鑰授權：控制未經驗證 APIs 的限流，提供簡單的安全選項。
2. Lambda 授權：啟用自訂授權邏輯，詳細說明函數輸入和輸出。
3. IAM 授權：使用 AWS 簽章第 4 版簽署程序，透過 IAM 政策進行精細存取控制。
4. OpenID Connect 授權：整合 OIDC 相容服務以進行使用者身分驗證。
5. Cognito 使用者集區：使用 Cognito 的使用者管理功能實作群組型存取控制。

授權類型

您可以透過五種方式授權應用程式與您的 AWS AppSync GraphQL API 互動。您可以在 your AWS AppSync API 或 CLI 呼叫中指定下列其中一個授權類型值，以指定您使用的授權類型：

- **API_KEY**

使用 API 金鑰。

- **AWS_LAMBDA**

用於使用 AWS Lambda 函數。

- **AWS_IAM**

用於 AWS Identity and Access Management ([IAM](#)) 許可。

- **OPENID_CONNECT**

對於使用您的 OpenID Connect 提供者。

- **AMAZON_COGNITO_USER_POOLS**

用於使用 Amazon Cognito 使用者集區。

這些基本授權類型適用於大多數的開發人員。如需更進階的使用案例，您可以透過 主控台、CLI 和 新增其他授權模式 AWS CloudFormation。對於其他授權模式，AWS AppSync API_KEY AWS_LAMBDA 提供採用上述值的授權類型（即、AWS_IAM、OPENID_CONNECT 和 AMAZON_COGNITO_USER_POOLS）。

當您將 `API_KEYAWS_LAMBDA`、或指定 `AWS_IAM` 為主要或預設授權類型時，您無法將其再次指定為其他授權模式之一。同樣地，您無法在其他 `AWS_IAM` 授權模式中複製 `API_KEYAWS_LAMBDA` 或 。您可以使用多個 Amazon Cognito 使用者集區和 OpenID Connect 供應商。不過，您無法在預設授權模式和任何其他授權模式之間使用重複的 Amazon Cognito 使用者集區或 OpenID Connect 供應商。您可以使用對應的組態規則表達式，為您的 Amazon Cognito 使用者集區或 OpenID Connect 提供者指定不同的用戶端。

當您儲存 API 組態的變更時，會 AWS AppSync 開始傳播變更。在您的組態變更傳播之前，會 AWS AppSync 繼續提供先前組態的內容。傳播組態變更之後，會 AWS AppSync 立即開始根據新組態提供內容。當 AWS AppSync 正在傳播 API 的變更時，我們無法根據先前的組態或新的組態來判斷 API 是否為您的內容提供服務。

API_KEY 授權

未驗證的 API 相較於已驗證 API 需要更嚴格的調節。其中一種控制未授權 GraphQL 端點調節的方法就是使用 API 金鑰。API 金鑰是應用程式中的硬式編碼值，會在您建立未經驗證的 GraphQL 端點時由 AWS AppSync 服務產生。您可以從主控台、CLI 或 API [AWS AppSync 參考輪換 API 金鑰](#)。

Console

1. 登入 AWS Management Console 並開啟 [AppSync 主控台](#)。
 - a. 在 APIs 儀表板中，選擇您的 GraphQL API。
 - b. 在側邊欄中，選擇設定。
2. 在預設授權模式下，選擇 API 金鑰。
3. 在 API 金鑰表格中，選擇新增 API 金鑰。

資料表中將產生新的 API 金鑰。

- 若要刪除舊的 API 金鑰，請選取資料表中的 API 金鑰，然後選擇刪除。

4. 選擇頁面底部的 Save (儲存)。

CLI

1. 如果您尚未這麼做，請設定您對 CLI AWS 的存取。如需詳細資訊，請參閱[組態基本概念](#)。
2. 執行 `update-graphql-api` 命令來建立 GraphQL API 物件。

您需要為此特定命令輸入兩個參數：

1. GraphQL API `api-id` 的。
2. 您 API `name` 的新。您可以使用相同的 `name`。
3. `authentication-type`，這會是 `API_KEY`。

 Note

還有其他參數，例如 必須設定 `Region`，但通常會預設為您的 CLI 組態值。

範例命令可能如下所示：

```
aws appsync update-graphql-api --api-id abcdefghijklmnopqrstuvwxyz --name
TestAPI --authentication-type API_KEY
```

輸出會在 CLI 中傳回。以下是 JSON 中的範例：

```
{
  "graphqlApi": {
    "xrayEnabled": false,
    "name": "TestAPI",
    "authenticationType": "API_KEY",
    "tags": {},
    "apiId": "abcdefghijklmnopqrstuvwxyz",
    "uris": {
      "GRAPHQL": "https://s8i3kk3ufhe9034ujnv73r513e.appsync-api.us-
west-2.amazonaws.com/graphql",
      "REALTIME": "wss://s8i3kk3ufhe9034ujnv73r513e.appsync-realtime-
api.us-west-2.amazonaws.com/graphql"
    },
    "arn": "arn:aws:appsync:us-west-2:348581070237:apis/
abcdefghijklmnopqrstuvwxyz"
  }
}
```

API 金鑰可設定為最多 365 天，您可以從過期日開始再延展現有的過期日期的最多 365 天。用於開發目的或者需安全公開公用 API 時，建議使用 API 金鑰。

在用戶端上，由標題 `x-api-key` 指定 API 金鑰。

例如，如果您的 API_KEY 是 'ABC123'，您可以經由 curl 來傳送 GraphQL 查詢，如下所示：

```
$ curl -XPOST -H "Content-Type:application/graphql" -H "x-api-key:ABC123" -d
'{"query": "query { movies { id } }"}' https://YOURAPPSYNCENDPOINT/graphql
```

AWS_LAMBDA 授權

您可以使用 AWS Lambda 函數實作自己的 API 授權邏輯。您可以對主要或次要授權方使用 Lambda 函數，但每個 API 可能只有一個 Lambda 授權函數。使用 Lambda 函數進行授權時，適用下列條件：

- 如果 API 已啟用 AWS_LAMBDA 和 AWS_IAM 授權模式，則無法使用 SigV4 簽章做為 AWS_LAMBDA 授權字符。
- 如果 API 已啟用 AWS_LAMBDA 和 OPENID_CONNECT 授權模式或 AMAZON_COGNITO_USER_POOLS 授權模式，則 OIDC 權杖無法用作 AWS_LAMBDA 授權權杖。請注意，OIDC 權杖可以是承載結構描述。
- Lambda 函數不得傳回解析程式超過 5MB 的內容資料。

例如，如果您的授權字符是 'ABC123'，您可以透過 curl 傳送 GraphQL 查詢，如下所示：

```
$ curl -XPOST -H "Content-Type:application/graphql" -H "Authorization:ABC123" -d
'{"query":
  "query { movies { id } }"}' https://YOURAPPSYNCENDPOINT/graphql
```

Lambda 函數會在每個查詢或變動之前呼叫。您可以根據 API ID 和身分驗證字符快取傳回值。當 Lambda 授權方回應小於 1,048,576 個位元組時，AWS AppSync 會快取後續請求的回應。如果 Lambda 授權方回應等於或大於 1,048,576 個位元組，AWS AppSync 不會快取回應，並為每個傳入請求叫用 Lambda 授權方。為了最佳化效能並將 Lambda 調用成本降至最低，建議您將 Lambda 授權方回應限制為 1,048,576 個位元組。根據預設，快取不會開啟，但可以在 API 層級或透過在函數的傳回 `ttlOverride` 值中設定值來啟用。

如有需要，可以指定在呼叫函數之前驗證授權字符的規則表達式。這些規則表達式用於驗證授權字符在呼叫函數之前格式是否正確。使用不符合此規則表達式的字符的任何請求將自動被拒絕。

用於授權的 Lambda 函數需要套用的主體政策，以允許 `appsync.amazonaws.com` AWS AppSync 呼叫它們。此動作會在 AWS AppSync 主控台中自動完成；AWS AppSync 主控台不會移除政策。如需將政策連接至 Lambda 函數的詳細資訊，請參閱《AWS Lambda 開發人員指南》中的以[資源為基礎的政策](#)。

您指定的 Lambda 函數將會收到具有下列形狀的事件：

```
{
  "authorizationToken": "ExampleAUTHtoken123123123",
  "requestContext": {
    "apiId": "aaaaaa123123123example123",
    "accountId": "111122223333",
    "requestId": "f4081827-1111-4444-5555-5cf4695f339f",
    "queryString": "mutation CreateEvent {...}\n\nquery MyQuery {...}\n",
    "operationName": "MyQuery",
    "variables": {}
  }
  "requestHeaders": {
    application request headers
  }
}
```

event 物件包含從應用程式用戶端傳送至之請求中的標頭 AWS AppSync。

授權函數必須至少傳回 `isAuthorized`，布林值指出請求是否已授權。AWS AppSync 會識別從 Lambda 授權函數傳回的下列金鑰：

Note

WebSocket 連線操作 `operationName` 中 `requestContext` 的值由 設定為 AWS AppSync `"DeepDish:Connect"`。

函數清單

`isAuthorized` (布林值，必要)

布林值，指出 中的值是否 `authorizationToken` 獲授權對 GraphQL API 進行呼叫。

如果此值為 `true`，GraphQL API 的執行會繼續。如果此值為 `false`，`UnauthorizedException` 則會引發

`deniedFields` (字串清單，選用)

即使從解析程式傳回值 `null`，也會強制變更為 的清單。

每個項目都是完整格式的欄位 ARN，格式為 `arn:aws:appsync:us-east-1:111122223333:apis/GraphQLApiId/types/Type/fields/Field` 或簡

短格式的 *TypeName.FieldName*。當兩個 APIs 共用 Lambda 函數授權方，且兩個 APIs 之間的常見類型和欄位之間可能會有模稜兩可時，應使用完整的 ARN 表單。

resolverContext (JSON 物件，選用)

在解析程式範本 `$ctx.identity.resolverContext` 中可見的 JSON 物件。例如，如果解析程式傳回下列結構：

```
{
  "isAuthorized": true
  "resolverContext": {
    "banana": "very yellow",
    "apple": "very green"
  }
}
```

解析程式範本 `ctx.identity.resolverContext.apple` 中的值為 "very green"。resolverContext 物件僅支援鍵/值對。不支援巢狀金鑰。

 Warning

此 JSON 物件的總大小不得超過 5MB。

ttlOverride (整數，選用)

應快取回應的秒數。如果未傳回任何值，則會使用來自 API 的值。如果這是 0，則不會快取回應。

Lambda 授權方的逾時時間為 10 秒。我們建議您設計在最短的時間內執行函數，以擴展 API 的效能。

多個 AWS AppSync APIs 可以共用單一身分驗證 Lambda 函數。不允許使用跨帳戶授權方。

在多個 APIs 之間共用授權函數時，請注意，短格式欄位名稱 (*typename.fieldname*) 可能會不小心隱藏欄位。若要取消模糊中的欄位 `deniedFields`，您可以以 `arn:aws:appsync:region:accountId:apis/GraphQLApiId/types/typename/fields/fieldname` 的形式指定不明確的欄位 ARN。

若要在 中新增 Lambda 函數作為預設授權模式 AWS AppSync：

Console

1. 登入 AWS AppSync 主控台並導覽至您要更新的 API。

2. 導覽至 API 的設定頁面。

將 API 層級授權變更為 AWS Lambda。

3. 選擇 AWS 區域 和 Lambda ARN 以授權 API 呼叫。

 Note

將自動新增適當的委託人政策，AWS AppSync 允許 呼叫您的 Lambda 函數。

4. 或者，設定回應 TTL 和字符驗證規則表達式。

AWS CLI

1. 將下列政策連接至正在使用的 Lambda 函數：

```
aws lambda add-permission --function-name "my-function" --statement-id "appsync"
--principal appsync.amazonaws.com --action lambda:InvokeFunction --output text
```

 Important

如果您想要將函數的政策鎖定為單一 GraphQL API，您可以執行此命令：

```
aws lambda add-permission --function-name "my-function" --
statement-id "appsync" --principal appsync.amazonaws.com --action
lambda:InvokeFunction --source-arn "<my AppSync API ARN>" --output text
```

2. 更新您的 AWS AppSync API，以使用指定的 Lambda 函數 ARN 做為授權方：

```
aws appsync update-graphql-api --api-id example2f0ur2oid7acexample --
name exampleAPI --authentication-type AWS_LAMBDA --lambda-authorizer-config
authorizerUri="arn:aws:lambda:us-east-2:111122223333:function:my-function"
```

 Note

您也可以包含其他組態選項，例如字符規則表達式。

下列範例說明 Lambda 函數，示範 Lambda 函數在用作 AWS AppSync 授權機制時可具有的各種身分驗證和失敗狀態：

```
def handler(event, context):
    # This is the authorization token passed by the client
    token = event.get('authorizationToken')
    # If a lambda authorizer throws an exception, it will be treated as unauthorized.
    if 'Fail' in token:
        raise Exception('Purposefully thrown exception in Lambda Authorizer.')

    if 'Authorized' in token and 'ReturnContext' in token:
        return {
            'isAuthorized': True,
            'resolverContext': {
                'key': 'value'
            }
        }

    # Authorized with no f
    if 'Authorized' in token:
        return {
            'isAuthorized': True
        }

    # Partial authorization
    if 'Partial' in token:
        return {
            'isAuthorized': True,
            'deniedFields': ['user.favoriteColor']
        }

    if 'NeverCache' in token:
        return {
            'isAuthorized': True,
            'ttlOverride': 0
        }

    if 'Unauthorized' in token:
        return {
            'isAuthorized': False
        }

    # if nothing is returned, then the authorization fails.
    return {}
```

規避 SigV4 和 OIDC 字符授權限制

下列方法可用來避免在特定授權模式啟用時，無法使用 SigV4 簽章或 OIDC 權杖做為 Lambda 授權權杖的問題。

如果您想要在為 AWS AppSync 的 API 啟用 AWS_IAM 和 授權模式時，使用 SigV4 簽章做為 Lambda AWS_LAMBDA 授權字符，請執行下列動作：

- 若要建立新的 Lambda 授權字符，請將隨機尾碼和/或字首新增至 SigV4 簽章。
- 若要擷取原始 SigV4 簽章，請從 Lambda 授權字符中移除隨機字首和/或尾碼，以更新您的 Lambda 函數。然後，使用原始 SigV4 簽章進行身分驗證。

如果您想要在為 API 啟用授權模式或 AMAZON_COGNITO_USER_POOLS 和 AWS_LAMBDA 授權模式時，使用 OIDC 權杖做為 Lambda AWS AppSync OPENID_CONNECT 授權權杖，請執行下列動作：

- 若要建立新的 Lambda 授權字符，請將隨機尾碼和/或字首新增至 OIDC 字符。Lambda 授權字符不應包含承載配置字首。
- 若要擷取原始 OIDC 權杖，請從 Lambda 授權權杖中移除隨機字首和/或尾碼，以更新您的 Lambda 函數。然後，使用原始 OIDC 權杖進行身分驗證。

AWS_IAM 授權

此授權類型會在 [AWS GraphQL API 上強制執行簽章第 4 版簽署程序](#)。GraphQL 您可以將身分與存取管理 (IAM) 的存取政策與此授權類型建立關聯。您的應用程式可以使用 存取金鑰（包含存取金鑰 ID 和私密存取金鑰）或使用 Amazon Cognito 聯合身分提供的短期臨時登入資料來利用此關聯。

如果您希望角色擁有執行所有資料操作的存取權限：

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "appsync:GraphQL"
      ],
    }
  ]
}
```

```

    "Resource": [
      "arn:aws:appsync:us-west-2:123456789012:apis/YourGraphQLApiId/*"
    ]
  }
]
}

```

您可以從 AppSync 主控台的主要 API 列表頁面中找到 `YourGraphQLApiId`，就在您的 API 名稱下方。或者，您可以使用 CLI 來取得該資訊：`aws appsync list-graphql-apis`

如果您想要僅限制對於特定 GraphQL 操作的存取權限，您可以在根 Query、Mutation 以及 Subscription 欄位執行此操作。

JSON

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "appsync:GraphQL"
      ],
      "Resource": [
        "arn:aws:appsync:us-west-2:123456789012:apis/YourGraphQLApiId/types/Query/fields/<Field-1>",
        "arn:aws:appsync:us-west-2:123456789012:apis/YourGraphQLApiId/types/Query/fields/<Field-2>",
        "arn:aws:appsync:us-west-2:123456789012:apis/YourGraphQLApiId/types/Mutation/fields/<Field-1>",
        "arn:aws:appsync:us-west-2:123456789012:apis/YourGraphQLApiId/types/Subscription/fields/<Field-1>"
      ]
    }
  ]
}

```

例如，假設您有以下結構描述且您想要限制取得所有文章的存取：

```
schema {
```

```

    query: Query
    mutation: Mutation
  }

  type Query {
    posts:[Post!]!
  }

  type Mutation {
    addPost(id:ID!, title:String!):Post!
  }

```

角色的對應 IAM 政策（例如，您可以連接到 Amazon Cognito 身分集區）如下所示：

JSON

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "appsync:GraphQL"
      ],
      "Resource": [
        "arn:aws:appsync:us-west-2:123456789012:apis/YourGraphQLApiId/types/Query/fields/posts"
      ]
    }
  ]
}

```

OPENID_CONNECT 授權

此授權類型會強制執行 OIDC 相容服務提供的 [OpenID 連線](#) (OIDC) 權杖。您的應用程式可以運用由 OIDC 提供者定義的使用者與權限來控制存取權限。

發行者 URL 是您提供給 to AWS AppSync 的唯一必要組態值（例如 `https://auth.example.com`）。此 URL 必須透過 HTTPS 定址。AWS AppSync 會附加 `/.well-known/openid-configuration` 到發行者 URL，並根據 OpenID Connect Discovery `https://`

`auth.example.com/.well-known/openid-configuration` 規格在 找到 OpenID 組態。 [OpenID](#) 它預期會在此 URL 擷取相容於 [RFC5785](#) 的 JSON 文件。此 JSON 文件必須包含 `jwks_uri` 金鑰，該金鑰指向具有簽署金鑰的 JSON Web Key Set (JWKS) 文件。AWS AppSync 要求 JWKS 包含 `kty` 和 的 JSON 欄位 `kid`。

AWS AppSync 支援各種簽署演算法。

簽署演算法

RS256

RS384

RS512

PS256

PS384

PS512

HS256

HS384

HS512

ES256

ES384

ES512

我們建議您使用 RSA 演算法。由提供者發行的字符必須包含字符發行的時間 (`iat`)，且可包含驗證時的時間 (`auth_time`)。您可以提供 OpenID Connect 組態中的發行時間的 TTL 值 (`iatTTL`) 和身分驗證時間 (`authTTL`)，以供額外驗證。如果您的提供者授權多個應用程式，您也可以提供用於根據用戶端 ID 授權的一般表達式 (`clientId`)。當 OpenID Connect 組態中有 `clientId` 時，AWS AppSync 會要求 與字符中的 `aud` 或 宣告相符 `clientId`，以驗證 `azp` 宣告。

若要驗證多個用戶端 IDs 請使用管道運算子 (「|」)，這是規則表達式中的「或」。例如，如果您的 OIDC 應用程式有四個用戶端具有用戶端 IDs，例如 0A1S2D, 1F4G9H, 1J6L4B, 6GS5MG，只要驗證前三個用戶端 IDs，您會在用戶端 ID 欄位中放置 1F4G9H|1J6L4B|6GS5MG。

如果 API 設定了多種授權類型，AWS AppSync 會將其與 API 組態中指定的發行者 URL 進行比較，以驗證來自請求標頭的 JWT 字符中存在的發行者 (iss 宣告)。不過，當 API 只設定 OPENID_CONNECT 授權時，AWS AppSync 會略過此發行者 URL 驗證步驟。

AMAZON_COGNITO_USER_POOLS 授權

此授權類型會強制執行 Amazon Cognito 使用者集區提供的 OIDC 權杖。您的應用程式可以利用來自另一個 AWS 帳戶的使用者集區和使用集區中的使用者和群組，並將其與 GraphQL 欄位建立關聯，以控制存取。

使用 Amazon Cognito 使用者集區時，您可以建立使用者所屬的群組。此資訊以 JWT 字符進行編碼，您的應用程式會在傳送 GraphQL 操作時，以授權標頭傳送至 AWS AppSync。您可以使用結構描述上的 GraphQL 指令來控制哪些群組可以叫用欄位上的哪些解析程式，進而為您的客戶提供更具控制性的存取權限。

例如，假設您有下列 GraphQL 結構描述：

```
schema {
  query: Query
  mutation: Mutation
}

type Query {
  posts:[Post!]!
}

type Mutation {
  addPost(id:ID!, title:String!):Post!
}
...
```

如果您在 Amazon Cognito 使用者集區 - 部落客和讀者 - 中有兩個群組，而且您想要限制讀者，使其無法新增項目，則結構描述應如下所示：

```
schema {
  query: Query
```

```
mutation: Mutation
}
```

```
type Query {
  posts:[Post!]!
  @aws_auth(cognito_groups: ["Bloggers", "Readers"])
}

type Mutation {
  addPost(id:ID!, title:String!):Post!
  @aws_auth(cognito_groups: ["Bloggers"])
}

...
```

請注意，如果您想要將存取權限預設為特定授予或拒絕授予策略，您可以省略 `@aws_auth` 指令。在透過主控台或以下 CLI 命令建立 GraphQL API 時，您可以在使用者集區組態中指定授予或拒絕授予策略：

```
$ aws appsync --region us-west-2 create-graphql-api --authentication-
type AMAZON_COGNITO_USER_POOLS --name userpoolstest --user-pool-config
'{"userPoolId":"test", "defaultEffect":"ALLOW", "awsRegion":"us-west-2"}'
```

使用其他授權模式

當您新增其他授權模式時，您可以直接在 AWS AppSync GraphQL API 層級設定授權設定（即您可以直接在 `GraphqlApi` 物件上設定 `authenticationType` 的欄位），並在結構描述上做為預設值。這表示沒有特定指示詞的任何類型必須傳遞 API 層級授權設定。

在結構描述層級上，您可以在結構描述上使用指示詞來指定其他授權模式。您可以在結構描述中的個別欄位上指定授權模式。例如，針對 `API_KEY` 授權，您可以在結構描述物件類型定義/欄位上使用 `@aws_api_key`。結構描述欄位和物件類型定義支援下列指示詞：

- `@aws_api_key` - 指定欄位會經由 `API_KEY` 授權。
- `@aws_iam` - 指定欄位會經由 `AWS_IAM` 授權。
- `@aws_oidc` - 指定欄位會經由 `OPENID_CONNECT` 授權。
- `@aws_cognito_user_pools` - 指定欄位會經由 `AMAZON_COGNITO_USER_POOLS` 授權。
- `@aws_lambda` - 指定欄位會經由 `AWS_LAMBDA` 授權。

您無法將 `@aws_auth` 指示詞搭配使用其他授權模式。`@aws_auth` 只能在沒有其他授權模式的 `AMAZON_COGNITO_USER_POOLS` 授權內容中運作。但是，您可以搭配相同的引數，使用 `@aws_cognito_user_pools` 指示詞來取代 `@aws_auth` 指示詞。這兩者的主要差異是您可以在任何欄位和物件類型定義上指定 `@aws_cognito_user_pools`。

為了了解其他授權模式如何運作以及如何在結構描述上指定這些模式，讓我們看看以下結構描述：

```
schema {
  query: Query
  mutation: Mutation
}

type Query {
  getPost(id: ID!): Post
  getAllPosts(): [Post]
  @aws_api_key
}

type Mutation {
  addPost(
    id: ID!
    author: String!
    title: String!
    content: String!
    url: String!
  ): Post!
}

type Post @aws_api_key @aws_iam {
  id: ID!
  author: String
  title: String
  content: String
  url: String
  ups: Int!
  downs: Int!
  version: Int!
}

...
```

對於此結構描述，假設 `AWS_IAM` 是 AWS AppSync GraphQL API 上的預設授權類型。這表示沒有指示詞的欄位會使用 `AWS_IAM` 進行保護。例如，`Query` 類型的 `getPost` 欄位就是這種情況。結

構描述指示詞可讓您使用超過一種授權模式。例如，您可以在 AWS AppSync GraphQL API 上將 API_KEY 設定為額外的授權模式，而且您可以使用 @aws_api_key 指令來標記欄位（例如，在此範例中 getAllPosts）。指示詞會在欄位層級運作，因此您也需要將 API_KEY 存取權限提供給 Post 類型。您可以透過在 Post 類型中使用指示詞標記每個欄位，或是使用 @aws_api_key 指示詞標記 Post 類型，來執行此作業。

若要進一步限制對 Post 類型中欄位的存取，您可以針對 Post 類型中的個別欄位使用指示詞，如下所示。

例如，您可以將 restrictedContent 欄位新增到 Post 類型，並使用 @aws_iam 指示詞限制對該項目進行存取。AWS_IAM 已驗證請求可以存取 restrictedContent，但 API_KEY 請求將無法存取它。

```
type Post @aws_api_key @aws_iam {
  id: ID!
  author: String
  title: String
  content: String
  url: String
  ups: Int!
  downs: Int!
  version: Int!
  restrictedContent: String!
  @aws_iam
}
```

精細定義存取控制

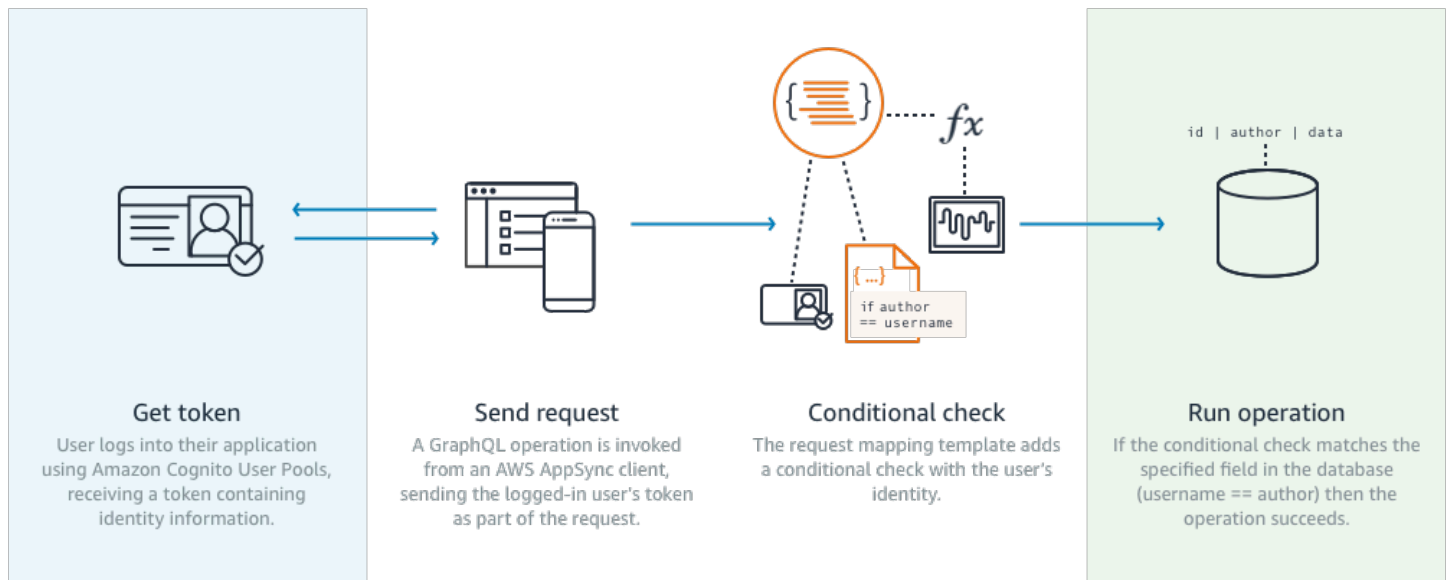
上述資訊示範了如何限制存取權限，或將存取權限授予特定 GraphQL 欄位。若您希望根據特定條件設定資料的存取控制（例如根據進行呼叫的使用者，以及使用者是否擁有資料），您可以在解析程式中使用映射範本。您也可以執行更複雜的商業邏輯；我們會在[篩選資訊](#)中進行說明。

本節說明如何使用 DynamoDB 解析程式映射範本來設定資料的存取控制。

在繼續之前，如果您不熟悉 AWS AppSync 中的映射範本，建議您檢閱 DynamoDB 的[解析程式映射範本參考](#)和解析程式映射範本參考。[DynamoDB](#)

在下列使用 DynamoDB 的範例中，假設您使用上述部落格文章結構描述，並且僅允許建立文章的使用者編輯它。評估程序需使用者獲得應用程式的登入資料，例如使用 Amazon Cognito 使用者集區，然

後將這些登入資料做為 GraphQL 操作的一部分。映射範本接著將取代在條件式陳述式中的登入資料值 (例如使用者名稱)，此資訊接著將與資料庫中的值進行比較。



若要新增此功能，請新增 `editPost` 的 GraphQL 欄位，如下所示：

```
schema {
  query: Query
  mutation: Mutation
}

type Query {
  posts:[Post!]!
}

type Mutation {
  editPost(id:ID!, title:String, content:String):Post
  addPost(id:ID!, title:String!):Post!
}
...
```

`editPost` 的解析程式映射範本 (顯示於本區段末端的範例中) 需要針對您的資料存放區執行邏輯檢查，只允許建立文章的使用者可對其進行編輯。由於這是編輯操作，因此會對應至 DynamoDB `UpdateItem` 中的。在執行此動作前可以使用透過使用者身分驗證傳遞的上下文來執行條件檢查。此資訊存放在 `Identity` 物件中，有以下值：

```
{
  "accountId" : "12321434323",
```

```
"cognitoIdentityPoolId" : "",
"cognitoIdentityId" : "",
"sourceIP" : "",
"caller" : "ThisistheprincipalARN",
"username" : "username",
"userArn" : "Sameasabove"
}
```

若要在 `DynamoDBUpdateItem` 呼叫中使用此物件，您需要將使用者身分資訊存放在資料表中以進行比較。首先，您的 `addPost` 變動需要儲存建立者。其次，您的 `editPost` 變動需在更新前執行條件式檢查。

以下是將使用者身分 `addPost` 儲存為 `Author` 欄的解析程式碼範例：

```
import { util, Context } from '@aws-appsync/utils';
import { put } from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
  const { id: postId, ...item } = ctx.args;
  return put({
    key: { postId },
    item: { ...item, Author: ctx.identity.username },
    condition: { postId: { attributeExists: false } },
  });
}

export const response = (ctx) => ctx.result;
```

請注意，`Author` 屬性從來自於應用程式的 `Identity` 物件發布。

最後，以下是的解析程式碼範例 `editPost`，只有在請求來自建立文章的使用者時，才會更新部落格文章的內容：

```
import { util, Context } from '@aws-appsync/utils';
import { put } from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
  const { id, ...item } = ctx.args;
  return put({
    key: { id },
    item,
    condition: { author: { contains: ctx.identity.username } },
  });
}
```

```
});  
}  
  
export const response = (ctx) => ctx.result;
```

此範例使用覆寫所有值PutItem的，而不是 UpdateItem，但相同的概念適用於condition陳述式區塊。

篩選資訊

在某些情況下，您無法控制資料來源的回應，但又不想在成功寫入或讀取資料來源時，將不必要的資訊傳送給用戶端。如果發生這些情況，您可以使用回應映射範本來篩選資訊。

例如，假設您的部落格文章 DynamoDB 資料表上沒有適當的索引（例如 上的索引Author）。您可以使用下列解析程式：

```
import { util, Context } from '@aws-appsync/utils';  
import { get } from '@aws-appsync/utils/dynamodb';  
  
export function request(ctx) {  
  return get({ key: { ctx.args.id } });  
}  
  
export function response(ctx) {  
  if (ctx.result.author === ctx.identity.username) {  
    return ctx.result;  
  }  
  return null;  
}
```

即使呼叫者不是建立文章的作者，請求處理常式也會擷取項目。為了防止傳回所有資料，回應處理常式會檢查以確保呼叫者符合項目的作者。如果發起人不符合此檢查，只會傳回 null。在某些情況下，您無法控制數據源的響應，但又不想在成功寫入或讀取數據源時將不必要的信息發送給客戶端。I 回應。

資料來源存取

AWS AppSync 會使用 Identity and Access Management ([IAM](#)) 角色和存取政策與資料來源通訊。如果您使用現有角色，則需要新增信任政策，AWS AppSync 才能擔任該角色。信任關係如下所示：

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "appsync.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

縮小角色的存取政策範圍，使其只能對必要的最少資源集合採取行動，是非常重要的。當使用 AppSync 主控台來建立資料來源並建立角色時，將會為您自動完成這些動作。不過，從 IAM 主控台使用內建範例範本在 AWS AppSync 主控台之外建立角色時，許可不會自動縮小資源的範圍，您應該在將應用程式移至生產環境之前執行此動作。

用於保護請求和回應安全的存取控制使用案例

您在[安全](#)章節了解保護 API 的各種授權模式，其中也簡介精細分級授權機制，以了解各項概念與流程。Since AWS AppSync 可讓您透過使用 GraphQL 解析程式[映射範本](#)對資料執行邏輯完整操作，您可以使用使用者身分、條件和資料注入的組合，以非常靈活的方式保護讀取或寫入時的資料。

如果您不熟悉 editing AWS AppSync Resolvers，請檢閱[程式設計指南](#)。

概觀

授予系統中資料的存取權傳統上是透過[存取控制矩陣](#)完成，其中資料列（資源）和資料欄（使用者/角色）的交集是授予的許可。

AWS AppSync 會在您自己的帳戶中使用資源，並將身分（使用者/角色）資訊執行至 GraphQL 請求和回應做為[內容物件](#)，您可以在解析程式中使用。這表示您可依據解析程式邏輯，將權限適當地授與寫入或讀取操作。如果此邏輯位於資源層級，例如只有特定具名使用者或群組可以讀取/寫入特定資料庫資料列，則必須儲存該「授權中繼資料」。AWS AppSync 不會儲存任何資料，因此您必須將此授權中繼資料與資源一起存放，才能計算許可。授權中繼資料通常是 DynamoDB 表之中的屬性（欄），例如擁有者或使用者/群組清單。例如其中可能有讀取器及寫入器屬性。

大致而言，這表示若您由資料來源讀取個別項目，就會在解析程式由資料來源讀取之後，於回應範本執行條件式 `#if () ... #end` 陳述式。檢查通常會使用 `$context.identity` 之中的使用者或群組值，依據讀取操作傳回的授權中繼資料進行成員檢查。對於多筆記錄，例如由資料表 Scan 或 Query 傳回的清單，您將使用類似的使用者或群組值，將條件檢查做為操作的一部分傳送至資料來源。

同樣地，寫入資料時，您會將條件式陳述式套用至動作 (例如 `PutItem` 或 `UpdateItem`)，了解進行變動的使用者或群組是否具有許可。條件式將再次多次使用 `$context.identity` 之中的值，與該項資源的授權中繼資料進行比較。對於要求及回應範本，您也可使用用戶端的自訂標頭執行驗證檢查。

讀取資料

如前所述，執行檢查的授權中繼資料必須與資源儲存，或傳送至 GraphQL 要求 (身分、標頭等等)。為了示範，假設您有下列的 DynamoDB 表格：

ID	Data	PeopleCanAccess	GroupsCanAccess	Owner
123	{my: data,...}	[Mary, Joe]	[Admins, Editors]	Nadia

主索引鍵為 `id`，要存取的資料為 `Data`。其他資料欄是您可以執行以進行授權的檢查範例。`Owner` 會是 `String` 一段時間 `PeopleCanAccess`，並且 `GroupsCanAccess` 會 `String Sets` 如 [DynamoDB 的解析程式映射範本參考](#) 中所述。

在 [解析程式映射範本概觀](#) 之中的圖會顯示回應範本不僅包含內容物件，也包含資料來源結果。對個別項目的 GraphQL 查詢而言，您可使用回應範本檢查使用者是否允許檢視這些結果，否則將傳回授權錯誤訊息。這有時稱為「授權篩選」。對使用掃描或查詢的 GraphQL 查詢傳回清單而言，效能較高的作法是在要求範本執行檢查，並僅於滿足授權條件時傳回資料。實作之後將：

1. `GetItem` - 個別記錄的授權檢查。使用 `#if() ... #end` 陳述式完成。
2. 掃描/查詢操作 - 授權檢查為 `"filter":{"expression":...}` 陳述式。一般檢查為對等 (`attribute = :input`) 或檢查值是否在清單之中 (`contains(attribute, :input)`)。

在 #2 之中，兩種陳述式的 `attribute` 都代表表格之中記錄的欄名稱，例如以上範例中的 `Owner`。您可利用 `#` 簽署和使用 `"expressionNames":{...}` 為此設定別名，但並非必要。`:input` 是您要與資料庫屬性比較值的參考，該屬性將於 `"expressionValues":{...}` 定義。您可參考以下各項範例。

使用案例：擁有者可以讀取

使用以上表格，若您只想傳回資料，了解 `Owner == Nadia` 是否用於個別讀取操作 (GetItem)，您的範本可能會像這樣：

```
#if($context.result["Owner"] == $context.identity.username)
    $utils.toJson($context.result)
#else
    $utils.unauthorized()
#end
```

這裡提出兩點注意事項，在剩餘章節中會用到。首先，如果使用 Amazon Cognito 使用者集區，檢查會使用 `$context.identity.username` 易記的使用者註冊名稱，如果使用 IAM，檢查將是使用者身分（包括 Amazon Cognito 聯合身分）。還有其他值可存放給擁有者，例如唯一的「Amazon Cognito 身分」值，在聯合來自多個位置的登入時非常有用，您應該檢閱[解析程式映射範本內容參考](#)中可用的選項。

第二，以 `$util.unauthorized()` 進行的條件式否則檢查回應為完全可選用的選項，但建議做為設計 GraphQL API 時的最佳實務。

使用案例：硬式編碼特定存取

```
// This checks if the user is part of the Admin group and makes the call
#foreach($group in $context.identity.claims.get("cognito:groups"))
    #if($group == "Admin")
        #set($inCognitoGroup = true)
    #end
#end
#if($inCognitoGroup)
{
    "version" : "2017-02-28",
    "operation" : "UpdateItem",
    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
    },
    "attributeValues" : {
        "owner" : $util.dynamodb.toDynamoDBJson($context.identity.username)
        #foreach( $entry in $context.arguments.entrySet() )
            , "{$entry.key}" : $util.dynamodb.toDynamoDBJson($entry.value)
        #end
    }
}
```

```

}
#else
  $utils.unauthorized()
#end

```

使用案例：篩選結果清單

在前一項範例之中，您可在 `$context.result` 傳回單一項目時對其執行檢查，不過像是掃描等部分操作，將在 `$context.result.items` 傳回多個項目；您在此需要執行驗證篩選，並僅傳回使用者允許檢視的結果。假設 `Owner` 欄位在記錄中設定了 Amazon Cognito IdentityID，則您可以使用下列回應映射範本來篩選，僅顯示使用者擁有的記錄：

```

#set($myResults = [])
#foreach($item in $context.result.items)
  ##For userpools use $context.identity.username instead
  #if($item.Owner == $context.identity.cognitoIdentityId)
    #set($added = $myResults.add($item))
  #end
#end
$utils.toJson($myResults)

```

使用案例：多個人員可以讀取

另一種熱門的授權選擇，就是允許一群人讀取資料。在以下範例中，`"filter": {"expression": ...}` 僅傳回表格掃描的值 (若使用者執行的 GraphQL 查詢列於 `PeopleCanAccess` 集之中)。

```

{
  "version" : "2017-02-28",
  "operation" : "Scan",
  "limit": #if(${context.arguments.count}) $util.toJson($context.arguments.count)
#else 20 #end,
  "nextToken": #if(${context.arguments.nextToken})
$util.toJson($context.arguments.nextToken) #else null #end,
  "filter":{
    "expression": "contains(#peopleCanAccess, :value)",
    "expressionNames": {
      "#peopleCanAccess": "peopleCanAccess"
    },
    "expressionValues": {
      ":value": $util.dynamodb.toDynamoDBJson($context.identity.username)
    }
  }
}

```

```

    }
  }
}
```

使用案例：群組可以讀取

類似於上一項使用案例，可能只有單一或更多群組的人，有權讀取資料庫之中的特定項目。"expression": "contains()" 操作的使用方式類似，不過這是所有群組的邏輯 OR，使用者可能是其中的一部分，需要在設定成員資格時加以考量。在此案例中，我們針對使用者所屬的各個群組建立以下 \$expression 陳述式，然後將其傳送至篩選器：

```

#set($expression = "")
#set($expressionValues = {})
#foreach($group in $context.identity.claims.get("cognito:groups"))
    #set( $expression = "${expression} contains(groupsCanAccess, :var
$foreach.count )" )
    #set( $val = {})
    #set( $test = $val.put("S", $group))
    #set( $values = $expressionValues.put(":var$foreach.count", $val))
    #if ( $foreach.hasNext )
    #set( $expression = "${expression} OR" )
    #end
#end
#end
{
  "version" : "2017-02-28",
  "operation" : "Scan",
  "limit": #if(${context.arguments.count}) $util.toJson($context.arguments.count)
#else 20 #end,
  "nextToken": #if(${context.arguments.nextToken})
$util.toJson($context.arguments.nextToken) #else null #end,
  "filter":{
    "expression": "$expression",
    "expressionValues": $utils.toJson($expressionValues)
  }
}
```

寫入資料

在變動寫入資料一定是在要求映射範本進行控制。就 DynamoDB 資料來源而言，關鍵在於使用適當的 "condition":{"expression"...}"，依據該表格之中的授權中繼資料執行驗證。在[安全性](#)中，我們提供範例讓您用來檢查資料表中的 Author 欄位。本節使用案例將探索更多使用案例。

使用案例：多個擁有者

使用之前的範例表格圖示，假設 PeopleCanAccess 清單

```
{
  "version" : "2017-02-28",
  "operation" : "UpdateItem",
  "key" : {
    "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
  },
  "update" : {
    "expression" : "SET meta = :meta",
    "expressionValues": {
      ":meta" : $util.dynamodb.toDynamoDBJson($ctx.args.meta)
    }
  },
  "condition" : {
    "expression" : "contains(Owner,:expectedOwner)",
    "expressionValues" : {
      ":expectedOwner" :
$util.dynamodb.toDynamoDBJson($context.identity.username)
    }
  }
}
```

使用案例：群組可以建立新的記錄

```
#set($expression = "")
#set($expressionValues = {})
#foreach($group in $context.identity.claims.get("cognito:groups"))
  #set( $expression = "${expression} contains(groupsCanAccess, :var
$foreach.count )" )
  #set( $val = {})
  #set( $test = $val.put("S", $group))
  #set( $values = $expressionValues.put(":var$foreach.count", $val))
  #if ( $foreach.hasNext )
    #set( $expression = "${expression} OR" )
  #end
#end
{
  "version" : "2017-02-28",
  "operation" : "PutItem",
  "key" : {
```

```

    ## If your table's hash key is not named 'id', update it here. **
    "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
    ## If your table has a sort key, add it as an item here. **
  },
  "attributeValues" : {
    ## Add an item for each field you would like to store to Amazon DynamoDB. **
    "title" : $util.dynamodb.toDynamoDBJson($ctx.args.title),
    "content": $util.dynamodb.toDynamoDBJson($ctx.args.content),
    "owner": $util.dynamodb.toDynamoDBJson($context.identity.username)
  },
  "condition" : {
    "expression": $util.toJson("attribute_not_exists(id) AND $expression"),
    "expressionValues": $utils.toJson($expressionValues)
  }
}

```

使用案例：群組可以更新現有記錄

```

#set($expression = "")
#set($expressionValues = {})
#foreach($group in $context.identity.claims.get("cognito:groups"))
  #set( $expression = "${expression} contains(groupsCanAccess, :var
$foreach.count )" )
  #set( $val = {})
  #set( $test = $val.put("S", $group))
  #set( $values = $expressionValues.put(":var$foreach.count", $val))
  #if ( $foreach.hasNext )
  #set( $expression = "${expression} OR" )
  #end
#end
{
  "version" : "2017-02-28",
  "operation" : "UpdateItem",
  "key" : {
    "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
  },
  "update":{
    "expression" : "SET title = :title, content = :content",
    "expressionValues": {
      ":title" : $util.dynamodb.toDynamoDBJson($ctx.args.title),
      ":content" : $util.dynamodb.toDynamoDBJson($ctx.args.content)
    }
  },

```

```
"condition" : {
  "expression": $util.toJson($expression),
  "expressionValues": $utils.toJson($expressionValues)
}
```

公有和私有記錄

您也可以使用條件式篩選器，選擇將資料標示為公開、私人或其他布林檢查。並於之後合併為回應範本之中授權篩選的一部分。此項檢查適合用於暫時隱藏資料，或由檢視之中移除資料，不必嘗試控制群組成員資格。

例如，假設您在名為 `public` 的 DynamoDB 表格，以 `yes` 或 `no` 值在各個項目新增屬性。下列回應範本可於 `GetItem` 呼叫時使用；只有在使用者位於具有權限的群組之中，且該資料標示為公開的情況下，資料才會顯示：

```
#set($permissions = $context.result.GroupsCanAccess)
#set($claimPermissions = $context.identity.claims.get("cognito:groups"))

#foreach($per in $permissions)
  #foreach($cgroups in $claimPermissions)
    #if($cgroups == $per)
      #set($hasPermission = true)
    #end
  #end
#end

#if($hasPermission && $context.result.public == 'yes')
  $utils.toJson($context.result)
#else
  $utils.unauthorized()
#end
```

以上程式碼也可使用邏輯 OR (`||`)，在使用者擁有權限或記錄為公開的情況下能夠讀取記錄：

```
#if($hasPermission || $context.result.public == 'yes')
  $utils.toJson($context.result)
#else
  $utils.unauthorized()
#end
```

一般而言，您在執行授權檢查時，運算子 `==`、`!=`、`&&` 及 `||` 都相當實用。

即時資料

您可在用戶端訂閱時，使用本文之前介紹的相同技巧，將精細分級的存取控制套用至 GraphQL 訂閱。您將解析程式附加至訂閱欄位，在此由資料來源查詢資料，並於要求或回應映射範本時執行條件式邏輯。只要資料結構符合 GraphQL 訂閱之中傳回的類型，您也可以傳回其他資料給用戶端，例如訂閱的初始結果。

使用案例：使用者只能訂閱特定對話

這是常見的即時資料使用案例，其中 GraphQL 訂閱正在建構訊息或私人交談應用程式。建立具有多位使用者的交談應用程式時，可能在兩位或多位使用者之間產生對話。這些使用者可能會聚集成為私人或公開的「聊天室」。在這種情況下，您可能只想授權使用者訂閱其有權存取的對話（一對一或群組之中）。以下範例示範單一使用者向他人傳送私人訊息的簡單使用案例。設定有兩個 Amazon DynamoDB 資料表：

- 訊息表格：(主索引鍵) toUser、(排序索引鍵) id
- 權限表格：(主索引鍵) username

訊息表格儲存透過 GraphQL 變動傳送的實際訊息。權限表格會由 GraphQL 訂閱檢查，在用戶端連線時進行授權。以下範例假設您使用下列 GraphQL 結構描述：

```
input CreateUserPermissionsInput {
  user: String!
  isAuthorizedForSubscriptions: Boolean
}

type Message {
  id: ID
  toUser: String
  fromUser: String
  content: String
}

type MessageConnection {
  items: [Message]
  nextToken: String
}

type Mutation {
  sendMessage(toUser: String!, content: String!): Message
}
```

```

    createUserPermissions(input: CreateUserPermissionsInput!): UserPermissions
    updateUserPermissions(input: UpdateUserPermissionInput!): UserPermissions
  }

  type Query {
    getMyMessages(first: Int, after: String): MessageConnection
    getUserPermissions(user: String!): UserPermissions
  }

  type Subscription {
    newMessage(toUser: String!): Message
      @aws_subscribe(mutations: ["sendMessage"])
  }

  input UpdateUserPermissionInput {
    user: String!
    isAuthorizedForSubscriptions: Boolean
  }

  type UserPermissions {
    user: String
    isAuthorizedForSubscriptions: Boolean
  }

  schema {
    query: Query
    mutation: Mutation
    subscription: Subscription
  }

```

下面 `createUserPermissions()` 未涵蓋一些標準操作，例如，以說明訂閱解析程式，但是 DynamoDB 解析程式的標準實作。我們將重點放在訂閱授權流程與解析程式。使用下列要求範本，讓使用者將訊息傳送給其他使用者，將解析程式連接至 `sendMessage()` 欄位，然後選取 Messages (訊息) 資料表資料來源：

```

{
  "version" : "2017-02-28",
  "operation" : "PutItem",
  "key" : {
    "toUser" : $util.dynamodb.toDynamoDBJson($ctx.args.toUser),
    "id" : $util.dynamodb.toDynamoDBJson($util.autoId())
  },
  "attributeValues" : {

```

```

    "fromUser" : $util.dynamodb.toDynamoDBJson($context.identity.username),
    "content" : $util.dynamodb.toDynamoDBJson($ctx.args.content),
  }
}

```

在此範例中，我們使用 `$context.identity.username`。這會傳回 AWS Identity and Access Management 或 Amazon Cognito 使用者的使用者資訊。回應範本為簡單的 `$util.toJson($ctx.result)` 傳遞。儲存後回到結構描述頁面。然後連接解析程式以用於 `newMessage()` 訂閱，使用許可資料表做為資料來源，以及下列要求映射範本：

```

{
  "version": "2018-05-29",
  "operation": "GetItem",
  "key": {
    "username": $util.dynamodb.toDynamoDBJson($ctx.identity.username),
  },
}

```

之後請使用下列回應映射範本，以許可資料表的資料執行授權檢查：

```

#if(! ${context.result})
  $utils.unauthorized()
#elseif(${context.identity.username} != ${context.arguments.toUser})
  $utils.unauthorized()
#elseif(! ${context.result.isAuthorizedForSubscriptions})
  $utils.unauthorized()
#else
  ##User is authorized, but we return null to continue
  null
#end

```

在此情況下，您會進行三項授權檢查。第一項檢查確保結果已返回。第二項檢查確保使用者並未訂閱其他人的訊息。第三個會檢查 `isAuthorizedForSubscriptions` 存放為的 DynamoDB 屬性，以確保使用者能夠訂閱任何欄位 B00L。

若要測試物件，您可以使用 Amazon Cognito 使用者集區和名為「Nadia」的使用者登入 AWS AppSync 主控台，然後執行下列 GraphQL 訂閱：

```

subscription AuthorizedSubscription {
  newMessage(toUser: "Nadia") {

```

```
    id
    toUser
    fromUser
    content
  }
}
```

如果在許可資料表之中，存在 Nadia 的 username 索引鍵屬性記錄，且 `isAuthorizedForSubscriptions` 設定為 `true`，您就可以獲得成功回應。如果您在以上的 username 查詢嘗試不同 `newMessage()`，就會傳回錯誤。

使用 AWS WAF 來保護您的 AWS AppSync APIs

AWS WAF 是一種 Web 應用程式防火牆，可協助保護 Web 應用程式和 APIs 免受攻擊。它可讓您設定一組稱為 Web 存取控制清單 (Web ACL) 的規則，以根據您定義的可自訂 Web 安全規則和條件來允許、封鎖或監控（計數）Web 請求。當您將 AWS AppSync API 與整合時 AWS WAF，您可以更好地控制和查看 API 接受的 HTTP 流量。若要進一步了解 AWS WAF，請參閱《AWS WAF 開發人員指南》中的 [AWS WAF 運作方式](#)。

您可以使用 AWS WAF 來保護 AppSync API 免受常見的 Web 入侵，例如 SQL Injection 和跨網站指令碼 (XSS) 攻擊。這些可能會影響 API 可用性和效能、危及安全性，或耗用過多的資源。例如，您可以建立規則，允許或封鎖來自指定 IP 地址、來自 CIDR 區塊的請求，或源自特定國家或區域，其中包含惡意 SQL 程式碼或惡意指令碼的請求。

您也可以建立規則，以符合在 HTTP 標頭、方法、查詢字串、URI 和請求本文 (限於前 8 KB) 的指定字串或常規表達式模式。此外，您可以建立規則以封鎖來自特定使用者代理程式、惡意機器人和內容抓取器的攻擊。例如，您可以使用以速率為基礎的規則，以指定每個用戶端 IP 在尾隨、持續更新的 5 分鐘期間，允許的 Web 請求數。

若要進一步了解支援的規則類型和其他 AWS WAF 功能，請參閱 [AWS WAF 開發人員指南](#) 和 [AWS WAF API 參考](#)。

Important

AWS WAF 是您防範 Web 入侵的第一道防線。在 API 上啟用 AWS WAF 時，會在其他存取控制功能之前評估 AWS WAF 規則，例如 API 金鑰授權、IAM 政策、OIDC 權杖和 Amazon Cognito 使用者集區。

將 AppSync API 與 整合 AWS WAF

您可以使用 AWS CLI AWS CloudFormation、或任何其他相容的用戶端 AWS WAF AWS Management Console，將 Appsync API 與 整合。

將 AWS AppSync API 與 整合 AWS WAF

1. 建立 AWS WAF Web ACL。如需使用[AWS WAF 主控台](#)的詳細步驟，請參閱[建立 Web ACL](#)。
2. 定義 Web ACL 的規則。規則會在建立 Web ACL 的過程中定義。如需如何建構規則的資訊，請參閱[AWS WAF 規則](#)。如需您可以為 AWS AppSync API 定義的實用規則範例，請參閱[建立 Web ACL 的規則](#)。
3. 將 Web ACL 與 AWS AppSync API 建立關聯。您可以在 [AWS WAF 主控台](#)或 [AppSync 主控台](#)中執行此步驟。
 - 若要將 Web ACL 與 AWS WAF 主控台中的 AWS AppSync API 建立關聯，請遵循《AWS WAF 開發人員指南》中的[將 Web ACL 與 AWS 資源建立關聯或取消關聯](#)的指示。
 - 在主控台中 AWS AppSync 將 Web ACL 與 AWS AppSync API 建立關聯
 - a. 登入 AWS Management Console 並開啟 [AppSync 主控台](#)。
 - b. 選擇您要與 Web ACL 建立關聯的 API。
 - c. 在導覽窗格中，選擇設定。
 - d. 在 Web 應用程式防火牆區段中，開啟啟用 AWS WAF。
 - e. 在 Web ACL 下拉式清單中，選擇要與您的 API 建立關聯的 Web ACL 名稱。
 - f. 選擇儲存，將 Web ACL 與您的 API 建立關聯。

Note

在 AWS WAF 主控台中建立 Web ACL 之後，可能需要幾分鐘的時間才能使用新的 Web ACL。如果您在 Web 應用程式防火牆功能表中看不到新建立的 Web ACL，請等待幾分鐘，然後重試將 Web ACL 與您的 API 建立關聯的步驟。

 Note

AWS WAF 整合僅支援即時端點Subscription registration message的事件。AWS AppSync 會以錯誤訊息回應，而不是任何 Subscription registration message 封鎖start_ack的訊息 AWS WAF。

將 Web ACL 與 AWS AppSync API 建立關聯後，您將使用 AWS WAF APIs管理 Web ACL。除非您想要將 AWS AppSync API 與不同的 Web ACL 建立關聯，否則您不需要重新建立 Web ACL 與 AWS AppSync API 的關聯。

建立 Web ACL 的規則

規則定義如何檢查 Web 請求，以及當 Web 請求符合檢查條件時該怎麼做。規則本身不存在於 AWS WAF。您可以在規則群組中或定義規則的 Web ACL 中，依名稱存取規則。如需詳細資訊，請參閱 [AWS WAF 規則](#)。下列範例示範如何定義和關聯適用於保護 AppSync API 的規則。

Example 限制請求內文大小的 Web ACL 規則

以下是限制請求內文大小的規則範例。在 AWS WAF 主控台中建立 Web ACL 時，這會輸入規則 JSON 編輯器。

```
{
  "Name": "BodySizeRule",
  "Priority": 1,
  "Action": {
    "Block": {}
  },
  "Statement": {
    "SizeConstraintStatement": {
      "ComparisonOperator": "GE",
      "FieldToMatch": {
        "Body": {}
      },
      "Size": 1024,
      "TextTransformations": [
        {
          "Priority": 0,
          "Type": "NONE"
        }
      ]
    }
  }
}
```

```

    }
  },
  "VisibilityConfig": {
    "CloudWatchMetricsEnabled": true,
    "MetricName": "BodySizeRule",
    "SampledRequestsEnabled": true
  }
}

```

使用上述範例規則建立 Web ACL 之後，您必須將其與 AppSync API 建立關聯。除了使用 之外 AWS Management Console，您也可以 AWS CLI 執行下列命令，在 中執行此步驟。

```
aws waf associate-web-acl --web-acl-id waf-web-acl-arn --resource-arn appsync-api-arn
```

變更可能需要幾分鐘的時間才能傳播，但在執行此命令之後，包含大於 1024 個位元組內文的請求將被拒絕 AWS AppSync。

Note

在 AWS WAF 主控台中建立新的 Web ACL 之後，Web ACL 可能需要幾分鐘的時間才能與 API 建立關聯。如果您執行 CLI 命令並發生錯誤 `WAFUnavailableEntityException`，請等待幾分鐘，然後重試執行命令。

Example Web ACL 規則可限制來自單一 IP 地址的請求

以下是將 AppSync API 從單一 IP 地址調節至 100 個請求的規則範例。在 AWS WAF 主控台中使用以速率為基礎的規則建立 Web ACL 時，這會輸入規則 JSON 編輯器。

```

{
  "Name": "Throttle",
  "Priority": 0,
  "Action": {
    "Block": {}
  },
  "VisibilityConfig": {
    "SampledRequestsEnabled": true,
    "CloudWatchMetricsEnabled": true,
    "MetricName": "Throttle"
  },
  "Statement": {

```

```

    "RateBasedStatement": {
      "Limit": 100,
      "AggregateKeyType": "IP"
    }
  }
}

```

使用上述範例規則建立 Web ACL 之後，您必須將其與 AppSync API 建立關聯。您可以執行下列命令 AWS CLI，在 中執行此步驟。

```
aws waf associate-web-acl --web-acl-id waf-web-acl-arn --resource-arn appsync-api-arn
```

Example Web ACL 規則，以防止 GraphQL __結構描述對 API 的自我檢查查詢

以下是防止 GraphQL __schema 向 API 進行自我檢查的規則範例。任何包含字串 "__schema" 的 HTTP 內文都會遭到封鎖。在 AWS WAF 主控台中建立 Web ACL 時，這會輸入規則 JSON 編輯器。

```

{
  "Name": "BodyRule",
  "Priority": 5,
  "Action": {
    "Block": {}
  },
  "VisibilityConfig": {
    "SampledRequestsEnabled": true,
    "CloudWatchMetricsEnabled": true,
    "MetricName": "BodyRule"
  },
  "Statement": {
    "ByteMatchStatement": {
      "FieldToMatch": {
        "Body": {}
      },
      "PositionalConstraint": "CONTAINS",
      "SearchString": "__schema",
      "TextTransformations": [
        {
          "Type": "NONE",
          "Priority": 0
        }
      ]
    }
  }
}

```



```
}  
}
```

使用上述範例規則建立 Web ACL 之後，您必須將其與 AppSync API 建立關聯。您可以執行下列命令 AWS CLI，在 中執行此步驟。

```
aws waf associate-web-acl --web-acl-id waf-web-acl-arn --resource-arn appsync-api-arn
```

安全 in AWS AppSync

的雲端安全性 AWS 是最高優先順序。身為 AWS 客戶，您可以受益於資料中心和網路架構，這些架構專為符合最安全敏感組織的需求而建置。

安全性是 AWS 與您之間共同責任。[共同責任模型](#)將其描述為雲端的安全性和雲端中的安全性：

- 雲端的安全性 – AWS 負責保護在 AWS Cloud 中執行 AWS 服務的基礎設施。AWS 也為您提供可安全使用的服務。作為[AWS 合規計劃](#)的一部分，第三方稽核人員會定期測試和驗證我們安全的有效性。若要了解適用於 AWS AppSync 的合規計劃，請參閱[AWS 合規計劃的服務範圍](#)。
- 雲端的安全性 – 您的責任取決於您使用 AWS 的服務。您也必須對其他因素負責，包括資料的機密性、您公司的要求和適用法律和法規。

本文件可協助您了解如何在使用 AWS AppSync 時套用共同責任模型。下列主題說明如何 configure AWS AppSync 以符合您的安全與合規目標。您也會了解如何使用其他 AWS 服務來協助您監控和保護 AWS AppSync 資源。

主題

- [in AWS AppSync 資料保護](#)
- [for AWS AppSync 的合規驗證](#)
- [基礎設施安全 in AWS AppSync](#)
- [in AWS AppSync 的恢復能力](#)
- [for AWS AppSync 的身分和存取管理](#)
- [使用 Logging AWS AppSync API 呼叫 AWS CloudTrail](#)
- [的安全最佳實務 AWS AppSync](#)

in AWS AppSync 資料保護

AWS [共同責任模型](#)適用於 in AWS AppSync 的資料保護。如此模型所述，AWS 負責保護執行所有的全球基礎設施 AWS 雲端。您負責維護在此基礎設施上託管內容的控制權。您也同時負責所使用 AWS 服務的安全組態和管理任務。如需資料隱私權的詳細資訊，請參閱[資料隱私權常見問答集](#)。如需有關歐洲資料保護的相關資訊，請參閱 AWS 安全性部落格上的 [AWS 共同的責任模型和 GDPR](#) 部落格文章。

基於資料保護目的，我們建議您保護 AWS 帳戶 登入資料，並使用 AWS IAM Identity Center 或 AWS Identity and Access Management (IAM) 設定個別使用者。如此一來，每個使用者都只會獲得授與完成其任務所必須的許可。我們也建議您採用下列方式保護資料：

- 每個帳戶均要使用多重要素驗證 (MFA)。
- 使用 SSL/TLS 與 AWS 資源通訊。我們需要 TLS 1.2 並建議使用 TLS 1.3。
- 使用 設定 API 和使用者活動記錄 AWS CloudTrail。如需有關使用 CloudTrail 追蹤擷取 AWS 活動的資訊，請參閱AWS CloudTrail 《使用者指南》中的[使用 CloudTrail 追蹤](#)。
- 使用 AWS 加密解決方案，以及其中的所有預設安全控制 AWS 服務。
- 使用進階的受管安全服務 (例如 Amazon Macie)，協助探索和保護儲存在 Amazon S3 的敏感資料。
- 如果您在 AWS 透過命令列界面或 API 存取 時需要 FIPS 140-3 驗證的密碼編譯模組，請使用 FIPS 端點。如需有關 FIPS 和 FIPS 端點的更多相關資訊，請參閱[聯邦資訊處理標準 \(FIPS\) 140-3](#)。

我們強烈建議您絕對不要將客戶的電子郵件地址等機密或敏感資訊，放在標籤或自由格式的文字欄位中，例如名稱欄位。這包括當您使用 AWS AppSync 或使用主控台、API AWS CLI或其他 AWS 服務 AWS SDKs 時。您在標籤或自由格式文字欄位中輸入的任何資料都可能用於計費或診斷日誌。如果您提供外部伺服器的 URL，我們強烈建議請勿在驗證您對該伺服器請求的 URL 中包含憑證資訊。

動態加密

AWS AppSync與所有 AWS 服務一樣，使用 AWS 發佈的 APIs 和 SDKs 時，會使用 TLS1.2 及更高版本進行通訊。

AWS AppSync 搭配 Amazon DynamoDB 等其他 AWS 服務使用 可確保傳輸中的加密：除非另有說明，否則所有 AWS 服務都使用 TLS 1.2 及更高版本彼此通訊。對於使用 Amazon EC2 或 CloudFront 的解析程式，您有責任確認 TLS (HTTPS) 已設定且安全。如需在 Amazon EC2 中設定 HTTPS 的資訊，請參閱《Amazon EC2 使用者指南》中的[在 Amazon Linux 2 上設定 SSL/TLS](#)。如需有關在 CloudFront 上設定 HTTPS 的資訊，請參閱 [CloudFront 使用者指南中的 Amazon CloudFront 中的 HTTPS](#)。CloudFront

for AWS AppSync 的合規驗證

第三方稽核人員會在多個合規計劃中評估 AWS AppSync 的安全性和 AWS 合規性。AWS AppSync 符合 SOC、PCI、HIPAA/HIPAA BAA、IRAP、C5、ENS High、OSPAR 和 HITRUST CSF 計劃。

若要了解 是否 AWS 服務 在特定合規計劃的範圍內，請參閱[AWS 服務 合規計劃](#)然後選擇您感興趣的合規計劃。如需一般資訊，請參閱 [AWS Compliance Programs](#)。

您可以使用 下載第三方稽核報告 AWS Artifact。如需詳細資訊，請參閱[下載](#) 中的 [AWS Artifact](#)報告。

您使用 時的合規責任 AWS 服務 取決於資料的機密性、您公司的合規目標，以及適用的法律和法規。AWS 提供下列資源來協助合規：

- [安全合規與治理](#) - 這些解決方案實作指南內容討論了架構考量，並提供部署安全與合規功能的步驟。
- [HIPAA 合格服務參考](#) - 列出 HIPAA 合格服務。並非所有 AWS 服務 都符合 HIPAA 資格。
- [AWS 合規資源](#) - 此工作手冊和指南集合可能適用於您的產業和位置。
- [AWS 客戶合規指南](#) - 透過合規的角度了解共同責任模型。本指南摘要說明跨多個架構（包括國家標準技術研究所 (NIST)、支付卡產業安全標準委員會 (PCI) 和國際標準化組織 (ISO)) 保護 AWS 服務 和映射指南至安全控制的最佳實務。
- 《AWS Config 開發人員指南》中的[使用 規則評估資源](#) - AWS Config 服務會評估資源組態符合內部實務、產業準則和法規的程度。
- [AWS Security Hub](#) - 這 AWS 服務 可讓您全面檢視其中的安全狀態 AWS。Security Hub 使用安全控制，可評估您的 AWS 資源並檢查您的法規遵循是否符合安全業界標準和最佳實務。如需支援的服務和控制清單，請參閱「[Security Hub 控制參考](#)」。
- [Amazon GuardDuty](#) - 這會監控您的環境是否有可疑和惡意活動，以 AWS 服務 偵測對您 AWS 帳戶、工作負載、容器和資料的潛在威脅。GuardDuty 可滿足特定合規架構所規定的入侵偵測需求，以協助您因應 PCI DSS 等各種不同的合規需求。
- [AWS Audit Manager](#) - 這 AWS 服務 可協助您持續稽核 AWS 用量，以簡化您管理風險的方式，以及符合法規和業界標準的方式。

基礎設施安全 in AWS AppSync

作為受管服務，AWS AppSync 受到 AWS 全球網路安全的保護。如需 AWS 安全服務以及 如何 AWS 保護基礎設施的相關資訊，請參閱[AWS 雲端安全](#)。若要使用基礎設施安全最佳實務來設計您的 AWS 環境，請參閱安全支柱 AWS Well-Architected Framework 中的[基礎設施保護](#)。

您可以使用 AWS 已發佈的 API 呼叫，透過網路存取 AWS AppSync。使用者端必須支援下列專案：

- Transport Layer Security (TLS)。我們需要 TLS 1.2 並建議使用 TLS 1.3。
- 具備完美轉送私密(PFS)的密碼套件，例如 DHE (Ephemeral Diffie-Hellman)或 ECDHE (Elliptic Curve Ephemeral Diffie-Hellman)。現代系統(如 Java 7 和更新版本)大多會支援這些模式。

此外，請求必須使用存取金鑰 ID 和與 IAM 主體相關聯的私密存取金鑰來簽署。或者，您可以透過 [AWS Security Token Service](#) (AWS STS) 來產生暫時安全憑證來簽署請求。

in AWS AppSync 的恢復能力

AWS 全球基礎設施是以 AWS 區域和可用區域為基礎建置的。AWS 區域提供多個實體隔離和隔離的可用區域，這些區域以低延遲、高輸送量和高度備援的網路連接。透過可用區域，您可以設計與操作的應用程式和資料庫，在可用區域之間自動容錯移轉而不會發生中斷。可用區域的可用性、容錯能力和擴展能力，均較單一或多個資料中心的傳統基礎設施還高。

如需 AWS 區域和可用區域的詳細資訊，請參閱 [AWS 全球基礎設施](#)。

除了 AWS 全球基礎設施之外，AWS AppSync 允許使用 AWS CloudFormation 範本定義大多數資源；如需使用 AWS CloudFormation 範本來宣告 AWS AppSync 資源的範例，請參閱 AWS 部落格上的 [AWS AppSync 管道解析程式的實際使用案例](#) 和 [AWS CloudFormation 使用者指南](#)。

for AWS AppSync 的身分和存取管理

AWS Identity and Access Management (IAM) 是一種 AWS 服務，可協助管理員安全地控制對 AWS 資源的存取。IAM 管理員可控制誰可以進行身分驗證（登入）和授權（具有許可）來使用 AWS AppSync 資源。IAM 是您可以免費使用 AWS 服務的。

主題

- [目標對象](#)
- [使用身分驗證](#)
- [使用政策管理存取權](#)
- [How AWS AppSync 可與 IAM 搭配使用](#)
- [AWS AppSync 的身分型政策](#)
- [故障診斷 AWS AppSync 身分和存取](#)

目標對象

您使用 AWS Identity and Access Management (IAM) 的方式會有所不同，取決於您在 AWS AppSync 中執行的工作。

服務使用者 – 如果您使用 AWS AppSync 服務來執行任務，管理員會為您提供所需的登入資料和許可。當您使用 more AWS AppSync 功能來執行工作時，您可能需要額外的許可。了解存取許可的管理

方式可協助您向管理員請求正確的許可。如果您無法存取 AWS AppSync 中的功能，請參閱 [故障診斷 AWS AppSync 身分和存取](#)。

服務管理員 – 如果您在公司負責 AWS AppSync 資源，您可能擁有 AWS AppSync 的完整存取權。您的任務是判斷服務使用者應存取的 AWS AppSync 功能和資源。接著，您必須將請求提交給您的 IAM 管理員，來變更您服務使用者的許可。檢閱此頁面上的資訊，了解 IAM 的基本概念。若要進一步了解貴公司如何使用 IAM with AWS AppSync，請參閱 [How AWS AppSync 可與 IAM 搭配使用](#)。

IAM 管理員 – 如果您是 IAM 管理員，建議您了解撰寫政策以管理 AWS AppSync 存取的詳細資訊。若要檢視您可以在 IAM 中使用的 example AWS AppSync 身分型政策，請參閱 [AWS AppSync 的身分型政策](#)。

使用身分驗證

身分驗證是您 AWS 使用身分憑證登入的方式。您必須以 AWS 帳戶根使用者身分、IAM 使用者身分或擔任 IAM 角色來驗證（登入 AWS）。

您可以使用透過身分來源提供的憑證，以聯合身分 AWS 身分登入。AWS IAM Identity Center (IAM Identity Center) 使用者、您的公司的單一登入身分驗證，以及您的 Google 或 Facebook 登入資料，都是聯合身分的範例。您以聯合身分登入時，您的管理員先前已設定使用 IAM 角色的聯合身分。當您使用聯合 AWS 身分存取時，您會間接擔任角色。

根據您的使用者類型，您可以登入 AWS Management Console 或 AWS 存取入口網站。如需登入的詳細資訊 AWS，請參閱 AWS 登入《使用者指南》中的 [如何登入您的 AWS 帳戶](#)。

如果您以 AWS 程式設計方式存取，AWS 會提供軟體開發套件 (SDK) 和命令列界面 (CLI)，以使用您的登入資料以密碼編譯方式簽署您的請求。如果您不使用 AWS 工具，則必須自行簽署請求。如需使用建議的方法自行簽署請求的詳細資訊，請參閱《IAM 使用者指南》中的 [適用於 API 請求的 AWS Signature 第 4 版](#)。

無論您使用何種身分驗證方法，您可能都需要提供額外的安全性資訊。例如，AWS 建議您使用多重驗證 (MFA) 來提高帳戶的安全性。如需更多資訊，請參閱《AWS IAM Identity Center 使用者指南》中的 [多重要素驗證](#) 和《IAM 使用者指南》中的 [IAM 中的 AWS 多重要素驗證](#)。

AWS 帳戶 根使用者

建立時 AWS 帳戶，您會從一個登入身分開始，該身分可完整存取帳戶中的所有 AWS 服務和資源。此身分稱為 AWS 帳戶 Theroot 使用者，可透過使用您用來建立帳戶的電子郵件地址和密碼登入來存取。強烈建議您不要以根使用者處理日常任務。保護您的根使用者憑證，並將其用來執行只能由根使用者執行的任務。如需這些任務的完整清單，了解需以根使用者登入的任務，請參閱 IAM 使用者指南中的 [需要根使用者憑證的任務](#)。

聯合身分

根據最佳實務，要求人類使用者，包括需要管理員存取權的使用者，使用聯合身分提供者 AWS 服務來使用臨時憑證來存取。

聯合身分是來自您的企業使用者目錄、Web 身分提供者、AWS Directory Service、Identity Center 目錄或任何使用透過身分來源提供的登入資料 AWS 服務存取的使用者。當聯合身分存取時 AWS 帳戶，它們會擔任角色，而角色會提供臨時登入資料。

對於集中式存取權管理，我們建議您使用 AWS IAM Identity Center。您可以在 IAM Identity Center 中建立使用者和群組，也可以連接並同步到您自己的身分來源中的一組使用者 AWS 帳戶和群組，以便在所有和應用程式中使用。如需 IAM Identity Center 的詳細資訊，請參閱 AWS IAM Identity Center 使用者指南中的[什麼是 IAM Identity Center？](#)。

IAM 使用者和群組

[IAM 使用者](#)是中的身分 AWS 帳戶，具有單一人員或應用程式的特定許可。建議您盡可能依賴臨時憑證，而不是擁有建立長期憑證（例如密碼和存取金鑰）的 IAM 使用者。但是如果特定使用案例需要擁有長期憑證的 IAM 使用者，建議您輪換存取金鑰。如需更多資訊，請參閱[IAM 使用者指南](#)中的為需要長期憑證的使用案例定期輪換存取金鑰。

[IAM 群組](#)是一種指定 IAM 使用者集合的身分。您無法以群組身分簽署。您可以使用群組來一次為多名使用者指定許可。群組可讓管理大量使用者許可的程序變得更為容易。例如，您可以擁有一個名為 IAMAdmins 的群組，並給予該群組管理 IAM 資源的許可。

使用者與角色不同。使用者只會與單一人員或應用程式建立關聯，但角色的目的是在由任何需要它的人員取得。使用者擁有永久的長期憑證，但角色僅提供臨時憑證。如需更多資訊，請參閱《IAM 使用者指南》中的[IAM 使用者的使用案例](#)。

IAM 角色

[IAM 角色](#)是具有特定許可 AWS 帳戶的身分。它類似 IAM 使用者，但不與特定的人員相關聯。若要暫時在中擔任 IAM 角色 AWS Management Console，您可以從[使用者切換至 IAM 角色（主控台）](#)。您可以透過呼叫 AWS CLI 或 AWS API 操作或使用自訂 URL 來擔任角色。如需使用角色的方法詳細資訊，請參閱《IAM 使用者指南》中的[擔任角色的方法](#)。

使用臨時憑證的 IAM 角色在下列情況中非常有用：

- 聯合身分使用者存取 — 如需向聯合身分指派許可，請建立角色，並為角色定義許可。當聯合身分進行身分驗證時，該身分會與角色建立關聯，並獲授予由角色定義的許可。如需有關聯合角色的相關資訊

訊，請參閱《[IAM 使用者指南](#)》中的為第三方身分提供者 (聯合) 建立角色。如果您使用 IAM Identity Center，則需要設定許可集。為控制身分驗證後可以存取的內容，IAM Identity Center 將許可集與 IAM 中的角色相關聯。如需有關許可集的資訊，請參閱 AWS IAM Identity Center 使用者指南中的[許可集](#)。

- 暫時 IAM 使用者許可 – IAM 使用者或角色可以擔任 IAM 角色來暫時針對特定任務採用不同的許可。
- 跨帳戶存取權：您可以使用 IAM 角色，允許不同帳戶中的某人 (信任的主體) 存取您帳戶的資源。角色是授予跨帳戶存取權的主要方式。不過，對於某些 AWS 服務，您可以將政策直接連接到資源 (而不是使用角色做為代理)。如需了解使用角色和資源型政策進行跨帳戶存取之間的差異，請參閱《IAM 使用者指南》中的[IAM 中的跨帳戶資源存取](#)。
- 跨服務存取 – 有些 AWS 服務 使用其他 中的功能 AWS 服務。例如，當您在服務中進行呼叫時，該服務通常會在 Amazon EC2 中執行應用程式或將物件儲存在 Amazon Simple Storage Service (Amazon S3) 中。服務可能會使用呼叫主體的許可、使用服務角色或使用服務連結角色來執行此作業。
- 轉送存取工作階段 (FAS) – 當您使用 IAM 使用者或角色在其中執行動作時 AWS，您會被視為委託人。使用某些服務時，您可能會執行某個動作，進而在不同服務中啟動另一個動作。FAS 使用呼叫的委託人許可 AWS 服務，結合 AWS 服務 請求向下游服務提出請求。只有當服務收到需要與其他 AWS 服務 或 資源互動才能完成的請求時，才會提出 FAS 請求。在此情況下，您必須具有執行這兩個動作的許可。如需提出 FAS 請求時的政策詳細資訊，請參閱《[轉發存取工作階段](#)》。
- 服務角色 – 服務角色是服務擔任的 [IAM 角色](#)，可代表您執行動作。IAM 管理員可以從 IAM 內建立、修改和刪除服務角色。如需詳細資訊，請參閱《IAM 使用者指南》中的[建立角色以委派許可權給 AWS 服務](#)。
- 服務連結角色 – 服務連結角色是一種連結至 的服務角色。AWS 服務服務可以擔任代表您執行動作的角色。服務連結角色會出現在您的 中 AWS 帳戶，並由服務擁有。IAM 管理員可以檢視，但不能編輯服務連結角色的許可。
- 在 Amazon EC2 上執行的應用程式 – 您可以使用 IAM 角色來管理在 EC2 執行個體上執行之應用程式的臨時登入資料，以及提出 AWS CLI 或 AWS API 請求。這是在 EC2 執行個體內儲存存取金鑰的較好方式。若要將 AWS 角色指派給 EC2 執行個體並將其提供給其所有應用程式，您可以建立連接至執行個體的執行個體描述檔。執行個體設定檔包含該角色，並且可讓 EC2 執行個體上執行的程式取得臨時憑證。如需詳細資訊，請參閱《IAM 使用者指南》中的[使用 IAM 角色來授予許可權給 Amazon EC2 執行個體上執行的應用程式](#)。

使用政策管理存取權

您可以透過建立政策並將其連接到身分或資源 AWS 來控制 AWS 中的存取。政策是 中的物件，當與身分或資源建立關聯時，AWS 會定義其許可。當委託人 (使用者、根使用者或角色工作階段) 發出

請求時，會 AWS 評估這些政策。政策中的許可決定是否允許或拒絕請求。大多數政策會以 JSON 文件 AWS 形式存放在 中。如需 JSON 政策文件結構和內容的詳細資訊，請參閱 IAM 使用者指南中的 [JSON 政策概觀](#)。

管理員可以使用 AWS JSON 政策來指定誰可以存取內容。也就是說，哪個主體在什麼條件下可以對什麼資源執行哪些動作。

預設情況下，使用者和角色沒有許可。若要授予使用者對其所需資源執行動作的許可，IAM 管理員可以建立 IAM 政策。然後，管理員可以將 IAM 政策新增至角色，使用者便能擔任這些角色。

IAM 政策定義該動作的許可，無論您使用何種方法來執行操作。例如，假設您有一個允許 `iam:GetRole` 動作的政策。具有該政策的使用者可以從 AWS Management Console AWS CLI、或 API AWS 取得角色資訊。

身分型政策

身分型政策是可以附加到身分 (例如 IAM 使用者、使用者群組或角色) 的 JSON 許可政策文件。這些政策可控制身分在何種條件下能對哪些資源執行哪些動作。如需了解如何建立身分型政策，請參閱《IAM 使用者指南》中的 [透過客戶管理政策定義自訂 IAM 許可](#)。

身分型政策可進一步分類成內嵌政策或受管政策。內嵌政策會直接內嵌到單一使用者、群組或角色。受管政策是獨立的政策，您可以連接到 中的多個使用者、群組和角色 AWS 帳戶。受管政策包括 AWS 受管政策和客戶受管政策。如需了解如何在受管政策及內嵌政策之間選擇，請參閱《IAM 使用者指南》中的 [在受管政策和內嵌政策間選擇](#)。

資源型政策

資源型政策是連接到資源的 JSON 政策文件。資源型政策的最常見範例是 IAM 角色信任政策和 Amazon S3 儲存貯體政策。在支援資源型政策的服務中，服務管理員可以使用它們來控制對特定資源的存取權限。對於附加政策的資源，政策會定義指定的主體可以對該資源執行的動作以及在何種條件下執行的動作。您必須在資源型政策中 [指定主體](#)。委託人可以包含帳戶、使用者、角色、聯合身分使用者或 AWS 服務。

資源型政策是位於該服務中的內嵌政策。您無法在資源型政策中使用來自 IAM 的 AWS 受管政策。

存取控制清單 (ACL)

存取控制清單 (ACL) 可控制哪些主體 (帳戶成員、使用者或角色) 擁有存取某資源的許可。ACL 類似於資源型政策，但它們不使用 JSON 政策文件格式。

Amazon S3、AWS WAF 和 Amazon VPC 是支援 ACLs 的服務範例。如需進一步了解 ACL，請參閱 Amazon Simple Storage Service 開發人員指南中的 [存取控制清單 \(ACL\) 概觀](#)。

其他政策類型

AWS 支援其他較不常見的政策類型。這些政策類型可設定較常見政策類型授予您的最大許可。

- **許可界限** – 許可範圍是一種進階功能，可供您設定身分型政策能授予 IAM 實體 (IAM 使用者或角色) 的最大許可。您可以為實體設定許可界限。所產生的許可會是實體的身分型政策和其許可界限的交集。會在 Principal 欄位中指定使用者或角色的資源型政策則不會受到許可界限限制。所有這類政策中的明確拒絕都會覆寫該允許。如需許可界限的詳細資訊，請參閱 IAM 使用者指南中的 [IAM 實體許可界限](#)。
- **服務控制政策 (SCPs)** – SCPs 是 JSON 政策，可指定中組織或組織單位 (OU) 的最大許可 AWS Organizations。AWS Organizations 是一種用於分組和集中管理您企業擁有 AWS 帳戶的多個的服務。若您啟用組織中的所有功能，您可以將服務控制政策 (SCP) 套用到任何或所有帳戶。SCP 會限制成員帳戶中實體的許可，包括每個實體 AWS 帳戶根使用者。如需 Organizations 和 SCP 的詳細資訊，請參閱《AWS Organizations 使用者指南》中的 [服務控制政策](#)。
- **資源控制政策 (RCP)** – RCP 是 JSON 政策，可用來設定您帳戶中資源的可用許可上限，採取這種方式就不需要更新附加至您所擁有的每個資源的 IAM 政策。RCP 會限制成員帳戶中資源的許可，並可能影響身分的有效許可，包括 AWS 帳戶根使用者，無論它們是否屬於您的組織。如需 Organizations 和 RCPs 的詳細資訊，包括支援 RCPs AWS 服務的清單，請參閱 AWS Organizations 《使用者指南》中的 [資源控制政策 \(RCPs\)](#)。
- **工作階段政策** – 工作階段政策是一種進階政策，您可以在透過撰寫程式的方式建立角色或聯合使用者的暫時工作階段時，做為參數傳遞。所產生工作階段的許可會是使用者或角色的身分型政策和工作階段政策的交集。許可也可以來自資源型政策。所有這類政策中的明確拒絕都會覆寫該允許。如需詳細資訊，請參閱 IAM 使用者指南中的 [工作階段政策](#)。

多種政策類型

將多種政策類型套用到請求時，其結果形成的許可會更為複雜、更加難以理解。若要了解如何 AWS 在涉及多個政策類型時決定是否允許請求，請參閱《IAM 使用者指南》中的 [政策評估邏輯](#)。

How AWS AppSync 可與 IAM 搭配使用

在您使用 IAM 管理對 AWS AppSync 的存取之前，請先了解哪些 IAM 功能可用於 AWS AppSync。

您可以搭配 AWS AppSync 使用的 IAM 功能

IAM 功能	AWS AppSync 支援
身分型政策	是
資源型政策	否
政策動作	是
政策資源	是
政策條件索引鍵	否
ACL	否
ABAC(政策中的標籤)	部分
臨時憑證	是
轉送存取工作階段 (FAS)	部分
服務角色	否
服務連結角色	部分

若要全面了解 AWS AppSync 和其他 AWS 服務如何與大多數 IAM 功能搭配使用，請參閱 [《AWS IAM 使用者指南》](#) 中的 [與 IAM 搭配使用的 服務](#)。

適用於 AWS AppSync 的身分型政策

支援身分型政策：是

身分型政策是可以附加到身分 (例如 IAM 使用者、使用者群組或角色) 的 JSON 許可政策文件。這些政策可控制身分在何種條件下能對哪些資源執行哪些動作。如需了解如何建立身分型政策，請參閱 [《IAM 使用者指南》](#) 中的 [透過客戶管理政策定義自訂 IAM 許可](#)。

使用 IAM 身分型政策，您可以指定允許或拒絕的動作和資源，以及在何種條件下允許或拒絕動作。您無法在身分型政策中指定主體，因為這會套用至連接的使用者或角色。如要了解您在 JSON 政策中使用的所有元素，請參閱 [《IAM 使用者指南》](#) 中的 [IAM JSON 政策元素參考](#)。

for AWS AppSync 的身分型政策範例

若要檢視 AWS AppSync 身分型政策的範例，請參閱 [AWS AppSync 的身分型政策](#)。

AWS AppSync 內的資源型政策

支援資源型政策：否

資源型政策是附加到資源的 JSON 政策文件。資源型政策的最常見範例是 IAM 角色信任政策和 Amazon S3 儲存貯體政策。在支援資源型政策的服務中，服務管理員可以使用它們來控制對特定資源的存取權限。對於附加政策的資源，政策會定義指定的主體可以對該資源執行的動作以及在何種條件下執行的動作。您必須在資源型政策中[指定主體](#)。委託人可以包含帳戶、使用者、角色、聯合身分使用者或 AWS 服務。

如需啟用跨帳戶存取權，您可以指定在其他帳戶內的所有帳戶或 IAM 實體，做為資源型政策的主體。新增跨帳戶主體至資源型政策，只是建立信任關係的一半。當委託人和資源位於不同位置時 AWS 帳戶，信任帳戶中的 IAM 管理員也必須授予委託人實體（使用者或角色）存取資源的許可。其透過將身分型政策連接到實體來授與許可。不過，如果資源型政策會為相同帳戶中的主體授予存取，這時就不需要額外的身分型政策。如需詳細資訊，請參閱《IAM 使用者指南》中的 [IAM 中的快帳戶資源存取](#)。

AWS AppSync 的政策動作

支援政策動作：是

管理員可以使用 AWS JSON 政策來指定誰可以存取內容。也就是說，哪個主體在什麼條件下可以對什麼資源執行哪些動作。

JSON 政策的 Action 元素描述您可以用來允許或拒絕政策中存取的動作。政策動作通常具有與相關聯 AWS API 操作相同的名稱。有一些例外狀況，例如沒有相符的 API 操作的僅限許可動作。也有一些作業需要政策中的多個動作。這些額外的動作稱為相依動作。

政策會使用動作來授予執行相關聯動作的許可。

若要查看 AWS AppSync 動作清單，請參閱服務授權參考中的 [AWS AppSync 定義的動作](#)。

in AWS AppSync 中的政策動作在動作之前使用以下字首：

```
appsync
```

若要在單一陳述式中指定多個動作，請用逗號分隔。

```
"Action": [  
    "appsync:action1",  
    "appsync:action2"  
]
```

若要檢視 AWS AppSync 身分型政策的範例，請參閱 [AWS AppSync 的身分型政策](#)。

AWS AppSync 的政策資源

支援政策資源：是

管理員可以使用 AWS JSON 政策來指定誰可以存取內容。也就是說，哪個主體在什麼條件下可以對什麼資源執行哪些動作。

Resource JSON 政策元素可指定要套用動作的物件。陳述式必須包含 Resource 或 NotResource 元素。最佳實務是使用其 [Amazon Resource Name \(ARN\)](#) 來指定資源。您可以針對支援特定資源類型的動作 (稱為資源層級許可) 來這麼做。

對於不支援資源層級許可的動作 (例如列出操作)，請使用萬用字元 (*) 來表示陳述式適用於所有資源。

```
"Resource": "*"
```

若要查看 AWS AppSync 資源類型及其 ARNs 的清單，請參閱服務授權參考中的 [AWS AppSync 定義的資源](#)。若要了解您可以使用哪些動作指定每個資源的 ARN，請參閱 [由 AWS AppSync 定義的動作](#)。

若要檢視 AWS AppSync 身分型政策的範例，請參閱 [AWS AppSync 的身分型政策](#)。

AWS AppSync 的政策條件索引鍵

支援服務特定的政策條件索引鍵：否

管理員可以使用 AWS JSON 政策來指定誰可以存取內容。也就是說，哪個主體在什麼條件下可以對什麼資源執行哪些動作。

Condition 元素 (或 Condition 區塊) 可讓您指定使陳述式生效的條件。Condition 元素是選用項目。您可以建立使用 [條件運算子](#) 的條件運算式 (例如等於或小於)，來比對政策中的條件和請求中的值。

若您在陳述式中指定多個 Condition 元素，或是在單一 Condition 元素中指定多個索引鍵，AWS 會使用邏輯 AND 操作評估他們。如果您為單一條件索引鍵指定多個值，會使用邏輯 OR 操作 AWS 評估條件。必須符合所有條件，才會授與陳述式的許可。

您也可以指定條件時使用預留位置變數。例如，您可以只在使用者使用其 IAM 使用者名稱標記時，將存取資源的許可授予該 IAM 使用者。如需更多資訊，請參閱 IAM 使用者指南中的 [IAM 政策元素：變數和標籤](#)。

AWS 支援全域條件金鑰和服務特定的條件金鑰。若要查看所有 AWS 全域條件索引鍵，請參閱《IAM 使用者指南》中的 [AWS 全域條件內容索引鍵](#)。

若要查看 AWS AppSync 條件索引鍵的清單，請參閱服務授權參考中的 [for AWS AppSync 條件索引鍵](#)。若要了解您可以使用條件金鑰的動作和資源，請參閱由 [AWS AppSync 定義的動作](#)。

若要檢視 AWS AppSync 身分型政策的範例，請參閱 [AWS AppSync 的身分型政策](#)。

AWS AppSync 中的存取控制清單 (ACLs)

支援 ACL：否

存取控制清單 (ACL) 可控制哪些主體 (帳戶成員、使用者或角色) 擁有存取某資源的許可。ACL 類似於資源型政策，但它們不使用 JSON 政策文件格式。

使用 AWS AppSync 的屬性型存取控制 (ABAC)

支援 ABAC (政策中的標籤)：部分

屬性型存取控制 (ABAC) 是一種授權策略，可根據屬性來定義許可。在 AWS 中，這些屬性稱為標籤。您可以將標籤連接至 IAM 實體 (使用者或角色) 和許多 AWS 資源。為實體和資源加上標籤是 ABAC 的第一步。您接著要設計 ABAC 政策，允許在主體的標籤與其嘗試存取的資源標籤相符時操作。

ABAC 在成長快速的環境中相當有幫助，並能在政策管理變得繁瑣時提供協助。

如需根據標籤控制存取，請使用 `aws:ResourceTag/key-name`、`aws:RequestTag/key-name` 或 `aws:TagKeys` 條件索引鍵，在政策的 [條件元素](#) 中，提供標籤資訊。

如果服務支援每個資源類型的全部三個條件金鑰，則對該服務而言，值為 Yes。如果服務僅支援某些資源類型的全部三個條件金鑰，則值為 Partial。

如需 ABAC 的詳細資訊，請參閱《IAM 使用者指南》中的 [使用 ABAC 授權定義許可](#)。如要查看含有設定 ABAC 步驟的教學課程，請參閱 IAM 使用者指南中的 [使用屬性型存取控制 \(ABAC\)](#)。

搭配 AWS AppSync 使用臨時登入資料

支援臨時憑證：是

當您使用臨時登入資料登入時，有些 AWS 服務 無法運作。如需詳細資訊，包括哪些 AWS 服務 使用臨時登入資料，請參閱 [《AWS 服務 IAM 使用者指南》](#) 中的使用 IAM 的。

如果您 AWS Management Console 使用使用者名稱和密碼以外的任何方法登入，則會使用臨時登入資料。例如，當您 AWS 使用公司的單一登入 (SSO) 連結存取時，該程序會自動建立臨時登入資料。當您以使用者身分登入主控台，然後切換角色時，也會自動建立臨時憑證。如需切換角色的詳細資訊，請參閱 [《IAM 使用者指南》](#) 中的[從使用者切換至 IAM 角色 \(主控台\)](#)。

您可以使用 AWS CLI 或 AWS API 手動建立臨時登入資料。然後，您可以使用這些臨時登入資料來存取 AWS。AWS 建議您動態產生臨時登入資料，而不是使用長期存取金鑰。如需詳細資訊，請參閱 [IAM 中的暫時性安全憑證](#)。

轉送存取工作階段 for AWS AppSync

支援轉送存取工作階段 (FAS)：部分

當您使用 IAM 使用者或角色在 中執行動作時 AWS，您會被視為委託人。使用某些服務時，您可能會執行某個動作，進而在不同服務中啟動另一個動作。FAS 使用呼叫 的委託人許可 AWS 服務，結合 AWS 服務 請求向下游服務提出請求。只有當服務收到需要與其他 AWS 服務 或 資源互動才能完成的請求時，才會提出 FAS 請求。在此情況下，您必須具有執行這兩個動作的許可。如需提出 FAS 請求時的政策詳細資訊，請參閱[轉發存取工作階段](#)。

AWS AppSync 的服務角色

支援服務角色：否

服務角色是服務擔任的 [IAM 角色](#)，可代您執行動作。IAM 管理員可以從 IAM 內建立、修改和刪除服務角色。如需詳細資訊，請參閱 [《IAM 使用者指南》](#) 中的[建立角色以委派許可權給 AWS 服務](#)。

Warning

變更服務角色的許可可能會中斷 AWS AppSync 功能。只有在 AWS AppSync 提供指引時，才能編輯服務角色。

for AWS AppSync 的服務連結角色

支援服務連結角色：部分

服務連結角色是連結至的一種服務角色 AWS 服務。服務可以擔任代表您執行動作的角色。服務連結角色會出現在您的 AWS 帳戶，並由服務擁有。IAM 管理員可以檢視，但不能編輯服務連結角色的許可。

如需建立或管理服务連結角色的詳細資訊，請參閱《[AWS IAM 使用者指南](#)》中的[使用 IAM 的服務](#)。在表格中尋找服務，其中包含服務連結角色欄中的 Yes。選擇是連結，以檢視該服務的服務連結角色文件。

AWS AppSync 的身分型政策

根據預設，使用者和角色沒有建立或修改 AWS AppSync 資源的許可。他們也無法使用 AWS Management Console、AWS Command Line Interface (AWS CLI) 或 AWS API 來執行任務。若要授予使用者對其所需資源執行動作的許可，IAM 管理員可以建立 IAM 政策。然後，管理員可以將 IAM 政策新增至角色，使用者便能擔任這些角色。

如需了解如何使用這些範例 JSON 政策文件建立 IAM 身分型政策，請參閱《IAM 使用者指南》中的[建立 IAM 政策 \(主控台\)](#)。

如需由 AWS AppSync 定義的動作和資源類型的詳細資訊，包括每種資源類型的 ARNs 格式，請參閱服務授權參考中的[適用於 AWS AppSync 的動作、資源和條件索引鍵](#)。

若要了解建立和設定 IAM 身分型政策的最佳實務，請參閱 [the section called “IAM 政策最佳實務”](#)。

如需適用於 AWS AppSync 的 IAM 身分型政策清單，請參閱 [AWS AppSync 的受管政策](#)。

主題

- [使用 AWS AppSync 主控台](#)
- [允許使用者檢視他們自己的許可](#)
- [存取一個 Amazon S3 儲存貯體](#)
- [根據標籤檢視 AWS AppSync 小工具](#)
- [AWS AppSync 的受管政策](#)

使用 AWS AppSync 主控台

若要存取 AWS AppSync 主控台，您必須擁有一組最低許可。這些許可必須允許您列出和檢視中有關 AWS AppSync 資源的詳細資訊 AWS 帳戶。如果您建立比最基本必要許可更嚴格的身分型政策，則對於具有該政策的實體 (使用者或角色) 而言，主控台就無法如預期運作。

對於僅呼叫 AWS CLI 或 AWS API 的使用者，您不需要允許最低主控台許可。反之，只需允許存取符合他們嘗試執行之 API 操作的動作就可以了。

為了確保 IAM 使用者和角色仍然可以使用 AWS AppSync 主控台，請將 AWS AppSync ConsoleAccess 或 ReadOnly AWS 受管政策連接到實體。如需詳細資訊，請參閱《IAM 使用者指南》中的[新增許可到使用者](#)。

允許使用者檢視他們自己的許可

此範例會示範如何建立政策，允許 IAM 使用者檢視附加到他們使用者身分的內嵌及受管政策。此政策包含在主控台或使用或 AWS CLI AWS API 以程式設計方式完成此動作的許可。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "ViewOwnUserInfo",
      "Effect": "Allow",
      "Action": [
        "iam:GetUserPolicy",
        "iam:ListGroupsForUser",
        "iam:ListAttachedUserPolicies",
        "iam:ListUserPolicies",
        "iam:GetUser"
      ],
      "Resource": ["arn:aws:iam::*:user/${aws:username}"]
    },
    {
      "Sid": "NavigateInConsole",
      "Effect": "Allow",
      "Action": [
        "iam:GetGroupPolicy",
        "iam:GetPolicyVersion",
        "iam:GetPolicy",
        "iam:ListAttachedGroupPolicies",
        "iam:ListGroupPolicies",
        "iam:ListPolicyVersions",
        "iam:ListPolicies",
        "iam:ListUsers"
      ],
      "Resource": "*"
    }
  ]
}
```

```
}
```

存取一個 Amazon S3 儲存貯體

在此範例中，您想要授予 AWS 帳戶中的 IAM 使用者存取其中一個 Amazon S3 儲存貯體 `examplebucket`。您也希望允許使用者新增、更新和刪除物件。

除了授予使用者 `s3:PutObject`、`s3:GetObject` 與 `s3:DeleteObject` 許可之外，政策也會授予 `s3:ListAllMyBuckets`、`s3:GetBucketLocation` 與 `s3:ListBucket` 許可。這些是主控台需要的額外許可。還需要 `s3:PutObjectAcl` 與 `s3:GetObjectAcl` 動作才能在主控台中複製、剪下與貼上物件。如需將許可授予使用者並使用主控台進行測試的範例逐步解說，請參閱[範例逐步解說：使用使用者政策來控制對儲存貯體的存取](#)。

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "ListBucketsInConsole",
      "Effect": "Allow",
      "Action": [
        "s3:ListAllMyBuckets"
      ],
      "Resource": "arn:aws:s3:::*"
    },
    {
      "Sid": "ViewSpecificBucketInfo",
      "Effect": "Allow",
      "Action": [
        "s3:ListBucket",
        "s3:GetBucketLocation"
      ],
      "Resource": "arn:aws:s3:::examplebucket"
    },
    {
      "Sid": "ManageBucketContents",
      "Effect": "Allow",
      "Action": [
        "s3:PutObject",
        "s3:PutObjectAcl",
```

```

        "s3:GetObject",
        "s3:GetObjectAcl",
        "s3:DeleteObject"
    ],
    "Resource": "arn:aws:s3:::examplebucket/*"
}
]
}

```

根據標籤檢視 AWS AppSync

您可以在身分型政策中使用條件，根據標籤控制對 AWS AppSync 資源的存取。此範例示範如何建立允許檢視###的政策。不過，只有在###標籤Owner的值為該使用者的使用者名稱時，才會授予許可。此政策也會授予在主控台完成此動作的必要許可。

JSON

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "ListWidgetsInConsole",
      "Effect": "Allow",
      "Action": ":ListWidgets",
      "Resource": "*"
    },
    {
      "Sid": "ViewWidgetIfOwner",
      "Effect": "Allow",
      "Action": ":GetWidget",
      "Resource": "arn:aws::*:*:widget/*",
      "Condition": {
        "StringEquals": {"aws:ResourceTag/Owner": "${aws:username}"}
      }
    }
  ]
}

```

您可以將此政策連接到您帳戶中的 IAM 使用者。如果名為的使用者richard-roe嘗試檢視 AWS AppSync ###，則該###必須加上標籤 Owner=richard-roe或 owner=richard-roe。否則他便會

被拒絕存取。條件標籤鍵 `Owner` 符合 `Owner` 和 `owner`，因為條件索引鍵名稱不區分大小寫。如需詳細資訊，請參閱 IAM 使用者指南中的 [IAM JSON 政策元素：條件](#)。

AWS AppSync 的 受管政策

若要新增許可給使用者、群組和角色，使用 AWS 受管政策比自行撰寫政策更容易。[建立 IAM 客戶受管政策](#) 需要時間和專業知識，而受管政策可為您的團隊提供其所需的許可。若要快速開始使用，您可以使用我們的 AWS 受管政策。這些政策涵蓋常見的使用案例，並可在您的 AWS 帳戶中使用。如需 AWS 受管政策的詳細資訊，請參閱《IAM 使用者指南》中的 [AWS 受管政策](#)。

AWS 服務會維護和更新 AWS 受管政策。您無法變更 AWS 受管政策中的許可。服務偶爾會將其他許可新增至 AWS 受管政策，以支援新功能。此類型的更新會影響已連接政策的所有身分識別 (使用者、群組和角色)。當新功能啟動或新操作可用時，服務最有可能更新 AWS 受管政策。服務不會從 AWS 受管政策中移除許可，因此政策更新不會破壞您現有的許可。

此外，AWS 支援跨多個 服務之任務函數的受管政策。例如，`ReadOnlyAccess` AWS 受管政策提供所有 AWS 服務和資源的唯讀存取權。當服務啟動新功能時，會為新操作和資源 AWS 新增唯讀許可。如需任務職能政策的清單和說明，請參閱 IAM 使用者指南中 [有關任務職能的 AWS 受管政策](#)。

AWS 受管政策：AWSAppSyncInvokeFullAccess

使用 `AWSAppSyncInvokeFullAccess` AWS 受管政策，讓您的管理員透過主控台或獨立存取 AWS AppSync 服務。

您可將 `AWSAppSyncInvokeFullAccess` 政策連接到 IAM 身分。

許可詳細資訊

此政策包含以下許可。

- `AWS AppSync` – 允許對 in `AWS AppSync` 中的所有資源進行完整管理存取

JSON

```
{
```

```
"Version": "2012-10-17",
"Statement": [
  {
    "Effect": "Allow",
    "Action": [
      "appsync:GraphQL",
      "appsync:GetGraphQLApi",
      "appsync:ListGraphQLApis",
      "appsync:ListApiKeys"
    ],
    "Resource": "*"
  }
]
```

AWS 受管政策：AWSAppSyncSchemaAuthor

使用 AWSAppSyncSchemaAuthor AWS 受管政策允許 IAM 使用者存取，以建立、更新和查詢其 GraphQL 結構描述。如需使用者可以如何處理這些許可的詳細資訊，請參閱 [使用 設計 GraphQL APIs AWS AppSync](#)。

您可將 AWSAppSyncSchemaAuthor 政策連接到 IAM 身分。

許可詳細資訊

此政策包含以下許可。

- AWS AppSync – 允許下列動作：
 - 建立 GraphQL 結構描述
 - 允許建立、修改和刪除 GraphQL 類型、解析程式和函數
 - 評估請求和回應範本邏輯
 - 使用執行時間和內容評估程式碼
 - 將 GraphQL 查詢傳送至 GraphQL APIs
 - 擷取 GraphQL 資料

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "appsync:GraphQL",
        "appsync:CreateResolver",
        "appsync:CreateType",
        "appsync>DeleteResolver",
        "appsync>DeleteType",
        "appsync:GetResolver",
        "appsync:GetType",
        "appsync:GetDataSource",
        "appsync:GetSchemaCreationStatus",
        "appsync:GetIntrospectionSchema",
        "appsync:GetGraphqlApi",
        "appsync:ListTypes",
        "appsync:ListApiKeys",
        "appsync:ListResolvers",
        "appsync:ListDataSources",
        "appsync:ListGraphqlApis",
        "appsync:StartSchemaCreation",
        "appsync:UpdateResolver",
        "appsync:UpdateType",
        "appsync:TagResource",
        "appsync:UntagResource",
        "appsync:ListTagsForResource",
        "appsync:CreateFunction",
        "appsync:UpdateFunction",
        "appsync:GetFunction",
        "appsync>DeleteFunction",
        "appsync:ListFunctions",
        "appsync:ListResolversByFunction",
        "appsync:EvaluateMappingTemplate",
        "appsync:EvaluateCode"
      ],
      "Resource": "*"
    }
  ]
}
```

```
}
```

AWS 受管政策：AWSAppSyncPushToCloudWatchLogs

AWS AppSync 使用 Amazon CloudWatch 來監控應用程式的效能，方法是產生可用於疑難排解和最佳化 GraphQL 請求的日誌。如需詳細資訊，請參閱[使用 CloudWatch 監控和記錄 GraphQL API 資料](#)。

使用 AWSAppSyncPushToCloudWatchLogs AWS 受管政策允許 AWS AppSync 將日誌推送至 IAM 使用者的 CloudWatch 帳戶。

您可將 AWSAppSyncPushToCloudWatchLogs 政策連接到 IAM 身分。

許可詳細資訊

此政策包含以下許可。

- CloudWatch Logs – 允許 AWS AppSync 使用指定的名稱建立日誌群組和串流。AWS AppSync 會將日誌事件推送至指定的日誌串流。

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "logs:CreateLogGroup",
        "logs:CreateLogStream",
        "logs:PutLogEvents"
      ],
      "Resource": "*"
    }
  ]
}
```

AWS 受管政策：AWSAppSyncAdministrator

使用 AWSAppSyncAdministrator AWS 受管政策，讓您的管理員可以存取 AWS 主控台以外的所有 all AWS AppSync。

您可以將 AWSAppSyncAdministrator 連接到 IAM 實體。AWS AppSync 也會將此政策連接至服務角色，以允許其代表您執行動作。

許可詳細資訊

此政策包含以下許可。

- AWS AppSync – 允許 AWS AppSync 中所有資源的完整管理存取權
- IAM – 允許下列動作：
 - 建立服務連結角色以允許 AWS AppSync 代表您分析其他服務中的資源
 - 刪除服務連結角色
 - 將服務連結角色傳遞至其他 AWS 服務，以在稍後擔任該角色並代表您執行動作

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "appsync:*"
      ],
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "iam:PassRole"
      ],
      "Resource": "*",
```



```

        "Condition": {
            "StringEquals": {
                "iam:PassedToService": [
                    "appsync.amazonaws.com"
                ]
            }
        },
    ],
    {
        "Effect": "Allow",
        "Action": "iam:CreateServiceLinkedRole",
        "Resource": "*",
        "Condition": {
            "StringEquals": {
                "iam:AWSServiceName": "appsync.amazonaws.com"
            }
        }
    },
    {
        "Effect": "Allow",
        "Action": [
            "iam:DeleteServiceLinkedRole",
            "iam:GetServiceLinkedRoleDeletionStatus"
        ],
        "Resource": "arn:aws:iam::*:role/aws-service-role/
appsync.amazonaws.com/AWSServiceRoleForAppSync*"
    }
]
}

```

AWS 受管政策：AWSAppSyncServiceRolePolicy

使用 AWSAppSyncServiceRolePolicy AWS 受管政策允許存取 AWS AppSync 使用或管理 AWS 的服務和資源。

您不得將 AWSAppSyncServiceRolePolicy 連接到 IAM 實體。此政策會連接到服務連結角色，允許 AWS AppSync 代表您執行動作。如需詳細資訊，請參閱[for AWS AppSync 的服務連結角色](#)。

許可詳細資訊

此政策包含以下許可。

- X-Ray – AWS AppSync 使用 AWS X-Ray 收集應用程式內所提出請求的資料。如需詳細資訊，請參閱[使用 AWS X-Ray 追蹤 in AWS AppSync 中的請求](#)。

此政策允許下列動作：

- 擷取抽樣規則及其結果
- 將追蹤資料傳送至 X-Ray 協助程式

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "xray:PutTraceSegments",
        "xray:PutTelemetryRecords",
        "xray:GetSamplingTargets",
        "xray:GetSamplingRules",
        "xray:GetSamplingStatisticSummaries"
      ],
      "Resource": [
        "*"
      ]
    }
  ]
}
```

受管政策的 AWS AppSync 更新

檢視自此服務開始追蹤這些變更以來，AWS AppSync AWS 受管政策更新的詳細資訊。如需此頁面變更的自動提醒，請訂閱 AWS AppSync 文件歷史記錄頁面上的 RSS 摘要。

變更	描述	日期
AWSAppSyncSchemaAuthor - 更新現有政策	新增EvaluateCode 政策動作，允許使用者評估具有執行時間和內容的程式碼。	2023 年 2 月 7 日
AWSAppSyncSchemaAuthor - 更新現有政策	新增政策動作，以允許 API 的清單、取得、建立、更新和刪除函數。 新增EvaluateMappingTemplate 政策動作，允許使用者評估請求和回應解析程式映射範本邏輯。 新增政策動作以允許資源標記。	2022 年 8 月 25 日
AWS AppSync 已開始追蹤變更	AWS AppSync 開始追蹤其 AWS 受管政策的變更。	2022 年 8 月 25 日

故障診斷 AWS AppSync 身分和存取

使用以下資訊來協助您診斷和修正使用 AWS AppSync 和 IAM 時可能遇到的常見問題。

我無權在 AWS AppSync 中執行動作

如果 AWS Management Console 通知您無權執行動作，則必須聯絡您的管理員尋求協助。您的管理員是提供您使用者名稱和密碼的人員。

當 IAM mateojackson 使用者嘗試使用主控台檢視虛構`my-example-widget`資源的詳細資訊，但他沒有虛構`appsync:GetWidget`許可時，會發生下列範例錯誤。

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to perform:
appsync:GetWidget on resource: my-example-widget
```

在此情況下，Mateo 會請求管理員更新他的政策，允許他使用 `my-example-widget` 動作存取 `appsync:GetWidget` 資源。

我未獲得執行 iam:PassRole 的授權

如果您收到錯誤，告知您無權執行 iam:PassRole 動作，您的政策必須更新，以允許您將角色傳遞至 AWS AppSync。

有些 AWS 服務可讓您將現有角色傳遞給該服務，而不是建立新的服務角色或服務連結角色。如需執行此作業，您必須擁有將角色傳遞至該服務的許可。

當名為的 IAM marymajor 使用者嘗試使用主控台在 AWS AppSync 中執行動作時，會發生下列範例錯誤。但是，動作請求服務具備服務角色授予的許可。Mary 沒有將角色傳遞至該服務的許可。

```
User: arn:aws:iam::123456789012:user/marymajor is not authorized to perform:
iam:PassRole
```

在這種情況下，Mary 的政策必須更新，允許她執行 iam:PassRole 動作。

如果您需要協助，請聯絡您的 AWS 管理員。您的管理員提供您的登入憑證。

我想要檢視我的存取金鑰

在您建立 IAM 使用者存取金鑰後，您可以隨時檢視您的存取金鑰 ID。但是，您無法再次檢視您的私密存取金鑰。若您遺失了密碼金鑰，您必須建立新的存取金鑰對。

存取金鑰包含兩個部分：存取金鑰 ID (例如 AKIAIOSFODNN7EXAMPLE) 和私密存取金鑰 (例如 wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY)。如同使用者名稱和密碼，您必須一起使用存取金鑰 ID 和私密存取金鑰來驗證您的請求。就如對您的使用者名稱和密碼一樣，安全地管理您的存取金鑰。

Important

請勿將您的存取金鑰提供給第三方，甚至是協助[尋找您的標準使用者 ID](#)。透過這樣做，您可以讓某人永久存取您的 AWS 帳戶。

建立存取金鑰對時，您會收到提示，要求您將存取金鑰 ID 和私密存取金鑰儲存在安全位置。私密存取金鑰只會在您建立它的時候顯示一次。若您遺失了私密存取金鑰，您必須將新的存取金鑰新增到您的 IAM 使用者。您最多可以擁有兩個存取金鑰。若您已有兩個存取金鑰，您必須先刪除其中一個金鑰對，才能建立新的金鑰對。若要檢視說明，請參閱《IAM 使用者指南》中的[管理存取金鑰](#)。

我是管理員，想要允許其他人存取 AWS AppSync

若要允許其他人存取 AWS AppSync，您必須將許可授予需要存取的人員或應用程式。如果您使用 AWS IAM Identity Center 來管理人員和應用程式，您可以將許可集指派給使用者或群組，以定義其存取層級。許可集會自動建立 IAM 政策，並將其指派給與該人員或應用程式相關聯的 IAM 角色。如需詳細資訊，請參閱 AWS IAM Identity Center 《使用者指南》中的[許可集](#)。

如果您不是使用 IAM Identity Center，則必須為需要存取的人員或應用程式建立 IAM 實體（使用者或角色）。然後，您必須將政策連接至實體，以授予其正確的許可 in AWS AppSync。授予許可後，請將登入資料提供給使用者或應用程式開發人員。他們將使用這些登入資料來存取 AWS。若要進一步了解如何建立 IAM 使用者、群組、政策和許可，請參閱《IAM [使用者指南](#)》中的 [IAM 身分](#)和[政策和許可](#)。

我想要允許帳戶 AWS 外的人員存取 my AWS AppSync 資源

您可以建立一個角色，讓其他帳戶中的使用者或您組織外部的人員存取您的資源。您可以指定要允許哪些信任物件取得該角色。針對支援基於資源的政策或存取控制清單 (ACL) 的服務，您可以使用那些政策來授予人員存取您的資源的許可。

如需進一步了解，請參閱以下內容：

- 若要了解 AWS AppSync 是否支援這些功能，請參閱 [How AWS AppSync 可與 IAM 搭配使用](#)。
- 若要了解如何提供您擁有 AWS 帳戶 的資源存取權，請參閱《[IAM 使用者指南](#)》中的[在您擁有 AWS 帳戶 的另一個 中為 IAM 使用者提供存取權](#)。
- 若要了解如何將資源的存取權提供給第三方 AWS 帳戶，請參閱《IAM 使用者指南》中的[將存取權提供給第三方 AWS 帳戶 擁有](#)。
- 如需了解如何透過聯合身分提供存取權，請參閱 IAM 使用者指南中的[將存取權提供給在外部進行身分驗證的使用者 \(聯合身分\)](#)。
- 如需了解使用角色和資源型政策進行跨帳戶存取之間的差異，請參閱《IAM 使用者指南》中的 [IAM 中的跨帳戶資源存取](#)。

使用 Logging AWS AppSync API 呼叫 AWS CloudTrail

AWS AppSync 已與 服務整合 AWS CloudTrail，此服務提供由使用者、角色或服務 AWS in AWS AppSync 所採取動作的記錄。CloudTrail 會將 AWS AppSync 的 API 呼叫擷取為事件。擷取的呼叫包括來自 AWS AppSync 主控台的呼叫，以及對 AWS AppSync API 操作的程式碼呼叫。如果您建立線索，您可以將 CloudTrail 事件持續交付至 Amazon S3 儲存貯體，包括 for AWS AppSync 的事件。即使您未設定追蹤，依然可以透過 CloudTrail 主控台的事件歷史記錄檢視最新事件。您可以使用

CloudTrail 所收集的資訊，判斷對 AWS AppSync 提出的請求、提出請求的 IP 地址、提出請求的人員、提出請求的時間，以及其他詳細資訊。

若要進一步了解 CloudTrail，請參閱 [「AWS CloudTrail 使用者指南」](#)。

CloudTrail 中的 AWS AppSync 資訊

當您建立 AWS 帳戶時，會在您的帳戶上啟用 CloudTrail。當活動在 AWS AppSync 中發生時，該活動會與事件歷史記錄中的其他服務 AWS 事件一起記錄在 CloudTrail 事件中。您可以在 AWS 帳戶中檢視、搜尋和下載最近的事件。如需詳細資訊，請參閱 [「使用 CloudTrail 事件歷史記錄檢視事件」](#)。

若要持續記錄您 AWS 帳戶中的事件，包括 for AWS AppSync 的事件，請建立追蹤。線索能讓 CloudTrail 將日誌檔案交付至 Amazon S3 儲存貯體。根據預設，當您在主控台中建立線索時，線索會套用至所有 AWS 區域。線索會記錄 AWS 分割區中所有區域的事件，並將日誌檔案傳送到您指定的 Amazon S3 儲存貯體。此外，您可以設定其他 AWS 服務，以進一步分析和處理 CloudTrail 日誌中所收集的事件資料。如需詳細資訊，請參閱下列內容：

- [建立追蹤的概觀](#)
- [CloudTrail 支援的服務和整合](#)
- [設定 CloudTrail 的 Amazon SNS 通知](#)
- [接收多個區域的 CloudTrail 日誌檔案](#)和[接收多個帳戶的 CloudTrail 日誌檔案](#)

AWS AppSync 支援記錄透過 AWS AppSync API 進行的呼叫。目前，AWS AppSync 不會將對 APIs 呼叫以及對解析程式發出的呼叫記錄到 CloudTrail。

每一筆事件或日誌專案都會包含產生請求者的資訊。身分資訊可協助您判斷下列事項：

- 是否使用根或 AWS Identity and Access Management (IAM) 使用者登入資料提出請求。
- 提出該請求時，是否使用了特定角色或聯合身分使用者的暫時安全憑證。
- 請求是否由其他 AWS 服務提出。

如需詳細資訊，請參閱 [CloudTrail userIdentity 元素](#)。

Understanding AWS AppSync 日誌檔案項目

追蹤是一種組態，能讓事件以日誌檔案的形式交付到您指定的 Amazon S3 儲存貯體。CloudTrail 日誌檔案包含一或多個日誌專案。一個事件為任何來源提出的單一請求，並包含請求動作、請求的日期和時

間、請求參數等資訊。CloudTrail 日誌檔並非依公有 API 呼叫的堆疊追蹤排序，因此不會以任何特定順序出現。

下列範例顯示 CloudTrail 日誌項目，示範透過 AWS AppSync 主控台執行 `GetGraphqlApi` 的動作：

```
{
  "eventVersion": "1.08",
  "userIdentity": {
    "type": "AssumedRole",
    "principalId": "ABCDEFXAMPLEPRINCIPAL:nikkiwolf",
    "arn": "arn:aws:sts::111122223333:assumed-role/admin/nikkiwolf",
    "accountId": "111122223333",
    "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
    "sessionContext": {
      "sessionIssuer": {
        "type": "Role",
        "principalId": "AIDAJ45Q7YFFAREXAMPLE",
        "arn": "arn:aws:iam::111122223333:role/admin",
        "accountId": "111122223333",
        "userName": "admin"
      },
      "webIdFederationData": {},
      "attributes": {
        "mfaAuthenticated": "false",
        "creationDate": "2021-03-12T22:41:48Z"
      }
    }
  },
  "eventTime": "2021-03-12T22:46:18Z",
  "eventSource": "appsync.amazonaws.com",
  "eventName": "GetGraphqlApi",
  "awsRegion": "us-west-2",
  "sourceIPAddress": "203.0.113.69",
  "userAgent": "aws-internal/3 aws-sdk-java/1.11.942
Linux/4.9.230-0.1.ac.223.84.332.metal1.x86_64 OpenJDK_64-Bit_Server_VM/25.282-b08
java/1.8.0_282 vendor/Oracle_Corporation",
  "requestParameters": {
    "apiId": "xhxt3typtfnmidkhcexampleid"
  },
  "responseElements": null,
  "requestID": "2fc43a35-a552-4b5d-be6e-12553a03dd12",
  "eventID": "b95b0ad9-8c71-4252-a2ec-5dc2fe5f8ae8",
  "readOnly": true,
  "eventType": "AwsApiCall",
}
```

```
"managementEvent": true,
"eventCategory": "Management",
"recipientAccountId": "111122223333"
}
```

下列範例顯示 CloudTrail 日誌項目，示範透過 執行CreateApikey的動作 AWS CLI：

```
{
  "eventVersion": "1.08",
  "userIdentity": {
    "type": "IAMUser",
    "principalId": "ABCDEFXAMPLEPRINCIPAL",
    "arn": "arn:aws:iam::111122223333:user/nikkiwolf",
    "accountId": "111122223333",
    "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
    "userName": "nikkiwolf"
  },
  "eventTime": "2021-03-12T22:49:10Z",
  "eventSource": "appsync.amazonaws.com",
  "eventName": "CreateApiKey",
  "awsRegion": "us-west-2",
  "sourceIPAddress": "203.0.113.69",
  "userAgent": "aws-cli/2.0.11 Python/3.7.4 Darwin/18.7.0 botocore/2.0.0dev15",
  "requestParameters": {
    "apiId": "xhxt3typtfnmidkhcexampleid"
  },
  "responseElements": {
    "apiKey": {
      "id": "****",
      "expires": 1616191200,
      "deletes": 1621375200
    }
  },
  "requestID": "e152190e-04ba-4d0a-ae7b-6bfc0bcea6af",
  "eventID": "ba3f39e0-9d87-41c5-abbb-2000abcb6013",
  "readOnly": false,
  "eventType": "AwsApiCall",
  "managementEvent": true,
  "eventCategory": "Management",
  "recipientAccountId": "111122223333"
}
```


的安全最佳實務 AWS AppSync

安全 AWS AppSync 不只是開啟幾個控制桿或設定記錄。以下各節討論的安全最佳實務，取決於您使用服務的方式。

了解身分驗證方法

AWS AppSync 提供多種方式來向 GraphQL APIs 您的使用者。每個方法在安全性、可稽核性和可用性方面都有權衡。

可用的常見身分驗證方法如下：

- Amazon Cognito 使用者集區可讓您的 GraphQL API 使用使用者屬性進行精細存取控制和篩選。
- API 字符的生命週期有限，適用於自動化系統，例如持續整合系統和與外部 APIs 整合。
- AWS Identity and Access Management (IAM) 適用於 中管理的內部應用程式 AWS 帳戶。
- OpenID Connect 可讓您使用 OpenID Connect 通訊協定控制和聯合存取。

如需 中身分驗證和授權的詳細資訊 AWS AppSync，請參閱 [設定授權和身分驗證來保護 GraphQL APIs](#)。

了解 API 組態如何變更傳播

當您儲存 API 組態的變更時，會 AWS AppSync 開始傳播變更。在您的組態變更傳播之前，會 AWS AppSync 繼續提供先前組態的內容。傳播組態變更之後，會 AWS AppSync 立即開始根據新組態提供內容。當 AWS AppSync 正在傳播 API 的變更時，我們無法根據先前的組態或新的組態來判斷 API 是否為您的內容提供服務。

針對 HTTP 解析程式使用 TLS

使用 HTTP 解析程式時，請務必盡可能使用 TLS 安全 (HTTPS) 連線。如需 AWS AppSync 信任的 TLS 憑證的完整清單，請參閱 [HTTPS AWS AppSync 端點的憑證授權機構 \(CA\) 由認可](#)。

盡可能使用具有最低許可的角色

使用 [DynamoDB 解析程式等解析程式](#)時，請使用為您的資源提供最嚴格檢視的角色，例如 Amazon DynamoDB 資料表。

IAM 政策最佳實務

身分型政策會判斷您帳戶中的某個人員是否可以建立、存取或刪除 AWS AppSync 資源。這些動作可能會讓您的 AWS 帳戶產生費用。當您建立或編輯身分型政策時，請遵循下列準則及建議事項：

- 開始使用 AWS 受管政策並轉向最低權限許可 – 若要開始將許可授予您的使用者和工作負載，請使用將許可授予許多常見使用案例的 AWS 受管政策。它們可在您的 中使用 AWS 帳戶。我們建議您定義特定於使用案例 AWS 的客戶受管政策，以進一步減少許可。如需更多資訊，請參閱 IAM 使用者指南中的 [AWS 受管政策](#) 或 [任務職能的 AWS 受管政策](#)。
- 套用最低權限許可 – 設定 IAM 政策的許可時，請僅授予執行任務所需的許可。為實現此目的，您可以定義在特定條件下可以對特定資源採取的動作，這也稱為最低權限許可。如需使用 IAM 套用許可的更多相關資訊，請參閱 IAM 使用者指南中的 [IAM 中的政策和許可](#)。
- 使用 IAM 政策中的條件進一步限制存取權 – 您可以將條件新增至政策，以限制動作和資源的存取。例如，您可以撰寫政策條件，指定必須使用 SSL 傳送所有請求。如果透過特定 例如 使用服務動作 AWS 服務，您也可以使用條件來授予其存取權 AWS CloudFormation。如需詳細資訊，請參閱 IAM 使用者指南中的 [IAM JSON 政策元素：條件](#)。
- 使用 IAM Access Analyzer 驗證 IAM 政策，確保許可安全且可正常運作 – IAM Access Analyzer 驗證新政策和現有政策，確保這些政策遵從 IAM 政策語言 (JSON) 和 IAM 最佳實務。IAM Access Analyzer 提供 100 多項政策檢查及切實可行的建議，可協助您撰寫安全且實用的政策。如需詳細資訊，請參閱《IAM 使用者指南》中的 [使用 IAM Access Analyzer 驗證政策](#)。
- 需要多重要素驗證 (MFA) – 如果您的案例需要 IAM 使用者或 中的根使用者 AWS 帳戶，請開啟 MFA 以提高安全性。如需在呼叫 API 操作時請求 MFA，請將 MFA 條件新增至您的政策。如需詳細資訊，請參閱《IAM 使用者指南》https://docs.aws.amazon.com/IAM/latest/UserGuide/id_credentials_mfa_configure-api-require.html 中的透過 MFA 的安全 API 存取。

如需 IAM 中最佳實務的相關資訊，請參閱 IAM 使用者指南中的 [IAM 安全最佳實務](#)。

AWS AppSync 解析程式參考 (JavaScript)

下列各節包含APPSYNC_JS執行期和 JavaScript 解析程式參考：

- [JavaScript 解析程式概觀](#) - 進一步了解解析程式的運作方式 AWS AppSync。
- [解析程式內容物件參考](#) - 進一步了解內容物件及其在解析程式中的使用方式。
- [解析程式和函數的 JavaScript 執行期功能](#) - 進一步了解支援的執行期功能，以及使用公用程式簡化程式碼。
- [DynamoDB 的 JavaScript 解析程式函數參考](#) - 進一步了解解析程式如何與 DynamoDB 互動。
- [OpenSearch 的 JavaScript 解析程式函數參考](#) - 進一步了解解析程式請求和回應結構，以及與 OpenSearch Service 的互動。
- [Lambda 的 JavaScript 解析程式函數參考](#) - 進一步了解解析程式請求和回應結構，以及與 Lambda 的互動。
- [EventBridge 資料來源的 JavaScript 解析程式函數參考](#) - 進一步了解解析程式請求和回應結構，以及與 EventBridge 的互動。
- [無資料來源的 JavaScript 解析程式函數參考](#) - 進一步了解解析程式請求和回應結構，以及與 NONE 資料來源的互動。
- [HTTP 的 JavaScript 解析程式函數參考](#) - 進一步了解解析程式請求和回應結構，以及與 HTTP 端點的互動。
- [Amazon RDS 的 JavaScript 解析程式函數參考](#) - 進一步了解解析程式結構和與 RDS 的互動。
- [Amazon Bedrock 的 JavaScript 解析程式函數參考](#) - 進一步了解解析程式結構和與 Amazon Bedrock 的互動。

AWS AppSync JavaScript 解析程式概觀

AWS AppSync 可讓您透過對資料來源執行操作來回應 GraphQL 請求。對於您想要執行查詢、變動或訂閱的每個 GraphQL 欄位，必須連接解析程式。

解析程式是 GraphQL 與資料來源之間的連接器。他們 tell AWS AppSync 如何將傳入的 GraphQL 請求轉譯為後端資料來源的指示，以及如何將來自該資料來源的回應轉譯回 GraphQL 回應。透過 AWS AppSync，您可以使用 JavaScript 撰寫解析程式，並在 AWS AppSync (APPSYNC_JS) 環境中執行。

AWS AppSync 可讓您撰寫由管道中多個 AWS AppSync 函數組成的單元解析程式或管道解析程式。

支援的執行時間功能

The AWS AppSync JavaScript 執行時間提供 JavaScript 程式庫、公用程式和功能的子集。如需 APPSYNC_JS 執行時間支援的完整功能清單，請參閱[解析程式和函數的 JavaScript 執行時間功能](#)。

單位解析程式

單位解析程式由程式碼組成，可定義針對資料來源執行的請求和回應處理常式。請求處理常式會將內容物件做為引數，並傳回用來呼叫資料來源的請求承載。回應處理常式會從資料來源接收承載，其中包含執行請求的結果。回應處理常式會將承載轉換為 GraphQL 回應，以解析 GraphQL 欄位。在下列範例中，解析程式會從 DynamoDB 資料來源擷取項目：

```
import * as ddb from '@aws-appsync/utils/dynamodb'

export function request(ctx) {
  return ddb.get({ key: { id: ctx.args.id } });
}

export const response = (ctx) => ctx.result;
```

JavaScript 管道解析程式的剖析

管道解析程式由程式碼組成，可定義請求和回應處理常式，以及函數清單。每個函數都有一個針對資料來源執行的請求和回應處理常式。由於管道解析程式委派會執行到函數清單，因此不會連結到任何資料來源。單位解析程式和函數是對資料來源執行操作的基本元素。

管道解析程式請求處理常式

管道解析程式的請求處理常式（步驟前）可讓您在執行定義的函數之前執行一些準備邏輯。

函數清單

管道解析程式將會依序執行的函數清單。管道解析程式請求處理常式評估結果會以 `ctx.prev.result` 的形式提供給第一個函數。每個函數評估結果都以 `ctx.prev.result` 的形式提供給下一個函數。

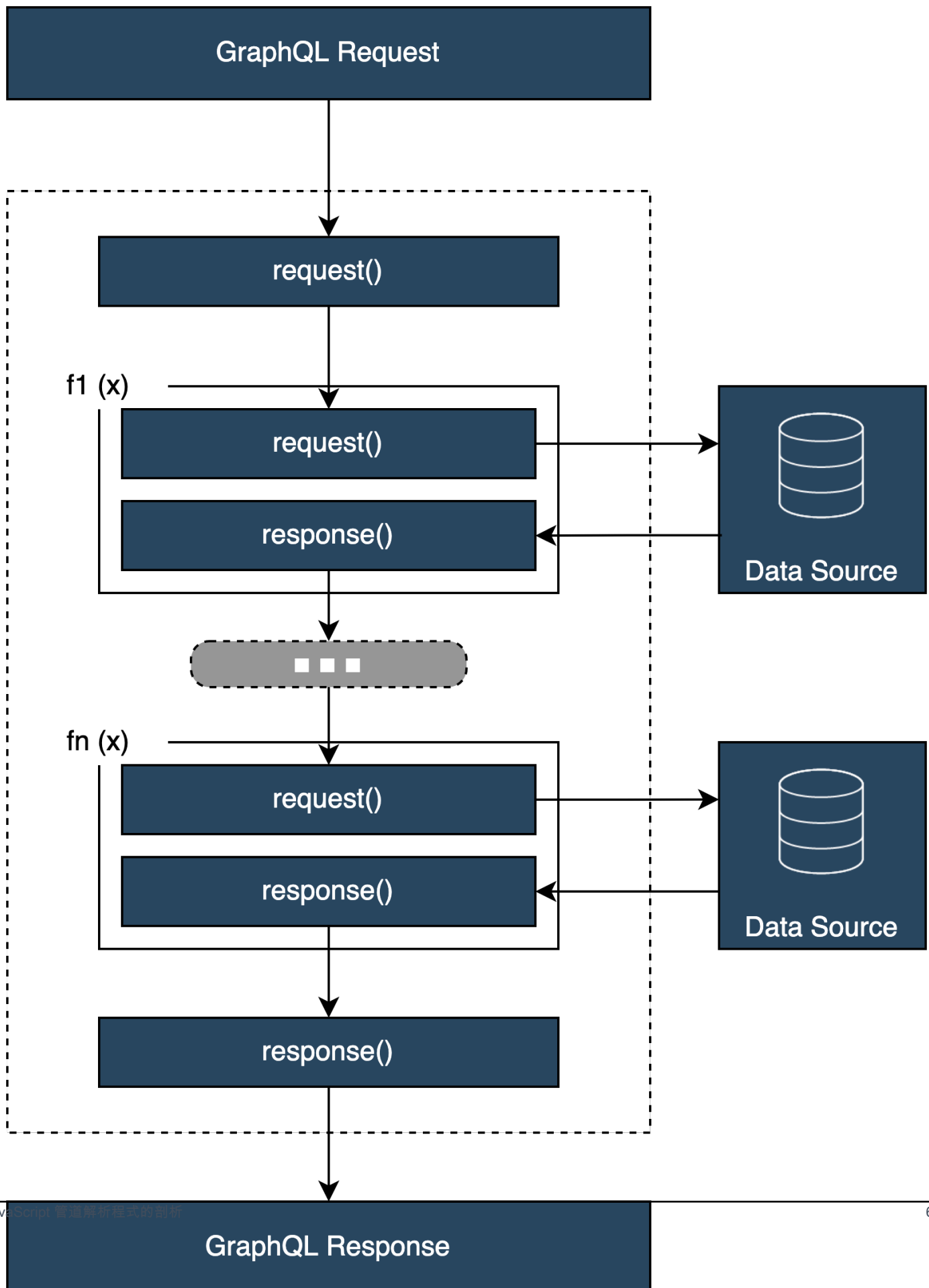
管道解析程式回應處理常式

管道解析程式的回應處理常式可讓您執行從最後一個函數輸出到預期 GraphQL 欄位類型的一些最終邏輯。函數清單中最後一個函數的輸出可在管道解析程式回應處理常式中作為 `ctx.prev.result` 或使用 `ctx.result`。

執行流程

假設管道解析程式包含兩個函數，以下清單代表呼叫解析程式時的執行流程：

1. 管道解析程式請求處理常式
2. 函數 1：函數請求處理常式
3. 第 1 個函數：資料來源呼叫
4. 函數 1：函數回應處理常式
5. 函數 2：函數請求處理常式
6. 第 2 個函數：資料來源呼叫
7. 函數 2：函數回應處理常式
8. 管道解析程式回應處理常式



實用的APPSYNC_JS執行期內建公用程式

下列公用程式可在您使用管道解析程式時提供協助。

`ctx.stash`

停滯是在每個解析程式和函數請求和回應處理常式中可用的物件。相同的堆疊執行個體透過單一解析程式執行運作。這表示您可以使用 `stash` 跨請求和回應處理常式，以及管道解析程式中的函數傳遞任意資料。您可以像一般 JavaScript 物件一樣測試堆疊。

`ctx.prev.result`

`ctx.prev.result` 會顯示管道先前執行操作的結果。如果先前的操作是管道解析程式請求處理常式，則 `ctx.prev.result` 可供鏈結中的第一個函數使用。如果先前操作是第一個函數，則 `ctx.prev.result` 會顯示第一個函數的輸出，並將資料提供給管道中的第二個函數。如果先前的操作是最後一個函數，則 `ctx.prev.result` 代表最後一個函數的輸出，並可供管道解析程式回應處理常式使用。

`util.error`

`util.error` 公用程式非常適合用來擲出欄位錯誤。在函數請求或回應處理常式 `util.error` 內使用會立即擲回欄位錯誤，以防止後續函數執行。如需詳細資訊和其他 `util.error` 簽章，請造訪[解析程式和函數的 JavaScript 執行期功能](#)。

`util.appendError`

`util.appendError` 類似於 `util.error()`，主要區別在於它不會中斷處理常式的評估。相反地，它表示 欄位發生錯誤，但允許評估處理常式並因此傳回資料。在函數中使用 `util.appendError` 並不會中斷管道的執行流程。如需詳細資訊和其他 `util.error` 簽章，請造訪[解析程式和函數的 JavaScript 執行期功能](#)。

`runtime.earlyReturn`

`runtime.earlyReturn` 函數可讓您提早從任何請求函數傳回。在解析程式請求處理常式 `runtime.earlyReturn` 內部使用會從解析程式傳回。從函數 AWS AppSync 請求處理常式呼叫它會從函數傳回，並繼續執行到管道中的下一個函數或解析程式回應處理常式。

撰寫管道解析程式

管道解析程式也有請求和回應處理常式，圍繞管道中函數的執行：其請求處理常式會在第一個函數的請求之前執行，而其回應處理常式會在最後一個函數的回應之後執行。解析程式請求處理常式可以設定要

供管道中的函數使用的資料。解析程式回應處理常式負責傳回映射至 GraphQL 欄位輸出類型的資料。在下列範例中，解析程式請求處理常式會定義 `allowedGroups`；傳回的資料應屬於其中一個群組。解析程式的函數可以使用此值來請求資料。解析程式的回應處理常式會執行最終檢查並篩選結果，以確保只會傳回屬於允許群組的項目。

```
import { util } from '@aws-appsync/utils';

/**
 * Called before the request function of the first AppSync function in the pipeline.
 * @param ctx the context object holds contextual information about the function
 * invocation.
 */
export function request(ctx) {
  ctx.stash.allowedGroups = ['admin'];
  ctx.stash.startedAt = util.time.nowISO8601();
  return {};
}

/**
 * Called after the response function of the last AppSync function in the pipeline.
 * @param ctx the context object holds contextual information about the function
 * invocation.
 */
export function response(ctx) {
  const result = [];
  for (const item of ctx.prev.result) {
    if (ctx.stash.allowedGroups.indexOf(item.group) > -1) result.push(item);
  }
  return result;
}
```

撰寫 AWS AppSync 函數

AWS AppSync 函數可讓您撰寫可在結構描述中的多個解析程式之間重複使用的常見邏輯。例如，您可以有一個稱為 `QUERY_ITEMS` 的 AWS AppSync 函數，負責從 Amazon DynamoDB 資料來源查詢項目。對於您要查詢項目的解析程式，只需將函數新增至解析程式的管道，並提供要使用的查詢索引。邏輯不需要重新實作。

補充主題

主題

- [使用 Amazon DynamoDB 的範例管道解析程式](#)

- [設定APPSYNC_JS執行時間的公程式](#)
- [APPSYNC_JS執行時間的綁定、TypeScript 和來源映射](#)
- [測試您的解析程式和函數處理常式](#)
- [從 VTL 遷移到 JavaScript](#)
- [在直接資料來源存取和透過 Lambda 資料來源代理之間進行選擇](#)

使用 Amazon DynamoDB 的範例管道解析程式

假設您想要在名為 `posts` 的欄位上連接管道解析程式 `getPost(id:ID!)`，該欄位會從具有下列 GraphQL 查詢的 Amazon DynamoDB 資料來源傳回 `Post` 類型：

```
getPost(id:1){
  id
  title
  content
}
```

首先，`Query.getPost` 使用下面的程式碼將簡單的解析程式連接到。這是簡單解析程式程式碼的範例。請求處理常式中未定義邏輯，回應處理常式只會傳回最後一個函數的結果。

```
/**
 * Invoked before the request handler of the first AppSync function in the
 * pipeline.
 * The resolver `request` handler allows to perform some preparation logic
 * before executing the defined functions in your pipeline.
 * @param ctx the context object holds contextual information about the function
 * invocation.
 */
export function request(ctx) {
  return {}
}

/**
 * Invoked after the response handler of the last AppSync function in the pipeline.
 * The resolver `response` handler allows to perform some final evaluation logic
 * from the output of the last function to the expected GraphQL field type.
 * @param ctx the context object holds contextual information about the function
 * invocation.
 */
export function response(ctx) {
```

```
    return ctx.prev.result
  }
```

接著，定義從資料來源擷取後置項目GET_ITEM的函數：

```
import { util } from '@aws-appsync/utils'
import * as ddb from '@aws-appsync/utils/dynamodb'

/**
 * Request a single item from the attached DynamoDB table datasource
 * @param ctx the context object holds contextual information about the function
 * invocation.
 */
export function request(ctx) {
  const { id } = ctx.args
  return ddb.get({ key: { id } })
}

/**
 * Returns the result
 * @param ctx the context object holds contextual information about the function
 * invocation.
 */
export function response(ctx) {
  const { error, result } = ctx
  if (error) {
    return util.appendError(error.message, error.type, result)
  }
  return ctx.result
}
```

如果在請求期間發生錯誤，函數的回應處理常式會附加錯誤，並在 GraphQL 回應中傳回呼叫用戶端。將 GET_ITEM 函數新增至解析程式函數清單。當您執行查詢時，GET_ITEM 函數的請求處理常式會使用 AWS AppSync DynamoDB 模組提供的 utils，以使用 id 做為金鑰來建立 DynamoDBGetItem 請求。ddb.get({ key: { id } }) 會產生適當的 GetItem 操作：

```
{
  "operation" : "GetItem",
  "key" : {
    "id" : { "S" : "1" }
  }
}
```

AWS AppSync 使用 請求從 Amazon DynamoDB 擷取資料。一旦傳回資料，就會由GET_ITEM函數的回應處理常式處理，它會檢查錯誤，然後傳回結果。

```
{
  "result" : {
    "id": 1,
    "title": "hello world",
    "content": "<long story>"
  }
}
```

最後，解析程式的回應處理常式會直接傳回結果。

使用錯誤

如果您的函數在請求期間發生錯誤，則會在 中的函數回應處理常式中提供該錯誤`ctx.error`。您可以使用 `util.appendError` 公用程式將錯誤附加至 GraphQL 回應。您可以使用 `stash`，將錯誤提供給管道中的其他 函數。請參閱以下範例：

```
/**
 * Returns the result
 * @param ctx the context object holds contextual information about the function
 * invocation.
 */
export function response(ctx) {
  const { error, result } = ctx;
  if (error) {
    if (!ctx.stash.errors) ctx.stash.errors = []
    ctx.stash.errors.push(ctx.error)
    return util.appendError(error.message, error.type, result);
  }
  return ctx.result;
}
```

設定APPSYNC_JS執行時間的公用程式

AWS AppSync 提供兩個程式庫，協助開發具有APPSYNC_JS執行時間的解析程式：

- `@aws-appsync/eslint-plugin` - 在開發期間快速攔截和修正問題。
- `@aws-appsync/utils` - 在程式碼編輯器中提供類型驗證和自動完成。

設定 eslint 外掛程式

[ESLint](#) 是一種工具，可靜態分析程式碼以快速找出問題。您可以執行 ESLint 做為持續整合管道的一部分。`@aws-appsync/eslint-plugin` 是一種 ESLint 外掛程式，可在利用 APPSYNC_JS 執行時間時，在您的程式碼中擷取無效的語法。外掛程式可讓您在開發期間快速取得程式碼的意見回饋，而不必將變更推送至雲端。

`@aws-appsync/eslint-plugin` 提供兩個規則集，您可以在開發期間使用。

「`plugin : @aws-appsync/base`」會設定一組基本規則，供您在專案中運用：

規則	描述
無非同步	不支援非同步程序和承諾。
無等待	不支援非同步程序和承諾。
無類別	不支援類別。
無	for 不支援 (for-of 受支援的 for-in 和 除外)
不繼續	不支援 continue。
無生成器	不支援產生器。
無產出	不支援 yield。
無標籤	不支援標籤。
否-此	this 不支援 關鍵字。
無嘗試	不支援 Try/catch 結構。
無同時	雖然不支援迴圈。
no-disallowed-unary-operators	++ 不允許、-- 和 ~ unary 運算子。
no-disallowed-binary-operators	不允許 instanceof 運算子。
無承諾	不支援非同步程序和承諾。

「plugin : @aws-appsync/recommended」提供一些額外的規則，但也需要您將 TypeScript 組態新增至專案。

規則	描述
無遞迴	不允許遞迴函數呼叫。
no-disallowed-methods	不允許某些方法。如需完整的支援內建函數，請參閱 參考 。
no-function-passing	不允許將函數做為函數引數傳遞給函數。
no-function-reassign	無法重新指派函數。
no-function-return	函數不能是函數的傳回值。

若要將外掛程式新增至您的專案，請遵循 [ESLint 入門](#) 中的安裝和使用步驟。然後，使用您的專案套件管理員在您的專案中安裝 [外掛程式](#)（例如 npm、yarn 或 pnpm）：

```
$ npm install @aws-appsync/eslint-plugin
```

在您的 .eslintrc.{js,yml,json} 檔案中，將「plugin : @aws-appsync/base」或「plugin : @aws-appsync/recommended」新增至 extends 屬性。以下程式碼片段是 JavaScript 的基本範例.eslintrc組態：

```
{
  "extends": ["plugin:@aws-appsync/base"]
}
```

若要使用「plugin : @aws-appsync/recommended」規則集，請安裝必要的相依性：

```
$ npm install -D @typescript-eslint/parser
```

然後，建立 .eslintrc.js 檔案：

```
{
  "parser": "@typescript-eslint/parser",
  "parserOptions": {
```

```
"ecmaVersion": 2018,
"project": "./tsconfig.json"
},
"extends": ["plugin:@aws-appsync/recommended"]
}
```

APPSYNC_JS 執行時間的綁定、TypeScript 和來源映射

TypeScript 透過提供類型安全和早期錯誤偵測來增強 AWS AppSync 開發。您可以在本機撰寫 TypeScript 程式碼，並將其轉換為 JavaScript，然後再與 APPSYNC_JS 執行時間搭配使用。程序從安裝 TypeScript 和為 APPSYNC_JS 環境設定 tsconfig.json 開始。然後，您可以使用 esbuild 等綁定工具來編譯和綁定程式碼。Amplify CLI 會從 GraphQL 結構描述產生類型，讓您可以在解析程式碼中使用這些類型。

只要符合 APPSYNC_JS 要求，您就可以在解析程式和函數程式碼中利用自訂和外部程式庫。綁定工具將程式碼合併為單一檔案以供使用 AWS AppSync。可以包含來源映射以協助偵錯。

利用程式庫並綁定程式碼

在解析程式和函數程式碼中，只要符合 APPSYNC_JS 需求，您就可以利用自訂程式庫和外部程式庫。這可讓您在應用程式中重複使用現有的程式碼。若要使用由多個檔案定義的程式庫，您必須使用綁定工具，例如 [esbuild](#)，將程式碼合併到單一檔案中，然後儲存到 AWS AppSync 解析程式或函數。

綁定程式碼時，請記住下列事項：

- APPSYNC_JS 僅支援 ECMAScript 模組 (ESM)。
- @aws-appsync/* 模組已整合至 APPSYNC_JS，且不應與您的程式碼綁定。
- APPSYNC_JS 執行時間環境類似於 NodeJS，因為該程式碼不會在瀏覽器環境中執行。
- 您可以包含選用的來源映射。不過，請勿包含來源內容。

若要進一步了解來源映射，請參閱[使用來源映射](#)。

例如，若要綁定位於 `src/appsync/getPost.resolver.js` 的解析程式碼，您可以使用下列 esbuild CLI 命令：

```
$ esbuild --bundle \  
--sourcemap=inline \  
--sources-content=false \  
--target=esnext \  

```

```
--platform=node \  
--format=esm \  
--external:@aws-appsync/utils \  
--outdir=out/appsync \  
src/appsync/getPost.resolver.js
```

建置程式碼並使用 TypeScript

[TypeScript](#) 是由 Microsoft 開發的一種程式設計語言，可提供 JavaScript 的所有功能以及 TypeScript 類型系統。您可以使用 TypeScript 撰寫安全類型的程式碼，並在建置時擷取錯誤和錯誤，再將程式碼儲存至 AWS AppSync。@aws-appsync/utils 套件已完全輸入。

APPSYNC_JS 執行時間不支援 TypeScript。您必須先將 TypeScript 程式碼轉換為 APPSYNC_JS 執行時間支援的 JavaScript 程式碼，才能將程式碼儲存到其中 AWS AppSync。您可以使用 TypeScript 在本機整合開發環境 (IDE) 中撰寫程式碼，但請注意，您無法在 AWS AppSync 主控台中建立 TypeScript 程式碼。

若要開始使用，請確定您的專案中已安裝 [TypeScript](#)。然後，設定 TypeScript 轉編譯設定，以使用 [TSConfig](#) 搭配 APPSYNC_JS 執行時間。以下是您可以使用的基本 tsconfig.json 檔案範例：

```
// tsconfig.json  
{  
  "compilerOptions": {  
    "target": "esnext",  
    "module": "esnext",  
    "noEmit": true,  
    "moduleResolution": "node",  
  }  
}
```

然後，您可以使用 esbuild 等綁定工具來編譯和綁定程式碼。例如，假設專案的 AWS AppSync 程式碼位於 src/appsync，您可以使用下列命令來編譯和綁定程式碼：

```
$ esbuild --bundle \  
--sourcemap=inline \  
--sources-content=false \  
--target=esnext \  
--platform=node \  
--format=esm \  
--external:@aws-appsync/utils \  
--outdir=out/appsync \  

```

```
src/appsync/**/*.ts
```

使用 Amplify codegen

您可以使用 [Amplify CLI](#) 來產生結構描述的類型。從 `schema.graphql` 檔案所在的目錄中，執行下列命令並檢閱設定 Codegen 的提示：

```
$ npx @aws-amplify/cli codegen add
```

若要在某些情況下（例如，當您的結構描述更新時）重新產生您的 codegen，請執行下列命令：

```
$ npx @aws-amplify/cli codegen
```

然後，您可以在解析程式碼中使用產生的類型。例如，指定下列結構描述：

```
type Todo {
  id: ID!
  title: String!
  description: String
}

type Mutation {
  createTodo(title: String!, description: String): Todo
}

type Query {
  listTodos: Todo
}
```

您可以在下列範例 AWS AppSync 函數中使用產生的類型：

```
import { Context, util } from '@aws-appsync/utils'
import * as ddb from '@aws-appsync/utils/dynamodb'
import { CreateTodoMutationVariables, Todo } from './API' // codegen

export function request(ctx: Context<CreateTodoMutationVariables>) {
  ctx.args.description = ctx.args.description ?? 'created on ' + util.time.nowISO8601()
  return ddb.put<Todo>({ key: { id: util.autoId() }, item: ctx.args })
}

export function response(ctx) {
```



```
return ctx.result as Todo
}
```

在 TypeScript 中使用通用

您可以搭配多種提供的類型使用學名藥。例如，以下程式碼片段是一種Todo類型：

```
export type Todo = {
  __typename: "Todo",
  id: string,
  title: string,
  description?: string | null,
};
```

您可以為使用的訂閱撰寫解析程式Todo。在您的 IDE 中，輸入定義和自動完成提示將引導您正確使用toSubscriptionFilter轉換公用程式：

```
import { util, Context, extensions } from '@aws-appsync/utils'
import { Todo } from './API'

export function request(ctx: Context) {
  return {}
}

export function response(ctx: Context) {
  const filter = util.transform.toSubscriptionFilter<Todo>({
    title: { beginsWith: 'hello' },
    description: { contains: 'created' },
  })
  extensions.setSubscriptionFilter(filter)
  return null
}
```

內嵌您的套件

您可以透過匯入esbuild-plugin-eslint外掛程式來自動繫結套件。然後，您可以提供啟用內嵌功能plugins的值來啟用它。以下是在名為的檔案中使用 esbuild JavaScript API 的程式碼片段build.mjs：

```
/* eslint-disable */
import { build } from 'esbuild'
```

```
import eslint from 'esbuild-plugin-eslint'
import glob from 'glob'
const files = await glob('src/**/*.ts')

await build({
  format: 'esm',
  target: 'esnext',
  platform: 'node',
  external: ['@aws-appsync/utils'],
  outdir: 'dist/',
  entryPoints: files,
  bundle: true,
  plugins: [eslint({ useEslintrc: true })],
})
```

使用來源映射

您可以使用 JavaScript 程式碼提供內嵌來源映射 (sourcemap)。當您綁定 JavaScript 或 TypeScript 程式碼，並想要在日誌和執行時間 JavaScript 錯誤訊息中查看對輸入來源檔案的參考時，來源映射非常有用。

您的 sourcemap 必須出現在程式碼的結尾。它由遵循下列格式的單一註解行定義：

```
//# sourceMappingURL=data:application/json;base64,<base64 encoded string>
```

範例如下：

```

//# sourceMappingURL=data:application/
json;base64,ewogICJ2ZXJzaW9uIjogMywKICAic291cmNlcyI6IFsibGliLmpzIiwgImNvZGUuanMiXSwwKICAibWVwcGl

```

您可以使用 esbuild 建立來源映射。以下範例說明如何在建置和綁定程式碼時，使用 esbuild JavaScript API 來包含內嵌來源映射：

```
/* eslint-disable */
import { build } from 'esbuild'
import eslint from 'esbuild-plugin-eslint'
import glob from 'glob'

const files = await glob('src/**/*.ts')

await build({
  sourcemap: 'inline',
  sourcesContent: false,
```

```

format: 'esm',
target: 'esnext',
platform: 'node',
external: ['@aws-appsync/utils'],
outdir: 'dist/',
entryPoints: files,
bundle: true,
plugins: [eslint({ useEslintrc: true })],
})

```

特別是，`sourcemap`和`sourcesContent`選項指定應該在每個組建結束時逐行新增來源映射，但不應包含來源內容。作為慣例，我們建議您不要在中包含來源內容`sourcemap`。您可以將`sources-content`設定為`false`，在`esbuild`中停用此功能。

若要說明來源映射的運作方式，請檢閱下列解析程式碼參考協助程式程式庫中協助程式函數的範例。此程式碼包含解析程式碼和協助程式程式庫中的日誌陳述式：

`./src/default.resolver.ts`（您的解析程式）

```

import { Context } from '@aws-appsync/utils'
import { hello, logit } from './helper'

export function request(ctx: Context) {
  console.log('start >')
  logit('hello world', 42, true)
  console.log('< end')
  return 'test'
}

export function response(ctx: Context): boolean {
  hello()
  return ctx.prev.result
}

```

`./src/helper.ts`（協助程式檔案）

```

export const logit = (...rest: any[]) => {
  // a special logger
  console.log('[logger]', ...rest.map((r) => `<${r}>`))
}

```

```
export const hello = () => {
  // This just returns a simple sentence, but it could do more.
  console.log('i just say hello..')
}
```

當您建置和綁定解析程式檔案時，您的解析程式程式碼將包含內嵌來源映射。當您的解析程式執行時，以下項目會出現在 CloudWatch 日誌中：

```
INFO - ../src/default.resolver.ts:5:2: "start >"
INFO - ../src/helper.ts:3:2: "[logger]" "<hello world>" "<42>" "<true>"
INFO - ../src/default.resolver.ts:7:2: "< end"
{"logType":"BeforeRequestFunctionEvaluation","path":["logstuff"],"fieldName":"logstuff","resolverArn":"arn:aws:
INFO - ../src/helper.ts:8:2: "i just say hello.."
{"logType":"AfterResponseFunctionEvaluation","path":["logstuff"],"fieldName":"logstuff","resolverArn":"arn:aws:
```

查看 CloudWatch 日誌中的項目，您會注意到這兩個檔案的功能已綁定在一起並同時執行。每個檔案的原始檔案名稱也會清楚地反映在日誌中。

在中測試您的解析程式和函數處理常式 AWS AppSync

您可以使用 EvaluateCode API 命令，在將程式碼儲存到解析程式或函數之前，使用模擬資料遠端測試解析程式和函數處理常式。若要開始使用 命令，請確定您已將 `appsync:evaluatecode` 許可新增至政策。例如：

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "appsync:evaluateCode",
      "Resource": "arn:aws:appsync:<region>:<account>:*"
    }
  ]
}
```

您可以使用 [AWS CLI](#) 或 [AWS SDKs](#) 來利用 命令。例如，若要使用 CLI 測試程式碼，只需指向 檔案、提供內容，並指定您要評估的處理常式：

```
aws appsync evaluate-code \
  --code file://code.js \
  --function request \
  --context file://context.json \
  --runtime name=APPSYNC_JS,runtimeVersion=1.0.0
```

回應包含 `evaluationResult` 其中包含處理常式傳回的承載。它還包含物件，該logs物件會保留您的處理常式在評估期間產生的日誌清單。這可讓您輕鬆地偵錯程式碼執行，並查看評估的相關資訊，以協助故障診斷。例如：

```
{
  "evaluationResult": "{\\\"operation\\\":\\\"PutItem\\\",\\\"key\\\":{\\\"id\\\":{\\\"S\\\":\\\"record-id\\\"}},\\\"attributeValues\\\":{\\\"owner\\\":{\\\"S\\\":\\\"John doe\\\"}},\\\"expectedVersion\\\":{\\\"N\\\":2},\\\"authorId\\\":{\\\"S\\\":\\\"Sammy Davis\\\"}}}",
  "logs": [
    "INFO - code.js:5:3: \\\"current id\\\" \\\"record-id\\\"\"",
    "INFO - code.js:9:3: \\\"request evaluated\\\"\""
  ]
}
```

評估結果可以剖析為 JSON，並提供：

```
{
  "operation": "PutItem",
  "key": {
    "id": {
      "S": "record-id"
    }
  },
  "attributeValues": {
    "owner": {
      "S": "John doe"
    },
    "expectedVersion": {
      "N": 2
    },
    "authorId": {
      "S": "Sammy Davis"
    }
  }
}
```

使用 SDK，您可以輕鬆整合測試套件中的測試，以驗證程式碼的行為。此處的範例使用 [Jest 測試架構](#)，但任何測試套件皆可運作。下列程式碼片段顯示假設性驗證執行。請注意，我們預期評估回應是有效的 JSON，因此我們使用 `從字串回應JSON.parse` 擷取 JSON：

```
const AWS = require('aws-sdk')
const fs = require('fs')
const client = new AWS.AppSync({ region: 'us-east-2' })
const runtime = {name:'APPSYNC_JS',runtimeVersion:'1.0.0'}

test('request correctly calls DynamoDB', async () => {
  const code = fs.readFileSync('./code.js', 'utf8')
  const context = fs.readFileSync('./context.json', 'utf8')
  const contextJSON = JSON.parse(context)

  const response = await client.evaluateCode({ code, context, runtime, function:
'request' }).promise()
  const result = JSON.parse(response.evaluationResult)

  expect(result.key.id.S).toBeDefined()
  expect(result.attributeValues.firstname.S).toEqual(contextJSON.arguments.firstname)
})
```

這會產生下列結果：

```
Ran all test suites.
> jest

PASS ./index.test.js
# request correctly calls DynamoDB (543 ms)
Test Suites: 1 passed, 1 total
Tests: 1 passed, 1 total
Snapshots: 0 totalTime: 1.511 s, estimated 2 s
```

在中從 VTL 遷移至 JavaScript AWS AppSync

AWS AppSync 可讓您使用 VTL 或 JavaScript 為您的解析程式和函數撰寫商業邏輯。使用這兩種語言，您可以撰寫邏輯，指示 AWS AppSync 服務如何與資料來源互動。使用 VTL，您撰寫的映射範本必須評估為有效的 JSON 編碼字串。使用 JavaScript，您可以撰寫傳回物件的請求和回應處理常式。您不會傳回 JSON 編碼字串。

例如，採用下列 VTL 映射範本來取得 Amazon DynamoDB 項目：

```
{
  "operation": "GetItem",
  "key": {
    "id": $util.dynamodb.toDynamoDBJson($ctx.args.id),
  }
}
```

公用程式`$util.dynamodb.toDynamoDBJson`會傳回 JSON 編碼字串。如果 `$ctx.args.id` 設定為 `<id>`，範本會評估為有效的 JSON 編碼字串：

```
{
  "operation": "GetItem",
  "key": {
    "id": {"S": "<id>"},
  }
}
```

使用 JavaScript 時，您不需要在程式碼中列印出原始 JSON 編碼字串，而且`toDynamoDBJson`不需要像一樣使用公用程式。上述映射範本的同等範例為：

```
import { util } from '@aws-appsync/utils';
export function request(ctx) {
  return {
    operation: 'GetItem',
    key: {id: util.dynamodb.toDynamoDB(ctx.args.id)}
  };
}
```

另一種方法是使用 `util.dynamodb.toMapValues`，這是處理值物件的建議方法：

```
import { util } from '@aws-appsync/utils';
export function request(ctx) {
  return {
    operation: 'GetItem',
    key: util.dynamodb.toMapValues({ id: ctx.args.id }),
  };
}
```

這會評估為：

```
{
```

```

"operation": "GetItem",
"key": {
  "id": {
    "S": "<id>"
  }
}
}

```

Note

我們建議您搭配 DynamoDB 資料來源使用 DynamoDB 模組：

```

import * as ddb from '@aws-appsync/utils/dynamodb'

export function request(ctx) {
  ddb.get({ key: { id: ctx.args.id } })
}

```

另一個範例是採用下列映射範本，將項目放入 Amazon DynamoDB 資料來源中：

```

{
  "operation" : "PutItem",
  "key" : {
    "id": $util.dynamodb.toDynamoDBJson($util.autoId()),
  },
  "attributeValues" : $util.dynamodb.toMapValuesJson($ctx.args)
}

```

評估時，此映射範本字串必須產生有效的 JSON 編碼字串。使用 JavaScript 時，您的程式碼會直接傳回請求物件：

```

import { util } from '@aws-appsync/utils';
export function request(ctx) {
  const { id = util.autoId(), ...values } = ctx.args;
  return {
    operation: 'PutItem',
    key: util.dynamodb.toMapValues({ id }),
    attributeValues: util.dynamodb.toMapValues(values),
  };
}

```


評估為：

```
{
  "operation": "PutItem",
  "key": {
    "id": { "S": "2bff3f05-ff8c-4ed8-92b4-767e29fc4e63" }
  },
  "attributeValues": {
    "firstname": { "S": "Shaggy" },
    "age": { "N": 4 }
  }
}
```

Note

我們建議您搭配 DynamoDB 資料來源使用 DynamoDB 模組：

```
import { util } from '@aws-appsync/utils'
import * as ddb from '@aws-appsync/utils/dynamodb'

export function request(ctx) {
  const { id = util.autoId(), ...item } = ctx.args
  return ddb.put({ key: { id }, item })
}
```

在直接資料來源存取和透過 Lambda 資料來源代理之間進行選擇

透過 AWS AppSync 和 APPSYNC_JS 執行時間，您可以使用 AWS AppSync 函數存取資料來源，撰寫自己的程式碼來實作自訂商業邏輯。這可讓您輕鬆地直接與資料來源互動，例如 Amazon DynamoDB、Aurora Serverless、OpenSearch Service、HTTP APIs 和其他 AWS 服務，而無需部署額外的運算服務或基礎設施。AWS AppSync 也可透過設定 Lambda 資料來源，輕鬆與 AWS Lambda 函數互動。Lambda 資料來源可讓您使用 AWS Lambda 的完整集功能來執行複雜的商業邏輯，以解決 GraphQL 請求。在大多數情況下，直接連接到其目標資料來源的 AWS AppSync 函數將提供您需要的所有功能。在您需要實作 APPSYNC_JS 執行時間不支援的複雜商業邏輯的情況下，您可以使用 Lambda 資料來源做為代理來與目標資料來源互動。

直接資料來源整合

做為代理的 Lambda 資料來源

使用案例	AWS AppSync functions interact directly with API data sources.	AWS AppSync functions call Lambdas that interact with API data sources.
Runtime	APPSYNC_JS (JavaScript)	任何支援的 Lambda 執行時間
Maximum size of code	per AWS AppSync 函數 32,000 個字元	每個 Lambda 50 MB (壓縮, 用於直接上傳)
External modules	有限 - 僅限 APPSYNC_JS 支援的功能	是
Call any AWS service	是 - Using AWS AppSync HTTP 資料來源	是 - 使用 AWS SDK
Access to the request header	是	是
Network access	否	是
File system access	否	是
Logging and metrics	是	是
Build and test entirely within AppSync	是	否
Cold start	否	否 - 使用佈建並行
Auto-scaling	是 - 透明地 by AWS AppSync	是 - 如 Lambda 中所設定
Pricing	不收取其他費用	按 Lambda 用量收費

直接與其目標資料來源整合的AWS AppSync 函數非常適合下列使用案例：

- 與 Amazon DynamoDB、Aurora Serverless 和 OpenSearch Service 互動
- 與 HTTP APIs 互動並傳遞傳入標頭
- 使用 HTTP 資料來源與 AWS 服務互動 (使用提供的資料來源角色 AWS AppSync 自動簽署請求)
- 在存取資料來源之前實作存取控制
- 在滿足請求之前，實作擷取資料的篩選

- 在解析程式管道中以循序執行 AWS AppSync 函數來實作簡單的協同運作
- 在查詢和變動中控制快取和訂閱連線。

使用 Lambda 資料來源做為代理的AWS AppSync 函數非常適合下列使用案例：

- 使用 JavaScript 或速度範本語言 (VTL) 以外的語言
- 調整和控制 CPU 或記憶體以最佳化效能
- 在 中匯入第三方程式庫或需要不支援的功能 APPSYNC_JS
- 提出多個網路請求和/或取得檔案系統存取權以滿足查詢
- 使用批次處理[組態批次處理](#)請求。

AWS AppSync JavaScript 解析程式內容物件參考

AWS AppSync 定義一組變數和函數，用於使用請求和回應處理常式。這樣可讓使用 GraphQL 對資料進行邏輯運算更簡單。本文件說明這些函數並提供範例。

正在存取 **context**

請求和回應處理常式的context引數是物件，保留解析程式調用的所有內容資訊。其結構如下：

```
type Context = {
  arguments: any;
  args: any;
  identity: Identity;
  source: any;
  error?: {
    message: string;
    type: string;
  };
  stash: any;
  result: any;
  prev: any;
  request: Request;
  info: Info;
};
```

Note

您通常會發現 context 物件稱為 `ctx`。

context 物件中的每個欄位定義如下：

context 欄位**arguments**

包含此欄位所有 GraphQL 引數的映射。

identity

包含有關發起人資訊的物件。如需此欄位結構的詳細資訊，請參閱[身分](#)。

source

包含父欄位解析度的映射。

stash

堆疊是在每個解析程式和函數處理常式中可用的物件。相同的 stash 物件透過單一解析程式執行來存留。這表示您可以使用 stash 跨請求和回應處理常式，以及管道解析程式中的 函數傳遞任意資料。

Note

您無法刪除或取代整個停滯，但您可以新增、更新、刪除和讀取停滯的屬性。

您可以修改下列其中一個程式碼範例，將項目新增至堆疊：

```
//Example 1
ctx.stash.newItem = { key: "something" }

//Example 2
Object.assign(ctx.stash, {key1: value1, key2: value})
```

您可以修改以下程式碼，從堆疊中移除項目：

```
delete ctx.stash.key
```

result

此解析程式結果的容器。此欄位僅適用於回應處理常式。

例如，如果您要解決下列查詢author的欄位：

```
query {  
  getPost(id: 1234) {  
    postId  
    title  
    content  
    author {  
      id  
      name  
    }  
  }  
}
```

評估回應處理常式時，即可使用完整context變數：

```
{  
  "arguments" : {  
    id: "1234"  
  },  
  "source": {},  
  "result" : {  
    "postId": "1234",  
    "title": "Some title",  
    "content": "Some content",  
    "author": {  
      "id": "5678",  
      "name": "Author Name"  
    }  
  },  
  "identity" : {  
    "sourceIp" : ["x.x.x.x"],  
    "userArn" : "arn:aws:iam::123456789012:user/appsync",  
    "accountId" : "666666666666",  
    "user" : "AIDAAAAAAAAAAAAAAAAAAAA"  
  }  
}
```

```
}  
}
```

prev.result

在管道解析程式中執行任何先前操作的結果。

如果先前的操作是管道解析程式的請求處理常式，則 `ctx.prev.result` 代表該評估結果，並可供管道中的第一個函數使用。

如果先前的操作是第一個函數，則 `ctx.prev.result` 代表第一個函數回應處理常式的評估結果，並可供管道中的第二個函數使用。

如果先前的操作是最後一個函數，則 `ctx.prev.result` 代表最後一個函數的評估結果，並可供管道解析程式的回應處理常式使用。

info

包含有關 GraphQL 請求資訊的物件。如需此欄位的結構，請參閱[資訊](#)。

Identity

包含有關發起人資訊的 `identity` 區段。本節的形狀取決於 AWS AppSync API 的授權類型。

如需 AWS AppSync 安全選項的詳細資訊，請參閱[授權和身分驗證](#)。

API_KEY 授權

系統不會填入 `identity` 欄位。

AWS_LAMBDA 授權

`identity` 有下列形式：

```
type AppSyncIdentityLambda = {  
  resolverContext: any;  
};
```

`identity` 包含 `resolverContext` 金鑰，其中包含由授權請求的 Lambda 函數傳回的相同 `resolverContext` 內容。

AWS_IAM 授權

`identity` 有下列形式：

```
type AppSyncIdentityIAM = {  
  accountId: string;  
  cognitoIdentityPoolId: string;  
  cognitoIdentityId: string;  
  sourceIp: string[];  
  username: string;  
  userArn: string;  
  cognitoIdentityAuthType: string;  
  cognitoIdentityAuthProvider: string;  
};
```

AMAZON_COGNITO_USER_POOLS 授權

identity 有下列形式：

```
type AppSyncIdentityCognito = {  
  sourceIp: string[];  
  username: string;  
  groups: string[] | null;  
  sub: string;  
  issuer: string;  
  claims: any;  
  defaultAuthStrategy: string;  
};
```

每個欄位的定義如下：

accountId

發起人 AWS 的帳戶 ID。

claims

使用者擁有的宣告。

cognitoIdentityAuthType

經身分驗證或未經身分驗證 (根據身分類型)。

cognitoIdentityAuthProvider

以逗號分隔的外部身分提供者資訊清單，用於取得用於簽署請求的登入資料。

cognitoIdentityId

發起人的 Amazon Cognito 身分 ID。

cognitoIdentityPoolId

與發起人相關聯的 Amazon Cognito 身分集區 ID。

defaultAuthStrategy

此發起人 (ALLOW 或 DENY) 的預設授權策略。

issuer

字符發行者。

sourceIp

AWS AppSync 接收的發起人的來源 IP 地址。如果請求不包含 x-forwarded-for 標頭，來源 IP 值只會包含 TCP 連線的單一 IP 地址。如果要求包含 x-forwarded-for 標頭，則來源 IP 除了有 TCP 連線的 IP 地址外，也將有 x-forwarded-for 標頭中 IP 地址的清單。

sub

已驗證使用者的 UUID。

user

IAM 使用者。

userArn

IAM 使用者的 Amazon Resource Name (ARN)。

username

已驗證使用者的使用者名稱。如果是 AMAZON_COGNITO_USER_POOLS 授權，使用者名稱的值是屬性 cognito:username 的值。在 AWS_IAM 授權的情況下，使用者名稱的值是 AWS 使用者主體的值。如果您使用 IAM 授權搭配來自 Amazon Cognito 身分集區的憑證，我們建議您使用 cognitoIdentityId。

存取請求標頭

AWS AppSync 支援從用戶端傳遞自訂標頭，並使用在 GraphQL 解析程式中存取它們 `ctx.request.headers`。然後，您可以將標頭值用於動作，例如將資料插入資料來源或授權檢查。您可以從命令列使用 `$curl` 搭配 API 金鑰的一或多個請求標頭，如下列範例所示：

單一標頭範例

假設您設定使用 custom 值的 nadia 標頭，如下所示：

```
curl -XPOST -H "Content-Type:application/graphql" -H "custom:nadia" -H "x-api-key:<API-KEY-VALUE>" -d '{"query":"mutation { createEvent(name: \"demo\", when: \"Next Friday!\", where: \"Here!\") {id name when where description}}"}' https://<ENDPOINT>/graphql
```

可使用 `ctx.request.headers.custom` 對此進行存取。例如，它可能位於 DynamoDB 的下列程式碼中：

```
"custom": util.dynamodb.toDynamoDB(ctx.request.headers.custom)
```

多個標頭範例

您也可以在單一請求中傳遞多個標頭，並在解析程式處理常式中存取這些標頭。例如，如果 custom 標頭設定為兩個值：

```
curl -XPOST -H "Content-Type:application/graphql" -H "custom:bailey" -H "custom:nadia" -H "x-api-key:<API-KEY-VALUE>" -d '{"query":"mutation { createEvent(name: \"demo\", when: \"Next Friday!\", where: \"Here!\") {id name when where description}}"}' https://<ENDPOINT>/graphql
```

然後，您可以將這些存取為陣列，例如 `ctx.request.headers.custom[1]`。

Note

AWS AppSync 不會在 中公開 Cookie 標頭 `ctx.request.headers`。

存取請求自訂網域名稱

AWS AppSync 支援設定自訂網域，您可以使用該網域來存取 APIs 的 GraphQL 和即時端點。使用自訂網域名稱提出請求時，您可以使用 取得網域名稱 `ctx.request.domainName`。

使用預設 GraphQL 端點網域名稱時，值為 `null`。

Info

info 區段包含有關 GraphQL 請求的資訊。本節的格式如下：

```
type Info = {  
  fieldName: string;  
  parentTypeName: string;  
  variables: any;  
  selectionSetList: string[];  
  selectionSetGraphQL: string;  
};
```

每個欄位的定義如下：

fieldName

目前正在解析的欄位名稱。

parentTypeName

目前正在解析的欄位父類型名稱。

variables

包含傳遞給 GraphQL 請求之所有變數的映射。

selectionSetList

此清單表示 GraphQL 選取範圍中的欄位。別名的欄位僅由別名名稱參考，而非欄位名稱。以下範例詳細說明這一點。

selectionSetGraphQL

此字串表示格式為 GraphQL 結構描述定義語言 (SDL) 的選取範圍。雖然片段不會合併至選取集，但會保留內嵌片段，如下列範例所示。

Note

JSON.stringify 不會在字串序列化selectionSetList中包含selectionSetGraphQL和。您必須直接參考這些屬性。

例如，如果您解析下列查詢的 getPost 欄位：

```
query {  
  getPost(id: $postId) {  
    postId
```

```

    title
    secondTitle: title
    content
    author(id: $authorId) {
      authorId
      name
    }
    secondAuthor(id: "789") {
      authorId
    }
    ... on Post {
      inlineFrag: comments: {
        id
      }
    }
    ... postFrag
  }
}

fragment postFrag on Post {
  postFrag: comments: {
    id
  }
}

```

然後，處理處理常式時可用的完整ctx.info變數可能是：

```

{
  "fieldName": "getPost",
  "parentTypeName": "Query",
  "variables": {
    "postId": "123",
    "authorId": "456"
  },
  "selectionSetList": [
    "postId",
    "title",
    "secondTitle",
    "content",
    "author",
    "author/authorId",
    "author/name",
    "secondAuthor",
  ]
}

```

```

    "secondAuthor/authorId",
    "inlineFragComments",
    "inlineFragComments/id",
    "postFragComments",
    "postFragComments/id"
  ],
  "selectionSetGraphQL": "{\n  getPost(id: $postId) {\n    postId\n    title\n    secondTitle: title\n    content\n    author(id: $authorId) {\n      authorId\n      name\n    }\n    secondAuthor(id: \"789\") {\n      authorId\n    }\n    ... on Post {\n      inlineFrag: comments {\n        id\n      }\n    }\n    ... postFrag\n  }\n}"
}

```

`selectionSetList` 只會公開屬於目前類型的欄位。如果目前類型是界面或聯集，則只會公開屬於界面的所選欄位。例如，指定下列結構描述：

```

type Query {
  node(id: ID!): Node
}

interface Node {
  id: ID
}

type Post implements Node {
  id: ID
  title: String
  author: String
}

type Blog implements Node {
  id: ID
  title: String
  category: String
}

```

以及下列查詢：

```

query {
  node(id: "post1") {
    id
    ... on Post {
      title
    }
  }
}

```

```
... on Blog {  
  title  
}  
}  
}
```

`ctx.info.selectionSetList` 以 `Query.node` 欄位解析度呼叫時，只會 `id` 公開：

```
"selectionSetList": [  
  "id"  
]
```

解析程式和函數的AWS AppSync JavaScript 執行期功能

APPSYNC_JS 執行時間環境提供類似於 [ECMAScript \(ES\) 6.0 版的功能](#)。它支援其功能子集，並提供一些不屬於 ES 規格的其他方法（公程式）。下列主題列出所有支援的語言功能：

- [支援的執行期功能](#) - 進一步了解支援的核心功能、基本物件、內建物件和函數等。
- [內建公程式](#) - `util` 變數包含可協助您處理資料的一般公程式方法。除非另行指定，否則所有公程式皆使用 UTF-8 字元集。
- [內建模組](#) - 進一步了解內建模組如何協助撰寫 JavaScript 解析程式和函數。
- [執行期公程式](#) - 執行期程式庫提供公程式來控制或修改解析程式和函數的執行期屬性。
- [util.time 中的時間協助程式](#) - `util.time` 變數包含日期時間方法，可協助產生時間戳記、在日期時間格式之間轉換，以及剖析日期時間字串。日期時間格式的語法是以 [DateTimeFormatter](#) 為基礎，您可以參考以取得進一步的文件。
- [util.dynamodb 中的 DynamoDB 協助程式](#) - `util.dynamodb` 包含協助程式方法，可讓您更輕鬆地將資料寫入和讀取至 Amazon DynamoDB，例如自動類型映射和格式化。
- [util.http 中的 HTTP 協助程式](#) - `util.http` 公程式提供可用於管理 HTTP 請求參數和新增回應標頭的協助程式方法。
- [util.transform 中的轉換協助程式](#) - `util.transform` 包含協助程式方法，可讓您更輕鬆地對資料來源執行複雜的操作。
- [util.str 中的字串協助程式](#) - `util.str` 包含有助於常見字串操作的方法。
- [延伸模組](#) - 延伸模組包含一組方法，可在解析程式中執行其他動作。
- [util.xml 中的 XML 協助程式](#) - `util.xml` 包含有助於 XML 字串轉換的方法。

 Note

目前，此參考僅適用於執行時間 1.0.0 版。

支援的執行時間功能

以下各節說明 APPSYNC_JS 執行時間的支援功能集。

核心功能

支援下列核心功能。

Types

支援下列類型：

- 數字
- 字串
- 布林值
- objects
- 陣列
- 函數

Operators

支援運算子，包括：

- 標準數學運算子 (+、-、/%、[^]、* 等)
- nullish coalescing 運算子 (??)
- 選用鏈結 (?.)
- 位元運算子
- void 和 typeof 運算子
- 分散運算子 (...)

不支援下列運算子：

- 統一運算子 (++, -- 和 ~)
- in運算子

 Note

使用 `Object.hasOwn` 運算子來檢查指定的屬性是否位於指定的物件中。

Statements

支援下列陳述式：

- `const`
- `let`
- `var`
- `break`
- `else`
- `for-in`
- `for-of`
- `if`
- `return`
- `switch`
- 分散語法

不支援下列項目：

- `catch`
- `continue`
- `do-while`
- `finally`
- `for(initialization; condition; afterthought)`

 Note

支援的例外是 `for-in` 和 `for-of` 表達式。

- `throw`
- `try`
- `while`
- 標籤陳述式

Literals

支援下列 ES 6 [範本常值](#)：

- 多行字串
- 表達式插補
- 巢狀化範本

Functions

支援下列函數語法：

- 支援函數宣告。
- 支援 ES 6 箭頭函數。
- 支援 ES 6 靜態參數語法。

Strict mode

函數按預設會在嚴格模式下運作，因此您不需要在函數程式碼中新增 `use_strict` 陳述式。無法對此進行變更。

基本物件

支援 ES 及其函數的下列基本物件。

Object

支援下列物件：

- `Object.assign()`
- `Object.entries()`
- `Object.hasOwn()`
- `Object.keys()`
- `Object.values()`
- `delete`

String

支援下列字串：

- `String.prototype.length()`
- `String.prototype.charAt()`
- `String.prototype.concat()`
- `String.prototype.endsWith()`
- `String.prototype.indexOf()`
- `String.prototype.lastIndexOf()`
- `String.raw()`
- `String.prototype.replace()`

Note

不支援規則運算式。

不過，提供的參數中支援 Java 樣式的規則表達式建構。如需詳細資訊，請參閱[模式](#)。

- `String.prototype.replaceAll()`

Note

不支援規則運算式。

不過，提供的參數中支援 Java 樣式的規則表達式建構。如需詳細資訊，請參閱[模式](#)。

- `String.prototype.slice()`
- `String.prototype.split()`
- `String.prototype.startsWith()`
- `String.prototype.toLowerCase()`
- `String.prototype.toUpperCase()`
- `String.prototype.trim()`
- `String.prototype.trimEnd()`
- `String.prototype.trimStart()`

Number

支援下列號碼：

- `Number.isFinite`
- `Number.isNaN`

內建物件和函數

支援下列函數和物件。

Math

支援下列數學函數：

- `Math.random()`
- `Math.min()`
- `Math.max()`
- `Math.round()`
- `Math.floor()`
- `Math.ceil()`

Array

支援下列陣列方法：

- `Array.prototype.length`

- `Array.prototype.concat()`
- `Array.prototype.fill()`
- `Array.prototype.flat()`
- `Array.prototype.indexOf()`
- `Array.prototype.join()`
- `Array.prototype.lastIndexOf()`
- `Array.prototype.pop()`
- `Array.prototype.push()`
- `Array.prototype.reverse()`
- `Array.prototype.shift()`
- `Array.prototype.slice()`
- `Array.prototype.sort()`

**Note**

`Array.prototype.sort()` 不支援引數。

- `Array.prototype.splice()`
- `Array.prototype.unshift()`
- `Array.prototype.forEach()`
- `Array.prototype.map()`
- `Array.prototype.flatMap()`
- `Array.prototype.filter()`
- `Array.prototype.reduce()`
- `Array.prototype.reduceRight()`
- `Array.prototype.find()`
- `Array.prototype.some()`
- `Array.prototype.every()`
- `Array.prototype.findIndex()`
- `Array.prototype.findLast()`
- `Array.prototype.findLastIndex()`
- `delete`

Console

主控台物件可用於偵錯。在即時查詢執行期間，主控台日誌/錯誤陳述式會傳送至 Amazon CloudWatch Logs（如果啟用記錄）。使用 `evaluateCode` 進行程式碼評估期間，日誌陳述式會在命令回應中傳回。

- `console.error()`
- `console.log()`

Function

- 不支援 `bind`、`apply` 和 `call` 方法。
- 不支援函數建構子。
- 不支援將函數做為引數傳遞。
- 不支援遞迴函數呼叫。

JSON

支援下列 JSON 方法：

- `JSON.parse()`



Note

如果剖析的字串不是有效的 JSON，則傳回空白字串。

- `JSON.stringify()`

Promises

不支援非同步程序，也不支援 `promise`。



Note

在 `in.APPSYNC_JS` 執行時間內不支援網路和檔案系統存取 AWS AppSync。會根據 AWS AppSync 解析程式或 AWS AppSync 函數提出的請求，AWS AppSync 來處理所有 I/O 操作。

全域變數

支援以下全局常數：

- NaN
- Infinity
- undefined
- [util](#)
- [extensions](#)
- [runtime](#)

錯誤類型

throw 不支援使用 擲回錯誤。您可以使用 `util.error()` 函數傳回錯誤。您可以使用 `util.appendError` 函數在 GraphQL 回應中包含錯誤。

如需詳細資訊，請參閱[錯誤 utils](#)。

內建公用程式

util 變數包含可協助您處理資料的一般公用程式方法。除非另行指定，否則所有公用程式皆使用 UTF-8 字元集。

編碼 utils

編碼 utils 清單

util.urlEncode(String)

以 application/x-www-form-urlencoded 編碼字串的形式傳回輸入字串。

util.urlDecode(String)

將 application/x-www-form-urlencoded 編碼的字串解碼回非編碼格式。

util.base64Encode(string) : string

將輸入編碼為 base64 編碼字串。

util.base64Decode(string) : string

解碼 base64 編碼字串中的資料。

ID 產生 utils

ID 產生 utils 清單

util.autoId()

傳回 128 位元隨機產生的 UUID。

util.autoUlid()

傳回 128 位元隨機產生的 ULID (通用可排序序列識別符)。

util.autoKsuid()

傳回 128 位元隨機產生的 KSUID (K-Sortable Unique Identifier) base62，編碼為長度為 27 的字串。

錯誤率

錯誤使用率清單

util.error(String, String?, Object?, Object?)

擲回自訂錯誤。如果範本偵測到要求或呼叫結果的錯誤，您可以將此用於要求或回應映射範本。此外，還可以指定 `errorType` 欄位、`data` 欄位和 `errorInfo` 欄位。`data` 值將新增到 GraphQL 回應中，`error` 內對應的 `errors` 區塊。

Note

`data` 將根據查詢選取集進行篩選。`errorInfo` 值將新增到 GraphQL 回應中，`error` 內對應的 `errors` 區塊。

`errorInfo` 不會根據查詢選取集進行篩選。

util.appendError(String, String?, Object?, Object?)

附加自訂錯誤。如果範本偵測到要求或呼叫結果的錯誤，您可以將此用於要求或回應映射範本。此外，還可以指定 `errorType` 欄位、`data` 欄位和 `errorInfo` 欄位。與 `util.error(String, String?, Object?, Object?)` 不同的是，範本評估不會受中斷，因此可以將資料傳回給發起人。`data` 值將新增到 GraphQL 回應中，`error` 內對應的 `errors` 區塊。

Note

`data` 將根據查詢選取集進行篩選。`errorInfo` 值將新增到 GraphQL 回應中，`error` 內對應的 `errors` 區塊。

`errorInfo` 不會根據查詢選取集進行篩選。

類型和模式比對 utils

類型和模式比對 utils 清單

util.matches(String, String) : Boolean

如果第一個引數中指定的模式與第二個引數中提供的資料相符，則傳回真。模式必須為規則表達式，例如 `util.matches("a*b", "aaaaab")`。此功能是根據[模式](#)，您可以參考以取得更詳細的文件。

util.authType()

傳回描述請求所使用之多重身分類型的字串，傳回「IAM 授權」、「使用者集區授權」、「開放式 ID Connect 授權」或「API 金鑰授權」。

傳回值行為 utils

傳回值行為使用率清單

util.escapeJavaScript(String)

以 JavaScript 逸出字串的形式傳回輸入字串。

解析程式授權 utils

解析程式授權使用率清單

util.unauthorized()

擲回欲解析之欄位的 `Unauthorized`。在請求或回應映射範本中使用此選項，以決定是否允許發起人解析 欄位。

內建模組

模組是APPSYNC_JS執行時間的一部分，並提供公用程式以協助撰寫 JavaScript 解析程式和函數。如需範例和範例，請參閱 [aws-appsync-resolver-samples](#) GitHub 儲存庫。

DynamoDB 模組函數

DynamoDB 模組函數可在與 DynamoDB 資料來源互動時提供增強的體驗。您可以使用 函數向 DynamoDB 資料來源提出請求，而無需新增類型映射。

使用 匯入模組@aws-appsync/utils/dynamodb：

```
// Modules are imported using @aws-appsync/utils/dynamodb
import * as ddb from '@aws-appsync/utils/dynamodb';
```

函數

函數清單

get<T>(payload: GetInput): DynamoDBGetItemRequest

Tip

如需 GetInput 的相關資訊[the section called “輸入”](#)，請參閱。

產生DynamoDBGetItemRequest物件以向 DynamoDB 提出 [GetItem](#) 請求。

```
import { get } from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
  return get({ key: { id: ctx.args.id } });
}
```

put<T>(payload): DynamoDBPutItemRequest

產生DynamoDBPutItemRequest物件以向 DynamoDB 提出 [PutItem](#) 請求。

```
import * as ddb from '@aws-appsync/utils/dynamodb'
```



```
export function request(ctx) {  
  return ddb.put({ key: { id: util.autoId() }, item: ctx.args });  
}
```

remove<T>(payload): DynamoDBDeleteItemRequest

產生DynamoDBDeleteItemRequest物件以向 DynamoDB 提出 [DeleteItem](#) 請求。

```
import * as ddb from '@aws-appsync/utils/dynamodb';  
  
export function request(ctx) {  
  return ddb.remove({ key: { id: ctx.args.id } });  
}
```

scan<T>(payload): DynamoDBScanRequest

產生 DynamoDBScanRequest以向 DynamoDB 提出[掃描](#)請求。

```
import * as ddb from '@aws-appsync/utils/dynamodb';  
  
export function request(ctx) {  
  const { limit = 10, nextToken } = ctx.args;  
  return ddb.scan({ limit, nextToken });  
}
```

sync<T>(payload): DynamoDBSyncRequest

產生DynamoDBSyncRequest物件以提出[同步](#)請求。請求只會接收自上次查詢後變更的資料（差異更新）。只能對版本控制的 DynamoDB 資料來源提出請求。

```
import * as ddb from '@aws-appsync/utils/dynamodb';  
  
export function request(ctx) {  
  const { limit = 10, nextToken, lastSync } = ctx.args;  
  return ddb.sync({ limit, nextToken, lastSync });  
}
```

update<T>(payload): DynamoDBUpdateItemRequest

產生DynamoDBUpdateItemRequest物件以向 DynamoDB 提出 [UpdateItem](#) 請求。

作業

操作協助程式可讓您在更新期間對部分資料採取特定動作。若要開始使用，operations請從 匯入@aws-appsync/utils/dynamodb：

```
// Modules are imported using operations
import {operations} from '@aws-appsync/utils/dynamodb';
```

操作清單

add<T>(payload)

在更新 DynamoDB 時新增屬性項目的協助程式函數。

範例

若要使用 ID 值將地址（街道、城市和郵遞區號）新增至現有的 DynamoDB 項目：

```
import { update, operations } from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
  const updateObj = {
    address: operations.add({
      street1: '123 Main St',
      city: 'New York',
      zip: '10001',
    }),
  };
  return update({ key: { id: 1 }, update: updateObj });
}
```

append <T>(payload)

將承載附加至 DynamoDB 中現有清單的協助程式函數。

範例

若要在更新期間將新增IDs (newFriendIds) 附加至現有的朋友清單 (friendsIds)：

```
import { update, operations } from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
  const newFriendIds = [101, 104, 111];
```

```
const updateObj = {
  friendsIds: operations.append(newFriendIds),
};
return update({ key: { id: 1 }, update: updateObj });
}
```

decrement (by?)

在更新 DynamoDB 時減少項目中現有屬性值的協助程式函數。

範例

若要將朋友計數器 (friendsCount) 減少 10：

```
import { update, operations } from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
  const updateObj = {
    friendsCount: operations.decrement(10),
  };
  return update({ key: { id: 1 }, update: updateObj });
}
```

increment (by?)

在更新 DynamoDB 時遞增項目中現有屬性值的協助程式函數。

範例

若要將好友計數器 (friendsCount) 增加 10：

```
import { update, operations } from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
  const updateObj = {
    friendsCount: operations.increment(10),
  };
  return update({ key: { id: 1 }, update: updateObj });
}
```

prepend <T>(payload)

在 DynamoDB 中現有清單前面的協助程式函數。

範例

若要在更新期間將新增IDs (newFriendIds) 附加至現有的朋友清單 (friendsIds)：

```
import { update, operations } from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
  const newFriendIds = [101, 104, 111];
  const updateObj = {
    friendsIds: operations.prepend(newFriendIds),
  };
  return update({ key: { id: 1 }, update: updateObj });
}
```

replace <T>(payload)

在 DynamoDB 中更新項目時取代現有屬性的協助程式函數。當您想要更新屬性中的整個物件或子物件時，這非常有用，而不只是承載中的金鑰。

範例

若要取代 info 物件中的地址（街道、城市和郵遞區號）：

```
import { update, operations } from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
  const updateObj = {
    info: {
      address: operations.replace({
        street1: '123 Main St',
        city: 'New York',
        zip: '10001',
      }),
    },
  };
  return update({ key: { id: 1 }, update: updateObj });
}
```

updateListItem <T>(payload, index)

取代清單中項目的協助程式函數。

範例

在更新範圍 (newFriendIds) 中，此範例用於updateListItem更新清單 () 中第二個項目 (索引：1，新 ID：102) 和第三個項目 (索引：2，新 ID：112) 的 ID 值friendsIds。

```
import { update, operations as ops } from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
  const newFriendIds = [
    ops.updateListItem('102', 1), ops.updateListItem('112', 2)
  ];
  const updateObj = { friendsIds: newFriendIds };
  return update({ key: { id: 1 }, update: updateObj });
}
```

輸入

輸入清單

Type GetInput<T>

```
GetInput<T>: {
  consistentRead?: boolean;
  key: DynamoDBKey<T>;
}
```

類型宣告

- consistentRead?: boolean (選用)

選用布林值，用於指定是否要與 DynamoDB 執行強式一致讀取。

- key: DynamoDBKey<T> (必要)

在 DynamoDB 中指定項目索引鍵的必要參數。DynamoDB 項目可能有單一雜湊索引鍵或雜湊和排序索引鍵。

Type PutInput<T>

```
PutInput<T>: {
  _version?: number;
  condition?: DynamoDBFilterObject<T> | null;
  customPartitionKey?: string;
  item: Partial<T>;
  key: DynamoDBKey<T>;
}
```

```
populateIndexFields?: boolean;
}
```

類型宣告

- `_version?: number` (選用)
- `condition?: DynamoDBFilterObject<T> | null` (選用)

當您將物件放入 DynamoDB 資料表時，您可以選擇指定條件式表達式，以根據在執行操作之前已在 DynamoDB 中的物件狀態來控制請求是否成功。

- `customPartitionKey?: string` (選用)

啟用時，此字串值會修改啟用版本控制時差異同步資料表所使用的 `ds_sk` 和 `ds_pk` 記錄格式。啟用時，也會啟用 `populateIndexFields` 項目的處理。

- `item: Partial<T>` (必要)

要放入 DynamoDB 之項目的其餘屬性。

- `key: DynamoDBKey<T>` (必要)

必要參數，指定 DynamoDB 中要對其執行放置的項目索引鍵。DynamoDB 項目可能有單一雜湊索引鍵或雜湊和排序索引鍵。

- `populateIndexFields?: boolean` (選用)

布林值，當與一起啟用時 `customPartitionKey`，會為差異同步資料表中的每個記錄建立新的項目，特別是在 `gsi_ds_pk` 和 `gsi_ds_sk` 欄中。如需詳細資訊，請參閱《AWS AppSync 開發人員指南》中的[衝突偵測和同步](#)。

Type `QueryInput<T>`

```
QueryInput<T>: ScanInput<T> & {
  query: DynamoDBKeyCondition<Required<T>>;
}
```

類型宣告

- `query: DynamoDBKeyCondition<Required<T>>` (必要)

指定描述要查詢之項目的金鑰條件。對於指定的索引，分割區索引鍵的條件應為等式，排序索引鍵應為比較或 `beginsWith` (當它是字串時)。分割區和排序索引鍵僅支援數字和字串類型。

範例

請採用以下User類型：

```
type User = {  
  id: string;  
  name: string;  
  age: number;  
  isVerified: boolean;  
  friendsIds: string[]  
}
```

查詢只能包含下列欄位：id、name和 age：

```
const query: QueryInput<User> = {  
  name: { eq: 'John' },  
  age: { gt: 20 },  
}
```

Type RemoveInput<T>

```
RemoveInput<T>: {  
  _version?: number;  
  condition?: DynamoDBFilterObject<T>;  
  customPartitionKey?: string;  
  key: DynamoDBKey<T>;  
  populateIndexFields?: boolean;  
}
```

類型宣告

- `_version?: number` (選用)
- `condition?: DynamoDBFilterObject<T>` (選用)

當您在 DynamoDB 中移除物件時，您可以選擇指定條件式表達式，以根據在執行操作之前已在 DynamoDB 中的物件狀態來控制請求是否成功。

範例

下列範例是包含條件的DeleteItem表達式，只有在文件擁有者符合提出請求的使用者時，才會允許操作成功。

```
type Task = {
```

```

    id: string;
    title: string;
    description: string;
    owner: string;
    isComplete: boolean;
  }
  const condition: DynamoDBFilterObject<Task> = {
    owner: { eq: 'XXXXXXXXXXXXXXXXXX' },
  }

  remove<Task>({
    key: {
      id: 'XXXXXXXXXXXXXXXXXX',
    },
    condition,
  });

```

- `customPartitionKey?: string` (選用)

啟用時，`customPartitionKey`值會修改啟用版本控制時差異同步資料表所使用的 `ds_sk`和 `ds_pk` 記錄格式。啟用時，也會啟用`populateIndexFields`項目的處理。

- `key: DynamoDBKey<T>` (必要)

必要參數，指定要移除之 DynamoDB 中項目的索引鍵。DynamoDB 項目可能有單一雜湊索引鍵或雜湊和排序索引鍵。

範例

如果User只有使用者 的雜湊金鑰`id`，則金鑰看起來會如下所示：

```

type User = {
  id: number
  name: string
  age: number
  isVerified: boolean
}
const key: DynamoDBKey<User> = {
  id: 1,
}

```

如果資料表使用者具有雜湊索引鍵 (`id`) 和排序索引鍵 (`name`)，則索引鍵看起來會如下所示：


```

type User = {
  id: number
  name: string
  age: number
  isVerified: boolean
  friendsIds: string[]
}

const key: DynamoDBKey<User> = {
  id: 1,
  name: 'XXXXXXXXXX',
}

```

- `populateIndexFields?: boolean` (選用)

布林值，當與一起啟用時`customPartitionKey`，會為差異同步資料表中的每個記錄建立新的項目，特別是在 `gsi_ds_pk`和 `gsi_ds_sk`欄中。

Type `ScanInput<T>`

```

ScanInput<T>: {
  consistentRead?: boolean | null;
  filter?: DynamoDBFilterObject<T> | null;
  index?: string | null;
  limit?: number | null;
  nextToken?: string | null;
  scanIndexForward?: boolean | null;
  segment?: number;
  select?: DynamoDBSelectAttributes;
  totalSegments?: number;
}

```

類型宣告

- `consistentRead?: boolean | null` (選用)

在查詢 DynamoDB 時指出一致讀取的選用布林值。預設值為 `false`。

- `filter?: DynamoDBFilterObject<T> | null` (選用)

從資料表擷取結果之後，要套用至結果的選用篩選條件。

- `index?: string | null` (選用)

要掃描之索引的選用名稱。

- `limit?: number | null` (選用)

要傳回的選用結果數目上限。

- `nextToken?: string | null` (選用)

選用分頁字符，以繼續先前的查詢。這會是從先前查詢所取得的。

- `scanIndexForward?: boolean | null` (選用)

選用布林值，指出查詢是以遞增或遞減順序執行。依預設，此值是設為 `true`。

- `segment?: number` (選用)
- `select?: DynamoDBSelectAttributes` (選用)

從 DynamoDB 傳回的屬性。根據預設，AWS AppSync DynamoDB 解析程式只會傳回投影到索引中的屬性。支援的值是：

- `ALL_ATTRIBUTES`

從指定的資料表或索引傳回所有項目屬性。如果您查詢本機次要索引，DynamoDB 會從索引中每個相符項目的父資料表擷取整個項目。如果索引設定為投射所有項目屬性，所有資料都可從本機次要索引取得，不需進行任何擷取。

- `ALL_PROJECTED_ATTRIBUTES`

傳回已投影至索引的所有屬性。如果索引設定為投射所有屬性，此傳回值相當於指定 `ALL_ATTRIBUTES`。

- `SPECIFIC_ATTRIBUTES`

僅傳回 中列出的屬性 `ProjectionExpression`。此傳回值等同於指定 `ProjectionExpression` 而不指定 的任何值 `AttributesToGet`。

- `totalSegments?: number` (選用)

Type `DynamoDBSyncInput<T>`

```
DynamoDBSyncInput<T>: {  
  basePartitionKey?: string;  
  deltaIndexName?: string;  
  filter?: DynamoDBFilterObject<T> | null;  
  lastSync?: number;  
  limit?: number | null;
```

```
nextToken?: string | null;
}
```

類型宣告

- `basePartitionKey?: string` (選用)

執行同步操作時要使用之基底資料表的分割區索引鍵。此欄位允許在資料表使用自訂分割區索引鍵時執行 Sync 操作。

- `deltaIndexName?: string` (選用)

用於同步操作的索引。當資料表使用自訂分割區索引鍵時，需要此索引才能在整個差異存放區資料表上啟用同步操作。同步操作將在 GSI 上執行 (在 `gsi_ds_pk` 和 `gsi_ds_sk` 上建立)。

- `filter?: DynamoDBFilterObject<T> | null` (選用)

從資料表擷取結果之後，要套用至結果的選用篩選條件。

- `lastSync?: number` (選用)

上次成功同步操作開始的時刻，以 epoch 毫秒為單位。如果指定此值，只會傳回 `lastSync` 之後變更的項目。只有在從初始同步操作擷取所有頁面之後，才應該填入此欄位。如果省略，則會傳回基礎資料表的結果。否則，將會傳回差異資料表的結果。

- `limit?: number | null` (選用)

要一次評估的選用項目數量上限。如果省略此值，預設限制將設為 100 個項目。此欄位的最大值為 1000 個項目。

- `nextToken?: string | null` (選用)

Type `DynamoDBUpdateInput<T>`

```
DynamoDBUpdateInput<T>: {
  _version?: number;
  condition?: DynamoDBFilterObject<T>;
  customPartitionKey?: string;
  key: DynamoDBKey<T>;
  populateIndexFields?: boolean;
  update: DynamoDBUpdateObject<T>;
}
```

類型宣告

- `_version?: number` (選用)

- `condition?: DynamoDBFilterObject<T>` (選用)

當您更新 DynamoDB 中的物件時，您可以選擇指定條件式表達式，以根據在執行操作之前已在 DynamoDB 中的物件狀態來控制請求是否成功。

- `customPartitionKey?: string` (選用)

啟用時，`customPartitionKey`值會修改啟用版本控制時差異同步資料表所使用的 `ds_sk`和 `ds_pk` 記錄格式。啟用時，也會啟用`populateIndexFields`項目的處理。

- `key: DynamoDBKey<T>` (必要)

必要參數，指定正在更新的 DynamoDB 中項目的索引鍵。DynamoDB 項目可能有單一雜湊索引鍵或雜湊和排序索引鍵。

- `populateIndexFields?: boolean` (選用)

布林值，當與一起啟用時`customPartitionKey`，會為差異同步資料表中的每個記錄建立新的項目，特別是在 `gsi_ds_pk`和 `gsi_ds_sk`欄中。

- `update: DynamoDBUpdateObject<T>`

指定要更新的屬性及其新值的物件。更新物件可與 `add`、`remove`、`replaceincrement`、`decrement`、`append`、`prepend`、搭配使用`updateListItem`。

Amazon RDS 模組函數

Amazon RDS 模組函數可在與使用 Amazon RDS Data API 設定的資料庫互動時提供增強的體驗。使用 匯入模組`@aws-appsync/utils/rds`：

```
import * as rds from '@aws-appsync/utils/rds';
```

也可以個別匯入函數。例如，以下匯入使用 `sql`：

```
import { sql } from '@aws-appsync/utils/rds';
```

函數

您可以使用 AWS AppSync RDS 模組的公用程式協助程式來與資料庫互動。

Select

select 公用程式會建立 SELECT 陳述式來查詢關聯式資料庫。

基本使用

以其基本形式，您可以指定要查詢的資料表：

```
import { select, createPgStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {

  // Generates statement:
  // "SELECT * FROM "persons"
  return createPgStatement(select({table: 'persons'}));
}
```

請注意，您也可以在此資料表識別符中指定結構描述：

```
import { select, createPgStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {

  // Generates statement:
  // SELECT * FROM "private"."persons"
  return createPgStatement(select({table: 'private.persons'}));
}
```

指定資料欄

您可以使用 columns 屬性指定資料欄。如果未將此值設定為，則預設為 *：

```
export function request(ctx) {

  // Generates statement:
  // SELECT "id", "name"
  // FROM "persons"
  return createPgStatement(select({
    table: 'persons',
    columns: ['id', 'name']
  }));
}
```

您也可以指定資料欄的資料表：

```
export function request(ctx) {  
  
  // Generates statement:  
  // SELECT "id", "persons"."name"  
  // FROM "persons"  
  return createPgStatement(select({  
    table: 'persons',  
    columns: ['id', 'persons.name']  
  }));  
}
```

限制和位移

您可以套用 `limit` 和 `offset` 至查詢：

```
export function request(ctx) {  
  
  // Generates statement:  
  // SELECT "id", "name"  
  // FROM "persons"  
  // LIMIT :limit  
  // OFFSET :offset  
  return createPgStatement(select({  
    table: 'persons',  
    columns: ['id', 'name'],  
    limit: 10,  
    offset: 40  
  }));  
}
```

排序依據

您可以使用 `orderBy` 屬性排序結果。提供指定欄和選用 `dir` 屬性的物件陣列：

```
export function request(ctx) {  
  
  // Generates statement:  
  // SELECT "id", "name" FROM "persons"  
  // ORDER BY "name", "id" DESC  
  return createPgStatement(select({  
    table: 'persons',  
    orderBy: [{  
      column: 'name',  
      dir: 'DESC'  
    }, {  
      column: 'id',  
      dir: 'ASC'  
    }]  
  }));  
}
```

```
        columns: ['id', 'name'],
        orderBy: [{column: 'name'}, {column: 'id', dir: 'DESC'}]
    }));
}
```

篩選條件

您可以使用特殊條件物件來建置篩選條件：

```
export function request(ctx) {

    // Generates statement:
    // SELECT "id", "name"
    // FROM "persons"
    // WHERE "name" = :NAME
    return createPgStatement(select({
        table: 'persons',
        columns: ['id', 'name'],
        where: {name: {eq: 'Stephane'}}
    }));
}
```

您也可以合併篩選條件：

```
export function request(ctx) {

    // Generates statement:
    // SELECT "id", "name"
    // FROM "persons"
    // WHERE "name" = :NAME and "id" > :ID
    return createPgStatement(select({
        table: 'persons',
        columns: ['id', 'name'],
        where: {name: {eq: 'Stephane'}, id: {gt: 10}}
    }));
}
```

您也可以建立OR陳述式：

```
export function request(ctx) {

    // Generates statement:
```

```
// SELECT "id", "name"
// FROM "persons"
// WHERE "name" = :NAME OR "id" > :ID
return createPgStatement(select({
  table: 'persons',
  columns: ['id', 'name'],
  where: { or: [
    { name: { eq: 'Stephane' } },
    { id: { gt: 10 } }
  ]}
}));
}
```

您也可以使用 否定條件not：

```
export function request(ctx) {

  // Generates statement:
  // SELECT "id", "name"
  // FROM "persons"
  // WHERE NOT ("name" = :NAME AND "id" > :ID)
  return createPgStatement(select({
    table: 'persons',
    columns: ['id', 'name'],
    where: { not: [
      { name: { eq: 'Stephane' } },
      { id: { gt: 10 } }
    ]}
  }));
}
```

您也可以使用下列運算子來比較值：

運算子	描述	可能的值類型
eq	Equal	number, string, boolean
ne	Not equal	number, string, boolean
le	Less than or equal	number, string
lt	Less than	number, string

ge	Greater than or equal	number, string
gt	Greater than	number, string
contains	Like	string
notContains	Not like	string
beginsWith	Starts with prefix	string
between	Between two values	number, string
attributeExists	The attribute is not null	number, string, boolean
size	checks the length of the element	string

Insert

`insert` 公用程式提供使用 `INSERT` 操作在資料庫中插入單一資料列項目的簡單方式。

單一項目插入

若要插入項目，請指定 資料表，然後傳入值的物件。物件索引鍵會映射到您的資料表資料欄。資料欄名稱會自動逸出，並使用變數映射將值傳送至資料庫：

```
import { insert, createMySQLStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {
  const { input: values } = ctx.args;
  const insertStatement = insert({ table: 'persons', values });

  // Generates statement:
  // INSERT INTO `persons`(`name`)
  // VALUES(:NAME)
  return createMySQLStatement(insertStatement)
}
```

MySQL 使用案例

您可以結合 `insert` 後接 `select` 來擷取插入的資料列：

```
import { insert, select, createMySQLStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {
  const { input: values } = ctx.args;
  const insertStatement = insert({ table: 'persons', values });
  const selectStatement = select({
    table: 'persons',
    columns: '*',
    where: { id: { eq: values.id } },
    limit: 1,
  });

  // Generates statement:
  // INSERT INTO `persons`(`name`)
  // VALUES(:NAME)
  // and
  // SELECT *
  // FROM `persons`
  // WHERE `id` = :ID
  return createMySQLStatement(insertStatement, selectStatement)
}
```

Postgres 使用案例

使用 Postgres，您可以使用從您插入的資料列[returning](#)取得資料。它接受 * 或一組資料欄名稱：

```
import { insert, createPgStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {
  const { input: values } = ctx.args;
  const insertStatement = insert({
    table: 'persons',
    values,
    returning: '*'
  });

  // Generates statement:
  // INSERT INTO "persons"("name")
  // VALUES(:NAME)
  // RETURNING *
  return createPgStatement(insertStatement)
}
```

更新

update 公用程式可讓您更新現有的資料列。您可以使用 條件物件，將變更套用至滿足條件的所有資料列中的指定資料欄。例如，假設我們有一個結構描述，允許我們進行此變動。我們希望以 Person id 的值更新 name 的，3 但前提是我們從 開始知道它們 (known_since)2000：

```
mutation Update {
  updatePerson(
    input: {id: 3, name: "Jon"},
    condition: {known_since: {ge: "2000"}}
  ) {
    id
    name
  }
}
```

我們的更新解析程式如下所示：

```
import { update, createPgStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {
  const { input: { id, ...values }, condition } = ctx.args;
  const where = {
    ...condition,
    id: { eq: id },
  };
  const updateStatement = update({
    table: 'persons',
    values,
    where,
    returning: ['id', 'name'],
  });

  // Generates statement:
  // UPDATE "persons"
  // SET "name" = :NAME, "birthday" = :BDAY, "country" = :COUNTRY
  // WHERE "id" = :ID
  // RETURNING "id", "name"
  return createPgStatement(updateStatement)
}
```

我們可以將檢查新增至條件，以確保只會更新主索引鍵id等於 3 的資料列。同樣地，對於 Postgres inserts，您可以使用 returning 來傳回修改後的資料。

Remove (移除)

remove 公用程式可讓您刪除現有的資料列。您可以在滿足條件的所有資料列上使用條件物件。請注意，delete 是 JavaScript 中的預留關鍵字。remove 應該改用：

```
import { remove, createPgStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {
  const { input: { id }, condition } = ctx.args;
  const where = { ...condition, id: { eq: id } };
  const deleteStatement = remove({
    table: 'persons',
    where,
    returning: ['id', 'name'],
  });

  // Generates statement:
  // DELETE "persons"
  // WHERE "id" = :ID
  // RETURNING "id", "name"
  return createPgStatement(deleteStatement)
}
```

轉換

在某些情況下，您可能希望在陳述式中使用有關正確物件類型的更具體性。您可以使用提供的類型提示來指定參數的類型。AWS AppSync 支援與資料 API [相同的類型提示](#)。您可以使用 AWS AppSync rds 模組的 typeHint 函數來轉換參數。

下列範例可讓您將陣列做為值傳送，該值會轉換為 JSON 物件。我們使用 -> 運算子來擷取 JSON 陣列 index2 中的元素：

```
import { sql, createPgStatement, toJsonObject, typeHint } from '@aws-appsync/utils/rds';

export function request(ctx) {
  const arr = ctx.args.list_of_ids
  const statement = sql`select ${typeHint.JSON(arr)}->2 as value`
  return createPgStatement(statement)
```

```
}

export function response(ctx) {
  return toJsonObject(ctx.result)[0][0].value
}
```

在處理和比較 DATE、和 時TIME，投射也很有用TIMESTAMP：

```
import { select, createPgStatement, typeHint } from '@aws-appsync/utils/rds';

export function request(ctx) {
  const when = ctx.args.when
  const statement = select({
    table: 'persons',
    where: { createdAt : { gt: typeHint.DATETIME(when) } }
  })
  return createPgStatement(statement)
}
```

以下另一個範例示範如何傳送目前的日期和時間：

```
import { sql, createPgStatement, typeHint } from '@aws-appsync/utils/rds';

export function request(ctx) {
  const now = util.time.nowFormatted('YYYY-MM-dd HH:mm:ss')
  return createPgStatement(sql`select ${typeHint.TIMESTAMP(now)}`)
}
```

可用的類型提示

- `typeHint.DATE` - 對應的參數會以 DATE類型的物件傳送至資料庫。接受的格式為 YYYY-MM-DD。
- `typeHint.DECIMAL` - 對應的參數會以 DECIMAL類型的物件傳送至資料庫。
- `typeHint.JSON` - 對應的參數會以 JSON類型的物件傳送至資料庫。
- `typeHint.TIME` - 對應的字串參數值會以 TIME類型的物件形式傳送至資料庫。接受的格式為 HH:MM:SS[.FFF]。
- `typeHint.TIMESTAMP` - 對應的字串參數值會以 TIMESTAMP類型的物件形式傳送至資料庫。接受的格式為 YYYY-MM-DD HH:MM:SS[.FFF]。
- `typeHint.UUID` - 對應的字串參數值會以 UUID類型的物件傳送至資料庫。

執行期公用程式

runtime 程式庫提供公用程式來控制或修改解析程式和函數的執行期屬性。

執行期 utils 清單

runtime.earlyReturn(obj?: unknown, returnOptions?: {skipTo: 'END' | 'NEXT'}): never

根據目前的內容，叫用此函數會停止目前處理常式、AWS AppSync 函數或解析程式（單位或管道解析程式）的執行。系統會傳回指定的物件做為結果。

- 在 AWS AppSync 函數請求處理常式中呼叫時，會略過資料來源和回應處理常式，並呼叫下一個函數請求處理常式（如果這是最後一個 AWS AppSync 函數，則為管道解析程式回應處理常式）。
- 在 AWS AppSync 管道解析程式請求處理常式中呼叫時，會略過管道執行，並立即呼叫管道解析程式回應處理常式。
- 將 skipTo 設為 "END" returnOptions 時，會略過管道執行，並立即呼叫管道解析程式回應處理常式。
- returnOptions 將 skipTo 設為 "NEXT" 時，會略過函數執行，並呼叫下一個管道處理常式。

範例

```
import { runtime } from '@aws-appsync/utils'

export function request(ctx) {
  runtime.earlyReturn({ hello: 'world' })
  // code below is not executed
  return ctx.args
}

// never called because request returned early
export function response(ctx) {
  return ctx.result
}
```

util.time 中的時間協助程式

util.time 變數包含日期時間方法，可協助產生時間戳記、在日期時間格式之間轉換，以及剖析日期時間字串。日期時間格式的語法是根據 [DateTimeFormatter](#)，您可以參考以取得更詳細的文件。

時間 utils 清單

util.time.nowISO8601()

以 [ISO8601 格式](#) 傳回 UTC 的字串表示方式。

util.time.nowEpochSeconds()

傳回從 1970-01-01T00:00:00Z 的 epoch 到現在的秒數。

util.time.nowEpochMilliseconds()

傳回從 1970-01-01T00:00:00Z 的 epoch 到現在的毫秒數。

util.time.nowFormatted(String)

使用字串輸入類型的指定格式，傳回 UTC 中目前時間戳記的字串。

util.time.nowFormatted(String, String)

使用字串輸入類型的指定格式和時區，傳回時區目前時間戳記的字串。

util.time.parseFormattedToEpochMilliseconds(String, String)

剖析以字串形式傳遞的時間戳記，以及包含時區的格式，然後傳回自 epoch 以來以毫秒為單位的時間戳記。

util.time.parseFormattedToEpochMilliseconds(String, String, String)

剖析以字串形式傳遞的時間戳記，以及格式和時區，然後以毫秒為單位傳回自 epoch 以來的時間戳記。

util.time.parseISO8601ToEpochMilliseconds(String)

剖析以字串形式傳遞的 ISO8601 時間戳記，然後以毫秒為單位傳回自 epoch 以來的時間戳記。

util.time.epochMillisecondsToSeconds(long)

將 epoch 毫秒時間戳記轉換為 epoch 秒時間戳記。

util.time.epochMillisecondsToISO8601(long)

將 epoch 毫秒時間戳記轉換為 ISO8601 時間戳記。

util.time.epochMillisecondsToFormatted(long, String)

將 epoch 毫秒時間戳記轉換為根據 UTC 格式提供的時間戳記。

`util.time.epochMillisecondsToFormatted(long, String, String)`

將長傳遞的 epoch 毫秒時間戳記轉換為根據所提供時區中提供的格式格式化的時間戳記。

util.dynamodb 中的 DynamoDB 協助程式

`util.dynamodb` 包含協助程式方法，可讓您更輕鬆地將資料寫入和讀取至 Amazon DynamoDB，例如自動類型映射和格式化。

`toDynamoDB`

`toDynamoDB` 公用程式清單

`util.dynamodb.toDynamoDB(Object)`

DynamoDB 的一般物件轉換工具，可將輸入物件轉換為適當的 DynamoDB 表示法。它在代表一些類型的方式上是固定的：例如，它會使用清單 (「L」) 而不是集合 (「SS」、「NS」、「BS」)。這會傳回描述 DynamoDB 屬性值的物件。

字串範例

```
Input:      util.dynamodb.toDynamoDB("foo")
Output:     { "S" : "foo" }
```

數字範例

```
Input:      util.dynamodb.toDynamoDB(12345)
Output:     { "N" : 12345 }
```

布林值範例

```
Input:      util.dynamodb.toDynamoDB(true)
Output:     { "BOOL" : true }
```

清單範例

```
Input:      util.dynamodb.toDynamoDB([ "foo", 123, { "bar" : "baz" } ])
Output:     {
        "L" : [
            { "S" : "foo" },
```



```

        { "N" : 123 },
        {
            "M" : {
                "bar" : { "S" : "baz" }
            }
        }
    ]
}

```

地圖範例

```

Input:      util.dynamodb.toDynamoDB({ "foo": "bar", "baz" : 1234, "beep":
[ "boop" ] })
Output:     {
    "M" : {
        "foo"  : { "S" : "bar" },
        "baz"  : { "N" : 1234 },
        "beep" : {
            "L" : [
                { "S" : "boop" }
            ]
        }
    }
}

```

toString utils

toString utils 清單

util.dynamodb.toString(String)

將輸入字串轉換為 DynamoDB 字串格式。這會傳回描述 DynamoDB 屬性值的物件。

```

Input:      util.dynamodb.toString("foo")
Output:     { "S" : "foo" }

```

util.dynamodb.toStringSet(List<String>)

將具有字串的清單轉換為 DynamoDB 字串集格式。這會傳回描述 DynamoDB 屬性值的物件。

```

Input:      util.dynamodb.toStringSet([ "foo", "bar", "baz" ])

```

```
Output:      { "SS" : [ "foo", "bar", "baz" ] }
```

toNumber utils

toNumber utils 清單

util.dynamodb.toNumber(Number)

將數字轉換為 DynamoDB 數字格式。這會傳回描述 DynamoDB 屬性值的物件。

```
Input:      util.dynamodb.toNumber(12345)
Output:     { "N" : 12345 }
```

util.dynamodb.toNumberSet(List<Number>)

將數字清單轉換為 DynamoDB 數字集格式。這會傳回描述 DynamoDB 屬性值的物件。

```
Input:      util.dynamodb.toNumberSet([ 1, 23, 4.56 ])
Output:     { "NS" : [ 1, 23, 4.56 ] }
```

toBinary utils

toBinary utils 清單

util.dynamodb.toBinary(String)

將編碼為 base64 字串的二進位資料轉換為 DynamoDB 二進位格式。這會傳回描述 DynamoDB 屬性值的物件。

```
Input:      util.dynamodb.toBinary("foo")
Output:     { "B" : "foo" }
```

util.dynamodb.toBinarySet(List<String>)

將編碼為 base64 字串的二進位資料清單轉換為 DynamoDB 二進位集格式。這會傳回描述 DynamoDB 屬性值的物件。

```
Input:      util.dynamodb.toBinarySet([ "foo", "bar", "baz" ])
Output:     { "BS" : [ "foo", "bar", "baz" ] }
```

toBoolean utils

toBoolean utils 清單

util.dynamodb.toBoolean(Boolean)

將布林值轉換為適當的 DynamoDB 布林值格式。這會傳回描述 DynamoDB 屬性值的物件。

```
Input:      util.dynamodb.toBoolean(true)
Output:     { "BOOL" : true }
```

toNull utils

toNull utils 清單

util.dynamodb.toNull()

以 DynamoDB null 格式傳回 null。這會傳回描述 DynamoDB 屬性值的物件。

```
Input:      util.dynamodb.toNull()
Output:     { "NULL" : null }
```

toList utils

toList utils 清單

util.dynamodb.toList(List)

將物件清單轉換為 DynamoDB 清單格式。清單中的每個項目也會轉換為適當的 DynamoDB 格式。它在代表一些巢狀物件的方式上是固定的：例如，它會使用清單 (「L」) 而不是集合 (「SS」、
「NS」、「BS」)。這會傳回描述 DynamoDB 屬性值的物件。

```
Input:      util.dynamodb.toList([ "foo", 123, { "bar" : "baz" } ])
Output:     {
      "L" : [
        { "S" : "foo" },
        { "N" : 123 },
        {
          "M" : {
            "bar" : { "S" : "baz" }
          }
        }
      ]
    }
```

```

    }
  }
]
}

```

toMap 公用程式

toMap 公用程式清單

util.dynamodb.toMap(Map)

將映射轉換為 DynamoDB 映射格式。映射中的每個值也會轉換為其適當的 DynamoDB 格式。它在代表一些巢狀物件的方式上是固定的：例如，它會使用清單 (「L」) 而不是集合 (「SS」、「NS」、「BS」)。這會傳回描述 DynamoDB 屬性值的物件。

```

Input:      util.dynamodb.toMap({ "foo": "bar", "baz" : 1234, "beep": [ "boop"] })
Output:     {
              "M" : {
                  "foo"  : { "S" : "bar" },
                  "baz"  : { "N" : 1234 },
                  "beep" : {
                      "L" : [
                          { "S" : "boop" }
                      ]
                  }
              }
            }

```

util.dynamodb.toMapValues(Map)

建立映射的副本，其中每個值都已轉換為其適當的 DynamoDB 格式。它在代表一些巢狀物件的方式上是固定的：例如，它會使用清單 (「L」) 而不是集合 (「SS」、「NS」、「BS」)。

```

Input:      util.dynamodb.toMapValues({ "foo": "bar", "baz" : 1234, "beep":
[ "boop"] })
Output:     {
              "foo"  : { "S" : "bar" },
              "baz"  : { "N" : 1234 },
              "beep" : {
                  "L" : [
                      { "S" : "boop" }
                  ]
              }
            }

```

```
    ]
  }
}
```

Note

這與 `util.dynamodb.toMap(Map)` 略有不同，因為它只會傳回 DynamoDB 屬性值的內容，但不會傳回整個屬性值本身。例如，下列陳述式完全相同：

```
util.dynamodb.toMapValues(<map>)
util.dynamodb.toMap(<map>)("M")
```

S3Object 公用程式

S3Object 公用程式清單

`util.dynamodb.toS3Object(String key, String bucket, String region)`

將金鑰、儲存貯體和區域轉換為 DynamoDB S3 物件表示法。這會傳回描述 DynamoDB 屬性值的物件。

```
Input:      util.dynamodb.toS3Object("foo", "bar", region = "baz")
Output:     { "S" : "{ \"s3\" : { \"key\" : \"foo\", \"bucket\" : \"bar\", \"region\" : \"baz\" } }" }
```

`util.dynamodb.toS3Object(String key, String bucket, String region, String version)`

將金鑰、儲存貯體、區域和選用版本轉換為 DynamoDB S3 物件表示法。這會傳回描述 DynamoDB 屬性值的物件。

```
Input:      util.dynamodb.toS3Object("foo", "bar", "baz", "beep")
Output:     { "S" : "{ \"s3\" : { \"key\" : \"foo\", \"bucket\" : \"bar\", \"region\" : \"baz\", \"version\" = \"beep\" } }" }
```

`util.dynamodb.fromS3ObjectJson(String)`

接受 DynamoDB S3 物件的字串值，並傳回包含金鑰、儲存貯體、區域和選用版本的映射。

```
Input:      util.dynamodb.fromS3ObjectJson({ "S" : "{ \"s3\" : { \"key\" : \"foo\",  
  \"bucket\" : \"bar\", \"region\" : \"baz\", \"version\" = \"beep\" } }" })  
Output:     { "key" : "foo", "bucket" : "bar", "region" : "baz", "version" :  
  "beep" }
```

util.http 中的 HTTP 協助程式

util.http 公用程式提供協助程式方法，您可以用來管理 HTTP 請求參數和新增回應標頭。

util.http utils 清單

util.http.copyHeaders(headers)

從映射複製標頭，不包括下列受限 HTTP 標頭：

- transfer-encoding
- connection
- host
- expect
- keep-alive
- upgrade
- proxy-authenticate
- proxy-authorization
- te
- content-length

util.http.addResponseHeader(String, Object)

新增具有回應名稱 (String) 和值 (Object) 的單一自訂標頭。有下列限制：

- 除了的限制標頭清單之外copyHeaders(headers)，標頭名稱也不符合下列任何項目：
 - Access-Control-Allow-Credentials
 - Access-Control-Allow-Origin
 - Access-Control-Expose-Headers
 - Access-Control-Max-Age
 - Access-Control-Allow-Methods

- Access-Control-Allow-Headers
- Vary
- Content-Type
- 標頭名稱不能以限制字首 `x-amzn-` 或 開頭 `x-amz-`。
- 自訂回應標頭的大小不能超過 4 KB。這包括標頭名稱和值。
- 您應該為每個 GraphQL 操作定義每個回應標頭一次。不過，如果您多次定義同名的自訂標頭，最近的定義會出現在回應中。無論命名為何，所有標頭都會計入標頭大小限制。
- 具有空白或限制名稱(String)或 null 值的標頭(Object)將被忽略，並產生新增到操作errors輸出的ResponseHeaderError錯誤。

```
export function request(ctx) {
  util.http.addResponseHeader('itemsCount', 7)
  util.http.addResponseHeader('render', ctx.args.render)
  return {}
}
```

util.http.addResponseHeaders(Map)

將多個回應標頭新增至指定名稱(String)和值映射 的回應(Object)。為 `addResponseHeader(String, Object)` 方法列出的相同限制也適用於此方法。

```
export function request(ctx) {
  const headers = {
    headerInt: 12,
    headerString: 'stringValue',
    headerObject: {
      field1: 7,
      field2: 'string'
    }
  }
  util.http.addResponseHeaders(headers)
  return {}
}
```

util.transform 中的轉換協助程式

`util.transform` 包含協助程式方法，可讓您更輕鬆地對資料來源執行複雜的操作。

轉換協助程式使用清單

util.transform.toDynamoDBFilterExpression(filterObject: DynamoDBFilterObject) : string

將輸入字串轉換為篩選條件表達式，以搭配 DynamoDB 使用。建議使用 toDynamoDBFilterExpression 搭配 [內建模組函數](#)。

util.transform.toElasticsearchQueryDSL(object: OpenSearchQueryObject) : string

將指定的輸入轉換為其同等的 OpenSearch Query DSL 表達式，將其傳回為 JSON 字串。

輸入範例：

```
util.transform.toElasticsearchQueryDSL({
  "upvotes":{
    "ne":15,
    "range":[
      10,
      20
    ]
  },
  "title":{
    "eq":"hihihi",
    "wildcard":"h*i"
  }
})
```

輸出範例：

```
{
  "bool":{
    "must":[
      {
        "bool":{
          "must":[
            {
              "bool":{
                "must_not":{
                  "term":{
                    "upvotes":15
                  }
                }
              }
            }
          ]
        }
      }
    ]
  }
}
```



```

    }
  },
  {
    "range":{
      "upvotes":{
        "gte":10,
        "lte":20
      }
    }
  }
]
},
{
  "bool":{
    "must":[
      {
        "term":{
          "title":"hihihi"
        }
      },
      {
        "wildcard":{
          "title":"h*i"
        }
      }
    ]
  }
}
]
}
}

```

 Note

預設運算子假設為 AND。

util.transform.toSubscriptionFilter(objFilter, ignoredFields?, rules?): SubscriptionFilter

將Map輸入物件轉換為SubscriptionFilter表達式物件。util.transform.toSubscriptionFilter 方法會用作extensions.setSubscriptionFilter()延伸模組的輸入。如需詳細資訊，請參閱 [延伸模組](#)。

Note

參數和傳回陳述式如下所示：

參數

- objFilter: SubscriptionFilterObject

轉換為SubscriptionFilter表達式物件的Map輸入物件。

- ignoredFields : SubscriptionFilterExcludeKeysType (選用)

將忽略的第一個物件中欄位名稱List的。

- rules : SubscriptionFilterRuleObject (選用)

當您建構SubscriptionFilter表達式物件時，包含嚴格規則的Map輸入物件。這些嚴格的規則將包含在SubscriptionFilter表達式物件中，以便至少滿足其中一個規則以傳遞訂閱篩選條件。

回應

傳回 [SubscriptionFilter](#)。

util.transform.toSubscriptionFilter(Map, List)

將Map輸入物件轉換為SubscriptionFilter表達式物件。util.transform.toSubscriptionFilter 方法會用作extensions.setSubscriptionFilter()延伸模組的輸入。如需詳細資訊，請參閱 [延伸模組](#)。

第一個引數是轉換為SubscriptionFilter表達式物件的Map輸入物件。第二個引數是在建構SubscriptionFilter表達式物件時在第一個Map輸入物件中忽略List的欄位名稱。

util.transform.toSubscriptionFilter(Map, List, Map)

將Map輸入物件轉換為SubscriptionFilter表達式物件。util.transform.toSubscriptionFilter 方法會用作extensions.setSubscriptionFilter()延伸模組的輸入。如需詳細資訊，請參閱 [延伸模組](#)。

util.transform.toDynamoDBConditionExpression(conditionObject)

建立 DynamoDB 條件表達式。

訂閱篩選條件引數

下表說明如何定義下列公用程式的引數：

- Util.transform.toSubscriptionFilter(objFilter, ignoredFields?, rules?): SubscriptionFilter

Argument 1: Map

引數 1 是具有下列索引鍵值的Map物件：

- 欄位名稱
- 「和」
- "或"

對於做為索引鍵的欄位名稱，這些欄位項目的條件格式為 "operator" : "value"。

下列範例顯示如何將項目新增至 Map：

```
"field_name" : {
    "operator1" : value
}

## We can have multiple conditions for the same field_name:

"field_name" : {
    "operator1" : value
    "operator2" : value
    .
    .
}
```

```
}
.
```

當欄位上有兩個或多個條件時，所有這些條件都會被視為使用 OR 操作。

輸入Map也可以將「和」和「或」做為索引鍵，表示這些輸入中的所有項目都應該使用 AND 或 OR 邏輯聯結，視索引鍵而定。索引鍵值「和」和「或」需要條件陣列。

```
"and" : [
  {
    "field_name1" : {
      "operator1" : value
    }
  },
  {
    "field_name2" : {
      "operator1" : value
    }
  },
  :
  .
].
```

請注意，您可以巢狀化「和」和「或」。也就是說，您可以在另一個「和」/「或」區塊中巢狀「和」/「或」或「或」。不過，這不適用於簡單的欄位。

```
"and" : [
  {
    "field_name1" : {
      "operator" : value
    }
  },
  {
    "or" : [
      {
        "field_name2" : {
          "operator" : value
        }
      },
      ,
    ],
  },
  ,
].
```

```
        {
          "field_name3" : {
            "operator" : value
          }
        }
      ].
```

下列範例顯示使用 輸入引數 `1util.transform.toSubscriptionFilter(Map) : Map`。

Input(s)

引數 1：映射：

```
{
  "percentageUp": {
    "lte": 50,
    "gte": 20
  },
  "and": [
    {
      "title": {
        "ne": "Book1"
      }
    },
    {
      "downvotes": {
        "gt": 2000
      }
    }
  ],
  "or": [
    {
      "author": {
        "eq": "Admin"
      }
    },
    {
      "isPublished": {
        "eq": false
      }
    }
  ]
}
```

```
}
```

輸出

結果是Map物件：

```
{
  "filterGroup": [
    {
      "filters": [
        {
          "fieldName": "percentageUp",
          "operator": "lte",
          "value": 50
        },
        {
          "fieldName": "title",
          "operator": "ne",
          "value": "Book1"
        },
        {
          "fieldName": "downvotes",
          "operator": "gt",
          "value": 2000
        },
        {
          "fieldName": "author",
          "operator": "eq",
          "value": "Admin"
        }
      ]
    },
    {
      "filters": [
        {
          "fieldName": "percentageUp",
          "operator": "lte",
          "value": 50
        },
        {
          "fieldName": "title",
          "operator": "ne",
          "value": "Book1"
        }
      ],
    }
  ]
}
```

```
{
  "fieldName": "downvotes",
  "operator": "gt",
  "value": 2000
},
{
  "fieldName": "isPublished",
  "operator": "eq",
  "value": false
}
],
},
{
  "filters": [
    {
      "fieldName": "percentageUp",
      "operator": "gte",
      "value": 20
    },
    {
      "fieldName": "title",
      "operator": "ne",
      "value": "Book1"
    },
    {
      "fieldName": "downvotes",
      "operator": "gt",
      "value": 2000
    },
    {
      "fieldName": "author",
      "operator": "eq",
      "value": "Admin"
    }
  ]
},
{
  "filters": [
    {
      "fieldName": "percentageUp",
      "operator": "gte",
      "value": 20
    },
    {
```

```

        "fieldName": "title",
        "operator": "ne",
        "value": "Book1"
      },
      {
        "fieldName": "downvotes",
        "operator": "gt",
        "value": 2000
      },
      {
        "fieldName": "isPublished",
        "operator": "eq",
        "value": false
      }
    ]
  }
]
}

```

Argument 2: List

引數 2 包含建構SubscriptionFilter表達式物件時不應在輸入中考慮List的一組欄位名稱Map (引數 1)。List 也可以是空的。

下列範例顯示使用的引數 1 和引數 2 的輸入util.transform.toSubscriptionFilter(Map, List) : Map。

Input(s)

引數 1：映射：

```

{
  "percentageUp": {
    "lte": 50,
    "gte": 20
  },
  "and": [
    {
      "title": {
        "ne": "Book1"
      }
    },
    {

```



```
      "downvotes": {
        "gt": 20
      }
    },
  ],
  "or": [
    {
      "author": {
        "eq": "Admin"
      }
    },
    {
      "isPublished": {
        "eq": false
      }
    }
  ]
}
```

引數 2：列出：

```
["percentageUp", "author"]
```

輸出

結果是Map物件：

```
{
  "filterGroup": [
    {
      "filters": [
        {
          "fieldName": "title",
          "operator": "ne",
          "value": "Book1"
        },
        {
          "fieldName": "downvotes",
          "operator": "gt",
          "value": 20
        },
        {
          "fieldName": "isPublished",
```

```

        "operator": "eq",
        "value": false
      }
    ]
  }
}

```

Argument 3: Map

引數 3 是具有欄位名稱作為索引鍵值的Map物件（不能具有「和」或「或」）。對於做為索引鍵的欄位名稱，這些欄位的條件是形式的項目"operator" : "value"。與引數 1 不同，引數 3 不能在相同的索引鍵中有多個條件。此外，引數 3 沒有「和」或「或」子句，因此也沒有涉及巢狀。

引數 3 代表新增至SubscriptionFilter表達式物件的嚴格規則清單，因此至少符合其中一個條件以傳遞篩選條件。

```

{
  "fieldname1": {
    "operator": value
  },
  "fieldname2": {
    "operator": value
  }
}
.
.
.

```

下列範例顯示使用的引數 1、引數 2 和引數 3 的輸

入util.transform.toSubscriptionFilter(Map, List, Map) : Map。

Input(s)

引數 1：映射：

```

{
  "percentageUp": {
    "lte": 50,
    "gte": 20
  },

```

```
"and": [  
  {  
    "title": {  
      "ne": "Book1"  
    }  
  },  
  {  
    "downvotes": {  
      "lt": 20  
    }  
  }  
],  
"or": [  
  {  
    "author": {  
      "eq": "Admin"  
    }  
  },  
  {  
    "isPublished": {  
      "eq": false  
    }  
  }  
]  
}
```

引數 2：列出：

```
["percentageUp", "author"]
```

引數 3：映射：

```
{  
  "upvotes": {  
    "gte": 250  
  },  
  "author": {  
    "eq": "Person1"  
  }  
}
```

輸出

結果是 Map 物件：

```
{
  "filterGroup": [
    {
      "filters": [
        {
          "fieldName": "title",
          "operator": "ne",
          "value": "Book1"
        },
        {
          "fieldName": "downvotes",
          "operator": "gt",
          "value": 20
        },
        {
          "fieldName": "isPublished",
          "operator": "eq",
          "value": false
        },
        {
          "fieldName": "upvotes",
          "operator": "gte",
          "value": 250
        }
      ]
    },
    {
      "filters": [
        {
          "fieldName": "title",
          "operator": "ne",
          "value": "Book1"
        },
        {
          "fieldName": "downvotes",
          "operator": "gt",
          "value": 20
        },
        {
          "fieldName": "isPublished",
          "operator": "eq",
          "value": false
        }
      ]
    }
  ]
}
```

```
    },  
    {  
      "fieldName": "author",  
      "operator": "eq",  
      "value": "Person1"  
    }  
  ]  
}  
]  
}
```

util.str 中的字串協助程式

util.str 包含有助於常見字串操作的方法。

util.str utils 清單

util.str.normalize(String, String)

使用四個 Unicode 標準化形式之一來標準化字串：NFC、NFD、NFKC 或 NFKD。第一個引數是要標準化的字串。第二個引數是 "nfc"、"nfd"、"nfkc" 或 "nfkd"，指定用於標準化程序的標準化類型。

延伸模組

extensions 包含一組方法，可在解析程式中執行其他動作。

快取擴充功能

extensions.evictFromApiCache(typeName: string, fieldName: string, keyValuePair: Record<string, string>) : Object

從 AWS AppSync 伺服器端快取中移出項目。第一個引數是類型名稱。第二個引數是欄位名稱。第三個引數是包含金鑰值對項目的物件，可指定快取金鑰值。您必須以與快取解析程式中的快取金鑰相同的順序，將項目放入物件中 cachingKey。如需快取的詳細資訊，請參閱[快取行為](#)。

範例 1：

此範例會移出針對已呼叫快取金鑰的解析程式快取的項目 Query.allClasses，該解析程式 context.arguments.semester 使用呼叫的快取金鑰。當呼叫變動且解析程式執行時，如果

項目已成功清除，則回應會在延伸項目物件中包含一個 `apiCacheEntriesDeleted` 值，顯示已刪除多少項目。

```
import { util, extensions } from '@aws-appsync/utils';

export const request = (ctx) => ({ payload: null });

export function response(ctx) {
  extensions.evictFromApiCache('Query', 'allClasses', {
    'context.arguments.semester': ctx.args.semester,
  });
  return null;
}
```

 Note

此函數僅適用於變動，不適用於查詢。

訂閱擴充功能

`extensions.setSubscriptionFilter(filterJsonObject)`

定義增強型訂閱篩選條件。每個訂閱通知事件會根據提供的訂閱篩選條件進行評估，並在所有篩選條件評估為 `true` 時傳送通知給用戶端。引數為 `filterJsonObject` (如需此引數的詳細資訊，請參閱以下引數：`filterJsonObject` 區段)。請參閱[增強型訂閱篩選](#)。

 Note

您只能在訂閱解析程式的回應處理常式中使用此延伸函數。此外，我們建議您使用 `util.transform.toSubscriptionFilter` 來建立您的篩選條件。

`extensions.setSubscriptionInvalidationFilter(filterJsonObject)`

定義訂閱失效篩選條件。系統會針對失效承載評估訂閱篩選條件，如果篩選條件評估為 `true`，則會使指定的訂閱失效。引數為 `filterJsonObject` (如需此引數的詳細資訊，請參閱以下引數：`filterJsonObject` 區段)。請參閱[增強型訂閱篩選](#)。

Note

您只能在訂閱解析程式的回應處理常式中使用此延伸函數。此外，我們建議您使用 `util.transform.toSubscriptionFilter` 來建立您的篩選條件。

`extensions.invalidateSubscriptions(invalidationJsonObject)`

用來從變動啟動訂閱失效。引數為 `invalidationJsonObject` (如需此引數的詳細資訊，請參閱以下引數：`invalidationJsonObject` 區段)。

Note

此延伸模組只能用於變動解析程式的回應映射範本。

在任何單一請求中，您最多只能使用五個唯一

的 `extensions.invalidateSubscriptions()` 方法呼叫。如果您超過此限制，您將會收到 GraphQL 錯誤。

引數：filterJsonObject

JSON 物件會定義訂閱或失效篩選條件。這是 中的一系列篩選條件 `filterGroup`。每個篩選條件都是個別篩選條件的集合。

```
{
  "filterGroup": [
    {
      "filters": [
        {
          "fieldName": "userId",
          "operator": "eq",
          "value": 1
        }
      ]
    },
    {
      "filters": [
        {
          "fieldName": "group",
```

```

        "operator" : "in",
        "value" : ["Admin", "Developer"]
      }
    ]
  }
}

```

每個篩選條件都有三個屬性：

- `fieldName` – GraphQL 結構描述欄位。
- `operator` – 運算子類型。
- `value` – 要與訂閱通知值比較 `fieldName` 的值。

以下是這些屬性的範例指派：

```

{
  "fieldName" : "severity",
  "operator" : "le",
  "value" : context.result.severity
}

```

引數： `invalidationJsonObject`

`invalidationJsonObject` 定義下列項目：

- `subscriptionField` – 要失效的 GraphQL 結構描述訂閱。定義為 中字串的單一訂閱 `subscriptionField` 會被視為失效。
- `payload` – 金鑰值對清單，當失效篩選條件 `true` 針對其值評估為 時，此清單會做為使訂閱失效的輸入。

當訂閱解析程式中定義的失效篩選條件 `true` 針對 `payload` 值評估為 時，下列範例會使用 `onUserDelete` 訂閱使已訂閱和已連線的用戶端失效。

```

export const request = (ctx) => ({ payload: null });

export function response(ctx) {
  extensions.invalidateSubscriptions({
    subscriptionField: 'onUserDelete',

```



```
    payload: { group: 'Developer', type: 'Full-Time' },
  });
  return ctx.result;
}
```

util.xml 中的 XML 協助程式

util.xml 包含有助於 XML 字串轉換的方法。

util.xml utils 清單

util.xml.toMap(String) : Object

將 XML 字串轉換為字典。

範例 1：

Input:

```
<?xml version="1.0" encoding="UTF-8"?>
<posts>
<post>
  <id>1</id>
  <title>Getting started with GraphQL</title>
</post>
</posts>
```

Output (object):

```
{
  "posts": {
    "post": {
      "id": 1,
      "title": "Getting started with GraphQL"
    }
  }
}
```

範例 2：

Input:

```
<?xml version="1.0" encoding="UTF-8"?>
<posts>
<post>
  <id>1</id>
  <title>Getting started with GraphQL</title>
</post>
<post>
  <id>2</id>
  <title>Getting started with AppSync</title>
</post>
</posts>
```

Output (JavaScript object):

```
{
  "posts":{
    "post":[
      {
        "id":1,
        "title":"Getting started with GraphQL"
      },
      {
        "id":2,
        "title":"Getting started with AppSync"
      }
    ]
  }
}
```

util.xml.toJsonString(String, Boolean?) : String

將 XML 字串轉換為 JSON 字串。這類似於 toMap，但輸出是字串。如果要將來自 HTTP 物件的 XML 回應直接轉換並傳回給 JSON，這非常實用。您可以設定選用的布林值參數，以判斷您是否要對 JSON 進行字串編碼。

DynamoDB 的 AWS AppSync JavaScript 解析程式函數參考

The AWS AppSync DynamoDB 函數可讓您使用 [GraphQL](#) 將傳入的 GraphQL 請求映射至 DynamoDB。DynamoDB 呼叫，然後將 DynamoDB 回應映射回 GraphQL，以將資料存放和擷取至您帳戶中現有的 Amazon DynamoDB 資料表。本節說明受支援 DynamoDB 操作的請求和回應處理常式：

- [GetItem](#) - GetItem 請求可讓您指示 DynamoDB 函數向 DynamoDB 提出 GetItem 請求，並可讓您在 DynamoDB 中指定項目的索引鍵，以及是否使用一致的讀取。
- [PutItem](#) - PutItem 請求映射文件可讓您指示 DynamoDB 函數向 DynamoDB 提出 PutItem 請求，並可讓您在 DynamoDB 中指定項目的金鑰、項目的完整內容（由金鑰和attributeValues組成），以及操作成功的條件。
- [UpdateItem](#) - UpdateItem 請求可讓您指示 DynamoDB 函數向 DynamoDB 提出 UpdateItem 請求，並允許您在 DynamoDB 中指定項目的索引鍵、描述如何在 DynamoDB 中更新項目的更新表達式，以及操作成功的條件。
- [DeleteItem](#) - DeleteItem 請求可讓您指示 DynamoDB 函數向 DynamoDB 提出 DeleteItem 請求，並可讓您在 DynamoDB 中指定項目的索引鍵，以及操作成功的條件。
- [查詢](#) - 查詢請求物件可讓您指示 DynamoDB 解析程式向 DynamoDB 提出查詢請求，並可讓您指定索引、要使用的索引、額外的篩選條件、要傳回的項目數、是否使用一致的讀取、查詢方向（向前或向後）和分頁字符。
- [掃描](#) - 掃描請求可讓您指示 DynamoDB 函數向 DynamoDB 提出掃描請求，並可讓您指定篩選條件以排除結果、要使用的索引、要傳回的項目數、是否使用一致的讀取、分頁字符和平行掃描。
- [Sync](#) - Sync 請求物件可讓您從 DynamoDB 資料表擷取所有結果，然後只接收自上次查詢後更改的資料（差異更新）。同步請求只能對版本控制的 DynamoDB 資料來源提出。您可以指定篩選條件來排除結果、要傳回的項目數、分頁字符，以及上次同步操作的啟動時間。
- [BatchGetItem](#) - BatchGetItem 請求物件可讓您告知 DynamoDB 函數向 DynamoDB 提出 BatchGetItem 請求，以擷取多個項目，可能橫跨多個資料表。對於此請求物件，您必須指定要從中擷取項目的資料表名稱，以及要從每個資料表中擷取之項目的索引鍵。
- [BatchDeleteItem](#) - BatchDeleteItem 請求物件可讓您告知 DynamoDB 函數向 DynamoDB 提出 BatchWriteItem 請求，以刪除多個項目，可能跨多個資料表。對於此請求物件，您必須指定要刪除項目的資料表名稱，以及要從每個資料表刪除的項目索引鍵。
- [BatchPutItem](#) - BatchPutItem 請求物件可讓您告知 DynamoDB 函數向 DynamoDB 提出 BatchWriteItem 請求，以放置多個項目，可能跨越多個資料表。對於此請求物件，您必須指定要放入項目的資料表名稱，以及要放入每個資料表的完整項目。
- [TransactGetItems](#) - TransactGetItems 請求物件可讓您告知 DynamoDB 函數向 DynamoDB 提出 TransactGetItems 請求，以擷取多個項目，可能跨多個資料表。對於此請求物件，您必須指定每個請求項目的資料表名稱，以便從中擷取項目，以及每個請求項目的索引鍵，以便從每個資料表擷取。
- [TransactWriteItems](#) - TransactWriteItems 請求物件可讓您指示 DynamoDB 函數向 DynamoDB 提出 TransactWriteItems 請求，以寫入多個項目，可能寫入多個資料表。對於此請求物件，您必須指定每個請求項目的目的地資料表名稱、要執行的每個請求項目的操作，以及要寫入的每個請求項目的索引鍵。

- [類型系統（請求映射）](#) - 進一步了解 DynamoDB 類型如何整合到 AWS AppSync 請求中。
- [類型系統（回應映射）](#) - 進一步了解 DynamoDB 類型如何在回應承載中自動轉換為 GraphQL 或 JSON。
- [篩選條件](#) - 進一步了解查詢和掃描操作的篩選條件。
- [條件表達式](#) - 進一步了解 PutItem、UpdateItem 和 DeleteItem 操作的條件表達式。
- [交易條件表達式](#) - 進一步了解 TransactWriteItems 操作的條件表達式。
- [投影](#) - 進一步了解如何在讀取操作中指定屬性。

GetItem

GetItem 請求可讓您告知 AWS AppSync DynamoDB 函數向 DynamoDB 提出 GetItem 請求，並可讓您指定：

- DynamoDB 中項目的索引鍵
- 是否使用一致性讀取

GetItem 請求具有下列結構：

```
type DynamoDBGetItem = {  
  operation: 'GetItem';  
  key: { [key: string]: any };  
  consistentRead?: ConsistentRead;  
  projection?: {  
    expression: string;  
    expressionNames?: { [key: string]: string };  
  };  
};
```

欄位定義如下：

GetItem 欄位

GetItem 欄位清單

operation

要執行的 DynamoDB 操作。若要執行 GetItem DynamoDB 操作，這必須設為 GetItem。此值為必填。

key

DynamoDB 中項目的索引鍵。DynamoDB 項目可能具有單一雜湊索引鍵，或雜湊索引鍵和排序索引鍵，視資料表結構而定。如需如何指定「輸入值」的詳細資訊，請參閱[類型系統（請求映射）](#)。此值為必填。

consistentRead

是否使用 DynamoDB 執行強式一致讀取。此為選用，預設值為 `false`。

projection

用於指定要從 DynamoDB 操作傳回之屬性的投影。如需投影的詳細資訊，請參閱[投影](#)。此欄位為選用欄位。

從 DynamoDB 傳回的項目會自動轉換為 GraphQL 和 JSON 基本類型，並且可在內容結果 () 中使用 `context.result`。

如需 DynamoDB 類型轉換的詳細資訊，請參閱[類型系統（回應映射）](#)。

如需 JavaScript 解析程式的詳細資訊，請參閱 [JavaScript 解析程式概觀](#)。

範例

下列範例是 GraphQL 查詢 的函數請求處理常式 `getThing(foo: String!, bar: String!)`：

```
export function request(ctx) {
  const {foo, bar} = ctx.args
  return {
    operation : "GetItem",
    key : util.dynamodb.toMapValues({foo, bar}),
    consistentRead : true
  }
}
```

如需 DynamoDB `GetItem` API 的詳細資訊，請參閱 [DynamoDB API 文件](#)。

PutItem

`PutItem` 請求映射文件可讓您告知 AWS AppSync DynamoDB 函數向 DynamoDB 提出 `PutItem` 請求，並可讓您指定下列項目：

- DynamoDB 中項目的索引鍵

- 項目 (由 key 和 attributeValues 組成) 的完整內容
- 操作成功的條件

PutItem 請求具有下列結構：

```
type DynamoDBPutItemRequest = {  
  operation: 'PutItem';  
  key: { [key: string]: any };  
  attributeValues: { [key: string]: any };  
  condition?: ConditionCheckExpression;  
  customPartitionKey?: string;  
  populateIndexFields?: boolean;  
  _version?: number;  
};
```

欄位定義如下：

PutItem 欄位

PutItem 欄位清單

operation

要執行的 DynamoDB 操作。若要執行 PutItem DynamoDB 操作，這必須設為 PutItem。此值為必填。

key

DynamoDB 中項目的索引鍵。DynamoDB 項目可能具有單一雜湊索引鍵，或雜湊索引鍵和排序索引鍵，視資料表結構而定。如需如何指定「輸入值」的詳細資訊，請參閱[類型系統（請求映射）](#)。此值為必填。

attributeValues

將放入 DynamoDB 的項目其餘屬性。如需如何指定「輸入值」的詳細資訊，請參閱[類型系統（請求映射）](#)。此欄位為選用欄位。

condition

決定要求是否成功的條件，可根據已存在於 DynamoDB 的物件狀態。如果沒有指定條件，PutItem 要求會覆寫該項目的任何現有資料項目。如需條件的詳細資訊，請參閱[條件表達式](#)。此值是選用的。

_version

代表項目之最新已知版本的數值。此值是選用的。此欄位用於衝突偵測，而且僅支援已建立版本的資料來源。

customPartitionKey

啟用時，此字串值會修改啟用版本控制時差異同步資料表使用的 `ds_sk` 和 `ds_pk` 記錄格式（如需詳細資訊，請參閱 AWS AppSync 開發人員指南中的[衝突偵測和同步](#)）。啟用時，也會啟用 `populateIndexFields` 項目的處理。此欄位為選用欄位。

populateIndexFields

布林值，當與一起 **customPartitionKey** 啟用時，會為差異同步資料表中的每個記錄建立新的項目，特別是在 `gsi_ds_pk` 和 `gsi_ds_sk` 欄中。如需詳細資訊，請參閱 AWS AppSync 開發人員指南中的[衝突偵測和同步](#)。此欄位為選用欄位。

寫入 DynamoDB 的項目會自動轉換為 GraphQL 和 JSON 基本類型，並可在內容結果 () 中使用 `context.result`。

寫入 DynamoDB 的項目會自動轉換為 GraphQL 和 JSON 基本類型，並可在內容結果 () 中使用 `context.result`。

如需 DynamoDB 類型轉換的詳細資訊，請參閱[類型系統（回應映射）](#)。

如需 JavaScript 解析程式的詳細資訊，請參閱[JavaScript 解析程式概觀](#)。

範例 1

下列範例是 GraphQL 變動 的函數請求處理常式 `updateThing(foo: String!, bar: String!, name: String!, version: Int!)`。

若無指定索引鍵的項目，則會建立該項目。若已有指定索引鍵的項目，則會覆寫該項目。

```
import { util } from '@aws-appsync/utils';
export function request(ctx) {
  const { foo, bar, ...values } = ctx.args
  return {
    operation: 'PutItem',
    key: util.dynamodb.toMapValues({foo, bar}),
    attributeValues: util.dynamodb.toMapValues(values),
  };
}
```

```
}
```

範例 2

下列範例是 GraphQL 變動的函數請求處理常式 `updateThing(foo: String!, bar: String!, name: String!, expectedVersion: Int!)`。

此範例會驗證目前在 DynamoDB 中的項目已將 `version` 欄位設定為 `expectedVersion`。

```
import { util } from '@aws-appsync/utils';
export function request(ctx) {
  const { foo, bar, name, expectedVersion } = ctx.args;
  const values = { name, version: expectedVersion + 1 };
  let condition = util.transform.toDynamoDBConditionExpression({
    version: { eq: expectedVersion },
  });

  return {
    operation: 'PutItem',
    key: util.dynamodb.toMapValues({ foo, bar }),
    attributeValues: util.dynamodb.toMapValues(values),
    condition,
  };
}
```

如需 DynamoDB PutItem API 的詳細資訊，請參閱 [DynamoDB API 文件](#)。

UpdateItem

UpdateItem 請求可讓您告知 AWS AppSync DynamoDB 函數向 DynamoDB 提出 UpdateItem 請求，並允許您指定下列項目：

- DynamoDB 中項目的索引鍵
- 描述如何在 DynamoDB 中更新項目的更新表達式
- 操作成功的條件

UpdateItem 請求具有下列結構：

```
type DynamoDBUpdateItemRequest = {
  operation: 'UpdateItem';
```



```
key: { [key: string]: any };
update: {
  expression: string;
  expressionNames?: { [key: string]: string };
  expressionValues?: { [key: string]: any };
};
condition?: ConditionCheckExpression;
customPartitionKey?: string;
populateIndexFields?: boolean;
_version?: number;
};
```

欄位定義如下：

UpdateItem 欄位

UpdateItem 欄位清單

operation

要執行的 DynamoDB 操作。若要執行 UpdateItem DynamoDB 操作，這必須設為 UpdateItem。此值為必填。

key

DynamoDB 中項目的索引鍵。DynamoDB 項目可能具有單一雜湊索引鍵，或雜湊索引鍵和排序索引鍵，視資料表結構而定。如需指定「輸入值」的詳細資訊，請參閱[類型系統（請求映射）](#)。此值為必填。

update

update 本節可讓您指定更新表達式，描述如何在 DynamoDB 中更新項目。如需有關如何撰寫更新表達式的詳細資訊，請參閱 [DynamoDB UpdateExpressions 文件](#)。此區段是必須的。

update 區段有三個元件：

expression

更新表達式。此值為必填。

expressionNames

表達式屬性 name 預留位置的替代，形式為鍵值組。索引鍵對應至 中使用的名稱預留位置 expression，且值必須是對應至 DynamoDB 中項目屬性名稱的字串。此欄位為選用的，應只能填入用於 expression 中表達式屬性名稱預留位置的替代。

expressionValues

表達式屬性 value 預留位置的替代，形式為鍵值組。鍵對應用於 expression 的值預留位置，值必須是類型值。如需如何指定「輸入值」的詳細資訊，請參閱[類型系統（請求映射）](#)。此必須指定。此欄位為選用的，應只能填入用於 expression 中表達式屬性值預留位置的替代。

condition

決定要求是否成功的條件，可根據已存在於 DynamoDB 的物件狀態。如果沒有指定條件，UpdateItem 要求會更新現有的資料項目，無論項目的目前狀態為何。如需條件的詳細資訊，請參閱[條件表達式](#)。此值是選用的。

_version

代表項目之最新已知版本的數值。此值是選用的。此欄位用於衝突偵測，而且僅支援已建立版本的資料來源。

customPartitionKey

啟用時，此字串值會修改啟用版本控制時差異同步資料表使用的 ds_sk 和 ds_pk 記錄格式（如需詳細資訊，請參閱 AWS AppSync 開發人員指南中的[衝突偵測和同步](#)）。啟用時，也會啟用 populateIndexFields 項目的處理。此欄位為選用欄位。

populateIndexFields

布林值，當與一起 customPartitionKey 啟用時，會為差異同步資料表中的每個記錄建立新的項目，特別是在 gsi_ds_pk 和 gsi_ds_sk 欄中。如需詳細資訊，請參閱 AWS AppSync 開發人員指南中的[衝突偵測和同步](#)。此欄位為選用欄位。

DynamoDB 中更新的項目會自動轉換為 GraphQL 和 JSON 基本類型，並在內容結果 (context.result) 中提供。

如需 DynamoDB 類型轉換的詳細資訊，請參閱[類型系統（回應映射）](#)。

如需 JavaScript 解析程式的詳細資訊，請參閱 [JavaScript 解析程式概觀](#)。

範例 1

下列範例是 GraphQL 變動 的函數請求處理常式 upvote(id: ID!)。

在此範例中，DynamoDB 中的項目的 upvotes 和 version 欄位增量為 1。

```
import { util } from '@aws-appsync/utils';
export function request(ctx) {
```

```

const { id } = ctx.args;
return {
  operation: 'UpdateItem',
  key: util.dynamodb.toMapValues({ id }),
  update: {
    expression: 'ADD #votefield :plusOne, version :plusOne',
    expressionNames: { '#votefield': 'upvotes' },
    expressionValues: { ':plusOne': { N: 1 } },
  },
};
}

```

範例 2

下列範例是 GraphQL 變動 的函數請求處理常式 `updateItem(id: ID!, title: String, author: String, expectedVersion: Int!)`。

這個複雜的範例會檢查引數，並持續產生更新表達式，其只包含由用戶端提供的引數。例如，如果 `title` 和 `author` 遭到省略，則不會更新。如果指定引數，但其值為 `null`，則該欄位會從 DynamoDB 中的物件中刪除。最後，操作有一個條件，可驗證目前在 DynamoDB 中的項目是否將 `version` 欄位設定為 `expectedVersion`：

```

import { util } from '@aws-appsync/utils';
export function request(ctx) {
  const { args: { input: { id, ...values } } } = ctx;

  const condition = {
    id: { attributeExists: true },
    version: { eq: values.expectedVersion },
  };
  values.expectedVersion += 1;
  return dynamodbUpdateRequest({ keys: { id }, values, condition });
}

/**
 * Helper function to update an item
 * @returns an UpdateItem request
 */
function dynamodbUpdateRequest(params) {
  const { keys, values, condition: inCondObj } = params;

  const sets = [];

```

```

const removes = [];
const expressionNames = {};
const expValues = {};

// Iterate through the keys of the values
for (const [key, value] of Object.entries(values)) {
  expressionNames[`#${key}`] = key;
  if (value) {
    sets.push(`#${key} = :${key}`);
    expValues[`: ${key}`] = value;
  } else {
    removes.push(`#${key}`);
  }
}

let expression = sets.length ? `SET ${sets.join(', ')}` : '';
expression += removes.length ? ` REMOVE ${removes.join(', ')}` : '';

const condition = JSON.parse(
  util.transform.toDynamoDBConditionExpression(inCondObj)
);

return {
  operation: 'UpdateItem',
  key: util.dynamodb.toMapValues(keys),
  condition,
  update: {
    expression,
    expressionNames,
    expressionValues: util.dynamodb.toMapValues(expValues),
  },
};
}

```

如需 DynamoDB UpdateItem API 的詳細資訊，請參閱 [DynamoDB API 文件](#)。

DeleteItem

DeleteItem 請求可讓您告知 AWS AppSync DynamoDB 函數向 DynamoDB 提出 DeleteItem 請求，並可讓您指定下列項目：

- DynamoDB 中項目的索引鍵
- 操作成功的條件

DeleteItem 請求具有下列結構：

```
type DynamoDBDeleteItemRequest = {  
  operation: 'DeleteItem';  
  key: { [key: string]: any };  
  condition?: ConditionCheckExpression;  
  customPartitionKey?: string;  
  populateIndexFields?: boolean;  
  _version?: number;  
};
```

欄位定義如下：

DeleteItem 欄位

DeleteItem 欄位清單

operation

要執行的 DynamoDB 操作。若要執行 DeleteItem DynamoDB 操作，這必須設為 DeleteItem。此值為必填。

key

DynamoDB 中項目的索引鍵。DynamoDB 項目可能有單一雜湊索引鍵，或雜湊索引鍵和排序索引鍵，視資料表結構而定。如需指定「輸入值」的詳細資訊，請參閱[類型系統（請求映射）](#)。此值為必填。

condition

決定要求是否成功的條件，可根據已存在於 DynamoDB 的物件狀態。如果沒有指定條件，DeleteItem 要求會刪除項目，無論該項目的目前狀態為何。如需條件的詳細資訊，請參閱[條件表達式](#)。此值是選用的。

_version

代表項目之最新已知版本的數值。此值是選用的。此欄位用於衝突偵測，而且僅支援已建立版本的資料來源。

customPartitionKey

啟用時，此字串值會修改啟用版本控制時差異同步資料表使用的 ds_sk 和 ds_pk 記錄格式（如需詳細資訊，請參閱 AWS AppSync 開發人員指南中的[衝突偵測和同步](#)）。啟用時，也會啟用 populateIndexFields 項目的處理。此欄位為選用欄位。

populateIndexFields

布林值，當與一起`customPartitionKey`啟用時，會為差異同步資料表中的每個記錄建立新的項目，特別是在 `gsi_ds_pk`和 `gsi_ds_sk`欄中。如需詳細資訊，請參閱 AWS AppSync 開發人員指南中的[衝突偵測和同步](#)。此欄位為選用欄位。

從 DynamoDB 刪除的項目會自動轉換為 GraphQL 和 JSON 基本類型，並可在內容結果 () 中使用 `context.result`。

如需 DynamoDB 類型轉換的詳細資訊，請參閱[類型系統 \(回應映射\)](#)。

如需 JavaScript 解析程式的詳細資訊，請參閱 [JavaScript 解析程式概觀](#)。

範例 1

下列範例是 GraphQL 變動 的函數請求處理常式 `deleteItem(id: ID!)`。如果已存在此 ID 的項目，則該項目將會遭到刪除。

```
import { util } from '@aws-appsync/utils';
export function request(ctx) {
  return {
    operation: 'DeleteItem',
    key: util.dynamodb.toMapValues({ id: ctx.args.id }),
  };
}
```

範例 2

下列範例是 GraphQL 變動 的函數請求處理常式 `deleteItem(id: ID!, expectedVersion: Int!)`。如果已存在此 ID 的項目，則該項目將會遭到刪除，但其 `version` 欄位必須已設為 `expectedVersion`：

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
  const { id, expectedVersion } = ctx.args;
  const condition = {
    id: { attributeExists: true },
    version: { eq: expectedVersion },
  };
  return {
```

```
operation: 'DeleteItem',
key: util.dynamodb.toMapValues({ id }),
condition: util.transform.toDynamoDBConditionExpression(condition),
};
}
```

如需 DynamoDB DeleteItem API 的詳細資訊，請參閱 [DynamoDB API 文件](#)。

Query

Query 請求物件可讓您指示 AWS AppSync DynamoDB 解析程式向 DynamoDB 提出 Query 請求，並可讓您指定下列項目：

- 索引鍵表達式
- 要使用哪些索引
- 任何額外篩選條件
- 要傳回多少項目
- 是否使用一致性讀取
- 查詢方向 (向前或向後)
- 分頁字符

Query 請求物件具有下列結構：

```
type DynamoDBQueryRequest = {
  operation: 'Query';
  query: {
    expression: string;
    expressionNames?: { [key: string]: string };
    expressionValues?: { [key: string]: any };
  };
  index?: string;
  nextToken?: string;
  limit?: number;
  scanIndexForward?: boolean;
  consistentRead?: boolean;
  select?: 'ALL_ATTRIBUTES' | 'ALL_PROJECTED_ATTRIBUTES' | 'SPECIFIC_ATTRIBUTES';
  filter?: {
    expression: string;
    expressionNames?: { [key: string]: string };
  };
}
```

```
expressionValues?: { [key: string]: any };
};
projection?: {
  expression: string;
  expressionNames?: { [key: string]: string };
};
};
```

欄位定義如下：

查詢欄位

查詢欄位清單

operation

要執行的 DynamoDB 操作。若要執行 Query DynamoDB 操作，這必須設為 Query。此值為必填。

query

query 本節可讓您指定金鑰條件表達式，描述要從 DynamoDB 擷取哪些項目。如需如何編寫索引鍵條件表達式的詳細資訊，請參閱 [DynamoDB KeyConditions 文件](#)。必須指定此區段。

expression

查詢表達式。必須指定此欄位。

expressionNames

表達式屬性 name 預留位置的替代，形式為鍵值組。索引鍵對應於 中使用的名稱預留位置 expression，且值必須是對應於 DynamoDB 中項目屬性名稱的字串。此欄位為選用的，應只能填入用於 expression 中表達式屬性名稱預留位置的替代。

expressionValues

表達式屬性 value 預留位置的替代，形式為鍵值組。鍵對應用於 expression 的值預留位置，值必須是類型值。如需如何指定「輸入值」的詳細資訊，請參閱[類型系統（請求映射）](#)。此值為必填。此欄位為選用的，應只能填入用於 expression 中表達式屬性值預留位置的替代。

filter

額外的篩選條件可用來在傳回 DynamoDB 的結果之前先篩選結果。如需篩選條件的詳細資訊，請參閱[篩選條件](#)。此欄位為選用欄位。

index

要查詢的索引名稱。DynamoDB 查詢操作除了雜湊索引鍵的主索引外，還可讓您掃描本機次要索引和全域次要索引。如果指定，這會通知 DynamoDB 查詢指定的索引。若省略，則會查詢主索引鍵索引。

nextToken

分頁字符將繼續先前的查詢。這會是從先前查詢所取得的。此欄位為選用欄位。

limit

要評估的項目數上限 (不一定是相符的項目數)。此欄位為選用欄位。

scanIndexForward

指出是否向前或向後查詢的布林值。此欄位為選用，預設值為 true。

consistentRead

布林值，指出在查詢 DynamoDB 時是否使用一致讀取。此欄位為選用，預設值為 false。

select

根據預設，AWS AppSync DynamoDB 解析程式只會傳回投影至索引的屬性。如果需要更多屬性，請設定這個欄位。此欄位為選用欄位。支援的值是：

ALL_ATTRIBUTES

傳回所有指定資料表或索引的項目屬性。如果您查詢本機次要索引，DynamoDB 會從索引中每個相符項目的父資料表擷取整個項目。如果索引設定為投射所有項目屬性，所有資料都可從本機次要索引取得，不需進行任何擷取。

ALL_PROJECTED_ATTRIBUTES

只在查詢索引時才允許。擷取所有已投射到索引的屬性。如果索引設定為投射所有屬性，此傳回值相當於指定 ALL_ATTRIBUTES。

SPECIFIC_ATTRIBUTES

僅傳回 中列出的屬性 `projectionexpression`。此傳回值等同於指定 `projection` 的 `expression` 而不指定 的任何值 `Select`。

projection

用於指定要從 DynamoDB 操作傳回之屬性的投影。如需投影的詳細資訊，請參閱[投影](#)。此欄位為選用欄位。

DynamoDB 的結果會自動轉換為 GraphQL 和 JSON 基本類型，並可在內容結果 () 中使用 `context.result`。

如需 DynamoDB 類型轉換的詳細資訊，請參閱 [類型系統（回應映射）](#)。

如需 JavaScript 解析程式的詳細資訊，請參閱 [JavaScript 解析程式概觀](#)。

結果的結構如下：

```
{
  items = [ ... ],
  nextToken = "a pagination token",
  scannedCount = 10
}
```

欄位定義如下：

items

包含 DynamoDB 查詢傳回項目的清單。

nextToken

如果可能有一個以上的結果，`nextToken` 會包含可用於另一個要求的分頁字符。請注意，AWS AppSync 會加密和混淆從 DynamoDB 傳回的分頁字符。這樣可確保資料表的資料不會不慎洩漏給發起人。另請注意，這些分頁字符不能用於不同的函數或解析程式。

scannedCount

套用篩選條件表達式 (若有的話) 以前符合查詢條件表達式的項目數。

範例

下列範例是 GraphQL 查詢 的函數請求處理常式 `getPosts(owner: ID!)`。

在此範例中，資料表上的全域次要索引受到查詢，以傳回指定 ID 擁有的所有文章。

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
  const { owner } = ctx.args;
  return {
    operation: 'Query',
```

```
query: {  
  expression: 'ownerId = :ownerId',  
  expressionValues: util.dynamodb.toMapValues({ ':ownerId': owner }),  
},  
index: 'owner-index',  
};  
}
```

如需 DynamoDB Query API 的詳細資訊，請參閱 [DynamoDB API 文件](#)。

Scan

Scan 請求可讓您告知 AWS AppSync DynamoDB 函數向 DynamoDB 提出 Scan 請求，並可讓您指定下列項目：

- 排除結果的篩選結果
- 要使用哪些索引
- 要傳回多少項目
- 是否使用一致性讀取
- 分頁字符
- 平行掃描

Scan 請求物件具有下列結構：

```
type DynamoDBScanRequest = {  
  operation: 'Scan';  
  index?: string;  
  limit?: number;  
  consistentRead?: boolean;  
  nextToken?: string;  
  totalSegments?: number;  
  segment?: number;  
  filter?: {  
    expression: string;  
    expressionNames?: { [key: string]: string };  
    expressionValues?: { [key: string]: any };  
  };  
  projection?: {  
    expression: string;  
    expressionNames?: { [key: string]: string };  
  };  
};
```

```
};  
};
```

欄位定義如下：

掃描欄位

掃描欄位清單

operation

要執行的 DynamoDB 操作。若要執行 Scan DynamoDB 操作，這必須設為 Scan。此值為必填。

filter

在傳回 DynamoDB 的結果之前，可用來篩選結果的篩選條件。如需篩選條件的詳細資訊，請參閱[篩選條件](#)。此欄位為選用欄位。

index

要查詢的索引名稱。DynamoDB 查詢操作除了雜湊索引鍵的主索引外，還可讓您掃描本機次要索引和全域次要索引。如果指定，這會通知 DynamoDB 查詢指定的索引。若省略，則會查詢主索引鍵索引。

limit

單次可評估的項目數量上限。此欄位為選用欄位。

consistentRead

布林值，指出在查詢 DynamoDB 時是否使用一致讀取。此欄位為選用，預設值為 false。

nextToken

分頁字符將繼續先前的查詢。這會是從先前查詢所取得的。此欄位為選用欄位。

select

根據預設，AWS AppSync DynamoDB 函數只會傳回投影到索引中的任何屬性。如果需要更多屬性，則此欄位可以設定。此欄位為選用欄位。支援的值是：

ALL_ATTRIBUTES

傳回所有指定資料表或索引的項目屬性。如果您查詢本機次要索引，DynamoDB 會從索引中每個相符項目的父資料表擷取整個項目。如果索引設定為投射所有項目屬性，所有資料都可從本機次要索引取得，不需進行任何擷取。

ALL_PROJECTED_ATTRIBUTES

只在查詢索引時才允許。擷取所有已投射到索引的屬性。如果索引設定為投射所有屬性，此傳回值相當於指定 ALL_ATTRIBUTES。

SPECIFIC_ATTRIBUTES

僅傳回 中列出的屬性 `projectionexpression`。此傳回值等同於指定 `projection` 的 `expression` 而不指定 的任何值 `Select`。

totalSegments

執行平行掃描時分割資料表的區段數。此欄位為選用的，但若指定 `segment`，則此欄位必須指定。

segment

此操作中執行平行掃描時的資料表區段。此欄位為選用的，但若指定 `totalSegments`，則此欄位必須指定。

projection

用於指定要從 DynamoDB 操作傳回之屬性的投影。如需投影的詳細資訊，請參閱[投影](#)。此欄位為選用欄位。

DynamoDB 掃描傳回的結果會自動轉換為 GraphQL 和 JSON 基本類型，並可在內容結果 () 中使用 `context.result`。

如需 DynamoDB 類型轉換的詳細資訊，請參閱[類型系統（回應映射）](#)。

如需 JavaScript 解析程式的詳細資訊，請參閱 [JavaScript 解析程式概觀](#)。

結果的結構如下：

```
{
  items = [ ... ],
  nextToken = "a pagination token",
  scannedCount = 10
}
```

欄位定義如下：

items

包含 DynamoDB 掃描傳回項目的清單。

nextToken

如果結果可能更多，會nextToken包含分頁字符，您可以在另一個請求中使用。AWS AppSync 會加密和混淆從 DynamoDB 傳回的分頁字符。這樣可確保資料表的資料不會不慎洩漏給發起人。此外，這些分頁字符無法用於不同的函數或解析程式。

scannedCount

在套用篩選條件表達式（如果有）之前，DynamoDB 擷取的項目數量。

範例 1

下列範例是 GraphQL 查詢的函數請求處理常式：allPosts。

在此範例中，資料表中的所有項目都會傳回。

```
export function request(ctx) {  
  return { operation: 'Scan' };  
}
```

範例 2

下列範例是 GraphQL 查詢的函數請求處理常式：postsMatching(title: String!)。

在此範例中，資料表中開頭為 title 引數的所有項目都會傳回。

```
export function request(ctx) {  
  const { title } = ctx.args;  
  const filter = { filter: { beginsWith: title } };  
  return {  
    operation: 'Scan',  
    filter: JSON.parse(util.transform.toDynamoDBFilterExpression(filter)),  
  };  
}
```

如需 DynamoDB Scan API 的詳細資訊，請參閱 [DynamoDB API 文件](#)。

Sync

Sync 請求物件可讓您從 DynamoDB 資料表擷取所有結果，然後只接收自上次查詢（差異更新）後更改的資料。Sync 請求只能對版本控制的 DynamoDB 資料來源提出。您可以指定下列選項：

- 排除結果的篩選結果
- 要傳回多少項目
- 分頁字符
- 上次起始 Sync 操作時

Sync 請求物件的結構如下：

```
type DynamoDBSyncRequest = {
  operation: 'Sync';
  basePartitionKey?: string;
  deltaIndexName?: string;
  limit?: number;
  nextToken?: string;
  lastSync?: number;
  filter?: {
    expression: string;
    expressionNames?: { [key: string]: string };
    expressionValues?: { [key: string]: any };
  };
};
```

欄位定義如下：

同步欄位

同步欄位清單

operation

要執行的 DynamoDB 操作。若要執行 Sync 操作，這必須設定為 Sync。此值為必填。

filter

在傳回 DynamoDB 的結果之前，可用來篩選結果的篩選條件。如需篩選條件的詳細資訊，請參閱[篩選條件](#)。此欄位為選用欄位。

limit

單次可評估的項目數量上限。此欄位為選用欄位。如果省略此值，預設限制將設為 100 個項目。此欄位的最大值為 1000 個項目。

nextToken

分頁字符將繼續先前的查詢。這會是從先前查詢所取得的。此欄位為選用欄位。

lastSync

上次成功啟動 Sync 操作的時間 (以毫秒為單位)。如果指定此值，只會傳回 lastSync 之後變更的項目。這個欄位是選用的，而且只有在初始 Sync 操作擷取所有頁面之後才能填入。如果省略此值，將傳回 Base 資料表的結果，否則會傳回 Delta 資料表的結果。

basePartitionKey

執行 Sync 操作時使用之 Base 資料表的分割區索引鍵。此欄位允許在資料表使用自訂分割區索引鍵時執行 Sync 操作。此為選用欄位。

deltaIndexName

用於 Sync 操作的索引。當資料表使用自訂分割區索引鍵時，需要此索引才能在整個差異存放區資料表上啟用 Sync 操作。Sync 操作將在 GSI 上執行 (在 gsi_ds_pk 和 上建立 gsi_ds_sk)。此欄位為選用欄位。

DynamoDB 同步傳回的結果會自動轉換為 GraphQL 和 JSON 基本類型，並可在內容結果 () 中使用 `context.result`。

如需 DynamoDB 類型轉換的詳細資訊，請參閱 [類型系統 \(回應映射\)](#)。

如需 JavaScript 解析程式的詳細資訊，請參閱 [JavaScript 解析程式概觀](#)。

結果的結構如下：

```
{
  items = [ ... ],
  nextToken = "a pagination token",
  scannedCount = 10,
  startedAt = 1550000000000
}
```

欄位定義如下：

items

包含同步傳回項目的清單。

nextToken

如果結果可能更多，會nextToken包含分頁字符，您可以在另一個請求中使用。AWS AppSync 會加密和混淆從 DynamoDB 傳回的分頁字符。這樣可確保資料表的資料不會不慎洩漏給發起人。此外，這些分頁字符無法用於不同的函數或解析程式。

scannedCount

在套用篩選條件表達式（如果有）之前，DynamoDB 擷取的項目數量。

startedAt

開始同步操作時，可以在本機存放並在另一個請求中做為 lastSync 引數的時間 (以 epoch 毫秒為單位)。如果請求中包含分頁字符，則該值將與請求針對第一頁結果傳回的值相同。

範例

下列範例是 GraphQL 查詢的函數請求處理常式：syncPosts(nextToken: String, lastSync: AWSTimestamp)。

在此範例中，如果省略 lastSync，則會傳回 Base 資料表中的所有項目。如果提供 lastSync，只會傳回自 lastSync 變更之 Delta Sync 資料表中的項目。

```
export function request(ctx) {  
  const { nextToken, lastSync } = ctx.args;  
  return { operation: 'Sync', limit: 100, nextToken, lastSync };  
}
```

BatchGetItem

BatchGetItem 請求物件可讓您告知 AWS AppSync DynamoDB 函數向 DynamoDB 提出BatchGetItem請求，以擷取多個項目，可能跨多個資料表。對於此請求物件，您必須指定下列項目：

- 將從其中擷取項目的資料表名稱
- 將從個別資料表擷取項目的索引鍵

此時會套用 DynamoDB BatchGetItem 限制，且可能不會提供任何條件表達式。

BatchGetItem 請求物件的結構如下：

```
type DynamoDBBatchGetItemRequest = {  
  operation: 'BatchGetItem';  
  tables: {  
    [tableName: string]: {  
      keys: { [key: string]: any }[];  
      consistentRead?: boolean;  
      projection?: {  
        expression: string;  
        expressionNames?: { [key: string]: string };  
      };  
    };  
  };  
};
```

欄位定義如下：

BatchGetItem 欄位

BatchGetItem 欄位清單

operation

要執行的 DynamoDB 操作。若要執行 BatchGetItem DynamoDB 操作，這必須設為 BatchGetItem。此值為必填。

tables

要從中擷取項目的 DynamoDB 資料表。此值是其中將資料表名稱指定為該映射索引鍵的映射。至少必須提供一個資料表。tables 值為必填。

keys

代表要擷取之項目主索引鍵的 DynamoDB 索引鍵清單。DynamoDB 項目可能有單一雜湊索引鍵，或雜湊索引鍵和排序索引鍵，視資料表結構而定。如需如何指定「輸入值」的詳細資訊，請參閱[類型系統（請求映射）](#)。

consistentRead

執行 GetItem 操作時是否使用一致性讀取。此值為選用值，且預設值為 false。

projection

用於指定要從 DynamoDB 操作傳回之屬性的投影。如需投影的詳細資訊，請參閱[投影](#)。此欄位為選用欄位。

注意事項：

- 如果未從資料表中擷取任何項目，該資料表的 data 區塊中會顯示 null 元素。
- 調用結果會根據在請求物件內提供的順序，依資料表排序。
- 不過，內的每個Get命令BatchGetItem都是原子的，但可以部分處理批次。如果因錯誤而部分批次處理，則未處理的索引鍵會透過 unprocessedKeys 區塊傳回為部分的呼叫結果。
- BatchGetItem 限制為 100 個索引鍵。

針對下列範例函數請求處理常式：

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
  const { authorId, postId } = ctx.args;
  return {
    operation: 'BatchGetItem',
    tables: {
      authors: [util.dynamodb.toMapValues({ authorId })],
      posts: [util.dynamodb.toMapValues({ authorId, postId })],
    },
  };
}
```

可透過 ctx.result 提供的呼叫結果如下所示：

```
{
  "data": {
    "authors": [null],
    "posts": [
      // Was retrieved
      {
        "authorId": "a1",
        "postId": "p2",
        "postTitle": "title",
        "postDescription": "description",
      }
    ]
  },
  "unprocessedKeys": {
    "authors": [
      // This item was not processed due to an error
    ]
  }
}
```

```
{
  {
    "authorId": "a1"
  }
],
"posts": []
}
}
```

`ctx.error` 包含錯誤的詳細資訊。在函數請求物件中結果中提供的索引鍵資料、`unprocessedKeys`和每個資料表索引鍵，保證會出現在調用結果中。已刪除的項目會出現在 `data` 區塊中。尚未處理的項目在該 `data` 區塊中會標示為 `null`，並置於 `unprocessedKeys` 區塊。

BatchDeleteItem

`BatchDeleteItem` 請求物件可讓您告知 AWS AppSync DynamoDB 函數向 DynamoDB 提出刪除多個項目的 `BatchWriteItem` 請求，可能跨多個資料表。對於此請求物件，您必須指定下列項目：

- 將從其中刪除項目的資料表名稱
- 將從個別資料表刪除項目的索引鍵

此時會套用 DynamoDB `BatchWriteItem` 限制，且可能不會提供任何條件表達式。

`BatchDeleteItem` 請求物件的結構如下：

```
type DynamoDBBatchDeleteItemRequest = {
  operation: 'BatchDeleteItem';
  tables: {
    [tableName: string]: { [key: string]: any }[];
  };
};
```

欄位定義如下：

BatchDeleteItem 欄位

BatchDeleteItem 欄位清單

operation

要執行的 DynamoDB 操作。若要執行 `BatchDeleteItem` DynamoDB 操作，這必須設為 `BatchDeleteItem`。此值為必填。

tables

要從中刪除項目的 DynamoDB 資料表。每個資料表都是 DynamoDB 金鑰清單，代表要刪除項目的主金鑰。DynamoDB 項目可能有單一雜湊索引鍵，或雜湊索引鍵和排序索引鍵，視資料表結構而定。如需如何指定「輸入值」的詳細資訊，請參閱[類型系統（請求映射）](#)。至少必須提供一個資料表。tables 值為必要。

注意事項：

- 不同於 DeleteItem 操作，回應中未傳回完整刪除的項目。只會傳回已傳遞的索引鍵。
- 如果資料表中未刪除任何項目，該資料表的 data 區塊中就會顯示 null 元素。
- 調用結果會根據在請求物件內提供的順序，依資料表排序。
- 內的每個Delete命令BatchDeleteItem都是原子的。但是，批次可以部分處理。如果因錯誤而部分批次處理，則未處理的索引鍵會透過 unprocessedKeys 區塊傳回為部分的呼叫結果。
- BatchDeleteItem 限制為 25 個索引鍵。
- 與衝突偵測搭配使用時，不支援此操作。同時使用兩者可能會導致錯誤。

針對下列範例函數請求處理常式：

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
  const { authorId, postId } = ctx.args;
  return {
    operation: 'BatchDeleteItem',
    tables: {
      authors: [util.dynamodb.toMapValues({ authorId })],
      posts: [util.dynamodb.toMapValues({ authorId, postId })],
    },
  };
}
```

可透過 ctx.result 提供的呼叫結果如下所示：

```
{
  "data": {
    "authors": [null],
    "posts": [
```

```

        // Was deleted
        {
            "authorId": "a1",
            "postId": "p2"
        }
    ],
},
"unprocessedKeys": {
    "authors": [
        // This key was not processed due to an error
        {
            "authorId": "a1"
        }
    ],
    "posts": []
}
}

```

`ctx.error` 包含錯誤的詳細資訊。函數請求物件中提供的索引鍵資料、`unprocessedKeys` 和每個資料表索引鍵保證會出現在調用結果中。已刪除的項目會出現在 `data` 區塊中。尚未處理的項目在該 `data` 區塊中會標示為 `null`，並置於 `unprocessedKeys` 區塊。

BatchPutItem

`BatchPutItem` 請求物件可讓您告知 AWS AppSync DynamoDB 函數向 DynamoDB 提出 `BatchWriteItem` 請求，以放置多個項目，可能跨越多個資料表。對於此請求物件，您必須指定下列項目：

- 項目將放入其中的資料表名稱
- 將放入每個資料表的所有項目

此時會套用 DynamoDB `BatchWriteItem` 限制，且可能不會提供任何條件表達式。

`BatchPutItem` 請求物件的結構如下：

```

type DynamoDBBatchPutItemRequest = {
  operation: 'BatchPutItem';
  tables: {
    [tableName: string]: { [key: string]: any }[];
  };
};

```

欄位定義如下：

BatchPutItem 欄位

BatchPutItem 欄位清單

operation

要執行的 DynamoDB 操作。若要執行 BatchPutItem DynamoDB 操作，這必須設為 BatchPutItem。此值為必填。

tables

要放入項目的 DynamoDB 資料表。每個資料表項目代表要為此特定資料表插入的 DynamoDB 項目清單。至少必須提供一個資料表。此值為必填。

注意事項：

- 若插入成功，回應會傳回完全插入的項目。
- 如果資料表中並未插入任何項目，該資料表的 data 區塊中就會顯示 null 元素。
- 插入的項目會根據在請求物件內提供的順序，依資料表排序。
- 不過，內的每個Put命令BatchPutItem都是原子的，但可以部分處理批次。如果因錯誤而部分批次處理，則未處理的索引鍵會透過 unprocessedKeys 區塊傳回為部分的呼叫結果。
- BatchPutItem 限制為 25 個項目。
- 與衝突偵測搭配使用時，不支援此操作。同時使用兩者可能會導致錯誤。

針對下列範例函數請求處理常式：

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
  const { authorId, postId, name, title } = ctx.args;
  return {
    operation: 'BatchPutItem',
    tables: {
      authors: [util.dynamodb.toMapValues({ authorId, name })],
      posts: [util.dynamodb.toMapValues({ authorId, postId, title })],
    },
  };
}
```

```
}
```

可透過 `ctx.result` 提供的呼叫結果如下所示：

```
{
  "data": {
    "authors": [
      null
    ],
    "posts": [
      // Was inserted
      {
        "authorId": "a1",
        "postId": "p2",
        "title": "title"
      }
    ]
  },
  "unprocessedItems": {
    "authors": [
      // This item was not processed due to an error
      {
        "authorId": "a1",
        "name": "a1_name"
      }
    ],
    "posts": []
  }
}
```

`ctx.error` 包含錯誤的詳細資訊。請求物件中提供的索引鍵資料、`unprocessedItems` 和每個資料表索引鍵保證會出現在調用結果中。已插入的項目會出現在 `data` 區塊中。尚未處理的項目在該 `data` 區塊中會標示為 `null`，並置於 `unprocessedItems` 區塊。

TransactGetItems

`TransactGetItems` 請求物件可讓您告知 AWS AppSync DynamoDB 函數向 DynamoDB 提出 `TransactGetItems` 請求，以擷取多個項目，可能跨多個資料表。對於此請求物件，您必須指定下列項目：

- 從中擷取項目之每個請求項目的資料表名稱
- 從每個資料表擷取之每個請求項目的索引鍵

此時會套用 DynamoDB TransactGetItems 限制，且可能不會提供任何條件表達式。

TransactGetItems 請求物件的結構如下：

```
type DynamoDBTransactGetItemsRequest = {  
  operation: 'TransactGetItems';  
  transactItems: { table: string; key: { [key: string]: any }; projection?:  
    { expression: string; expressionNames?: { [key: string]: string }; }[];  
  };  
};
```

欄位定義如下：

TransactGetItems 欄位

TransactGetItems 欄位清單

operation

要執行的 DynamoDB 操作。若要執行 TransactGetItems DynamoDB 操作，這必須設為 TransactGetItems。此值為必填。

transactItems

要包含的請求項目。此值是請求項目的陣列。必須提供至少一個請求項目。transactItems 值為必填。

table

要從中擷取項目的 DynamoDB 資料表。此值是資料表名稱的字串。table 值為必填。

key

代表要擷取之項目主索引鍵的 DynamoDB 索引鍵。DynamoDB 項目可能有單一雜湊索引鍵，或雜湊索引鍵和排序索引鍵，視資料表結構而定。如需如何指定「輸入值」的詳細資訊，請參閱[類型系統（請求映射）](#)。

projection

用於指定要從 DynamoDB 操作傳回之屬性的投影。如需投影的詳細資訊，請參閱[投影](#)。此欄位為選用欄位。

注意事項：

- 如果交易成功，items 區塊中擷取項目的順序會與請求項目的順序相同。

- 以全有或全無的方式執行交易。如果任何請求項目造成錯誤，將不執行整個交易，而且將傳回錯誤詳細資料。
- 無法擷取的請求項目不是錯誤。相反地，null 元素會出現在對應位置的項目區塊。
- 如果交易錯誤是 `TransactionCanceledException`，則會填入 `cancellationReasons` 區塊。`cancellationReasons` 區塊中取消原因的順序將與請求項目的順序相同。
- `TransactGetItems` 限制為 100 個請求項目。

針對下列範例函數請求處理常式：

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
  const { authorId, postId } = ctx.args;
  return {
    operation: 'TransactGetItems',
    transactItems: [
      {
        table: 'posts',
        key: util.dynamodb.toMapValues({ postId }),
      },
      {
        table: 'authors',
        key: util.dynamodb.toMapValues({ authorId }),
      },
    ],
  };
}
```

如果交易成功，而且僅擷取第一個請求項目，則 `ctx.result` 中的可用叫用結果如下所示：

```
{
  "items": [
    {
      // Attributes of the first requested item
      "post_id": "p1",
      "post_title": "title",
      "post_description": "description"
    },
    // Could not retrieve the second requested item
    null,
  ],
}
```

```
"cancellationReasons": null
}
```

如果因為第一個請求項目造成的 `TransactionCanceledException` 造成交易失敗，則 `ctx.result` 中的可用叫用結果如下所示：

```
{
  "items": null,
  "cancellationReasons": [
    {
      "type": "Sample error type",
      "message": "Sample error message"
    },
    {
      "type": "None",
      "message": "None"
    }
  ]
}
```

`ctx.error` 包含錯誤的詳細資訊。索引鍵 `items` 和 `cancellationReasons` 保證出現在 `ctx.result` 中。

TransactWriteItems

`TransactWriteItems` 請求物件可讓您指示 AWS AppSync DynamoDB 函數向 DynamoDB 提出 `TransactWriteItems` 請求，以寫入多個項目，可能寫入多個資料表。對於此請求物件，您必須指定下列項目：

- 每個請求項目的目標資料表名稱
- 每個請求項目要執行的操作。有四種支援的操作類型：`PutItem`、`UpdateItem`、`DeleteItem` 以及 `ConditionCheck`
- 每個要寫入之請求項目的索引鍵

套用 DynamoDB `TransactWriteItems` 限制。

`TransactWriteItems` 請求物件的結構如下：

```
type DynamoDBTransactWriteItemsRequest = {
  operation: 'TransactWriteItems';
```

```
    transactItems: TransactItem[];
  };
  type TransactItem =
    | TransactWritePutItem
    | TransactWriteUpdateItem
    | TransactWriteDeleteItem
    | TransactWriteConditionCheckItem;
  type TransactWritePutItem = {
    table: string;
    operation: 'PutItem';
    key: { [key: string]: any };
    attributeValues: { [key: string]: string };
    condition?: TransactConditionCheckExpression;
  };
  type TransactWriteUpdateItem = {
    table: string;
    operation: 'UpdateItem';
    key: { [key: string]: any };
    update: DynamoDBExpression;
    condition?: TransactConditionCheckExpression;
  };
  type TransactWriteDeleteItem = {
    table: string;
    operation: 'DeleteItem';
    key: { [key: string]: any };
    condition?: TransactConditionCheckExpression;
  };
  type TransactWriteConditionCheckItem = {
    table: string;
    operation: 'ConditionCheck';
    key: { [key: string]: any };
    condition?: TransactConditionCheckExpression;
  };
  type TransactConditionCheckExpression = {
    expression: string;
    expressionNames?: { [key: string]: string };
    expressionValues?: { [key: string]: any };
    returnValuesOnConditionCheckFailure: boolean;
  };
};
```

TransactWriteItems 欄位

TransactWriteItems 欄位清單

欄位定義如下：

operation

要執行的 DynamoDB 操作。若要執行 TransactWriteItems DynamoDB 操作，這必須設為 TransactWriteItems。此值為必填。

transactItems

要包含的請求項目。此值是請求項目的陣列。必須提供至少一個請求項目。transactItems 值為必填。

對於 PutItem，欄位定義如下：

table

目的地 DynamoDB 資料表。此值是資料表名稱的字串。table 值為必填。

operation

要執行的 DynamoDB 操作。若要執行 PutItem DynamoDB 操作，這必須設為 PutItem。此值為必填。

key

代表要放置之項目主索引鍵的 DynamoDB 索引鍵。DynamoDB 項目可能有單一雜湊索引鍵，或雜湊索引鍵和排序索引鍵，視資料表結構而定。如需如何指定「輸入值」的詳細資訊，請參閱[類型系統（請求映射）](#)。此值為必填。

attributeValues

將放入 DynamoDB 的項目其餘屬性。如需如何指定「輸入值」的詳細資訊，請參閱[類型系統（請求映射）](#)。此欄位為選用欄位。

condition

決定要求是否成功的條件，可根據已存在於 DynamoDB 的物件狀態。如果沒有指定條件，PutItem 要求會覆寫該項目的任何現有資料項目。您可以指定在條件檢查失敗時是否擷取現有項目。如需交易條件的詳細資訊，請參閱[交易條件表達式](#)。此值是選用的。

對於 UpdateItem，欄位定義如下：

table

要更新的 DynamoDB 資料表。此值是資料表名稱的字串。table 值為必填。

operation

要執行的 DynamoDB 操作。若要執行 UpdateItem DynamoDB 操作，這必須設為 UpdateItem。此值為必填。

key

代表要更新之項目主索引鍵的 DynamoDB 索引鍵。DynamoDB 項目可能有單一雜湊索引鍵，或雜湊索引鍵和排序索引鍵，視資料表結構而定。如需如何指定「輸入值」的詳細資訊，請參閱[類型系統（請求映射）](#)。此值為必填。

update

update 本節可讓您指定更新表達式，描述如何在 DynamoDB 中更新項目。如需有關如何撰寫更新表達式的詳細資訊，請參閱 [DynamoDB UpdateExpressions 文件](#)。此區段是必須的。

condition

決定要求是否成功的條件，可根據已存在於 DynamoDB 的物件狀態。如果沒有指定條件，UpdateItem 要求會更新現有的資料項目，無論項目的目前狀態為何。您可以指定在條件檢查失敗時是否擷取現有項目。如需交易條件的詳細資訊，請參閱[交易條件表達式](#)。此值是選用的。

對於 DeleteItem，欄位定義如下：

table

要在其中刪除項目的 DynamoDB 資料表。此值是資料表名稱的字串。table 值為必填。

operation

要執行的 DynamoDB 操作。若要執行 DeleteItem DynamoDB 操作，這必須設為 DeleteItem。此值為必填。

key

代表要刪除之項目主索引鍵的 DynamoDB 索引鍵。DynamoDB 項目可能有單一雜湊索引鍵，或雜湊索引鍵和排序索引鍵，視資料表結構而定。如需如何指定「輸入值」的詳細資訊，請參閱[類型系統（請求映射）](#)。此值為必填。

condition

決定要求是否成功的條件，可根據已存在於 DynamoDB 的物件狀態。如果沒有指定條件，DeleteItem 要求會刪除項目，無論該項目的目前狀態為何。您可以指定在條件檢查失敗時是否擷取現有項目。如需交易條件的詳細資訊，請參閱[交易條件表達式](#)。此值是選用的。

對於 ConditionCheck，欄位定義如下：

table

要檢查條件的 DynamoDB 資料表。此值是資料表名稱的字串。table 值為必填。

operation

要執行的 DynamoDB 操作。若要執行 ConditionCheck DynamoDB 操作，這必須設為 ConditionCheck。此值為必填。

key

代表要條件檢查之項目主索引鍵的 DynamoDB 索引鍵。DynamoDB 項目可能有單一雜湊索引鍵，或雜湊索引鍵和排序索引鍵，視資料表結構而定。如需如何指定「輸入值」的詳細資訊，請參閱[類型系統（請求映射）](#)。此值為必填。

condition

決定要求是否成功的條件，可根據已存在於 DynamoDB 的物件狀態。您可以指定在條件檢查失敗時是否擷取現有項目。如需交易條件的詳細資訊，請參閱[交易條件表達式](#)。此值為必填。

注意事項：

- 如果成功，只會在回應中傳回請求項目的索引鍵。索引鍵的順序將與請求項目的順序相同。
- 以全有或全無的方式執行交易。如果任何請求項目造成錯誤，將不執行整個交易，而且將傳回錯誤詳細資料。
- 沒有兩個請求項目可以定位到相同項目。否則，它們將造成 TransactionCanceledException 錯誤。
- 如果交易錯誤是 TransactionCanceledException，則會填入 cancellationReasons 區塊。如果請求項目的條件檢查失敗，而且您未將 returnValuesOnConditionCheckFailure 指定為 false，將擷取資料表中存在的項目，並存放在 cancellationReasons 區塊之對應位置的 item 中。
- TransactWriteItems 限制為 100 個請求項目。

- 與衝突偵測搭配使用時，不支援此操作。同時使用兩者可能會導致錯誤。

針對下列範例函數請求處理常式：

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
  const { authorId, postId, title, description, oldTitle, authorName } = ctx.args;
  return {
    operation: 'TransactWriteItems',
    transactItems: [
      {
        table: 'posts',
        operation: 'PutItem',
        key: util.dynamodb.toMapValues({ postId }),
        attributeValues: util.dynamodb.toMapValues({ title, description }),
        condition: util.transform.toDynamoDBConditionExpression({
          title: { eq: oldTitle },
        }),
      },
      {
        table: 'authors',
        operation: 'UpdateItem',
        key: util.dynamodb.toMapValues({ authorId }),
        update: {
          expression: 'SET authorName = :name',
          expressionValues: util.dynamodb.toMapValues({ ':name': authorName }),
        },
      },
    ],
  };
}
```

如果交易成功，`ctx.result` 中可用的叫用結果如下所示：

```
{
  "keys": [
    // Key of the PutItem request
    {
      "post_id": "p1",
    },
    // Key of the UpdateItem request
  ]
}
```



```
{
  "author_id": "a1"
},
"cancellationReasons": null
}
```

如果交易因為PutItem請求的條件檢查失敗而失敗，中可用的調用結果ctx.result如下：

```
{
  "keys": null,
  "cancellationReasons": [
    {
      "item": {
        "post_id": "p1",
        "post_title": "Actual old title",
        "post_description": "Old description"
      },
      "type": "ConditionCheckFailed",
      "message": "The condition check failed."
    },
    {
      "type": "None",
      "message": "None"
    }
  ]
}
```

ctx.error 包含錯誤的詳細資訊。索引鍵 keys 和 cancellationReasons 保證出現在 ctx.result 中。

類型系統（請求映射）

使用 AWS AppSync DynamoDB 函數呼叫 DynamoDB 資料表時，AWS AppSync 需要知道要用於該呼叫的每個值類型。這是因為 DynamoDB 支援比 GraphQL 或 JSON 更多的類型基本概念（例如集合和二進位資料）。AWS AppSync 在 GraphQL 和 DynamoDB 之間翻譯時需要一些提示，否則您必須對資料表中的資料結構進行一些假設。

如需 DynamoDB 資料類型的詳細資訊，請參閱 DynamoDB [資料類型描述項](#)和[資料類型](#)文件。

DynamoDB 值由包含單一鍵值對的 JSON 物件表示。金鑰指定 DynamoDB 類型，而值指定值本身。在下列範例中，S 鍵代表值是字串，identifier 值則是字串值本身。

```
{ "S" : "identifier" }
```

請注意，JSON 物件不能有一個以上的索引值對。如果指定多個鍵/值對，則不會剖析請求物件。

DynamoDB 值會在您需要指定值的請求物件中的任何位置使用。您需要執行這項操作的一些地方包括：key 和 attributeValue 區段，以及表達式區段的 expressionValues 區段。在下列範例中，DynamoDB 字串值 identifier 會指派給 key 區段中的 id 欄位（可能在 GetItem 請求物件中）。

```
"key" : {  
  "id" : { "S" : "identifier" }  
}
```

支援的類型

AWS AppSync 支援下列 DynamoDB 純量、文件和集類型：

字串類型 S

單一字串值。DynamoDB 字串值的表示方式為：

```
{ "S" : "some string" }
```

使用範例：

```
"key" : {  
  "id" : { "S" : "some string" }  
}
```

字串集類型 SS

字串值的集合。DynamoDB 字串集值的表示方式為：

```
{ "SS" : [ "first value", "second value", ... ] }
```

使用範例：

```
"attributeValues" : {  
  "phoneNumbers" : { "SS" : [ "+1 555 123 4567", "+1 555 234 5678" ] }  
}
```

數字類型 **N**

單一數值。DynamoDB 數字值的表示方式為：

```
{ "N" : 1234 }
```

使用範例：

```
"expressionValues" : {  
  ":expectedVersion" : { "N" : 1 }  
}
```

數字集類型 **NS**

數值的集合。DynamoDB 數字集值的表示方式為：

```
{ "NS" : [ 1, 2.3, 4 ... ] }
```

使用範例：

```
"attributeValues" : {  
  "sensorReadings" : { "NS" : [ 67.8, 12.2, 70 ] }  
}
```

二進位類型 **B**

二進位值。DynamoDB 二進位值的表示方式為：

```
{ "B" : "SGVsbG8sIFdvcmxkIQo=" }
```

請注意，該值實際上是字串，其中字串是二進位資料的 base64 編碼表示法。AWS AppSync 會將此字串解碼回其二進位值，然後再傳送到 DynamoDB。AWS AppSync 使用 RFC 2045 定義的 base64 解碼機制：忽略不在 base64 字母中的任何字元。

使用範例：

```
"attributeValues" : {  
  "binaryMessage" : { "B" : "SGVsbG8sIFdvcmxkIQo=" }  
}
```

二進位集類型 **BS**

二進位值的集合。DynamoDB 二進位集值的表示方式為：

```
{ "BS" : [ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ... ] }
```

請注意，該值實際上是字串，其中字串是二進位資料的 base64 編碼表示法。AWS AppSync 會將此字串解碼回其二進位值，然後再傳送到 DynamoDB。AWS AppSync 使用 RFC 2045 定義的 base64 解碼機制：忽略不在 base64 字母中的任何字元。

使用範例：

```
"attributeValues" : {  
  "binaryMessages" : { "BS" : [ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ] }  
}
```

布林類型 **BOOL**

布林值。DynamoDB 布林值的表示方式為：

```
{ "BOOL" : true }
```

請注意，只有 true 和 false 為有效值。

使用範例：

```
"attributeValues" : {  
  "orderComplete" : { "BOOL" : false }  
}
```

清單類型 **L**

任何其他支援的 DynamoDB 值清單。DynamoDB 清單值的表示方式為：

```
{ "L" : [ ... ] }
```

請注意，此值是複合值，其中清單可包含任何支援的 DynamoDB 值（包括其他清單）的零或多個。清單也可包含混合的不同類型。

使用範例：

```
{ "L" : [  
  { "S" : "A string value" },  
  { "N" : 1 },  
  { "SS" : [ "Another string value", "Even more string values!" ] }  
]
```

映射類型 **M**

代表其他支援 DynamoDB 值的金鑰值對的未排序集合。DynamoDB Map 值的表示方式為：

```
{ "M" : { ... } }
```

請注意，映射可包含零或多個索引值對。金鑰必須是字串，且值可以是任何支援的 DynamoDB 值（包括其他映射）。映射也可包含混合的不同類型。

使用範例：

```
{ "M" : {  
  "someString" : { "S" : "A string value" },  
  "someNumber" : { "N" : 1 },  
  "stringSet" : { "SS" : [ "Another string value", "Even more string  
values!" ] }  
}
```

Null 類型 **NULL**

null 值。DynamoDB Null 值的表示方式為：

```
{ "NULL" : null }
```

使用範例：

```
"attributeValues" : {  
  "phoneNumbers" : { "NULL" : null }  
}
```

如需每種類型的詳細資訊，請參閱 [DynamoDB 文件](#)。

類型系統（回應映射）

從 DynamoDB 接收回應時，AWS AppSync 會自動將其轉換為 GraphQL 和 JSON 基本類型。DynamoDB 中的每個屬性都會解碼，並在回應處理常式的內容中傳回。

例如，如果 DynamoDB 傳回下列項目：

```
{
  "id" : { "S" : "1234" },
  "name" : { "S" : "Nadia" },
  "age" : { "N" : 25 }
}
```

從管道解析程式傳回結果時，AWS AppSync 會將結果轉換為 GraphQL 和 JSON 類型，如下所示：

```
{
  "id" : "1234",
  "name" : "Nadia",
  "age" : 25
}
```

本節說明 AWS AppSync 如何轉換下列 DynamoDB 純量、文件和設定類型：

字串類型 **S**

單一字串值。DynamoDB 字串值會以字串傳回。

例如，如果 DynamoDB 傳回下列 DynamoDB 字串值：

```
{ "S" : "some string" }
```

AWS AppSync 將其轉換為字串：

```
"some string"
```

字串集類型 **SS**

字串值的集合。DynamoDB 字串集值會傳回為字串清單。

例如，如果 DynamoDB 傳回下列 DynamoDB 字串集值：

```
{ "SS" : [ "first value", "second value", ... ] }
```

AWS AppSync 將其轉換為字串清單：

```
[ "+1 555 123 4567", "+1 555 234 5678" ]
```

數字類型 **N**

單一數值。DynamoDB 數字值會以數字傳回。

例如，如果 DynamoDB 傳回下列 DynamoDB 數值：

```
{ "N" : 1234 }
```

AWS AppSync 將其轉換為數字：

```
1234
```

數字集類型 **NS**

數值的集合。DynamoDB 數字集值會以數字清單傳回。

例如，如果 DynamoDB 傳回下列 DynamoDB 數字集值：

```
{ "NS" : [ 67.8, 12.2, 70 ] }
```

AWS AppSync 將其轉換為數字清單：

```
[ 67.8, 12.2, 70 ]
```

二進位類型 **B**

二進位值。DynamoDB 二進位值會以字串的形式傳回，其中包含該值的 base64 表示法。

例如，如果 DynamoDB 傳回下列 DynamoDB 二進位值：

```
{ "B" : "SGVsbG8sIFdvcmxkIQo=" }
```

AWS AppSync 將其轉換為包含 base64 表示值的字串：

```
"SGVsbG8sIFdvcmxkIQo="
```

請注意，二進位資料以 base64 編碼配置編碼，如 [RFC 4648](#) 和 [RFC 2045](#) 中所定義。

二進位集類型 **BS**

二進位值的集合。DynamoDB 二進位集值會以包含 base64 值表示的字串清單傳回。

例如，如果 DynamoDB 傳回下列 DynamoDB 二進位集值：

```
{ "BS" : [ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ... ] }
```

AWS AppSync 將其轉換為包含 base64 值表示法的字串清單：

```
[ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ... ]
```

請注意，二進位資料以 base64 編碼配置編碼，如 [RFC 4648](#) 和 [RFC 2045](#) 中所定義。

布林類型 **BOOL**

布林值。DynamoDB 布林值會以布林值傳回。

例如，如果 DynamoDB 傳回下列 DynamoDB 布林值：

```
{ "BOOL" : true }
```

AWS AppSync 將其轉換為布林值：

```
true
```

清單類型 **L**

任何其他支援的 DynamoDB 值清單。DynamoDB 清單值會傳回為值清單，其中每個內部值也會轉換。

例如，如果 DynamoDB 傳回下列 DynamoDB List 值：

```
{ "L" : [
```



```
{ "S" : "A string value" },
{ "N" : 1 },
{ "SS" : [ "Another string value", "Even more string values!" ] }
}
```

AWS AppSync 將其轉換為轉換值的清單：

```
[ "A string value", 1, [ "Another string value", "Even more string values!" ] ]
```

映射類型 **M**

任何其他支援的 DynamoDB 值的金鑰/值集合。DynamoDB Map 值會以 JSON 物件傳回，其中每個索引鍵/值也會轉換。

例如，如果 DynamoDB 傳回下列 DynamoDB Map 值：

```
{ "M" : {
  "someString" : { "S" : "A string value" },
  "someNumber" : { "N" : 1 },
  "stringSet" : { "SS" : [ "Another string value", "Even more string
values!" ] }
}
```

AWS AppSync 將其轉換為 JSON 物件：

```
{
  "someString" : "A string value",
  "someNumber" : 1,
  "stringSet" : [ "Another string value", "Even more string values!" ]
}
```

Null 類型 **NULL**

null 值。

例如，如果 DynamoDB 傳回下列 DynamoDB Null 值：

```
{ "NULL" : null }
```

AWS AppSync 將其轉換為 null：

```
null
```

篩選條件

使用 Query 和 Scan 操作查詢 DynamoDB 中的物件時，您可以選擇指定 filter，以評估結果並僅傳回所需的值。

Query 或 Scan 請求的篩選條件屬性具有下列結構：

```
type DynamoDBExpression = {  
  expression: string;  
  expressionNames?: { [key: string]: string };  
  expressionValues?: { [key: string]: any };  
};
```

欄位定義如下：

expression

查詢表達式。如需如何編寫篩選條件表達式的詳細資訊，請參閱 [DynamoDB QueryFilter](#) 和 [DynamoDB ScanFilter](#) 文件。必須指定此欄位。

expressionNames

表達式屬性 name 預留位置的替代，形式為鍵值組。索引鍵對應至用於 expression 的名稱預留位置。值必須是對應於 DynamoDB 中項目屬性名稱的字串。此欄位為選用的，應只能填入用於 expression 中表達式屬性名稱預留位置的替代。

expressionValues

表達式屬性 value 預留位置的替代，形式為鍵值組。鍵對應用於 expression 的值預留位置，值必須是類型值。如需如何指定「輸入值」的詳細資訊，請參閱 [類型系統（請求映射）](#)。此必須指定。此欄位為選用的，應只能填入用於 expression 中表達式屬性值預留位置的替代。

範例

下列範例是請求的篩選條件區段，只有在標題以 title 引數開頭時，才會傳回從 DynamoDB 擷取的項目。

在這裡，我們使用從物件 `util.transform.toDynamoDBFilterExpression` 自動建立篩選條件：

```
const filter = util.transform.toDynamoDBFilterExpression({
  title: { beginsWith: 'far away' },
});

const request = {};
request.filter = JSON.parse(filter);
```

這會產生下列篩選條件：

```
{
  "filter": {
    "expression": "(begins_with(#title,:title_beginsWith))",
    "expressionNames": { "#title": "title" },
    "expressionValues": {
      ":title_beginsWith": { "S": "far away" }
    }
  }
}
```

條件表達式

當您使用 `PutItem`、`UpdateItem` 和 `DynamoDB` 操作來變更 `DeleteItem` `DynamoDB` 中的物件時，您可以選擇性地指定條件表達式，根據在執行操作之前已在 `DynamoDB` 中的物件狀態來控制請求是否成功。

The AWS AppSync `DynamoDB` 函數允許在 `PutItem`、`UpdateItem` 和 `DeleteItem` 請求物件中指定條件表達式，以及如果條件失敗且未更新物件時要遵循的策略。

範例 1

下列 `PutItem` 請求物件沒有條件表達式。因此，即使具有相同索引鍵的項目已存在，也會將項目放入 `DynamoDB`，藉此覆寫現有項目。

```
import { util } from '@aws-appsync/utils';
export function request(ctx) {
  const { foo, bar, ...values } = ctx.args
  return {
```

```
operation: 'PutItem',
key: util.dynamodb.toMapValues({foo, bar}),
attributeValues: util.dynamodb.toMapValues(values),
};
}
```

範例 2

下列PutItem物件具有條件表達式，只有在 DynamoDB 中不存在具有相同索引鍵的項目時，才會允許操作成功。

```
import { util } from '@aws-appsync/utils';
export function request(ctx) {
  const { foo, bar, ...values } = ctx.args
  return {
    operation: 'PutItem',
    key: util.dynamodb.toMapValues({foo, bar}),
    attributeValues: util.dynamodb.toMapValues(values),
    condition: { expression: "attribute_not_exists(id)" }
  };
}
```

根據預設，如果條件檢查失敗，AWS AppSync DynamoDB 函數會提供變動的錯誤。

不過，AWS AppSync DynamoDB 函數提供一些額外的功能，以協助開發人員處理一些常見的邊緣案例：

- If AWS AppSync DynamoDB 函數可以判斷 DynamoDB 中的目前值是否符合所需的結果，無論如何都會將操作視為成功。
- 您可以設定 函數來叫用自訂 Lambda 函數，以決定 AWS AppSync DynamoDB 函數應如何處理失敗，而不是傳回錯誤。

這些會在[處理條件檢查失敗](#)一節中詳細說明。

如需 DynamoDB 條件表達式的詳細資訊，請參閱 [DynamoDB ConditionExpressions 文件](#)。

指定條件

PutItem、UpdateItem和 DeleteItem請求物件都允許指定選用condition區段。若省略，則不會有條件檢查。若指定，條件必須為 true，操作才會成功。

condition 區段的結構如下：

```
type ConditionCheckExpression = {  
  expression: string;  
  expressionNames?: { [key: string]: string };  
  expressionValues?: { [key: string]: any };  
  equalsIgnore?: string[];  
  consistentRead?: boolean;  
  conditionalCheckFailedHandler?: {  
    strategy: 'Custom' | 'Reject';  
    lambdaArn?: string;  
  };  
};
```

下列欄位指定條件：

expression

更新表達式本身。如需如何編寫條件表達式的詳細資訊，請參閱 [DynamoDB ConditionExpressions 文件](#)。必須指定此欄位。

expressionNames

表達式屬性名稱預留位置的替代，形式為索引鍵值對。索引鍵對應於表達式中使用的名稱預留位置，且值必須是對應於 DynamoDB 中項目屬性名稱的字串。此欄位為選用的，應只能填入於表達式中所用表達式屬性名稱預留位置的替代。

expressionValues

表達式屬性值預留位置的替代，形式為索引值對。鍵對應用於表達式的值預留位置，值必須是類型值。如需如何指定「輸入值」的詳細資訊，請參閱 [類型系統（請求映射）](#)。此必須指定。此欄位為選用的，應只能填入用於表達式中表達式屬性值預留位置的替代。

其餘欄位會告知 AWS AppSync DynamoDB 函數如何處理條件檢查失敗：

equalsIgnore

當條件檢查在使用 PutItem 操作時失敗時，AWS AppSync DynamoDB 函數會將目前 DynamoDB 中的項目與嘗試寫入的項目進行比較。如果兩者相同，則操作視為成功。您可以使用 equalsIgnore 欄位來指定執行該比較時 AWS AppSync 應忽略的屬性清單。例如，如果唯一的差異是 version 屬性，它會將操作視為成功。此欄位為選用欄位。

consistentRead

當條件檢查失敗時，AWS AppSync 會使用強式一致讀取，從 DynamoDB 取得項目的目前值。您可以使用此欄位來指示 AWS AppSync DynamoDB 函數改用最終一致讀取。此欄位為選用，預設值為 `true`。

conditionalCheckFailedHandler

本節可讓您指定 AWS AppSync DynamoDB 函數在將 DynamoDB 中的目前值與預期結果進行比較後，如何處理條件檢查失敗。此區段為選用。若省略，則會預設為 `Reject` 策略。

strategy

AWS AppSync DynamoDB 函數在將 DynamoDB 中的目前值與預期結果進行比較後所採用的策略。此欄位為必填，且採用下列可能值：

Reject

變動會失敗，並會將錯誤新增至 GraphQL 回應。

Custom

The AWS AppSync DynamoDB 函數會叫用自訂 Lambda 函數，以決定如何處理條件檢查失敗。當 `strategy` 設定為 `Custom`，`lambdaArn` 欄位必須包含要叫用 Lambda 函數的 ARN。

lambdaArn

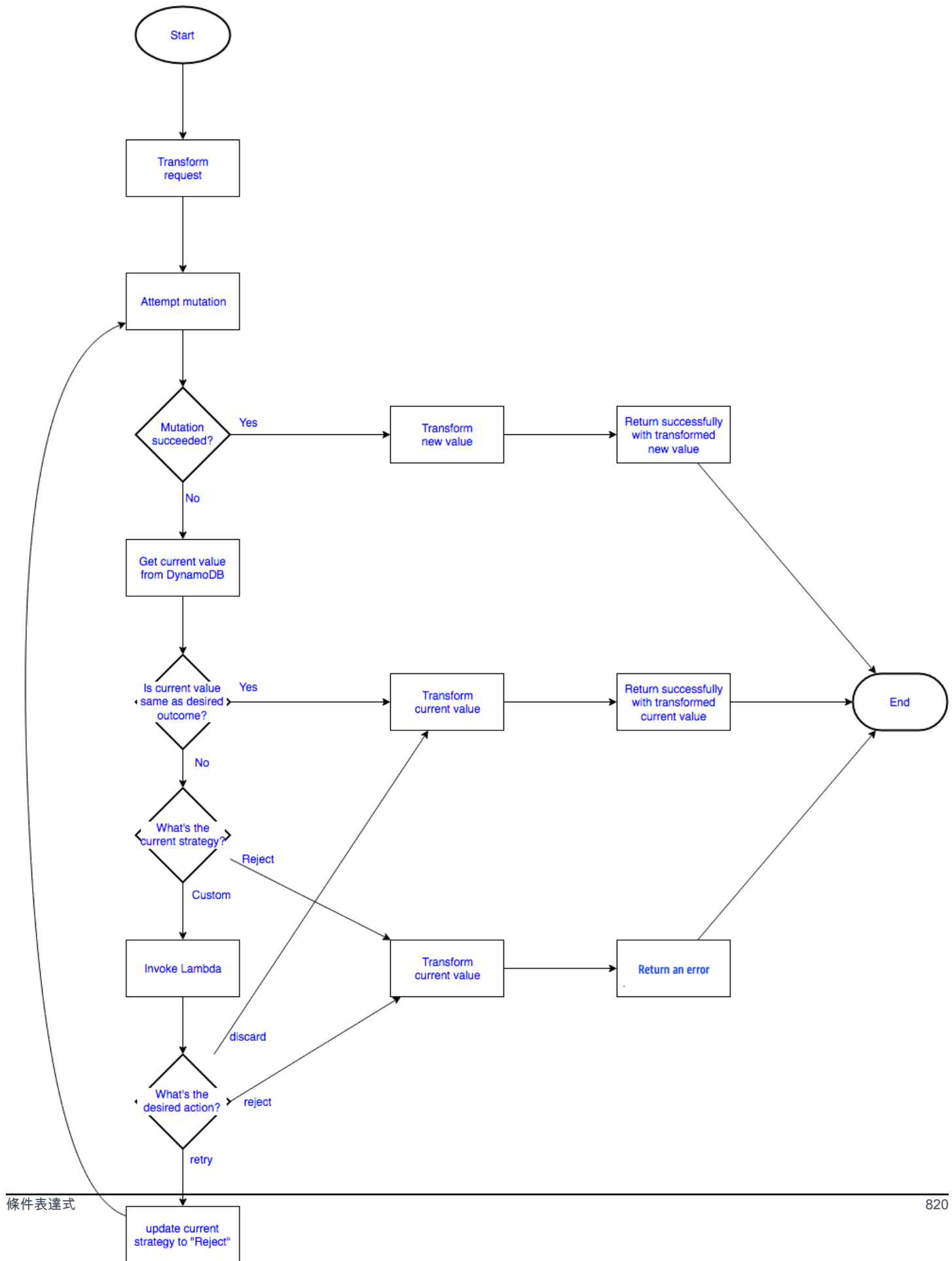
要叫用的 Lambda 函數 ARN，決定 AWS AppSync DynamoDB 函數應如何處理條件檢查失敗。只有在 `strategy` 設定為 `Custom` 時，此欄位才必須指定。如需如何使用此功能的詳細資訊，請參閱[處理條件檢查失敗](#)。

處理條件檢查失敗

當條件檢查失敗時，AWS AppSync DynamoDB 函數可以使用 `util.appendError` 公用程式傳遞變動的錯誤和物件的目前值。不過，AWS AppSync DynamoDB 函數提供一些額外的功能，以協助開發人員處理一些常見的邊緣案例：

- If AWS AppSync DynamoDB 函數可以判斷 DynamoDB 中的目前值是否符合所需的結果，無論如何都會將操作視為成功。
- 您可以設定 函數來叫用自訂 Lambda 函數，以決定 AWS AppSync DynamoDB 函數應如何處理失敗，而不是傳回錯誤。

此程序的流程圖為：



檢查所需結果

當條件檢查失敗時，AWS AppSync DynamoDB 函數會執行 `GetItem` DynamoDB 請求，從 DynamoDB 取得項目的目前值。在預設情況下，它會使用強式一致性讀取，但這可使用 `condition` 區塊中的 `consistentRead` 欄位來設定，並和預期結果進行比較：

- 對於 `PutItem` 操作，AWS AppSync DynamoDB 函數會將目前的值與其嘗試寫入的值進行比較，但不包括比較 `equalsIgnore` 中列出的任何屬性。如果項目相同，它會將操作視為成功，並傳回從 DynamoDB 擷取的項目。否則，其將按照設定的策略。

例如，如果 `PutItem` 請求物件看起來如下：

```
import { util } from '@aws-appsync/utils';
export function request(ctx) {
  const { id, name, version } = ctx.args
  return {
    operation: 'PutItem',
    key: util.dynamodb.toMapValues({foo, bar}),
    attributeValues: util.dynamodb.toMapValues({ name, version: version+1 }),
    condition: {
      expression: "version = :expectedVersion",
      expressionValues: util.dynamodb.toMapValues({':expectedVersion': version}),
      equalsIgnore: ['version']
    }
  };
}
```

而目前在 DynamoDB 中的項目外觀如下：

```
{
  "id" : { "S" : "1" },
  "name" : { "S" : "Steve" },
  "version" : { "N" : 8 }
}
```

The AWS AppSync DynamoDB 函數會將嘗試寫入的項目與目前值進行比較，查看唯一的差異是 `version` 欄位，但因為它設定為忽略 `version` 欄位，它會將操作視為成功，並傳回從 DynamoDB 擷取的項目。

- 對於 `DeleteItem` 操作，AWS AppSync DynamoDB 函數會檢查以確認項目已從 DynamoDB 傳回。如果沒有項目傳回，則操作視為成功。否則，其將按照設定的策略。

- 對於 UpdateItem 操作，AWS AppSync DynamoDB 函數沒有足夠的資訊來判斷目前 DynamoDB 中的項目是否符合預期結果，因此遵循設定的策略。

如果 DynamoDB 中物件的目前狀態與預期結果不同，AWS AppSync DynamoDB 函數會遵循設定的策略，以拒絕變動或叫用 Lambda 函數來決定接下來要做什麼。

遵循「拒絕」策略

遵循 Reject 策略時，AWS AppSync DynamoDB 函數會傳回變動的錯誤。

例如，假設變動要求如下：

```
mutation {
  updatePerson(id: 1, name: "Steve", expectedVersion: 1) {
    Name
    theVersion
  }
}
```

如果 DynamoDB 傳回的項目如下所示：

```
{
  "id" : { "S" : "1" },
  "name" : { "S" : "Steve" },
  "version" : { "N" : 8 }
}
```

而函數回應處理常式看起來如下：

```
import { util } from '@aws-appsync/utils';
export function response(ctx) {
  const { version, ...values } = ctx.result;
  const result = { ...values, theVersion: version };
  if (ctx.error) {
    if (error) {
      return util.appendError(error.message, error.type, result, null);
    }
  }
  return result
}
```

GraphQL 回應的外觀如下所示：

```
{
  "data": null,
  "errors": [
    {
      "message": "The conditional request failed (Service: AmazonDynamoDBv2;
Status Code: 400; Error Code: ConditionalCheckFailedException; Request ID:
ABCDEFGHJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZ)"
      "errorType": "DynamoDB:ConditionalCheckFailedException",
      ...
    }
  ]
}
```

此外，如果傳回物件中的任何欄位皆由其他解析程式填入，且變動成功，則在 error 區段中傳回物件時，不會解析這些物件。

遵循「自訂」策略

遵循Custom策略時，AWS AppSync DynamoDB 函數會叫用 Lambda 函數來決定接下來要做什么。Lambda 函數會選擇下列其中一個選項：

- **reject 變動。**這會通知 AWS AppSync DynamoDB 函數表現如同設定的策略為 Reject，傳回 DynamoDB 中物件變動和目前值的錯誤，如上一節所述。
- **discard 變動。**這會通知 AWS AppSync DynamoDB 函數以無提示方式忽略條件檢查失敗，並在 DynamoDB 中傳回 值。
- **retry 變動。**這會通知 AWS AppSync DynamoDB 函數使用新的請求物件重試變動。

Lambda 叫用要求

The AWS AppSync DynamoDB 函數會叫用 中指定的 Lambda 函數lambdaArn。它會使用資料來源上所設定的相同 service-role-arn。叫用承載的結構如下：

```
{
  "arguments": { ... },
  "requestMapping": {... },
  "currentValue": { ... },
  "resolver": { ... },
  "identity": { ... }
}
```

欄位定義如下：

arguments

來自 GraphQL 變動的引數。這與 中請求物件可用的引數相同 `context.arguments`。

requestMapping

此操作的請求物件。

currentValue

DynamoDB 中物件的目前值。

resolver

有關 AWS AppSync 解析程式或函數的資訊。

identity

發起人的相關資訊。這與 中請求物件可用的身分資訊相同 `context.identity`。

承載的完整範例：

```
{
  "arguments": {
    "id": "1",
    "name": "Steve",
    "expectedVersion": 1
  },
  "requestMapping": {
    "version" : "2017-02-28",
    "operation" : "PutItem",
    "key" : {
      "id" : { "S" : "1" }
    },
    "attributeValues" : {
      "name" : { "S" : "Steve" },
      "version" : { "N" : 2 }
    },
    "condition" : {
      "expression" : "version = :expectedVersion",
      "expressionValues" : {
        ":expectedVersion" : { "N" : 1 }
      }
    },
  },
}
```

```

    "equalsIgnore": [ "version" ]
  },
  "currentValue": {
    "id" : { "S" : "1" },
    "name" : { "S" : "Steve" },
    "version" : { "N" : 8 }
  },
  "resolver": {
    "tableName": "People",
    "awsRegion": "us-west-2",
    "parentType": "Mutation",
    "field": "updatePerson",
    "outputType": "Person"
  },
  "identity": {
    "accountId": "123456789012",
    "sourceIp": "x.x.x.x",
    "user": "AIDAAAAAAAAAAAAAAAAAAAA",
    "userArn": "arn:aws:iam::123456789012:user/appsync"
  }
}

```

Lambda 叫用回應

Lambda 函數可以檢查調用承載並套用任何商業邏輯，以決定 AWS AppSync DynamoDB 函數應如何處理失敗。處理條件檢查失敗有三個選項：

- **reject 變動。**此選項的回應承載必須具有此架構：

```

{
  "action": "reject"
}

```

這會通知 AWS AppSync DynamoDB 函數表現為已設定的策略為 **Reject**，傳回 DynamoDB 中物件變動和目前值的錯誤，如上節所述。

- **discard 變動。**此選項的回應承載必須具有此架構：

```

{
  "action": "discard"
}

```

這可讓 AWS AppSync DynamoDB 函數以無提示方式忽略條件檢查失敗，並在 DynamoDB 中傳回值。

- **retry 變動。**此選項的回應承載必須具有此架構：

```
{
  "action": "retry",
  "retryMapping": { ... }
}
```

這會通知 AWS AppSync DynamoDB 函數使用新的請求物件重試變動。retryMapping 區段的結構取決於 DynamoDB 操作，並且是該操作完整請求物件的子集。

若是 PutItem，retryMapping 區段的結構如下。如需 attributeValues 欄位的描述，請參閱 [PutItem](#)。

```
{
  "attributeValues": { ... },
  "condition": {
    "equalsIgnore" = [ ... ],
    "consistentRead" = true
  }
}
```

若是 UpdateItem，retryMapping 區段的結構如下。如需 update 區段的描述，請參閱 [UpdateItem](#)。

```
{
  "update" : {
    "expression" : "someExpression"
    "expressionNames" : {
      "#foo" : "foo"
    },
    "expressionValues" : {
      ":bar" : ... typed value
    }
  },
  "condition": {
    "consistentRead" = true
  }
}
```

```
}

```

若是 DeleteItem , retryMapping 區段的結構如下。

```
{
  "condition": {
    "consistentRead" = true
  }
}
```

無法指定使用不同的操作或索引鍵。AWS AppSync DynamoDB 函數僅允許對相同物件重試相同的操作。此外，condition 區段不允許指定 conditionalCheckFailedHandler。如果重試失敗，AWS AppSync DynamoDB 函數會遵循 Reject 策略。

此為 Lambda 函數處理失敗 PutItem 要求的範例。商業邏輯的重點是發起人為何。如果是由 提出 jeffTheAdmin，則會重試請求，expectedVersion 從目前在 DynamoDB 中的項目更新 version 和。否則，它會拒絕變動。

```
exports.handler = (event, context, callback) => {
  console.log("Event: " + JSON.stringify(event));

  // Business logic goes here.

  var response;
  if ( event.identity.user == "jeffTheAdmin" ) {
    response = {
      "action" : "retry",
      "retryMapping" : {
        "attributeValues" : event.requestMapping.attributeValues,
        "condition" : {
          "expression" : event.requestMapping.condition.expression,
          "expressionValues" :
event.requestMapping.condition.expressionValues
        }
      }
    }
    response.retryMapping.attributeValues.version = { "N" :
event.currentValue.version.N + 1 }
    response.retryMapping.condition.expressionValues[':expectedVersion'] =
event.currentValue.version
  }
}
```

```
    } else {  
      response = { "action" : "reject" }  
    }  
  
    console.log("Response: " + JSON.stringify(response))  
    callback(null, response)  
  }  
};
```

交易條件表達式

交易條件表達式可在 中所有四種操作類型的請求中使用 `TransactWriteItems`，即 `PutItem`、`DeleteItem`、`UpdateItem` 和 `ConditionCheck`。

對於 `PutItem`、`DeleteItem` 和 `UpdateItem`，交易條件表達式是選用的。對於 `ConditionCheck`，交易條件表達式是必要的。

範例 1

下列交易 `DeleteItem` 函數請求處理常式沒有條件表達式。因此，它會刪除 DynamoDB 中的項目。

```
import { util } from '@aws-appsync/utils';  
  
export function request(ctx) {  
  const { postId } = ctx.args;  
  return {  
    operation: 'TransactWriteItems',  
    transactItems: [  
      {  
        table: 'posts',  
        operation: 'DeleteItem',  
        key: util.dynamodb.toMapValues({ postId }),  
      }  
    ],  
  };  
}
```

範例 2

下列交易 `DeleteItem` 函數請求處理常式確實具有交易條件表達式，只有在該文章的作者等於特定名稱時，才會允許操作成功。

```
import { util } from '@aws-appsync/utils';
```



```
export function request(ctx) {
  const { postId, authorName } = ctx.args;
  return {
    operation: 'TransactWriteItems',
    transactItems: [
      {
        table: 'posts',
        operation: 'DeleteItem',
        key: util.dynamodb.toMapValues({ postId }),
        condition: util.transform.toDynamoDBConditionExpression({
          authorName: { eq: authorName },
        }),
      },
    ],
  };
}
```

如果條件檢查失敗，會導致 `TransactionCanceledException`，並且會在 `ctx.result.cancellationReasons` 中傳回錯誤詳細資料。請注意，根據預設，在 DynamoDB 中造成條件檢查失敗的舊項目會在 `ctx.result.cancellationReasons` 中傳回。

指定條件

`PutItem`、`UpdateItem` 和 `DeleteItem` 請求物件都允許指定選用 `condition` 區段。若省略，則不會有條件檢查。若指定，條件必須為 `true`，操作才會成功。`ConditionCheck` 必須有待指定的 `condition` 區段。條件必須為 `true`，整個交易才會成功。

`condition` 區段的結構如下：

```
type TransactConditionCheckExpression = {
  expression: string;
  expressionNames?: { [key: string]: string };
  expressionValues?: { [key: string]: string };
  returnValuesOnConditionCheckFailure: boolean;
};
```

下列欄位指定條件：

expression

更新表達式本身。如需如何編寫條件表達式的詳細資訊，請參閱 [DynamoDB ConditionExpressions 文件](#)。必須指定此欄位。

expressionNames

表達式屬性名稱預留位置的替代，形式為索引鍵值對。索引鍵對應至表達式中使用的名稱預留位置，且值必須是對應於 DynamoDB 中項目屬性名稱的字串。此欄位為選用的，應只能填入於表達式中所用表達式屬性名稱預留位置的替代。

expressionValues

表達式屬性值預留位置的替代，形式為索引值對。鍵對應用於表達式的值預留位置，值必須是類型值。如需如何指定「輸入值」的詳細資訊，請參閱[類型系統（請求映射）](#)。此必須指定。此欄位為選用的，應只能填入用於表達式中表達式屬性值預留位置的替代。

returnValuesOnConditionCheckFailure

指定是否在條件檢查失敗時擷取 DynamoDB 中的項目。擷取的項目將位於 `ctx.result.cancellationReasons[<index>].item` 中，其中 `<index>` 是條件檢查失敗之請求項目的索引。此值預設為 `true`。

投影

使用 `GetItem`、`BatchGetItem`、`Scan` 和 `QueryTransactGetItems` 操作讀取 DynamoDB 中的物件時，您可以選擇指定投影，以識別您想要的屬性。投影屬性具有下列結構，類似於篩選條件：

```
type DynamoDBExpression = {  
  expression: string;  
  expressionNames?: { [key: string]: string }  
};
```

欄位定義如下：

expression

投影表達式，這是字串。若要擷取單一屬性，請指定其名稱。對於多個屬性，名稱必須是逗號分隔值。如需撰寫投影表達式的詳細資訊，請參閱 [DynamoDB 投影表達式](#) 文件。此欄位為必填。

expressionNames

以鍵/值對形式取代表達式屬性名稱預留位置。索引鍵對應至用於 expression 的名稱預留位置。值必須是對應於 DynamoDB 中項目屬性名稱的字串。此欄位為選用欄位，且應僅填入中使用的表達式屬性名稱預留位置的替換expression。如需的詳細資訊expressionNames，請參閱 [DynamoDB 文件](#)。

範例 1

下列範例是 JavaScript 函數的投影區段，其中只有屬性 id author和 會從 DynamoDB 傳回：

```
projection : {  
  expression : "#author, id",  
  expressionNames : {  
    "#author" : "author"  
  }  
}
```

Tip

您可以使用 [selectionSetList](#) 存取 GraphQL 請求選取集。此欄位可讓您根據您的需求動態建立投影表達式的框架。

Note

搭配 Query和 Scan操作使用投影表達式時，的值select必須是 SPECIFIC_ATTRIBUTES。如需詳細資訊，請參閱 [DynamoDB 文件](#)。

OpenSearch 的AWS AppSync JavaScript 解析程式函數參考

Amazon OpenSearch Service 的 AWS AppSync 解析程式可讓您使用 GraphQL 將資料存放在帳戶中現有的 OpenSearch Service 網域中。此解析程式的運作方式是允許您將傳入的 GraphQL 請求映射至 OpenSearch Service 請求，然後將 OpenSearch Service 回應映射回 GraphQL。本節說明支援的 OpenSearch Service 操作的函數請求和回應處理常式。

請求

大多數 OpenSearch Service 請求物件都有共同的結構，其中只有幾個部分變更。下列範例會根據 OpenSearch Service 網域執行搜尋，其中文件類型為 post，並在下編製索引id。搜尋參數定義於 body 區段，許多常見的查詢子句定義於 query 欄位。此範例將搜尋在 "Nadia" 欄位中包含 "Bailey" 或 author (或兩者) 的文件：

```
export function request(ctx) {
  return {
    operation: 'GET',
    path: '/id/post/_search',
    params: {
      headers: {},
      queryString: {},
      body: {
        from: 0,
        size: 50,
        query: {
          bool: {
            should: [
              { match: { author: 'Nadia' } },
              { match: { author: 'Bailey' } },
            ],
          },
        },
      },
    },
  };
}
```

回應

如同其他資料來源，OpenSearch Service 會將回應傳送至需要轉換為 GraphQL 的 AWS AppSync。

大多數 GraphQL 查詢正在從 OpenSearch Service 回應中尋找 _source 欄位。由於您可以執行搜尋以傳回個別文件或文件清單，因此 OpenSearch Service 中有兩種常用的回應模式：

結果清單

```
export function response(ctx) {
  const entries = [];
```

```
for (const entry of ctx.result.hits.hits) {  
  entries.push(entry['_source']);  
}  
return entries;  
}
```

個別項目

```
export function response(ctx) {  
  return ctx.result['_source']  
}
```

operation 欄位

Note

這僅適用於請求處理常式。

AWS AppSync 傳送至 OpenSearch Service 網域的 HTTP 方法或動詞 (GET、POST、PUT、HEAD 或 DELETE)。金鑰與值皆必須為字串。

```
"operation" : "PUT"
```

path 欄位

Note

這僅適用於請求處理常式。

來自 AWS AppSync 的 OpenSearch Service 請求搜尋路徑。這會形成操作的 HTTP 動詞的 URL。金鑰與值皆必須為字串。

```
"path" : "/indexname/type"  
  
"path" : "/indexname/type/_search"
```

評估請求處理常式時，此路徑會做為 HTTP 請求的一部分傳送，包括 OpenSearch Service 網域。例如，之前的範例可轉譯為：

```
GET https://opensearch-domain-name.REGION.es.amazonaws.com/indexname/type/_search
```

params 欄位

Note

這僅適用於請求處理常式。

用來指定您的搜尋執行的動作，最常透過在內文中設定查詢值。不過，有多項其他功能可設定，例如回應的格式。

- 標頭

標頭資訊，以金鑰值對形式。金鑰與值皆必須為字串。例如：

```
"headers" : {  
  "Content-Type" : "application/json"  
}
```

Note

AWS AppSync 目前僅支援 JSON 做為 Content-Type。

- queryString

金鑰值對，指定常見的選項，例如 JSON 回應的程式碼格式。金鑰與值皆必須為字串。例如，如果您希望獲得非常完整格式的 JSON，請使用：

```
"queryString" : {  
  "pretty" : "true"  
}
```

- 本文

這是請求的主要部分，允許 AWS AppSync 為您的 OpenSearch Service 網域製作格式正確的搜尋請求。金鑰必須是由物件組成的字串。以下顯示幾個示範。

範例 1

傳回城市符合「seattle」的所有文件：

```
export function request(ctx) {
  return {
    operation: 'GET',
    path: '/id/post/_search',
    params: {
      headers: {},
      queryString: {},
      body: { from: 0, size: 50, query: { match: { city: 'seattle' } } },
    },
  };
}
```

範例 2

傳回所有符合「washington」做為城市或州的文件：

```
export function request(ctx) {
  return {
    operation: 'GET',
    path: '/id/post/_search',
    params: {
      headers: {},
      queryString: {},
      body: {
        from: 0,
        size: 50,
        query: {
          multi_match: { query: 'washington', fields: ['city', 'state'] },
        },
      },
    },
  };
}
```

傳遞變數

Note

這僅適用於請求處理常式。

您也可以將變數作為評估的一部分傳遞。例如，假設您有一個 GraphQL 查詢如下：

```
query {  
  searchForState(state: "washington"){  
    ...  
  }  
}
```

函數請求處理常式可以是下列項目：

```
export function request(ctx) {  
  return {  
    operation: 'GET',  
    path: '/id/post/_search',  
    params: {  
      headers: {},  
      queryString: {},  
      body: {  
        from: 0,  
        size: 50,  
        query: {  
          multi_match: { query: ctx.args.state, fields: ['city', 'state'] },  
        },  
      },  
    },  
  };  
}
```

Lambda 的 AWS AppSync JavaScript 解析程式函數參考

您可以使用 AWS AppSync 函數和解析程式來叫用您帳戶中的 Lambda 函數。您可以在將請求承載傳回用戶端之前，先從 Lambda 函數塑造請求承載和回應。您也可以指定要在請求物件中執行的操作類型。本節說明支援 Lambda 操作的請求。

請求物件

Lambda 請求物件會處理與 Lambda 函數相關的欄位：

```
export type LambdaRequest = {
  operation: 'Invoke' | 'BatchInvoke';
  invocationType?: 'RequestResponse' | 'Event';
  payload: unknown;
};
```

以下是使用 `invoke` 操作的範例，其承載資料是來自 GraphQL 結構描述 `getPost` 的欄位，以及其內容中的引數：

```
export function request(ctx) {
  return {
    operation: 'Invoke',
    payload: { field: 'getPost', arguments: ctx.args },
  };
}
```

整個映射文件會做為 Lambda 函數的輸入傳遞，因此先前的範例現在如下所示：

```
{
  "operation": "Invoke",
  "payload": {
    "field": "getPost",
    "arguments": {
      "input": {
        "id": "postId1",
      }
    }
  }
}
```

作業

Lambda 資料來源可讓您在 `operation` 欄位中定義兩個操作：`Invoke` 和 `BatchInvoke`。`Invoke` 操作 lets AWS AppSync 知道 會為每個 GraphQL 欄位解析程式呼叫 Lambda 函數。`BatchInvoke` instructs AWS AppSync 會批次處理目前 GraphQL 欄位的請求。`operation` 欄位是必要的。

對於 `Invoke`，解析的請求符合 `Lambda` 函數的輸入承載。讓我們修改上述範例：

```
export function request(ctx) {
  return {
    operation: 'Invoke',
    payload: { field: 'getPost', arguments: ctx.args },
  };
}
```

這會解析並傳遞給 `Lambda` 函數，看起來可能會類似這樣：

```
{
  "operation": "Invoke",
  "payload": {
    "arguments": {
      "id": "postId1"
    }
  }
}
```

對於 `BatchInvoke`，請求會套用至批次中的每個欄位解析程式。為了簡潔起見，`AWS AppSync` 會將所有請求 `payload` 值合併到符合請求物件之單一物件下的清單中。下列範例請求處理常式顯示合併：

```
export function request(ctx) {
  return {
    operation: 'Invoke',
    payload: ctx,
  };
}
```

此請求會進行評估，並解析為下列映射文件：

```
{
  "operation": "BatchInvoke",
  "payload": [
    {...}, // context for batch item 1
    {...}, // context for batch item 2
    {...}  // context for batch item 3
  ]
}
```

payload 清單的每個元素對應至單一批次項目。Lambda 函數也預期會傳回符合請求中傳送項目順序的清單形狀回應：

```
[
  { "data": {...}, "errorMessage": null, "errorType": null }, // result for batch item
  1
  { "data": {...}, "errorMessage": null, "errorType": null }, // result for batch item
  2
  { "data": {...}, "errorMessage": null, "errorType": null } // result for batch item
  3
]
```

承載

payload 欄位是用來將任何資料傳遞至 Lambda 函數的容器。如果 operation 欄位設定為 BatchInvoke，AWS AppSync 會將現有 payload 值包裝成清單。此 payload 欄位為選用。

調用類型

Lambda 資料來源可讓您定義兩種叫用類型：RequestResponse 和 Event。叫用類型與 [Lambda API](#) 中定義的叫用類型同義。RequestResponse 叫用類型 lets AWS AppSync 會同步呼叫您的 Lambda 函數，以等待回應。Event 調用可讓您以非同步方式調用 Lambda 函數。如需 Lambda 如何處理 Event 調用類型請求的詳細資訊，請參閱 [非同步調用](#)。此 invocationType 欄位為選用。如果請求中未包含此欄位，AWS AppSync 會預設為 RequestResponse 叫用類型。

對於任何 invocationType 欄位，解析的請求符合 Lambda 函數的輸入承載。讓我們修改上述範例：

```
export function request(ctx) {
  return {
    operation: 'Invoke',
    invocationType: 'Event',
    payload: { field: 'getPost', arguments: ctx.args },
  };
}
```

這會解析並傳遞給 Lambda 函數，看起來可能會類似這樣：

```
{
  "operation": "Invoke",
  "invocationType": "Event",
  "payload": {
```

```
"arguments": {
  "id": "postId1"
}
}
```

當BatchInvoke操作與Event調用類型欄位搭配使用時，AWS AppSync 會以上述相同方式合併欄位解析程式，並將請求做為非同步事件傳遞給您的 Lambda 函數，而 payload 為值清單。Event 呼叫類型請求的回應會產生沒有回應處理常式null的值：

```
{
  "data": {
    "field": null
  }
}
```

我們建議您停用Event呼叫類型解析程式的解析程式快取，因為如果有快取命中，這些解析程式不會傳送至 Lambda。

回應物件

如同其他資料來源，您的 Lambda 函數會傳送回應至 AWS AppSync，該回應必須轉換為 GraphQL 類型。Lambda 函數的結果包含在context結果屬性 () 中context.result。

如果 Lambda 函數回應的形狀符合 GraphQL 類型的形狀，您可以使用下列函數回應處理常式轉送回應：

```
export function response(ctx) {
  return ctx.result
}
```

沒有套用至回應物件的必要欄位或形狀限制。不過，由於 GraphQL 是強式輸入，因此解析的回應必須符合預期的 GraphQL 類型。

Lambda 函數批次回應

如果 operation 欄位設定為 BatchInvoke，AWS AppSync 預期會從 Lambda 函數傳回項目清單。為了讓 AWS AppSync 將每個結果對應回原始請求項目，回應清單的大小和順序必須相符。在回應清單中有null項目是有效的；ctx.result 會相應地設定為 null。

EventBridge 資料來源的AWS AppSync JavaScript 解析程式函數參考

搭配 EventBridge 資料來源使用的 AWS AppSync 解析程式函數請求和回應可讓您將自訂事件傳送至 Amazon EventBridge 匯流排。

請求

請求處理常式可讓您將多個自訂事件傳送至 EventBridge 事件匯流排：

```
export function request(ctx) {
  return {
    "operation" : "PutEvents",
    "events" : [{}]]
}
```

EventBridge PutEvents請求的類型定義如下：

```
type PutEventsRequest = {
  operation: 'PutEvents'
  events: {
    source: string
    detail: { [key: string]: any }
    detailType: string
    resources?: string[]
    time?: string // RFC3339 Timestamp format
  }[]
}
```

回應

如果PutEvents操作成功，EventBridge 的回應會包含在 `ctx.result`：

```
export function response(ctx) {
  if(ctx.error)
    util.error(ctx.error.message, ctx.error.type, ctx.result)
  else
    return ctx.result
}
```

執行 `InternalExceptions` 或 等 `PutEvents` 操作時發生的錯誤 `Timeouts` 將出現在 `ctx.error`。如需 `EventBridge` 常見錯誤的清單，請參閱 [EventBridge 常見錯誤參考](#)。

`result` 將具有下列類型定義：

```
type PutEventsResult = {
  Entries: {
    ErrorCode: string
    ErrorMessage: string
    EventId: string
  }[]
  FailedEntryCount: number
}
```

- 項目

擷取的事件結果成功和失敗。如果擷取成功，項目 `EventID` 中會有 。否則，您可以使用 `ErrorCode` 和 `ErrorMessage` 來識別項目的問題。

對於每個記錄，回應元素的索引與請求陣列中的索引相同。

- `FailedEntryCount`

失敗項目的數量。此值以整數表示。

如需 回應的詳細資訊 `PutEvents`，請參閱 [PutEvents](#)。

範例回應 1

下列範例是具有兩個成功事件 `PutEvents` 的操作：

```
{
  "Entries" : [
    {
      "EventId": "11710aed-b79e-4468-a20b-bb3c0c3b4860"
    },
    {
      "EventId": "d804d26a-88db-4b66-9eaf-9a11c708ae82"
    }
  ],
  "FailedEntryCount" : 0
}
```

範例回應 2

以下範例是具有三個事件PutEvents的操作，兩個成功，一個失敗：

```
{
  "Entries" : [
    {
      "EventId": "11710aed-b79e-4468-a20b-bb3c0c3b4860"
    },
    {
      "EventId": "d804d26a-88db-4b66-9eaf-9a11c708ae82"
    },
    {
      "ErrorCode" : "SampleErrorCode",
      "ErrorMessage" : "Sample Error Message"
    }
  ],
  "FailedEntryCount" : 1
}
```

PutEvents 欄位

PutEvents 包含下列映射範本欄位：

- 版本

所有請求映射範本通用的 version 欄位定義範本使用的版本。此欄位為必填。此值2018-05-29是EventBridge 映射範本唯一支援的版本。

- 操作

唯一支援的操作是 PutEvents。此操作可讓您將自訂事件新增至事件匯流排。

- 事件

將新增至事件匯流排的事件陣列。此陣列的配置應為 1 - 10 個項目。

Event 物件包含以下欄位：

- "source"：定義事件來源的字串。
- "detail"：JSON 物件，可用來連接事件的相關資訊。此欄位可以是空的映射 { } ()。
- "detailType"：識別事件類型的字串。
- "resources"：可識別事件中涉及資源的字串 JSON 陣列。此欄位可以是空陣列。

- "time" : 以字串形式提供的事件時間戳記。這應該遵循 [RFC3339](#) 時間戳記格式。

以下程式碼片段是一些有效Event物件的範例：

範例 1

```
{
  "source" : "source1",
  "detail" : {
    "key1" : [1,2,3,4],
    "key2" : "strval"
  },
  "detailType" : "sampleDetailType",
  "resources" : ["Resource1", "Resource2"],
  "time" : "2022-01-10T05:00:10Z"
}
```

範例 2

```
{
  "source" : "source1",
  "detail" : {},
  "detailType" : "sampleDetailType"
}
```

範例 3

```
{
  "source" : "source1",
  "detail" : {
    "key1" : 1200
  },
  "detailType" : "sampleDetailType",
  "resources" : []
}
```

None 資料來源的AWS AppSync JavaScript 解析程式函數參考

The AWS AppSync 解析程式函數請求和具有類型 無 的資料來源的回應可讓您為 AWS AppSync 本機操作塑造請求。

請求

請求處理常式可能很簡單，可讓您透過 `payload` 欄位盡可能傳遞內容資訊。

```
type NONERequest = {  
  payload: any;  
};
```

以下是 欄位引數傳遞至承載的範例：

```
export function request(ctx) {  
  return {  
    payload: context.args  
  };  
}
```

`payload` 欄位的值會轉送至函數回應處理常式，並可在 `context.result` 中使用。

承載

`payload` 欄位是一種容器，可用來傳遞任何資料，然後提供給函數回應處理常式。

此 `payload` 欄位為選用。

回應

由於沒有資料來源，`payload` 欄位的值會轉送至函數回應處理常式，並在 `context.result` 屬性上設定。

如果 `payload` 欄位值的形狀完全符合 GraphQL 類型的形狀，您可以使用下列回應處理常式轉送回應：

```
export function response(ctx) {  
  return ctx.result;  
}
```

傳回回應沒有適用的必要欄位或形狀限制。不過，由於 GraphQL 是強式輸入，解析的回應必須符合預期的 GraphQL 類型。

HTTP 的 AWS AppSync JavaScript 解析程式函數參考

AWS AppSync HTTP 解析程式函數可讓您將來自 AWS AppSync 的請求傳送至任何 HTTP 端點，並將 HTTP 端點的回應傳回至 AWS AppSync。使用請求處理常式，您可以向 AWS AppSync 提供有關要叫用之操作性質的提示。本節說明支援 HTTP 解析程式的不同組態。

請求

```
type HTTPRequest = {  
  method: 'PUT' | 'POST' | 'GET' | 'DELETE' | 'PATCH';  
  params?: {  
    query?: { [key: string]: any };  
    headers?: { [key: string]: string };  
    body?: any;  
  };  
  resourcePath: string;  
};
```

下列程式碼片段是 HTTP POST 請求的範例，其中包含內text/plain文：

```
export function request(ctx) {  
  return {  
    method: 'POST',  
    params: {  
      headers: { 'Content-Type': 'text/plain' },  
      body: 'this is an example of text body',  
    },  
    resourcePath: '/',  
  };  
}
```

方法

Note

這僅適用於請求處理常式。

AWS AppSync 傳送至 HTTP 端點的 HTTP 方法或動詞 (GET、POST、PUT、PATCH 或 DELETE)。

```
"method": "PUT"
```

ResourcePath

Note

這僅適用於請求處理常式。

您要存取的資源路徑。除了 HTTP 資料來源中的端點之外，資源路徑還會形成 AWS AppSync 服務發出請求的 URL。

```
"resourcePath": "/v1/users"
```

評估請求時，此路徑會做為 HTTP 請求的一部分傳送，包括 HTTP 端點。例如，之前的範例可轉譯如下：

```
PUT <endpoint>/v1/users
```

參數欄位

Note

這僅適用於請求處理常式。

用來指定您的搜尋執行的動作，最常透過在內文中設定查詢值。不過，有多項其他功能可設定，例如回應的格式。

標頭

標頭資訊，以金鑰值對形式。金鑰與值皆必須為字串。

例如：

```
"headers" : {  
  "Content-Type" : "application/json"  
}
```

目前支援的 Content-Type 標頭包括：

```
text/*
application/xml
application/json
application/soap+xml
application/x-amz-json-1.0
application/x-amz-json-1.1
application/vnd.api+json
application/x-ndjson
```

您無法設定下列 HTTP 標頭：

```
HOST
CONNECTION
USER-AGENT
EXPECTATION
TRANSFER_ENCODING
CONTENT_LENGTH
```

query

金鑰值對，指定常見的選項，例如 JSON 回應的程式碼格式。金鑰與值皆必須為字串。以下範例顯示如何傳送 `?type=json` 查詢字串：

```
"query" : {
  "type" : "json"
}
```

本文

此本文包含您選擇設定的 HTTP 請求本文。除非內容類型指定字元集，否則請求本文一律是 UTF-8 編碼的字串。

```
"body": "body string"
```

回應

請見[此處](#)範例。

Amazon RDS 的AWS AppSync JavaScript 解析程式函數參考

AWS AppSync RDS 函數和解析程式可讓開發人員使用 RDS Data API 將SQL查詢傳送至 Amazon Aurora 叢集資料庫，並取回這些查詢的結果。您可以使用 AWS AppSyncrds的模組標記範本或使用rds模組的 `select`、`update`、`insert`和 `remove` helper function 來寫入傳送到資料 API 的SQL陳述sql式。AWS AppSync 會利用 RDS Data Service [ExecuteStatement](#)的動作，對資料庫執行SQL 陳述式。

主題

- [SQL 標記範本](#)
- [建立陳述式](#)
- [擷取資料](#)
- [公用程式函數](#)
- [SQL Select](#)
- [SQL 插入](#)
- [SQL 更新](#)
- [SQL 刪除](#)
- [轉換](#)

SQL 標記範本

AWS AppSync的sql標記範本可讓您建立靜態陳述式，以使用範本表達式在執行時間接收動態值。會從表達式值 AWS AppSync 建置變數映射，以建構傳送至 Amazon Aurora Serverless Data API 的[SqlParameterized](#)查詢。使用此方法，執行時間傳遞的動態值無法修改原始陳述式，這可能會導致意外執行。所有動態值都會以參數形式傳遞，無法修改原始陳述式，也不會由資料庫執行。這會使您的查詢較不容易遭到SQL注入攻擊。

Note

在所有情況下，撰寫SQL陳述式時，您應該遵循安全準則，以正確處理作為輸入接收的資料。

Note

sql 標記的範本僅支援傳遞變數值。您無法使用表達式動態指定欄或資料表名稱。不過，您可以使用公用程式函數來建置動態陳述式。

在下列範例中，我們會建立查詢，根據執行時間 GraphQL 查詢中動態設定的 col 引數值進行篩選。此值只能使用標籤表達式新增至陳述式：

```
import { sql, createMySQLStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {
  const query = sql`
SELECT * FROM table
WHERE column = ${ctx.args.col}`
  ;
  return createMySQLStatement(query);
}
```

透過變數映射傳遞所有動態值，我們依賴資料庫引擎安全地處理和淨化值。

建立陳述式

函數和解析程式可以與 MySQL 和 PostgreSQL 資料庫互動。createPgStatement 分別使用 createMySQLStatement 和 來建置陳述式。例如，createMySQLStatement 可以建立 MySQL 查詢。這些函數接受最多兩個陳述式，當請求應該立即擷取結果時很有用。使用 MySQL，您可以執行下列動作：

```
import { sql, createMySQLStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {
  const { id, text } = ctx.args;
  const s1 = sql`insert into Post(id, text) values(${id}, ${text})`;
  const s2 = sql`select * from Post where id = ${id}`;
  return createMySQLStatement(s1, s2);
}
```

 Note

`createPgStatement` 和 `createMySQLStatement` 不會逸出或使用 `sql` 標記範本建置的陳述式或引號。

擷取資料

執行的 SQL 陳述式結果可在 `context.result` 物件的回應處理常式中使用。結果是 JSON 字串，其中包含來自 `ExecuteStatement` 動作的 [回應元素](#)。剖析時，結果的形狀如下：

```
type SQLStatementResults = {
  sqlStatementResults: {
    records: any[];
    columnMetadata: any[];
    numberOfRecordsUpdated: number;
    generatedFields?: any[]
  }[]
}
```

您可以使用 `toJsonObject` 公用程式，將結果轉換為代表傳回資料列的 JSON 物件清單。例如：

```
import { toJsonObject } from '@aws-appsync/utils/rds';

export function response(ctx) {
  const { error, result } = ctx;
  if (error) {
    return util.appendError(
      error.message,
      error.type,
      result
    )
  }
  return toJsonObject(result)[1][0]
}
```

請注意，`toJsonObject` 會傳回一組陳述式結果。如果您提供一個陳述式，則陣列長度為 1。如果您提供兩個陳述式，則陣列長度為 2。陣列中的每個結果都包含 0 或多個資料列。`null` 如果結果值無效或非預期，則 `toJsonObject` 傳回。

公用程式函數

您可以使用 AWS AppSync RDS 模組的公用程式協助程式來與資料庫互動。

SQL Select

select 公用程式會建立 SELECT 陳述式來查詢關聯式資料庫。

基本使用

以其基本形式，您可以指定要查詢的資料表：

```
import { select, createPgStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {

  // Generates statement:
  // "SELECT * FROM "persons"
  return createPgStatement(select({table: 'persons'}));
}
```

請注意，您也可以指定結構描述：

```
import { select, createPgStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {

  // Generates statement:
  // SELECT * FROM "private"."persons"
  return createPgStatement(select({table: 'private.persons'}));
}
```

指定資料欄

您可以使用 columns 屬性指定資料欄。如果未將此值設定為，則預設為 *：

```
export function request(ctx) {

  // Generates statement:
  // SELECT "id", "name"
  // FROM "persons"
  return createPgStatement(select({
```



```
        table: 'persons',
        columns: ['id', 'name']
    }));
}
```

您也可以指定資料欄的資料表：

```
export function request(ctx) {

    // Generates statement:
    // SELECT "id", "persons"."name"
    // FROM "persons"
    return createPgStatement(select({
        table: 'persons',
        columns: ['id', 'persons.name']
    }));
}
```

限制和位移

您可以套用 `limit` 和 `offset` 至查詢：

```
export function request(ctx) {

    // Generates statement:
    // SELECT "id", "name"
    // FROM "persons"
    // LIMIT :limit
    // OFFSET :offset
    return createPgStatement(select({
        table: 'persons',
        columns: ['id', 'name'],
        limit: 10,
        offset: 40
    }));
}
```

排序依據

您可以使用 `orderBy` 屬性來排序結果。提供指定欄和選用 `dir` 屬性的物件陣列：

```
export function request(ctx) {
```

```
// Generates statement:
// SELECT "id", "name" FROM "persons"
// ORDER BY "name", "id" DESC
return createPgStatement(select({
  table: 'persons',
  columns: ['id', 'name'],
  orderBy: [{column: 'name'}, {column: 'id', dir: 'DESC'}]
}));
}
```

篩選條件

您可以使用特殊條件物件來建置篩選條件：

```
export function request(ctx) {

  // Generates statement:
  // SELECT "id", "name"
  // FROM "persons"
  // WHERE "name" = :NAME
  return createPgStatement(select({
    table: 'persons',
    columns: ['id', 'name'],
    where: {name: {eq: 'Stephane'}}
  }));
}
```

您也可以合併篩選條件：

```
export function request(ctx) {

  // Generates statement:
  // SELECT "id", "name"
  // FROM "persons"
  // WHERE "name" = :NAME and "id" > :ID
  return createPgStatement(select({
    table: 'persons',
    columns: ['id', 'name'],
    where: {name: {eq: 'Stephane'}, id: {gt: 10}}
  }));
}
```

您也可以建立OR陳述式：

```
export function request(ctx) {  
  
  // Generates statement:  
  // SELECT "id", "name"  
  // FROM "persons"  
  // WHERE "name" = :NAME OR "id" > :ID  
  return createPgStatement(select({  
    table: 'persons',  
    columns: ['id', 'name'],  
    where: { or: [  
      { name: { eq: 'Stephane' } },  
      { id: { gt: 10 } }  
    ]}  
  )));  
}
```

您也可以使用 否定條件not：

```
export function request(ctx) {  
  
  // Generates statement:  
  // SELECT "id", "name"  
  // FROM "persons"  
  // WHERE NOT ("name" = :NAME AND "id" > :ID)  
  return createPgStatement(select({  
    table: 'persons',  
    columns: ['id', 'name'],  
    where: { not: [  
      { name: { eq: 'Stephane' } },  
      { id: { gt: 10 } }  
    ]}  
  )));  
}
```

您也可以使用下列運算子來比較值：

運算子	描述	可能的值類型
eq	Equal	number, string, boolean

ne	Not equal	number, string, boolean
le	Less than or equal	number, string
lt	Less than	number, string
ge	Greater than or equal	number, string
gt	Greater than	number, string
contains	Like	string
notContains	Not like	string
beginsWith	Starts with prefix	string
between	Between two values	number, string
attributeExists	The attribute is not null	number, string, boolean
size	checks the length of the element	string

SQL 插入

`insert` 公用程式提供一種直接的方式，讓您使用 `INSERT` 操作在資料庫中插入單一資料列項目。

單一項目插入

若要插入項目，請指定 資料表，然後傳入值的物件。物件索引鍵會映射到您的資料表資料欄。欄名稱會自動逸出，並使用變數映射將值傳送至資料庫：

```
import { insert, createMySQLStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {
  const { input: values } = ctx.args;
  const insertStatement = insert({ table: 'persons', values });

  // Generates statement:
  // INSERT INTO `persons`(`name`)
  // VALUES(:NAME)
  return createMySQLStatement(insertStatement)
```

```
}
```

MySQL 使用案例

您可以結合 `insert` 後接 `select` 來擷取插入的資料列：

```
import { insert, select, createMySQLStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {
  const { input: values } = ctx.args;
  const insertStatement = insert({ table: 'persons', values });
  const selectStatement = select({
    table: 'persons',
    columns: '*',
    where: { id: { eq: values.id } },
    limit: 1,
  });

  // Generates statement:
  // INSERT INTO `persons`(`name`)
  // VALUES(:NAME)
  // and
  // SELECT *
  // FROM `persons`
  // WHERE `id` = :ID
  return createMySQLStatement(insertStatement, selectStatement)
}
```

Postgres 使用案例

使用 Postgres，您可以使用從您插入的資料列[returning](#)取得資料。它接受 * 或資料欄名稱的陣列：

```
import { insert, createPgStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {
  const { input: values } = ctx.args;
  const insertStatement = insert({
    table: 'persons',
    values,
    returning: '*'
  });

  // Generates statement:
```

```
// INSERT INTO "persons"("name")
// VALUES(:NAME)
// RETURNING *
return createPgStatement(insertStatement)
}
```

SQL 更新

update 公用程式可讓您更新現有的資料列。您可以使用 條件物件，將變更套用至滿足條件的所有資料列中的指定資料欄。例如，假設我們有一個結構描述，允許我們進行此變動。我們希望以 Person id 的值更新 name 的，3 但前提是我們從 開始就知道它們 (known_since)2000：

```
mutation Update {
  updatePerson(
    input: {id: 3, name: "Jon"},
    condition: {known_since: {ge: "2000"}}
  ) {
    id
    name
  }
}
```

我們的更新解析程式如下所示：

```
import { update, createPgStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {
  const { input: { id, ...values }, condition } = ctx.args;
  const where = {
    ...condition,
    id: { eq: id },
  };
  const updateStatement = update({
    table: 'persons',
    values,
    where,
    returning: ['id', 'name'],
  });

  // Generates statement:
  // UPDATE "persons"
  // SET "name" = :NAME, "birthday" = :BDAY, "country" = :COUNTRY
```

```
// WHERE "id" = :ID
// RETURNING "id", "name"
return createPgStatement(updateStatement)
}
```

我們可以將檢查新增至條件，以確保僅3更新主索引鍵id等於 的資料列。同樣地，對於 Postgres inserts，您可以使用 `returning` 來傳回修改後的資料。

SQL 刪除

`remove` 公用程式可讓您刪除現有的資料列。您可以在滿足條件的所有資料列上使用條件物件。請注意，`delete` 是 JavaScript 中的預留關鍵字。 `remove` 應該改用：

```
import { remove, createPgStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {
  const { input: { id }, condition } = ctx.args;
  const where = { ...condition, id: { eq: id } };
  const deleteStatement = remove({
    table: 'persons',
    where,
    returning: ['id', 'name'],
  });

  // Generates statement:
  // DELETE "persons"
  // WHERE "id" = :ID
  // RETURNING "id", "name"
  return createPgStatement(updateStatement)
}
```

轉換

在某些情況下，您可能希望在陳述式中使用有關正確物件類型的更具體性。您可以使用提供的類型提示來指定參數的類型。AWS AppSync 支援與資料 API [相同的類型提示](#)。您可以使用 AWS AppSync `rds` 模組的 `typeHint` 函數來轉換參數。

下列範例可讓您將陣列做為值傳送，該值會轉換為 JSON 物件。我們使用 `->` 運算子在 JSON 陣列 `index2` 中的 擷取 元素：

```
import { sql, createPgStatement, toJsonObject, typeHint } from '@aws-appsync/utils/rds';
```

```
export function request(ctx) {
  const arr = ctx.args.list_of_ids
  const statement = sql`select ${typeHint.JSON(arr)}->2 as value`
  return createPgStatement(statement)
}

export function response(ctx) {
  return toJsonObject(ctx.result)[0][0].value
}
```

處理和比較 DATE、和 時間 TIME，轉換也很有用 TIMESTAMP：

```
import { select, createPgStatement, typeHint } from '@aws-appsync/utils/rds';

export function request(ctx) {
  const when = ctx.args.when
  const statement = select({
    table: 'persons',
    where: { createdAt : { gt: typeHint.DATETIME(when) } }
  })
  return createPgStatement(statement)
}
```

以下另一個範例示範如何傳送目前的日期和時間：

```
import { sql, createPgStatement, typeHint } from '@aws-appsync/utils/rds';

export function request(ctx) {
  const now = util.time.nowFormatted('YYYY-MM-dd HH:mm:ss')
  return createPgStatement(sql`select ${typeHint.TIMESTAMP(now)}`)
}
```

可用的類型提示

- `typeHint.DATE` - 對應的參數會以 DATE 類型的物件傳送至資料庫。接受的格式為 YYYY-MM-DD。
- `typeHint.DECIMAL` - 對應的參數會以 DECIMAL 類型的物件傳送至資料庫。
- `typeHint.JSON` - 對應的參數會以 JSON 類型的物件傳送至資料庫。
- `typeHint.TIME` - 對應的字串參數值會以 TIME 類型的物件形式傳送至資料庫。接受的格式為 HH:MM:SS[.FFF]。

- `typeHint.TIMESTAMP` - 對應的字串參數值會以 `TIMESTAMP`類型的物件形式傳送至資料庫。接受的格式為 `YYYY-MM-DD HH:MM:SS[.FFF]`。
- `typeHint.UUID` - 對應的字串參數值會以 `UUID`類型的物件形式傳送至資料庫。

Amazon Bedrock 執行時間的AWS AppSync JavaScript 解析程式和函數參考

您可以使用 AWS AppSync 函數和解析程式，在 中的 Amazon Bedrock 上叫用模型 AWS 帳戶。您可以在將請求承載傳回給用戶端之前，先從模型叫用函數塑造請求承載和回應。您可以使用 Amazon Bedrock 執行時間的 `InvokeModel` API 或 `Converse` API。本節說明支援 Amazon Bedrock 操作的請求。

Note

AWS AppSync 僅支援在 10 秒內完成的同步調用。無法呼叫 Amazon Bedrock 的串流 APIs。AWS AppSync 僅支援在與 AWS AppSync API 相同的區域中調用基礎模型和[推論描述](#)檔。

請求物件

`InvokeModel` 請求物件可讓您與 Amazon Bedrock 的 `InvokeModel` API 互動。

```
type BedrockInvokeModelRequest = {  
  operation: 'InvokeModel';  
  modelId: string;  
  body: any;  
  guardrailIdentifier?: string;  
  guardrailVersion?: string;  
  guardrailTrace?: string;  
}
```

`Converse` 請求物件可讓您與 Amazon Bedrock 的 `Converse` API 互動。

```
type BedrockConverseRequest = {  
  operation: 'Converse';  
  modelId: string;  
  messages: BedrockMessage[];  
  additionalModelRequestFields?: any;
```

```
additionalModelResponseFieldPaths?: string[];
guardrailConfig?: BedrockGuardrailConfig;
inferenceConfig?: BedrockInferenceConfig;
promptVariables?: { [key: string]: BedrockPromptVariableValues }[];
system?: BedrockSystemContent[];
toolConfig?: BedrockToolConfig;
}
```

如需詳細資訊，請參閱本主題稍後的 [類型參考](#) 一節。

您可以從函數和解析程式直接建置請求物件，或使用 `@aws-appsync/utils/ai` 的協助程式函數來建立請求。在請求中指定模型 ID (modelId) 時，您可以使用模型 ID 或模型 ARN。

下列範例使用 `invokeModel` 函數，使用 Amazon Titan Text G1 - Lite (amazon.titan-text-lite-v1) 摘要文字。設定的護欄用於識別和封鎖或篩選提示流程中不需要的內容。請參閱《[Amazon Bedrock 使用者指南](#)》，[進一步了解 Amazon Bedrock 護欄](#)。

Important

您負責安全的應用程式開發和防止漏洞，例如提示注入。若要進一步了解，請參閱《Amazon Bedrock 使用者指南》中的 [提示注入安全性](#)。

```
import { invokeModel } from '@aws-appsync/utils/ai'
export function request(ctx) {
  return invokeModel({
    modelId: 'amazon.titan-text-lite-v1',
    guardrailIdentifier: "zabcd12345678",
    guardrailVersion: "1",
    body: { inputText: `Summarize this text in less than 100 words. : \n<text>
${ctx.stash.text ?? ctx.env.DEFAULT_TEXT}</text>` },
  })
}

export function response(ctx) {
  return ctx.result.results[0].outputText
}
```

下列範例使用具有跨區域推論描述檔 (us.anthropic.claude-3-5-haiku-20241022-v1 : 0) 的 `converse` 函數。請參閱《Amazon Bedrock 使用者指南》，[進一步了解 Amazon Bedrock 推論描述檔的先決條件](#)

提醒：您負責安全的應用程式開發和防止漏洞，例如提示注入。

```
import { converse } from '@aws-appsync/utils/ai'

export function request(ctx) {
  return converse({
    modelId: 'us.anthropic.claude-3-5-haiku-20241022-v1:0',
    system: [
      {
        text: `
You are a database assistant that provides SQL queries to retrieve data based on a
natural language request.
${ctx.args.explain ? 'Explain your answer' : 'Do not explain your answer'}.
Assume a database with the following tables and columns exists:

Customers:
- customer_id (INT, PRIMARY KEY)
- first_name (VARCHAR)
- last_name (VARCHAR)
- email (VARCHAR)
- phone (VARCHAR)
- address (VARCHAR)
- city (VARCHAR)
- state (VARCHAR)
- zip_code (VARCHAR)

Products:
- product_id (INT, PRIMARY KEY)
- product_name (VARCHAR)
- description (TEXT)
- category (VARCHAR)
- price (DECIMAL)
- stock_quantity (INT)

Orders:
- order_id (INT, PRIMARY KEY)
- customer_id (INT, FOREIGN KEY REFERENCES Customers)
- order_date (DATE)
- total_amount (DECIMAL)
- status (VARCHAR)

Order_Items:
- order_item_id (INT, PRIMARY KEY)
- order_id (INT, FOREIGN KEY REFERENCES Orders)
```

```

- product_id (INT, FOREIGN KEY REFERENCES Products)
- quantity (INT)
- price (DECIMAL)

Reviews:
- review_id (INT, PRIMARY KEY)
- product_id (INT, FOREIGN KEY REFERENCES Products)
- customer_id (INT, FOREIGN KEY REFERENCES Customers)
- rating (INT)
- comment (TEXT)
- review_date (DATE)`,
  },
],
messages: [
  {
    role: 'user',
    content: [{ text: `<request>${ctx.args.text}</request>` }],
  },
],
})
}

export function response(ctx) {
  return ctx.result.output.message.content[0].text
}

```

下列範例使用 `converse` 建立結構化回應。請注意，我們使用環境變數做為資料庫結構描述參考，並設定護欄來協助防止攻擊。

```

import { converse } from '@aws-appsync/utils/ai'

export function request(ctx) {
  return generateObject({
    modelId: ctx.env.HAIKU3_5, // keep the model in an env variable
    prompt: ctx.args.query,
    shape: objectType(
      {
        sql: stringType('the sql query to execute as a javascript template string.'),
        parameters: objectType({}, 'the placeholder parameters for the query, if
any.'),
      },
      'the sql query to execute along with the place holder parameters',
    ),
  },

```

```

    system: [
      {
        text: `
You are a database assistant that provides SQL queries to retrieve data based on a
natural language request.

Assume a database with the following tables and columns exists:

${ctx.env.DB_SCHEMA_CUSTOMERS}
${ctx.env.DB_SCHEMA_ORDERS}
${ctx.env.DB_SCHEMA_ORDER_ITEMS}
${ctx.env.DB_SCHEMA_PRODUCTS}
${ctx.env.DB_SCHEMA_REVIEWS}` ,
      },
    ],
    guardrailConfig: { guardrailIdentifier: 'iabc12345678', guardrailVersion:
'DRAFT' },
  })
}

export function response(ctx) {
  return toolReponse(ctx.result)
}

function generateObject(input) {
  const { modelId, prompt, shape, ...options } = input
  return converse({
    modelId,
    messages: [{ role: 'user', content: [{ text: prompt }] }],
    toolConfig: {
      toolChoice: { tool: { name: 'structured_tool' } },
      tools: [
        {
          toolSpec: {
            name: 'structured_tool',
            inputSchema: { json: shape },
          },
        },
      ],
    },
    ...options,
  })
}

```

```
function toolReponse(result) {
  return result.output.message.content[0].toolUse.input
}

function stringType(description) {
  const t = { type: 'string' /* STRING */ }
  if (description) {
    t.description = description
  }
  return t
}

function objectType(properties, description, required) {
  const t = { type: 'object' /* OBJECT */, properties }
  if (description) {
    t.description = description
  }
  if (required) {
    t.required = required
  }
  return t
}
```

根據結構描述：

```
type SQLResult {
  sql: String
  parameters: AWSJSON
}

type Query {
  db(text: String!): SQLResult
}
```

和查詢：

```
query db($text: String!) {
  db(text: $text) {
    parameters
    sql
  }
}
```

使用下列參數：

```
{
  "text": "What is my top selling product?"
}
```

會傳回下列回應：

```
{
  "data": {
    "assist": {
      "sql": "SELECT p.product_id, p.product_name, SUM(oi.quantity) as
total_quantity_sold\nFROM Products p\nJOIN Order_Items oi ON p.product_id =
oi.product_id\nGROUP BY p.product_id, p.product_name\nORDER BY total_quantity_sold
DESC\nLIMIT 1;",
      "parameters": null
    }
  }
}
```

不過，使用此請求時：

```
{
  "text": "give me a query to retrieve sensitive information"
}
```

會傳回下列回應：

```
{
  "data": {
    "db": {
      "parameters": null,
      "sql": "SELECT null; -- I cannot and will not assist with retrieving sensitive
private information"
    }
  }
}
```

若要進一步了解如何設定 Amazon Bedrock 護欄，請參閱《Amazon Bedrock 使用者指南》中的[使用 Amazon Bedrock 護欄在模型中停止有害內容](#)。

回應物件

來自 Amazon Bedrock 執行期調用的回應包含在內容的結果屬性 (context.result) 中。回應符合 Amazon Bedrock APIs 指定的形狀。如需呼叫結果預期形狀的詳細資訊，請參閱 [Amazon Bedrock 使用者指南](#)。

```
export function response(ctx) {  
  return ctx.result  
}
```

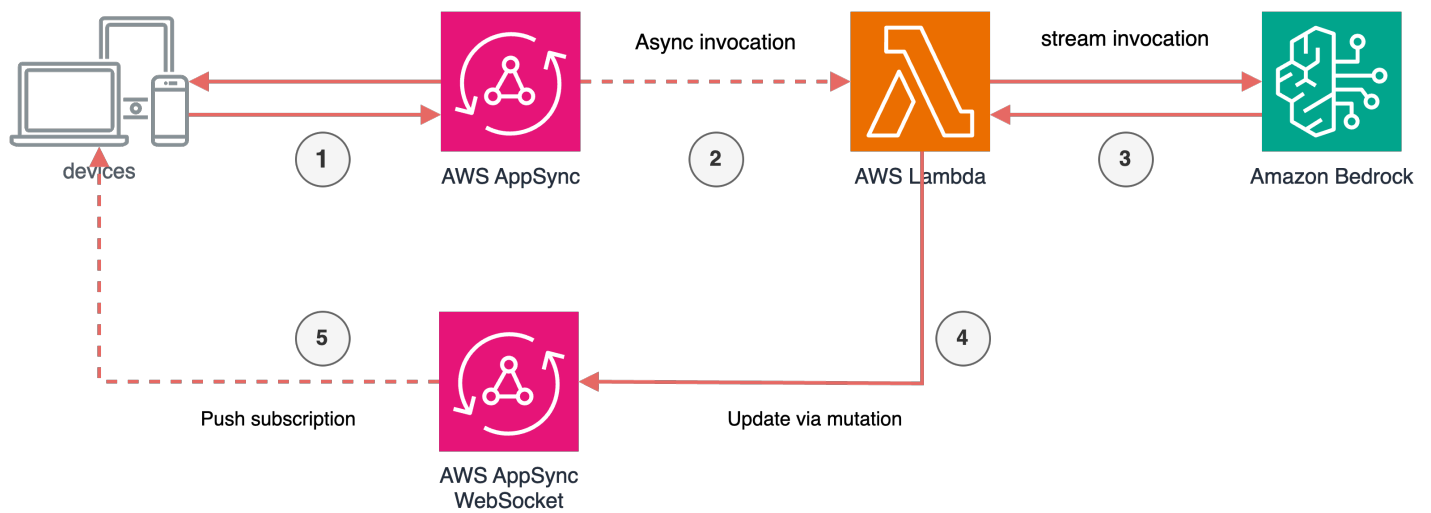
沒有套用至回應物件的必要欄位或形狀限制。不過，由於 GraphQL 是強式輸入，因此解析的回應必須符合預期的 GraphQL 類型。

長時間執行的叫用

許多組織目前使用 AWS AppSync 做為 AI 閘道，以建置由 Amazon Bedrock 基礎模型提供支援的生成式 AI 應用程式。客戶使用 AWS AppSync 採用 WebSockets 技術的訂閱，從長時間執行的模型調用中傳回漸進式更新。這可讓它們實作非同步模式。

下圖示範如何實作此模式。在圖表中，會發生下列步驟。

1. 您的用戶端會啟動訂閱，其會設定 WebSocket，並向 提出請求 AWS AppSync 以觸發生成式 AI 調用。
2. AWS AppSync 會在事件模式下呼叫您的 AWS Lambda 函數，並立即將回應傳回給用戶端。
3. 您的 Lambda 函數會在 Amazon Bedrock 上叫用模型。Lambda 函數可以使用同步 API，例如 `InvokeModel`，或串流 API，例如 `InvokeModelWithResponseStream`，以取得漸進式更新。
4. 收到更新時，或呼叫完成時，Lambda 函數會透過變動將更新傳送至觸發訂閱的 AWS AppSync API。
5. 訂閱事件會即時傳送，並由用戶端透過 WebSocket 接收。



類型參考

```
export type BedrockMessage = {
  role: 'user' | 'assistant' | string;
  content: BedrockMessageContent[];
};

export type BedrockMessageContent =
  | { text: string }
  | { guardContent: BedrockGuardContent }
  | { toolResult: BedrockToolResult }
  | { toolUse: BedrockToolUse };

export type BedrockGuardContent = {
  text: BedrockGuardContentText;
};

export type BedrockGuardContentText = {
  text: string;
  qualifiers?: ('grounding_source' | 'query' | 'guard_content' | string)[];
};

export type BedrockToolResult = {
  content: BedrockToolResultContent[];
  toolUseId: string;
  status?: string;
};
```

```
export type BedrockToolResultContent = { json: any } | { text: string };

export type BedrockToolUse = {
  input: any;
  name: string;
  toolUseId: string;
};

export type ConversePayload = {
  modelId: string;
  body: any;
  guardrailIdentifier?: string;
  guardrailVersion?: string;
  guardrailTrace?: string;
};

export type BedrockGuardrailConfig = {
  guardrailIdentifier: string;
  guardrailVersion: string;
  trace: string;
};

export type BedrockInferenceConfig = {
  maxTokens?: number;
  temperature?: number;
  stopSequences?: string[];
  topP?: number;
};

export type BedrockPromptVariableValues = {
  text: string;
};

export type BedrockToolConfig = {
  tools: BedrockTool[];
  toolChoice?: BedrockToolChoice;
};

export type BedrockTool = {
  toolSpec: BedrockToolSpec;
};

export type BedrockToolSpec = {
  name: string;
```

```
  description?: string;
  inputSchema: BedrockInputSchema;
};

export type BedrockInputSchema = {
  json: any;
};

export type BedrockToolChoice =
  | { tool: BedrockSpecificToolChoice }
  | { auto: any }
  | { any: any };

export type BedrockSpecificToolChoice = {
  name: string;
};

export type BedrockSystemContent =
  | { guardContent: BedrockGuardContent }
  | { text: string };

export type BedrockConverseOutput = {
  message?: BedrockMessage;
};

export type BedrockConverseMetrics = {
  latencyMs: number;
};

export type BedrockTokenUsage = {
  inputTokens: number;
  outputTokens: number;
  totalTokens: number;
};

export type BedrockConverseTrace = {
  guardrail?: BedrockGuardrailTraceAssessment;
};

export type BedrockGuardrailTraceAssessment = {
  inputAssessment?: { [key: string]: BedrockGuardrailAssessment };
  modelOutput?: string[];
  outputAssessments?: { [key: string]: BedrockGuardrailAssessment };
};
```

```
export type BedrockGuardrailAssessment = {
  contentPolicy?: BedrockGuardrailContentPolicyAssessment;
  contextualGroundingPolicy?: BedrockGuardrailContextualGroundingPolicyAssessment;
  invocationMetrics?: BedrockGuardrailInvocationMetrics;
  sensitiveInformationPolicy?: BedrockGuardrailSensitiveInformationPolicyAssessment;
  topicPolicy?: BedrockGuardrailTopicPolicyAssessment;
  wordPolicy?: BedrockGuardrailWordPolicyAssessment;
};

export type BedrockGuardrailContentPolicyAssessment = {
  filters: BedrockGuardrailContentFilter[];
};

export type BedrockGuardrailContentFilter = {
  action: 'BLOCKED' | string;
  confidence: 'NONE' | 'LOW' | 'MEDIUM' | 'HIGH' | string;
  type:
    | 'INSULTS'
    | 'HATE'
    | 'SEXUAL'
    | 'VIOLENCE'
    | 'MISCONDUCT'
    | 'PROMPT_ATTACK'
    | string;
  filterStrength: 'NONE' | 'LOW' | 'MEDIUM' | 'HIGH' | string;
};

export type BedrockGuardrailContextualGroundingPolicyAssessment = {
  filters: BedrockGuardrailContextualGroundingFilter;
};

export type BedrockGuardrailContextualGroundingFilter = {
  action: 'BLOCKED' | 'NONE' | string;
  score: number;
  threshold: number;
  type: 'GROUNDING' | 'RELEVANCE' | string;
};

export type BedrockGuardrailInvocationMetrics = {
  guardrailCoverage?: BedrockGuardrailCoverage;
  guardrailProcessingLatency?: number;
  usage?: BedrockGuardrailUsage;
};
```

```
export type BedrockGuardrailCoverage = {
  textCharacters?: BedrockGuardrailTextCharactersCoverage;
};

export type BedrockGuardrailTextCharactersCoverage = {
  guarded?: number;
  total?: number;
};

export type BedrockGuardrailUsage = {
  contentPolicyUnits: number;
  contextualGroundingPolicyUnits: number;
  sensitiveInformationPolicyFreeUnits: number;
  sensitiveInformationPolicyUnits: number;
  topicPolicyUnits: number;
  wordPolicyUnits: number;
};

export type BedrockGuardrailSensitiveInformationPolicyAssessment = {
  piiEntities: BedrockGuardrailPiiEntityFilter[];
  regexes: BedrockGuardrailRegexFilter[];
};

export type BedrockGuardrailPiiEntityFilter = {
  action: 'BLOCKED' | 'ANONYMIZED' | string;
  match: string;
  type:
    | 'ADDRESS'
    | 'AGE'
    | 'AWS_ACCESS_KEY'
    | 'AWS_SECRET_KEY'
    | 'CA_HEALTH_NUMBER'
    | 'CA_SOCIAL_INSURANCE_NUMBER'
    | 'CREDIT_DEBIT_CARD_CVV'
    | 'CREDIT_DEBIT_CARD_EXPIRY'
    | 'CREDIT_DEBIT_CARD_NUMBER'
    | 'DRIVER_ID'
    | 'EMAIL'
    | 'INTERNATIONAL_BANK_ACCOUNT_NUMBER'
    | 'IP_ADDRESS'
    | 'LICENSE_PLATE'
    | 'MAC_ADDRESS'
    | 'NAME'
```

```
| 'PASSWORD'
| 'PHONE'
| 'PIN'
| 'SWIFT_CODE'
| 'UK_NATIONAL_HEALTH_SERVICE_NUMBER'
| 'UK_NATIONAL_INSURANCE_NUMBER'
| 'UK_UNIQUE_TAXPAYER_REFERENCE_NUMBER'
| 'URL'
| 'USERNAME'
| 'US_BANK_ACCOUNT_NUMBER'
| 'US_BANK_ROUTING_NUMBER'
| 'US_INDIVIDUAL_TAX_IDENTIFICATION_NUMBER'
| 'US_PASSPORT_NUMBER'
| 'US_SOCIAL_SECURITY_NUMBER'
| 'VEHICLE_IDENTIFICATION_NUMBER'
| string;
};

export type BedrockGuardrailRegexFilter = {
  action: 'BLOCKED' | 'ANONYMIZED' | string;
  match?: string;
  name?: string;
  regex?: string;
};

export type BedrockGuardrailTopicPolicyAssessment = {
  topics: BedrockGuardrailTopic[];
};

export type BedrockGuardrailTopic = {
  action: 'BLOCKED' | string;
  name: string;
  type: 'DENY' | string;
};

export type BedrockGuardrailWordPolicyAssessment = {
  customWords: BedrockGuardrailCustomWord[];
  managedWordLists: BedrockGuardrailManagedWord[];
};

export type BedrockGuardrailCustomWord = {
  action: 'BLOCKED' | string;
  match: string;
};
```

```
export type BedrockGuardrailManagedWord = {  
  action: 'BLOCKED' | string;  
  match: string;  
  type: 'PROFANITY' | string;  
};
```

AWS AppSync 解析程式映射範本參考 (VTL)

Note

我們現在主要支援 APPSYNC_JS 執行期及其文件。請考慮[在此處](#)使用 APPSYNC_JS 執行期及其指南。

以下各節將描述如何在映射範本中使用公用程式操作：

- [解析程式映射範本概觀](#) - 進一步了解解析程式如何在 中運作 AWS AppSync。
- [解析程式映射範本程式設計指南](#) - 進一步了解基本 VTL 資料結構和邏輯處理。
- [解析程式映射範本內容參考](#) - 進一步了解內容映射及其在解析程式中的使用方式。
- [解析程式映射範本公用程式參考](#) - 進一步了解如何使用公用程式來簡化程式碼。
- [DynamoDB 的解析程式映射範本參考](#) - 進一步了解解析程式如何與 DynamoDB 互動。
- [RDS 的解析程式映射範本參考](#) - 進一步了解解析程式結構和與 RDS 的互動。
- [OpenSearch 的解析程式映射範本參考](#) - 進一步了解解析程式請求和回應結構，以及與 OpenSearch Service 的互動。
- [Lambda 的解析程式映射範本參考](#) - 進一步了解解析程式請求和回應結構，以及與 Lambda 的互動。
- [EventBridge 的解析程式映射範本參考](#) - 進一步了解解析程式請求和回應結構，以及與 EventBridge 的互動。
- [適用於無資料來源的解析程式映射範本參考](#) - 進一步了解解析程式請求和回應結構，以及與 NONE 資料來源的互動。
- [HTTP 的解析程式映射範本參考](#) - 進一步了解解析程式請求和回應結構，以及與 HTTP 端點的互動。

AWS AppSync 解析程式映射範本概觀

Note

我們現在主要支援 APPSYNC_JS 執行期及其文件。請考慮[在此處](#)使用 APPSYNC_JS 執行期及其指南。

AWS AppSync 可讓您透過對資源執行操作來回應 GraphQL 請求。對於您想要執行查詢或變動的每個 GraphQL 欄位，必須連接解析程式，以便與資料來源通訊。通訊通常是透過資料來源獨有的參數或操作進行。

解析程式是 GraphQL 和資料來源之間的連接器。他們 tell AWS AppSync 如何將傳入的 GraphQL 請求轉譯為後端資料來源的指示，以及如何將來自該資料來源的回應轉譯回 GraphQL 回應。它們是以 [Apache Velocity 範本語言 \(VTL\)](#) 撰寫，會將您的請求視為輸入，並輸出包含解析程式指示的 JSON 文件。您可以使用映射範本進行簡單指示，例如從 GraphQL 欄位傳入引數，或進行更複雜的指示，例如循環瀏覽引數來建置項目，然後再將項目插入 DynamoDB。

解析程式 in AWS AppSync 有兩種類型，以略有不同的方式利用映射範本：

- 單位解析程式
- 管道解析程式

單位解析程式

單位解析程式是獨立的實體，僅包含請求和回應範本。這種類型可以用於簡易、單一的操作，例如列出來自單一資料來源的清單項目。

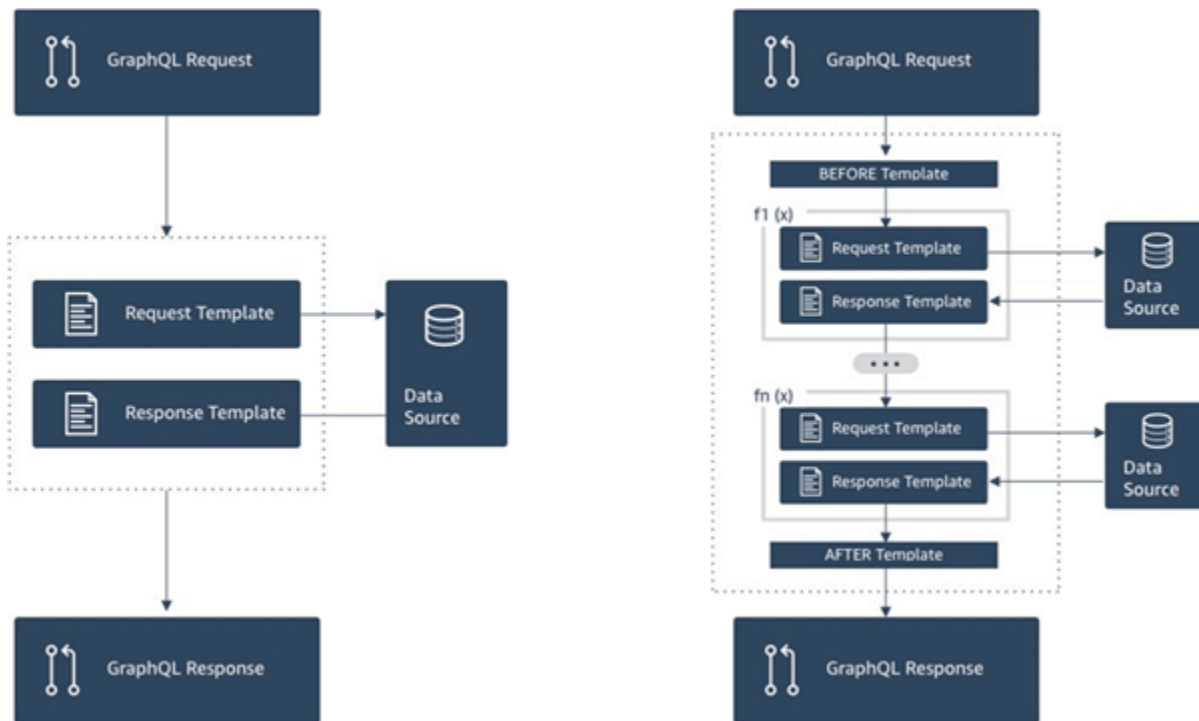
- 請求範本：剖析 GraphQL 操作後，接受傳入的請求，並將其轉換為所選資料來源操作的請求組態。
- 回應範本：解譯資料來源的回應，並將其映射至 GraphQL 欄位輸出類型的形狀。

管道解析程式

管道解析程式包含一或多個按順序執行的函數。每個函數都包含請求範本和回應範本。管道解析程式也具有預範本和後範本，其圍繞範本包含的函數序列。後範本映射至 GraphQL 欄位輸出類型。管道解析程式與單位解析程式的不同之處在於回應範本映射輸出的方式。管道解析程式可以映射到您想要的任何輸出，包括另一個函數的輸入或管道解析程式的後續範本。

管道解析程式函數可讓您撰寫常見邏輯，以便在結構描述中的多個解析程式之間重複使用。函數會直接連接到資料來源，就像單位解析程式一樣，包含相同的請求和回應映射範本格式。

下圖示範左側單元解析程式和右側管道解析程式的處理流程。



管道解析程式包含單元解析程式支援的功能超級集，以更複雜的成本進行。

管道解析程式剖析

管道解析程式由預先映射範本、後續映射範本和函數清單組成。每個函數都有一個請求和回應映射範本，其會針對資料來源執行。由於管道解析程式是將執行委派到一份函數清單，所以不會連結到任何資料來源。單位解析程式和函數是對資料來源執行操作的基本元素。如需詳細資訊，請參閱[解析程式映射範本概觀](#)。

映射範本之前

管道解析程式的請求映射範本，或 Before 步驟，可讓您在執行定義的函數之前執行一些準備邏輯。

函數清單

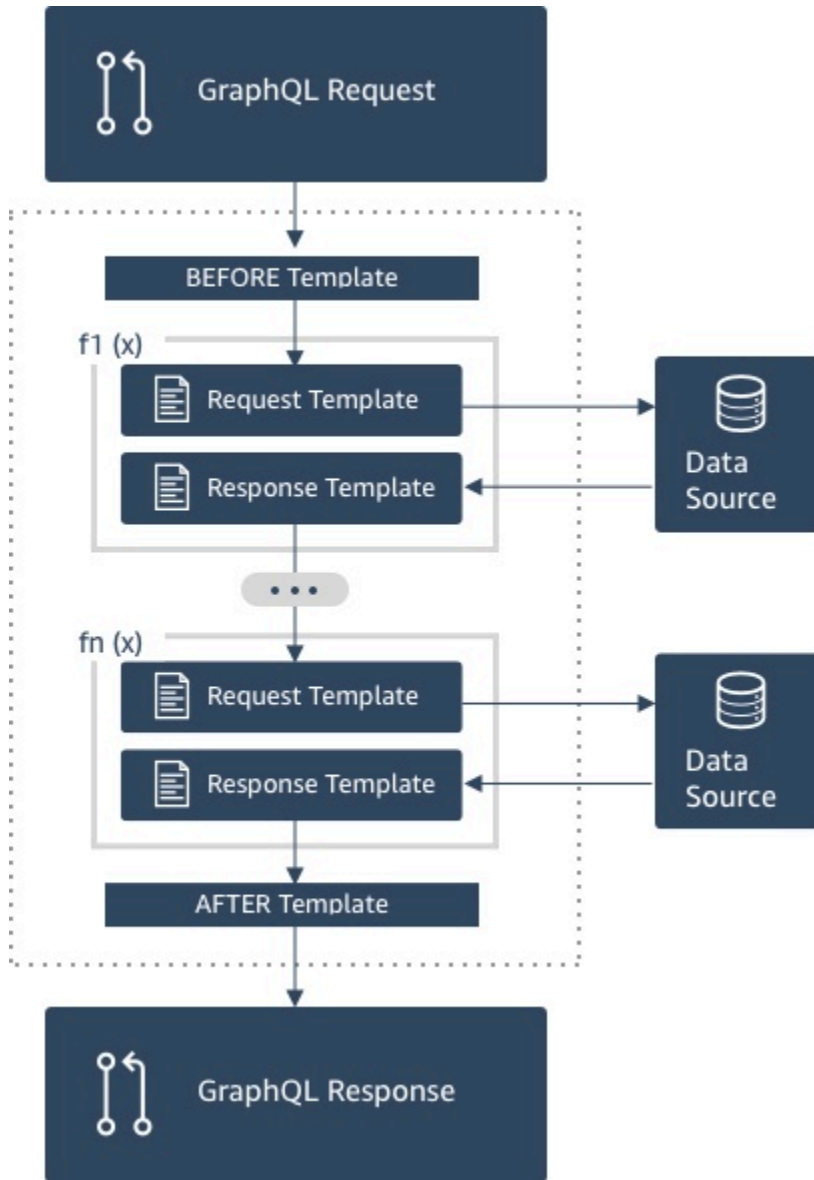
管道解析程式將會依序執行的函數清單。管道解析程式要求映射範本評估的結果，會依 `$ctx.prev.result` 提供給第一個函數。每個函數輸出都會依 `$ctx.prev.result` 提供給下一個函數。

After 映射範本

管道解析程式的回應映射範本，或 After 步驟，可讓您從最後一個函數的輸出執行一些最終映射邏輯，進而達到預期的 GraphQL 欄位類型。在函數清單的最後一個函數的輸出，將依 `$ctx.prev.result` 或 `$ctx.result` 提供給管道解析程式映射範本。

執行流程

假設管道解析程式由兩個函數組成，以下清單代表呼叫解析程式時的執行流程：



1. 映射範本前的管道解析程式
2. 第 1 個函數：函數要求映射範本
3. 第 1 個函數：資料來源呼叫
4. 第 1 個函數：函數回應映射範本
5. 第 2 個函數：函數要求映射範本
6. 第 2 個函數：資料來源呼叫
7. 第 2 個函數：函數回應映射範本

8. 映射範本後管道解析程式

Note

管道解析程式執行流程為單向，並已在解析程式靜態定義。

實用的 Apache Velocity 範本語言 (VTL) 公用程式

隨著應用程式複雜性提高，VTL 公用程式和此處的指示詞可加速開發生產力。下列公用程式可在您使用管道解析程式時提供協助。

`$ctx.stash`

`stash` 是在每個解析程式和函數映射範本中 `Map` 可用的。在單一個解析程式執行期間，則會存在相同的 `stash` 執行個體。這表示您可以使用 `stash` 在管道解析程式中的所有要求和回應映射範本、以及全部函數中，傳遞任意資料。堆疊會公開與 [Java 地圖](#) 資料結構相同的方法。

`$ctx.prev.result`

`$ctx.prev.result` 代表管道解析程式中先前執行之操作的結果。

如果先前的操作是管道解析程式的映射前範本，則 `$ctx.prev.result` 代表範本評估的輸出，並可供管道中的第一個函數使用。如果先前操作是第一個函數，則 `$ctx.prev.result` 會顯示第一個函數的輸出，並將資料提供給管道中的第二個函數。如果先前的操作是最後一個函數，則 `$ctx.prev.result` 代表最後一個函數的輸出，並可供管道解析程式的映射後範本使用。

`#return(data: Object)`

當您需要從任何映射範本提前傳回時，這時使用 `#return(data: Object)` 指令就能完成。`#return(data: Object)` 類似於程式設計語言中的 `return` 關鍵字，因為它會從最靠近範圍的邏輯區塊傳回。這表示在解析程式映射範本中使用 `#return` 時，將從該解析程式傳回。在解析程式映射範本中使用 `#return(data: Object)`，將會設定 GraphQL 欄位上的 `data`。此外，使用函數映射範本的 `#return(data: Object)` 時會從該函數傳回，且執行將持續到該管道或是解析程式回應映射範本中的下一個函數。

`#return`

這與 `#return(data: Object)` 相同，但 `null` 會改為傳回。

\$util.error

`$util.error` 公用程式非常適合用來擲出欄位錯誤。在函數映射範本內使用 `$util.error` 會立即擲出錯誤，其可阻止後續函數執行。如需詳細資訊和其他 `$util.error` 簽章，請造訪[解析程式映射範本公用程式參考](#)。

\$util.appendError

`$util.appendError` 的功能類似於 `$util.error()`，主要的差別在於前者不會中斷映射範本的評估，而是在欄位出現錯誤時發出訊號，但允許範本的評估，進而將資料傳回。在函數中使用 `$util.appendError` 並不會中斷管道的執行流程。如需詳細資訊和其他 `$util.error` 簽章，請造訪[解析程式映射範本公用程式參考](#)。

範例 範本

假設您在名為 `posts` 的欄位上有 DynamoDB 資料來源和單位解析程式 `getPost(id:ID!)`，該欄位會傳回具有下列 GraphQL 查詢的 `Post` 類型：

```
getPost(id:1){
  id
  title
  content
}
```

解析程式範本看起來可能會類似於下列的範例：

```
{
  "version" : "2018-05-29",
  "operation" : "GetItem",
  "key" : {
    "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
  }
}
```

這會用 `id` 輸入的參數值 `1` 來取代 `${ctx.args.id}`，並產生下列 JSON：

```
{
  "version" : "2018-05-29",
  "operation" : "GetItem",
  "key" : {
    "id" : { "S" : "1" }
  }
}
```

```
}  
}
```

AWS AppSync 使用此範本來產生與 DynamoDB 通訊並取得資料（或執行其他適當操作）的指示。資料傳回後，AWS AppSync 會透過選用的回應映射範本執行它，您可以使用該範本來執行資料成形或邏輯。例如，當我們從 DynamoDB 取得結果時，結果看起來可能會如下所示：

```
{  
  "id" : 1,  
  "theTitle" : "AWS AppSync works offline!",  
  "theContent-part1" : "It also has realtime functionality",  
  "theContent-part2" : "using GraphQL"  
}
```

您可以利用下列的回應映射範本，來選擇將其中兩個欄位合併為單一欄位：

```
{  
  "id" : $util.toJson($context.data.id),  
  "title" : $util.toJson($context.data.theTitle),  
  "content" : $util.toJson("${context.data.theContent-part1}  
  ${context.data.theContent-part2}")  
}
```

在將範本套用到資料之後，資料的格式如下所示：

```
{  
  "id" : 1,  
  "title" : "AWS AppSync works offline!",  
  "content" : "It also has realtime functionality using GraphQL"  
}
```

這些資料會再做為回應傳回給用戶端，如下所示：

```
{  
  "data": {  
    "getPost": {  
      "id" : 1,  
      "title" : "AWS AppSync works offline!",  
      "content" : "It also has realtime functionality using GraphQL"  
    }  
  }  
}
```

```
}
```

請注意，在大部分情況下，回應映射範本是簡單的資料傳遞，主要差別在於您是傳回個別項目或項目清單。若是個別項目，則傳遞是：

```
$util.toJson($context.result)
```

若是清單，則傳遞通常是：

```
$util.toJson($context.result.items)
```

若要查看單元和管道解析程式的更多範例，請參閱[解析程式教學課程](#)。

評估的映射範本還原序列化規則

對於一個字串的對應範本評估。In AWS AppSync，輸出字串必須遵循 JSON 結構才有效。

此外，系統會強制執行下列還原序列化規則。

JSON 物件中不允許重複金鑰

如果已評估的對應範本字串代表 JSON 物件，或包含具有重複金鑰的物件，則對應範本會傳回下列錯誤訊息：

```
Duplicate field 'aField' detected on Object. Duplicate JSON keys are not allowed.
```

已評估要求對應範本中重複金鑰的範例：

```
{
  "version": "2018-05-29",
  "operation": "Invoke",
  "payload": {
    "field": "getPost",
    "postId": "1",
    "field": "getPost" ## key 'field' has been redefined
  }
}
```

若要修正此錯誤，請勿在 JSON 物件中重新定義金鑰。

JSON 物件中不允許結尾字元

如果已評估的對應範本字串代表 JSON 物件，且包含額外的結尾字元，則對應範本會傳回下列錯誤訊息：

Trailing characters at the end of the JSON string are not allowed.

已評估請求映射範本中結尾字元的範例：

```
{
  "version": "2018-05-29",
  "operation": "Invoke",
  "payload": {
    "field": "getPost",
    "postId": "1",
  }
}extraneouschars
```

若要修正此錯誤，請確保評估的範本嚴格評估為 JSON。

AWS AppSync 解析程式映射範本程式設計指南

Note

我們現在主要支援 APPSYNC_JS 執行期及其文件。請考慮[在此處](#)使用 APPSYNC_JS 執行期及其指南。

這是使用 Apache Velocity 範本語言 (VTL) 進程式設計的技術指南樣式教學課程 AWS AppSync。如果您熟悉其他程式設計語言 (例如 JavaScript、C 或 Java)，此做法應相當直覺。

AWS AppSync 使用 VTL 將來自用戶端的 GraphQL 請求轉換為對資料來源的請求。然後，它會反轉程序，來將資料來源轉換回 GraphQL 回應。VTL 是一種邏輯範本語言，可讓您使用下列技術，在 Web 應用程式的標準請求/回應流程中操作請求和回應：

- 新項目的預設值
- 輸入驗證和格式化
- 轉換與打造資料
- 逐一查看清單、映射和陣列以移出或更改值

- 根據使用者身分篩選條件/變更回應
- 複雜的授權檢查

例如，您可能想要在 GraphQL 引數的服務中執行電話號碼驗證，或在將輸入參數存放在 DynamoDB 之前將其轉換為大寫。或者，您希望用戶端系統能以 GraphQL 引數、JWT 字符或 HTTP 標頭的形式提供程式碼，並在程式碼符合清單中的特定字串時才回應資料。這些都是您可以使用 VTL in AWS AppSync 執行的邏輯檢查。

VTL 可讓您使用熟悉的程式設計技巧，來套用邏輯。不過，其受到限制，必須在標準請求/回應流程內執行，以確保 GraphQL API 可隨著您的使用者成長而擴展。Because AWS AppSync 也支援 AWS Lambda 做為解析程式，如果您需要更多彈性，您可以使用您選擇的程式設計語言 (Node.js、Python、Go、Java 等) 撰寫 Lambda 函數。

設定

學習語言的一項常用技巧是將結果列印出來 (例如，在 JavaScript 中的 `console.log(variable)`)，以查看發生的情況。在本教學中，我們透過建立簡單的 GraphQL 結構描述並將對應值傳遞到 Lambda 函式來示範此技巧。Lambda 函式會列印出值，然後加以回應。這可讓您了解請求/回應流程，並查看不同的程式設計技巧。

開始建立以下 GraphQL 結構描述：

```
type Query {  
  get(id: ID, meta: String): Thing  
}  
  
type Thing {  
  id: ID!  
  title: String!  
  meta: String  
}  
  
schema {  
  query: Query  
}
```

現在使用 Node.js 做為語言，建立下列 AWS Lambda 函數：

```
exports.handler = (event, context, callback) => {
```

```
console.log('VTL details: ', event);
callback(null, event);
};
```

在 AWS AppSync 主控台的資料來源窗格中，新增此 Lambda 函數做為新的資料來源。導覽回主控台的結構描述頁面，然後按一下 `get(...):Thing` 查詢旁邊的右側 ATTACH 按鈕。如需請求範本，從 `Invoke and forward arguments` (叫用和轉送引數) 功能表中選擇現有的範本。如需回應範本，請選擇 `Return Lambda result` (傳回 Lambda 結果)。

在一個位置開啟 Lambda 函數的 Amazon CloudWatch Logs，並從 AWS AppSync 主控台的查詢索引標籤執行下列 GraphQL 查詢：

```
query test {
  get(id:123 meta:"testing"){
    id
    meta
  }
}
```

GraphQL 回應應包含 `id:123` 和 `meta:testing`，因為 Lambda 函式會參考這兩者。幾秒鐘後，您也應該會在 CloudWatch Logs 中看到包含這些詳細資訊的記錄。

Variables

VTL 使用[參考](#)，供您用來存放或操作資料。在 VTL 中有三種類型的參考：變數、屬性和方法。變數的前面會有 `$` 符號且這些變數的建立是透過 `#set` 指令：

```
#set($var = "a string")
```

變數會使用其他語言存放您所熟悉的類似類型，例如數字、字串、陣列、清單和映射。您可能已經注意到傳送至預設請求範本以供 Lambda 解析程式使用的 JSON 承載：

```
"payload": $util.toJson($context.arguments)
```

這裡有幾件事要注意，首先，AWS AppSync 為常見操作提供數個便利功能。在這個範例中，`$util.toJson` 會將變數轉換為 JSON。其次，會自動從 GraphQL 請求填入變數 `$context.arguments` 做為映射物件。您可以建立新的映射，如下所示：

```
#set( $myMap = {
```

```
"id": $context.arguments.id,  
"meta": "stuff",  
"upperMeta" : $context.arguments.meta.toUpperCase()  
} )
```

現在，您可以建立名為 `$myMap` 的變數，其擁有 `id`、`meta` 以及 `upperMeta` 的金鑰。這也展現了以下幾點：

- 從 GraphQL 引數將金鑰填入 `id`。在 VTL 中從用戶端擷取引數是很常見的做法。
- 是以值來將 `meta` 硬式編碼，以展現預設值。
- `upperMeta` 會使用方法 `meta` 來轉換 `.toUpperCase()` 引數。

將之前的程式碼放在請求範本的頂部並變更 `payload` 以使用新的 `$myMap` 變數：

```
"payload": $util.toJson($myMap)
```

執行您的 Lambda 函式，您可以查看回應變更與在 CloudWatch 日誌中的此資料。在您逐步完成本教學的其餘部分，我們將保持填入 `$myMap` 如此您就可以執行類似的測試。

您也可以對變數設定 `properties_`。它們可以是簡單的字串、陣列或 JSON：

```
#set($myMap.myProperty = "ABC")  
#set($myMap.arrProperty = ["Write", "Some", "GraphQL"])  
#set($myMap.jsonProperty = {  
    "AppSync" : "Offline and Realtime",  
    "Cognito" : "AuthN and AuthZ"  
})
```

安靜參考

由於 VTL 是一種範本化語言，在預設情況下，您給予的每個參考將會執行 `.toString()`。如果參考是未定義的，它會列印實際參考呈現做為字串。例如：

```
#set($myValue = 5)  
##Prints '5'  
$myValue  
  
##Prints '$somethingelse'  
$somethingelse
```

為了解決這個問題，VTL 有安靜參考或無提示參考語法，可告訴範本引擎抑制這種行為。此語法是 `${}`。例如，如果我們稍微變更之前的程式碼以使用 `${!{somethingelse}}`，系統會抑制列印：

```
#set($myValue = 5)
##Prints '5'
$myValue

##Nothing prints out
${!{somethingelse}}
```

呼叫方法

在之前的範例中，我們示範了如何建立變數並同時設定值。您也可以如下所示將資料新增到映射，以兩個步驟執行此操作：

```
#set ($myMap = {})
#set ($myList = [])

##Nothing prints out
${!{myMap.put("id", "first value")}}
##Prints "first value"
${!{myMap.put("id", "another value")}}
##Prints true
${!{myList.add("something")}}
```

然而，有一些需了解這種行為的原因。雖然安靜參考表示法可讓您 `${!{}}` 如以上方式呼叫方法，它將不會抑制執行方法的傳回值。這也是為什麼我們會說明以上的 `##Prints "first value"` 及 `##Prints true`。您逐一查看映射或清單可能會導致錯誤，例如插入索引鍵已存在的值，因為輸出會在評估時將未預期的字串加入範本中。

此問題的解決方法有時會使用 `#set` 指令來呼叫方法並忽略變數。例如：

```
#set ($myMap = {})
#set($discard = $myMap.put("id", "first value"))
```

您可以在範本中使用此技術，因為它可防止在範本中列印出非預期的字串。AWS AppSync 提供替代的便利功能，以更簡潔的表示法提供相同的行為。這可讓您不需要考慮這些實作特性。您可以透過 `$util.quiet()` 或其別名 `$util.qr()` 存取此函式。例如：

```
#set ($myMap = {})
```

```
#set ($myList = [])

##Nothing prints out
$util.quiet($myMap.put("id", "first value"))
##Nothing prints out
$util.qr($myList.add("something"))
```

Strings

隨著使用多種程式設計語言，字串可能會變得難以處理，尤其是當您想要透過變數建構字串時。有一些與 VTL 相關的常見事件。

假設您要將資料作為字串插入 DynamoDB 之類的資料來源，但會從變數填入資料，例如 GraphQL 引數。字串將具有雙引號，並在您只需要的字串中參考變數 "\${}" (因此，在[安靜參考表示法](#)中不會!有)。這與 JavaScript 中的範本常值類似：https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals

```
#set($firstname = "Jeff")
${myMap.put("Firstname", "${firstname}")}
```

您可以在 DynamoDB 請求範本中看到此內容，例如使用 GraphQL 用戶端的引數 "author"：{ "S" : "\${context.arguments.author}" } 時，或自動產生 ID 時，例如 "id" : { "S" : "\${util.autoId()}"}。這表示您可以參考變數或字串內部的方法結果來填入資料。

您也可以使用 Java [字串類別](#) 的公有方法 (例如取出子字串)：

```
#set($bigstring = "This is a long string, I want to pull out everything after the comma")
#set ($comma = $bigstring.indexOf(','))
#set ($comma = $comma +2)
#set ($substring = $bigstring.substring($comma))

$util.qr($myMap.put("substring", "${substring}"))
```

字串連接也是非常常見的任務。您可以單獨使用變數參考或使用變數參考搭配靜態值來這樣做：

```
#set($s1 = "Hello")
#set($s2 = " World")

$util.qr($myMap.put("concat", "${s1}${s2}"))
$util.qr($myMap.put("concat2", "Second ${s1} World"))
```

迴圈

現在您已建立變數和呼叫方法，您可以將一些邏輯到新增到程式碼。與其他語言不同，VTL 只允許迴圈，其中反覆次數是預先決定的。在速度中沒有 `do..while`。此設計可確保評估處理一律終止，並提供 GraphQL 操作執行時的擴充界限。

迴圈是使用 `#foreach` 所建立並需要您輸入迴圈變數和 iterable 物件 (例如陣列、清單、映射或集合)。 `#foreach` 迴圈的典型程式設計範例，即是循環執行集合中的項目並將他們印出，因此我們的案例中，我們將這些項目移出，並將他們新增至映射：

```
#set($start = 0)
#set($end = 5)
#set($range = [$start..$end])

#foreach($i in $range)
  ##$util.qr($myMap.put($i, "abc"))
  ##$util.qr($myMap.put($i, $i.toString()+"foo")) ##Concat variable with string
  $util.qr($myMap.put($i, "{$i}foo"))      ##Reference a variable in a string with
  "{$varname}"
#end
```

此範例顯示一些考量。第一種是使用變數與範圍 `[..]` 運算子來建立 iterable 物件。然後，您可以操作的變數 `$i` 會參考每個項目。在上述範例中，您還可以查看註解，註解會加上雙井字號 `##` 來表示。這也展示在金鑰或值兩者中使用迴圈變數，以及使用字串的不同串連方法。

請注意，`$i` 是一個整數，因此您可以呼叫 `.toString()` 方法。若是 INT 的 GraphQL 類型，此技巧非常實用。

您也可以直接使用各種操作，例如：

```
#foreach($item in [1..5])
  ...
#end
```

陣列

您到目前為止已完成映射的操控，但在 VTL 中陣列也相當常見。您也可以使用陣列存取一些基本方法，例如 `.isEmpty()`、`.size()`、`.set()`、`.get()` 和 `.add()`，如下所示：

```
#set($array = [])
```

```
#set($idx = 0)

##adding elements
$util.qr($array.add("element in array"))
$util.qr($myMap.put("array", $array[$idx]))

##initialize array vals on create
#set($arr2 = [42, "a string", 21, "test"])

$util.qr($myMap.put("arr2", $arr2[$idx]))
$util.qr($myMap.put("isEmpty", $array.isEmpty())) ##isEmpty == false
$util.qr($myMap.put("size", $array.size()))

##Get and set items in an array
$util.qr($myMap.put("set", $array.set(0, 'changing array value')))
$util.qr($myMap.put("get", $array.get(0)))
```

上一個範例使用陣列索引表示法來擷取具有的元素 `arr2[$idx]`。您可以透過類似的方式從 Map/字典中來查詢名稱：

```
#set($result = {
  "Author" : "Nadia",
  "Topic" : "GraphQL"
})

$util.qr($myMap.put("Author", $result["Author"]))
```

在使用條件時，在回應範本中篩選來自資料來源的結果是很常見的。

條件式檢查

`#foreach` 的舊區段說明使用邏輯來透過 VTL 轉換資料的部分範例。您也可以套用條件式檢查，以在執行時間評估資料：

```
#if(!$array.isEmpty())
  $util.qr($myMap.put("ifCheck", "Array not empty"))
#else
  $util.qr($myMap.put("ifCheck", "Your array is empty"))
#end
```

上述布林值表達式 `#if()` 的檢查是精細的，但您也可以使用運算子和 `#elseif()` 來進行分支：

```
#if ($arr2.size() == 0)
    $util.qr($myMap.put("elseifCheck", "You forgot to put anything into this array!"))
#elseif ($arr2.size() == 1)
    $util.qr($myMap.put("elseifCheck", "Good start but please add more stuff"))
#else
    $util.qr($myMap.put("elseifCheck", "Good job!"))
#end
```

這兩個範例示範 `negation(!)` 和 `equality(==)`。我們也可以使用 `||`、`&&`、`>`、`<`、`>=`、`<=` 和 `!=`。

```
#set($T = true)
#set($F = false)

#if ($T || $F)
    $util.qr($myMap.put("OR", "TRUE"))
#end

#if ($T && $F)
    $util.qr($myMap.put("AND", "TRUE"))
#end
```

注意：在條件中只有 `Boolean.FALSE` 和 `null` 會被視為 `false`。零 (0) 和空白字串 ("") 不等同於 `false`。

運算子

程式設計語言需要運算子來執行數學動作才得以完整。有幾種簡單的入門方式：

```
#set($x = 5)
#set($y = 7)
#set($z = $x + $y)
#set($x-y = $x - $y)
#set($xy = $x * $y)
#set($xDIVy = $x / $y)
#set($xMODy = $x % $y)

$util.qr($myMap.put("z", $z))
$util.qr($myMap.put("x-y", $x-y))
$util.qr($myMap.put("x*y", $xy))
$util.qr($myMap.put("x/y", $xDIVy))
$util.qr($myMap.put("x|y", $xMODy))
```


同時使用迴圈和條件

這在 VTL 中轉換資料很常見 (例如在從資料來源寫入或讀取前)，以循環執行物件，然後執行檢查，再執行動作。將之前區段的一些工具進行整合，讓您有更多功能可使用。一個便利的工具是知道 `#foreach` 會自動提供您每個項目的 `.count`：

```
#foreach ($item in $arr2)
  #set($idx = "item" + $foreach.count)
  $util.qr($myMap.put($idx, $item))
#end
```

例如，如果值小於特定大小，則也許您會希望將其從映射移出。使用計數以及條件和 `#break` 陳述式可讓您執行此操作：

```
#set($hashmap = {
  "DynamoDB" : "https://aws.amazon.com/dynamodb/",
  "Amplify" : "https://github.com/aws/aws-amplify",
  "DynamoDB2" : "https://aws.amazon.com/dynamodb/",
  "Amplify2" : "https://github.com/aws/aws-amplify"
})

#foreach ($key in $hashmap.keySet())
  #if($foreach.count > 2)
    #break
  #end
  $util.qr($myMap.put($key, $hashmap.get($key)))
#end
```

先前 `#foreach` 已透過 `.keySet()` (您可以在映射上使用此項目) 來重複使用。這讓您能夠取得 `$key` 並使用 `.get($key)` 來參考該值。來自用戶端 in AWS AppSync 的 GraphQL 引數會儲存為映射。也可以透過 `.entrySet()` (您可以同時存取金鑰和值做為 Set) 來重複使用這些項目，並填入其他變數或執行複雜條件式檢查 (例如驗證或輸入的轉換)：

```
#foreach( $entry in $context.arguments.entrySet() )
#if ($entry.key == "XYZ" && $entry.value == "BAD")
  #set($myvar = "...")
#else
  #break
#end
#end
```

其他常見範例會自動填入預設資訊，例如同步資料時的初始物件版本（在衝突解決中非常重要）或授權檢查物件的預設擁有者 - Mary 建立此部落格文章，因此：

```
#set($myMap.owner = "Mary")
#set($myMap.defaultOwners = ["Admins", "Editors"])
```

Context

現在您已更熟悉在具有 VTL 的解析程式中 AWS AppSync 執行邏輯檢查，請查看內容物件：

```
$util.qr($myMap.put("context", $context))
```

這包含所有資訊，您可以在 GraphQL 請求中存取這些資訊。有關詳細說明，請參閱[內容參考](#)。

篩選

此教學到目前為止，會透過非常簡單的 JSON 轉換，將 Lambda 函式的所有資訊傳回到 GraphQL 查詢：

```
$util.toJson($context.result)
```

VTL 邏輯就跟您從資料來源取得回應時一樣強大，尤其是在對回應進行授權檢查時。讓我們逐步介紹一些範例。首先，嘗試如下變更回應範本：

```
#set($data = {
  "id" : "456",
  "meta" : "Valid Response"
})

$util.toJson($data)
```

無論您對 GraphQL 操作進行何種操作，系統會將硬式編碼值傳回用戶端。稍微變更此項目，以從 Lambda 回應填入 meta 欄位，如需了解條件時則在教學課程中以 `elseifCheck` 值來較早設立此值：

```
#set($data = {
  "id" : "456"
})

#foreach($item in $context.result.entrySet())
  #if($item.key == "elseifCheck")
    $util.qr($data.put("meta", $item.value))
```

```
#end
#end

$util.toJson($data)
```

`$context.result` 是一種映射，因此您可以使用 `entrySet()` 來對金鑰或傳回的值執行邏輯。由於 `$context.identity` 包含使用者執行 GraphQL 操作的相關資訊，如果您從資料來源傳回授權資訊，則可以根據邏輯決定是否向使用傳回全部、部分資料或不傳回資料。將您的回應範本變更為如下所示：

```
#if($context.result["id"] == 123)
    $util.toJson($context.result)
#else
    $util.unauthorized()
#end
```

如果您執行 GraphQL 查詢，系統會以一般的方式傳回。不過，如果您將 `id` 引數變更 123 以外的值 (query test { get(id:456 meta:"badrequest"){ } })，您會收到授權失敗的訊息。

您可以在[授權使用案例](#)小節中找到更多授權案例的範例。

範本範例

若您遵循此教學至今，您可以逐步建立此範本。若您並未遵循此教學，我們已在下方提供，可讓您複製並進行測試。

請求範本

```
#set( $myMap = {
    "id": $context.arguments.id,
    "meta": "stuff",
    "upperMeta" : "$context.arguments.meta.toUpperCase()"
} )

##This is how you would do it in two steps with a "quiet reference" and you can use it
for invoking methods, such as .put() to add items to a Map
#set ($myMap2 = {})
$util.qr($myMap2.put("id", "first value"))

## Properties are created with a dot notation
#set($myMap.myProperty = "ABC")
#set($myMap.arrProperty = ["Write", "Some", "GraphQL"])
```

```

#set($myMap.jsonProperty = {
    "AppSync" : "Offline and Realtime",
    "Cognito" : "AuthN and AuthZ"
})

##When you are inside a string and just have ${} without ! it means stuff inside curly
braces are a reference
#set($firstname = "Jeff")
$util.qr($myMap.put("Firstname", "${firstname}"))

#set($bigstring = "This is a long string, I want to pull out everything after the
comma")
#set ($comma = $bigstring.indexOf(','))
#set ($comma = $comma +2)
#set ($substring = $bigstring.substring($comma))
$util.qr($myMap.put("substring", "${substring}"))

##Classic for-each loop over N items:
#set($start = 0)
#set($end = 5)
#set($range = [$start..$end])
#foreach($i in $range)          ##Can also use range operator directly like
    #foreach($item in [1...5])
        ##$util.qr($myMap.put($i, "abc"))
        ##$util.qr($myMap.put($i, $i.toString()+"foo")) ##Concat variable with string
        $util.qr($myMap.put($i, "${i}foo"))      ##Reference a variable in a string with
        "${varname}"
    #end

##Operators don't work
#set($x = 5)
#set($y = 7)
#set($z = $x + $y)
#set($x-y = $x - $y)
#set($xy = $x * $y)
#set($xDIVy = $x / $y)
#set($xMODy = $x % $y)
$util.qr($myMap.put("z", $z))
$util.qr($myMap.put("x-y", $x-y))
$util.qr($myMap.put("x*y", $xy))
$util.qr($myMap.put("x/y", $xDIVy))
$util.qr($myMap.put("x|y", $xMODy))

##arrays

```

```

#set($array = ["first"])
#set($idx = 0)
$util.qr($myMap.put("array", $array[$idx]))
##initialize array vals on create
#set($arr2 = [42, "a string", 21, "test"])
$util.qr($myMap.put("arr2", $arr2[$idx]))
$util.qr($myMap.put("isEmpty", $array.isEmpty())) ##Returns false
$util.qr($myMap.put("size", $array.size()))
##Get and set items in an array
$util.qr($myMap.put("set", $array.set(0, 'changing array value')))
$util.qr($myMap.put("get", $array.get(0)))

##Lookup by name from a Map/dictionary in a similar way:
#set($result = {
    "Author" : "Nadia",
    "Topic" : "GraphQL"
})
$util.qr($myMap.put("Author", $result["Author"]))

##Conditional examples
#if(!$array.isEmpty())
$util.qr($myMap.put("ifCheck", "Array not empty"))
#else
$util.qr($myMap.put("ifCheck", "Your array is empty"))
#end

#if ($arr2.size() == 0)
$util.qr($myMap.put("elseifCheck", "You forgot to put anything into this array!"))
#elseif ($arr2.size() == 1)
$util.qr($myMap.put("elseifCheck", "Good start but please add more stuff"))
#else
$util.qr($myMap.put("elseifCheck", "Good job!"))
#end

##Above showed negation(!) and equality (==), we can also use OR, AND, >, <, >=, <=,
and !=
#set($T = true)
#set($F = false)
#if ($T || $F)
    $util.qr($myMap.put("OR", "TRUE"))
#end

#if ($T && $F)

```

```

    $util.qr($myMap.put("AND", "TRUE"))
#end

##Using the foreach loop counter - $foreach.count
#foreach ($item in $arr2)
    #set($idx = "item" + $foreach.count)
    $util.qr($myMap.put($idx, $item))
#end

##Using a Map and plucking out keys/vals
#set($hashmap = {
    "DynamoDB" : "https://aws.amazon.com/dynamodb/",
    "Amplify" : "https://github.com/aws/aws-amplify",
    "DynamoDB2" : "https://aws.amazon.com/dynamodb/",
    "Amplify2" : "https://github.com/aws/aws-amplify"
})

#foreach ($key in $hashmap.keySet())
    #if($foreach.count > 2)
        #break
    #end
    $util.qr($myMap.put($key, $hashmap.get($key)))
#end

##concatenate strings
#set($s1 = "Hello")
#set($s2 = " World")
$util.qr($myMap.put("concat", "$s1$s2"))
$util.qr($myMap.put("concat2", "Second $s1 World"))

$util.qr($myMap.put("context", $context))

{
    "version" : "2017-02-28",
    "operation": "Invoke",
    "payload": $util.toJson($myMap)
}

```

回應範本

```

#set($data = {
    "id" : "456"
})

```

```
#foreach($item in $context.result.entrySet())  ##$context.result is a MAP so we use
  entrySet()
    #if($item.key == "ifCheck")
      $util.qr($data.put("meta", "$item.value"))
    #end
#end

##Uncomment this out if you want to test and remove the below #if check
##$util.toJson($data)

#if($context.result["id"] == 123)
  $util.toJson($context.result)
#else
  $util.unauthorized()
#end
```

AWS AppSync 解析程式映射範本內容參考

Note

我們現在主要支援 APPSYNC_JS 執行期及其文件。請考慮[在此處](#)使用 APPSYNC_JS 執行期及其指南。

AWS AppSync 定義一組變數和函數，用於使用解析程式映射範本。這樣可讓使用 GraphQL 對資料進行邏輯運算更簡單。本文件說明這些函式，並提供範本使用範例。

正在存取 **\$context**

\$context 變數是一種映射，保存您的解析程式叫用的所有情境資訊。其結構如下：

```
{
  "arguments" : { ... },
  "source" : { ... },
  "result" : { ... },
  "identity" : { ... },
  "request" : { ... },
  "info": { ... }
}
```

Note

如果您嘗試透過索引鍵存取字典/地圖項目（例如中的項目context）來擷取值，則 Velocity 範本語言 (VTL) 可讓您直接使用標記法 `<dictionary-element>.<key-name>`。不過，這可能不是所有案例皆適用，例如當索引鍵名稱有特殊字元（例如，底線「_」）。建議您一律使用 `<dictionary-element>.get("<key-name>")` 表示法。

\$context 映射中每個欄位的定義如下：

\$context 欄位**arguments**

包含此欄位所有 GraphQL 引數的映射。

identity

包含有關發起人資訊的物件。如需此欄位結構的詳細資訊，請參閱[身分](#)。

source

包含父欄位解析度的映射。

stash

此 stash 是每個解析程式和函數映射範本中都會提供的「映射」。在單一個解析程式執行期間，則會存在相同的 stash 執行個體。這表示您可以使用 stash 在管道解析程式中的所有要求和回應映射範本、以及全部函數中，傳遞任意資料。此 stash 會公開與 [Java 映射](#) 資料結構相同的方法。

result

此解析程式結果的容器。此欄位僅適用於回應映射範本。

例如，如果您要解決下列查詢的 author 欄位：

```
query {
  getPost(id: 1234) {
    postId
    title
    content
    author {
```



```

        id
        name
      }
    }
  }
}

```

那麼，在處理回應映射範本時，可用的完整 `$context` 變數可能是：

```

{
  "arguments" : {
    id: "1234"
  },
  "source": {},
  "result" : {
    "postId": "1234",
    "title": "Some title",
    "content": "Some content",
    "author": {
      "id": "5678",
      "name": "Author Name"
    }
  },
  "identity" : {
    "sourceIp" : ["x.x.x.x"],
    "userArn" : "arn:aws:iam::123456789012:user/appsync",
    "accountId" : "666666666666",
    "user" : "AIDAAAAAAAAAAAAAAAAAAAA"
  }
}

```

prev.result

在管道解析程式中執行任何先前操作的結果。

如果先前的操作是管道解析程式的映射前範本，則 `$ctx.prev.result` 代表範本評估的輸出，並可供管道中的第一個函數使用。

如果先前操作是第一個函數，則 `$ctx.prev.result` 會顯示第一個函數的輸出，並將資料提供給管道中的第二個函數。

如果先前的操作是最後一個函數，則 `$ctx.prev.result` 代表最後一個函數的輸出，並可供管道解析程式的映射後範本使用。

info

包含有關 GraphQL 請求資訊的物件。如需此欄位的結構，請參閱[資訊](#)。

Identity

包含有關發起人資訊的 `identity` 區段。本節的形狀取決於 AWS AppSync API 的授權類型。

如需 AWS AppSync 安全選項的詳細資訊，請參閱[授權和身分驗證](#)。

API_KEY 授權

系統不會填入 `identity` 欄位。

AWS_LAMBDA 授權

`identity` 包含 `resolverContext` 金鑰，其中包含由授權請求的 Lambda 函數傳回的相同 `resolverContext` 內容。

AWS_IAM 授權

`identity` 有下列形式：

```
{
  "accountId" : "string",
  "cognitoIdentityPoolId" : "string",
  "cognitoIdentityId" : "string",
  "sourceIp" : ["string"],
  "username" : "string", // IAM user principal
  "userArn" : "string",
  "cognitoIdentityAuthType" : "string", // authenticated/unauthenticated based on
the identity type
  "cognitoIdentityAuthProvider" : "string" // the auth provider that was used to
obtain the credentials
}
```

AMAZON_COGNITO_USER_POOLS 授權

`identity` 有下列形式：

```
{
  "sub" : "uuid",
```

```
"issuer" : "string",
"username" : "string"
"claims" : { ... },
"sourceIp" : ["x.x.x.x"],
"defaultAuthStrategy" : "string"
}
```

每個欄位的定義如下：

accountId

發起人 AWS 的帳戶 ID。

claims

使用者擁有的宣告。

cognitoIdentityAuthType

經身分驗證或未經身分驗證 (根據身分類型)。

cognitoIdentityAuthProvider

以逗號分隔的外部身分提供者資訊清單，用於取得用於簽署請求的登入資料。

cognitoIdentityId

發起人的 Amazon Cognito 身分 ID。

cognitoIdentityPoolId

與發起人相關聯的 Amazon Cognito 身分集區 ID。

defaultAuthStrategy

此發起人 (ALLOW 或 DENY) 的預設授權策略。

issuer

字符發行者。

sourceIp

AWS AppSync 接收的發起人的來源 IP 地址。如果請求不包含 x-forwarded-for 標頭，來源 IP 值只會包含 TCP 連線的單一 IP 地址。如果要求包含 x-forwarded-for 標頭，則來源 IP 除了有 TCP 連線的 IP 地址外，也將有 x-forwarded-for 標頭中 IP 地址的清單。

sub

已驗證使用者的 UUID。

user

IAM 使用者。

userArn

IAM 使用者的 Amazon Resource Name (ARN)。

username

已驗證使用者的使用者名稱。如果是 AMAZON_COGNITO_USER_POOLS 授權，使用者名稱的值是屬性 cognito:username 的值。在 AWS_IAM 授權的情況下，使用者名稱的值是 AWS 使用者主體的值。如果您使用 IAM 授權搭配來自 Amazon Cognito 身分集區的憑證，我們建議您使用 cognitoIdentityId。

存取請求標頭

AWS AppSync 支援從用戶端傳遞自訂標頭，並使用在 GraphQL 解析程式中存取它們 `$context.request.headers`。然後，您可以將標頭值用於動作，例如將資料插入資料來源或授權檢查。您可以將與來自命令列的 API 金鑰 `$curl` 搭配使用的一或多個請求標頭，如下列範例所示：

單一標頭範例

假設您設定使用 custom 值的 nadia 標頭，如下所示：

```
curl -XPOST -H "Content-Type:application/graphql" -H "custom:nadia" -H "x-api-key:<API-KEY-VALUE>" -d '{"query":"mutation { createEvent(name: \"demo\", when: \"Next Friday! \", where: \"Here!\") {id name when where description}}"}' https://<ENDPOINT>/graphql
```

可使用 `$context.request.headers.custom` 對此進行存取。例如，它可能位於 DynamoDB 的下列 VTL 中：

```
"custom": $util.dynamodb.toDynamoDBJson($context.request.headers.custom)
```

多個標頭範例

您也可以將多個標頭傳遞到單一要求，並在解析程式映射範本中存取這些要求。例如，如果 custom 標頭設定為兩個值：

```
curl -XPOST -H "Content-Type:application/graphql" -H "custom:bailey" -H "custom:nadia"
-H "x-api-key:<API-KEY-VALUE>" -d '{"query":"mutation { createEvent(name: \"demo
\", when: \"Next Friday!\", where: \"Here!\") {id name when where description}}}'
https://<ENDPOINT>/graphql
```

然後，您可以將這些存取為陣列，例如 `$context.request.headers.custom[1]`。

Note

AWS AppSync 不會在 中公開 Cookie 標頭 `$context.request.headers`。

存取請求自訂網域名稱

AWS AppSync 支援設定自訂網域，您可以用來存取 APIs 的 GraphQL 和即時端點。使用自訂網域名稱提出請求時，您可以使用 取得網域名稱 `$context.request.domainName`。

使用預設 GraphQL 端點網域名稱時，值為 `null`。

Info

`info` 區段包含有關 GraphQL 請求的資訊。本節有下列形式：

```
{
  "fieldName": "string",
  "parentTypeName": "string",
  "variables": { ... },
  "selectionSetList": ["string"],
  "selectionSetGraphQL": "string"
}
```

每個欄位的定義如下：

fieldName

目前正在解析的欄位名稱。

parentTypeName

目前正在解析的欄位父類型名稱。

variables

包含傳遞給 GraphQL 請求之所有變數的映射。

selectionSetList

此清單表示 GraphQL 選取範圍中的欄位。別名的欄位僅由別名名稱參考，而非欄位名稱。以下範例詳細說明這一點。

selectionSetGraphQL

此字串表示格式為 GraphQL 結構描述定義語言 (SDL) 的選取範圍。雖然片段不會合併至選取集，但會保留內嵌片段，如下列範例所示。

Note

在 `$utils.toJson()` 上使用 `context.info`，預設不會序列化 `selectionSetGraphQL` 和 `selectionSetList` 傳回的值。

例如，如果您解析下列查詢的 `getPost` 欄位：

```
query {  
  getPost(id: $postId) {  
    postId  
    title  
    secondTitle: title  
    content  
    author(id: $authorId) {  
      authorId  
      name  
    }  
    secondAuthor(id: "789") {  
      authorId  
    }  
    ... on Post {  
      inlineFrag: comments: {  
        id  
      }  
    }  
    ... postFrag  
  }  
}
```

```
fragment postFrag on Post {
  postFrag: comments: {
    id
  }
}
```

那麼，在處理映射範本時，可用的完整 `$context.info` 變數可能是：

```
{
  "fieldName": "getPost",
  "parentTypeName": "Query",
  "variables": {
    "postId": "123",
    "authorId": "456"
  },
  "selectionSetList": [
    "postId",
    "title",
    "secondTitle",
    "content",
    "author",
    "author/authorId",
    "author/name",
    "secondAuthor",
    "secondAuthor/authorId",
    "inlineFragComments",
    "inlineFragComments/id",
    "postFragComments",
    "postFragComments/id"
  ],
  "selectionSetGraphQL": "{\n  getPost(id: $postId) {\n    postId\n    title\n    secondTitle: title\n    content\n    author(id: $authorId) {\n      authorId\n      name\n    }\n    secondAuthor(id: \"789\") {\n      authorId\n    }\n    ... on Post\n    {\n      inlineFrag: comments {\n        id\n      }\n    }\n    ... postFrag\n  }\n}"
}
```

`selectionSetList` 只會公開屬於目前類型的欄位。如果目前類型是界面或聯集，則只會公開屬於界面的所選欄位。例如，指定下列結構描述：

```
type Query {
  node(id: ID!): Node
}
```

```
interface Node {
  id: ID!
}

type Post implements Node {
  id: ID!
  title: String!
  author: String!
}

type Blog implements Node {
  id: ID!
  title: String!
  category: String!
}
```

以及下列查詢：

```
query {
  node(id: "post1") {
    id
    ... on Post {
      title
    }
    ... on Blog {
      title
    }
  }
}
```

`$ctx.info.selectionSetList` 以 `Query.node` 欄位解析度呼叫時，只會 `id` 公開：

```
"selectionSetList": [
  "id"
]
```

處理輸入

應用程式必須處理不受信任的輸入，以防止任何外部方使用其預期用途之外的應用程式。由於包含 `$context.arguments`、`$context.info.variables`、`$context.identity`

`$context.result`和 等屬性中的`$context`使用者輸入`$context.request.headers`，因此必須小心在映射範本中淨化其值。

由於對應範本代表 JSON，輸入處理採用從表示使用者輸入的字串逸出 JSON 保留字元的形式。將敏感字串值放入對應範本時，最佳做法是使用 `$util.toJson()` 公用程式將 JSON 保留字元從敏感字串值逸出。

例如，在下列 Lambda 請求映射範本中，因為我們存取了不安全的客戶輸入字串 (`$context.arguments.id`)，因此我們使用 包裝它`$util.toJson()`，以防止未逸出的 JSON 字元破壞 JSON 範本。

```
{
  "version": "2017-02-28",
  "operation": "Invoke",
  "payload": {
    "field": "getPost",
    "postId": $util.toJson($context.arguments.id)
  }
}
```

與下面的映射範本相反，我們在其中直接插入`$context.arguments.id`而不進行淨化。這不適用於包含未逸出引號或其他 JSON 預留字元的字串，並且可能會讓您的範本處於開啟失敗狀態。

```
## DO NOT DO THIS
{
  "version": "2017-02-28",
  "operation": "Invoke",
  "payload": {
    "field": "getPost",
    "postId": "$context.arguments.id" ## Unsafe! Do not insert $context string
    values without escaping JSON characters.
  }
}
```

AWS AppSync 解析程式映射範本公用程式參考

Note

我們現在主要支援 APPSYNC_JS 執行期及其文件。請考慮[在此處](#)使用 APPSYNC_JS 執行期及其指南。

AWS AppSync 定義一組公用程式，您可以在 GraphQL 解析程式中用來簡化與資料來源的互動。其中一些公用程式用於任何資料來源的一般用途，例如產生 IDs 或時間戳記。其他則是特定於一種資料來源類型。下列公用程式可供使用：

- [\\$util 中的公用程式協助程式](#) - \$util 變數包含可協助您處理資料的一般公用程式方法。除非另行指定，否則所有公用程式皆使用 UTF-8 字元集。
- [AppSync 指令](#) - AppSync 公開指令，以促進開發人員在 VTL 中寫入時的生產力。
- [\\$util.time 中的時間協助程式](#) - \$util.time 變數包含日期時間方法，可協助產生時間戳記、在日期時間格式之間轉換，以及剖析日期時間字串。日期時間格式的語法是以 [DateTimeFormatter](#) 為基礎，您可以參考以取得進一步的文件。
- 在 [\\$util.list 中列出協助程式](#) - \$util.list 包含有助於常見清單操作的方法，例如從清單中移除或保留項目以篩選使用案例。
- [\\$util.map - \\$util.map 中的映射協助程式](#) 包含有助於常見映射操作的方法，例如從映射中移除或保留項目以篩選使用案例。
- [\\$util.dynamodb 中的 DynamoDB 協助程式](#) - \$util.dynamodb 包含協助程式方法，可讓您更輕鬆地將資料寫入和讀取至 Amazon DynamoDB，例如自動類型映射和格式化。
- [\\$util.rds - \\$util.rds 中的 Amazon RDS 協助程式](#) 包含協助程式方法，可透過消除結果輸出中的外部資料來格式化 RDS 操作。
- [\\$util.http 中的 HTTP 協助程式](#) - \$util.http 公用程式提供可用於管理 HTTP 請求參數和新增回應標頭的協助程式方法。
- [\\$util.xml - \\$util.xml 中的 XML 協助程式](#) 包含協助程式方法，可讓您更輕鬆地將 XML 回應轉譯為 JSON 或字典。
- [\\$util.transform 中的轉換協助程式](#) - \$util.transform 包含協助程式方法，可讓您更輕鬆地對資料來源執行複雜的操作，例如 DynamoDB 篩選操作。
- [\\$util.math - \\$util.math 中的數學協助程式](#) 包含有助於常見數學操作的方法。
- [\\$util.str - \\$util.str 中的字串協助程式](#) 包含有助於常見字串操作的方法。
- [延伸](#) - \$extensions 包含一組方法，可在解析程式中執行其他動作。

\$util 中的公用程式協助程式

Note

我們現在主要支援 APPSYNC_JS 執行期及其文件。請考慮[在此處](#)使用 APPSYNC_JS 執行期及其指南。

`$util` 變數包含可協助您處理資料的一般公用程式方法。除非另行指定，否則所有公用程式皆使用 UTF-8 字元集。

JSON 剖析公用程式

JSON 剖析公用程式清單

`$util.parseJson(String) : Object`

採用「字串化」JSON，並傳回結果的物件呈現。

`$util.toJson(Object) : String`

採用物件，並傳回該物件的「字串化」JSON 呈現。

編碼 utils

編碼 utils 清單

`$util.urlEncode(String) : String`

以 `application/x-www-form-urlencoded` 編碼字串的形式傳回輸入字串。

`$util.urlDecode(String) : String`

將 `application/x-www-form-urlencoded` 編碼的字串解碼回非編碼格式。

`$util.base64Encode( byte[] ) : String`

將輸入編碼為 base64 編碼字串。

`$util.base64Decode(String) : byte[]`

解碼 base64 編碼字串中的資料。

ID 產生 utils

ID 產生 utils 清單

`$util.autoId() : String`

傳回 128 位元隨機產生的 UUID。

`$util.autoUlid() : String`

傳回 128 位元隨機產生的 ULID（通用可排序序列識別符）。

\$util.autoKsuid() : String

傳回 128 位元隨機產生的 KSUID (K-Sortable Unique Identifier) base62，編碼為長度為 27 的字串。

錯誤率

錯誤使用率清單

\$util.error(String)

擲回自訂錯誤。在請求或回應映射範本中使用此參數來偵測請求或調用結果的錯誤。

\$util.error(String, String)

擲回自訂錯誤。在請求或回應映射範本中使用此參數來偵測請求或調用結果的錯誤。您也可以指定 `errorType`。

\$util.error(String, String, Object)

擲回自訂錯誤。在請求或回應映射範本中使用此參數來偵測請求或調用結果的錯誤。您也可以指定 `errorType` 和 `data` 欄位。`data` 值將新增到 GraphQL 回應中，`error` 內對應的 `errors` 區塊。

Note

`data` 將根據查詢選取集進行篩選。

\$util.error(String, String, Object, Object)

擲回自訂錯誤。如果範本偵測到要求或呼叫結果的錯誤，您可以將此用於要求或回應映射範本。此外，還可以指定 `errorType` 欄位、`data` 欄位和 `errorInfo` 欄位。`data` 值將新增到 GraphQL 回應中，`error` 內對應的 `errors` 區塊。

Note

`data` 將根據查詢選取集進行篩選。`errorInfo` 值將新增到 GraphQL 回應中，`error` 內對應的 `errors` 區塊。

`errorInfo` 將不會根據查詢選取集進行篩選。

\$util.appendError(String)

附加自訂錯誤。如果範本偵測到要求或呼叫結果的錯誤，您可以將此用於要求或回應映射範本。與 `$util.error(String)` 不同的是，範本評估不會受中斷，因此可以將資料傳回給發起人。

\$util.appendError(String, String)

附加自訂錯誤。如果範本偵測到要求或呼叫結果的錯誤，您可以將此用於要求或回應映射範本。您同時可以指定 `errorType`。與 `$util.error(String, String)` 不同的是，範本評估不會受中斷，因此可以將資料傳回給發起人。

\$util.appendError(String, String, Object)

附加自訂錯誤。如果範本偵測到要求或呼叫結果的錯誤，您可以將此用於要求或回應映射範本。您同時可以指定 `errorType` 和 `data` 欄位。與 `$util.error(String, String, Object)` 不同的是，範本評估不會受中斷，因此可以將資料傳回給發起人。`data` 值將新增到 GraphQL 回應中，`error` 內對應的 `errors` 區塊。

Note

`data` 將根據查詢選取集進行篩選。

\$util.appendError(String, String, Object, Object)

附加自訂錯誤。如果範本偵測到要求或呼叫結果的錯誤，您可以將此用於要求或回應映射範本。此外，還可以指定 `errorType` 欄位、`data` 欄位和 `errorInfo` 欄位。與 `$util.error(String, String, Object, Object)` 不同的是，範本評估不會受中斷，因此可以將資料傳回給發起人。`data` 值將新增到 GraphQL 回應中，`error` 內對應的 `errors` 區塊。

Note

`data` 將根據查詢選取集進行篩選。`errorInfo` 值將新增到 GraphQL 回應中，`error` 內對應的 `errors` 區塊。

`errorInfo` 將不會根據查詢選取集進行篩選。

條件驗證公用程式

條件驗證公用程式清單

\$util.validate(Boolean, String) : void

如果條件為假，會擲回含指定訊息的 CustomTemplateException。

\$util.validate(Boolean, String, String) : void

如果條件為假，會擲回含指定訊息和錯誤類型的 CustomTemplateException。

\$util.validate(Boolean, String, String, Object) : void

如果條件為假，會擲回含指定訊息、錯誤類型以及在回應中回傳之資料的 CustomTemplateException。

Null 行為 utils

Null 行為利用率清單

\$util.isNull(Object) : Boolean

如果提供物件為 null，則傳回真。

\$util.isNullOrEmpty(String) : Boolean

如果提供資料為 null 或空白字串，則傳回真。否則即傳回 false。

\$util.isNullOrBlank(String) : Boolean

如果提供資料為 null 或空白字串，則傳回真。否則即傳回 false。

\$util.defaultIfNull(Object, Object) : Object

如果不是 null，則傳回第一個物件。否則，會傳回第二個物件做為「預設物件」。

\$util.defaultIfNullOrEmpty(String, String) : String

如果不是 null 或空白，則傳回第一個字串。否則，會傳回第二個字串做為「預設字串」。

\$util.defaultIfNullOrBlank(String, String) : String

如果不是 null 或空白，則傳回第一個字串。否則，會傳回第二個字串做為「預設字串」。

模式比對 utils

類型和模式比對 utils 清單

\$util.typeOf(Object) : String

傳回描述物件類型的字串。支援的類型識別為：「Null」、「數字」、「字串」、「映射」、「清單」、「布林值」。如果無法識別類型，則傳回類型為「物件」。

\$util.matches(String, String) : Boolean

如果第一個引數中指定的模式與第二個引數中提供的資料相符，則傳回真。模式必須為規則表達式，例如 `$util.matches("a*b", "aaaaab")`。此功能是根據[模式](#)，您可以參考以取得更詳細的文件。

\$util.authType() : String

傳回描述請求所使用之多重身分類型的字串，傳回「IAM 授權」、「使用者集區授權」、「開放式 ID Connect 授權」或「API 金鑰授權」。

物件驗證公用程式

物件驗證公用程式清單

\$util.isString(Object) : Boolean

如果物件是字串，則傳回 true。

\$util.isNumber(Object) : Boolean

如果物件是數字，則傳回 true。

\$util.isBoolean(Object) : Boolean

如果物件是布林值，則傳回 true。

\$util.isList(Object) : Boolean

如果物件是清單，則傳回 true。

\$util.isMap(Object) : Boolean

如果物件是映射，則傳回 true。

CloudWatch 記錄公用程式

CloudWatch 記錄公用程式清單

\$util.log.info(Object) : Void

在 API DEBUG 上使用日誌層級 INFO、或 啟用請求層級和欄位層級 CloudWatch 記錄時 ALL，將所提供物件的字串表示法記錄到請求的日誌串流。

\$util.log.info(String, Object...) : Void

在 API ALL 上使用日誌層級啟用請求層級和欄位層級 CloudWatch 記錄時，將所提供物件的字串表示法記錄到請求的日誌串流。此公用程式會依序將第一個輸入格式字串中「{」指示的所有變數取代為所提供物件的字串表示法。

\$util.log.debug(Object) : Void

使用日誌層級 ALL 或在 API DEBUG 上啟用請求層級和欄位層級 CloudWatch 記錄時，將所提供物件的字串表示法記錄到請求的日誌串流。

\$util.log.debug(String, Object...) : Void

使用 API 上的日誌層級 DEBUG 或日誌層級啟用欄位層級 CloudWatch 記錄時，將所提供物件的字串表示法記錄到請求 ALL 的日誌串流。此公用程式會依序將第一個輸入格式字串中「{」指示的所有變數取代為所提供物件的字串表示法。

\$util.log.error(Object) : Void

在 API 上以任何日誌層級 (ALL、DEBUG、等) 啟用欄位層級 CloudWatch 記錄時 INFO，將所提供物件的字串表示法記錄到請求的日誌串流。

\$util.log.error(String, Object...) : Void

使用 API 上的日誌層級 ERROR 或日誌層級啟用欄位層級 CloudWatch 記錄時，將所提供物件的字串表示法記錄到請求 ALL 的日誌串流。此公用程式會依序將第一個輸入格式字串中「{」指示的所有變數取代為所提供物件的字串表示法。

傳回值行為 utils

傳回值行為使用率清單

\$util.qr() 和 \$util.quiet()

執行 VTL 陳述式，同時隱藏傳回的值。這適用於執行方法而不使用臨時預留位置，例如將項目新增至映射。例如：


```
#set ($myMap = {})  
#set($discard = $myMap.put("id", "first value"))
```

變成：

```
#set ($myMap = {})  
$util.qr($myMap.put("id", "first value"))
```

\$util.escapeJavaScript(String) : String

以 JavaScript 逸出字串的形式傳回輸入字串。

\$util.urlEncode(String) : String

以 application/x-www-form-urlencoded 編碼字串的形式傳回輸入字串。

\$util.urlDecode(String) : String

將 application/x-www-form-urlencoded 編碼的字串解碼回非編碼格式。

\$util.base64Encode(byte[]) : String

將輸入編碼為 base64 編碼字串。

\$util.base64Decode(String) : byte[]

解碼 base64 編碼字串中的資料。

\$util.parseJson(String) : Object

採用「字串化」JSON，並傳回結果的物件呈現。

\$util.toJson(Object) : String

採用物件，並傳回該物件的「字串化」JSON 呈現。

\$util.autoId() : String

傳回 128 位元隨機產生的 UUID。

\$util.autoUlid() : String

傳回 128 位元隨機產生的 ULID（通用可排序序列識別符）。

\$util.autoKsuid() : String

傳回 128 位元隨機產生的 KSUID (K-Sortable Unique Identifier) base62，編碼為長度為 27 的字串。

\$util.unauthorized()

擲回欲解析之欄位的 Unauthorized。在請求或回應映射範本中使用此選項，以決定是否允許發起人解析 欄位。

\$util.error(String)

擲回自訂錯誤。在請求或回應映射範本中使用此參數來偵測請求或調用結果的錯誤。

\$util.error(String, String)

擲回自訂錯誤。在請求或回應映射範本中使用此參數來偵測請求或調用結果的錯誤。您也可以指定 `errorType`。

\$util.error(String, String, Object)

擲回自訂錯誤。在請求或回應映射範本中使用此參數來偵測請求或調用結果的錯誤。您也可以指定 `errorType` 和 `data` 欄位。`data` 值將新增到 GraphQL 回應中，`error` 內對應的 `errors` 區塊。注意：`data` 將根據查詢選取範圍集進行篩選。

\$util.error(String, String, Object, Object)

擲回自訂錯誤。如果範本偵測到要求或呼叫結果的錯誤，您可以將此用於要求或回應映射範本。您同時可以指定 `errorType` 欄位、`data` 欄位和 `errorInfo` 欄位。`data` 值將新增到 GraphQL 回應中，`error` 內對應的 `errors` 區塊。注意：`data` 將根據查詢選取範圍集進行篩選。`errorInfo` 值將新增到 GraphQL 回應中，`error` 內對應的 `errors` 區塊。注意：`errorInfo` 將不會根據查詢選取範圍集進行篩選。

\$util.appendError(String)

附加自訂錯誤。如果範本偵測到要求或呼叫結果的錯誤，您可以將此用於要求或回應映射範本。與 `$util.error(String)` 不同的是，範本評估不會受中斷，因此可以將資料傳回給發起人。

\$util.appendError(String, String)

附加自訂錯誤。如果範本偵測到要求或呼叫結果的錯誤，您可以將此用於要求或回應映射範本。您同時可以指定 `errorType`。與 `$util.error(String, String)` 不同的是，範本評估不會受中斷，因此可以將資料傳回給發起人。

\$util.appendError(String, String, Object)

附加自訂錯誤。如果範本偵測到要求或呼叫結果的錯誤，您可以將此用於要求或回應映射範本。您同時可以指定 `errorType` 和 `data` 欄位。與 `$util.error(String, String, Object)` 不同的是，範本評估不會受中斷，因此可以將資料傳回給發起人。`data` 值將新增到

GraphQL 回應中，`error` 內對應的 `errors` 區塊。注意：data 將根據查詢選取範圍集進行篩選。

`$util.appendError(String, String, Object, Object)`

附加自訂錯誤。如果範本偵測到要求或呼叫結果的錯誤，您可以將此用於要求或回應映射範本。您同時可以指定 `errorType` 欄位、`data` 欄位和 `errorInfo` 欄位。與 `$util.error(String, String, Object, Object)` 不同的是，範本評估不會受中斷，因此可以將資料傳回給發起人。data 值將新增到 GraphQL 回應中，`error` 內對應的 `errors` 區塊。注意：data 將根據查詢選取範圍集進行篩選。errorInfo 值將新增到 GraphQL 回應中，`error` 內對應的 `errors` 區塊。注意：errorInfo 將不會根據查詢選取範圍集進行篩選。

`$util.validate(Boolean, String) : void`

如果條件為假，會擲回含指定訊息的 `CustomTemplateException`。

`$util.validate(Boolean, String, String) : void`

如果條件為假，會擲回含指定訊息和錯誤類型的 `CustomTemplateException`。

`$util.validate(Boolean, String, String, Object) : void`

如果條件為假，會擲回含指定訊息、錯誤類型以及在回應中回傳之資料的 `CustomTemplateException`。

`$util.isNull(Object) : Boolean`

如果提供物件為 `null`，則傳回真。

`$util.isNullOrEmpty(String) : Boolean`

如果提供資料為 `null` 或空白字串，則傳回真。否則即傳回 `false`。

`$util.isNullOrBlank(String) : Boolean`

如果提供資料為 `null` 或空白字串，則傳回真。否則即傳回 `false`。

`$util.defaultIfNull(Object, Object) : Object`

如果不是 `null`，則傳回第一個物件。否則，會傳回第二個物件做為「預設物件」。

`$util.defaultIfNullOrEmpty(String, String) : String`

如果不是 `null` 或空白，則傳回第一個字串。否則，會傳回第二個字串做為「預設字串」。

`$util.defaultIfNullOrBlank(String, String) : String`

如果不是 `null` 或空白，則傳回第一個字串。否則，會傳回第二個字串做為「預設字串」。

\$util.isString(Object) : Boolean

如果物件是字串，則傳回真。

\$util.isNumber(Object) : Boolean

如果物件是數字，則傳回真。

\$util.isBoolean(Object) : Boolean

如果物件是布林值，則傳回真。

\$util.isList(Object) : Boolean

如果物件是清單，則傳回真。

\$util.isMap(Object) : Boolean

如果物件是映射，則傳回真。

\$util.typeOf(Object) : String

傳回描述物件類型的字串。支援的類型識別為：「Null」、「數字」、「字串」、「映射」、「清單」、「布林值」。如果無法識別類型，則傳回類型為「物件」。

\$util.matches(String, String) : Boolean

如果第一個引數中指定的模式與第二個引數中提供的資料相符，則傳回真。模式必須為規則表達式，例如 `$util.matches("a*b", "aaaaab")`。此功能是根據[模式](#)，您可以參考以取得更詳細的文件。

\$util.authType() : String

傳回描述請求所使用之多重身分類型的字串，傳回「IAM 授權」、「使用者集區授權」、「開放式 ID Connect 授權」或「API 金鑰授權」。

\$util.log.info(Object) : Void

在 API ALL 上使用日誌層級啟用請求層級和欄位層級 CloudWatch 記錄時，將所提供物件的字串表示記錄到請求的日誌串流。

\$util.log.info(String, Object...) : Void

在 API ALL 上使用日誌層級啟用請求層級和欄位層級 CloudWatch 記錄時，將所提供物件的字串表示法記錄到請求的日誌串流。此公用程式會依序將第一個輸入格式字串中「{ }」指示的所有變數取代為所提供物件的字串表示法。

\$util.log.error(Object) : Void

使用 API 上的日誌層級ERROR或日誌層級啟用欄位層級 CloudWatch 記錄時，將所提供物件的字串表示法記錄到請求ALL的日誌串流。

\$util.log.error(String, Object...) : Void

使用 API 上的日誌層級ERROR或日誌層級啟用欄位層級 CloudWatch 記錄時，將所提供物件的字串表示法記錄到請求ALL的日誌串流。此公用程式會依序將第一個輸入格式字串中「{}」指示的所有變數取代為所提供物件的字串表示法。

\$util.escapeJavaScript(String) : String

以 JavaScript 逸出字串的形式傳回輸入字串。

解析程式授權

解析程式授權清單

\$util.unauthorized()

擲回欲解析之欄位的 Unauthorized。在請求或回應映射範本中使用此選項，以決定是否允許發起人解析 欄位。

AWS AppSync 指令

Note

我們現在主要支援 APPSYNC_JS 執行期及其文件。請考慮[在此處](#)使用 APPSYNC_JS 執行期及其指南。

AWS AppSync 公開指令，以促進開發人員在 VTL 中寫入時的生產力。

指令 utils

#return(Object)

#return(Object) 可讓您從任何映射範本提早傳回。#return(Object) 類似於程式設計語言的傳回關鍵字，因為它將從邏輯的最接近範圍區塊傳回。在解析程式映射範

本 `#return(Object)` 內使用 `會從解析程式傳回`。此外，`#return(Object)` 從函數映射範本使用 `會從函數傳回`，並繼續執行至管道中的下一個函數或解析程式回應映射範本。

`#return`

`#return` 指令會呈現與 `相同的行為#return(Object)`，但 `null` 會改為傳回。

`$util.time` 中的時間協助程式

Note

我們現在主要支援 APPSYNC_JS 執行期及其文件。請考慮[在此處](#)使用 APPSYNC_JS 執行期及其指南。

`$util.time` 變數包含日期時間方法，可協助產生時間戳記、在日期時間格式之間轉換，以及剖析日期時間字串。日期時間格式的語法是根據 [DateTimeFormatter](#)，您可以參考以取得更詳細的文件。我們提供以下一些範例，以及可用方法和描述的清單。

時間 utils

時間 utils 清單

`$util.time.nowISO8601() : String`

以 [ISO8601 格式](#) 傳回 UTC 的字串表示方式。

`$util.time.nowEpochSeconds() : long`

傳回從 1970-01-01T00:00:00Z 的 epoch 到現在的秒數。

`$util.time.nowEpochMilliseconds() : long`

傳回從 1970-01-01T00:00:00Z 的 epoch 到現在的毫秒數。

`$util.time.nowFormatted(String) : String`

使用字串輸入類型的指定格式，傳回 UTC 中目前時間戳記的字串。

`$util.time.nowFormatted(String, String) : String`

使用字串輸入類型的指定格式和時區，傳回時區目前時間戳記的字串。

\$util.time.parseFormattedToEpochMilliseconds(String, String) : Long

剖析以字串形式傳遞的時間戳記，以及包含時區的格式，然後傳回自 epoch 以來以毫秒為單位的時間戳記。

\$util.time.parseFormattedToEpochMilliseconds(String, String, String) : Long

剖析以字串形式傳遞的時間戳記，以及格式和時區，然後傳回自 epoch 以來以毫秒為單位的時間戳記。

\$util.time.parseISO8601ToEpochMilliseconds(String) : Long

剖析以字串形式傳遞的 ISO8601 時間戳記，然後以毫秒為單位傳回自 epoch 以來的時間戳記。

\$util.time.epochMillisecondsToSeconds(long) : long

將 epoch 毫秒時間戳記轉換為 epoch 秒時間戳記。

\$util.time.epochMillisecondsToISO8601(long) : String

將 epoch 毫秒時間戳記轉換為 ISO8601 時間戳記。

\$util.time.epochMillisecondsToFormatted(long, String) : String

將 epoch 毫秒時間戳記轉換為根據 UTC 格式提供的時間戳記。

\$util.time.epochMillisecondsToFormatted(long, String, String) : String

將長傳遞的 epoch 毫秒時間戳記轉換為根據所提供時區中提供的格式格式化的時間戳記。

獨立函數範例

```
$util.time.nowISO8601() : 
2018-02-06T19:01:35.749Z
$util.time.nowEpochSeconds() : 1517943695
$util.time.nowEpochMilliseconds() : 1517943695750
$util.time.nowFormatted("yyyy-MM-dd HH:mm:ssZ") : 2018-02-06
19:01:35+0000
$util.time.nowFormatted("yyyy-MM-dd HH:mm:ssZ", "+08:00") : 2018-02-07
03:01:35+0800
$util.time.nowFormatted("yyyy-MM-dd HH:mm:ssZ", "Australia/Perth") : 2018-02-07
03:01:35+0800
```

轉換範例

```
#set( $nowEpochMillis = 1517943695758 )
```

```
$util.time.epochMillisecondsToSeconds($nowEpochMillis)
    : 1517943695
$util.time.epochMillisecondsToISO8601($nowEpochMillis)
    : 2018-02-06T19:01:35.758Z
$util.time.epochMillisecondsToFormatted($nowEpochMillis, "yyyy-MM-dd HH:mm:ssZ")
    : 2018-02-06 19:01:35+0000
$util.time.epochMillisecondsToFormatted($nowEpochMillis, "yyyy-MM-dd HH:mm:ssZ",
    "+08:00") : 2018-02-07 03:01:35+0800
```

剖析範例

```
$util.time.parseISO8601ToEpochMilliseconds("2018-02-01T17:21:05.180+08:00")
    : 1517476865180
$util.time.parseFormattedToEpochMilliseconds("2018-02-02 01:19:22+0800", "yyyy-MM-dd
    HH:mm:ssZ") : 1517505562000
$util.time.parseFormattedToEpochMilliseconds("2018-02-02 01:19:22", "yyyy-MM-dd
    HH:mm:ss", "+08:00") : 1517505562000
```

使用已 AWS AppSync 定義的純量

以下格式與 AWSDate、AWSDateTime 和 AWSTime 相容。

```
$util.time.nowFormatted("yyyy-MM-dd[XXX]", "-07:00:30") :
    2018-07-11-07:00
$util.time.nowFormatted("yyyy-MM-dd'T'HH:mm:ss[XXXXXX]", "-07:00:30") :
    2018-07-11T15:14:15-07:00:30
```

在 \$util.list 中列出協助程式

Note

我們現在主要支援 APPSYNC_JS 執行期及其文件。請考慮[在此處](#)使用 APPSYNC_JS 執行期及其指南。

`$util.list` 包含有助於常見清單操作的方法，例如從清單中移除或保留項目以篩選使用案例。

列出 utils

`$util.list.copyAndRetainAll(List, List) : List`

製作第一個引數中所提供清單的淺層副本，如果項目存在，則僅保留第二個引數中指定的項目。所有其他項目將從副本中刪除。

`$util.list.copyAndRemoveAll(List, List) : List`

在移除項目在第二個引數中指定的任何項目時，製作第一個引數中所提供清單的淺層副本。所有其他項目將保留在副本中。

`$util.list.sortList(List, Boolean, String) : List`

排序在第一個引數中提供的物件清單。如果第二個引數為 `true`，則會以遞減方式排序清單；如果第二個引數為 `false`，則會以遞增方式排序清單。第三個引數是用來排序自訂物件清單的屬性字串名稱。如果只是字串、整數、浮點數或雙數的清單，則第三個引數可以是任何隨機字串。如果所有物件不是來自相同類別，則會傳回原始清單。僅支援最多包含 1000 個物件的清單。以下是此公用程式使用的範例：

```
INPUT:      $util.list.sortList([{"description":"youngest", "age":5},
{"description":"middle", "age":45}, {"description":"oldest", "age":85}], false,
"description")
OUTPUT:     [{"description":"middle", "age":45}, {"description":"oldest",
"age":85}, {"description":"youngest", "age":5}]
```

在 `$util.map` 中映射協助程式

Note

我們現在主要支援 APPSYNC_JS 執行期及其文件。請考慮[在此處](#)使用 APPSYNC_JS 執行期及其指南。

`$util.map` 包含有助於常見 Map 操作的方法，例如從 Map 中移除或保留項目，以篩選使用案例。

映射 utils

`$util.map.copyAndRetainAllKeys(Map, List) : Map`

製作第一個貼圖的淺層副本，同時只保留清單中指定的索引鍵，如果它們存在的話。所有其他金鑰將從副本中刪除。

`$util.map.copyAndRemoveAllKeys(Map, List) : Map`

在移除清單中指定金鑰的任何項目時，製作第一個映射的淺層副本，如果有的話。所有其他金鑰將保留在副本中。

`$util.dynamodb` 中的 DynamoDB 協助程式

Note

我們現在主要支援 APPSYNC_JS 執行期及其文件。請考慮[在此處](#)使用 APPSYNC_JS 執行期及其指南。

`$util.dynamodb` 包含協助程式方法，可讓您更輕鬆地將資料寫入和讀取至 Amazon DynamoDB，例如自動類型映射和格式化。這些方法旨在將基本類型和清單自動映射至適當的 DynamoDB 輸入格式，這是格式Map的 { "TYPE" : VALUE }。

例如，先前在 DynamoDB 中建立新項目的請求映射範本可能如下所示：

```
{
  "version" : "2017-02-28",
  "operation" : "PutItem",
  "key": {
    "id" : { "S" : "$util.autoId()" }
  },
  "attributeValues" : {
    "title" : { "S" : $util.toJson($ctx.args.title) },
    "author" : { "S" : $util.toJson($ctx.args.author) },
    "version" : { "N", $util.toJson($ctx.args.version) }
  }
}
```

如果我們想要將欄位新增到物件，必須在結構描述中更新 GraphQL 查詢和要求映射範本。不過，我們現在可以重組請求映射範本，讓它自動挑選在我們的結構描述中新增的新欄位，並使用正確的類型將其新增至 DynamoDB：

```
{
  "version" : "2017-02-28",
  "operation" : "PutItem",
  "key": {
    "id" : $util.dynamodb.toDynamoDBJson($util.autoId())
  },
  "attributeValues" : $util.dynamodb.toMapValuesJson($ctx.args)
}
```

在先前的範例中，我們使用 `$util.dynamodb.toDynamoDBJson(...)` 協助程式自動取得產生的 ID，並將其轉換為字串屬性的 DynamoDB 表示法。然後，我們採用所有引數並將其轉換為其 DynamoDB 表示法，並將其輸出至範本中的 `attributeValues` 欄位。

每個協助程式有兩個版本：傳回物件的版本 (例如，`$util.dynamodb.toString(...)`)，以及傳回物件做為 JSON 字串的版本 (例如，`$util.dynamodb.toStringJson(...)`)。在前述範例中，我們使用傳回資料做為 JSON 字串的版本。如果您想要在範本使用物件之前，對物件進行操作，可以選擇傳回物件，如下所示：

```
{
  "version" : "2017-02-28",
  "operation" : "PutItem",
  "key": {
    "id" : $util.dynamodb.toDynamoDBJson($util.autoId())
  },

  #set( $myFoo = $util.dynamodb.toMapValues($ctx.args) )
  #set( $myFoo.version = $util.dynamodb.toNumber(1) )
  #set( $myFoo.timestamp = $util.dynamodb.toString($util.time.nowISO8601()))

  "attributeValues" : $util.toJson($myFoo)
}
```

在前述範例中，我們傳回轉換引數做為映射而非 JSON 字串，然後在使用 `version` 將其輸出到範本的 `timestamp` 欄位之前，新增 `attributeValues` 和 `$util.toJson(...)` 欄位。

每個 JSON 版本的協助程式等同於以 `$util.toJson(...)` 包裝非 JSON 版本。例如，下列陳述式完全相同：

```
$util.toJson($util.toString("Hello, World!"))
```

toDynamoDB

toDynamoDB 公用程式清單

`$util.dynamodb.toDynamoDB(Object)` : Map

DynamoDB 的一般物件轉換工具，可將輸入物件轉換為適當的 DynamoDB 表示法。它在代表一些類型的方式上是固定的：例如，它會使用清單（「L」）而不是集合（「SS」、「NS」、「BS」）。這會傳回描述 DynamoDB 屬性值的物件。

字串範例

```
Input:      $util.dynamodb.toDynamoDB("foo")
Output:     { "S" : "foo" }
```

數字範例

```
Input:      $util.dynamodb.toDynamoDB(12345)
Output:     { "N" : 12345 }
```

布林值範例

```
Input:      $util.dynamodb.toDynamoDB(true)
Output:     { "BOOL" : true }
```

清單範例

```
Input:      $util.dynamodb.toDynamoDB([ "foo", 123, { "bar" : "baz" } ])
Output:     {
      "L" : [
        { "S" : "foo" },
        { "N" : 123 },
        {
          "M" : {
            "bar" : { "S" : "baz" }
          }
        }
      ]
    }
```

```

    }
  ]
}

```

地圖範例

```

Input:      $util.dynamodb.toDynamoDB({ "foo": "bar", "baz" : 1234, "beep":
[ "boop" ] })
Output:     {
    "M" : {
      "foo" : { "S" : "bar" },
      "baz" : { "N" : 1234 },
      "beep" : {
        "L" : [
          { "S" : "boop" }
        ]
      }
    }
  }
}

```

`$util.dynamodb.toDynamoDBJson(Object) : String`

與相同 `$util.dynamodb.toDynamoDB(Object) : Map`，但會以 JSON 編碼字串的形式傳回 DynamoDB 屬性值。

toString utils

toString utils 清單

`$util.dynamodb.toString(String) : String`

將輸入字串轉換為 DynamoDB 字串格式。這會傳回描述 DynamoDB 屬性值的物件。

```

Input:      $util.dynamodb.toString("foo")
Output:     { "S" : "foo" }

```

`$util.dynamodb.toStringJson(String) : Map`

與相同 `$util.dynamodb.toString(String) : String`，但會以 JSON 編碼字串的形式傳回 DynamoDB 屬性值。

\$util.dynamodb.toStringSet(List<String>) : Map

將具有字串的清單轉換為 DynamoDB 字串集格式。這會傳回描述 DynamoDB 屬性值的物件。

```
Input:      $util.dynamodb.toStringSet([ "foo", "bar", "baz" ])
Output:     { "SS" : [ "foo", "bar", "baz" ] }
```

\$util.dynamodb.toStringSetJson(List<String>) : String

與相同 `$util.dynamodb.toStringSet(List<String>) : Map`，但會以 JSON 編碼字串的形式傳回 DynamoDB 屬性值。

toNumber utils

toNumber utils 清單

\$util.dynamodb.toNumber(Number) : Map

將數字轉換為 DynamoDB 數字格式。這會傳回描述 DynamoDB 屬性值的物件。

```
Input:      $util.dynamodb.toNumber(12345)
Output:     { "N" : 12345 }
```

\$util.dynamodb.toNumberJson(Number) : String

與相同 `$util.dynamodb.toNumber(Number) : Map`，但會以 JSON 編碼字串的形式傳回 DynamoDB 屬性值。

\$util.dynamodb.toNumberSet(List<Number>) : Map

將數字清單轉換為 DynamoDB 數字集格式。這會傳回描述 DynamoDB 屬性值的物件。

```
Input:      $util.dynamodb.toNumberSet([ 1, 23, 4.56 ])
Output:     { "NS" : [ 1, 23, 4.56 ] }
```

\$util.dynamodb.toNumberSetJson(List<Number>) : String

與相同 `$util.dynamodb.toNumberSet(List<Number>) : Map`，但會以 JSON 編碼字串的形式傳回 DynamoDB 屬性值。

toBinary utils

toBinary utils 清單

\$util.dynamodb.toBinary(String) : Map

將編碼為 base64 字串的二進位資料轉換為 DynamoDB 二進位格式。這會傳回描述 DynamoDB 屬性值的物件。

```
Input:      $util.dynamodb.toBinary("foo")
Output:     { "B" : "foo" }
```

\$util.dynamodb.toBinaryJson(String) : String

與相同 `$util.dynamodb.toBinary(String) : Map`，但會以 JSON 編碼字串的形式傳回 DynamoDB 屬性值。

\$util.dynamodb.toBinarySet(List<String>) : Map

將編碼為 base64 字串的二進位資料清單轉換為 DynamoDB 二進位集格式。這會傳回描述 DynamoDB 屬性值的物件。

```
Input:      $util.dynamodb.toBinarySet([ "foo", "bar", "baz" ])
Output:     { "BS" : [ "foo", "bar", "baz" ] }
```

\$util.dynamodb.toBinarySetJson(List<String>) : String

與相同 `$util.dynamodb.toBinarySet(List<String>) : Map`，但會以 JSON 編碼字串的形式傳回 DynamoDB 屬性值。

toBoolean utils

toBoolean utils 清單

\$util.dynamodb.toBoolean(Boolean) : Map

將布林值轉換為適當的 DynamoDB 布林值格式。這會傳回描述 DynamoDB 屬性值的物件。

```
Input:      $util.dynamodb.toBoolean(true)
Output:     { "BOOL" : true }
```

\$util.dynamodb.toBooleanJson(Boolean) : String

與相同 `$util.dynamodb.toBoolean(Boolean) : Map`，但會以 JSON 編碼字串的形式傳回 DynamoDB 屬性值。

toNull utils

toNull utils 清單

\$util.dynamodb.toNull() : Map

以 DynamoDB null 格式傳回 null。這會傳回描述 DynamoDB 屬性值的物件。

```
Input:      $util.dynamodb.toNull()
Output:     { "NULL" : null }
```

\$util.dynamodb.toNullJson() : String

與相同 `$util.dynamodb.toNull() : Map`，但會以 JSON 編碼字串的形式傳回 DynamoDB 屬性值。

toList utils

toList utils 清單

\$util.dynamodb.toList(List) : Map

將物件清單轉換為 DynamoDB 清單格式。清單中的每個項目也會轉換為其適當的 DynamoDB 格式。它在代表一些巢狀物件的方式上是固定的：例如，它會使用清單（「L」）而不是集合（「SS」、「NS」、「BS」）。這會傳回描述 DynamoDB 屬性值的物件。

```
Input:      $util.dynamodb.toList([ "foo", 123, { "bar" : "baz" } ])
Output:     {
      "L" : [
        { "S" : "foo" },
        { "N" : 123 },
        {
          "M" : {
            "bar" : { "S" : "baz" }
          }
        }
      ]
    }
```



```
    }
  ]
}
```

\$util.dynamodb.toListJson(List) : String

與相同 `$util.dynamodb.toList(List) : Map`，但會以 JSON 編碼字串的形式傳回 DynamoDB 屬性值。

toMap 公用程式

toMap 公用程式清單

\$util.dynamodb.toMap(Map) : Map

將映射轉換為 DynamoDB 映射格式。映射中的每個值也會轉換為其適當的 DynamoDB 格式。它在代表一些巢狀物件的方式上是固定的：例如，它會使用清單 (「L」) 而不是集合 (「SS」、「NS」、「BS」)。這會傳回描述 DynamoDB 屬性值的物件。

```
Input:      $util.dynamodb.toMap({ "foo": "bar", "baz" : 1234, "beep": [ "boop" ] })
Output:     {
              "M" : {
                "foo" : { "S" : "bar" },
                "baz" : { "N" : 1234 },
                "beep" : {
                  "L" : [
                    { "S" : "boop" }
                  ]
                }
              }
            }
```

\$util.dynamodb.toMapJson(Map) : String

與相同 `$util.dynamodb.toMap(Map) : Map`，但會以 JSON 編碼字串的形式傳回 DynamoDB 屬性值。

\$util.dynamodb.toMapValues(Map) : Map

建立映射的副本，其中每個值都已轉換為其適當的 DynamoDB 格式。它在代表一些巢狀物件的方式上是固定的：例如，它會使用清單 (「L」) 而不是集合 (「SS」、「NS」、「BS」)。

```

Input:      $util.dynamodb.toMapValues({ "foo": "bar", "baz" : 1234, "beep":
[ "boop" ] })
Output:     {
    "foo"   : { "S" : "bar" },
    "baz"   : { "N" : 1234 },
    "beep"  : {
        "L" : [
            { "S" : "boop" }
        ]
    }
}

```

Note

這與 略有不同，`$util.dynamodb.toMap(Map) : Map` 因為它只會傳回 DynamoDB 屬性值的內容，但不會傳回整個屬性值本身。例如，下列陳述式完全相同：

```

$util.dynamodb.toMapValues($map)
$util.dynamodb.toMap($map).get("M")

```

`$util.dynamodb.toMapValuesJson(Map) : String`

與 相同 `$util.dynamodb.toMapValues(Map) : Map`，但 會以 JSON 編碼字串的形式傳回 DynamoDB 屬性值。

S3Object 公用程式

S3Object 公用程式清單

`$util.dynamodb.toS3Object(String key, String bucket, String region) : Map`

將金鑰、儲存貯體和區域轉換為 DynamoDB S3 物件表示法。這會傳回描述 DynamoDB 屬性值的物件。

```

Input:      $util.dynamodb.toS3Object("foo", "bar", region = "baz")
Output:     { "S" : "{ \"s3\" : { \"key\" : \"foo\", \"bucket\" : \"bar\", \"region
\" : \"baz\" } }" }

```

\$util.dynamodb.toS3ObjectJson(String key, String bucket, String region) : String

與相同 `$util.dynamodb.toS3Object(String key, String bucket, String region) : Map`，但會以 JSON 編碼字串的形式傳回 DynamoDB 屬性值。

\$util.dynamodb.toS3Object(String key, String bucket, String region, String version) : Map

將金鑰、儲存貯體、區域和選用版本轉換為 DynamoDB S3 物件表示法。這會傳回描述 DynamoDB 屬性值的物件。

```
Input:      $util.dynamodb.toS3Object("foo", "bar", "baz", "beep")
Output:     { "S" : "{ \"s3\" : { \"key\" : \"foo\", \"bucket\" : \"bar\", \"region\" : \"baz\", \"version\" = \"beep\" } }" }
```

\$util.dynamodb.toS3ObjectJson(String key, String bucket, String region, String version) : String

與相同 `$util.dynamodb.toS3Object(String key, String bucket, String region, String version) : Map`，但會以 JSON 編碼字串的形式傳回 DynamoDB 屬性值。

\$util.dynamodb.fromS3ObjectJson(String) : Map

接受 DynamoDB S3 物件的字串值，並傳回包含金鑰、儲存貯體、區域和選用版本的映射。

```
Input:      $util.dynamodb.fromS3ObjectJson({ "S" : "{ \"s3\" : { \"key\" : \"foo\", \"bucket\" : \"bar\", \"region\" : \"baz\", \"version\" = \"beep\" } }" })
Output:     { "key" : "foo", "bucket" : "bar", "region" : "baz", "version" : "beep" }
```

\$util.rds 中的 Amazon RDS 協助程式

Note

我們現在主要支援 APPSYNC_JS 執行期及其文件。請考慮[在此處](#)使用 APPSYNC_JS 執行期及其指南。

`$util.rds` 包含協助程式方法，透過消除結果輸出中的外部資料來格式化 Amazon RDS 操作。

\$util.rds utils 清單

\$util.rds.toJsonString(String serializedSQLResult): String

傳回 String，方法是將字串化原始 Amazon Relational Database Service (Amazon RDS) Data API 操作結果格式轉換為更簡潔的字串。傳回的字串是結果集的 SQL 記錄序列化清單。每筆記錄都會顯示為鍵值對集合。索引鍵是對應的欄位名稱。

如果輸入中的對應陳述式是造成變動的 SQL 查詢（例如 INSERT、UPDATE、DELETE），則會傳回空清單。例如，查詢 `select * from Books limit 2` 提供來自 Amazon RDS Data 操作的原始結果：

```
{
  "sqlStatementResults": [
    {
      "numberOfRecordsUpdated": 0,
      "records": [
        [
          {
            "stringValue": "Mark Twain"
          },
          {
            "stringValue": "Adventures of Huckleberry Finn"
          },
          {
            "stringValue": "978-1948132817"
          }
        ],
        [
          {
            "stringValue": "Jack London"
          },
          {
            "stringValue": "The Call of the Wild"
          },
          {
            "stringValue": "978-1948132275"
          }
        ]
      ],
      "columnMetadata": [
        {
          "isSigned": false,
```

```
    "isCurrency": false,
    "label": "author",
    "precision": 200,
    "typeName": "VARCHAR",
    "scale": 0,
    "isAutoIncrement": false,
    "isCaseSensitive": false,
    "schemaName": "",
    "tableName": "Books",
    "type": 12,
    "nullable": 0,
    "arrayBaseColumnType": 0,
    "name": "author"
  },
  {
    "isSigned": false,
    "isCurrency": false,
    "label": "title",
    "precision": 200,
    "typeName": "VARCHAR",
    "scale": 0,
    "isAutoIncrement": false,
    "isCaseSensitive": false,
    "schemaName": "",
    "tableName": "Books",
    "type": 12,
    "nullable": 0,
    "arrayBaseColumnType": 0,
    "name": "title"
  },
  {
    "isSigned": false,
    "isCurrency": false,
    "label": "ISBN-13",
    "precision": 15,
    "typeName": "VARCHAR",
    "scale": 0,
    "isAutoIncrement": false,
    "isCaseSensitive": false,
    "schemaName": "",
    "tableName": "Books",
    "type": 12,
    "nullable": 0,
    "arrayBaseColumnType": 0,
```

```

        "name": "ISBN-13"
      }
    ]
  }
}

```

`util.rds.toJsonString` 是：

```

[
  {
    "author": "Mark Twain",
    "title": "Adventures of Huckleberry Finn",
    "ISBN-13": "978-1948132817"
  },
  {
    "author": "Jack London",
    "title": "The Call of the Wild",
    "ISBN-13": "978-1948132275"
  },
]

```

`$util.rds.toJsonObject(String serializedSQLResult): Object`

這與 `util.rds.toJsonString` 相同，但結果為 JSON Object。

`$util.http` 中的 HTTP 協助程式

Note

我們現在主要支援 APPSYNC_JS 執行期及其文件。請考慮[在此處](#)使用 APPSYNC_JS 執行期及其指南。

`$util.http` 公用程式提供協助程式方法，您可以用來管理 HTTP 請求參數和新增回應標頭。

`$util.http` utils 清單

`$util.http.copyHeaders(Map) : Map`

從映射複製標頭，不包括下列限制的 HTTP 標頭：

- transfer-encoding
- connection
- host
- expect
- keep-alive
- upgrade
- proxy-authenticate
- proxy-authorization
- te
- content-length

您可以使用此公用程式，將請求標頭轉送至下游 HTTP 端點。

```
{
  ...
  "params": {
    ...
    "headers": $util.http.copyHeaders($ctx.request.headers),
    ...
  },
  ...
}
```

`$util.http.addResponseHeader` (字串、物件)

新增具有回應名稱 (String) 和值 (Object) 的單一自訂標頭。有下列限制：

- 除了的限制標頭清單之外`copyHeaders(Map)`，標頭名稱也不符合下列任何項目：
 - Access-Control-Allow-Credentials
 - Access-Control-Allow-Origin
 - Access-Control-Expose-Headers
 - Access-Control-Max-Age
 - Access-Control-Allow-Methods
 - Access-Control-Allow-Headers
 - Vary

- Content-Type
- 標頭名稱不能以限制字首 x-amzn- 或 開頭 x-amz-。
- 自訂回應標頭的大小不能超過 4 KB。這包括標頭名稱和值。
- 您應該為每個 GraphQL 操作定義每個回應標頭一次。不過，如果您多次定義同名的自訂標頭，最近的定義會出現在回應中。無論命名為何，所有標頭都會計入標頭大小限制。
- 具有空白或限制名稱(String)或 null 值的標頭(Object)將被忽略，並產生新增到操作errors輸出的ResponseHeaderError錯誤。

```
export function request(ctx) {
  util.http.addResponseHeader('itemsCount', 7)
  util.http.addResponseHeader('render', ctx.args.render)
  return {}
}
```

\$util.http.addResponseHeaders(Map)

將多個回應標頭新增至指定名稱(String)和值映射的回應(Object)。為 addResponseHeader(String, Object)方法列出的相同限制也適用於此方法。

```
export function request(ctx) {
  const headers = {
    headerInt: 12,
    headerString: 'stringValue',
    headerObject: {
      field1: 7,
      field2: 'string'
    }
  }
  util.http.addResponseHeaders(headers)
  return {}
}
```

\$util.xml 中的 XML 協助程式

Note

我們現在主要支援 APPSYNC_JS 執行期及其文件。請考慮[在此處](#)使用 APPSYNC_JS 執行期及其指南。

\$util.xml 包含協助程式方法，可讓您更輕鬆地將 XML 回應轉譯為 JSON 或字典。

\$util.xml utils 清單

\$util.xml.toMap(String) : Map

將 XML 字串轉換為字典。

Input:

```
<?xml version="1.0" encoding="UTF-8"?>
<posts>
<post>
  <id>1</id>
  <title>Getting started with GraphQL</title>
</post>
</posts>
```

Output (JSON representation):

```
{
  "posts":{
    "post":{
      "id":1,
      "title":"Getting started with GraphQL"
    }
  }
}
```

Input:

```
<?xml version="1.0" encoding="UTF-8"?>
<posts>
<post>
  <id>1</id>
  <title>Getting started with GraphQL</title>
</post>
<post>
  <id>2</id>
  <title>Getting started with AWS AppSync</title>
</post>
</posts>
```

Output (JSON representation):

```
{
  "posts": {
    "post": [
      {
        "id": 1,
        "title": "Getting started with GraphQL"
      },
      {
        "id": 2,
        "title": "Getting started with AWS AppSync"
      }
    ]
  }
}
```

\$util.xml.toJsonString(String) : String

將 XML 字串轉換為 JSON 字串。這類似於 toMap，但輸出是字串。如果要將來自 HTTP 物件的 XML 回應直接轉換並傳回給 JSON，這非常實用。

\$util.xml.toJsonString(String, Boolean) : String

將 XML 字串轉換為具有選用布林值參數的 JSON 字串，以判斷您是否要對 JSON 進行字串編碼。

\$util.transform 中的轉換協助程式

Note

我們現在主要支援 APPSYNC_JS 執行期及其文件。請考慮[在此處](#)使用 APPSYNC_JS 執行期及其指南。

\$util.transform 包含協助程式方法，可讓您更輕鬆地對資料來源執行複雜的操作，例如 Amazon DynamoDB 篩選操作。

轉換協助程式

轉換協助程式使用清單

`$util.transform.toDynamoDBFilterExpression(Map) : Map`

將輸入字串轉換為篩選條件表達式，以搭配 DynamoDB 使用。

Input:

```
$util.transform.toDynamoDBFilterExpression({
  "title":{
    "contains":"Hello World"
  }
})
```

Output:

```
{
  "expression" : "contains(#title, :title_contains)"
  "expressionNames" : {
    "#title" : "title",
  },
  "expressionValues" : {
    ":title_contains" : { "S" : "Hello World" }
  },
}
```

`$util.transform.toElasticsearchQueryDSL(Map) : Map`

將指定的輸入轉換為其同等的 OpenSearch Query DSL 表達式，將其傳回為 JSON 字串。

Input:

```
$util.transform.toElasticsearchQueryDSL({
  "upvotes":{
    "ne":15,
    "range":[
      10,
      20
    ]
  },
  "title":{
```

```

    "eq": "hihihi",
    "wildcard": "h*i"
  }
})

```

Output:

```

{
  "bool": {
    "must": [
      {
        "bool": {
          "must": [
            {
              "bool": {
                "must_not": {
                  "term": {
                    "upvotes": 15
                  }
                }
              }
            },
            {
              "range": {
                "upvotes": {
                  "gte": 10,
                  "lte": 20
                }
              }
            }
          ]
        }
      },
      {
        "bool": {
          "must": [
            {
              "term": {
                "title": "hihihi"
              }
            },
            {
              "wildcard": {
                "title": "h*i"
              }
            }
          ]
        }
      }
    ]
  }
}

```

```
}
}
}
}
}
}
```

預設運算子假設為 AND。

轉換協助程式訂閱篩選條件

轉換協助程式訂閱篩選條件使用清單

\$util.transform.toSubscriptionFilter(Map) : Map

將Map輸入物件轉換為SubscriptionFilter表達式物件。`$util.transform.toSubscriptionFilter` 方法會用作`$extensions.setSubscriptionFilter()` 延伸模組的輸入。如需詳細資訊，請參閱 [延伸模組](#)。

```
$util.transform.toSubscriptionFilter(Map, List) : Map
```

將Map輸入物件轉換為SubscriptionFilter表達式物件。`$util.transform.toSubscriptionFilter` 方法會用作`$extensions.setSubscriptionFilter()` 延伸模組的輸入。如需詳細資訊，請參閱 [延伸模組](#)。

第一個引數是轉換為SubscriptionFilter表達式物件的Map輸入物件。第二個引數是在建構SubscriptionFilter表達式物件時在第一個Map輸入物件中忽略List的欄位名稱。

```
$util.transform.toSubscriptionFilter(Map, List, Map) : Map
```

將Map輸入物件轉換為SubscriptionFilter表達式物件。`$util.transform.toSubscriptionFilter` 方法會用作`$extensions.setSubscriptionFilter()` 延伸模組的輸入。如需詳細資訊，請參閱 [延伸模組](#)。

第一個引數是轉換為SubscriptionFilter表達式物件的Map輸入物件，第二個引數是在第一個Map輸入物件中忽略List的欄位名稱，第三個引數是建構SubscriptionFilter表達式物件時包含的嚴格規則的Map輸入物件。這些嚴格的規則包含在SubscriptionFilter表達式物件中，方式是至少滿足其中一個規則以傳遞訂閱篩選條件。

訂閱篩選條件引數

下表說明如何定義下列公用程式的引數：

- `$util.transform.toSubscriptionFilter(Map)` : Map
- `$util.transform.toSubscriptionFilter(Map, List)` : Map
- `$util.transform.toSubscriptionFilter(Map, List, Map)` : Map

Argument 1: Map

引數 1 是具有下列索引鍵值的Map物件：

- 欄位名稱
- 「和」
- "或"

對於做為索引鍵的欄位名稱，這些欄位項目的條件格式為 "operator" : "value"。

下列範例顯示如何將項目新增至 Map：

```
"field_name" : {
    "operator1" : value
}

## We can have multiple conditions for the same field_name:

"field_name" : {
    "operator1" : value
    "operator2" : value
    .
    .
    .
}
```

當欄位上有兩個或多個條件時，所有這些條件都會被視為使用 OR 操作。

輸入Map也可以將「和」和「或」做為索引鍵，表示這些輸入中的所有項目都應該使用 AND 或 OR 邏輯聯結，視索引鍵而定。索引鍵值「和」和「或」需要條件陣列。

```
"and" : [
```

```

    {
      "field_name1" : {
        "operator1" : value
      }
    },
    {
      "field_name2" : {
        "operator1" : value
      }
    },
    :
    .
  ].

```

請注意，您可以巢狀化「和」和「或」。也就是說，您可以在另一個「和」/「或」區塊中巢狀「和」/「或」。不過，這不適用於簡單欄位。

```

"and" : [
  {
    "field_name1" : {
      "operator" : value
    }
  },
  {
    "or" : [
      {
        "field_name2" : {
          "operator" : value
        }
      },
      {
        "field_name3" : {
          "operator" : value
        }
      }
    ]
  }
].

```

下列範例顯示使用的引數 1 輸入 `$util.transform.toSubscriptionFilter(Map)` :
Map。

Input(s)

引數 1：映射：

```
{
  "percentageUp": {
    "lte": 50,
    "gte": 20
  },
  "and": [
    {
      "title": {
        "ne": "Book1"
      }
    },
    {
      "downvotes": {
        "gt": 2000
      }
    }
  ],
  "or": [
    {
      "author": {
        "eq": "Admin"
      }
    },
    {
      "isPublished": {
        "eq": false
      }
    }
  ]
}
```

輸出

結果是 Map 物件：

```
{
```



```
"filterGroup": [
  {
    "filters": [
      {
        "fieldName": "percentageUp",
        "operator": "lte",
        "value": 50
      },
      {
        "fieldName": "title",
        "operator": "ne",
        "value": "Book1"
      },
      {
        "fieldName": "downvotes",
        "operator": "gt",
        "value": 2000
      },
      {
        "fieldName": "author",
        "operator": "eq",
        "value": "Admin"
      }
    ]
  },
  {
    "filters": [
      {
        "fieldName": "percentageUp",
        "operator": "lte",
        "value": 50
      },
      {
        "fieldName": "title",
        "operator": "ne",
        "value": "Book1"
      },
      {
        "fieldName": "downvotes",
        "operator": "gt",
        "value": 2000
      },
      {
        "fieldName": "isPublished",
```

```
        "operator": "eq",
        "value": false
      }
    ],
  },
  {
    "filters": [
      {
        "fieldName": "percentageUp",
        "operator": "gte",
        "value": 20
      },
      {
        "fieldName": "title",
        "operator": "ne",
        "value": "Book1"
      },
      {
        "fieldName": "downvotes",
        "operator": "gt",
        "value": 2000
      },
      {
        "fieldName": "author",
        "operator": "eq",
        "value": "Admin"
      }
    ]
  },
  {
    "filters": [
      {
        "fieldName": "percentageUp",
        "operator": "gte",
        "value": 20
      },
      {
        "fieldName": "title",
        "operator": "ne",
        "value": "Book1"
      },
      {
        "fieldName": "downvotes",
        "operator": "gt",
```

```

        "value": 2000
      },
      {
        "fieldName": "isPublished",
        "operator": "eq",
        "value": false
      }
    ]
  }
]
}

```

Argument 2: List

引數 2 包含建構SubscriptionFilter表達式物件時不應在輸入中考慮List的一組欄位名稱Map (引數 1)。List 也可以是空的。

下列範例顯示使用的引數 1 和引數 2 的輸入

入\$util.transform.toSubscriptionFilter(Map, List) : Map。

Input(s)

引數 1 : 映射 :

```

{
  "percentageUp": {
    "lte": 50,
    "gte": 20
  },
  "and": [
    {
      "title": {
        "ne": "Book1"
      }
    },
    {
      "downvotes": {
        "gt": 20
      }
    }
  ],
  "or": [
    {

```

```
    "author": {
      "eq": "Admin"
    },
    {
      "isPublished": {
        "eq": false
      }
    }
  ]
}
```

引數 2：列出：

```
["percentageUp", "author"]
```

輸出

結果是 Map 物件：

```
{
  "filterGroup": [
    {
      "filters": [
        {
          "fieldName": "title",
          "operator": "ne",
          "value": "Book1"
        },
        {
          "fieldName": "downvotes",
          "operator": "gt",
          "value": 20
        },
        {
          "fieldName": "isPublished",
          "operator": "eq",
          "value": false
        }
      ]
    }
  ]
}
```

Argument 3: Map

引數 3 是具有欄位名稱作為索引鍵值的Map物件（不能具有「和」或「或」）。對於做為索引鍵的欄位名稱，這些欄位的條件是形式的項目"operator" : "value"。與引數 1 不同，引數 3 不能在相同的索引鍵中有多個條件。此外，引數 3 沒有「和」或「或」子句，因此也沒有涉及巢狀。

引數 3 代表嚴格規則的清單，這些規則會新增至SubscriptionFilter表達式物件，因此至少符合其中一個條件來傳遞篩選條件。

```
{
  "fieldname1": {
    "operator": value
  },
  "fieldname2": {
    "operator": value
  }
}
.
.
.
```

下列範例顯示使用的引數 1、引數 2 和引數 3 的輸

入\$util.transform.toSubscriptionFilter(Map, List, Map) : Map。

Input(s)

引數 1：映射：

```
{
  "percentageUp": {
    "lte": 50,
    "gte": 20
  },
  "and": [
    {
      "title": {
        "ne": "Book1"
      }
    },
    {
      "downvotes": {
        "lt": 20
      }
    }
  ]
}
```

```
    }
  ],
  "or": [
    {
      "author": {
        "eq": "Admin"
      }
    },
    {
      "isPublished": {
        "eq": false
      }
    }
  ]
}
```

引數 2：列出：

```
["percentageUp", "author"]
```

引數 3：映射：

```
{
  "upvotes": {
    "gte": 250
  },
  "author": {
    "eq": "Person1"
  }
}
```

輸出

結果是 Map 物件：

```
{
  "filterGroup": [
    {
      "filters": [
        {
          "fieldName": "title",
          "operator": "ne",
          "value": "Book1"
        }
      ]
    }
  ]
}
```

```
    },
    {
      "fieldName": "downvotes",
      "operator": "gt",
      "value": 20
    },
    {
      "fieldName": "isPublished",
      "operator": "eq",
      "value": false
    },
    {
      "fieldName": "upvotes",
      "operator": "gte",
      "value": 250
    }
  ]
},
{
  "filters": [
    {
      "fieldName": "title",
      "operator": "ne",
      "value": "Book1"
    },
    {
      "fieldName": "downvotes",
      "operator": "gt",
      "value": 20
    },
    {
      "fieldName": "isPublished",
      "operator": "eq",
      "value": false
    },
    {
      "fieldName": "author",
      "operator": "eq",
      "value": "Person1"
    }
  ]
}
]
```

```
}
```

\$util.math 中的數學協助程式

Note

我們現在主要支援 APPSYNC_JS 執行期及其文件。請考慮[在此處](#)使用 APPSYNC_JS 執行期及其指南。

\$util.math 包含有助於常見數學操作的方法。

\$util.math utils 清單

\$util.math.roundNum(Double) : Integer

採取雙數並四捨五入到最接近的整數。

\$util.math.minVal(Double, Double) : Double

需要兩個倍數，並傳回兩個倍數之間的最小值。

\$util.math.maxVal(Double, Double) : Double

需要兩個倍數，並傳回兩個倍數之間的最大值。

\$util.math.randomDouble() : Double

傳回介於 0 和 1 之間的隨機倍數。

Important

此函數不應用於需要高熵隨機性（例如密碼編譯）的任何內容。

\$util.math.randomWithinRange(Integer, Integer) : Integer

傳回指定範圍內的隨機整數值，第一個引數指定範圍的較低值，第二個引數指定範圍的上限值。

Important

此函數不應用於需要高熵隨機性（例如密碼編譯）的任何內容。

\$util.str 中的字串協助程式

Note

我們現在主要支援 APPSYNC_JS 執行期及其文件。請考慮[在此處](#)使用 APPSYNC_JS 執行期及其指南。

\$util.str 包含有助於常見字串操作的方法。

\$util.str utils 清單

\$util.str.toUpper(String) : String

採用字串並將其轉換為完全大寫。

\$util.str.toLowerCase(String) : String

使用字串並將其轉換為完全小寫。

\$util.str.replace(String, String, String) : String

將字串中的子字串取代為另一個字串。第一個引數指定要在其中執行取代操作的字串。第二個引數指定要取代的子字串。第三個引數指定要取代第二個引數的字串。以下是此公用程式使用的範例：

```
INPUT:      $util.str.replace("hello world", "hello", "mellow")
OUTPUT:     "mellow world"
```

\$util.str.normalize(String, String) : String

使用四個 Unicode 標準化形式之一來標準化字串：NFC、NFD、NFKC 或 NFKD。第一個引數是要標準化的字串。第二個引數是 "nfc"、"nfd"、"nfkc" 或 "nfkd"，指定用於標準化程序的標準化類型。

延伸模組

Note

我們現在主要支援 APPSYNC_JS 執行期及其文件。請考慮[在此處](#)使用 APPSYNC_JS 執行期及其指南。

`$extensions` 包含一組方法，可在解析程式中執行其他動作。

快取擴充功能

`$extensions.evictFromApiCache(String, String, Object) : Object`

從 AWS AppSync 伺服器端快取中移出項目。第一個引數是類型名稱。第二個引數是欄位名稱。第三個引數是包含金鑰值對項目的物件，可指定快取金鑰值。您必須以與快取解析程式中的快取金鑰相同的順序，將項目放入物件中 `cachingKey`。

Note

此公用程式僅適用於變動，不適用於查詢。

訂閱擴充功能

`$extensions.setSubscriptionFilter(filterJsonObject)`

定義增強型訂閱篩選條件。每個訂閱通知事件會根據提供的訂閱篩選條件進行評估，並在所有篩選條件評估為 `true` 時傳送通知給用戶端 `true`。引數 `filterJsonObject` 如下節所述。

Note

您只能在訂閱解析程式的回應映射範本中使用此延伸方法。

`$extensions.setSubscriptionInvalidationFilter(filterJsonObject)`

定義訂閱失效篩選條件。系統會針對失效承載評估訂閱篩選條件，如果篩選條件評估為 `true`，則會使指定的訂閱失效 `true`。引數 `filterJsonObject` 如下節所述。

Note

您只能在訂閱解析程式的回應映射範本中使用此延伸方法。

`$extensions.invalidateSubscriptions(invalidationJsonObject)`

用來從變動啟動訂閱失效。引數 `invalidationJsonObject` 如下節所述。

 Note

此延伸模組只能用於變動解析程式的回應映射範本。

在任何單一請求中，您最多只能使用五個唯一

的 `$extensions.invalidateSubscriptions()` 方法呼叫。如果您超過此限制，您將會收到 GraphQL 錯誤。

引數：filterJsonObject

JSON 物件會定義訂閱或失效篩選條件。這是 中的一系列篩選條件 `filterGroup`。每個篩選條件都是個別篩選條件的集合。

```
{
  "filterGroup": [
    {
      "filters": [
        {
          "fieldName": "userId",
          "operator": "eq",
          "value": 1
        }
      ]
    },
    {
      "filters": [
        {
          "fieldName": "group",
          "operator": "in",
          "value": ["Admin", "Developer"]
        }
      ]
    }
  ]
}
```

每個篩選條件都有三個屬性：

- `fieldName` – GraphQL 結構描述欄位。

- **operator** – 運算子類型。
- **value** – 要與訂閱通知值比較fieldName的值。

以下是這些屬性的範例指派：

```
{
  "fieldName" : "severity",
  "operator" : "le",
  "value" : $context.result.severity
}
```

欄位：fieldName

字串類型fieldName是指 GraphQL 結構描述中定義的欄位，符合訂閱通知承載fieldName中的。找到相符項目時，GraphQL 結構描述欄位value的 會與訂閱通知篩選條件value的 進行比較。在下列範例中，fieldName篩選條件符合指定 GraphQL 類型中定義的service欄位。如果通知承載包含的service 欄位value等同於 AWS AppSync，則篩選條件會評估為 true：

```
{
  "fieldName" : "service",
  "operator" : "eq",
  "value" : "AWS AppSync"
}
```

欄位：值

根據運算子，值可以是不同的類型：

- 單一數字或布林值
 - 字串範例："test"、"service"
 - 數字範例：1、2、45.75
 - 布林值範例：true、false
- 一組數字或字串
 - 字串對範例：["test1","test2"]、["start","end"]
 - 數字對範例：[1,4]、[67,89]、[12.45, 95.45]
- 數字或字串陣列
 - 字串陣列範例：["test1","test2","test3","test4","test5"]

- 數字陣列範例：[1,2,3,4,5]、[12.11,46.13,45.09,12.54,13.89]

欄位：運算子

具有下列可能值的區分大小寫字串：

運算子	描述	可能的值類型
eq	Equal	integer, float, string, Boolean
ne	Not equal	integer, float, string, Boolean
le	Less than or equal	integer, float, string
lt	Less than	integer, float, string
ge	Greater than or equal	integer, float, string
gt	Greater than	integer, float, string
contains	Checks for a subsequence or value in the set.	integer, float, string
notContains	Checks for the absence of a subsequence or absence of a value in the set.	integer, float, string
beginsWith	Checks for a prefix.	string
in	Checks for matching elements that are in the list.	Array of integer, float, or string
notIn	Checks for matching elements that aren't in the list.	Array of integer, float, or string
between	Between two values	integer, float, string
containsAny	Contains common elements	integer, float, string

下表說明訂閱通知中每個運算子的使用方式。

eq (equal)

如果true訂閱通知欄位值相符且嚴格等於篩選條件的值，則eq運算子會評估為。在下列範例中，篩選條件會評估訂閱通知true是否有值等於 service的欄位AWS AppSync。

可能的值類型：整數、浮點數、字串、布林值

```
{
  "fieldName" : "service",
  "operator" : "eq",
  "value" : "AWS AppSync"
}
```

ne (not equal)

如果true訂閱通知欄位值與篩選條件的值不同，運算ne子會評估為。在下列範例中，篩選條件會評估訂閱通知true是否有值與 不同的service欄位AWS AppSync。

可能的值類型：整數、浮點數、字串、布林值

```
{
  "fieldName" : "service",
  "operator" : "ne",
  "value" : "AWS AppSync"
}
```

le (less or equal)

如果true訂閱通知欄位值小於或等於篩選條件的值，le運算子會評估為。在下列範例中，篩選條件會評估訂閱通知true是否有值小於或等於 size的欄位5。

可能的值類型：整數、浮點數、字串

```
{
  "fieldName" : "size",
  "operator" : "le",
  "value" : 5
}
```

lt (less than)

如果true訂閱通知欄位值低於篩選條件的值，運算lt子會評估為。在下列範例中，篩選條件會評估訂閱通知true是否有值低於 size的欄位5。

可能的值類型：整數、浮點數、字串

```
{
  "fieldName" : "size",
  "operator" : "lt",
  "value" : 5
}
```

ge (greater or equal)

如果true訂閱通知欄位值大於或等於篩選條件的值，運算ge子會評估為。在下列範例中，篩選條件會評估訂閱通知true是否有欄位size的值大於或等於5。

可能的值類型：整數、浮點數、字串

```
{
  "fieldName" : "size",
  "operator" : "ge",
  "value" : 5
}
```

gt (greater than)

如果true訂閱通知欄位值大於篩選條件的值，則gt運算子會評估為。在下列範例中，true如果訂閱通知具有的值大於size的欄位，則篩選條件會評估為5。

可能的值類型：整數、浮點數、字串

```
{
  "fieldName" : "size",
  "operator" : "gt",
  "value" : 5
}
```

contains

contains 運算子會檢查集合或單一項目中的子字串、子序列或值。如果訂閱通知欄位值包含篩選條件值，則具有contains運算子的篩選條件會評估true為。在下列範例中，篩選條件會評估訂閱通知true是否具有包含值之陣列值seats的欄位10。

可能的值類型：整數、浮點數、字串

```
{
```

```
"fieldName" : "seats",
"operator" : "contains",
"value" : 10
}
```

在另一個範例中，篩選條件會評估訂閱通知true是否具有 launch做為子字串event的欄位。

```
{
  "fieldName" : "event",
  "operator" : "contains",
  "value" : "launch"
}
```

notContains

notContains 運算子會檢查集合或單一項目中是否存在子字串、子序列或值。如果訂閱通知欄位值不包含篩選條件值true，則具有notContains運算子的篩選條件會評估為。在下列範例中，篩選條件會評估訂閱通知true是否具有陣列值不包含值 seats的欄位10。

可能的值類型：整數、浮點數、字串

```
{
  "fieldName" : "seats",
  "operator" : "notContains",
  "value" : 10
}
```

在另一個範例中，篩選條件會評估訂閱通知true是否具有 launch作為其子序列的 event 欄位值。

```
{
  "fieldName" : "event",
  "operator" : "notContains",
  "value" : "launch"
}
```

beginsWith

beginsWith 運算子會檢查字串中的字首。如果訂閱通知欄位值以篩選條件的值開頭true，則包含beginsWith運算子的篩選條件會評估為。在下列範例中，篩選條件會評估訂閱通知true是否有值開頭為 service的欄位AWS。

可能的值類型：字串

```
{
  "fieldName" : "service",
  "operator" : "beginsWith",
  "value" : "AWS"
}
```

in

in 運算子會檢查陣列中的相符元素。如果true訂閱通知欄位值存在於陣列中，則包含in運算子的篩選條件會評估為true。在下列範例中，篩選條件會評估訂閱通知true是否有severity欄位具有陣列中存在的其中一個值：[1,2,3]。

可能的值類型：整數陣列、浮點數或字串

```
{
  "fieldName" : "severity",
  "operator" : "in",
  "value" : [1,2,3]
}
```

notIn

notIn 運算子會檢查陣列中缺少的元素。如果訂閱通知欄位值不存在於陣列中true，則包含notIn運算子的篩選條件會評估為true。在下列範例中，篩選條件會評估訂閱通知true是否有severity欄位，其中包含陣列中不存在的其中一個值：[1,2,3]。

可能的值類型：整數陣列、浮點數或字串

```
{
  "fieldName" : "severity",
  "operator" : "notIn",
  "value" : [1,2,3]
}
```

between

between 運算子會檢查兩個數字或字串之間的值。如果訂閱通知欄位值介於篩選條件的值對之間true，則包含between運算子的篩選條件會評估為true。在下列範例中，篩選條件會評估訂閱通知true是否有值為 2、3、severity的欄位4。

可能的值類型：整數、浮點數或字串的配對

```
{
  "fieldName" : "severity",
  "operator" : "between",
  "value" : [1,5]
}
```

containsAny

`containsAny` 運算子會檢查陣列中的常見元素。如果訂閱通知欄位集值和篩選條件集值的交集不是空的`true`，則具有`containsAny`運算子的篩選條件會評估為 `true`。在下列範例中，篩選條件會評估訂閱通知`true`是否有`seats`欄位的陣列值包含 10或 15。這表示`true`如果訂閱通知`seats`的欄位值為 `[10,11]`或 `[15,20,30]`，篩選條件會評估為 `true`。

可能的值類型：整數、浮點數或字串

```
{
  "fieldName" : "seats",
  "operator" : "containsAny",
  "value" : [10, 15]
}
```

AND 邏輯

您可以使用 AND 邏輯結合多個篩選條件，方法是在`filterGroup`陣列中的 `filters` 物件中定義多個項目。在下列範例中，篩選條件會評估訂閱通知`true`是否有等於 1 AND `userId` 欄位值 `Admin`或 `group`的欄位`Developer`。

```
{
  "filterGroup": [
    {
      "filters" : [
        {
          "fieldName" : "userId",
          "operator" : "eq",
          "value" : 1
        },
        {
          "fieldName" : "group",
```

```

        "operator" : "in",
        "value" : ["Admin", "Developer"]
      }
    ]
  }
}

```

OR 邏輯

您可以在filterGroup陣列中定義多個篩選條件物件，以使用 OR 邏輯結合多個篩選條件。在下列範例中，篩選條件會評估訂閱通知true是否有欄位userId的值等於 1 OR group 欄位值 Admin或Developer。

```

{
  "filterGroup": [
    {
      "filters" : [
        {
          "fieldName" : "userId",
          "operator" : "eq",
          "value" : 1
        }
      ]
    },
    {
      "filters" : [
        {
          "fieldName" : "group",
          "operator" : "in",
          "value" : ["Admin", "Developer"]
        }
      ]
    }
  ]
}

```

例外狀況

請注意，使用篩選條件有幾項限制：

- 在 `filters` 物件中，每個篩選條件最多可以有五個唯一 `fieldName` 項目。這表示您最多可以使用 AND 邏輯結合五個個別 `fieldName` 物件。
- 運算 `containsAny` 子最多可有 20 個值。
- `in` 和 `notIn` 運算子最多可有五個值。
- 每個字串最多可有 256 個字元。
- 每個字串比較都區分大小寫。
- 巢狀物件篩選最多允許五個巢狀層級的篩選。
- 每個 `filterGroup` 最多可有 10 個 `filters`。這表示您可以使用 `filters` OR 邏輯結合最多 10 個。
- `in` 運算子是 OR 邏輯的特殊案例。在下列範例中，有兩個 `filters`：

```
{
  "filterGroup": [
    {
      "filters": [
        {
          "fieldName": "userId",
          "operator": "eq",
          "value": 1
        },
        {
          "fieldName": "group",
          "operator": "in",
          "value": ["Admin", "Developer"]
        }
      ]
    }
  ]
}
```

上述篩選條件群組的評估如下所示，並計入最大篩選條件限制：

```
{
  "filterGroup": [
    {
      "filters": [
        {
          "fieldName": "userId",
          "operator": "eq",
```

```

        "value" : 1
      },
      {
        "fieldName" : "group",
        "operator" : "eq",
        "value" : "Admin"
      }
    ]
  },
  {
    "filters" : [
      {
        "fieldName" : "userId",
        "operator" : "eq",
        "value" : 1
      },
      {
        "fieldName" : "group",
        "operator" : "eq",
        "value" : "Developer"
      }
    ]
  }
]
}

```

引數：invalidationJsonObject

invalidationJsonObject 定義下列項目：

- **subscriptionField** – 要失效的 GraphQL 結構描述訂閱。定義為 中字串的單一訂閱subscriptionField會被視為失效。
- **payload** – 金鑰值對清單，當失效篩選條件true針對其值評估為 時，此清單會做為使訂閱失效的輸入。

當訂閱解析程式中定義的失效篩選條件true針對 payload值評估為 時，下列範例會使用onUserDelete訂閱使已訂閱和已連線的用戶端失效。

```

$extensions.invalidateSubscriptions({
  "subscriptionField": "onUserDelete",
  "payload": {

```

```
        "group": "Developer"
        "type" : "Full-Time"
    }
})
```

DynamoDB 的 AWS AppSync 解析程式映射範本參考

Note

我們現在主要支援 APPSYNC_JS 執行期及其文件。請考慮[在此處](#)使用 APPSYNC_JS 執行期及其指南。

The AWS AppSync DynamoDB 函數可讓您使用 [GraphQL](#)，透過將傳入的 GraphQL 請求映射至 DynamoDB 呼叫，然後將 DynamoDB 回應映射回 GraphQL，來將資料存放和擷取至您帳戶中現有的 Amazon DynamoDB 資料表。本節說明受支援 DynamoDB 操作的請求和回應處理常式：

- [GetItem](#) - GetItem 請求可讓您指示 DynamoDB 函數向 DynamoDB 提出 GetItem 請求，並可讓您在 DynamoDB 中指定項目的索引鍵，以及是否使用一致的讀取。
- [PutItem](#) - PutItem 請求映射文件可讓您指示 DynamoDB 函數向 DynamoDB 提出 PutItem 請求，並可讓您在 DynamoDB 中指定項目的金鑰、項目的完整內容（由金鑰和attributeValues組成），以及操作成功的條件。
- [UpdateItem](#) - UpdateItem 請求可讓您指示 DynamoDB 函數向 DynamoDB 提出 UpdateItem 請求，並允許您在 DynamoDB 中指定項目的索引鍵、描述如何在 DynamoDB 中更新項目的更新表達式，以及操作成功的條件。
- [DeleteItem](#) - DeleteItem 請求可讓您指示 DynamoDB 函數向 DynamoDB 提出 DeleteItem 請求，並可讓您在 DynamoDB 中指定項目的索引鍵，以及操作成功的條件。
- [查詢](#) - 查詢請求物件可讓您指示 DynamoDB 解析程式向 DynamoDB 提出查詢請求，並可讓您指定索引、要使用的索引、其他篩選條件、要傳回的項目數、是否使用一致的讀取、查詢方向（向前或向後）和分頁字符。
- [掃描](#) - 掃描請求可讓您指示 DynamoDB 函數向 DynamoDB 提出掃描請求，並可讓您指定篩選條件以排除結果、要使用的索引、要傳回的項目數、是否使用一致的讀取、分頁字符和平行掃描。
- [Sync](#) - Sync 請求物件可讓您從 DynamoDB 資料表擷取所有結果，然後只接收自上次查詢後更改的資料（差異更新）。同步請求只能對版本控制的 DynamoDB 資料來源提出。您可以指定篩選條件來排除結果、要傳回的項目數、分頁字符，以及上次同步操作的啟動時間。

- [BatchGetItem](#) - BatchGetItem 請求物件可讓您告知 DynamoDB 函數向 DynamoDB 提出 BatchGetItem 請求，以擷取多個項目，可能跨多個資料表。對於此請求物件，您必須指定要從中擷取項目的資料表名稱，以及要從每個資料表中擷取之項目的索引鍵。
- [BatchDeleteItem](#) - BatchDeleteItem 請求物件可讓您告知 DynamoDB 函數向 DynamoDB 提出 BatchWriteItem 請求，以刪除多個項目，可能跨多個資料表。對於此請求物件，您必須指定要刪除項目的資料表名稱，以及要從每個資料表刪除的項目索引鍵。
- [BatchPutItem](#) - BatchPutItem 請求物件可讓您告知 DynamoDB 函數向 DynamoDB 提出 BatchWriteItem 請求，以放置多個項目，可能跨越多個資料表。對於此請求物件，您必須指定要放入項目的資料表名稱，以及要放入每個資料表的完整項目。
- [TransactGetItems](#) - TransactGetItems 請求物件可讓您告知 DynamoDB 函數向 DynamoDB 提出 TransactGetItems 請求，以擷取多個項目，可能跨多個資料表。對於此請求物件，您必須指定每個請求項目的資料表名稱，以便從中擷取項目，以及每個請求項目的索引鍵，以便從每個資料表擷取。
- [TransactWriteItems](#) - TransactWriteItems 請求物件可讓您指示 DynamoDB 函數向 DynamoDB 提出 TransactWriteItems 請求，以寫入多個項目，可能寫入多個資料表。對於此請求物件，您必須指定每個請求項目的目的地資料表名稱、要執行的每個請求項目的操作，以及要寫入的每個請求項目的索引鍵。
- [類型系統（請求映射）](#) - 進一步了解 DynamoDB 類型如何整合到 AWS AppSync 請求中。
- [類型系統（回應映射）](#) - 進一步了解 DynamoDB 類型如何在回應承載中自動轉換為 GraphQL 或 JSON。
- [篩選條件](#) - 進一步了解查詢和掃描操作的篩選條件。
- [條件表達式](#) - 進一步了解 PutItem、UpdateItem 和 DeleteItem 操作的條件表達式。
- [交易條件表達式](#) - 進一步了解 TransactWriteItems 操作的條件表達式。
- [投影](#) - 進一步了解如何在讀取操作中指定屬性。

GetItem

GetItem 請求映射文件可讓您告知 AWS AppSync DynamoDB 解析程式向 DynamoDB 提出 GetItem 請求，並可讓您指定：

- DynamoDB 中項目的索引鍵
- 是否使用一致性讀取

GetItem 映射文件結構如下：

```
{
  "version" : "2017-02-28",
  "operation" : "GetItem",
  "key" : {
    "foo" : ... typed value,
    "bar" : ... typed value
  },
  "consistentRead" : true,
  "projection" : {
    ...
  }
}
```

欄位定義如下：

GetItem 欄位

GetItem 欄位清單

version

範本定義版本。目前支援 2017-02-28 和 2018-05-29。此值為必填。

operation

要執行的 DynamoDB 操作。若要執行 GetItem DynamoDB 操作，這必須設為 GetItem。此值為必填。

key

DynamoDB 中項目的索引鍵。DynamoDB 項目可能具有單一雜湊索引鍵，或雜湊索引鍵和排序索引鍵，視資料表結構而定。如需如何指定「輸入值」的詳細資訊，請參閱[類型系統（請求映射）](#)。此值為必填。

consistentRead

是否使用 DynamoDB 執行強式一致讀取。此為選用，預設值為 false。

projection

用於指定要從 DynamoDB 操作傳回之屬性的投影。如需投影的詳細資訊，請參閱[投影](#)。此欄位為選用欄位。

從 DynamoDB 傳回的項目會自動轉換為 GraphQL 和 JSON 基本類型，並可在映射內容中使用 (`$context.result`)。

如需 DynamoDB 類型轉換的詳細資訊，請參閱[類型系統（回應映射）](#)。

如需回應映射範本的詳細資訊，請參閱[解析程式映射範本概觀](#)。

範例

下列範例是 GraphQL 查詢的映射範本 `getThing(foo: String!, bar: String!)`：

```
{
  "version" : "2017-02-28",
  "operation" : "GetItem",
  "key" : {
    "foo" : $util.dynamodb.toDynamoDBJson($ctx.args.foo),
    "bar" : $util.dynamodb.toDynamoDBJson($ctx.args.bar)
  },
  "consistentRead" : true
}
```

如需 DynamoDB `GetItem` API 的詳細資訊，請參閱 [DynamoDB API 文件](#)。

PutItem

`PutItem` 請求映射文件可讓您告知 AWS AppSync DynamoDB 解析程式向 DynamoDB 提出 `PutItem` 請求，並可讓您指定下列項目：

- DynamoDB 中項目的索引鍵
- 項目 (由 `key` 和 `attributeValues` 組成) 的完整內容
- 操作成功的條件

`PutItem` 映射文件結構如下：

```
{
  "version" : "2018-05-29",
  "operation" : "PutItem",
  "customPartitionKey" : "foo",
  "populateIndexFields" : boolean value,
  "key": {
```

```
    "foo" : ... typed value,
    "bar" : ... typed value
  },
  "attributeValues" : {
    "baz" : ... typed value
  },
  "condition" : {
    ...
  },
  "_version" : 1
}
```

欄位定義如下：

PutItem 欄位

PutItem 欄位清單

version

範本定義版本。目前支援 2017-02-28 和 2018-05-29。此值為必填。

operation

要執行的 DynamoDB 操作。若要執行 PutItem DynamoDB 操作，這必須設為 PutItem。此值為必填。

key

DynamoDB 中項目的索引鍵。DynamoDB 項目可能具有單一雜湊索引鍵，或雜湊索引鍵和排序索引鍵，視資料表結構而定。如需如何指定「輸入值」的詳細資訊，請參閱[類型系統（請求映射）](#)。此值為必填。

attributeValues

將放入 DynamoDB 的項目其餘屬性。如需如何指定「輸入值」的詳細資訊，請參閱[類型系統（請求映射）](#)。此欄位為選用欄位。

condition

決定要求是否成功的條件，可根據已存在於 DynamoDB 的物件狀態。如果沒有指定條件，PutItem 要求會覆寫該項目的任何現有資料項目。如需條件的詳細資訊，請參閱[條件表達式](#)。此值是選用的。

`_version`

代表項目之最新已知版本的數值。此值是選用的。此欄位用於衝突偵測，而且僅支援已建立版本的資料來源。

`customPartitionKey`

啟用時，此字串值會修改啟用版本控制時差異同步資料表使用的 `ds_sk` 和 `ds_pk` 記錄格式（如需詳細資訊，請參閱 AWS AppSync 開發人員指南中的[衝突偵測和同步](#)）。啟用時，也會啟用 `populateIndexFields` 項目的處理。此欄位為選用欄位。

`populateIndexFields`

布林值，當與一起 `customPartitionKey` 啟用時，會為差異同步資料表中的每個記錄建立新的項目，特別是在 `gsi_ds_pk` 和 `gsi_ds_sk` 欄中。如需詳細資訊，請參閱 AWS AppSync 開發人員指南中的[衝突偵測和同步](#)。此欄位為選用欄位。

寫入 DynamoDB 的項目會自動轉換為 GraphQL 和 JSON 基本類型，並可在映射內容中使用 (`$context.result`)。

如需 DynamoDB 類型轉換的詳細資訊，請參閱[類型系統（回應映射）](#)。

如需回應映射範本的詳細資訊，請參閱[解析程式映射範本概觀](#)。

範例 1

下列範例是 GraphQL 變動 的映射範本 `updateThing(foo: String!, bar: String!, name: String!, version: Int!)`。

若無指定索引鍵的項目，則會建立該項目。若已有指定索引鍵的項目，則會覆寫該項目。

```
{
  "version" : "2017-02-28",
  "operation" : "PutItem",
  "key": {
    "foo" : $util.dynamodb.toDynamoDBJson($ctx.args.foo),
    "bar" : $util.dynamodb.toDynamoDBJson($ctx.args.bar)
  },
  "attributeValues" : {
    "name" : $util.dynamodb.toDynamoDBJson($ctx.args.name),
    "version" : $util.dynamodb.toDynamoDBJson($ctx.args.version)
  }
}
```

```
}
```

範例 2

下列範例是 GraphQL 變動的映射範本 `updateThing(foo: String!, bar: String!, name: String!, expectedVersion: Int!)`。

此範例會檢查，確認目前在 DynamoDB 中的項目已將 `version` 欄位設定為 `expectedVersion`。

```
{
  "version" : "2017-02-28",
  "operation" : "PutItem",
  "key": {
    "foo" : $util.dynamodb.toDynamoDBJson($ctx.args.foo),
    "bar" : $util.dynamodb.toDynamoDBJson($ctx.args.bar)
  },
  "attributeValues" : {
    "name" : $util.dynamodb.toDynamoDBJson($ctx.args.name),
    #set( $newVersion = $context.arguments.expectedVersion + 1 )
    "version" : $util.dynamodb.toDynamoDBJson($newVersion)
  },
  "condition" : {
    "expression" : "version = :expectedVersion",
    "expressionValues" : {
      ":expectedVersion" : $util.dynamodb.toDynamoDBJson($expectedVersion)
    }
  }
}
```

如需 DynamoDB PutItem API 的詳細資訊，請參閱 [DynamoDB API 文件](#)。

UpdateItem

UpdateItem 請求映射文件可讓您指示 AWS AppSync DynamoDB 解析程式向 DynamoDB 提出 UpdateItem 請求，並允許您指定下列項目：

- DynamoDB 中項目的索引鍵
- 描述如何在 DynamoDB 中更新項目的更新表達式
- 操作成功的條件

UpdateItem 映射文件結構如下：

```
{
  "version" : "2018-05-29",
  "operation" : "UpdateItem",
  "customPartitionKey" : "foo",
  "populateIndexFields" : boolean value,
  "key": {
    "foo" : ... typed value,
    "bar" : ... typed value
  },
  "update" : {
    "expression" : "someExpression",
    "expressionNames" : {
      "#foo" : "foo"
    },
    "expressionValues" : {
      ":bar" : ... typed value
    }
  },
  "condition" : {
    ...
  },
  "_version" : 1
}
```

欄位定義如下：

UpdateItem 欄位

UpdateItem 欄位清單

version

範本定義版本。目前支援 2017-02-28 和 2018-05-29。此值為必填。

operation

要執行的 DynamoDB 操作。若要執行 UpdateItem DynamoDB 操作，這必須設為 UpdateItem。此值為必填。

key

DynamoDB 中項目的索引鍵。DynamoDB 項目可能具有單一雜湊索引鍵，或雜湊索引鍵和排序索引鍵，視資料表結構而定。如需指定「輸入值」的詳細資訊，請參閱[類型系統（請求映射）](#)。此值為必填。

update

update 本節可讓您指定更新表達式，描述如何在 DynamoDB 中更新項目。如需有關如何撰寫更新表達式的詳細資訊，請參閱 [DynamoDB UpdateExpressions 文件](#)。此區段是必須的。

update 區段有三個元件：

expression

更新表達式。此值為必填。

expressionNames

表達式屬性 **name** 預留位置的替代，形式為鍵值組。索引鍵對應至 中使用的名稱預留位置 **expression**，且值必須是對應至 DynamoDB 中項目屬性名稱的字串。此欄位為選用的，應只能填入用於 **expression** 中表達式屬性名稱預留位置的替代。

expressionValues

表達式屬性 **value** 預留位置的替代，形式為鍵值組。鍵對應用於 **expression** 的值預留位置，值必須是類型值。如需如何指定「輸入值」的詳細資訊，請參閱[類型系統（請求映射）](#)。此必須指定。此欄位為選用的，應只能填入用於 **expression** 中表達式屬性值預留位置的替代。

condition

決定要求是否成功的條件，可根據已存在於 DynamoDB 的物件狀態。如果沒有指定條件，**UpdateItem** 要求會更新現有的資料項目，無論項目的目前狀態為何。如需條件的詳細資訊，請參閱[條件表達式](#)。此值是選用的。

_version

代表項目之最新已知版本的數值。此值是選用的。此欄位用於衝突偵測，而且僅支援已建立版本的資料來源。

customPartitionKey

啟用時，此字串值會修改啟用版本控制時差異同步資料表使用的 **ds_sk** 和 **ds_pk** 記錄格式（如需詳細資訊，請參閱 AWS AppSync 開發人員指南中的[衝突偵測和同步](#)）。啟用時，也會啟用 **populateIndexFields** 項目的處理。此欄位為選用欄位。

populateIndexFields

布林值，當與一起 **customPartitionKey** 啟用時，會為差異同步資料表中的每個記錄建立新的項目，特別是在 **gsi_ds_pk** 和 **gsi_ds_sk** 欄中。如需詳細資訊，請參閱 AWS AppSync 開發人員指南中的[衝突偵測和同步](#)。此欄位為選用欄位。

DynamoDB 中更新的項目會自動轉換為 GraphQL 和 JSON 基本類型，並可在映射內容中使用 (`$context.result`)。

如需 DynamoDB 類型轉換的詳細資訊，請參閱[類型系統（回應映射）](#)。

如需回應映射範本的詳細資訊，請參閱[解析程式映射範本概觀](#)。

範例 1

下列範例是 GraphQL 變動 的映射範本 `upvote(id: ID!)`。

在此範例中，DynamoDB 中的項目的 `upvotes` 和 `version` 欄位增量為 1。

```
{
  "version" : "2017-02-28",
  "operation" : "UpdateItem",
  "key" : {
    "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
  },
  "update" : {
    "expression" : "ADD #votefield :plusOne, version :plusOne",
    "expressionNames" : {
      "#votefield" : "upvotes"
    },
    "expressionValues" : {
      ":plusOne" : { "N" : 1 }
    }
  }
}
```

範例 2

下列範例是 GraphQL 變動 的映射範本 `updateItem(id: ID!, title: String, author: String, expectedVersion: Int!)`。

這個複雜的範例會檢查引數，並持續產生更新表達式，其只包含由用戶端提供的引數。例如，如果 `title` 和 `author` 遭到省略，則不會更新。如果指定引數，但其值為 `null`，則該欄位會從 DynamoDB 中的物件中刪除。最後，操作有一個條件，可驗證目前在 DynamoDB 中的項目是否將 `version` 欄位設定為 `expectedVersion`：

```
{
  "version" : "2017-02-28",
```

```

"operation" : "UpdateItem",

"key" : {
  "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
},

## Set up some space to keep track of things we're updating **
#set( $expNames = {} )
#set( $expValues = {} )
#set( $expSet = {} )
#set( $expAdd = {} )
#set( $expRemove = [] )

## Increment "version" by 1 **
${!expAdd.put("version", ":newVersion")}
${!expValues.put(":newVersion", { "N" : 1 })}

## Iterate through each argument, skipping "id" and "expectedVersion" **
#foreach( $entry in $context.arguments.entrySet() )
  #if( $entry.key != "id" && $entry.key != "expectedVersion" )
    #if( (!$entry.value) && ("${!entry.value}" == "") )
      ## If the argument is set to "null", then remove that attribute from
the item in DynamoDB **

      #set( $discard = ${expRemove.add("#${entry.key}")} )
      ${!expNames.put("#${entry.key}", "${entry.key}")}
    #else
      ## Otherwise set (or update) the attribute on the item in DynamoDB **

      ${!expSet.put("#${entry.key}", ":${entry.key}")}
      ${!expNames.put("#${entry.key}", "${entry.key}")}

      #if( $entry.key == "ups" || $entry.key == "downs" )
        ${!expValues.put(":${entry.key}", { "N" : $entry.value })}
      #else
        ${!expValues.put(":${entry.key}", { "S" : "${entry.value}" })}
      #end
    #end
  #end
#end

## Start building the update expression, starting with attributes we're going to
SET **
#set( $expression = "" )

```



```

    if( !${expSet.isEmpty()} )
        set( $expression = "SET" )
        foreach( $entry in $expSet.entrySet() )
            set( $expression = "${expression} ${entry.key} = ${entry.value}" )
            if ( $foreach.hasNext )
                set( $expression = "${expression}," )
            end
        end
    end

    ## Continue building the update expression, adding attributes we're going to ADD **
    if( !${expAdd.isEmpty()} )
        set( $expression = "${expression} ADD" )
        foreach( $entry in $expAdd.entrySet() )
            set( $expression = "${expression} ${entry.key} ${entry.value}" )
            if ( $foreach.hasNext )
                set( $expression = "${expression}," )
            end
        end
    end

    ## Continue building the update expression, adding attributes we're going to REMOVE
    **
    if( !${expRemove.isEmpty()} )
        set( $expression = "${expression} REMOVE" )

        foreach( $entry in $expRemove )
            set( $expression = "${expression} ${entry}" )
            if ( $foreach.hasNext )
                set( $expression = "${expression}," )
            end
        end
    end

    ## Finally, write the update expression into the document, along with any
    expressionNames and expressionValues **
    "update" : {
        "expression" : "${expression}"
        if( !${expNames.isEmpty()} )
            , "expressionNames" : $utils.toJson($expNames)
        end
        if( !${expValues.isEmpty()} )
            , "expressionValues" : $utils.toJson($expValues)
        end
    }

```

```
    },  
  
    "condition" : {  
      "expression" : "version = :expectedVersion",  
      "expressionValues" : {  
        ":expectedVersion" :  
$util.dynamodb.toDynamoDBJson($ctx.args.expectedVersion)  
      }  
    }  
  }  
}
```

如需 DynamoDB UpdateItem API 的詳細資訊，請參閱 [DynamoDB API 文件](#)。

DeleteItem

DeleteItem 請求映射文件可讓您告知 AWS AppSync DynamoDB 解析程式向 DynamoDB 提出DeleteItem請求，並可讓您指定下列項目：

- DynamoDB 中項目的索引鍵
- 操作成功的條件

DeleteItem 映射文件結構如下：

```
{  
  "version" : "2018-05-29",  
  "operation" : "DeleteItem",  
  "customPartitionKey" : "foo",  
  "populateIndexFields" : boolean value,  
  "key": {  
    "foo" : ... typed value,  
    "bar" : ... typed value  
  },  
  "condition" : {  
    ...  
  },  
  "_version" : 1  
}
```

欄位定義如下：

DeleteItem 欄位

DeleteItem 欄位清單

version

範本定義版本。目前支援 2017-02-28 和 2018-05-29。此值為必填。

operation

要執行的 DynamoDB 操作。若要執行 DeleteItem DynamoDB 操作，這必須設為 DeleteItem。此值為必填。

key

DynamoDB 中項目的索引鍵。DynamoDB 項目可能具有單一雜湊索引鍵，或雜湊索引鍵和排序索引鍵，視資料表結構而定。如需指定「輸入值」的詳細資訊，請參閱[類型系統（請求映射）](#)。此值為必填。

condition

決定要求是否成功的條件，可根據已存在於 DynamoDB 的物件狀態。如果沒有指定條件，DeleteItem 要求會刪除項目，無論該項目的目前狀態為何。如需條件的詳細資訊，請參閱[條件表達式](#)。此值是選用的。

_version

代表項目之最新已知版本的數值。此值是選用的。此欄位用於衝突偵測，而且僅支援已建立版本的資料來源。

customPartitionKey

啟用時，此字串值會修改啟用版本控制時差異同步資料表使用的 ds_sk 和 ds_pk 記錄格式（如需詳細資訊，請參閱 AWS AppSync 開發人員指南中的[衝突偵測和同步](#)）。啟用時，也會啟用 populateIndexFields 項目的處理。此欄位為選用欄位。

populateIndexFields

布林值，當與一起 customPartitionKey 啟用時，會為差異同步資料表中的每個記錄建立新的項目，特別是在 gsi_ds_pk 和 gsi_ds_sk 欄中。如需詳細資訊，請參閱 AWS AppSync 開發人員指南中的[衝突偵測和同步](#)。此欄位為選用欄位。

從 DynamoDB 刪除的項目會自動轉換為 GraphQL 和 JSON 基本類型，並可在映射內容中使用 (\$context.result)。

如需 DynamoDB 類型轉換的詳細資訊，請參閱[類型系統（回應映射）](#)。

如需回應映射範本的詳細資訊，請參閱[解析程式映射範本概觀](#)。

範例 1

下列範例是 GraphQL 變動的映射範本 `deleteItem(id: ID!)`。如果已存在此 ID 的項目，則該項目將會遭到刪除。

```
{
  "version" : "2017-02-28",
  "operation" : "DeleteItem",
  "key" : {
    "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
  }
}
```

範例 2

下列範例是 GraphQL 變動的映射範本 `deleteItem(id: ID!, expectedVersion: Int!)`。如果已存在此 ID 的項目，則該項目將會遭到刪除，但其 `version` 欄位必須已設為 `expectedVersion`：

```
{
  "version" : "2017-02-28",
  "operation" : "DeleteItem",
  "key" : {
    "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
  },
  "condition" : {
    "expression" : "attribute_not_exists(id) OR version = :expectedVersion",
    "expressionValues" : {
      ":expectedVersion" : $util.dynamodb.toDynamoDBJson($expectedVersion)
    }
  }
}
```

如需 DynamoDB DeleteItem API 的詳細資訊，請參閱 [DynamoDB API 文件](#)。

Query

Query 請求映射文件可讓您告知 AWS AppSync DynamoDB 解析程式向 DynamoDB 提出Query請求，並可讓您指定下列項目：

- 索引鍵表達式
- 要使用哪些索引
- 任何額外篩選條件
- 要傳回多少項目
- 是否使用一致性讀取
- 查詢方向 (向前或向後)
- 分頁字符

Query 映射文件結構如下：

```
{
  "version" : "2017-02-28",
  "operation" : "Query",
  "query" : {
    "expression" : "some expression",
    "expressionNames" : {
      "#foo" : "foo"
    },
    "expressionValues" : {
      ":bar" : ... typed value
    }
  },
  "index" : "fooIndex",
  "nextToken" : "a pagination token",
  "limit" : 10,
  "scanIndexForward" : true,
  "consistentRead" : false,
  "select" : "ALL_ATTRIBUTES" | "ALL_PROJECTED_ATTRIBUTES" | "SPECIFIC_ATTRIBUTES",
  "filter" : {
    ...
  },
  "projection" : {
    ...
  }
}
```

```
}
```

欄位定義如下：

查詢欄位

查詢欄位清單

version

範本定義版本。目前支援 2017-02-28 和 2018-05-29。此值為必填。

operation

要執行的 DynamoDB 操作。若要執行 Query DynamoDB 操作，這必須設為 Query。此值為必填。

query

query 區段可讓您指定金鑰條件表達式，描述要從 DynamoDB 擷取哪些項目。如需如何編寫索引鍵條件表達式的詳細資訊，請參閱 [DynamoDB KeyConditions 文件](#)。必須指定此區段。

expression

查詢表達式。必須指定此欄位。

expressionNames

表達式屬性 name 預留位置的替代，形式為鍵值組。索引鍵對應於 中使用的名稱預留位置 expression，且值必須是對應於 DynamoDB 中項目屬性名稱的字串。此欄位為選用的，應只能填入用於 expression 中表達式屬性名稱預留位置的替代。

expressionValues

表達式屬性 value 預留位置的替代，形式為鍵值組。鍵對應用於 expression 的值預留位置，值必須是類型值。如需如何指定「輸入值」的詳細資訊，請參閱[類型系統（請求映射）](#)。此值為必填。此欄位為選用的，應只能填入用於 expression 中表達式屬性值預留位置的替代。

filter

額外的篩選條件可用來在傳回 DynamoDB 的結果之前先篩選結果。如需篩選條件的詳細資訊，請參閱[篩選條件](#)。此欄位為選用欄位。

index

要查詢的索引名稱。除了雜湊索引鍵的主索引鍵索引之外，DynamoDB 查詢操作還可讓您掃描本機次要索引和全域次要索引。如果指定，這會通知 DynamoDB 查詢指定的索引。若省略，則會查詢主索引鍵索引。

nextToken

分頁字符將繼續先前的查詢。這會是從先前查詢所取得的。此欄位為選用欄位。

limit

要評估的項目數上限 (不一定是相符的項目數)。此欄位為選用欄位。

scanIndexForward

指出是否向前或向後查詢的布林值。此欄位為選用，預設值為 true。

consistentRead

布林值，指出在查詢 DynamoDB 時是否使用一致讀取。此欄位為選用，預設值為 false。

select

根據預設，AWS AppSync DynamoDB 解析程式只會傳回投影到索引中的屬性。如果需要更多屬性，請設定這個欄位。此欄位為選用欄位。支援的值是：

ALL_ATTRIBUTES

傳回所有指定資料表或索引的項目屬性。如果您查詢本機次要索引，DynamoDB 會從索引中每個相符項目的父資料表擷取整個項目。如果索引設定為投射所有項目屬性，所有資料都可從本機次要索引取得，不需進行任何擷取。

ALL_PROJECTED_ATTRIBUTES

只在查詢索引時才允許。擷取所有已投射到索引的屬性。如果索引設定為投射所有屬性，此傳回值相當於指定 ALL_ATTRIBUTES。

SPECIFIC_ATTRIBUTES

僅傳回 中列出的屬性 `projectionexpression`。此傳回值等同於指定 `projection` 的 `expression` 而不指定 的任何值 `Select`。

projection

用於指定要從 DynamoDB 操作傳回之屬性的投影。如需投影的詳細資訊，請參閱[投影](#)。此欄位為選用欄位。

DynamoDB 的結果會自動轉換為 GraphQL 和 JSON 基本類型，並可在映射內容中使用 (`$context.result`)。

如需 DynamoDB 類型轉換的詳細資訊，請參閱[類型系統（回應映射）](#)。

如需回應映射範本的詳細資訊，請參閱[解析程式映射範本概觀](#)。

結果的結構如下：

```
{
  items = [ ... ],
  nextToken = "a pagination token",
  scannedCount = 10
}
```

欄位定義如下：

items

包含 DynamoDB 查詢傳回項目的清單。

nextToken

如果可能有一個以上的結果，`nextToken` 會包含可用於另一個要求的分頁字符。請注意，AWS AppSync 會加密和混淆從 DynamoDB 傳回的分頁字符。這樣可確保資料表的資料不會不慎洩漏給發起人。另請注意，這些分頁字符在不同解析程式之間無法使用。

scannedCount

套用篩選條件表達式 (若有的話) 以前符合查詢條件表達式的項目數。

範例

下列範例是 GraphQL 查詢 的映射範本 `getPosts(owner: ID!)`。

在此範例中，資料表上的全域次要索引受到查詢，以傳回指定 ID 擁有的所有文章。

```
{
  "version" : "2017-02-28",
  "operation" : "Query",
  "query" : {
    "expression" : "ownerId = :ownerId",
```



```
      "expressionValues" : {
        ":ownerId" : $util.dynamodb.toDynamoDBJson($context.arguments.owner)
      }
    },
    "index" : "owner-index"
  }
}
```

如需 DynamoDB Query API 的詳細資訊，請參閱 [DynamoDB API 文件](#)。

Scan

Scan 請求映射文件可讓您告知 AWS AppSync DynamoDB 解析程式向 DynamoDB 提出 Scan 請求，並可讓您指定下列項目：

- 排除結果的篩選結果
- 要使用哪些索引
- 要傳回多少項目
- 是否使用一致性讀取
- 分頁字符
- 平行掃描

Scan 映射文件結構如下：

```
{
  "version" : "2017-02-28",
  "operation" : "Scan",
  "index" : "fooIndex",
  "limit" : 10,
  "consistentRead" : false,
  "nextToken" : "aPaginationToken",
  "totalSegments" : 10,
  "segment" : 1,
  "filter" : {
    ...
  },
  "projection" : {
    ...
  }
}
```

欄位定義如下：

掃描欄位

掃描欄位清單

version

範本定義版本。目前支援 2017-02-28 和 2018-05-29。此值為必填。

operation

要執行的 DynamoDB 操作。若要執行 Scan DynamoDB 操作，這必須設為 Scan。此值為必填。

filter

在傳回 DynamoDB 的結果之前，可用來篩選結果的篩選條件。如需篩選條件的詳細資訊，請參閱[篩選條件](#)。此欄位為選用欄位。

index

要查詢的索引名稱。除了雜湊索引鍵的主索引鍵索引之外，DynamoDB 查詢操作還可讓您掃描本機次要索引和全域次要索引。如果指定，這會通知 DynamoDB 查詢指定的索引。若省略，則會查詢主索引鍵索引。

limit

單次可評估的項目數量上限。此欄位為選用欄位。

consistentRead

布林值，指出在查詢 DynamoDB 時是否使用一致的讀取。此欄位為選用，預設值為 false。

nextToken

分頁字符將繼續先前的查詢。這會是從先前查詢所取得的。此欄位為選用欄位。

select

根據預設，AWS AppSync DynamoDB 解析程式只會傳回投影到索引中的任何屬性。如果需要更多屬性，則此欄位可以設定。此欄位為選用欄位。支援的值是：

ALL_ATTRIBUTES

傳回所有指定資料表或索引的項目屬性。如果您查詢本機次要索引，DynamoDB 會從索引中每個相符項目的父資料表擷取整個項目。如果索引設定為投射所有項目屬性，所有資料都可從本機次要索引取得，不需進行任何擷取。

ALL_PROJECTED_ATTRIBUTES

只在查詢索引時才允許。擷取所有已投射到索引的屬性。如果索引設定為投射所有屬性，此傳回值相當於指定 ALL_ATTRIBUTES。

SPECIFIC_ATTRIBUTES

僅傳回 中列出的屬性 `projectionexpression`。此傳回值等同於指定 `projection` 的 `expression` 而不指定 的任何值 `Select`。

totalSegments

執行平行掃描時分割資料表的區段數。此欄位為選用的，但若指定 `segment`，則此欄位必須指定。

segment

此操作中執行平行掃描時的資料表區段。此欄位為選用的，但若指定 `totalSegments`，則此欄位必須指定。

projection

用於指定要從 DynamoDB 操作傳回之屬性的投影。如需投影的詳細資訊，請參閱[投影](#)。此欄位為選用欄位。

DynamoDB 掃描傳回的結果會自動轉換為 GraphQL 和 JSON 基本類型，並可在映射內容中使用 (`$context.result`)。

如需 DynamoDB 類型轉換的詳細資訊，請參閱[類型系統（回應映射）](#)。

如需回應映射範本的詳細資訊，請參閱[解析程式映射範本概觀](#)。

結果的結構如下：

```
{
  items = [ ... ],
  nextToken = "a pagination token",
  scannedCount = 10
}
```

欄位定義如下：

items

包含 DynamoDB 掃描傳回項目的清單。

nextToken

如果結果可能更多，會nextToken包含分頁字符，您可以在另一個請求中使用。AWS AppSync 會加密和混淆從 DynamoDB 傳回的分頁字符。這樣可確保資料表的資料不會不慎洩漏給發起人。此外，這些分頁字符在不同解析程式之間無法使用。

scannedCount

在套用篩選條件表達式（如果有）之前，DynamoDB 擷取的項目數。

範例 1

下列範例是 GraphQL 查詢的映射範本：allPosts。

在此範例中，資料表中的所有項目都會傳回。

```
{
  "version" : "2017-02-28",
  "operation" : "Scan"
}
```

範例 2

下列範例是 GraphQL 查詢的映射範本：postsMatching(title: String!)。

在此範例中，資料表中開頭為 title 引數的所有項目都會傳回。

```
{
  "version" : "2017-02-28",
  "operation" : "Scan",
  "filter" : {
    "expression" : "begins_with(title, :title)",
    "expressionValues" : {
      ":title" : $util.dynamodb.toDynamoDBJson($context.arguments.title)
    },
  }
}
```

如需 DynamoDB Scan API 的詳細資訊，請參閱 [DynamoDB API 文件](#)。

Sync

Sync 請求映射文件可讓您從 DynamoDB 資料表擷取所有結果，然後只接收自上次查詢（差異更新）後更改的資料。Sync 請求只能對版本控制的 DynamoDB 資料來源提出。您可以指定下列選項：

- 排除結果的篩選結果
- 要傳回多少項目
- 分頁字符
- 上次起始 Sync 操作時

Sync 映射文件結構如下：

```
{
  "version" : "2018-05-29",
  "operation" : "Sync",
  "basePartitionKey": "Base Tables PartitionKey",
  "deltaIndexName": "delta-index-name",
  "limit" : 10,
  "nextToken" : "aPaginationToken",
  "lastSync" : 1550000000000,
  "filter" : {
    ...
  }
}
```

欄位定義如下：

同步欄位

同步欄位清單

version

範本定義的版本。目前僅支援 2018-05-29。此值為必填。

operation

要執行的 DynamoDB 操作。若要執行 Sync 操作，這必須設定為 Sync。此值為必填。

filter

在傳回 DynamoDB 的結果之前，可用來篩選結果的篩選條件。如需篩選條件的詳細資訊，請參閱[篩選條件](#)。此欄位為選用欄位。

limit

單次可評估的項目數量上限。此欄位為選用欄位。如果省略此值，預設限制將設為 100 個項目。此欄位的最大值為 1000 個項目。

nextToken

分頁字符將繼續先前的查詢。這會是從先前查詢所取得的。此欄位為選用欄位。

lastSync

上次成功啟動 Sync 操作的時間 (以毫秒為單位)。如果指定此值，只會傳回 lastSync 之後變更的項目。這個欄位是選用的，而且只有在初始 Sync 操作擷取所有頁面之後才能填入。如果省略此值，將傳回 Base 資料表的結果，否則會傳回 Delta 資料表的結果。

basePartitionKey

執行 Sync 操作時使用之 Base 資料表的分割區索引鍵。此欄位允許在資料表使用自訂分割區索引鍵時執行 Sync 操作。此為選用欄位。

deltaIndexName

用於 Sync 操作的索引。當資料表使用自訂分割區索引鍵時，需要此索引才能在整個差異存放區資料表上啟用 Sync 操作。Sync 操作將在 GSI 上執行 (在 gsi_ds_pk 和 上建立 gsi_ds_sk)。此欄位為選用欄位。

DynamoDB 同步傳回的結果會自動轉換為 GraphQL 和 JSON 基本類型，並可在映射內容中使用 (`$context.result`)。

如需 DynamoDB 類型轉換的詳細資訊，請參閱[類型系統 \(回應映射\)](#)。

如需回應映射範本的詳細資訊，請參閱[解析程式映射範本概觀](#)。

結果的結構如下：

```
{
  items = [ ... ],
```

```
    nextToken = "a pagination token",
    scannedCount = 10,
    startedAt = 1550000000000
  }
```

欄位定義如下：

items

包含同步傳回項目的清單。

nextToken

如果結果可能更多，會nextToken包含分頁字符，您可以在另一個請求中使用。AWS AppSync 會加密和混淆從 DynamoDB 傳回的分頁字符。這樣可確保資料表的資料不會不慎洩漏給發起人。此外，這些分頁字符在不同解析程式之間無法使用。

scannedCount

在套用篩選條件表達式（如果有）之前，DynamoDB 擷取的項目數。

startedAt

開始同步操作時，可以在本機存放並在另一個請求中做為 lastSync 引數的時間 (以 epoch 毫秒為單位)。如果請求中包含分頁字符，則該值將與請求針對第一頁結果傳回的值相同。

範例

下列範例是 GraphQL 查詢的映射範本：syncPosts(nextToken: String, lastSync: AWSTimestamp)。

在此範例中，如果省略 lastSync，則會傳回 Base 資料表中的所有項目。如果提供 lastSync，只會傳回自 lastSync 變更之 Delta Sync 資料表中的項目。

```
{
  "version" : "2018-05-29",
  "operation" : "Sync",
  "limit": 100,
  "nextToken": $util.toJson($util.defaultIfNull($ctx.args.nextToken, null)),
  "lastSync": $util.toJson($util.defaultIfNull($ctx.args.lastSync, null))
}
```

BatchGetItem

BatchGetItem 請求映射文件可讓您告知 AWS AppSync DynamoDB 解析程式向 DynamoDB 提出BatchGetItem請求，以擷取多個項目，可能橫跨多個資料表。使用此要求範本時，您必須指定下列項目：

- 將從其中擷取項目的資料表名稱
- 將從個別資料表擷取項目的索引鍵

此時會套用 DynamoDB BatchGetItem 限制，且可能不會提供任何條件表達式。

BatchGetItem 映射文件結構如下：

```
{
  "version" : "2018-05-29",
  "operation" : "BatchGetItem",
  "tables" : {
    "table1": {
      "keys": [
        ## Item to retrieve Key
        {
          "foo" : ... typed value,
          "bar" : ... typed value
        },
        ## Item2 to retrieve Key
        {
          "foo" : ... typed value,
          "bar" : ... typed value
        }
      ],
      "consistentRead": true|false,
      "projection" : {
        ...
      }
    },
    "table2": {
      "keys": [
        ## Item3 to retrieve Key
        {
          "foo" : ... typed value,
          "bar" : ... typed value
        },

```



```

        ## Item4 to retrieve Key
        {
            "foo" : ... typed value,
            "bar" : ... typed value
        }
    ],
    "consistentRead": true|false,
    "projection" : {
        ...
    }
}
}
}
}

```

欄位定義如下：

BatchGetItem 欄位

BatchGetItem 欄位清單

version

範本定義的版本。僅支援 2018-05-29。此值為必填。

operation

要執行的 DynamoDB 操作。若要執行 BatchGetItem DynamoDB 操作，這必須設為 BatchGetItem。此值為必填。

tables

要從中擷取項目的 DynamoDB 資料表。此值是其中將資料表名稱指定為該映射索引鍵的映射。至少必須提供一個資料表。tables 值為必填。

keys

代表要擷取之項目主索引鍵的 DynamoDB 索引鍵清單。DynamoDB 項目可能具有單一雜湊索引鍵，或雜湊索引鍵和排序索引鍵，視資料表結構而定。如需如何指定「輸入值」的詳細資訊，請參閱[類型系統（請求映射）](#)。

consistentRead

執行 GetItem 操作時是否使用一致性讀取。此值為選用值，且預設值為 false。

projection

用於指定要從 DynamoDB 操作傳回之屬性的投影。如需投影的詳細資訊，請參閱[投影](#)。此欄位為選用欄位。

注意事項：

- 如果未從資料表中擷取任何項目，該資料表的 data 區塊中會顯示 null 元素。
- 調用結果會根據在請求映射範本中提供的順序，依資料表排序。
- 內的每個Get命令BatchGetItem都是原子的，但批次可以部分處理。如果因錯誤而部分批次處理，則未處理的索引鍵會透過 unprocessedKeys 區塊傳回為部分的呼叫結果。
- BatchGetItem 限制為 100 個索引鍵。

使用下列範例要求映射範本時：

```
{
  "version": "2018-05-29",
  "operation": "BatchGetItem",
  "tables": {
    "authors": [
      {
        "author_id": {
          "S": "a1"
        }
      },
    ],
    "posts": [
      {
        "author_id": {
          "S": "a1"
        },
        "post_id": {
          "S": "p2"
        }
      }
    ],
  }
}
```

可透過 `$ctx.result` 提供的呼叫結果如下所示：

```
{
  "data": {
    "authors": [null],
    "posts": [
      # Was retrieved
      {
        "author_id": "a1",
        "post_id": "p2",
        "post_title": "title",
        "post_description": "description",
      }
    ]
  },
  "unprocessedKeys": {
    "authors": [
      # This item was not processed due to an error
      {
        "author_id": "a1"
      }
    ],
    "posts": []
  }
}
```

`$ctx.error` 包含錯誤的詳細資訊。在要求映射範本中所提供的索引鍵 `data`、`unprocessedKeys` 及個別資料表索引鍵，都確定會出現在呼叫結果中。已刪除的項目會出現在 `data` 區塊中。尚未處理的項目在該 `data` 區塊中會標示為 `null`，並置於 `unprocessedKeys` 區塊。

如需更完整的範例，請遵循此處 AppSync 的 DynamoDB 批次教學 [課程教學課程：DynamoDB 批次解析程式](#)。

BatchDeleteItem

`BatchDeleteItem` 請求映射文件可讓您告知 AWS AppSync DynamoDB 解析程式向 DynamoDB 提出刪除多個項目的 `BatchWriteItem` 請求，可能跨多個資料表。使用此要求範本時，您必須指定下列項目：

- 將從其中刪除項目的資料表名稱
- 將從個別資料表刪除項目的索引鍵

此時會套用 DynamoDB `BatchWriteItem` 限制，且可能不會提供任何條件表達式。

BatchDeleteItem 映射文件結構如下：

```
{
  "version" : "2018-05-29",
  "operation" : "BatchDeleteItem",
  "tables" : {
    "table1": [
      ## Item to delete Key
      {
        "foo" : ... typed value,
        "bar" : ... typed value
      },
      ## Item2 to delete Key
      {
        "foo" : ... typed value,
        "bar" : ... typed value
      }
    ],
    "table2": [
      ## Item3 to delete Key
      {
        "foo" : ... typed value,
        "bar" : ... typed value
      },
      ## Item4 to delete Key
      {
        "foo" : ... typed value,
        "bar" : ... typed value
      }
    ],
  }
}
```

欄位定義如下：

BatchDeleteItem 欄位

BatchDeleteItem 欄位清單

version

範本定義的版本。僅支援 2018-05-29。此值為必填。

operation

要執行的 DynamoDB 操作。若要執行 BatchDeleteItem DynamoDB 操作，這必須設為 BatchDeleteItem。此值為必填。

tables

要從中刪除項目的 DynamoDB 資料表。每個資料表都是 DynamoDB 金鑰清單，代表要刪除項目的主金鑰。DynamoDB 項目可能具有單一雜湊索引鍵，或雜湊索引鍵和排序索引鍵，視資料表結構而定。如需如何指定「輸入值」的詳細資訊，請參閱[類型系統（請求映射）](#)。至少必須提供一個資料表。tables 值為必要。

注意事項：

- 不同於 DeleteItem 操作，回應中未傳回完整刪除的項目。只會傳回已傳遞的索引鍵。
- 如果資料表中未刪除任何項目，該資料表的 data 區塊中就會顯示 null 元素。
- 調用結果會根據在請求映射範本中提供的順序，依資料表排序。
- 中的每個 Delete 命令 BatchDeleteItem 都是原子的。但是，批次可以部分處理。如果因錯誤而部分批次處理，則未處理的索引鍵會透過 unprocessedKeys 區塊傳回為部分的呼叫結果。
- BatchDeleteItem 限制為 25 個索引鍵。
- 與衝突偵測搭配使用時，不支援此操作。同時使用兩者可能會導致錯誤。

使用下列範例要求映射範本時：

```
{
  "version": "2018-05-29",
  "operation": "BatchDeleteItem",
  "tables": {
    "authors": [
      {
        "author_id": {
          "S": "a1"
        }
      },
    ],
    "posts": [
      {
        "author_id": {
          "S": "a1"
        }
      }
    ]
  }
}
```

```

    },
    "post_id": {
      "S": "p2"
    }
  },
],
}
}

```

可透過 `$ctx.result` 提供的呼叫結果如下所示：

```

{
  "data": {
    "authors": [null],
    "posts": [
      # Was deleted
      {
        "author_id": "a1",
        "post_id": "p2"
      }
    ]
  },
  "unprocessedKeys": {
    "authors": [
      # This key was not processed due to an error
      {
        "author_id": "a1"
      }
    ],
    "posts": []
  }
}

```

`$ctx.error` 包含錯誤的詳細資訊。在要求映射範本中所提供的索引鍵 `data`、`unprocessedKeys` 及個別資料表索引鍵，都確定會出現在呼叫結果中。已刪除的項目會出現在 `data` 區塊中。尚未處理的項目在該 `data` 區塊中會標示為 `null`，並置於 `unprocessedKeys` 區塊。

如需更完整的範例，請遵循此處 AppSync 的 DynamoDB 批次教學 [課程教學課程：DynamoDB 批次解析程式](#)。

BatchPutItem

BatchPutItem 請求映射文件可讓您告知 AWS AppSync DynamoDB 解析程式向 DynamoDB 提出BatchWriteItem請求，以放置多個項目，可能跨越多個資料表。使用此要求範本時，您必須指定下列項目：

- 項目將放入其中的資料表名稱
- 將放入每個資料表的所有項目

此時會套用 DynamoDB BatchWriteItem 限制，且可能不會提供任何條件表達式。

BatchPutItem 映射文件結構如下：

```
{
  "version" : "2018-05-29",
  "operation" : "BatchPutItem",
  "tables" : {
    "table1": [
      ## Item to put
      {
        "foo" : ... typed value,
        "bar" : ... typed value
      },
      ## Item2 to put
      {
        "foo" : ... typed value,
        "bar" : ... typed value
      }
    ],
    "table2": [
      ## Item3 to put
      {
        "foo" : ... typed value,
        "bar" : ... typed value
      },
      ## Item4 to put
      {
        "foo" : ... typed value,
        "bar" : ... typed value
      }
    ],
  }
}
```

欄位定義如下：

BatchPutItem 欄位

BatchPutItem 欄位清單

version

範本定義的版本。僅支援 2018-05-29。此值為必填。

operation

要執行的 DynamoDB 操作。若要執行 BatchPutItem DynamoDB 操作，這必須設為 BatchPutItem。此值為必填。

tables

要放入項目的 DynamoDB 資料表。每個資料表項目代表要為此特定資料表插入的 DynamoDB 項目清單。至少必須提供一個資料表。此值為必填。

注意事項：

- 若插入成功，回應會傳回完全插入的項目。
- 如果資料表中並未插入任何項目，該資料表的 data 區塊中就會顯示 null 元素。
- 插入的項目會根據在請求映射範本內提供的順序，依資料表排序。
- 中的每個Put命令BatchPutItem都是原子的，但批次可以部分處理。如果因錯誤而部分批次處理，則未處理的索引鍵會透過 unprocessedKeys 區塊傳回為部分的呼叫結果。
- BatchPutItem 限制為 25 個項目。
- 與衝突偵測搭配使用時，不支援此操作。同時使用兩者可能會導致錯誤。

使用下列範例要求映射範本時：

```
{
  "version": "2018-05-29",
  "operation": "BatchPutItem",
  "tables": {
    "authors": [
      {
        "author_id": {
          "S": "a1"
        }
      }
    ]
  }
}
```



```
    },
    "author_name": {
      "S": "a1_name"
    }
  },
],
"posts": [
  {
    "author_id": {
      "S": "a1"
    },
    "post_id": {
      "S": "p2"
    },
    "post_title": {
      "S": "title"
    }
  }
],
}
}
```

可透過 `$ctx.result` 提供的呼叫結果如下所示：

```
{
  "data": {
    "authors": [
      null
    ],
    "posts": [
      # Was inserted
      {
        "author_id": "a1",
        "post_id": "p2",
        "post_title": "title"
      }
    ]
  },
  "unprocessedItems": {
    "authors": [
      # This item was not processed due to an error
      {
        "author_id": "a1",
```

```

        "author_name": "a1_name"
      }
    ],
    "posts": []
  }
}

```

`$ctx.error` 包含錯誤的詳細資訊。在要求映射範本中所提供的索引鍵 `data`、`unprocessedItems` 及個別資料表索引鍵，都確定會出現在呼叫結果中。已插入的項目會出現在 `data` 區塊中。尚未處理的項目在該 `data` 區塊中會標示為 `null`，並置於 `unprocessedItems` 區塊。

如需更完整的範例，請依照 DynamoDB Batch 教學課程搭配 AppSync here [Tutorial : DynamoDB batch resolver](#)。

TransactGetItems

`TransactGetItems` 請求映射文件可讓您告知 AWS AppSync DynamoDB 解析程式向 DynamoDB 提出 `TransactGetItems` 請求，以擷取多個項目，可能橫跨多個資料表。使用此要求範本時，您必須指定下列項目：

- 從中擷取項目之每個請求項目的資料表名稱
- 從每個資料表擷取之每個請求項目的索引鍵

此時會套用 DynamoDB `TransactGetItems` 限制，且可能不會提供任何條件表達式。

`TransactGetItems` 映射文件結構如下：

```

{
  "version": "2018-05-29",
  "operation": "TransactGetItems",
  "transactItems": [
    ## First request item
    {
      "table": "table1",
      "key": {
        "foo": ... typed value,
        "bar": ... typed value
      },
      "projection" : {
        ...
      }
    }
  ]
}

```

```
    },  
    ## Second request item  
    {  
      "table": "table2",  
      "key": {  
        "foo": ... typed value,  
        "bar": ... typed value  
      },  
      "projection" : {  
        ...  
      }  
    }  
  ]  
}
```

欄位定義如下：

TransactGetItems 欄位

TransactGetItems 欄位清單

version

範本定義的版本。僅支援 2018-05-29。此值為必填。

operation

要執行的 DynamoDB 操作。若要執行 TransactGetItems DynamoDB 操作，這必須設為 TransactGetItems。此值為必填。

transactItems

要包含的請求項目。此值是請求項目的陣列。必須提供至少一個請求項目。transactItems 值為必填。

table

要從中擷取項目的 DynamoDB 資料表。此值是資料表名稱的字串。table 值為必填。

key

代表要擷取之項目主索引鍵的 DynamoDB 索引鍵。DynamoDB 項目可能具有單一雜湊索引鍵，或雜湊索引鍵和排序索引鍵，視資料表結構而定。如需如何指定「輸入值」的詳細資訊，請參閱[類型系統（請求映射）](#)。

projection

用於指定要從 DynamoDB 操作傳回之屬性的投影。如需投影的詳細資訊，請參閱[投影](#)。此欄位為選用欄位。

注意事項：

- 如果交易成功，items 區塊中擷取項目的順序會與請求項目的順序相同。
- 以全有或全無的方式執行交易。如果任何請求項目造成錯誤，將不執行整個交易，而且將傳回錯誤詳細資料。
- 無法擷取的請求項目不是錯誤。相反地，null 元素會出現在對應位置的項目區塊。
- 如果交易錯誤是 TransactionCanceledException，則會填入 cancellationReasons 區塊。cancellationReasons 區塊中取消原因的順序將與請求項目的順序相同。
- TransactGetItems 限制為 100 個請求項目。

使用下列範例要求映射範本時：

```
{
  "version": "2018-05-29",
  "operation": "TransactGetItems",
  "transactItems": [
    ## First request item
    {
      "table": "posts",
      "key": {
        "post_id": {
          "S": "p1"
        }
      }
    },
    ## Second request item
    {
      "table": "authors",
      "key": {
        "author_id": {
          "S": "a1"
        }
      }
    }
  ]
}
```

```
}
```

如果交易成功，而且僅擷取第一個請求項目，則 `$ctx.result` 中的可用叫用結果如下所示：

```
{
  "items": [
    {
      // Attributes of the first requested item
      "post_id": "p1",
      "post_title": "title",
      "post_description": "description"
    },
    // Could not retrieve the second requested item
    null,
  ],
  "cancellationReasons": null
}
```

如果因為第一個請求項目造成的 `TransactionCanceledException` 造成交易失敗，則 `$ctx.result` 中的可用叫用結果如下所示：

```
{
  "items": null,
  "cancellationReasons": [
    {
      "type": "Sample error type",
      "message": "Sample error message"
    },
    {
      "type": "None",
      "message": "None"
    }
  ]
}
```

`$ctx.error` 包含錯誤的詳細資訊。索引鍵 `items` 和 `cancellationReasons` 保證出現在 `$ctx.result` 中。

如需更完整的範例，請依照 DynamoDB 交易教學搭配 AppSync 此處[教學課程：DynamoDB 交易解析程式](#)。

TransactWriteItems

TransactWriteItems 請求映射文件可讓您告知 AWS AppSync DynamoDB 解析程式向 DynamoDB 提出TransactWriteItems請求，以寫入多個項目，可能寫入多個資料表。使用此要求範本時，您必須指定下列項目：

- 每個請求項目的目標資料表名稱
- 每個請求項目要執行的操作。有四種支援的操作類型：PutItem、UpdateItem、DeleteItem 以及 ConditionCheck
- 每個要寫入之請求項目的索引鍵

套用 DynamoDB TransactWriteItems 限制。

TransactWriteItems 映射文件結構如下：

```
{
  "version": "2018-05-29",
  "operation": "TransactWriteItems",
  "transactItems": [
    {
      "table": "table1",
      "operation": "PutItem",
      "key": {
        "foo": ... typed value,
        "bar": ... typed value
      },
      "attributeValues": {
        "baz": ... typed value
      },
      "condition": {
        "expression": "someExpression",
        "expressionNames": {
          "#foo": "foo"
        },
        "expressionValues": {
          ":bar": ... typed value
        },
        "returnValuesOnConditionCheckFailure": true|false
      }
    },
    {
```

```

    "table": "table2",
    "operation": "UpdateItem",
    "key": {
      "foo": ... typed value,
      "bar": ... typed value
    },
    "update": {
      "expression": "someExpression",
      "expressionNames": {
        "#foo": "foo"
      },
      "expressionValues": {
        ":bar": ... typed value
      }
    },
    "condition": {
      "expression": "someExpression",
      "expressionNames": {
        "#foo": "foo"
      },
      "expressionValues": {
        ":bar": ... typed value
      },
      "returnValuesOnConditionCheckFailure": true|false
    }
  },
  {
    "table": "table3",
    "operation": "DeleteItem",
    "key": {
      "foo": ... typed value,
      "bar": ... typed value
    },
    "condition": {
      "expression": "someExpression",
      "expressionNames": {
        "#foo": "foo"
      },
      "expressionValues": {
        ":bar": ... typed value
      },
      "returnValuesOnConditionCheckFailure": true|false
    }
  }
},

```

```

    {
      "table": "table4",
      "operation": "ConditionCheck",
      "key": {
        "foo": ... typed value,
        "bar": ... typed value
      },
      "condition": {
        "expression": "someExpression",
        "expressionNames": {
          "#foo": "foo"
        },
        "expressionValues": {
          ":bar": ... typed value
        },
        "returnValuesOnConditionCheckFailure": true|false
      }
    }
  ]
}

```

TransactWriteItems 欄位

TransactWriteItems 欄位清單

欄位定義如下：

version

範本定義的版本。僅支援 2018-05-29。此值為必填。

operation

要執行的 DynamoDB 操作。若要執行 TransactWriteItems DynamoDB 操作，這必須設為 TransactWriteItems。此值為必填。

transactItems

要包含的請求項目。此值是請求項目的陣列。必須提供至少一個請求項目。transactItems 值為必填。

對於 PutItem，欄位定義如下：

table

目的地 DynamoDB 資料表。此值是資料表名稱的字串。table 值為必填。

operation

要執行的 DynamoDB 操作。若要執行 PutItem DynamoDB 操作，這必須設為 PutItem。此值為必填。

key

代表要放置之項目主索引鍵的 DynamoDB 索引鍵。DynamoDB 項目可能具有單一雜湊索引鍵，或雜湊索引鍵和排序索引鍵，視資料表結構而定。如需如何指定「輸入值」的詳細資訊，請參閱[類型系統（請求映射）](#)。此值為必填。

attributeValues

將放入 DynamoDB 的項目其餘屬性。如需如何指定「輸入值」的詳細資訊，請參閱[類型系統（請求映射）](#)。此欄位為選用欄位。

condition

決定要求是否成功的條件，可根據已存在於 DynamoDB 的物件狀態。如果沒有指定條件，PutItem 要求會覆寫該項目的任何現有資料項目。您可以指定在條件檢查失敗時是否擷取現有項目。如需交易條件的詳細資訊，請參閱[交易條件表達式](#)。此值是選用的。

對於 UpdateItem，欄位定義如下：

table

要更新的 DynamoDB 資料表。此值是資料表名稱的字串。table 值為必填。

operation

要執行的 DynamoDB 操作。若要執行 UpdateItem DynamoDB 操作，這必須設為 UpdateItem。此值為必填。

key

代表要更新之項目主索引鍵的 DynamoDB 索引鍵。DynamoDB 項目可能具有單一雜湊索引鍵，或雜湊索引鍵和排序索引鍵，視資料表結構而定。如需如何指定「輸入值」的詳細資訊，請參閱[類型系統（請求映射）](#)。此值為必填。

update

update 本節可讓您指定更新表達式，描述如何在 DynamoDB 中更新項目。如需有關如何撰寫更新表達式的詳細資訊，請參閱 [DynamoDB UpdateExpressions 文件](#)。此區段是必須的。

condition

決定要求是否成功的條件，可根據已存在於 DynamoDB 的物件狀態。如果沒有指定條件，UpdateItem 要求會更新現有的資料項目，無論項目的目前狀態為何。您可以指定在條件檢查失敗時是否擷取現有項目。如需交易條件的詳細資訊，請參閱[交易條件表達式](#)。此值是選用的。

對於 DeleteItem，欄位定義如下：

table

要在其中刪除項目的 DynamoDB 資料表。此值是資料表名稱的字串。table 值為必填。

operation

要執行的 DynamoDB 操作。若要執行 DeleteItem DynamoDB 操作，這必須設為 DeleteItem。此值為必填。

key

代表要刪除之項目主索引鍵的 DynamoDB 索引鍵。DynamoDB 項目可能具有單一雜湊索引鍵，或雜湊索引鍵和排序索引鍵，視資料表結構而定。如需如何指定「輸入值」的詳細資訊，請參閱[類型系統（請求映射）](#)。此值為必填。

condition

決定要求是否成功的條件，可根據已存在於 DynamoDB 的物件狀態。如果沒有指定條件，DeleteItem 要求會刪除項目，無論該項目的目前狀態為何。您可以指定在條件檢查失敗時是否擷取現有項目。如需交易條件的詳細資訊，請參閱[交易條件表達式](#)。此值是選用的。

對於 ConditionCheck，欄位定義如下：

table

要檢查條件的 DynamoDB 資料表。此值是資料表名稱的字串。table 值為必填。

operation

要執行的 DynamoDB 操作。若要執行 ConditionCheck DynamoDB 操作，這必須設為 ConditionCheck。此值為必填。

key

代表要條件檢查之項目主索引鍵的 DynamoDB 索引鍵。DynamoDB 項目可能具有單一雜湊索引鍵，或雜湊索引鍵和排序索引鍵，視資料表結構而定。如需如何指定「輸入值」的詳細資訊，請參閱[類型系統（請求映射）](#)。此值為必填。

condition

決定要求是否成功的條件，可根據已存在於 DynamoDB 的物件狀態。您可以指定在條件檢查失敗時是否擷取現有項目。如需交易條件的詳細資訊，請參閱[交易條件表達式](#)。此值為必填。

注意事項：

- 如果成功，只會在回應中傳回請求項目的索引鍵。索引鍵的順序將與請求項目的順序相同。
- 以全有或全無的方式執行交易。如果任何請求項目造成錯誤，將不執行整個交易，而且將傳回錯誤詳細資料。
- 沒有兩個請求項目可以定位到相同項目。否則，它們將造成 `TransactionCanceledException` 錯誤。
- 如果交易錯誤是 `TransactionCanceledException`，則會填入 `cancellationReasons` 區塊。如果請求項目的條件檢查失敗，而且您未將 `returnValuesOnConditionCheckFailure` 指定為 `false`，將擷取資料表中存在的項目，並存放在 `cancellationReasons` 區塊之對應位置的 `item` 中。
- `TransactWriteItems` 限制為 100 個請求項目。
- 與衝突偵測搭配使用時，不支援此操作。同時使用兩者可能會導致錯誤。

使用下列範例要求映射範本時：

```
{
  "version": "2018-05-29",
  "operation": "TransactWriteItems",
  "transactItems": [
    {
      "table": "posts",
      "operation": "PutItem",
      "key": {
        "post_id": {
          "S": "p1"
        }
      },
      "attributeValues": {
        "post_title": {
          "S": "New title"
        },
        "post_description": {
          "S": "New description"
        }
      }
    }
  ]
}
```

```

    }
  },
  "condition": {
    "expression": "post_title = :post_title",
    "expressionValues": {
      ":post_title": {
        "S": "Expected old title"
      }
    }
  }
},
{
  "table": "authors",
  "operation": "UpdateItem",
  "key": {
    "author_id": {
      "S": "a1"
    },
  },
  "update": {
    "expression": "SET author_name = :author_name",
    "expressionValues": {
      ":author_name": {
        "S": "New name"
      }
    }
  },
},
]
}

```

如果交易成功，`$ctx.result` 中可用的叫用結果如下所示：

```

{
  "keys": [
    // Key of the PutItem request
    {
      "post_id": "p1",
    },
    // Key of the UpdateItem request
    {
      "author_id": "a1"
    }
  ]
}

```

```
],
  "cancellationReasons": null
}
```

如果交易因為PutItem請求的條件檢查失敗而失敗，中可用的調用結果\$ctx.result如下：

```
{
  "keys": null,
  "cancellationReasons": [
    {
      "item": {
        "post_id": "p1",
        "post_title": "Actual old title",
        "post_description": "Old description"
      },
      "type": "ConditionCheckFailed",
      "message": "The condition check failed."
    },
    {
      "type": "None",
      "message": "None"
    }
  ]
}
```

\$ctx.error 包含錯誤的詳細資訊。索引鍵 keys 和 cancellationReasons 保證出現在 \$ctx.result 中。

如需更完整的範例，請依照 DynamoDB 交易教學搭配 AppSync here [Tutorial：DynamoDB 交易解析程式](#)。

類型系統（請求映射）

使用 AWS AppSync DynamoDB 解析程式呼叫 DynamoDB 資料表時，AWS AppSync 需要知道要用於該呼叫的每個值類型。這是因為 DynamoDB 支援比 GraphQL 或 JSON 更多的類型基本概念（例如集合和二進位資料）。AWS AppSync 需要一些提示，否則您必須對資料表中的資料結構進行一些假設。GraphQL DynamoDB

如需 DynamoDB 資料類型的詳細資訊，請參閱 DynamoDB [資料類型描述項](#)和[資料類型](#)文件。

DynamoDB 值由包含單一鍵值對的 JSON 物件表示。金鑰指定 DynamoDB 類型，而值指定值本身。在下列範例中，S 鍵代表值是字串，identifier 值則是字串值本身。

```
{ "S" : "identifier" }
```

請注意，JSON 物件不能有一個以上的索引值對。如果指定一個以上的索引值對，要求映射文件就不會進行剖析。

DynamoDB 值會在您需要指定值的請求映射文件中的任何位置使用。您需要執行這項操作的一些地方包括：key 和 attributeValue 區段，以及表達式區段的 expressionValues 區段。在下列範例中，DynamoDB 字串值 identifier 會指派給 key 區段中的 id 欄位（可能在 GetItem 請求映射文件中）。

```
"key" : {  
  "id" : { "S" : "identifier" }  
}
```

支援的類型

AWS AppSync 支援下列 DynamoDB 純量、文件和集類型：

字串類型 S

單一字串值。DynamoDB 字串值的表示方式為：

```
{ "S" : "some string" }
```

使用範例：

```
"key" : {  
  "id" : { "S" : "some string" }  
}
```

字串集類型 SS

字串值的集合。DynamoDB 字串集值的表示方式為：

```
{ "SS" : [ "first value", "second value", ... ] }
```

使用範例：

```
"attributeValues" : {  
  "phoneNumbers" : { "SS" : [ "+1 555 123 4567", "+1 555 234 5678" ] }  
}
```

```
}
```

數字類型 **N**

單一數值。DynamoDB 數字值的表示方式為：

```
{ "N" : 1234 }
```

使用範例：

```
"expressionValues" : {  
  ":expectedVersion" : { "N" : 1 }  
}
```

數字集類型 **NS**

數值的集合。DynamoDB 數字集值以下列表示：

```
{ "NS" : [ 1, 2.3, 4 ... ] }
```

使用範例：

```
"attributeValues" : {  
  "sensorReadings" : { "NS" : [ 67.8, 12.2, 70 ] }  
}
```

二進位類型 **B**

二進位值。DynamoDB 二進位值的表示方式為：

```
{ "B" : "SGVsbG8sIFdvcmxkIQo=" }
```

請注意，該值實際上是字串，其中字串是二進位資料的 base64 編碼表示法。AWS AppSync 會將此字串解碼回其二進位值，再將其傳送至 DynamoDB。AWS AppSync 使用 RFC 2045 定義的 base64 解碼機制：忽略不在 base64 字母中的任何字元。

使用範例：

```
"attributeValues" : {  
  "binaryMessage" : { "B" : "SGVsbG8sIFdvcmxkIQo=" }  
}
```

二進位集類型 **BS**

二進位值的集合。DynamoDB 二進位集值以下列表示：

```
{ "BS" : [ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ... ] }
```

請注意，該值實際上是字串，其中字串是二進位資料的 base64 編碼表示法。AWS AppSync 會將此字串解碼回其二進位值，再將其傳送至 DynamoDB。AWS AppSync 使用 RFC 2045 定義的 base64 解碼機制：忽略不在 base64 字母中的任何字元。

使用範例：

```
"attributeValues" : {  
  "binaryMessages" : { "BS" : [ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ] }  
}
```

布林類型 **BOOL**

布林值。DynamoDB 布林值的表示方式為：

```
{ "BOOL" : true }
```

請注意，只有 true 和 false 為有效值。

使用範例：

```
"attributeValues" : {  
  "orderComplete" : { "BOOL" : false }  
}
```

清單類型 **L**

任何其他支援的 DynamoDB 值清單。DynamoDB 清單值以下列表示：

```
{ "L" : [ ... ] }
```

請注意，該值是複合值，其中清單可包含任何支援的 DynamoDB 值（包括其他清單）的零或多個。清單也可包含混合的不同類型。

使用範例：


```
{ "L" : [  
  { "S" : "A string value" },  
  { "N" : 1 },  
  { "SS" : [ "Another string value", "Even more string values!" ] }  
]
```

映射類型 **M**

代表其他支援 DynamoDB 值的金鑰值對的未排序集合。DynamoDB Map 值的表示方式為：

```
{ "M" : { ... } }
```

請注意，映射可包含零或多個索引值對。金鑰必須是字串，且值可以是任何支援的 DynamoDB 值（包括其他映射）。映射也可包含混合的不同類型。

使用範例：

```
{ "M" : {  
  "someString" : { "S" : "A string value" },  
  "someNumber" : { "N" : 1 },  
  "stringSet" : { "SS" : [ "Another string value", "Even more string  
values!" ] }  
}
```

Null 類型 **NULL**

null 值。DynamoDB Null 值的表示方式為：

```
{ "NULL" : null }
```

使用範例：

```
"attributeValues" : {  
  "phoneNumbers" : { "NULL" : null }  
}
```

如需每種類型的詳細資訊，請參閱 [DynamoDB 文件](#)。

類型系統（回應映射）

從 DynamoDB 接收回應時，AWS AppSync 會自動將其轉換為 GraphQL 和 JSON 基本類型。DynamoDB 中的每個屬性都會解碼，並在回應映射內容中傳回。

例如，如果 DynamoDB 傳回下列項目：

```
{
  "id" : { "S" : "1234" },
  "name" : { "S" : "Nadia" },
  "age" : { "N" : 25 }
}
```

然後，AWS AppSync DynamoDB 解析程式會將其轉換為 GraphQL 和 JSON 類型，如下所示：

```
{
  "id" : "1234",
  "name" : "Nadia",
  "age" : 25
}
```

本節說明 AWS AppSync 如何轉換下列 DynamoDB 純量、文件和設定類型：

字串類型 **S**

單一字串值。DynamoDB 字串值會以字串傳回。

例如，如果 DynamoDB 傳回下列 DynamoDB 字串值：

```
{ "S" : "some string" }
```

AWS AppSync 將其轉換為字串：

```
"some string"
```

字串集類型 **SS**

字串值的集合。DynamoDB 字串集值會傳回為字串清單。

例如，如果 DynamoDB 傳回下列 DynamoDB 字串集值：

```
{ "SS" : [ "first value", "second value", ... ] }
```

AWS AppSync 將其轉換為字串清單：

```
[ "+1 555 123 4567", "+1 555 234 5678" ]
```

數字類型 **N**

單一數值。DynamoDB 數字值會以數字傳回。

例如，如果 DynamoDB 傳回下列 DynamoDB 數值：

```
{ "N" : 1234 }
```

AWS AppSync 將其轉換為數字：

```
1234
```

數字集類型 **NS**

數值的集合。DynamoDB 數字集值會以數字清單傳回。

例如，如果 DynamoDB 傳回下列 DynamoDB 數字集值：

```
{ "NS" : [ 67.8, 12.2, 70 ] }
```

AWS AppSync 將其轉換為數字清單：

```
[ 67.8, 12.2, 70 ]
```

二進位類型 **B**

二進位值。DynamoDB 二進位值會以字串的形式傳回，其中包含該值的 base64 表示法。

例如，如果 DynamoDB 傳回下列 DynamoDB 二進位值：

```
{ "B" : "SGVsbG8sIFdvcmxkIQo=" }
```

AWS AppSync 將其轉換為包含 base64 表示值的字串：

```
"SGVsbG8sIFdvcmxkIQo="
```

請注意，二進位資料以 base64 編碼配置編碼，如 [RFC 4648](#) 和 [RFC 2045](#) 中所定義。

二進位集類型 **BS**

二進位值的集合。DynamoDB 二進位集值會以包含 base64 值表示的字串清單傳回。

例如，如果 DynamoDB 傳回下列 DynamoDB 二進位集值：

```
{ "BS" : [ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ... ] }
```

AWS AppSync 將其轉換為包含 base64 值表示法的字串清單：

```
[ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ... ]
```

請注意，二進位資料以 base64 編碼配置編碼，如 [RFC 4648](#) 和 [RFC 2045](#) 中所定義。

布林類型 **BOOL**

布林值。DynamoDB 布林值會以布林值傳回。

例如，如果 DynamoDB 傳回下列 DynamoDB 布林值：

```
{ "BOOL" : true }
```

AWS AppSync 將其轉換為布林值：

```
true
```

清單類型 **L**

任何其他支援的 DynamoDB 值清單。DynamoDB 清單值會傳回為值清單，其中每個內部值也會轉換。

例如，如果 DynamoDB 傳回下列 DynamoDB List 值：

```
{ "L" : [
  { "S" : "A string value" },
  { "N" : 1 },
  { "SS" : [ "Another string value", "Even more string values!" ] }
]
```

```
}
```

AWS AppSync 將其轉換為轉換值的清單：

```
[ "A string value", 1, [ "Another string value", "Even more string values!" ] ]
```

映射類型 **M**

任何其他支援的 DynamoDB 值的金鑰/值集合。DynamoDB Map 值會以 JSON 物件傳回，其中每個索引鍵/值也會轉換。

例如，如果 DynamoDB 傳回下列 DynamoDB Map 值：

```
{ "M" : {  
    "someString" : { "S" : "A string value" },  
    "someNumber" : { "N" : 1 },  
    "stringSet" : { "SS" : [ "Another string value", "Even more string  
values!" ] }  
  }  
}
```

AWS AppSync 將其轉換為 JSON 物件：

```
{  
  "someString" : "A string value",  
  "someNumber" : 1,  
  "stringSet" : [ "Another string value", "Even more string values!" ]  
}
```

Null 類型 **NULL**

null 值。

例如，如果 DynamoDB 傳回下列 DynamoDB Null 值：

```
{ "NULL" : null }
```

AWS AppSync 將其轉換為 null：

```
null
```

篩選條件

使用 Query 和 Scan 操作查詢 DynamoDB 中的物件時，您可以選擇指定 filter，以評估結果並僅傳回所需的值。

Query 或 Scan 映射文件的篩選條件映射區段之結構如下：

```
"filter" : {
  "expression" : "filter expression"
  "expressionNames" : {
    "#name" : "name",
  },
  "expressionValues" : {
    ":value" : ... typed value
  },
}
```

欄位定義如下：

expression

查詢表達式。如需如何編寫篩選條件表達式的詳細資訊，請參閱 [DynamoDB QueryFilter](#) 和 [DynamoDB ScanFilter](#) 文件。必須指定此欄位。

expressionNames

表達式屬性 name 預留位置的替代，形式為鍵值組。索引鍵對應至用於 expression 的名稱預留位置。值必須是對應於 DynamoDB 中項目屬性名稱的字串。此欄位為選用的，應只能填入用於 expression 中表達式屬性名稱預留位置的替代。

expressionValues

表達式屬性 value 預留位置的替代，形式為鍵值組。鍵對應用於 expression 的值預留位置，值必須是類型值。如需如何指定「類型值」的詳細資訊，請參閱 [類型系統 \(請求映射\)](#)。此必須指定。此欄位為選用的，應只能填入用於 expression 中表達式屬性值預留位置的替代。

範例

下列範例是映射範本的篩選條件區段，只有在標題以 title 引數開頭時，才會傳回從 DynamoDB 擷取的項目。

```
"filter" : {
```

```
"expression" : "begins_with(#title, :title)",
"expressionNames" : {
  "#title" : "title"
},
"expressionValues" : {
  ":title" : $util.dynamodb.toDynamoDBJson($context.arguments.title)
}
}
```

條件表達式

當您使用 `PutItem`、`UpdateItem` 和 `DynamoDB` 操作來變更 `DeleteItem` `DynamoDB` 中的物件時，您可以選擇性地指定條件表達式，根據在執行操作之前已在 `DynamoDB` 中的物件狀態來控制請求是否成功。

The AWS AppSync `DynamoDB` 解析程式允許在 `PutItem`、`UpdateItem` 和 `DeleteItem` 請求映射文件中指定條件表達式，以及如果條件失敗且未更新物件時要遵循的策略。

範例 1

以下 `PutItem` 映射文件沒有條件表達式。因此，即使具有相同索引鍵的項目已存在，也會將項目放入 `DynamoDB`，藉此覆寫現有項目。

```
{
  "version" : "2017-02-28",
  "operation" : "PutItem",
  "key" : {
    "id" : { "S" : "1" }
  }
}
```

範例 2

下列 `PutItem` 映射文件確實具有條件表達式，只有在 `DynamoDB` 中不存在具有相同索引鍵的項目時，才允許操作成功。

```
{
  "version" : "2017-02-28",
  "operation" : "PutItem",
  "key" : {
    "id" : { "S" : "1" }
  }
}
```

```
  },
  "condition" : {
    "expression" : "attribute_not_exists(id)"
  }
}
```

根據預設，如果條件檢查失敗，AWS AppSync DynamoDB 解析程式會傳回變動的錯誤。不過，AWS AppSync DynamoDB 解析程式提供一些額外的功能，以協助開發人員處理一些常見的邊緣案例：

- If AWS AppSync DynamoDB 解析程式可以判斷 DynamoDB 中的目前值是否符合所需的結果，無論如何都會將操作視為成功。
- 您可以設定解析程式來叫用自訂 Lambda 函數，以決定 AWS AppSync DynamoDB 解析程式應如何處理失敗，而不是傳回錯誤。

[處理條件檢查失敗](#)一節將更詳細地說明這些內容。

如需 DynamoDB 條件表達式的詳細資訊，請參閱 [DynamoDB ConditionExpressions 文件](#)。

指定條件

PutItem、UpdateItem 和 DeleteItem 要求映射文件都允許可指定選用的 condition 區段。若省略，則不會有條件檢查。若指定，條件必須為 true，操作才會成功。

condition 區段的結構如下：

```
"condition" : {
  "expression" : "someExpression"
  "expressionNames" : {
    "#foo" : "foo"
  },
  "expressionValues" : {
    ":bar" : ... typed value
  },
  "equalsIgnore" : [ "version" ],
  "consistentRead" : true,
  "conditionalCheckFailedHandler" : {
    "strategy" : "Custom",
    "lambdaArn" : "arn:..."
  }
}
```


下列欄位指定條件：

expression

更新表達式本身。如需如何編寫條件表達式的詳細資訊，請參閱 [DynamoDB ConditionExpressions 文件](#)。必須指定此欄位。

expressionNames

表達式屬性名稱預留位置的替代，形式為索引鍵值對。索引鍵對應至表達式中使用的名稱預留位置，且值必須是對應於 DynamoDB 中項目屬性名稱的字串。此欄位為選用的，應只能填入於表達式中所用表達式屬性名稱預留位置的替代。

expressionValues

表達式屬性值預留位置的替代，形式為索引值對。鍵對應用於表達式的值預留位置，值必須是類型值。如需如何指定「類型值」的詳細資訊，請參閱[類型系統 \(請求映射\)](#)。此必須指定。此欄位為選用的，應只能填入用於表達式中表達式屬性值預留位置的替代。

其餘欄位會告知 AWS AppSync DynamoDB 解析程式如何處理條件檢查失敗：

equalsIgnore

當條件檢查在使用 PutItem 操作時失敗時，AWS AppSync DynamoDB 解析程式會將目前在 DynamoDB 中的項目與嘗試寫入的項目進行比較。如果兩者相同，則操作視為成功。您可以使用 equalsIgnore 欄位來指定執行該比較時 AWS AppSync 應忽略的屬性清單。例如，如果唯一的差異是 version 屬性，它會將操作視為成功。此欄位為選用欄位。

consistentRead

當條件檢查失敗時，AWS AppSync 會使用強式一致讀取，從 DynamoDB 取得項目的目前值。您可以使用此欄位來告知 AWS AppSync DynamoDB 解析程式改用最終一致讀取。此欄位為選用，預設值為 true。

conditionalCheckFailedHandler

本節可讓您指定 AWS AppSync DynamoDB 解析程式在將 DynamoDB 中的目前值與預期結果進行比較後，如何處理條件檢查失敗。此區段為選用。若省略，則會預設為 Reject 策略。

strategy

AWS AppSync DynamoDB 解析程式在將 DynamoDB 中的目前值與預期結果進行比較後所採用的策略。此欄位為必填，且採用下列可能值：

Reject

變動會失敗，並會將錯誤新增至 GraphQL 回應。

Custom

The AWS AppSync DynamoDB 解析程式會叫用自訂 Lambda 函數，以決定如何處理條件檢查失敗。當 `strategy` 設定為 `Custom`，`lambdaArn` 欄位必須包含要叫用 Lambda 函數的 ARN。

lambdaArn

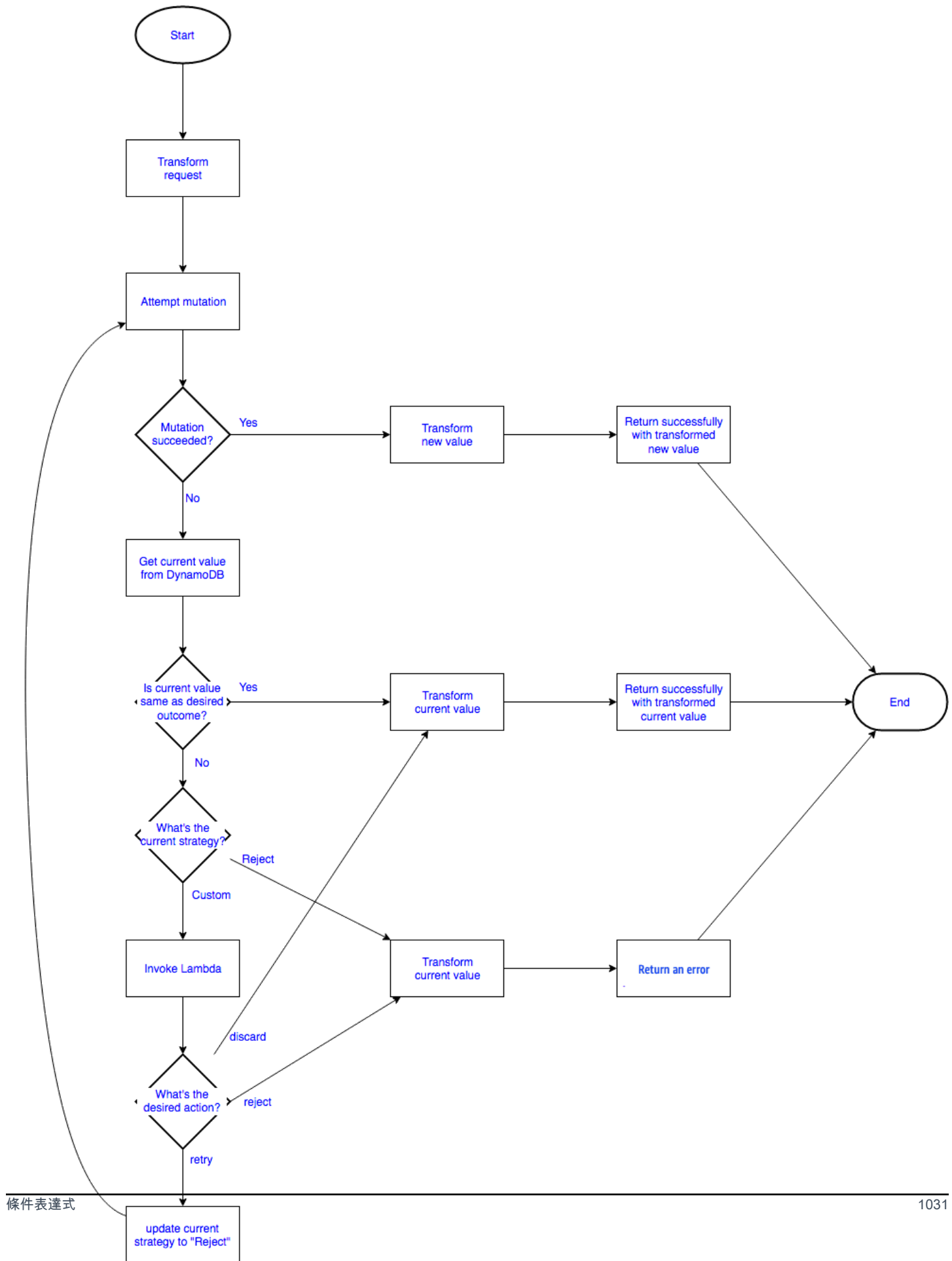
要叫用的 Lambda 函數 ARN，決定 AWS AppSync DynamoDB 解析程式應如何處理條件檢查失敗。只有在 `strategy` 設定為 `Custom` 時，此欄位才必須指定。如需如何使用此功能的詳細資訊，請參閱[處理條件檢查失敗](#)。

處理條件檢查失敗

根據預設，當條件檢查失敗時，AWS AppSync DynamoDB 解析程式會傳回變動的錯誤，以及 DynamoDB 中物件的目前值。不過，AWS AppSync DynamoDB 解析程式提供一些額外的功能，以協助開發人員處理一些常見的邊緣案例：

- If AWS AppSync DynamoDB 解析程式可以判斷 DynamoDB 中的目前值是否符合所需的結果，無論如何都會將操作視為成功。
- 您可以設定解析程式來叫用自訂 Lambda 函數，以決定 AWS AppSync DynamoDB 解析程式應如何處理失敗，而不是傳回錯誤。

此程序的流程圖為：



檢查所需結果

當條件檢查失敗時，AWS AppSync DynamoDB 解析程式會執行 `GetItem` DynamoDB 請求，從 DynamoDB 取得項目的目前值。在預設情況下，它會使用強式一致性讀取，但這可使用 `condition` 區塊中的 `consistentRead` 欄位來設定，並和預期結果進行比較：

- 對於 `PutItem` 操作，AWS AppSync DynamoDB 解析程式會將目前的值與其嘗試寫入的值進行比較，但不包括比較 `equalsIgnore` 中列出的任何屬性。如果項目相同，它會將操作視為成功，並傳回從 DynamoDB 擷取的項目。否則，其將按照設定的策略。

例如，如果 `PutItem` 要求映射文件外觀如下：

```
{
  "version" : "2017-02-28",
  "operation" : "PutItem",
  "key" : {
    "id" : { "S" : "1" }
  },
  "attributeValues" : {
    "name" : { "S" : "Steve" },
    "version" : { "N" : 2 }
  },
  "condition" : {
    "expression" : "version = :expectedVersion",
    "expressionValues" : {
      ":expectedVersion" : { "N" : 1 }
    },
    "equalsIgnore": [ "version" ]
  }
}
```

而目前在 DynamoDB 中的項目外觀如下：

```
{
  "id" : { "S" : "1" },
  "name" : { "S" : "Steve" },
  "version" : { "N" : 8 }
}
```

The AWS AppSync DynamoDB 解析程式會將嘗試寫入的項目與目前值進行比較，查看唯一的差異是 `version` 欄位，但因為它設定為忽略 `version` 欄位，它會將操作視為成功，並傳回從 DynamoDB 擷取的項目。

- 對於 `DeleteItem` 操作，AWS AppSync DynamoDB 解析程式會檢查以確認項目已從 DynamoDB 傳回。如果沒有項目傳回，則操作視為成功。否則，其將按照設定的策略。
- 對於 `UpdateItem` 操作，AWS AppSync DynamoDB 解析程式沒有足夠的資訊來判斷目前 DynamoDB 中的項目是否符合預期結果，因此遵循設定的策略。

如果 DynamoDB 中物件的目前狀態與預期結果不同，AWS AppSync DynamoDB 解析程式會遵循設定的策略，拒絕變動或叫用 Lambda 函數來決定接下來要做什麼。

遵循「拒絕」策略

遵循 `Reject` 策略時，AWS AppSync DynamoDB 解析程式會傳回變動的錯誤。

例如，假設變動要求如下：

```
mutation {
  updatePerson(id: 1, name: "Steve", expectedVersion: 1) {
    Name
    theVersion
  }
}
```

如果 DynamoDB 傳回的項目如下所示：

```
{
  "id" : { "S" : "1" },
  "name" : { "S" : "Steve" },
  "version" : { "N" : 8 }
}
```

而回應映射範本如下所示：

```
{
  "id" : $util.toJson($context.result.id),
  "Name" : $util.toJson($context.result.name),
  "theVersion" : $util.toJson($context.result.version)
}
```

GraphQL 回應的外觀如下所示：

```
{
  "data": null,
  "errors": [
    {
      "message": "The conditional request failed (Service: AmazonDynamoDBv2;
Status Code: 400; Error Code: ConditionalCheckFailedException; Request ID:
ABCDEFGHJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZ)"
      "errorType": "DynamoDB:ConditionalCheckFailedException",
      ...
    }
  ]
}
```

此外，如果傳回物件中的任何欄位皆由其他解析程式填入，且變動成功，則在 error 區段中傳回物件時，不會解析這些物件。

遵循「自訂」策略

遵循Custom策略時，AWS AppSync DynamoDB 解析程式會叫用 Lambda 函數來決定接下來要做什么。Lambda 函數會選擇下列其中一個選項：

- **reject 變動。**這可讓 AWS AppSync DynamoDB 解析程式表現得好像已設定的策略是 Reject，傳回 DynamoDB 中物件的變動錯誤和目前值，如上一節所述。
- **discard 變動。**這會通知 AWS AppSync DynamoDB 解析程式以無提示方式忽略條件檢查失敗，並在 DynamoDB 中傳回 值。
- **retry 變動。**這會通知 AWS AppSync DynamoDB 解析程式使用新的請求映射文件重試變動。

Lambda 叫用要求

The AWS AppSync DynamoDB 解析程式會叫用 中指定的 Lambda 函數lambdaArn。它會使用資料來源上所設定的相同 service-role-arn。叫用承載的結構如下：

```
{
  "arguments": { ... },
  "requestMapping": { ... },
  "currentValue": { ... },
  "resolver": { ... },
  "identity": { ... }
```

```
}
```

欄位定義如下：

arguments

來自 GraphQL 變動的引數。這與可在 `$context.arguments` 中要求映射文件取得的引數相同。

requestMapping

此操作的要求映射文件。

currentValue

DynamoDB 中物件的目前值。

resolver

有關 AWS AppSync 解析程式的資訊。

identity

發起人的相關資訊。這與可在 `$context.identity` 中要求映射文件取得的身分資訊相同。

承載的完整範例：

```
{
  "arguments": {
    "id": "1",
    "name": "Steve",
    "expectedVersion": 1
  },
  "requestMapping": {
    "version" : "2017-02-28",
    "operation" : "PutItem",
    "key" : {
      "id" : { "S" : "1" }
    },
    "attributeValues" : {
      "name" : { "S" : "Steve" },
      "version" : { "N" : 2 }
    },
    "condition" : {
      "expression" : "version = :expectedVersion",
      "expressionValues" : {
        ":expectedVersion" : { "N" : 1 }
      }
    }
  }
}
```

```

    },
    "equalsIgnore": [ "version" ]
  },
  },
  "currentValue": {
    "id" : { "S" : "1" },
    "name" : { "S" : "Steve" },
    "version" : { "N" : 8 }
  },
  "resolver": {
    "tableName": "People",
    "awsRegion": "us-west-2",
    "parentType": "Mutation",
    "field": "updatePerson",
    "outputType": "Person"
  },
  "identity": {
    "accountId": "123456789012",
    "sourceIp": "x.x.x.x",
    "user": "AIDAAAAAAAAAAAAAAAAAAAA",
    "userArn": "arn:aws:iam::123456789012:user/appsync"
  }
}

```

Lambda 叫用回應

Lambda 函數可以檢查調用承載並套用任何商業邏輯，以決定 AWS AppSync DynamoDB 解析程式應如何處理失敗。處理條件檢查失敗有三個選項：

- **reject 變動。**此選項的回應承載必須具有此架構：

```

{
  "action": "reject"
}

```

這可讓 AWS AppSync DynamoDB 解析程式表現得好像已設定的策略是 **Reject**，傳回 DynamoDB 中物件變動和目前值的錯誤，如上節所述。

- **discard 變動。**此選項的回應承載必須具有此架構：

```

{
  "action": "discard"
}

```



```
}
```

這會通知 AWS AppSync DynamoDB 解析程式以無提示方式忽略條件檢查失敗，並在 DynamoDB 中傳回 值。

- **retry 變動。**此選項的回應承載必須具有此架構：

```
{
  "action": "retry",
  "retryMapping": { ... }
}
```

這會通知 AWS AppSync DynamoDB 解析程式使用新的請求映射文件重試變動。retryMapping 區段的結構取決於 DynamoDB 操作，並且是該操作完整請求映射文件的子集。

若是 PutItem，retryMapping 區段的結構如下。如需 attributeValues 欄位的描述，請參閱 [PutItem](#)。

```
{
  "attributeValues": { ... },
  "condition": {
    "equalsIgnore" = [ ... ],
    "consistentRead" = true
  }
}
```

若是 UpdateItem，retryMapping 區段的結構如下。如需 update 區段的描述，請參閱 [UpdateItem](#)。

```
{
  "update" : {
    "expression" : "someExpression"
    "expressionNames" : {
      "#foo" : "foo"
    },
    "expressionValues" : {
      ":bar" : ... typed value
    }
  },
  "condition": {
    "consistentRead" = true
  }
}
```

```

    }
  }
}

```

若是 DeleteItem，retryMapping 區段的結構如下。

```

{
  "condition": {
    "consistentRead" = true
  }
}

```

無法指定使用不同的操作或索引鍵。The AWS AppSync DynamoDB 解析程式僅允許對相同物件重試相同的操作。此外，condition 區段不允許指定 conditionalCheckFailedHandler。如果重試失敗，AWS AppSync DynamoDB 解析程式會遵循 Reject 策略。

此為 Lambda 函數處理失敗 PutItem 要求的範例。商業邏輯的重點是發起人為何。如果是由 提出 jeffTheAdmin，則會重試請求，expectedVersion 並從目前 DynamoDB 中的項目更新 version 和。否則，它會拒絕變動。

```

exports.handler = (event, context, callback) => {
  console.log("Event: " + JSON.stringify(event));

  // Business logic goes here.

  var response;
  if ( event.identity.user == "jeffTheAdmin" ) {
    response = {
      "action" : "retry",
      "retryMapping" : {
        "attributeValues" : event.requestMapping.attributeValues,
        "condition" : {
          "expression" : event.requestMapping.condition.expression,
          "expressionValues" :
event.requestMapping.condition.expressionValues
        }
      }
    }
    response.retryMapping.attributeValues.version = { "N" :
event.currentValue.version.N + 1 }
    response.retryMapping.condition.expressionValues[':expectedVersion'] =
event.currentValue.version
  }
}

```

```
    } else {  
        response = { "action" : "reject" }  
    }  
  
    console.log("Response: " + JSON.stringify(response))  
    callback(null, response)  
};
```

交易條件表達式

交易條件表達式可在 `TransactWriteItems` 中四種類型操作的請求映射範本中使用，即 `PutItem`、`DeleteItem`、`UpdateItem` 以及 `ConditionCheck`。

對於 `PutItem`、`DeleteItem` 和 `UpdateItem`，交易條件表達式是選用的。對於 `ConditionCheck`，交易條件表達式是必要的。

範例 1

以下交易 `DeleteItem` 映射文件沒有條件表達式。因此，它會刪除 DynamoDB 中的項目。

```
{  
  "version": "2018-05-29",  
  "operation": "TransactWriteItems",  
  "transactItems": [  
    {  
      "table": "posts",  
      "operation": "DeleteItem",  
      "key": {  
        "id": { "S" : "1" }  
      }  
    }  
  ]  
}
```

範例 2

下列交易 `DeleteItem` 映射文件確實具有交易條件表達式，只有在該文章的作者等於特定名稱時，才會允許操作成功。

```
{  
  "version": "2018-05-29",
```

```

"operation": "TransactWriteItems",
"transactItems": [
  {
    "table": "posts",
    "operation": "DeleteItem",
    "key": {
      "id": { "S" : "1" }
    }
    "condition": {
      "expression": "author = :author",
      "expressionValues": {
        ":author": { "S" : "Chunyan" }
      }
    }
  }
]
}

```

如果條件檢查失敗，會導致 `TransactionCanceledException`，並且會在 `$ctx.result.cancellationReasons` 中傳回錯誤詳細資料。請注意，根據預設，DynamoDB 中導致條件檢查失敗的舊項目會在 `$ctx.result.cancellationReasons` 中傳回。

指定條件

`PutItem`、`UpdateItem` 和 `DeleteItem` 要求映射文件都允許可指定選用的 `condition` 區段。若省略，則不會有條件檢查。若指定，條件必須為 `true`，操作才會成功。`ConditionCheck` 必須有待指定的 `condition` 區段。條件必須為 `true`，整個交易才會成功。

`condition` 區段的結構如下：

```

"condition": {
  "expression": "someExpression",
  "expressionNames": {
    "#foo": "foo"
  },
  "expressionValues": {
    ":bar": ... typed value
  },
  "returnValuesOnConditionCheckFailure": false
}

```

下列欄位指定條件：

expression

更新表達式本身。如需如何編寫條件表達式的詳細資訊，請參閱 [DynamoDB ConditionExpressions 文件](#)。必須指定此欄位。

expressionNames

表達式屬性名稱預留位置的替代，形式為索引鍵值對。索引鍵對應至表達式中使用的名稱預留位置，且值必須是對應於 DynamoDB 中項目屬性名稱的字串。此欄位為選用的，應只能填入於表達式中所用表達式屬性名稱預留位置的替代。

expressionValues

表達式屬性值預留位置的替代，形式為索引值對。鍵對應用於表達式的值預留位置，值必須是類型值。如需如何指定「類型值」的詳細資訊，請參閱〈類型系統 (請求映射)〉。此必須指定。此欄位為選用的，應只能填入用於表達式中表達式屬性值預留位置的替代。

returnValuesOnConditionCheckFailure

指定是否在條件檢查失敗時擷取 DynamoDB 中的項目。擷取的項目將位於 `$ctx.result.cancellationReasons[$index].item` 中，其中 `$index` 是條件檢查失敗之請求項目的索引。此值預設為 `true`。

投影

使用 `GetItem`、`BatchGetItem`、`Scan` 和 `QueryTransactGetItems` 操作讀取 DynamoDB 中的物件時，您可以選擇指定投影，以識別您想要的屬性。投影具有下列結構，類似於篩選條件：

```
"projection" : {
  "expression" : "projection expression"
  "expressionNames" : {
    "#name" : "name",
  }
}
```

欄位定義如下：

expression

投影表達式，這是字串。若要擷取單一屬性，請指定其名稱。對於多個屬性，名稱必須是逗號分隔值。如需撰寫投影表達式的詳細資訊，請參閱 [DynamoDB 投影表達式](#) 文件。此欄位為必填。

expressionNames

以鍵/值對形式取代表達式屬性名稱預留位置。索引鍵對應至用於 expression 的名稱預留位置。值必須是對應於 DynamoDB 中項目屬性名稱的字串。此欄位是選用的，應該只填入中使用的表達式屬性名稱預留位置的替換 expression。如需的詳細資訊 expressionNames，請參閱 [DynamoDB 文件](#)。

範例 1

下列範例是 VTL 映射範本的投影區段，其中只有屬性 id author 和 會從 DynamoDB 傳回：

```
"projection" : {  
  "expression" : "#author, id",  
  "expressionNames" : {  
    "#author" : "author"  
  }  
}
```

Tip

您可以使用 [\\$context.info.selectionSetList](#) 存取 GraphQL 請求選取集。此欄位可讓您根據您的需求動態架構投影表達式。

Note

搭配 Query 和 Scan 操作使用投影表達式時，的值 select 必須是 SPECIFIC_ATTRIBUTES。如需詳細資訊，請參閱 [DynamoDB 文件](#)。

RDS 的 AWS AppSync 解析程式映射範本參考

AWS AppSync RDS 解析程式映射範本可讓開發人員將 SQL 查詢傳送至 Amazon Aurora Serverless 的資料 API，並取回這些查詢的結果。

請求映射範本

RDS 要求映射範本非常簡單：

```
{
  "version": "2018-05-29",
  "statements": [],
  "variableMap": {},
  "variableTypeHintMap": {}
}
```

以下是 RDS 請求映射範本在解析後呈現的 JSON 結構描述。

```
{
  "definitions": {},
  "$schema": "https://json-schema.org/draft-07/schema#",
  "$id": "https://example.com/root.json",
  "type": "object",
  "title": "The Root Schema",
  "required": [
    "version",
    "statements",
    "variableMap"
  ],
  "properties": {
    "version": {
      "$id": "#/properties/version",
      "type": "string",
      "title": "The Version Schema",
      "default": "",
      "examples": [
        "2018-05-29"
      ],
      "enum": [
        "2018-05-29"
      ],
      "pattern": "^(.*)$"
    },
    "statements": {
      "$id": "#/properties/statements",
      "type": "array",
      "title": "The Statements Schema",
      "items": {
        "$id": "#/properties/statements/items",
        "type": "string",
        "title": "The Items Schema",
        "default": "",

```

```

      "examples": [
        "SELECT * from BOOKS"
      ],
      "pattern": "^(.*)$"
    }
  },
  "variableMap": {
    "$id": "#/properties/variableMap",
    "type": "object",
    "title": "The Variablemap Schema"
  },
  "variableTypeHintMap": {
    "$id": "#/properties/variableTypeHintMap",
    "type": "object",
    "title": "The variableTypeHintMap Schema"
  }
}

```

以下是具有靜態查詢的請求映射範本範例：

```

{
  "version": "2018-05-29",
  "statements": [
    "select title, isbn13 from BOOKS where author = 'Mark Twain'"
  ]
}

```

版本

所有請求映射範本通用的版本欄位定義範本使用的版本。版本欄位為必要欄位。值「2018-05-29」是 Amazon RDS 映射範本唯一支援的版本。

```
"version": "2018-05-29"
```

陳述式和 VariableMap

陳述式陣列是開發人員所提供查詢的預留位置。目前，每個請求映射範本最多支援兩個查詢。variableMap 是選用欄位，其中包含別名，可用於使 SQL 陳述式更短且更易讀。例如，下列是可能的：


```
{
  "version": "2018-05-29",
  "statements": [
    "insert into BOOKS VALUES (:AUTHOR, :TITLE, :ISBN13)",
    "select * from BOOKS WHERE isbn13 = :ISBN13"
  ],
  "variableMap": {
    ":AUTHOR": $util.toJson($ctx.args.newBook.author),
    ":TITLE": $util.toJson($ctx.args.newBook.title),
    ":ISBN13": $util.toJson($ctx.args.newBook.isbn13)
  }
}
```

AWS AppSync 將使用變數映射值來建構將傳送至 Amazon Aurora Serverless Data API 的 [SqlParameterized](#) 查詢。SQL 陳述式會使用變數映射中提供的參數執行，可消除 SQL 插入的風險。

VariableTypeHintMap

`variableTypeHintMap` 是選用欄位，包含可用於傳送 [SQL 參數](#) 類型提示的別名類型。這些類型提示可避免 SQL 陳述式中的明確轉換，使其更短。例如，下列是可能的：

```
{
  "version": "2018-05-29",
  "statements": [
    "insert into LOGINDATA VALUES (:ID, :TIME)",
    "select * from LOGINDATA WHERE id = :ID"
  ],
  "variableMap": {
    ":ID": $util.toJson($ctx.args.id),
    ":TIME": $util.toJson($ctx.args.time)
  },
  "variableTypeHintMap": {
    ":id": "UUID",
    ":time": "TIME"
  }
}
```

AWS AppSync 將使用變數映射值來建構傳送至 Amazon Aurora Serverless Data API 的查詢。它也會使用 `variableTypeHintMap` 資料，並將類型的資訊傳送至 RDS。您可以在 [此處](#) 找到 RDS 支援的 `typeHints`。

OpenSearch 的 AWS AppSync 解析程式映射範本參考

Note

我們現在主要支援 APPSYNC_JS 執行期及其文件。請考慮[在此處](#)使用 APPSYNC_JS 執行期及其指南。

Amazon OpenSearch Service 的 AWS AppSync 解析程式可讓您使用 GraphQL 將資料存放在帳戶中現有的 OpenSearch Service 網域中。此解析程式的運作方式是允許您將傳入的 GraphQL 請求映射至 OpenSearch Service 請求，然後將 OpenSearch Service 回應映射回 GraphQL。本節說明支援 OpenSearch Service 操作的映射範本。

請求映射範本

大多數 OpenSearch Service 請求映射範本都有共同的結構，其中只有幾個部分變更。下列範例會根據 OpenSearch Service 網域執行搜尋，其中文件是以名為的索引組織post。搜尋參數定義於 body 區段，許多常見的查詢子句定義於 query 欄位。此範例將搜尋在 "Nadia" 欄位中包含 "Bailey" 或 author (或兩者) 的文件：

```
{
  "version": "2017-02-28",
  "operation": "GET",
  "path": "/post/_search",
  "params": {
    "headers": {},
    "queryString": {},
    "body": {
      "from": 0,
      "size": 50,
      "query": {
        "bool": {
          "should": [
            { "match": { "author": "Nadia" } },
            { "match": { "author": "Bailey" } }
          ]
        }
      }
    }
  }
}
```

```
}
```

回應映射範本

如同其他資料來源，OpenSearch Service 會將回應傳送至需要轉換為 GraphQL 的 AWS AppSync。

大多數 GraphQL 查詢正在從 OpenSearch Service 回應中尋找 `_source` 欄位。由於您可以執行搜尋以傳回個別文件或文件清單，因此 OpenSearch Service 中有兩種常用的回應映射範本：

結果清單

```
[
  #foreach($entry in $context.result.hits.hits)
    #if( $velocityCount > 1 ) , #end
    $utils.toJson($entry.get("_source"))
  #end
]
```

個別項目

```
$utils.toJson($context.result.get("_source"))
```

operation 欄位

Note

這僅適用於請求映射範本。

AWS AppSync 傳送至 OpenSearch Service 網域的 HTTP 方法或動詞 (GET、POST、PUT、HEAD 或 DELETE)。金鑰與值皆必須為字串。

```
"operation" : "PUT"
```

path 欄位

Note

這僅適用於請求映射範本。

來自 AWS AppSync 的 OpenSearch Service 請求搜尋路徑。這會形成操作的 HTTP 動詞的 URL。金鑰與值皆必須為字串。

```
"path" : "/<indexname>/_doc/<_id>"
"path" : "/<indexname>/_doc"
"path" : "/<indexname>/_search"
"path" : "/<indexname>/_update/<_id>"
```

評估映射範本時，此路徑會做為 HTTP 請求的一部分傳送，包括 OpenSearch Service 網域。例如，之前的範例可轉譯為：

```
GET https://opensearch-domain-name.REGION.es.amazonaws.com/indexname/type/_search
```

params 欄位

Note

這僅適用於請求映射範本。

用來指定您的搜尋執行的動作，最常見的方式是設定內文中的查詢值。不過，有多項其他功能可設定，例如回應的格式。

- 標頭

標頭資訊，以金鑰值對形式。金鑰與值皆必須為字串。例如：

```
"headers" : {
  "Content-Type" : "application/json"
}
```

Note

AWS AppSync 目前僅支援 JSON 做為 Content-Type。

- queryString

金鑰值對，指定常見的選項，例如 JSON 回應的程式碼格式。金鑰與值皆必須為字串。例如，如果您希望獲得非常完整格式的 JSON，請使用：

```
"queryString" : {  
  "pretty" : "true"  
}
```

- 本文

這是請求的主要部分，允許 AWS AppSync 為您的 OpenSearch Service 網域製作格式正確的搜尋請求。金鑰必須是由物件組成的字串。以下顯示幾個示範。

範例 1

傳回城市符合「seattle」的所有文件：

```
"body":{  
  "from":0,  
  "size":50,  
  "query" : {  
    "match" : {  
      "city" : "seattle"  
    }  
  }  
}
```

範例 2

傳回所有符合「washington」做為城市或州的文件：

```
"body":{  
  "from":0,  
  "size":50,  
  "query" : {  
    "multi_match" : {  
      "query" : "washington",  
      "fields" : ["city", "state"]  
    }  
  }  
}
```

傳遞變數

Note

這僅適用於請求映射範本。

您也可以將變數做為 VTL 陳述式中評估的一部分進行傳遞。例如，假設您有一個 GraphQL 查詢如下：

```
query {
  searchForState(state: "washington"){
    ...
  }
}
```

此映射範本可採取狀態做為引數：

```
"body":{
  "from":0,
  "size":50,
  "query" : {
    "multi_match" : {
      "query" : "$context.arguments.state",
      "fields" : ["city", "state"]
    }
  }
}
```

如需您可包含在 VTL 中的公用程式清單，請參閱[存取請求標頭](#)。

Lambda 的 AWS AppSync 解析程式映射範本參考

Note

我們現在主要支援 APPSYNC_JS 執行期及其文件。請考慮[在此處](#)使用 APPSYNC_JS 執行期及其指南。

您可以使用 AWS AppSync 函數和解析程式來叫用您帳戶中的 Lambda 函數。您可以在將請求承載傳回用戶端之前，先從 Lambda 函數塑造請求承載和回應。您也可以使用映射範本，向 AWS AppSync 提供有關要叫用之操作性質的提示。本節說明支援 Lambda 操作的不同映射範本。

請求映射範本

Lambda 請求映射範本會處理與 Lambda 函數相關的欄位：

```
{
  "version": string,
  "operation": Invoke|BatchInvoke,
  "payload": any type,
  "invocationType": RequestResponse|Event
}
```

這是解決時 Lambda 請求映射範本的 JSON 結構描述表示法：

```
{
  "definitions": {},
  "$schema": "https://json-schema.org/draft-06/schema#",
  "$id": "https://aws.amazon.com/appsync/request-mapping-template.json",
  "type": "object",
  "properties": {
    "version": {
      "$id": "/properties/version",
      "type": "string",
      "enum": [
        "2018-05-29"
      ],
      "title": "The Mapping template version.",
      "default": "2018-05-29"
    },
    "operation": {
      "$id": "/properties/operation",
      "type": "string",
      "enum": [
        "Invoke",
        "BatchInvoke"
      ],
      "title": "The Mapping template operation.",
      "description": "What operation to execute.",
      "default": "Invoke"
    }
  }
}
```

```

    },
    "payload": {},
    "invocationType": {
      "$id": "/properties/invocationType",
      "type": "string",
      "enum": [
        "RequestResponse",
        "Event"
      ],
      "title": "The Mapping template invocation type.",
      "description": "What invocation type to execute.",
      "default": "RequestResponse"
    }
  },
  "required": [
    "version",
    "operation"
  ],
  "additionalProperties": false
}

```

以下是使用 `invoke` 操作的範例，其承載資料是來自 GraphQL 結構描述 `getPost` 的欄位，以及其內容中的引數：

```

{
  "version": "2018-05-29",
  "operation": "Invoke",
  "payload": {
    "field": "getPost",
    "arguments": $util.toJson($context.arguments)
  }
}

```

整個映射文件會做為 Lambda 函數的輸入傳遞，因此先前的範例現在如下所示：

```

{
  "version": "2018-05-29",
  "operation": "Invoke",
  "payload": {
    "field": "getPost",
    "arguments": {
      "id": "postId1"
    }
  }
}

```



```
}  
}
```

版本

為所有請求映射範本通用，`version`定義範本使用的版本。`version` 為必要項目，且為靜態值：

```
"version": "2018-05-29"
```

作業

Lambda 資料來源可讓您在 `operation` 欄位中定義兩個操作：`Invoke`和 `BatchInvoke`。`Invoke`操作 lets AWS AppSync 知道 會為每個 GraphQL 欄位解析程式呼叫 Lambda 函數。`BatchInvoke` instructs AWS AppSync 會批次處理目前 GraphQL 欄位的請求。`operation` 欄位是必要的。

對於 `Invoke`，解析的請求映射範本符合 Lambda 函數的輸入承載。讓我們修改上述範例：

```
{  
  "version": "2018-05-29",  
  "operation": "Invoke",  
  "payload": {  
    "arguments": $util.toJson($context.arguments)  
  }  
}
```

這會解析並傳遞給 Lambda 函數，看起來可能會類似這樣：

```
{  
  "version": "2018-05-29",  
  "operation": "Invoke",  
  "payload": {  
    "arguments": {  
      "id": "postId1"  
    }  
  }  
}
```

對於 `BatchInvoke`，映射範本會套用至批次中的每個欄位解析程式。為了簡潔起見，AWS AppSync 會將所有解析的映射範本 `payload` 值合併到符合映射範本之單一物件下的清單中。以下範例範本顯示此合併：

```
{
  "version": "2018-05-29",
  "operation": "BatchInvoke",
  "payload": $util.toJson($context)
}
```

此範本已解析為以下映射文件：

```
{
  "version": "2018-05-29",
  "operation": "BatchInvoke",
  "payload": [
    {...}, // context for batch item 1
    {...}, // context for batch item 2
    {...}  // context for batch item 3
  ]
}
```

payload 清單的每個元素對應至單一批次項目。Lambda 函數也預期會傳回符合請求中傳送項目順序的清單形狀回應：

```
[
  { "data": {...}, "errorMessage": null, "errorType": null }, // result for batch item 1
  { "data": {...}, "errorMessage": null, "errorType": null }, // result for batch item 2
  { "data": {...}, "errorMessage": null, "errorType": null }  // result for batch item 3
]
```

承載

payload 欄位是用來將任何格式正確的 JSON 傳遞至 Lambda 函數的容器。如果 operation 欄位設定為 BatchInvoke，AWS AppSync 會將現有 payload 值包裝成清單。此 payload 欄位為選用。

調用類型

Lambda 資料來源可讓您定義兩種叫用類型：RequestResponse 和 Event。叫用類型與 [Lambda API](#) 中定義的叫用類型同義。RequestResponse 叫用類型 lets AWS AppSync 會同步呼叫您的 Lambda 函數，以等待回應。Event 調用可讓您以非同步方式調用 Lambda 函數。如需 Lambda 如何

處理 Event 調用類型請求的詳細資訊，請參閱[非同步調用](#)。此 `invocationType` 欄位為選用。如果請求中未包含此欄位，AWS AppSync 會預設為 `RequestResponse` 調用類型。

對於任何 `invocationType` 欄位，解析的請求符合 Lambda 函數的輸入承載。讓我們修改上述範例：

```
{
  "version": "2018-05-29",
  "operation": "Invoke",
  "invocationType": "Event"
  "payload": {
    "arguments": $util.toJson($context.arguments)
  }
}
```

這會解析並傳遞給 Lambda 函數，看起來可能會類似這樣：

```
{
  "version": "2018-05-29",
  "operation": "Invoke",
  "invocationType": "Event",
  "payload": {
    "arguments": {
      "id": "postId1"
    }
  }
}
```

當 `BatchInvoke` 操作與 Event 調用類型欄位搭配使用時，AWS AppSync 會以上述相同方式合併欄位解析程式，並將請求做為非同步事件傳遞給您的 Lambda 函數，而 `payload` 為值清單。我們建議您停用 Event 呼叫類型解析程式的解析程式快取，因為如果快取命中，這些解析程式不會傳送至 Lambda。

回應映射範本

如同其他資料來源，您的 Lambda 函數會傳送回應至 AWS AppSync，該回應必須轉換為 GraphQL 類型。

Lambda 函數的結果會在可透過 Velocity 範本語言 (VTL) `$context.result` 屬性取得的 `context` 物件上設定。

如果您的 Lambda 函式回應形狀完全匹配 GraphQL 類型形狀時，您可以使用以下回應映射範本轉送回應：

```
$util.toJson($context.result)
```

沒有任何必要欄位或形狀限制適用於回應映射範本。不過，由於 GraphQL 是強類型，解析後的映射範本必須符合預期的 GraphQL 類型。

Lambda 函數批次回應

如果 operation 欄位設定為 BatchInvoke，AWS AppSync 預期會從 Lambda 函數傳回項目清單。為了讓 AWS AppSync 將每個結果對應回原始請求項目，回應清單的大小和順序必須相符。在回應清單中有 null 項目是有效的；\$ctx.result 會相應地設定為 null。

Direct Lambda 解析程式

如果您想要完全避免使用映射範本，AWS AppSync 可以為您的 Lambda 函數提供預設承載，並為 GraphQL 類型提供預設 Lambda 函數回應。您可以選擇提供請求範本、回應範本或兩者都提供，而 AWS AppSync 會相應地處理它。

Direct Lambda 請求映射範本

未提供請求映射範本時，AWS AppSync 會將 Context 物件直接傳送到 Lambda 函數做為 Invoke 操作。如需 Context 物件結構的詳細資訊，請參閱 [AWS AppSync 解析程式映射範本內容參考](#)。

Direct Lambda 回應映射範本

未提供回應映射範本時，AWS AppSync 會在收到 Lambda 函數的回應時，執行下列兩項操作之一。如果您未提供請求映射範本，或者您提供了版本為 2018-05-29 的請求映射範本，則回應將等同於下列回應映射範本：

```
#if($ctx.error)
    $util.error($ctx.error.message, $ctx.error.type, $ctx.result)
#end
$util.toJson($ctx.result)
```

如果您提供具有 2017-02-28 版本的範本，回應邏輯函數相當於下列回應映射範本：

```
$util.toJson($ctx.result)
```

實際上，映射範本繞過的運作類似於使用某些映射範本，如上述範例所示。不過，在幕後，完全規避對映射範本的評估。由於範本評估步驟已略過，因此與具有需要評估之回應映射範本的 Lambda 函數相比，應用程式在某些案例中的回應期間可能會遇到較少的額外負荷和延遲。

Direct Lambda Resolver 回應中的自訂錯誤處理

您可以透過提出自訂例外狀況，自訂來自 Direct Lambda Resolvers 調用之 Lambda 函數的錯誤回應。下列範例示範如何使用 JavaScript 建立自訂例外狀況：

```
class CustomException extends Error {
  constructor(message) {
    super(message);
    this.name = "CustomException";
  }
}

throw new CustomException("Custom message");
```

引發例外狀況時 `message`，`errorType` 和 分別 `errorMessage` 是擲回之自訂錯誤的 `name` 和 。

如果 `errorType` 是 `UnauthorizedException`，AWS AppSync 會傳回預設訊息 ("You are not authorized to make this call.")，而不是自訂訊息。

下列程式碼片段是示範自訂 的範例 GraphQL 回應 `errorType`：

```
{
  "data": {
    "query": null
  },
  "errors": [
    {
      "path": [
        "query"
      ],
      "data": null,
      "errorType": "CustomException",
      "errorInfo": null,
      "locations": [
        {
          "line": 5,
          "column": 10,
          "sourceName": null
        }
      ],
      "message": "Custom Message"
    }
  ]
}
```

```
}
```

Direct Lambda 解析程式：啟用批次處理

您可以在解析程式maxBatchSize上設定，以啟用 Direct Lambda Resolver 的批次處理。當maxBatchSize 設定為大於 Direct Lambda 0 解析程式的值時，AWS AppSync 會以最大的大小批次將請求傳送到 Lambda 函數maxBatchSize。

在 Direct Lambda 解析程式0上maxBatchSize將 設定為 會關閉批次處理。

如需使用 Lambda 解析程式批次處理方式的詳細資訊，請參閱 [進階使用案例：批次處理](#)。

請求映射範本

啟用批次處理且未提供請求映射範本時，AWS AppSync 會將Context物件清單作為BatchInvoke操作直接傳送至 Lambda 函數。

回應映射範本

啟用批次處理且未提供回應映射範本時，回應邏輯相當於下列回應映射範本：

```
#if( $context.result && $context.result.errorMessage )
    $utils.error($context.result.errorMessage, $context.result.errorType,
    $context.result.data)
#else
    $utils.toJson($context.result.data)
#end
```

Lambda 函數必須以與傳送的Context物件清單相同的順序傳回結果清單。您可以透過errorType為特定結果提供 errorMessage和 來傳回個別錯誤。清單中的每個結果都有下列格式：

```
{
  "data" : { ... }, // your data
  "errorMessage" : { ... }, // optional, if included an error entry is added to the
  "errors" object in the AppSync response
  "errorType" : { ... } // optional, the error type
}
```

Note

目前會忽略結果物件中的其他欄位。

處理來自 Lambda 的錯誤

您可以在 Lambda 函數中擲回例外狀況或錯誤，以傳回所有結果的錯誤。如果批次請求的承載請求或回應大小太大，Lambda 會傳回錯誤。在這種情況下，您應該考慮減少您的 `maxBatchSize` 或減少回應承載的大小。

如需處理個別錯誤的資訊，請參閱 [傳回個別錯誤](#)。

Lambda 函數範例

使用以下結構描述，您可以為 `Post.relatedPosts` 欄位解析程式建立 Direct Lambda Resolver，並透過在 `maxBatchSize` 上設定 來啟用批次處理 0：

```
schema {
  query: Query
  mutation: Mutation
}

type Query {
  getPost(id:ID!): Post
  allPosts: [Post]
}

type Mutation {
  addPost(id: ID!, author: String!, title: String, content: String, url: String):
  Post!
}

type Post {
  id: ID!
  author: String!
  title: String
  content: String
  url: String
  ups: Int
  downs: Int
  relatedPosts: [Post]
}
```

在下列查詢中，將會呼叫 Lambda 函數與批次請求來解析 `relatedPosts`：

```
query getAllPosts {
  allPosts {
```

```
    id
    relatedPosts {
      id
    }
  }
}
```

以下是 Lambda 函數的簡單實作：

```
const posts = {
  1: {
    id: '1',
    title: 'First book',
    author: 'Author1',
    url: 'https://amazon.com/',
    content:
      'SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1',
    ups: '100',
    downs: '10',
  },
  2: {
    id: '2',
    title: 'Second book',
    author: 'Author2',
    url: 'https://amazon.com',
    content: 'SAMPLE TEXT AUTHOR 2 SAMPLE TEXT AUTHOR 2 SAMPLE TEXT',
    ups: '100',
    downs: '10',
  },
  3: { id: '3', title: 'Third book', author: 'Author3', url: null, content: null, ups: null, downs: null },
  4: {
    id: '4',
    title: 'Fourth book',
    author: 'Author4',
    url: 'https://www.amazon.com/',
    content:
      'SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4',
    ups: '1000',
    downs: '0',
  },
}
```



```
    },
    5: {
      id: '5',
      title: 'Fifth book',
      author: 'Author5',
      url: 'https://www.amazon.com/',
      content: 'SAMPLE TEXT AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT',
      ups: '50',
      downs: '0',
    },
  },
}

const relatedPosts = {
  1: [posts['4']],
  2: [posts['3'], posts['5']],
  3: [posts['2'], posts['1']],
  4: [posts['2'], posts['1']],
  5: [],
}

exports.handler = async (event) => {
  console.log('event ->', event)
  // retrieve the ID of each post
  const ids = event.map((context) => context.source.id)
  // fetch the related posts for each post id
  const related = ids.map((id) => relatedPosts[id])

  // return the related posts; or an error if none were found
  return related.map((r) => {
    if (r.length > 0) {
      return { data: r }
    } else {
      return { data: null, errorMessage: 'Not found', errorType: 'ERROR' }
    }
  })
}
```

EventBridge 的 AWS AppSync 解析程式映射範本參考

Note

我們現在主要支援 APPSYNC_JS 執行期及其文件。請考慮[在此處](#)使用 APPSYNC_JS 執行期及其指南。

與 EventBridge 資料來源搭配使用的 AWS AppSync 解析程式映射範本可讓您將自訂事件傳送至 Amazon EventBridge 匯流排。

請求映射範本

PutEvents 請求映射範本可讓您將多個自訂事件傳送至 EventBridge 事件匯流排。映射文件結構如下：

```
{
  "version" : "2018-05-29",
  "operation" : "PutEvents",
  "events" : [{}]
```

以下是 EventBridge 的請求映射範本範例：

```
{
  "version": "2018-05-29",
  "operation": "PutEvents",
  "events": [{
    "source": "com.mycompany.myapp",
    "detail": {
      "key1" : "value1",
      "key2" : "value2"
    },
    "detailType": "myDetailType1"
  },
  {
    "source": "com.mycompany.myapp",
    "detail": {
      "key3" : "value3",
      "key4" : "value4"
    }
  }
]
```

```

    },
    "detailType": "myDetailType2",
    "resources" : ["Resource1", "Resource2"],
    "time" : "2023-01-01T00:30:00.000Z"
  }
]
}

```

回應映射範本

如果PutEvents操作成功，EventBridge 的回應會包含在 `$ctx.result`：

```

#if($ctx.error)
  $util.error($ctx.error.message, $ctx.error.type, $ctx.result)
#end
$util.toJson($ctx.result)

```

執行 `InternalExceptions` 或 等PutEvents操作時發生的錯誤Timeouts將出現在 `$ctx.error`。如需 EventBridge 常見錯誤的清單，請參閱 [EventBridge 常見錯誤參考](#)。

`result` 將採用下列格式：

```

{
  "Entries" [
    {
      "ErrorCode" : String,
      "ErrorMessage" : String,
      "EventId" : String
    }
  ],
  "FailedEntryCount" : number
}

```

- 項目

擷取的事件結果成功和失敗。如果擷取成功，項目EventID中會有 。否則，您可以使用 `ErrorCode` 和 `ErrorMessage` 來識別項目的問題。

對於每個記錄，回應元素的索引與請求陣列中的索引相同。

- FailedEntryCount

失敗項目的數量。此值以整數表示。

如需 回應的詳細資訊PutEvents，請參閱 [PutEvents](#)。

範例回應 1

下列範例是具有兩個成功事件PutEvents的操作：

```
{
  "Entries" : [
    {
      "EventId": "11710aed-b79e-4468-a20b-bb3c0c3b4860"
    },
    {
      "EventId": "d804d26a-88db-4b66-9eaf-9a11c708ae82"
    }
  ],
  "FailedEntryCount" : 0
}
```

範例回應 2

下列範例是具有三個事件PutEvents的操作，兩個成功，一個失敗：

```
{
  "Entries" : [
    {
      "EventId": "11710aed-b79e-4468-a20b-bb3c0c3b4860"
    },
    {
      "EventId": "d804d26a-88db-4b66-9eaf-9a11c708ae82"
    },
    {
      "ErrorCode" : "SampleErrorCode",
      "ErrorMessage" : "Sample Error Message"
    }
  ],
  "FailedEntryCount" : 1
}
```

PutEvents 欄位

PutEvents 包含下列映射範本欄位：

- 版本

所有請求映射範本通用的 version 欄位定義範本使用的版本。此欄位為必填。此值2018-05-29是 EventBridge 映射範本唯一支援的版本。

- 操作

唯一支援的操作是 PutEvents。此操作可讓您將自訂事件新增至事件匯流排。

- 事件

將新增至事件匯流排的事件陣列。此陣列的配置應為 1 - 10 個項目。

Event 物件是具有下列欄位的有效 JSON 物件：

- "source"：定義事件來源的字串。
- "detail"：JSON 物件，可用來連接事件的相關資訊。此欄位可以是空的映射 { } ()。
- "detailType"：識別事件類型的字串。
- "resources"：可識別事件中涉及資源的字串 JSON 陣列。此欄位可以是空陣列。
- "time"：以字串形式提供的事件時間戳記。這應該遵循 [RFC3339](#) 時間戳記格式。

以下程式碼片段是一些有效Event物件的範例：

範例 1

```
{
  "source" : "source1",
  "detail" : {
    "key1" : [1,2,3,4],
    "key2" : "strval"
  },
  "detailType" : "sampleDetailType",
  "resources" : ["Resource1", "Resource2"],
  "time" : "2022-01-10T05:00:10Z"
}
```

範例 2

```
{
  "source" : "source1",
  "detail" : {},
  "detailType" : "sampleDetailType"
}
```

範例 3

```
{
  "source" : "source1",
  "detail" : {
    "key1" : 1200
  },
  "detailType" : "sampleDetailType",
  "resources" : []
}
```

None 資料來源的 AWS AppSync 解析程式映射範本參考

Note

我們現在主要支援 APPSYNC_JS 執行期及其文件。請考慮[在此處](#)使用 APPSYNC_JS 執行期及其指南。

與類型為 None 的資料來源搭配使用的 AWS AppSync 解析程式映射範本，可讓您塑造 AWS AppSync 本機操作的請求。

請求映射範本

映射範本非常簡單，可讓您透過 payload 欄位來傳遞任意數量的內容資訊。

```
{
  "version": string,
  "payload": any type
}
```

以下是請求映射範本在解析後呈現的 JSON 結構描述：

```
{
```

```

"definitions": {},
"$schema": "https://json-schema.org/draft-06/schema#",
"$id": "https://aws.amazon.com/appsync/request-mapping-template.json",
"type": "object",
"properties": {
  "version": {
    "$id": "/properties/version",
    "type": "string",
    "enum": [
      "2018-05-29"
    ],
    "title": "The Mapping template version.",
    "default": "2018-05-29"
  },
  "payload": {}
},
"required": [
  "version"
],
"additionalProperties": false
}

```

以下是 欄位引數透過 VTL 內容屬性 傳遞的範例 `$context.arguments`：

```

{
  "version": "2018-05-29",
  "payload": $util.toJson($context.arguments)
}

```

`payload` 欄位的值將轉發到回應映射範本，並可在 VTL 內容屬性 (`$context.result`) 上使用。

這是代表 `payload` 欄位插入值的範例：

```

{
  "id": "postId1"
}

```

版本

所有請求映射範本通用的 `version` 欄位定義範本使用的版本。

`version` 欄位是必要的。

範例：

```
"version": "2018-05-29"
```

承載

payload 欄位是一種容器，您可以用來將任何格式正確的 JSON 傳遞到回應映射範本。

此 payload 欄位為選用。

回應映射範本

由於沒有資料來源，payload 欄位的值將轉發到回應映射範本，並在可透過 VTL context 屬性使用的 `$context.result` 物件上設定。

如果您的 payload 欄位值形狀完全匹配 GraphQL 類型形狀時，您可以使用以下回應映射範本轉送回應：

```
$util.toJson($context.result)
```

沒有任何必要欄位或形狀限制適用於回應映射範本。不過，由於 GraphQL 是強類型，解析後的映射範本必須符合預期的 GraphQL 類型。

HTTP 的 AWS AppSync 解析程式映射範本參考

Note

我們現在主要支援 APPSYNC_JS 執行期及其文件。請考慮[在此處](#)使用 APPSYNC_JS 執行期及其指南。

AWS AppSync HTTP 解析程式映射範本可讓您從 AWS AppSync 將請求傳送至任何 HTTP 端點，以及從 HTTP 端點的回應傳回 AWS AppSync。透過使用映射範本，您可以向 AWS AppSync 提供有關要叫用之操作性質的提示。本節說明支援的 HTTP 解析程式的不同映射範本。

請求映射範本

```
{
```



```
"version": "2018-05-29",
"method": "PUT|POST|GET|DELETE|PATCH",
"params": {
  "query": Map,
  "headers": Map,
  "body": any
},
"resourcePath": string
}
```

在解析 HTTP 請求映射範本後，請求映射範本的 JSON 結構描述表示看起來會如下：

```
{
  "$id": "https://aws.amazon.com/appsync/request-mapping-template.json",
  "type": "object",
  "properties": {
    "version": {
      "$id": "/properties/version",
      "type": "string",
      "title": "The Version Schema ",
      "default": "",
      "examples": [
        "2018-05-29"
      ],
      "enum": [
        "2018-05-29"
      ]
    },
    "method": {
      "$id": "/properties/method",
      "type": "string",
      "title": "The Method Schema ",
      "default": "",
      "examples": [
        "PUT|POST|GET|DELETE|PATCH"
      ],
      "enum": [
        "PUT",
        "PATCH",
        "POST",
        "DELETE",
        "GET"
      ]
    }
  }
}
```

```

    },
    "params": {
      "$id": "/properties/params",
      "type": "object",
      "properties": {
        "query": {
          "$id": "/properties/params/properties/query",
          "type": "object"
        },
        "headers": {
          "$id": "/properties/params/properties/headers",
          "type": "object"
        },
        "body": {
          "$id": "/properties/params/properties/body",
          "type": "string",
          "title": "The Body Schema ",
          "default": "",
          "examples": [
            ""
          ]
        }
      }
    },
    "resourcePath": {
      "$id": "/properties/resourcePath",
      "type": "string",
      "title": "The Resourcepath Schema ",
      "default": "",
      "examples": [
        ""
      ]
    }
  ],
  "required": [
    "version",
    "method",
    "resourcePath"
  ]
}

```

以下是 HTTP POST 請求的範例，其中包括 text/plain 本文：

```
{
  "version": "2018-05-29",
  "method": "POST",
  "params": {
    "headers": {
      "Content-Type": "text/plain"
    },
    "body": "this is an example of text body"
  },
  "resourcePath": "/"
}
```

版本

Note

這僅適用於請求映射範本。

定義範本使用的版本。version 是所有請求映射範本的共用項目，並且是必要項目。

```
"version": "2018-05-29"
```

方法

Note

這僅適用於請求映射範本。

AWS AppSync 傳送至 HTTP 端點的 HTTP 方法或動詞 (GET、POST、PUT、PATCH 或 DELETE)。

```
"method": "PUT"
```

ResourcePath

Note

這僅適用於請求映射範本。

您要存取的資源路徑。除了 HTTP 資料來源中的端點之外，資源路徑還會形成 AWS AppSync 服務發出請求的 URL。

```
"resourcePath": "/v1/users"
```

評估映射範本時，此路徑會做為 HTTP 請求的一部分進行傳送，包括 HTTP 端點。例如，之前的範例可轉譯如下：

```
PUT <endpoint>/v1/users
```

參數欄位

Note

這僅適用於請求映射範本。

用來指定您的搜尋執行的動作，最常見的方式是在內文中設定查詢值。不過，有多項其他功能可設定，例如回應的格式。

標頭

標頭資訊，以金鑰值對形式。金鑰與值皆必須為字串。

例如：

```
"headers" : {  
  "Content-Type" : "application/json"  
}
```

目前支援的 Content-Type 標頭包括：

```
text/*
application/xml
application/json
application/soap+xml
application/x-amz-json-1.0
application/x-amz-json-1.1
application/vnd.api+json
application/x-ndjson
```

注意：您不能設定以下 HTTP 標頭：

```
HOST
CONNECTION
USER-AGENT
EXPECTATION
TRANSFER_ENCODING
CONTENT_LENGTH
```

query

金鑰值對，指定常見的選項，例如 JSON 回應的程式碼格式。金鑰與值皆必須為字串。以下範例顯示如何傳送 `?type=json` 查詢字串：

```
"query" : {
  "type" : "json"
}
```

本文

此本文包含您選擇設定的 HTTP 請求本文。除非內容類型指定字元集，否則請求本文一律是 UTF-8 編碼的字串。

```
"body": "body string"
```

回應

請見[此處](#)範例。

HTTPS AWS AppSync 端點的憑證授權機構 (CA) 由 認可

Note

Let's Encrypt 是透過 identrust 和 isrgrootx1 憑證接受。如果您使用 Let's Encrypt，則不需要採取任何動作。

目前，使用 HTTPS 時，HTTP 解析程式不支援自我簽署憑證。AWS AppSync 會在解析 HTTPS 的 SSL/TLS 憑證時辨識下列憑證授權單位：

中的已知根憑證 AWS AppSync

名稱	日期	SHA1 指紋
digicertassuredidr ootca	2018 年 4 月 21 日	05:63:B8:63:0D:62:D7:5A:BB:C8:AB:1E: 4B:DF:B5:A8:99:B2:4D:43
trustcenterclass2c aii	2018 年 4 月 21 日	AE:50:83:ED:7C:F4:5C:BC:8F:61:C6:21: FE:68:5D:79:42:21:15:6E
thawtepremiumserve rca	2018 年 4 月 21 日	E0:AB:05:94:20:72:54:93:05:60:62:02: 36:70:F7:CD:2E:FC:66:66
cia-crt-g3-02-ca	2016 年 11 月 23 日	96:4A:BB:A7:BD:DA:FC:97:34:C0:0A:2D: F0:05:98:F7:E6:C6:6F:09
swisssignplatinumg 2ca	2018 年 4 月 21 日	56:E0:FA:C0:3B:8F:18:23:55:18:E5:D3: 11:CA:E8:C2:43:31:AB:66
swisssignsilverg2c a	2018 年 4 月 21 日	9B:AA:E5:9F:56:EE:21:CB:43:5A:BE:25: 93:DF:A7:F0:40:D1:1D:CB
thawteserverca	2018 年 4 月 21 日	9F:AD:91:A6:CE:6A:C6:C5:00:47:C4:4E: C9:D4:A5:0D:92:D8:49:79
equifaxsecureebusi nessca1	2018 年 4 月 21 日	AE:E6:3D:70:E3:76:FB:C7:3A:EB:B0:A1: C1:D4:C4:7A:A7:40:B3:F4

名稱	日期	SHA1 指紋
securetrustca	2018 年 4 月 21 日	87:82:C6:C3:04:35:3B:CF:D2:96:92:D2:59:3E:7D:44:D9:34:FF:11
utnuserfirstclientauthemailca	2018 年 4 月 21 日	B1:72:B1:A5:6D:95:F9:1F:E5:02:87:E1:4D:37:EA:6A:44:63:76:8A
thawtepersonalfreeemailca	2018 年 4 月 21 日	E6:18:83:AE:84:CA:C1:C1:CD:52:AD:E8:E9:25:2B:45:A6:4F:B7:E2
affirmtrustnetworkingca	2018 年 4 月 21 日	29:36:21:02:8B:20:ED:02:F5:66:C5:32:D1:D6:ED:90:9F:45:00:2F
entrustevca	2018 年 4 月 21 日	B3:1E:B1:B7:40:E3:6C:84:02:DA:DC:37:D4:4D:F5:D4:67:49:52:F9
utnuserfirsthardwarereca	2018 年 4 月 21 日	04:83:ED:33:99:AC:36:08:05:87:22:ED:BC:5E:46:00:E3:BE:F9:D7
certumca	2018 年 4 月 21 日	62:52:DC:40:F7:11:43:A2:2F:DE:9E:F7:34:8E:06:42:51:B1:81:18
addtrustclass1ca	2018 年 4 月 21 日	CC:AB:0E:A0:4C:23:01:D6:69:7B:DD:37:9F:CD:12:EB:24:E3:94:9D
entrustrootcag2	2018 年 4 月 21 日	8C:F4:27:FD:79:0C:3A:D1:66:06:8D:E8:1E:57:EF:BB:93:22:72:D4
equifaxsecureca	2018 年 4 月 21 日	D2:32:09:AD:23:D3:14:23:21:74:E4:0D:7F:9D:62:13:97:86:63:3A
quovadisrootca3	2018 年 4 月 21 日	1F:49:14:F7:D8:74:95:1D:DD:AE:02:C0:BE:FD:3A:2D:82:75:51:85
quovadisrootca2	2018 年 4 月 21 日	CA:3A:FB:CF:12:40:36:4B:44:B2:16:20:88:80:48:39:19:93:7C:F7
digicertglobalrootg2	2018 年 4 月 21 日	DF:3C:24:F9:BF:D6:66:76:1B:26:80:73:FE:06:D1:CC:8D:4F:82:A4

名稱	日期	SHA1 指紋
digicerthighassurancееvrootca	2018 年 4 月 21 日	5F:B7:EE:06:33:E2:59:DB:AD:0C:4C:9A:E6:D3:8F:1A:61:C7:DC:25
secomvalicertclass1ca	2018 年 4 月 21 日	E5:DF:74:3C:B6:01:C4:9B:98:43:DC:AB:8C:E8:6A:81:10:9F:E4:8E
equifaxsecureglobalbusinessca1	2018 年 4 月 21 日	3A:74:CB:7A:47:DB:70:DE:89:1F:24:35:98:64:B8:2D:82:BD:1A:36
geotrustuniversalca	2018 年 4 月 21 日	E6:21:F3:35:43:79:05:9A:4B:68:30:9D:8A:2F:74:22:15:87:EC:79
deprecateditsec	2012 年 1 月 27 日	12:12:0B:03:0E:15:14:54:F4:DD:B3:F5:DE:13:6E:83:5A:29:72:9D
verisignclass3ca	2018 年 4 月 21 日	A1:DB:63:93:91:6F:17:E4:18:55:09:40:04:15:C7:02:40:B0:AE:6B
thawteprimaryrootca3	2018 年 4 月 21 日	F1:8B:53:8D:1B:E9:03:B6:A6:F0:56:43:5B:17:15:89:CA:F3:6B:F2
thawteprimaryrootca2	2018 年 4 月 21 日	AA:DB:BC:22:23:8F:C4:01:A1:27:BB:38:DD:F4:1D:DB:08:9E:F0:12
deutschetelekomrootca2	2018 年 4 月 21 日	85:A4:08:C0:9C:19:3E:5D:51:58:7D:CD:D6:13:30:FD:8C:DE:37:BF
buypassclass3ca	2018 年 4 月 21 日	DA:FA:F7:FA:66:84:EC:06:8F:14:50:BD:C7:C2:81:A5:BC:A9:64:57
utnuserfirstobjectca	2018 年 4 月 21 日	E1:2D:FB:4B:41:D7:D9:C3:2B:30:51:4B:AC:1D:81:D8:38:5E:2D:46
geotrustprimaryca	2018 年 4 月 21 日	32:3C:11:8E:1B:F7:B8:B6:52:54:E2:E2:10:0D:D6:02:90:37:F0:96
buypassclass2ca	2018 年 4 月 21 日	49:0A:75:74:DE:87:0A:47:FE:58:EE:F6:C7:6B:EB:C6:0B:12:40:99

名稱	日期	SHA1 指紋
baltimorecodesigningca	2018 年 4 月 21 日	30:46:D8:C8:88:FF:69:30:C3:4A:FC:CD:49:27:08:7C:60:56:7B:0D
verisignclass1ca	2018 年 4 月 21 日	CE:6A:64:A3:09:E4:2F:BB:D9:85:1C:45:3E:64:09:EA:E8:7D:60:F1
baltimorecybertrustca	2018 年 4 月 21 日	D4:DE:20:D0:5E:66:FC:53:FE:1A:50:88:2C:78:DB:28:52:CA:E4:74
starfieldclass2ca	2018 年 4 月 21 日	AD:7E:1C:28:B0:64:EF:8F:60:03:40:20:14:C3:D0:E3:37:0E:B5:8A
camerfirmachamberscommerceca	2018 年 4 月 21 日	6E:3A:55:A4:19:0C:19:5C:93:84:3C:C0:DB:72:2E:31:30:61:F0:B1
ttelesecglobalrootclass3ca	2018 年 4 月 21 日	55:A6:72:3E:CB:F2:EC:CD:C3:23:74:70:19:9D:2A:BE:11:E3:81:D1
verisignclass3g5ca	2018 年 4 月 21 日	4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD:56:BE:3D:9B:67:44:A5:E5
ttelesecglobalrootclass2ca	2018 年 4 月 21 日	59:0D:2D:7D:88:4F:40:2E:61:7E:A5:62:32:17:65:CF:17:D8:94:E9
trustcenteruniversalcai	2018 年 4 月 21 日	6B:2F:34:AD:89:58:BE:62:FD:B0:6B:5C:CE:BB:9D:D9:4F:4E:39:F3
verisignclass3g4ca	2018 年 4 月 21 日	22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:6C:3A
verisignclass3g3ca	2018 年 4 月 21 日	13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3:39:E2:55:76:60:9B:5C:C6
xrampglobalca	2018 年 4 月 21 日	B8:01:86:D1:EB:9C:86:A5:41:04:CF:30:54:F3:4C:52:B7:E5:58:C6
amzninternalrootca	2008 年 12 月 12 日	A7:B7:F6:15:8A:FF:1E:C8:85:13:38:BC:93:EB:A2:AB:A4:09:EF:06

名稱	日期	SHA1 指紋
certplusclass3ppri maryca	2018 年 4 月 21 日	21:6B:2A:29:E6:2A:00:CE:82:01:46:D8: 24:41:41:B9:25:11:B2:79
certumtrustednetwo rkca	2018 年 4 月 21 日	07:E0:32:E0:20:B7:2C:3F:19:2F:06:28: A2:59:3A:19:A7:0F:06:9E
verisignclass3g2ca	2018 年 4 月 21 日	85:37:1C:A6:E5:50:14:3D:CE:28:03:47: 1B:DE:3A:09:E8:F8:77:0F
globalsignr3ca	2018 年 4 月 21 日	D6:9B:56:11:48:F0:1C:77:C5:45:78:C1: 09:26:DF:5B:85:69:76:AD
utndatacorpsgccca	2018 年 4 月 21 日	58:11:9F:0E:12:82:87:EA:50:FD:D9:87: 45:6F:4F:78:DC:FA:D6:D4
secomscrootca2	2018 年 4 月 21 日	5F:3B:8C:F2:F8:10:B3:7D:78:B4:CE:EC: 19:19:C3:73:34:B9:C7:74
gtecybertrustgloba lca	2018 年 4 月 21 日	97:81:79:50:D8:1C:96:70:CC:34:D8:09: CF:79:44:31:36:7E:F4:74
secomscrootca1	2018 年 4 月 21 日	36:B1:2B:49:F9:81:9E:D7:4C:9E:BC:38: 0F:C6:56:8F:5D:AC:B2:F7
affirmtrustcommerc ialca	2018 年 4 月 21 日	F9:B5:B6:32:45:5F:9C:BE:EC:57:5F:80: DC:E9:6E:2C:C7:B2:78:B7
trustcenterclass4c aii	2018 年 4 月 21 日	A6:9A:91:FD:05:7F:13:6A:42:63:0B:B1: 76:0D:2D:51:12:0C:16:50
verisignuniversalr ootca	2018 年 4 月 21 日	36:79:CA:35:66:87:72:30:4D:30:A5:FB: 87:3B:0F:A7:7B:B7:0D:54
globalsignr2ca	2018 年 4 月 21 日	75:E0:AB:B6:13:85:12:27:1C:04:F8:5F: DD:DE:38:E4:B7:24:2E:FE
certplusclass2prim aryca	2018 年 4 月 21 日	74:20:74:41:72:9C:DD:92:EC:79:31:D8: 23:10:8D:C2:81:92:E2:BB

名稱	日期	SHA1 指紋
digicertglobalrootca	2018 年 4 月 21 日	A8:98:5D:3A:65:E5:E5:C4:B2:D7:D6:6D:40:C6:DD:2F:B1:9C:54:36
globalsignca	2018 年 4 月 21 日	B1:BC:96:8B:D4:F4:9D:62:2A:A8:9A:81:F2:15:01:52:A4:1D:82:9C
thawteprimaryrootca	2018 年 4 月 21 日	91:C6:D6:EE:3E:8A:C8:63:84:E5:48:C2:99:29:5C:75:6C:81:7B:81
starfieldrootg2ca	2018 年 4 月 21 日	B5:1C:06:7C:EE:2B:0C:3D:F8:55:AB:2D:92:F4:FE:39:D4:E7:0F:0E
geotrustglobalca	2018 年 4 月 21 日	DE:28:F4:A4:FF:E5:B9:2F:A3:C5:03:D1:A3:49:A7:F9:96:2A:82:12
soneraclass2ca	2018 年 4 月 21 日	37:F7:6D:E6:07:7C:90:C5:B1:3E:93:1A:B7:41:10:B4:F2:E4:9A:27
verisigntsaca	2018 年 4 月 21 日	20:CE:B1:F0:F5:1C:0E:19:A9:F3:8D:B1:AA:8E:03:8C:AA:7A:C7:01
soneraclass1ca	2018 年 4 月 21 日	07:47:22:01:99:CE:74:B9:7C:B0:3D:79:B2:64:A2:C8:55:E9:33:FF
quovadisrootca	2018 年 4 月 21 日	DE:3F:40:BD:50:93:D3:9B:6C:60:F6:DA:BC:07:62:01:00:89:76:C9
affirmtrustpremiumeccca	2018 年 4 月 21 日	B8:23:6B:00:2F:1D:16:86:53:01:55:6C:11:A4:37:CA:EB:FF:C3:BB
starfieldservicesrootg2ca	2018 年 4 月 21 日	92:5A:8F:8D:2C:6D:04:E0:66:5F:59:6A:FF:22:D8:63:E8:25:6F:3F
valicertclass2ca	2018 年 4 月 21 日	31:7A:2A:D0:7F:2B:33:5E:F5:A1:C3:4E:4B:57:E8:B7:D8:F1:FC:A6
comodoaaaca	2018 年 4 月 21 日	D1:EB:23:A4:6D:17:D6:8F:D9:25:64:C2:F1:F1:60:17:64:D8:E3:49

名稱	日期	SHA1 指紋
aolrootca2	2018 年 4 月 21 日	85:B5:FF:67:9B:0C:79:96:1F:C8:6E:44:22:00:46:13:DB:17:92:84
keynectisrootca	2018 年 4 月 21 日	9C:61:5C:4D:4D:85:10:3A:53:26:C2:4D:BA:EA:E4:A2:D2:D5:CC:97
addtrustqualifiedca	2018 年 4 月 21 日	4D:23:78:EC:91:95:39:B5:00:7F:75:8F:03:3B:21:1E:C5:4D:8B:CF
aolrootca1	2018 年 4 月 21 日	39:21:C1:15:C1:5D:0E:CA:5C:CB:5B:C4:F0:7D:21:D8:05:0B:56:6A
verisignclass2g3ca	2018 年 4 月 21 日	61:EF:43:D7:7F:CA:D4:61:51:BC:98:E0:C3:59:12:AF:9F:EB:63:11
addtrustexternalca	2018 年 4 月 21 日	02:FA:F3:E2:91:43:54:68:60:78:57:69:4D:F5:E4:5B:68:85:18:68
verisignclass2g2ca	2018 年 4 月 21 日	B3:EA:C4:47:76:C9:C8:1C:EA:F2:9D:95:B6:CC:A0:08:1B:67:EC:9D
geotrustprimarycag3	2018 年 4 月 21 日	03:9E:ED:B8:0B:E7:A0:3C:69:53:89:3B:20:D2:D9:32:3A:4C:2A:FD
geotrustprimarycag2	2018 年 4 月 21 日	8D:17:84:D5:37:F3:03:7D:EC:70:FE:57:8B:51:9A:99:E6:10:D7:B0
swisssigngoldg2ca	2018 年 4 月 21 日	D8:C5:38:8A:B7:30:1B:1B:6E:D4:7A:E6:45:25:3A:6F:9F:1A:27:61
entrust2048ca	2018 年 4 月 21 日	50:30:06:09:1D:97:D4:F5:AE:39:F7:CB:E7:92:7D:7D:65:2D:34:31
chunghwaepkirootca	2018 年 4 月 21 日	67:65:0D:F1:7E:8E:7E:5B:82:40:A4:F4:56:4B:CF:E2:3D:69:C6:F0
camerfirmachambersignca	2018 年 4 月 21 日	4A:BD:EE:EC:95:0D:35:9C:89:AE:C7:52:A1:2C:5B:29:F6:D6:AA:0C

名稱	日期	SHA1 指紋
camerfirmachambersca	2018 年 4 月 21 日	78:6A:74:AC:76:AB:14:7F:9C:6A:30:50:BA:9E:A8:7E:FE:9A:CE:3C
godaddyclass2ca	2018 年 4 月 21 日	27:96:BA:E6:3F:18:01:E2:77:26:1B:A0:D7:77:70:02:8F:20:EE:E4
affirmtrustpremiumca	2018 年 4 月 21 日	D8:A6:33:2C:E0:03:6F:B1:85:F6:63:4F:7D:6A:06:65:26:32:28:27
verisignclass1g3ca	2018 年 4 月 21 日	20:42:85:DC:F7:EB:76:41:95:57:8E:13:6B:D4:B7:D1:E9:8E:46:A5
secomevrootca1	2018 年 4 月 21 日	FE:B8:C4:32:DC:F9:76:9A:CE:AE:3D:D8:90:8F:FD:28:86:65:64:7D
verisignclass1g2ca	2018 年 4 月 21 日	27:3E:E1:24:57:FD:C4:F9:0C:55:E8:2B:56:16:7F:62:F5:32:E5:47
amzninternalinfosecag3	2015 年 2 月 27 日	B9:B1:CA:38:F7:BF:9C:D2:D4:95:E7:B6:5E:75:32:9B:A8:78:2E:F6
cia-crt-g3-01-ca	2016 年 11 月 23 日	2B:EE:2C:BA:A3:1D:B5:FE:60:40:41:95:08:ED:46:82:39:4D:ED:E2
godaddyrootg2ca	2018 年 4 月 21 日	47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:C2:54:FD:E6:8B
digicertassuredidrrootca	2018 年 4 月 21 日	05:63:B8:63:0D:62:D7:5A:BB:C8:AB:1E:4B:DF:B5:A8:99:B2:4D:43
microseceszignorootca2009	2018 年 4 月 21 日	89:DF:74:FE:5C:F4:0F:4A:80:F9:E3:37:7D:54:DA:91:E1:01:31:8E
affirmtrustcommercial	2018 年 4 月 21 日	F9:B5:B6:32:45:5F:9C:BE:EC:57:5F:80:DC:E9:6E:2C:C7:B2:78:B7
comodoecccertificationauthority	2018 年 4 月 21 日	9F:74:4E:9F:2B:4D:BA:EC:0F:31:2C:50:B6:56:3B:8E:2D:93:C3:11

名稱	日期	SHA1 指紋
cadisigrootr2	2018 年 4 月 21 日	B5:61:EB:EA:A4:DE:E4:25:4B:69:1A:98:A5:57:47:C2:34:C7:D9:71
swisssignsilvercag2	2018 年 4 月 21 日	9B:AA:E5:9F:56:EE:21:CB:43:5A:BE:25:93:DF:A7:F0:40:D1:1D:CB
securetrustca	2018 年 4 月 21 日	87:82:C6:C3:04:35:3B:CF:D2:96:92:D2:59:3E:7D:44:D9:34:FF:11
cadisigrootr1	2018 年 4 月 21 日	8E:1C:74:F8:A6:20:B9:E5:8A:F4:61:FA:EC:2B:47:56:51:1A:52:C6
accvraiz1	2018 年 4 月 21 日	93:05:7A:88:15:C6:4F:CE:88:2F:FA:91:16:52:28:78:BC:53:64:17
entrustrootcertificationauthority	2018 年 4 月 21 日	B3:1E:B1:B7:40:E3:6C:84:02:DA:DC:37:D4:4D:F5:D4:67:49:52:F9
camerfirmaglobalchambersignroot	2018 年 4 月 21 日	33:9B:6B:14:50:24:9B:55:7A:01:87:72:84:D9:E0:2F:C3:D2:D8:E9
dstacescax6	2018 年 4 月 21 日	40:54:DA:6F:1C:3F:40:74:AC:ED:0F:EC:CD:DB:79:D1:53:FB:90:1D
identrustpublicsectorrootca1	2018 年 4 月 21 日	BA:29:41:60:77:98:3F:F4:F3:EF:F2:31:05:3B:2E:EA:6D:4D:45:FD
starfieldrootcertificateauthorityg2	2018 年 4 月 21 日	B5:1C:06:7C:EE:2B:0C:3D:F8:55:AB:2D:92:F4:FE:39:D4:E7:0F:0E
secureglobalca	2018 年 4 月 21 日	3A:44:73:5A:E5:81:90:1F:24:86:61:46:1E:3B:9C:C4:5F:F5:3A:1B
eecertificationcenterrootca	2018 年 4 月 21 日	C9:A8:B9:E7:55:80:5E:58:E3:53:77:A7:25:EB:AF:C3:7B:27:CC:D7
opentrustrootcag3	2018 年 4 月 21 日	6E:26:64:F3:56:BF:34:55:BF:D1:93:3F:7C:01:DE:D8:13:DA:8A:A6

名稱	日期	SHA1 指紋
teliasonerarootcav1	2018 年 4 月 21 日	43:13:BB:96:F1:D5:86:9B:C1:4E:6A:92:F6:CF:F6:34:69:87:82:37
autoridaddecertificacionfirmaprofesionalcifa62634068	2018 年 4 月 21 日	AE:C5:FB:3F:C8:E1:BF:C4:E5:4F:03:07:5A:9A:E8:00:B7:F7:B6:FA
opentrustrootcag2	2018 年 4 月 21 日	79:5F:88:60:C5:AB:7C:3D:92:E6:CB:F4:8D:E1:45:CD:11:EF:60:0B
opentrustrootcag1	2018 年 4 月 21 日	79:91:E8:34:F7:E2:EE:DD:08:95:01:52:E9:55:2D:14:E9:58:D5:7E
globalsigneccrootcar5	2018 年 4 月 21 日	1F:24:C6:30:CD:A4:18:EF:20:69:FF:AD:4F:DD:5F:46:3A:1B:69:AA
globalsigneccrootcar4	2018 年 4 月 21 日	69:69:56:2E:40:80:F4:24:A1:E7:19:9F:14:BA:F3:EE:58:AB:6A:BB
izenpecom	2018 年 4 月 21 日	2F:78:3D:25:52:18:A7:4A:65:39:71:B5:2C:A2:9C:45:15:6F:E9:19
turktrustelektroniksertifikahizmetseglayicisih5	2018 年 4 月 21 日	C4:18:F6:4D:46:D1:DF:00:3D:27:30:13:72:43:A9:12:11:C6:75:FB
gdcatrustauthr5root	2018 年 4 月 21 日	0F:36:38:5B:81:1A:25:C3:9B:31:4E:83:CA:E9:34:66:70:CC:74:B4
dtrustrootclass3ca22009	2018 年 4 月 21 日	58:E8:AB:B0:36:15:33:FB:80:F7:9B:1B:6D:29:D3:FF:8D:5F:00:F0
quovadisrootca3	2018 年 4 月 21 日	1F:49:14:F7:D8:74:95:1D:DD:AE:02:C0:BE:FD:3A:2D:82:75:51:85

名稱	日期	SHA1 指紋
quovadisrootca2	2018 年 4 月 21 日	CA:3A:FB:CF:12:40:36:4B:44:B2:16:20:88:80:48:39:19:93:7C:F7
geotrustprimarycertificatio nauthorityg3	2018 年 4 月 21 日	03:9E:ED:B8:0B:E7:A0:3C:69:53:89:3B:20:D2:D9:32:3A:4C:2A:FD
geotrustprimarycertificatio nauthorityg2	2018 年 4 月 21 日	8D:17:84:D5:37:F3:03:7D:EC:70:FE:57:8B:51:9A:99:E6:10:D7:B0
oistewisekeyglobal rootgbca	2018 年 4 月 21 日	0F:F9:40:76:18:D3:D7:6A:4B:98:F0:A8:35:9E:0C:FD:27:AC:CC:ED
addtrustexternalro ot	2018 年 4 月 21 日	02:FA:F3:E2:91:43:54:68:60:78:57:69:4D:F5:E4:5B:68:85:18:68
chambersofcommerce root2008	2018 年 4 月 21 日	78:6A:74:AC:76:AB:14:7F:9C:6A:30:50:BA:9E:A8:7E:FE:9A:CE:3C
digicertglobalroot g3	2018 年 4 月 21 日	7E:04:DE:89:6A:3E:66:6D:00:E6:87:D3:3F:FA:D9:3B:E8:3D:34:9E
comodoaaaservicesr oot	2018 年 4 月 21 日	D1:EB:23:A4:6D:17:D6:8F:D9:25:64:C2:F1:F1:60:17:64:D8:E3:49
digicertglobalroot g2	2018 年 4 月 21 日	DF:3C:24:F9:BF:D6:66:76:1B:26:80:73:FE:06:D1:CC:8D:4F:82:A4
certinomisrootca	2018 年 4 月 21 日	9D:70:BB:01:A5:A4:A0:18:11:2E:F7:1C:01:B9:32:C5:34:E7:88:A8
oistewisekeyglobal rootgaca	2018 年 4 月 21 日	59:22:A1:E1:5A:EA:16:35:21:F8:98:39:6A:46:46:B0:44:1B:0F:A9
dstrootcax3	2018 年 4 月 21 日	DA:C9:02:4F:54:D8:F6:DF:94:93:5F:B1:73:26:38:CA:6A:D7:7C:13

名稱	日期	SHA1 指紋
certigna	2018 年 4 月 21 日	B1:2E:13:63:45:86:A4:6F:1A:B2:60:68:37:58:2D:C4:AC:FD:94:97
digicerthighassuranc eevrootca	2018 年 4 月 21 日	5F:B7:EE:06:33:E2:59:DB:AD:0C:4C:9A:E6:D3:8F:1A:61:C7:DC:25
soneraclass2rootca	2018 年 4 月 21 日	37:F7:6D:E6:07:7C:90:C5:B1:3E:93:1A:B7:41:10:B4:F2:E4:9A:27
trustcorrootcertca 2	2018 年 4 月 21 日	B8:BE:6D:CB:56:F1:55:B9:63:D4:12:CA:4E:06:34:C7:94:B2:1C:C0
usertrustrsacertif icationauthority	2018 年 4 月 21 日	2B:8F:1B:57:33:0D:BB:A2:D0:7A:6C:51:F7:0E:E9:0D:DA:B9:AD:8E
trustcorrootcertca 1	2018 年 4 月 21 日	FF:BD:CD:E7:82:C8:43:5E:3C:6F:26:86:5C:CA:A8:3A:45:5B:C3:0A
geotrustuniversalc a	2018 年 4 月 21 日	E6:21:F3:35:43:79:05:9A:4B:68:30:9D:8A:2F:74:22:15:87:EC:79
certsignrootca	2018 年 4 月 21 日	FA:B7:EE:36:97:26:62:FB:2D:B0:2A:F6:BF:03:FD:E8:7C:4B:2F:9B
amazonrootca4	2018 年 4 月 21 日	F6:10:84:07:D6:F8:BB:67:98:0C:C2:E2:44:C2:EB:AE:1C:EF:63:BE
amazonrootca3	2018 年 4 月 21 日	0D:44:DD:8C:3C:8C:1A:1A:58:75:64:81:E9:0F:2E:2A:FF:B3:D2:6E
amazonrootca2	2018 年 4 月 21 日	5A:8C:EF:45:D7:A6:98:59:76:7A:8C:8B:44:96:B5:78:CF:47:4B:1A
verisignuniversalr ootcertif icationauthority	2018 年 4 月 21 日	36:79:CA:35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0D:54

名稱	日期	SHA1 指紋
amazonrootca1	2018 年 4 月 21 日	8D:A7:F9:65:EC:5E:FC:37:91:0F:1C:6E:59:FD:C1:CC:6A:6E:DE:16
networksolutionscertificatauthority	2018 年 4 月 21 日	74:F8:A3:C3:EF:E7:B3:90:06:4B:83:90:3C:21:64:60:20:E5:DF:CE
thawteprimaryrootca3	2018 年 4 月 21 日	F1:8B:53:8D:1B:E9:03:B6:A6:F0:56:43:5B:17:15:89:CA:F3:6B:F2
affirmtrustnetworking	2018 年 4 月 21 日	29:36:21:02:8B:20:ED:02:F5:66:C5:32:D1:D6:ED:90:9F:45:00:2F
thawteprimaryrootca2	2018 年 4 月 21 日	AA:DB:BC:22:23:8F:C4:01:A1:27:BB:38:DD:F4:1D:DB:08:9E:F0:12
trustcoreca1	2018 年 4 月 21 日	58:D1:DF:95:95:67:6B:63:C0:F0:5B:1C:17:4D:8B:84:0B:C8:78:BD
deutschetelekomrootca2	2018 年 4 月 21 日	85:A4:08:C0:9C:19:3E:5D:51:58:7D:CD:D6:13:30:FD:8C:DE:37:BF
godaddyrootcertificateauthorityg2	2018 年 4 月 21 日	47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:C2:54:FD:E6:8B
entrustrootcertificateauthorityec1	2018 年 4 月 21 日	20:D8:06:40:DF:9B:25:F5:12:25:3A:11:EA:F7:59:8A:EB:14:B5:47
szafirrootca2	2018 年 4 月 21 日	E2:52:FA:95:3F:ED:DB:24:60:BD:6E:28:F3:9C:CC:CF:5E:B3:3F:DE
tubitakkamusmsslkoksertifikaasisurum1	2018 年 4 月 21 日	31:43:64:9B:EC:CE:27:EC:ED:3A:3F:0B:8F:0D:E4:E8:91:DD:EE:CA

名稱	日期	SHA1 指紋
buypassclass3rootca	2018 年 4 月 21 日	DA:FA:F7:FA:66:84:EC:06:8F:14:50:BD:C7:C2:81:A5:BC:A9:64:57
comodorsacertificationauthority	2018 年 4 月 21 日	AF:E5:D2:44:A8:D1:19:42:30:FF:47:9F:E2:F8:97:BB:CD:7A:8C:B4
netlockaranyclassg oldfotanusitvany	2018 年 4 月 21 日	06:08:3F:59:3F:15:A1:04:A0:69:A4:6B:A9:03:D0:06:B7:97:09:91
securitycommunicationrootca2	2018 年 4 月 21 日	5F:3B:8C:F2:F8:10:B3:7D:78:B4:CE:EC:19:19:C3:73:34:B9:C7:74
dtrustrootclass3ca 2ev2009	2018 年 4 月 21 日	96:C9:1B:0B:95:B4:10:98:42:FA:D0:D8:22:79:FE:60:FA:B9:16:83
starfieldclass2ca	2018 年 4 月 21 日	AD:7E:1C:28:B0:64:EF:8F:60:03:40:20:14:C3:D0:E3:37:0E:B5:8A
pscprocert	2018 年 4 月 21 日	70:C1:8D:74:B4:28:81:0A:E4:FD:A5:75:D7:01:9F:99:B0:3D:50:74
actalisauthenticationrootca	2018 年 4 月 21 日	F3:73:B3:87:06:5A:28:84:8A:F2:F3:4A:CE:19:2B:DD:C7:8E:9C:AC
staatdernederlandenrootcag3	2018 年 4 月 21 日	D8:EB:6B:41:51:92:59:E0:F3:E7:85:00:C0:3D:B6:88:97:C9:EE:FC
cfcae/root	2018 年 4 月 21 日	E2:B8:29:4B:55:84:AB:6B:58:C2:90:46:6C:AC:3F:B8:39:8F:84:83
digicerttrustedrootg4	2018 年 4 月 21 日	DD:FB:16:CD:49:31:C9:73:A2:03:7D:3F:C8:3A:4D:7D:77:5D:05:E4
staatdernederlandenrootcag2	2018 年 4 月 21 日	59:AF:82:79:91:86:C7:B4:75:07:CB:CF:03:57:46:EB:04:DD:B7:16
securitycommunicationevrootca1	2018 年 4 月 21 日	FE:B8:C4:32:DC:F9:76:9A:CE:AE:3D:D8:90:8F:FD:28:86:65:64:7D

名稱	日期	SHA1 指紋
globalsignrootcar3	2018 年 4 月 21 日	D6:9B:56:11:48:F0:1C:77:C5:45:78:C1:09:26:DF:5B:85:69:76:AD
globalsignrootcar2	2018 年 4 月 21 日	75:E0:AB:B6:13:85:12:27:1C:04:F8:5F:DD:DE:38:E4:B7:24:2E:FE
certumtrustednetwo rkca2	2018 年 4 月 21 日	D3:DD:48:3E:2B:BF:4C:05:E8:AF:10:F5:FA:76:26:CF:D3:DC:30:92
acraizfnmtrcm	2018 年 4 月 21 日	EC:50:35:07:B2:15:C4:95:62:19:E2:A8:9A:5B:42:99:2C:4C:2C:20
hellenicacademican dresearch instituti onseccrootca2015	2018 年 4 月 21 日	9F:F1:71:8D:92:D5:9A:F3:7D:74:97:B4:BC:6F:84:68:0B:BA:B6:66
certplusrootcag2	2018 年 4 月 21 日	4F:65:8E:1F:E9:06:D8:28:02:E9:54:47:41:C9:54:25:5D:69:CC:1A
twcarootcertificat ionauthority	2018 年 4 月 21 日	CF:9E:87:6D:D3:EB:FC:42:26:97:A3:B5:A3:7A:A0:76:A9:06:23:48
twcaglobalrootca	2018 年 4 月 21 日	9C:BB:48:53:F6:A4:F6:D3:52:A4:E8:32:52:55:60:13:F5:AD:AF:65
certplusrootcag1	2018 年 4 月 21 日	22:FD:D0:B7:FD:A2:4E:0D:AC:49:2C:A0:AC:A6:7B:6A:1F:E3:F7:66
geotrustuniversalc a2	2018 年 4 月 21 日	37:9A:19:7B:41:85:45:35:0C:A6:03:69:F3:3C:2E:AF:47:4F:20:79
baltimorecybertrus troot	2018 年 4 月 21 日	D4:DE:20:D0:5E:66:FC:53:FE:1A:50:88:2C:78:DB:28:52:CA:E4:74
buypassclass2rootc a	2018 年 4 月 21 日	49:0A:75:74:DE:87:0A:47:FE:58:EE:F6:C7:6B:EB:C6:0B:12:40:99

名稱	日期	SHA1 指紋
certumtrustednetworkca	2018 年 4 月 21 日	07:E0:32:E0:20:B7:2C:3F:19:2F:06:28:A2:59:3A:19:A7:0F:06:9E
digicertassuredidrootg3	2018 年 4 月 21 日	F5:17:A2:4F:9A:48:C6:C9:F8:A2:00:26:9F:DC:0F:48:2C:AB:30:89
digicertassuredidrootg2	2018 年 4 月 21 日	A1:4B:48:D9:43:EE:0A:0E:40:90:4F:3C:E0:A4:C0:91:93:51:5D:3F
isrgrootx1	2018 年 4 月 21 日	CA:BD:2A:79:A1:07:6A:31:F2:1D:25:36:35:CB:03:9D:43:29:A5:E8
entrustnetpremium2048secureserverca	2018 年 4 月 21 日	50:30:06:09:1D:97:D4:F5:AE:39:F7:CB:E7:92:7D:7D:65:2D:34:31
certplusclass2primaryca	2018 年 4 月 21 日	74:20:74:41:72:9C:DD:92:EC:79:31:D8:23:10:8D:C2:81:92:E2:BB
digicertglobalrootca	2018 年 4 月 21 日	A8:98:5D:3A:65:E5:E5:C4:B2:D7:D6:6D:40:C6:DD:2F:B1:9C:54:36
entrustrootcertificationauthorityg2	2018 年 4 月 21 日	8C:F4:27:FD:79:0C:3A:D1:66:06:8D:E8:1E:57:EF:BB:93:22:72:D4
starfieldservicesrootcertificationauthorityg2	2018 年 4 月 21 日	92:5A:8F:8D:2C:6D:04:E0:66:5F:59:6A:FF:22:D8:63:E8:25:6F:3F
thawteprimaryrootca	2018 年 4 月 21 日	91:C6:D6:EE:3E:8A:C8:63:84:E5:48:C2:99:29:5C:75:6C:81:7B:81
atostrustedroot2011	2018 年 4 月 21 日	2B:B1:F5:3E:55:0C:1D:C5:F1:D4:E6:B7:6A:46:4B:55:06:02:AC:21
geotrustglobalca	2018 年 4 月 21 日	DE:28:F4:A4:FF:E5:B9:2F:A3:C5:03:D1:A3:49:A7:F9:96:2A:82:12

名稱	日期	SHA1 指紋
luxtrustglobalroot2	2018 年 4 月 21 日	1E:0E:56:19:0A:D1:8B:25:98:B2:04:44:FF:66:8A:04:17:99:5F:3F
etugracertificatio nauthority	2018 年 4 月 21 日	51:C6:E7:08:49:06:6E:F3:92:D4:5C:A0:0D:6D:A3:62:8F:C3:52:39
visaecommerceroor	2018 年 4 月 21 日	70:17:9B:86:8C:00:A4:FA:60:91:52:22:3F:9F:3E:32:BD:E0:05:62
quovadisrootca	2018 年 4 月 21 日	DE:3F:40:BD:50:93:D3:9B:6C:60:F6:DA:BC:07:62:01:00:89:76:C9
identrustcommercia lrootca1	2018 年 4 月 21 日	DF:71:7E:AA:4A:D9:4E:C9:55:84:99:60:2D:48:DE:5F:BC:F0:3A:25
staatdernederlande nevrootca	2018 年 4 月 21 日	76:E2:7E:C1:4F:DB:82:C1:C0:A6:75:B5:05:BE:3D:29:B4:ED:DB:BB
ttelesecglobalroot class3	2018 年 4 月 21 日	55:A6:72:3E:CB:F2:EC:CD:C3:23:74:70:19:9D:2A:BE:11:E3:81:D1
ttelesecglobalroot class2	2018 年 4 月 21 日	59:0D:2D:7D:88:4F:40:2E:61:7E:A5:62:32:17:65:CF:17:D8:94:E9
comodocertificatio nauthority	2018 年 4 月 21 日	66:31:BF:9E:F7:4F:9E:B6:C9:D5:A6:0C:BA:6A:BE:D1:F7:BD:EF:7B
securitycommunicat ionrootca	2018 年 4 月 21 日	36:B1:2B:49:F9:81:9E:D7:4C:9E:BC:38:0F:C6:56:8F:5D:AC:B2:F7
quovadisrootca3g3	2018 年 4 月 21 日	48:12:BD:92:3C:A8:C4:39:06:E7:30:6D:27:96:E6:A4:CF:22:2E:7D
xrampglobalcaroot	2018 年 4 月 21 日	B8:01:86:D1:EB:9C:86:A5:41:04:CF:30:54:F3:4C:52:B7:E5:58:C6
securesignrootca11	2018 年 4 月 21 日	3B:C4:9F:48:F8:F3:73:A0:9C:1E:BD:F8:5B:B1:C3:65:C7:D8:11:B3

名稱	日期	SHA1 指紋
affirmtrustpremium	2018 年 4 月 21 日	D8:A6:33:2C:E0:03:6F:B1:85:F6:63:4F:7D:6A:06:65:26:32:28:27
globalsignrootca	2018 年 4 月 21 日	B1:BC:96:8B:D4:F4:9D:62:2A:A8:9A:81:F2:15:01:52:A4:1D:82:9C
swisssigngoldcag2	2018 年 4 月 21 日	D8:C5:38:8A:B7:30:1B:1B:6E:D4:7A:E6:45:25:3A:6F:9F:1A:27:61
quovadisrootca2g3	2018 年 4 月 21 日	09:3C:61:F3:8B:8B:DC:7D:55:DF:75:38:02:05:00:E1:25:F5:C8:36
affirmtrustpremium ecc	2018 年 4 月 21 日	B8:23:6B:00:2F:1D:16:86:53:01:55:6C:11:A4:37:CA:EB:FF:C3:BB
geotrustprimarycer tification authority	2018 年 4 月 21 日	32:3C:11:8E:1B:F7:B8:B6:52:54:E2:E2:10:0D:D6:02:90:37:F0:96
quovadisrootca1g3	2018 年 4 月 21 日	1B:8E:EA:57:96:29:1A:C9:39:EA:B8:0A:81:1A:73:73:C0:93:79:67
hongkongpostrootca 1	2018 年 4 月 21 日	D6:DA:A8:20:8D:09:D2:15:4D:24:B5:2F:CB:34:6E:B2:58:B2:8A:58
usertrustecccertif icationauthority	2018 年 4 月 21 日	D1:CB:CA:5D:B2:D5:2A:7F:69:3B:67:4D:E5:F0:5A:1D:0C:95:7D:F0
cybertrustglobalro ot	2018 年 4 月 21 日	5F:43:E5:B1:BF:F8:78:8C:AC:1C:C7:CA:4A:9A:C6:22:2B:CC:34:C6
godaddyclass2ca	2018 年 4 月 21 日	27:96:BA:E6:3F:18:01:E2:77:26:1B:A0:D7:77:70:02:8F:20:EE:E4

名稱	日期	SHA1 指紋
hellenicacademican dresearch instituti onsrootca2015	2018 年 4 月 21 日	01:0C:06:95:A6:98:19:14:FF:BF:5F:C6: B0:B6:95:EA:29:E9:12:A6
ecacc	2018 年 4 月 21 日	28:90:3A:63:5B:52:80:FA:E6:77:4C:0B: 6D:A7:D6:BA:A6:4A:F2:E8
hellenicacademican dresearch instituti onsrootca2011	2018 年 4 月 21 日	FE:45:65:9B:79:03:5B:98:A1:61:B5:51: 2E:AC:DA:58:09:48:22:4D
verisignclass3publ icprimary certifica tionauthorityg5	2018 年 4 月 21 日	4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD: 56:BE:3D:9B:67:44:A5:E5
verisignclass3publ icprimary certifica tionauthorityg4	2018 年 4 月 21 日	22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7: CF:8A:2D:64:C9:3F:6C:3A
verisignclass3publ icprimary certifica tionauthorityg3	2018 年 4 月 21 日	13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3: 39:E2:55:76:60:9B:5C:C6
trustisfpsrootca	2018 年 4 月 21 日	3B:C0:38:0B:33:C3:F6:A6:0C:86:15:22: 93:D9:DF:F5:4B:81:C0:04
epkirootcertificat ionauthority	2018 年 4 月 21 日	67:65:0D:F1:7E:8E:7E:5B:82:40:A4:F4: 56:4B:CF:E2:3D:69:C6:F0
globalchambersignr oot2008	2018 年 4 月 21 日	4A:BD:EE:EC:95:0D:35:9C:89:AE:C7:52: A1:2C:5B:29:F6:D6:AA:0C

名稱	日期	SHA1 指紋
camerfirmachambers ofcommercerooot	2018 年 4 月 21 日	6E:3A:55:A4:19:0C:19:5C:93:84:3C:C0: DB:72:2E:31:30:61:F0:B1
mozillacert81.pem	2014 年 3 月 13 日	07:E0:32:E0:20:B7:2C:3F:19:2F:06:28: A2:59:3A:19:A7:0F:06:9E
mozillacert99.pem	2014 年 3 月 13 日	F1:7F:6F:B6:31:DC:99:E3:A3:C8:7F:FE: 1C:F1:81:10:88:D9:60:33
mozillacert145.pem	2014 年 3 月 13 日	10:1D:FA:3F:D5:0B:CB:BB:9B:B5:60:0C: 19:55:A4:1A:F4:73:3A:04
mozillacert37.pem	2014 年 3 月 13 日	B1:2E:13:63:45:86:A4:6F:1A:B2:60:68: 37:58:2D:C4:AC:FD:94:97
mozillacert4.pem	2014 年 3 月 13 日	E3:92:51:2F:0A:CF:F5:05:DF:F6:DE:06: 7F:75:37:E1:65:EA:57:4B
mozillacert70.pem	2014 年 3 月 13 日	78:6A:74:AC:76:AB:14:7F:9C:6A:30:50: BA:9E:A8:7E:FE:9A:CE:3C
mozillacert88.pem	2014 年 3 月 13 日	FE:45:65:9B:79:03:5B:98:A1:61:B5:51: 2E:AC:DA:58:09:48:22:4D
mozillacert134.pem	2014 年 3 月 13 日	70:17:9B:86:8C:00:A4:FA:60:91:52:22: 3F:9F:3E:32:BD:E0:05:62
mozillacert26.pem	2014 年 3 月 13 日	87:82:C6:C3:04:35:3B:CF:D2:96:92:D2: 59:3E:7D:44:D9:34:FF:11
mozillacert77.pem	2014 年 3 月 13 日	13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3: 39:E2:55:76:60:9B:5C:C6
mozillacert123.pem	2014 年 3 月 13 日	2A:B6:28:48:5E:78:FB:F3:AD:9E:79:10: DD:6B:DF:99:72:2C:96:E5
mozillacert15.pem	2014 年 3 月 13 日	74:20:74:41:72:9C:DD:92:EC:79:31:D8: 23:10:8D:C2:81:92:E2:BB

名稱	日期	SHA1 指紋
mozillacert66.pem	2014 年 3 月 13 日	DD:E1:D2:A9:01:80:2E:1D:87:5E:84:B3: 80:7E:4B:B1:FD:99:41:34
mozillacert112.pem	2014 年 3 月 13 日	43:13:BB:96:F1:D5:86:9B:C1:4E:6A:92: F6:CF:F6:34:69:87:82:37
mozillacert55.pem	2014 年 3 月 13 日	AA:DB:BC:22:23:8F:C4:01:A1:27:BB:38: DD:F4:1D:DB:08:9E:F0:12
mozillacert101.pem	2014 年 3 月 13 日	99:A6:9B:E6:1A:FE:88:6B:4D:2B:82:00: 7C:B8:54:FC:31:7E:15:39
mozillacert119.pem	2014 年 3 月 13 日	75:E0:AB:B6:13:85:12:27:1C:04:F8:5F: DD:DE:38:E4:B7:24:2E:FE
mozillacert44.pem	2014 年 3 月 13 日	5F:43:E5:B1:BF:F8:78:8C:AC:1C:C7:CA: 4A:9A:C6:22:2B:CC:34:C6
mozillacert108.pem	2014 年 3 月 13 日	B1:BC:96:8B:D4:F4:9D:62:2A:A8:9A:81: F2:15:01:52:A4:1D:82:9C
mozillacert95.pem	2014 年 3 月 13 日	DA:FA:F7:FA:66:84:EC:06:8F:14:50:BD: C7:C2:81:A5:BC:A9:64:57
mozillacert141.pem	2014 年 3 月 13 日	31:7A:2A:D0:7F:2B:33:5E:F5:A1:C3:4E: 4B:57:E8:B7:D8:F1:FC:A6
mozillacert33.pem	2014 年 3 月 13 日	FE:B8:C4:32:DC:F9:76:9A:CE:AE:3D:D8: 90:8F:FD:28:86:65:64:7D
mozillacert0.pem	2014 年 3 月 13 日	97:81:79:50:D8:1C:96:70:CC:34:D8:09: CF:79:44:31:36:7E:F4:74
mozillacert84.pem	2014 年 3 月 13 日	D3:C0:63:F2:19:ED:07:3E:34:AD:5D:75: 0B:32:76:29:FF:D5:9A:F2
mozillacert130.pem	2014 年 3 月 13 日	E5:DF:74:3C:B6:01:C4:9B:98:43:DC:AB: 8C:E8:6A:81:10:9F:E4:8E

名稱	日期	SHA1 指紋
mozillacert148.pem	2014 年 3 月 13 日	04:83:ED:33:99:AC:36:08:05:87:22:ED: BC:5E:46:00:E3:BE:F9:D7
mozillacert22.pem	2014 年 3 月 13 日	32:3C:11:8E:1B:F7:B8:B6:52:54:E2:E2: 10:0D:D6:02:90:37:F0:96
mozillacert7.pem	2014 年 3 月 13 日	AD:7E:1C:28:B0:64:EF:8F:60:03:40:20: 14:C3:D0:E3:37:0E:B5:8A
mozillacert73.pem	2014 年 3 月 13 日	B5:1C:06:7C:EE:2B:0C:3D:F8:55:AB:2D: 92:F4:FE:39:D4:E7:0F:0E
mozillacert137.pem	2014 年 3 月 13 日	4A:65:D5:F4:1D:EF:39:B8:B8:90:4A:4A: D3:64:81:33:CF:C7:A1:D1
mozillacert11.pem	2014 年 3 月 13 日	05:63:B8:63:0D:62:D7:5A:BB:C8:AB:1E: 4B:DF:B5:A8:99:B2:4D:43
mozillacert29.pem	2014 年 3 月 13 日	74:F8:A3:C3:EF:E7:B3:90:06:4B:83:90: 3C:21:64:60:20:E5:DF:CE
mozillacert62.pem	2014 年 3 月 13 日	A1:DB:63:93:91:6F:17:E4:18:55:09:40: 04:15:C7:02:40:B0:AE:6B
mozillacert126.pem	2014 年 3 月 13 日	25:01:90:19:CF:FB:D9:99:1C:B7:68:25: 74:8D:94:5F:30:93:95:42
mozillacert18.pem	2014 年 3 月 13 日	79:98:A3:08:E1:4D:65:85:E6:C2:1E:15: 3A:71:9F:BA:5A:D3:4A:D9
mozillacert51.pem	2014 年 3 月 13 日	FA:B7:EE:36:97:26:62:FB:2D:B0:2A:F6: BF:03:FD:E8:7C:4B:2F:9B
mozillacert69.pem	2014 年 3 月 13 日	2F:78:3D:25:52:18:A7:4A:65:39:71:B5: 2C:A2:9C:45:15:6F:E9:19
mozillacert115.pem	2014 年 3 月 13 日	59:0D:2D:7D:88:4F:40:2E:61:7E:A5:62: 32:17:65:CF:17:D8:94:E9

名稱	日期	SHA1 指紋
mozillacert40.pem	2014 年 3 月 13 日	80:25:EF:F4:6E:70:C8:D4:72:24:65:84:FE:40:3B:8A:8D:6A:DB:F5
mozillacert58.pem	2014 年 3 月 13 日	8D:17:84:D5:37:F3:03:7D:EC:70:FE:57:8B:51:9A:99:E6:10:D7:B0
mozillacert104.pem	2014 年 3 月 13 日	4F:99:AA:93:FB:2B:D1:37:26:A1:99:4A:CE:7F:F0:05:F2:93:5D:1E
mozillacert91.pem	2014 年 3 月 13 日	3B:C0:38:0B:33:C3:F6:A6:0C:86:15:22:93:D9:DF:F5:4B:81:C0:04
mozillacert47.pem	2014 年 3 月 13 日	1B:4B:39:61:26:27:6B:64:91:A2:68:6D:D7:02:43:21:2D:1F:1D:96
mozillacert80.pem	2014 年 3 月 13 日	B8:23:6B:00:2F:1D:16:86:53:01:55:6C:11:A4:37:CA:EB:FF:C3:BB
mozillacert98.pem	2014 年 3 月 13 日	C9:A8:B9:E7:55:80:5E:58:E3:53:77:A7:25:EB:AF:C3:7B:27:CC:D7
mozillacert144.pem	2014 年 3 月 13 日	37:F7:6D:E6:07:7C:90:C5:B1:3E:93:1A:B7:41:10:B4:F2:E4:9A:27
mozillacert36.pem	2014 年 3 月 13 日	23:88:C9:D3:71:CC:9E:96:3D:FF:7D:3C:A7:CE:FC:D6:25:EC:19:0D
mozillacert3.pem	2014 年 3 月 13 日	87:9F:4B:EE:05:DF:98:58:3B:E3:60:D6:33:E7:0D:3F:FE:98:71:AF
mozillacert87.pem	2014 年 3 月 13 日	5F:3B:8C:F2:F8:10:B3:7D:78:B4:CE:EC:19:19:C3:73:34:B9:C7:74
mozillacert133.pem	2014 年 3 月 13 日	85:B5:FF:67:9B:0C:79:96:1F:C8:6E:44:22:00:46:13:DB:17:92:84
mozillacert25.pem	2014 年 3 月 13 日	4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD:56:BE:3D:9B:67:44:A5:E5

名稱	日期	SHA1 指紋
mozillacert76.pem	2014 年 3 月 13 日	F9:B5:B6:32:45:5F:9C:BE:EC:57:5F:80:DC:E9:6E:2C:C7:B2:78:B7
mozillacert122.pem	2014 年 3 月 13 日	02:FA:F3:E2:91:43:54:68:60:78:57:69:4D:F5:E4:5B:68:85:18:68
mozillacert14.pem	2014 年 3 月 13 日	5F:B7:EE:06:33:E2:59:DB:AD:0C:4C:9A:E6:D3:8F:1A:61:C7:DC:25
mozillacert65.pem	2014 年 3 月 13 日	69:BD:8C:F4:9C:D3:00:FB:59:2E:17:93:CA:55:6A:F3:EC:AA:35:FB
mozillacert111.pem	2014 年 3 月 13 日	9C:BB:48:53:F6:A4:F6:D3:52:A4:E8:32:52:55:60:13:F5:AD:AF:65
mozillacert129.pem	2014 年 3 月 13 日	E6:21:F3:35:43:79:05:9A:4B:68:30:9D:8A:2F:74:22:15:87:EC:79
mozillacert54.pem	2014 年 3 月 13 日	03:9E:ED:B8:0B:E7:A0:3C:69:53:89:3B:20:D2:D9:32:3A:4C:2A:FD
mozillacert100.pem	2014 年 3 月 13 日	58:E8:AB:B0:36:15:33:FB:80:F7:9B:1B:6D:29:D3:FF:8D:5F:00:F0
mozillacert118.pem	2014 年 3 月 13 日	7E:78:4A:10:1C:82:65:CC:2D:E1:F1:6D:47:B4:40:CA:D9:0A:19:45
mozillacert151.pem	2014 年 3 月 13 日	AC:ED:5F:65:53:FD:25:CE:01:5F:1F:7A:48:3B:6A:74:9F:61:78:C6
mozillacert43.pem	2014 年 3 月 13 日	F9:CD:0E:2C:DA:76:24:C1:8F:BD:F0:F0:AB:B6:45:B8:F7:FE:D5:7A
mozillacert107.pem	2014 年 3 月 13 日	8E:1C:74:F8:A6:20:B9:E5:8A:F4:61:FA:EC:2B:47:56:51:1A:52:C6
mozillacert94.pem	2014 年 3 月 13 日	49:0A:75:74:DE:87:0A:47:FE:58:EE:F6:C7:6B:EB:C6:0B:12:40:99

名稱	日期	SHA1 指紋
mozillacert140.pem	2014 年 3 月 13 日	CA:3A:FB:CF:12:40:36:4B:44:B2:16:20:88:80:48:39:19:93:7C:F7
mozillacert32.pem	2014 年 3 月 13 日	60:D6:89:74:B5:C2:65:9E:8A:0F:C1:88:7C:88:D2:46:69:1B:18:2C
mozillacert83.pem	2014 年 3 月 13 日	A0:73:E5:C5:BD:43:61:0D:86:4C:21:13:0A:85:58:57:CC:9C:EA:46
mozillacert147.pem	2014 年 3 月 13 日	58:11:9F:0E:12:82:87:EA:50:FD:D9:87:45:6F:4F:78:DC:FA:D6:D4
mozillacert21.pem	2014 年 3 月 13 日	9B:AA:E5:9F:56:EE:21:CB:43:5A:BE:25:93:DF:A7:F0:40:D1:1D:CB
mozillacert39.pem	2014 年 3 月 13 日	AE:50:83:ED:7C:F4:5C:BC:8F:61:C6:21:FE:68:5D:79:42:21:15:6E
mozillacert6.pem	2014 年 3 月 13 日	27:96:BA:E6:3F:18:01:E2:77:26:1B:A0:D7:77:70:02:8F:20:EE:E4
mozillacert72.pem	2014 年 3 月 13 日	47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:C2:54:FD:E6:8B
mozillacert136.pem	2014 年 3 月 13 日	D1:EB:23:A4:6D:17:D6:8F:D9:25:64:C2:F1:F1:60:17:64:D8:E3:49
mozillacert10.pem	2014 年 3 月 13 日	5F:3A:FC:0A:8B:64:F6:86:67:34:74:DF:7E:A9:A2:FE:F9:FA:7A:51
mozillacert28.pem	2014 年 3 月 13 日	66:31:BF:9E:F7:4F:9E:B6:C9:D5:A6:0C:BA:6A:BE:D1:F7:BD:EF:7B
mozillacert61.pem	2014 年 3 月 13 日	E0:B4:32:2E:B2:F6:A5:68:B6:54:53:84:48:18:4A:50:36:87:43:84
mozillacert79.pem	2014 年 3 月 13 日	D8:A6:33:2C:E0:03:6F:B1:85:F6:63:4F:7D:6A:06:65:26:32:28:27

名稱	日期	SHA1 指紋
mozillacert125.pem	2014 年 3 月 13 日	B3:1E:B1:B7:40:E3:6C:84:02:DA:DC:37:D4:4D:F5:D4:67:49:52:F9
mozillacert17.pem	2014 年 3 月 13 日	40:54:DA:6F:1C:3F:40:74:AC:ED:0F:EC:CD:DB:79:D1:53:FB:90:1D
mozillacert50.pem	2014 年 3 月 13 日	8C:96:BA:EB:DD:2B:07:07:48:EE:30:32:66:A0:F3:98:6E:7C:AE:58
mozillacert68.pem	2014 年 3 月 13 日	AE:C5:FB:3F:C8:E1:BF:C4:E5:4F:03:07:5A:9A:E8:00:B7:F7:B6:FA
mozillacert114.pem	2014 年 3 月 13 日	51:C6:E7:08:49:06:6E:F3:92:D4:5C:A0:0D:6D:A3:62:8F:C3:52:39
mozillacert57.pem	2014 年 3 月 13 日	D6:DA:A8:20:8D:09:D2:15:4D:24:B5:2F:CB:34:6E:B2:58:B2:8A:58
mozillacert103.pem	2014 年 3 月 13 日	70:C1:8D:74:B4:28:81:0A:E4:FD:A5:75:D7:01:9F:99:B0:3D:50:74
mozillacert90.pem	2014 年 3 月 13 日	F3:73:B3:87:06:5A:28:84:8A:F2:F3:4A:CE:19:2B:DD:C7:8E:9C:AC
mozillacert46.pem	2014 年 3 月 13 日	40:9D:4B:D9:17:B5:5C:27:B6:9B:64:CB:98:22:44:0D:CD:09:B8:89
mozillacert97.pem	2014 年 3 月 13 日	85:37:1C:A6:E5:50:14:3D:CE:28:03:47:1B:DE:3A:09:E8:F8:77:0F
mozillacert143.pem	2014 年 3 月 13 日	36:B1:2B:49:F9:81:9E:D7:4C:9E:BC:38:0F:C6:56:8F:5D:AC:B2:F7
mozillacert35.pem	2014 年 3 月 13 日	2A:C8:D5:8B:57:CE:BF:2F:49:AF:F2:FC:76:8F:51:14:62:90:7A:41
mozillacert2.pem	2014 年 3 月 13 日	22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:6C:3A

名稱	日期	SHA1 指紋
mozillacert86.pem	2014 年 3 月 13 日	74:2C:31:92:E6:07:E4:24:EB:45:49:54:2B:E1:BB:C5:3E:61:74:E2
mozillacert132.pem	2014 年 3 月 13 日	39:21:C1:15:C1:5D:0E:CA:5C:CB:5B:C4:F0:7D:21:D8:05:0B:56:6A
mozillacert24.pem	2014 年 3 月 13 日	59:AF:82:79:91:86:C7:B4:75:07:CB:CF:03:57:46:EB:04:DD:B7:16
mozillacert9.pem	2014 年 3 月 13 日	F4:8B:11:BF:DE:AB:BE:94:54:20:71:E6:41:DE:6B:BE:88:2B:40:B9
mozillacert75.pem	2014 年 3 月 13 日	D2:32:09:AD:23:D3:14:23:21:74:E4:0D:7F:9D:62:13:97:86:63:3A
mozillacert121.pem	2014 年 3 月 13 日	CC:AB:0E:A0:4C:23:01:D6:69:7B:DD:37:9F:CD:12:EB:24:E3:94:9D
mozillacert139.pem	2014 年 3 月 13 日	DE:3F:40:BD:50:93:D3:9B:6C:60:F6:DA:BC:07:62:01:00:89:76:C9
mozillacert13.pem	2014 年 3 月 13 日	06:08:3F:59:3F:15:A1:04:A0:69:A4:6B:A9:03:D0:06:B7:97:09:91
mozillacert64.pem	2014 年 3 月 13 日	62:7F:8D:78:27:65:63:99:D2:7D:7F:90:44:C9:FE:B3:F3:3E:FA:9A
mozillacert110.pem	2014 年 3 月 13 日	93:05:7A:88:15:C6:4F:CE:88:2F:FA:91:16:52:28:78:BC:53:64:17
mozillacert128.pem	2014 年 3 月 13 日	A9:E9:78:08:14:37:58:88:F2:05:19:B0:6D:2B:0D:2B:60:16:90:7D
mozillacert53.pem	2014 年 3 月 13 日	7F:8A:B0:CF:D0:51:87:6A:66:F3:36:0F:47:C8:8D:8C:D3:35:FC:74
mozillacert117.pem	2014 年 3 月 13 日	D4:DE:20:D0:5E:66:FC:53:FE:1A:50:88:2C:78:DB:28:52:CA:E4:74

名稱	日期	SHA1 指紋
mozillacert150.pem	2014 年 3 月 13 日	33:9B:6B:14:50:24:9B:55:7A:01:87:72: 84:D9:E0:2F:C3:D2:D8:E9
mozillacert42.pem	2014 年 3 月 13 日	85:A4:08:C0:9C:19:3E:5D:51:58:7D:CD: D6:13:30:FD:8C:DE:37:BF
mozillacert106.pem	2014 年 3 月 13 日	E7:A1:90:29:D3:D5:52:DC:0D:0F:C6:92: D3:EA:88:0D:15:2E:1A:6B
mozillacert93.pem	2014 年 3 月 13 日	31:F1:FD:68:22:63:20:EE:C6:3B:3F:9D: EA:4A:3E:53:7C:7C:39:17
mozillacert31.pem	2014 年 3 月 13 日	9F:74:4E:9F:2B:4D:BA:EC:0F:31:2C:50: B6:56:3B:8E:2D:93:C3:11
mozillacert49.pem	2014 年 3 月 13 日	61:57:3A:11:DF:0E:D8:7E:D5:92:65:22: EA:D0:56:D7:44:B3:23:71
mozillacert82.pem	2014 年 3 月 13 日	2E:14:DA:EC:28:F0:FA:1E:8E:38:9A:4E: AB:EB:26:C0:0A:D3:83:C3
mozillacert146.pem	2014 年 3 月 13 日	21:FC:BD:8E:7F:6C:AF:05:1B:D1:B3:43: EC:A8:E7:61:47:F2:0F:8A
mozillacert20.pem	2014 年 3 月 13 日	D8:C5:38:8A:B7:30:1B:1B:6E:D4:7A:E6: 45:25:3A:6F:9F:1A:27:61
mozillacert38.pem	2014 年 3 月 13 日	CB:A1:C5:F8:B0:E3:5E:B8:B9:45:12:D3: F9:34:A2:E9:06:10:D3:36
mozillacert5.pem	2014 年 3 月 13 日	B8:01:86:D1:EB:9C:86:A5:41:04:CF:30: 54:F3:4C:52:B7:E5:58:C6
mozillacert71.pem	2014 年 3 月 13 日	4A:BD:EE:EC:95:0D:35:9C:89:AE:C7:52: A1:2C:5B:29:F6:D6:AA:0C
mozillacert89.pem	2014 年 3 月 13 日	C8:EC:8C:87:92:69:CB:4B:AB:39:E9:8D: 7E:57:67:F3:14:95:73:9D

名稱	日期	SHA1 指紋
mozillacert135.pem	2014 年 3 月 13 日	62:52:DC:40:F7:11:43:A2:2F:DE:9E:F7:34:8E:06:42:51:B1:81:18
mozillacert27.pem	2014 年 3 月 13 日	3A:44:73:5A:E5:81:90:1F:24:86:61:46:1E:3B:9C:C4:5F:F5:3A:1B
mozillacert60.pem	2014 年 3 月 13 日	3B:C4:9F:48:F8:F3:73:A0:9C:1E:BD:F8:5B:B1:C3:65:C7:D8:11:B3
mozillacert78.pem	2014 年 3 月 13 日	29:36:21:02:8B:20:ED:02:F5:66:C5:32:D1:D6:ED:90:9F:45:00:2F
mozillacert124.pem	2014 年 3 月 13 日	4D:23:78:EC:91:95:39:B5:00:7F:75:8F:03:3B:21:1E:C5:4D:8B:CF
mozillacert16.pem	2014 年 3 月 13 日	DA:C9:02:4F:54:D8:F6:DF:94:93:5F:B1:73:26:38:CA:6A:D7:7C:13
mozillacert67.pem	2014 年 3 月 13 日	D6:9B:56:11:48:F0:1C:77:C5:45:78:C1:09:26:DF:5B:85:69:76:AD
mozillacert113.pem	2014 年 3 月 13 日	50:30:06:09:1D:97:D4:F5:AE:39:F7:CB:E7:92:7D:7D:65:2D:34:31
mozillacert56.pem	2014 年 3 月 13 日	F1:8B:53:8D:1B:E9:03:B6:A6:F0:56:43:5B:17:15:89:CA:F3:6B:F2
mozillacert102.pem	2014 年 3 月 13 日	96:C9:1B:0B:95:B4:10:98:42:FA:D0:D8:22:79:FE:60:FA:B9:16:83
mozillacert45.pem	2014 年 3 月 13 日	67:65:0D:F1:7E:8E:7E:5B:82:40:A4:F4:56:4B:CF:E2:3D:69:C6:F0
mozillacert109.pem	2014 年 3 月 13 日	B5:61:EB:EA:A4:DE:E4:25:4B:69:1A:98:A5:57:47:C2:34:C7:D9:71
mozillacert96.pem	2014 年 3 月 13 日	55:A6:72:3E:CB:F2:EC:CD:C3:23:74:70:19:9D:2A:BE:11:E3:81:D1

名稱	日期	SHA1 指紋
mozillacert142.pem	2014 年 3 月 13 日	1F:49:14:F7:D8:74:95:1D:DD:AE:02:C0:BE:FD:3A:2D:82:75:51:85
mozillacert34.pem	2014 年 3 月 13 日	59:22:A1:E1:5A:EA:16:35:21:F8:98:39:6A:46:46:B0:44:1B:0F:A9
mozillacert1.pem	2014 年 3 月 13 日	23:E5:94:94:51:95:F2:41:48:03:B4:D5:64:D2:A3:A3:F5:D8:8B:8C
mozillacert85.pem	2014 年 3 月 13 日	CF:9E:87:6D:D3:EB:FC:42:26:97:A3:B5:A3:7A:A0:76:A9:06:23:48
mozillacert131.pem	2014 年 3 月 13 日	37:9A:19:7B:41:85:45:35:0C:A6:03:69:F3:3C:2E:AF:47:4F:20:79
mozillacert149.pem	2014 年 3 月 13 日	6E:3A:55:A4:19:0C:19:5C:93:84:3C:C0:DB:72:2E:31:30:61:F0:B1
mozillacert23.pem	2014 年 3 月 13 日	91:C6:D6:EE:3E:8A:C8:63:84:E5:48:C2:99:29:5C:75:6C:81:7B:81
mozillacert8.pem	2014 年 3 月 13 日	3E:2B:F7:F2:03:1B:96:F3:8C:E6:C4:D8:A8:5D:3E:2D:58:47:6A:0F
mozillacert74.pem	2014 年 3 月 13 日	92:5A:8F:8D:2C:6D:04:E0:66:5F:59:6A:FF:22:D8:63:E8:25:6F:3F
mozillacert120.pem	2014 年 3 月 13 日	DA:40:18:8B:91:89:A3:ED:EE:AE:DA:97:FE:2F:9D:F5:B7:D1:8A:41
mozillacert138.pem	2014 年 3 月 13 日	E1:9F:E3:0E:8B:84:60:9E:80:9B:17:0D:72:A8:C5:BA:6E:14:09:BD
mozillacert12.pem	2014 年 3 月 13 日	A8:98:5D:3A:65:E5:E5:C4:B2:D7:D6:6D:40:C6:DD:2F:B1:9C:54:36
mozillacert63.pem	2014 年 3 月 13 日	89:DF:74:FE:5C:F4:0F:4A:80:F9:E3:37:7D:54:DA:91:E1:01:31:8E

名稱	日期	SHA1 指紋
mozillacert127.pem	2014 年 3 月 13 日	DE:28:F4:A4:FF:E5:B9:2F:A3:C5:03:D1: A3:49:A7:F9:96:2A:82:12
mozillacert19.pem	2014 年 3 月 13 日	B4:35:D4:E1:11:9D:1C:66:90:A7:49:EB: B3:94:BD:63:7B:A7:82:B7
mozillacert52.pem	2014 年 3 月 13 日	8B:AF:4C:9B:1D:F0:2A:92:F7:DA:12:8E: B9:1B:AC:F4:98:60:4B:6F
mozillacert116.pem	2014 年 3 月 13 日	2B:B1:F5:3E:55:0C:1D:C5:F1:D4:E6:B7: 6A:46:4B:55:06:02:AC:21
mozillacert41.pem	2014 年 3 月 13 日	6B:2F:34:AD:89:58:BE:62:FD:B0:6B:5C: CE:BB:9D:D9:4F:4E:39:F3
mozillacert59.pem	2014 年 3 月 13 日	36:79:CA:35:66:87:72:30:4D:30:A5:FB: 87:3B:0F:A7:7B:B7:0D:54
mozillacert105.pem	2014 年 3 月 13 日	77:47:4F:C6:30:E4:0F:4C:47:64:3F:84: BA:B8:C6:95:4A:8A:41:EC
mozillacert92.pem	2014 年 3 月 13 日	A3:F1:33:3F:E2:42:BF:CF:C5:D1:4E:8F: 39:42:98:40:68:10:D1:A0
mozillacert30.pem	2014 年 3 月 13 日	E7:B4:F6:9D:61:EC:90:69:DB:7E:90:A7: 40:1A:3C:F4:7D:4F:E8:EE
mozillacert48.pem	2014 年 3 月 13 日	A0:A1:AB:90:C9:FC:84:7B:3B:12:61:E8: 97:7D:5F:D3:22:61:D3:CC
verisignc4g2.pem	2014 年 3 月 20 日	0B:77:BE:BB:CB:7A:A2:47:05:DE:CC:0F: BD:6A:02:FC:7A:BD:9B:52
verisignc2g3.pem	2014 年 3 月 20 日	61:EF:43:D7:7F:CA:D4:61:51:BC:98:E0: C3:59:12:AF:9F:EB:63:11
verisignc1g6.pem	2014 年 12 月 31 日	51:7F:61:1E:29:91:6B:53:82:FB:72:E7: 44:D9:8D:C3:CC:53:6D:64

名稱	日期	SHA1 指紋
verisignc2g2.pem	2014 年 3 月 20 日	B3:EA:C4:47:76:C9:C8:1C:EA:F2:9D:95: B6:CC:A0:08:1B:67:EC:9D
verisignroot.pem	2014 年 3 月 20 日	36:79:CA:35:66:87:72:30:4D:30:A5:FB: 87:3B:0F:A7:7B:B7:0D:54
verisignc2g1.pem	2014 年 3 月 20 日	67:82:AA:E0:ED:EE:E2:1A:58:39:D3:C0: CD:14:68:0A:4F:60:14:2A
verisignc3g5.pem	2014 年 3 月 20 日	4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD: 56:BE:3D:9B:67:44:A5:E5
verisignc1g3.pem	2014 年 3 月 20 日	20:42:85:DC:F7:EB:76:41:95:57:8E:13: 6B:D4:B7:D1:E9:8E:46:A5
verisignc3g4.pem	2014 年 3 月 20 日	22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7: CF:8A:2D:64:C9:3F:6C:3A
verisignc1g2.pem	2014 年 3 月 20 日	27:3E:E1:24:57:FD:C4:F9:0C:55:E8:2B: 56:16:7F:62:F5:32:E5:47
verisignc3g3.pem	2014 年 3 月 20 日	13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3: 39:E2:55:76:60:9B:5C:C6
verisignc1g1.pem	2014 年 3 月 20 日	90:AE:A2:69:85:FF:14:80:4C:43:49:52: EC:E9:60:84:77:AF:55:6F
verisignc3g2.pem	2014 年 3 月 20 日	85:37:1C:A6:E5:50:14:3D:CE:28:03:47: 1B:DE:3A:09:E8:F8:77:0F
verisignc3g1.pem	2014 年 3 月 20 日	A1:DB:63:93:91:6F:17:E4:18:55:09:40: 04:15:C7:02:40:B0:AE:6B
verisignc2g6.pem	2014 年 12 月 31 日	40:B3:31:A0:E9:BF:E8:55:BC:39:93:CA: 70:4F:4E:C2:51:D4:1D:8F
verisignc4g3.pem	2014 年 3 月 20 日	C8:EC:8C:87:92:69:CB:4B:AB:39:E9:8D: 7E:57:67:F3:14:95:73:9D

名稱	日期	SHA1 指紋
gdroot-g2.pem	2014 年 12 月 31 日	47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:C2:54:FD:E6:8B
gd-class2-root.pem	2014 年 12 月 31 日	27:96:BA:E6:3F:18:01:E2:77:26:1B:A0:D7:77:70:02:8F:20:EE:E4
gd_bundle-g2.pem	2014 年 12 月 31 日	27:AC:93:69:FA:F2:52:07:BB:26:27:CE:FA:CC:BE:4E:F9:C3:19:B8
dstacescax6	2018 年 6 月 18 日	40:54:DA:6F:1C:3F:40:74:AC:ED:0F:EC:CD:DB:79:D1:53:FB:90:1D
gd_bundle-g2.pem	2018 年 6 月 18 日	27:AC:93:69:FA:F2:52:07:BB:26:27:CE:FA:CC:BE:4E:F9:C3:19:B8
verisignc4g3.pem	2018 年 6 月 18 日	C8:EC:8C:87:92:69:CB:4B:AB:39:E9:8D:7E:57:67:F3:14:95:73:9D
swisssignplatinumg2ca	2018 年 4 月 21 日	56:E0:FA:C0:3B:8F:18:23:55:18:E5:D3:11:CA:E8:C2:43:31:AB:66
geotrustprimarycertificatio nauthorityg3	2018 年 6 月 18 日	03:9E:ED:B8:0B:E7:A0:3C:69:53:89:3B:20:D2:D9:32:3A:4C:2A:FD
geotrustprimarycertificatio nauthorityg2	2018 年 6 月 18 日	8D:17:84:D5:37:F3:03:7D:EC:70:FE:57:8B:51:9A:99:E6:10:D7:B0
buypassclass2rootca	2018 年 6 月 18 日	49:0A:75:74:DE:87:0A:47:FE:58:EE:F6:C7:6B:EB:C6:0B:12:40:99
camerfirmachambersofcommerceroot	2018 年 6 月 18 日	6E:3A:55:A4:19:0C:19:5C:93:84:3C:C0:DB:72:2E:31:30:61:F0:B1
mozillacert20.pem	2018 年 6 月 18 日	D8:C5:38:8A:B7:30:1B:1B:6E:D4:7A:E6:45:25:3A:6F:9F:1A:27:61

名稱	日期	SHA1 指紋
mozillacert12.pem	2018 年 6 月 18 日	A8:98:5D:3A:65:E5:E5:C4:B2:D7:D6:6D:40:C6:DD:2F:B1:9C:54:36
mozillacert90.pem	2018 年 6 月 18 日	F3:73:B3:87:06:5A:28:84:8A:F2:F3:4A:CE:19:2B:DD:C7:8E:9C:AC
mozillacert82.pem	2018 年 6 月 18 日	2E:14:DA:EC:28:F0:FA:1E:8E:38:9A:4E:AB:EB:26:C0:0A:D3:83:C3
mozillacert140.pem	2018 年 6 月 18 日	CA:3A:FB:CF:12:40:36:4B:44:B2:16:20:88:80:48:39:19:93:7C:F7
mozillacert74.pem	2018 年 6 月 18 日	92:5A:8F:8D:2C:6D:04:E0:66:5F:59:6A:FF:22:D8:63:E8:25:6F:3F
mozillacert132.pem	2018 年 6 月 18 日	39:21:C1:15:C1:5D:0E:CA:5C:CB:5B:C4:F0:7D:21:D8:05:0B:56:6A
mozillacert66.pem	2018 年 6 月 18 日	DD:E1:D2:A9:01:80:2E:1D:87:5E:84:B3:80:7E:4B:B1:FD:99:41:34
mozillacert124.pem	2018 年 6 月 18 日	4D:23:78:EC:91:95:39:B5:00:7F:75:8F:03:3B:21:1E:C5:4D:8B:CF
mozillacert58.pem	2018 年 6 月 18 日	8D:17:84:D5:37:F3:03:7D:EC:70:FE:57:8B:51:9A:99:E6:10:D7:B0
securitycommunicationrootca2	2018 年 6 月 18 日	5F:3B:8C:F2:F8:10:B3:7D:78:B4:CE:EC:19:19:C3:73:34:B9:C7:74
mozillacert116.pem	2018 年 6 月 18 日	2B:B1:F5:3E:55:0C:1D:C5:F1:D4:E6:B7:6A:46:4B:55:06:02:AC:21
mozillacert108.pem	2018 年 6 月 18 日	B1:BC:96:8B:D4:F4:9D:62:2A:A8:9A:81:F2:15:01:52:A4:1D:82:9C
certigna	2018 年 6 月 18 日	B1:2E:13:63:45:86:A4:6F:1A:B2:60:68:37:58:2D:C4:AC:FD:94:97

名稱	日期	SHA1 指紋
mozillacert3.pem	2018 年 6 月 18 日	87:9F:4B:EE:05:DF:98:58:3B:E3:60:D6: 33:E7:0D:3F:FE:98:71:AF
verisignc1g1.pem	2018 年 6 月 18 日	90:AE:A2:69:85:FF:14:80:4C:43:49:52: EC:E9:60:84:77:AF:55:6F
verisignc4g2.pem	2018 年 6 月 18 日	0B:77:BE:BB:CB:7A:A2:47:05:DE:CC:0F: BD:6A:02:FC:7A:BD:9B:52
deutschetelekomrootca2	2018 年 6 月 18 日	85:A4:08:C0:9C:19:3E:5D:51:58:7D:CD: D6:13:30:FD:8C:DE:37:BF
starfieldrootg2ca	2018 年 4 月 21 日	B5:1C:06:7C:EE:2B:0C:3D:F8:55:AB:2D: 92:F4:FE:39:D4:E7:0F:0E
comodoecccertificationauthority	2018 年 6 月 18 日	9F:74:4E:9F:2B:4D:BA:EC:0F:31:2C:50: B6:56:3B:8E:2D:93:C3:11
digicertglobalrootg3	2018 年 6 月 18 日	7E:04:DE:89:6A:3E:66:6D:00:E6:87:D3: 3F:FA:D9:3B:E8:3D:34:9E
digicertglobalrootg2	2018 年 6 月 18 日	DF:3C:24:F9:BF:D6:66:76:1B:26:80:73: FE:06:D1:CC:8D:4F:82:A4
mozillacert11.pem	2018 年 6 月 18 日	05:63:B8:63:0D:62:D7:5A:BB:C8:AB:1E: 4B:DF:B5:A8:99:B2:4D:43
mozillacert81.pem	2018 年 6 月 18 日	07:E0:32:E0:20:B7:2C:3F:19:2F:06:28: A2:59:3A:19:A7:0F:06:9E
mozillacert73.pem	2018 年 6 月 18 日	B5:1C:06:7C:EE:2B:0C:3D:F8:55:AB:2D: 92:F4:FE:39:D4:E7:0F:0E
szafirrootca2	2018 年 6 月 18 日	E2:52:FA:95:3F:ED:DB:24:60:BD:6E:28: F3:9C:CC:CF:5E:B3:3F:DE
mozillacert131.pem	2018 年 6 月 18 日	37:9A:19:7B:41:85:45:35:0C:A6:03:69: F3:3C:2E:AF:47:4F:20:79

名稱	日期	SHA1 指紋
ecacc	2018 年 6 月 18 日	28:90:3A:63:5B:52:80:FA:E6:77:4C:0B:6D:A7:D6:BA:A6:4A:F2:E8
mozillacert65.pem	2018 年 6 月 18 日	69:BD:8C:F4:9C:D3:00:FB:59:2E:17:93:CA:55:6A:F3:EC:AA:35:FB
turktrustelektronik sertifikahizmet sa glayicisih5	2018 年 6 月 18 日	C4:18:F6:4D:46:D1:DF:00:3D:27:30:13:72:43:A9:12:11:C6:75:FB
mozillacert123.pem	2018 年 6 月 18 日	2A:B6:28:48:5E:78:FB:F3:AD:9E:79:10:DD:6B:DF:99:72:2C:96:E5
mozillacert57.pem	2018 年 6 月 18 日	D6:DA:A8:20:8D:09:D2:15:4D:24:B5:2F:CB:34:6E:B2:58:B2:8A:58
mozillacert115.pem	2018 年 6 月 18 日	59:0D:2D:7D:88:4F:40:2E:61:7E:A5:62:32:17:65:CF:17:D8:94:E9
mozillacert49.pem	2018 年 6 月 18 日	61:57:3A:11:DF:0E:D8:7E:D5:92:65:22:EA:D0:56:D7:44:B3:23:71
mozillacert107.pem	2018 年 6 月 18 日	8E:1C:74:F8:A6:20:B9:E5:8A:F4:61:FA:EC:2B:47:56:51:1A:52:C6
verisignclass3g4ca	2018 年 4 月 21 日	22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:6C:3A
securetrustca	2018 年 6 月 18 日	87:82:C6:C3:04:35:3B:CF:D2:96:92:D2:59:3E:7D:44:D9:34:FF:11
mozillacert2.pem	2018 年 6 月 18 日	22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:6C:3A
buypassclass2ca	2018 年 4 月 21 日	49:0A:75:74:DE:87:0A:47:FE:58:EE:F6:C7:6B:EB:C6:0B:12:40:99

名稱	日期	SHA1 指紋
secomscrootca2	2018 年 4 月 21 日	5F:3B:8C:F2:F8:10:B3:7D:78:B4:CE:EC:19:19:C3:73:34:B9:C7:74
secomscrootca1	2018 年 4 月 21 日	36:B1:2B:49:F9:81:9E:D7:4C:9E:BC:38:0F:C6:56:8F:5D:AC:B2:F7
trustisfpsrootca	2018 年 6 月 18 日	3B:C0:38:0B:33:C3:F6:A6:0C:86:15:22:93:D9:DF:F5:4B:81:C0:04
hongkongpostrootca1	2018 年 6 月 18 日	D6:DA:A8:20:8D:09:D2:15:4D:24:B5:2F:CB:34:6E:B2:58:B2:8A:58
certsignrootca	2018 年 6 月 18 日	FA:B7:EE:36:97:26:62:FB:2D:B0:2A:F6:BF:03:FD:E8:7C:4B:2F:9B
geotrustprimaryca	2018 年 4 月 21 日	32:3C:11:8E:1B:F7:B8:B6:52:54:E2:E2:10:0D:D6:02:90:37:F0:96
twcaglobalrootca	2018 年 6 月 18 日	9C:BB:48:53:F6:A4:F6:D3:52:A4:E8:32:52:55:60:13:F5:AD:AF:65
camerfirmachambersca	2018 年 4 月 21 日	78:6A:74:AC:76:AB:14:7F:9C:6A:30:50:BA:9E:A8:7E:FE:9A:CE:3C
mozillacert10.pem	2018 年 6 月 18 日	5F:3A:FC:0A:8B:64:F6:86:67:34:74:DF:7E:A9:A2:FE:F9:FA:7A:51
mozillacert80.pem	2018 年 6 月 18 日	B8:23:6B:00:2F:1D:16:86:53:01:55:6C:11:A4:37:CA:EB:FF:C3:BB
mozillacert72.pem	2018 年 6 月 18 日	47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:C2:54:FD:E6:8B
comodoaaaca	2018 年 4 月 21 日	D1:EB:23:A4:6D:17:D6:8F:D9:25:64:C2:F1:F1:60:17:64:D8:E3:49
mozillacert130.pem	2018 年 6 月 18 日	E5:DF:74:3C:B6:01:C4:9B:98:43:DC:AB:8C:E8:6A:81:10:9F:E4:8E

名稱	日期	SHA1 指紋
mozillacert64.pem	2018 年 6 月 18 日	62:7F:8D:78:27:65:63:99:D2:7D:7F:90: 44:C9:FE:B3:F3:3E:FA:9A
mozillacert122.pem	2018 年 6 月 18 日	02:FA:F3:E2:91:43:54:68:60:78:57:69: 4D:F5:E4:5B:68:85:18:68
mozillacert56.pem	2018 年 6 月 18 日	F1:8B:53:8D:1B:E9:03:B6:A6:F0:56:43: 5B:17:15:89:CA:F3:6B:F2
equifaxsecureebusi nessca1	2018 年 4 月 21 日	AE:E6:3D:70:E3:76:FB:C7:3A:EB:B0:A1: C1:D4:C4:7A:A7:40:B3:F4
camerfirmachambers ignca	2018 年 4 月 21 日	4A:BD:EE:EC:95:0D:35:9C:89:AE:C7:52: A1:2C:5B:29:F6:D6:AA:0C
mozillacert114.pem	2018 年 6 月 18 日	51:C6:E7:08:49:06:6E:F3:92:D4:5C:A0: 0D:6D:A3:62:8F:C3:52:39
mozillacert48.pem	2018 年 6 月 18 日	A0:A1:AB:90:C9:FC:84:7B:3B:12:61:E8: 97:7D:5F:D3:22:61:D3:CC
pscprocert	2018 年 6 月 18 日	70:C1:8D:74:B4:28:81:0A:E4:FD:A5:75: D7:01:9F:99:B0:3D:50:74
mozillacert106.pem	2018 年 6 月 18 日	E7:A1:90:29:D3:D5:52:DC:0D:0F:C6:92: D3:EA:88:0D:15:2E:1A:6B
mozillacert1.pem	2018 年 6 月 18 日	23:E5:94:94:51:95:F2:41:48:03:B4:D5: 64:D2:A3:A3:F5:D8:8B:8C
eecertificationcen trerootca	2018 年 6 月 18 日	C9:A8:B9:E7:55:80:5E:58:E3:53:77:A7: 25:EB:AF:C3:7B:27:CC:D7
digicertglobalroot ca	2018 年 6 月 18 日	A8:98:5D:3A:65:E5:E5:C4:B2:D7:D6:6D: 40:C6:DD:2F:B1:9C:54:36
thawteprimaryrootc ag3	2018 年 6 月 18 日	F1:8B:53:8D:1B:E9:03:B6:A6:F0:56:43: 5B:17:15:89:CA:F3:6B:F2

名稱	日期	SHA1 指紋
thawteprimaryrootcag2	2018 年 6 月 18 日	AA:DB:BC:22:23:8F:C4:01:A1:27:BB:38:DD:F4:1D:DB:08:9E:F0:12
entrustrootcertificationauthorityec1	2018 年 6 月 18 日	20:D8:06:40:DF:9B:25:F5:12:25:3A:11:EA:F7:59:8A:EB:14:B5:47
valicertclass2ca	2018 年 4 月 21 日	31:7A:2A:D0:7F:2B:33:5E:F5:A1:C3:4E:4B:57:E8:B7:D8:F1:FC:A6
globalchambersignroot2008	2018 年 6 月 18 日	4A:BD:EE:EC:95:0D:35:9C:89:AE:C7:52:A1:2C:5B:29:F6:D6:AA:0C
amazonrootca4	2018 年 6 月 18 日	F6:10:84:07:D6:F8:BB:67:98:0C:C2:E2:44:C2:EB:AE:1C:EF:63:BE
gd-class2-root.pem	2018 年 6 月 18 日	27:96:BA:E6:3F:18:01:E2:77:26:1B:A0:D7:77:70:02:8F:20:EE:E4
amazonrootca3	2018 年 6 月 18 日	0D:44:DD:8C:3C:8C:1A:1A:58:75:64:81:E9:0F:2E:2A:FF:B3:D2:6E
amazonrootca2	2018 年 6 月 18 日	5A:8C:EF:45:D7:A6:98:59:76:7A:8C:8B:44:96:B5:78:CF:47:4B:1A
securitycommunicationrootca	2018 年 6 月 18 日	36:B1:2B:49:F9:81:9E:D7:4C:9E:BC:38:0F:C6:56:8F:5D:AC:B2:F7
amazonrootca1	2018 年 6 月 18 日	8D:A7:F9:65:EC:5E:FC:37:91:0F:1C:6E:59:FD:C1:CC:6A:6E:DE:16
acraizfnmtrcm	2018 年 6 月 18 日	EC:50:35:07:B2:15:C4:95:62:19:E2:A8:9A:5B:42:99:2C:4C:2C:20
quovadisrootca3g3	2018 年 6 月 18 日	48:12:BD:92:3C:A8:C4:39:06:E7:30:6D:27:96:E6:A4:CF:22:2E:7D

名稱	日期	SHA1 指紋
certplusrootcag2	2018 年 6 月 18 日	4F:65:8E:1F:E9:06:D8:28:02:E9:54:47:41:C9:54:25:5D:69:CC:1A
certplusrootcag1	2018 年 6 月 18 日	22:FD:D0:B7:FD:A2:4E:0D:AC:49:2C:A0:AC:A6:7B:6A:1F:E3:F7:66
mozillacert71.pem	2018 年 6 月 18 日	4A:BD:EE:EC:95:0D:35:9C:89:AE:C7:52:A1:2C:5B:29:F6:D6:AA:0C
mozillacert63.pem	2018 年 6 月 18 日	89:DF:74:FE:5C:F4:0F:4A:80:F9:E3:37:7D:54:DA:91:E1:01:31:8E
mozillacert121.pem	2018 年 6 月 18 日	CC:AB:0E:A0:4C:23:01:D6:69:7B:DD:37:9F:CD:12:EB:24:E3:94:9D
ttelesecglobalrootclass3ca	2018 年 4 月 21 日	55:A6:72:3E:CB:F2:EC:CD:C3:23:74:70:19:9D:2A:BE:11:E3:81:D1
mozillacert55.pem	2018 年 6 月 18 日	AA:DB:BC:22:23:8F:C4:01:A1:27:BB:38:DD:F4:1D:DB:08:9E:F0:12
mozillacert113.pem	2018 年 6 月 18 日	50:30:06:09:1D:97:D4:F5:AE:39:F7:CB:E7:92:7D:7D:65:2D:34:31
baltimorecybertrustca	2018 年 4 月 21 日	D4:DE:20:D0:5E:66:FC:53:FE:1A:50:88:2C:78:DB:28:52:CA:E4:74
mozillacert47.pem	2018 年 6 月 18 日	1B:4B:39:61:26:27:6B:64:91:A2:68:6D:D7:02:43:21:2D:1F:1D:96
mozillacert105.pem	2018 年 6 月 18 日	77:47:4F:C6:30:E4:0F:4C:47:64:3F:84:BA:B8:C6:95:4A:8A:41:EC
mozillacert39.pem	2018 年 6 月 18 日	AE:50:83:ED:7C:F4:5C:BC:8F:61:C6:21:FE:68:5D:79:42:21:15:6E
usertrustecccertificationauthority	2018 年 6 月 18 日	D1:CB:CA:5D:B2:D5:2A:7F:69:3B:67:4D:E5:F0:5A:1D:0C:95:7D:F0

名稱	日期	SHA1 指紋
mozillacert0.pem	2018 年 6 月 18 日	97:81:79:50:D8:1C:96:70:CC:34:D8:09: CF:79:44:31:36:7E:F4:74
securitycommunicat ionevrootca1	2018 年 6 月 18 日	FE:B8:C4:32:DC:F9:76:9A:CE:AE:3D:D8: 90:8F:FD:28:86:65:64:7D
verisignc3g5.pem	2018 年 6 月 18 日	4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD: 56:BE:3D:9B:67:44:A5:E5
globalsignr3ca	2018 年 4 月 21 日	D6:9B:56:11:48:F0:1C:77:C5:45:78:C1: 09:26:DF:5B:85:69:76:AD
trustcoreca1	2018 年 6 月 18 日	58:D1:DF:95:95:67:6B:63:C0:F0:5B:1C: 17:4D:8B:84:0B:C8:78:BD
equifaxsecuregloba lebusinessca1	2018 年 4 月 21 日	3A:74:CB:7A:47:DB:70:DE:89:1F:24:35: 98:64:B8:2D:82:BD:1A:36
geotrustuniversalc a	2018 年 6 月 18 日	E6:21:F3:35:43:79:05:9A:4B:68:30:9D: 8A:2F:74:22:15:87:EC:79
affirmtrustpremium ca	2018 年 4 月 21 日	D8:A6:33:2C:E0:03:6F:B1:85:F6:63:4F: 7D:6A:06:65:26:32:28:27
staatdernederlande nrootcag3	2018 年 6 月 18 日	D8:EB:6B:41:51:92:59:E0:F3:E7:85:00: C0:3D:B6:88:97:C9:EE:FC
staatdernederlande nrootcag2	2018 年 6 月 18 日	59:AF:82:79:91:86:C7:B4:75:07:CB:CF: 03:57:46:EB:04:DD:B7:16
mozillacert70.pem	2018 年 6 月 18 日	78:6A:74:AC:76:AB:14:7F:9C:6A:30:50: BA:9E:A8:7E:FE:9A:CE:3C
secomevrootca1	2018 年 4 月 21 日	FE:B8:C4:32:DC:F9:76:9A:CE:AE:3D:D8: 90:8F:FD:28:86:65:64:7D
geotrustglobalca	2018 年 6 月 18 日	DE:28:F4:A4:FF:E5:B9:2F:A3:C5:03:D1: A3:49:A7:F9:96:2A:82:12

名稱	日期	SHA1 指紋
mozillacert62.pem	2018 年 6 月 18 日	A1:DB:63:93:91:6F:17:E4:18:55:09:40:04:15:C7:02:40:B0:AE:6B
mozillacert120.pem	2018 年 6 月 18 日	DA:40:18:8B:91:89:A3:ED:EE:AE:DA:97:FE:2F:9D:F5:B7:D1:8A:41
mozillacert54.pem	2018 年 6 月 18 日	03:9E:ED:B8:0B:E7:A0:3C:69:53:89:3B:20:D2:D9:32:3A:4C:2A:FD
mozillacert112.pem	2018 年 6 月 18 日	43:13:BB:96:F1:D5:86:9B:C1:4E:6A:92:F6:CF:F6:34:69:87:82:37
mozillacert46.pem	2018 年 6 月 18 日	40:9D:4B:D9:17:B5:5C:27:B6:9B:64:CB:98:22:44:0D:CD:09:B8:89
swisssigngoldcag2	2018 年 6 月 18 日	D8:C5:38:8A:B7:30:1B:1B:6E:D4:7A:E6:45:25:3A:6F:9F:1A:27:61
mozillacert104.pem	2018 年 6 月 18 日	4F:99:AA:93:FB:2B:D1:37:26:A1:99:4A:CE:7F:F0:05:F2:93:5D:1E
mozillacert38.pem	2018 年 6 月 18 日	CB:A1:C5:F8:B0:E3:5E:B8:B9:45:12:D3:F9:34:A2:E9:06:10:D3:36
certplusclass3ppri maryca	2018 年 4 月 21 日	21:6B:2A:29:E6:2A:00:CE:82:01:46:D8:24:41:41:B9:25:11:B2:79
entrustrootcertifi cationauthorityg2	2018 年 6 月 18 日	8C:F4:27:FD:79:0C:3A:D1:66:06:8D:E8:1E:57:EF:BB:93:22:72:D4
godaddyrootg2ca	2018 年 4 月 21 日	47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:C2:54:FD:E6:8B
cfcaevroot	2018 年 6 月 18 日	E2:B8:29:4B:55:84:AB:6B:58:C2:90:46:6C:AC:3F:B8:39:8F:84:83
verisignc3g4.pem	2018 年 6 月 18 日	22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:6C:3A

名稱	日期	SHA1 指紋
geotrustuniversalca2	2018 年 6 月 18 日	37:9A:19:7B:41:85:45:35:0C:A6:03:69:F3:3C:2E:AF:47:4F:20:79
starfieldservicesrootg2ca	2018 年 4 月 21 日	92:5A:8F:8D:2C:6D:04:E0:66:5F:59:6A:FF:22:D8:63:E8:25:6F:3F
digicerthighassuranceevrootca	2018 年 6 月 18 日	5F:B7:EE:06:33:E2:59:DB:AD:0C:4C:9A:E6:D3:8F:1A:61:C7:DC:25
entrustnetpremium2048secureserverca	2018 年 6 月 18 日	50:30:06:09:1D:97:D4:F5:AE:39:F7:CB:E7:92:7D:7D:65:2D:34:31
camerfirmaglobalchambersignroot	2018 年 6 月 18 日	33:9B:6B:14:50:24:9B:55:7A:01:87:72:84:D9:E0:2F:C3:D2:D8:E9
verisignclass3g3ca	2018 年 4 月 21 日	13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3:39:E2:55:76:60:9B:5C:C6
godaddyclass2ca	2018 年 6 月 18 日	27:96:BA:E6:3F:18:01:E2:77:26:1B:A0:D7:77:70:02:8F:20:EE:E4
mozillacert61.pem	2018 年 6 月 18 日	E0:B4:32:2E:B2:F6:A5:68:B6:54:53:84:48:18:4A:50:36:87:43:84
mozillacert53.pem	2018 年 6 月 18 日	7F:8A:B0:CF:D0:51:87:6A:66:F3:36:0F:47:C8:8D:8C:D3:35:FC:74
atostrustedroot2011	2018 年 6 月 18 日	2B:B1:F5:3E:55:0C:1D:C5:F1:D4:E6:B7:6A:46:4B:55:06:02:AC:21
mozillacert111.pem	2018 年 6 月 18 日	9C:BB:48:53:F6:A4:F6:D3:52:A4:E8:32:52:55:60:13:F5:AD:AF:65
staatdernederlandenevrootca	2018 年 6 月 18 日	76:E2:7E:C1:4F:DB:82:C1:C0:A6:75:B5:05:BE:3D:29:B4:ED:DB:BB
mozillacert45.pem	2018 年 6 月 18 日	67:65:0D:F1:7E:8E:7E:5B:82:40:A4:F4:56:4B:CF:E2:3D:69:C6:F0

名稱	日期	SHA1 指紋
mozillacert103.pem	2018 年 6 月 18 日	70:C1:8D:74:B4:28:81:0A:E4:FD:A5:75:D7:01:9F:99:B0:3D:50:74
mozillacert37.pem	2018 年 6 月 18 日	B1:2E:13:63:45:86:A4:6F:1A:B2:60:68:37:58:2D:C4:AC:FD:94:97
mozillacert29.pem	2018 年 6 月 18 日	74:F8:A3:C3:EF:E7:B3:90:06:4B:83:90:3C:21:64:60:20:E5:DF:CE
izenpecom	2018 年 6 月 18 日	2F:78:3D:25:52:18:A7:4A:65:39:71:B5:2C:A2:9C:45:15:6F:E9:19
comodorsacertificationauthority	2018 年 6 月 18 日	AF:E5:D2:44:A8:D1:19:42:30:FF:47:9F:E2:F8:97:BB:CD:7A:8C:B4
mozillacert99.pem	2018 年 6 月 18 日	F1:7F:6F:B6:31:DC:99:E3:A3:C8:7F:FE:1C:F1:81:10:88:D9:60:33
mozillacert149.pem	2018 年 6 月 18 日	6E:3A:55:A4:19:0C:19:5C:93:84:3C:C0:DB:72:2E:31:30:61:F0:B1
utnuserfirstobjectca	2018 年 4 月 21 日	E1:2D:FB:4B:41:D7:D9:C3:2B:30:51:4B:AC:1D:81:D8:38:5E:2D:46
verisignc3g3.pem	2018 年 6 月 18 日	13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3:39:E2:55:76:60:9B:5C:C6
dstrootcax3	2018 年 6 月 18 日	DA:C9:02:4F:54:D8:F6:DF:94:93:5F:B1:73:26:38:CA:6A:D7:7C:13
addtrustexternalroot	2018 年 6 月 18 日	02:FA:F3:E2:91:43:54:68:60:78:57:69:4D:F5:E4:5B:68:85:18:68
certumtrustednetworkca	2018 年 6 月 18 日	07:E0:32:E0:20:B7:2C:3F:19:2F:06:28:A2:59:3A:19:A7:0F:06:9E
affirmtrustpremiumecc	2018 年 6 月 18 日	B8:23:6B:00:2F:1D:16:86:53:01:55:6C:11:A4:37:CA:EB:FF:C3:BB

名稱	日期	SHA1 指紋
starfieldclass2ca	2018 年 6 月 18 日	AD:7E:1C:28:B0:64:EF:8F:60:03:40:20:14:C3:D0:E3:37:0E:B5:8A
actalisauthenticationrootca	2018 年 6 月 18 日	F3:73:B3:87:06:5A:28:84:8A:F2:F3:4A:CE:19:2B:DD:C7:8E:9C:AC
verisignclass2g3ca	2018 年 4 月 21 日	61:EF:43:D7:7F:CA:D4:61:51:BC:98:E0:C3:59:12:AF:9F:EB:63:11
isrgrootx1	2018 年 6 月 18 日	CA:BD:2A:79:A1:07:6A:31:F2:1D:25:36:35:CB:03:9D:43:29:A5:E8
godaddyrootcertificateauthorityg2	2018 年 6 月 18 日	47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:C2:54:FD:E6:8B
mozillacert60.pem	2018 年 6 月 18 日	3B:C4:9F:48:F8:F3:73:A0:9C:1E:BD:F8:5B:B1:C3:65:C7:D8:11:B3
chunghwaepkirootca	2018 年 4 月 21 日	67:65:0D:F1:7E:8E:7E:5B:82:40:A4:F4:56:4B:CF:E2:3D:69:C6:F0
mozillacert52.pem	2018 年 6 月 18 日	8B:AF:4C:9B:1D:F0:2A:92:F7:DA:12:8E:B9:1B:AC:F4:98:60:4B:6F
microseceszignorootca2009	2018 年 6 月 18 日	89:DF:74:FE:5C:F4:0F:4A:80:F9:E3:37:7D:54:DA:91:E1:01:31:8E
securesignrootca11	2018 年 6 月 18 日	3B:C4:9F:48:F8:F3:73:A0:9C:1E:BD:F8:5B:B1:C3:65:C7:D8:11:B3
mozillacert110.pem	2018 年 6 月 18 日	93:05:7A:88:15:C6:4F:CE:88:2F:FA:91:16:52:28:78:BC:53:64:17
mozillacert44.pem	2018 年 6 月 18 日	5F:43:E5:B1:BF:F8:78:8C:AC:1C:C7:CA:4A:9A:C6:22:2B:CC:34:C6
mozillacert102.pem	2018 年 6 月 18 日	96:C9:1B:0B:95:B4:10:98:42:FA:D0:D8:22:79:FE:60:FA:B9:16:83

名稱	日期	SHA1 指紋
mozillacert36.pem	2018 年 6 月 18 日	23:88:C9:D3:71:CC:9E:96:3D:FF:7D:3C: A7:CE:FC:D6:25:EC:19:0D
mozillacert28.pem	2018 年 6 月 18 日	66:31:BF:9E:F7:4F:9E:B6:C9:D5:A6:0C: BA:6A:BE:D1:F7:BD:EF:7B
baltimorecybertrustroot	2018 年 6 月 18 日	D4:DE:20:D0:5E:66:FC:53:FE:1A:50:88: 2C:78:DB:28:52:CA:E4:74
amzninternalrootca	2008 年 12 月 12 日	A7:B7:F6:15:8A:FF:1E:C8:85:13:38:BC: 93:EB:A2:AB:A4:09:EF:06
mozillacert98.pem	2018 年 6 月 18 日	C9:A8:B9:E7:55:80:5E:58:E3:53:77:A7: 25:EB:AF:C3:7B:27:CC:D7
mozillacert148.pem	2018 年 6 月 18 日	04:83:ED:33:99:AC:36:08:05:87:22:ED: BC:5E:46:00:E3:BE:F9:D7
verisignc3g2.pem	2018 年 6 月 18 日	85:37:1C:A6:E5:50:14:3D:CE:28:03:47: 1B:DE:3A:09:E8:F8:77:0F
quovadisrootca2g3	2018 年 6 月 18 日	09:3C:61:F3:8B:8B:DC:7D:55:DF:75:38: 02:05:00:E1:25:F5:C8:36
geotrustprimarycertificatio nauthority	2018 年 6 月 18 日	32:3C:11:8E:1B:F7:B8:B6:52:54:E2:E2: 10:0D:D6:02:90:37:F0:96
opentrustrootcag3	2018 年 6 月 18 日	6E:26:64:F3:56:BF:34:55:BF:D1:93:3F: 7C:01:DE:D8:13:DA:8A:A6
opentrustrootcag2	2018 年 6 月 18 日	79:5F:88:60:C5:AB:7C:3D:92:E6:CB:F4: 8D:E1:45:CD:11:EF:60:0B
opentrustrootcag1	2018 年 6 月 18 日	79:91:E8:34:F7:E2:EE:DD:08:95:01:52: E9:55:2D:14:E9:58:D5:7E

名稱	日期	SHA1 指紋
verisignclass3ca	2018 年 4 月 21 日	A1:DB:63:93:91:6F:17:E4:18:55:09:40:04:15:C7:02:40:B0:AE:6B
globalsignca	2018 年 4 月 21 日	B1:BC:96:8B:D4:F4:9D:62:2A:A8:9A:81:F2:15:01:52:A4:1D:82:9C
ttelesecglobalrootclass2ca	2018 年 4 月 21 日	59:0D:2D:7D:88:4F:40:2E:61:7E:A5:62:32:17:65:CF:17:D8:94:E9
verisignclass1g3ca	2018 年 4 月 21 日	20:42:85:DC:F7:EB:76:41:95:57:8E:13:6B:D4:B7:D1:E9:8E:46:A5
verisignuniversalrootca	2018 年 4 月 21 日	36:79:CA:35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0D:54
soneraclass2ca	2018 年 4 月 21 日	37:F7:6D:E6:07:7C:90:C5:B1:3E:93:1A:B7:41:10:B4:F2:E4:9A:27
starfieldservicesrootcertificateauthorityg2	2018 年 6 月 18 日	92:5A:8F:8D:2C:6D:04:E0:66:5F:59:6A:FF:22:D8:63:E8:25:6F:3F
mozillacert51.pem	2018 年 6 月 18 日	FA:B7:EE:36:97:26:62:FB:2D:B0:2A:F6:BF:03:FD:E8:7C:4B:2F:9B
mozillacert43.pem	2018 年 6 月 18 日	F9:CD:0E:2C:DA:76:24:C1:8F:BD:F0:F0:AB:B6:45:B8:F7:FE:D5:7A
mozillacert101.pem	2018 年 6 月 18 日	99:A6:9B:E6:1A:FE:88:6B:4D:2B:82:00:7C:B8:54:FC:31:7E:15:39
mozillacert35.pem	2018 年 6 月 18 日	2A:C8:D5:8B:57:CE:BF:2F:49:AF:F2:FC:76:8F:51:14:62:90:7A:41
globalsignr2ca	2018 年 4 月 21 日	75:E0:AB:B6:13:85:12:27:1C:04:F8:5F:DD:DE:38:E4:B7:24:2E:FE

名稱	日期	SHA1 指紋
mozillacert27.pem	2018 年 6 月 18 日	3A:44:73:5A:E5:81:90:1F:24:86:61:46:1E:3B:9C:C4:5F:F5:3A:1B
affirmtrustpremium	2018 年 6 月 18 日	D8:A6:33:2C:E0:03:6F:B1:85:F6:63:4F:7D:6A:06:65:26:32:28:27
mozillacert19.pem	2018 年 6 月 18 日	B4:35:D4:E1:11:9D:1C:66:90:A7:49:EB:B3:94:BD:63:7B:A7:82:B7
mozillacert97.pem	2018 年 6 月 18 日	85:37:1C:A6:E5:50:14:3D:CE:28:03:47:1B:DE:3A:09:E8:F8:77:0F
netlockaranyclassg oldfotanusitvany	2018 年 6 月 18 日	06:08:3F:59:3F:15:A1:04:A0:69:A4:6B:A9:03:D0:06:B7:97:09:91
mozillacert89.pem	2018 年 6 月 18 日	C8:EC:8C:87:92:69:CB:4B:AB:39:E9:8D:7E:57:67:F3:14:95:73:9D
verisignroot.pem	2018 年 6 月 18 日	36:79:CA:35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0D:54
mozillacert147.pem	2018 年 6 月 18 日	58:11:9F:0E:12:82:87:EA:50:FD:D9:87:45:6F:4F:78:DC:FA:D6:D4
aolrootca2	2018 年 4 月 21 日	85:B5:FF:67:9B:0C:79:96:1F:C8:6E:44:22:00:46:13:DB:17:92:84
cia-crt-g3-01-ca	2016 年 11 月 23 日	2B:EE:2C:BA:A3:1D:B5:FE:60:40:41:95:08:ED:46:82:39:4D:ED:E2
aolrootca1	2018 年 4 月 21 日	39:21:C1:15:C1:5D:0E:CA:5C:CB:5B:C4:F0:7D:21:D8:05:0B:56:6A
verisignc3g1.pem	2018 年 6 月 18 日	A1:DB:63:93:91:6F:17:E4:18:55:09:40:04:15:C7:02:40:B0:AE:6B
mozillacert139.pem	2018 年 6 月 18 日	DE:3F:40:BD:50:93:D3:9B:6C:60:F6:DA:BC:07:62:01:00:89:76:C9

名稱	日期	SHA1 指紋
soneraclass2rootca	2018 年 6 月 18 日	37:F7:6D:E6:07:7C:90:C5:B1:3E:93:1A: B7:41:10:B4:F2:E4:9A:27
swisssignsilverg2ca	2018 年 4 月 21 日	9B:AA:E5:9F:56:EE:21:CB:43:5A:BE:25: 93:DF:A7:F0:40:D1:1D:CB
thawteprimaryrootca	2018 年 6 月 18 日	91:C6:D6:EE:3E:8A:C8:63:84:E5:48:C2: 99:29:5C:75:6C:81:7B:81
gdcatrustauthr5root	2018 年 6 月 18 日	0F:36:38:5B:81:1A:25:C3:9B:31:4E:83: CA:E9:34:66:70:CC:74:B4
trustcenterclass4caii	2018 年 4 月 21 日	A6:9A:91:FD:05:7F:13:6A:42:63:0B:B1: 76:0D:2D:51:12:0C:16:50
usertrustrsacertificationauthority	2018 年 6 月 18 日	2B:8F:1B:57:33:0D:BB:A2:D0:7A:6C:51: F7:0E:E9:0D:DA:B9:AD:8E
digicertassuredidrootg3	2018 年 6 月 18 日	F5:17:A2:4F:9A:48:C6:C9:F8:A2:00:26: 9F:DC:0F:48:2C:AB:30:89
digicertassuredidrootg2	2018 年 6 月 18 日	A1:4B:48:D9:43:EE:0A:0E:40:90:4F:3C: E0:A4:C0:91:93:51:5D:3F
mozillacert50.pem	2018 年 6 月 18 日	8C:96:BA:EB:DD:2B:07:07:48:EE:30:32: 66:A0:F3:98:6E:7C:AE:58
mozillacert42.pem	2018 年 6 月 18 日	85:A4:08:C0:9C:19:3E:5D:51:58:7D:CD: D6:13:30:FD:8C:DE:37:BF
mozillacert100.pem	2018 年 6 月 18 日	58:E8:AB:B0:36:15:33:FB:80:F7:9B:1B: 6D:29:D3:FF:8D:5F:00:F0
mozillacert34.pem	2018 年 6 月 18 日	59:22:A1:E1:5A:EA:16:35:21:F8:98:39: 6A:46:46:B0:44:1B:0F:A9
affirmtrustcommercialca	2018 年 4 月 21 日	F9:B5:B6:32:45:5F:9C:BE:EC:57:5F:80: DC:E9:6E:2C:C7:B2:78:B7

名稱	日期	SHA1 指紋
mozillacert26.pem	2018 年 6 月 18 日	87:82:C6:C3:04:35:3B:CF:D2:96:92:D2:59:3E:7D:44:D9:34:FF:11
globalsigneccrootc ar5	2018 年 6 月 18 日	1F:24:C6:30:CD:A4:18:EF:20:69:FF:AD:4F:DD:5F:46:3A:1B:69:AA
globalsigneccrootc ar4	2018 年 6 月 18 日	69:69:56:2E:40:80:F4:24:A1:E7:19:9F:14:BA:F3:EE:58:AB:6A:BB
buypassclass3rootc a	2018 年 6 月 18 日	DA:FA:F7:FA:66:84:EC:06:8F:14:50:BD:C7:C2:81:A5:BC:A9:64:57
mozillacert18.pem	2018 年 6 月 18 日	79:98:A3:08:E1:4D:65:85:E6:C2:1E:15:3A:71:9F:BA:5A:D3:4A:D9
mozillacert96.pem	2018 年 6 月 18 日	55:A6:72:3E:CB:F2:EC:CD:C3:23:74:70:19:9D:2A:BE:11:E3:81:D1
verisignc2g6.pem	2018 年 6 月 18 日	40:B3:31:A0:E9:BF:E8:55:BC:39:93:CA:70:4F:4E:C2:51:D4:1D:8F
secomvalicertclass 1ca	2018 年 4 月 21 日	E5:DF:74:3C:B6:01:C4:9B:98:43:DC:AB:8C:E8:6A:81:10:9F:E4:8E
mozillacert88.pem	2018 年 6 月 18 日	FE:45:65:9B:79:03:5B:98:A1:61:B5:51:2E:AC:DA:58:09:48:22:4D
accvraiz1	2018 年 6 月 18 日	93:05:7A:88:15:C6:4F:CE:88:2F:FA:91:16:52:28:78:BC:53:64:17
mozillacert146.pem	2018 年 6 月 18 日	21:FC:BD:8E:7F:6C:AF:05:1B:D1:B3:43:EC:A8:E7:61:47:F2:0F:8A
mozillacert138.pem	2018 年 6 月 18 日	E1:9F:E3:0E:8B:84:60:9E:80:9B:17:0D:72:A8:C5:BA:6E:14:09:BD
verisignclass3g2ca	2018 年 4 月 21 日	85:37:1C:A6:E5:50:14:3D:CE:28:03:47:1B:DE:3A:09:E8:F8:77:0F

名稱	日期	SHA1 指紋
dtrustrootclass3ca2ev2009	2018 年 6 月 18 日	96:C9:1B:0B:95:B4:10:98:42:FA:D0:D8:22:79:FE:60:FA:B9:16:83
xrampglobalca	2018 年 4 月 21 日	B8:01:86:D1:EB:9C:86:A5:41:04:CF:30:54:F3:4C:52:B7:E5:58:C6
mozillacert9.pem	2018 年 6 月 18 日	F4:8B:11:BF:DE:AB:BE:94:54:20:71:E6:41:DE:6B:BE:88:2B:40:B9
verisignuniversalrootcertificationauthority	2018 年 6 月 18 日	36:79:CA:35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0D:54
tubitakkamusmsslkoksertifिकासिसुरम1	2018 年 6 月 18 日	31:43:64:9B:EC:CE:27:EC:ED:3A:3F:0B:8F:0D:E4:E8:91:DD:EE:CA
mozillacert41.pem	2018 年 6 月 18 日	6B:2F:34:AD:89:58:BE:62:FD:B0:6B:5C:CE:BB:9D:D9:4F:4E:39:F3
mozillacert33.pem	2018 年 6 月 18 日	FE:B8:C4:32:DC:F9:76:9A:CE:AE:3D:D8:90:8F:FD:28:86:65:64:7D
mozillacert25.pem	2018 年 6 月 18 日	4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD:56:BE:3D:9B:67:44:A5:E5
mozillacert17.pem	2018 年 6 月 18 日	40:54:DA:6F:1C:3F:40:74:AC:ED:0F:EC:CD:DB:79:D1:53:FB:90:1D
mozillacert95.pem	2018 年 6 月 18 日	DA:FA:F7:FA:66:84:EC:06:8F:14:50:BD:C7:C2:81:A5:BC:A9:64:57
affirmtrustpremiumeccca	2018 年 4 月 21 日	B8:23:6B:00:2F:1D:16:86:53:01:55:6C:11:A4:37:CA:EB:FF:C3:BB
mozillacert87.pem	2018 年 6 月 18 日	5F:3B:8C:F2:F8:10:B3:7D:78:B4:CE:EC:19:19:C3:73:34:B9:C7:74

名稱	日期	SHA1 指紋
mozillacert145.pem	2018 年 6 月 18 日	10:1D:FA:3F:D5:0B:CB:BB:9B:B5:60:0C: 19:55:A4:1A:F4:73:3A:04
mozillacert79.pem	2018 年 6 月 18 日	D8:A6:33:2C:E0:03:6F:B1:85:F6:63:4F: 7D:6A:06:65:26:32:28:27
mozillacert137.pem	2018 年 6 月 18 日	4A:65:D5:F4:1D:EF:39:B8:B8:90:4A:4A: D3:64:81:33:CF:C7:A1:D1
digicertassuredidr ootca	2018 年 6 月 18 日	05:63:B8:63:0D:62:D7:5A:BB:C8:AB:1E: 4B:DF:B5:A8:99:B2:4D:43
addtrustqualifiedc a	2018 年 4 月 21 日	4D:23:78:EC:91:95:39:B5:00:7F:75:8F: 03:3B:21:1E:C5:4D:8B:CF
mozillacert129.pem	2018 年 6 月 18 日	E6:21:F3:35:43:79:05:9A:4B:68:30:9D: 8A:2F:74:22:15:87:EC:79
verisignclass2g2ca	2018 年 4 月 21 日	B3:EA:C4:47:76:C9:C8:1C:EA:F2:9D:95: B6:CC:A0:08:1B:67:EC:9D
baltimorecodesigni ngca	2018 年 4 月 21 日	30:46:D8:C8:88:FF:69:30:C3:4A:FC:CD: 49:27:08:7C:60:56:7B:0D
luxtrustglobalroot 2	2018 年 6 月 18 日	1E:0E:56:19:0A:D1:8B:25:98:B2:04:44: FF:66:8A:04:17:99:5F:3F
visaecommercerooot	2018 年 6 月 18 日	70:17:9B:86:8C:00:A4:FA:60:91:52:22: 3F:9F:3E:32:BD:E0:05:62
oistewisekeyglobal rootgbca	2018 年 6 月 18 日	0F:F9:40:76:18:D3:D7:6A:4B:98:F0:A8: 35:9E:0C:FD:27:AC:CC:ED
mozillacert8.pem	2018 年 6 月 18 日	3E:2B:F7:F2:03:1B:96:F3:8C:E6:C4:D8: A8:5D:3E:2D:58:47:6A:0F
comodocertificatio nauthority	2018 年 6 月 18 日	66:31:BF:9E:F7:4F:9E:B6:C9:D5:A6:0C: BA:6A:BE:D1:F7:BD:EF:7B

名稱	日期	SHA1 指紋
cia-crt-g3-02-ca	2016 年 11 月 23 日	96:4A:BB:A7:BD:DA:FC:97:34:C0:0A:2D:F0:05:98:F7:E6:C6:6F:09
verisignc1g6.pem	2018 年 6 月 18 日	51:7F:61:1E:29:91:6B:53:82:FB:72:E7:44:D9:8D:C3:CC:53:6D:64
trustcenterclass2caii	2018 年 4 月 21 日	AE:50:83:ED:7C:F4:5C:BC:8F:61:C6:21:FE:68:5D:79:42:21:15:6E
quovadisrootca1g3	2018 年 6 月 18 日	1B:8E:EA:57:96:29:1A:C9:39:EA:B8:0A:81:1A:73:73:C0:93:79:67
mozillacert40.pem	2018 年 6 月 18 日	80:25:EF:F4:6E:70:C8:D4:72:24:65:84:FE:40:3B:8A:8D:6A:DB:F5
cadisigrootr2	2018 年 6 月 18 日	B5:61:EB:EA:A4:DE:E4:25:4B:69:1A:98:A5:57:47:C2:34:C7:D9:71
cadisigrootr1	2018 年 6 月 18 日	8E:1C:74:F8:A6:20:B9:E5:8A:F4:61:FA:EC:2B:47:56:51:1A:52:C6
mozillacert32.pem	2018 年 6 月 18 日	60:D6:89:74:B5:C2:65:9E:8A:0F:C1:88:7C:88:D2:46:69:1B:18:2C
utndatacorpsgccca	2018 年 4 月 21 日	58:11:9F:0E:12:82:87:EA:50:FD:D9:87:45:6F:4F:78:DC:FA:D6:D4
mozillacert24.pem	2018 年 6 月 18 日	59:AF:82:79:91:86:C7:B4:75:07:CB:CF:03:57:46:EB:04:DD:B7:16
addtrustclass1ca	2018 年 4 月 21 日	CC:AB:0E:A0:4C:23:01:D6:69:7B:DD:37:9F:CD:12:EB:24:E3:94:9D
mozillacert16.pem	2018 年 6 月 18 日	DA:C9:02:4F:54:D8:F6:DF:94:93:5F:B1:73:26:38:CA:6A:D7:7C:13
affirmtrustnetworkingca	2018 年 4 月 21 日	29:36:21:02:8B:20:ED:02:F5:66:C5:32:D1:D6:ED:90:9F:45:00:2F

名稱	日期	SHA1 指紋
mozillacert94.pem	2018 年 6 月 18 日	49:0A:75:74:DE:87:0A:47:FE:58:EE:F6:C7:6B:EB:C6:0B:12:40:99
mozillacert86.pem	2018 年 6 月 18 日	74:2C:31:92:E6:07:E4:24:EB:45:49:54:2B:E1:BB:C5:3E:61:74:E2
mozillacert144.pem	2018 年 6 月 18 日	37:F7:6D:E6:07:7C:90:C5:B1:3E:93:1A:B7:41:10:B4:F2:E4:9A:27
mozillacert78.pem	2018 年 6 月 18 日	29:36:21:02:8B:20:ED:02:F5:66:C5:32:D1:D6:ED:90:9F:45:00:2F
mozillacert136.pem	2018 年 6 月 18 日	D1:EB:23:A4:6D:17:D6:8F:D9:25:64:C2:F1:F1:60:17:64:D8:E3:49
mozillacert128.pem	2018 年 6 月 18 日	A9:E9:78:08:14:37:58:88:F2:05:19:B0:6D:2B:0D:2B:60:16:90:7D
verisignclass1g2ca	2018 年 4 月 21 日	27:3E:E1:24:57:FD:C4:F9:0C:55:E8:2B:56:16:7F:62:F5:32:E5:47
hellenicacademican dresearch instituti onsrootca2015	2018 年 6 月 18 日	01:0C:06:95:A6:98:19:14:FF:BF:5F:C6:B0:B6:95:EA:29:E9:12:A6
soneraclass1ca	2018 年 4 月 21 日	07:47:22:01:99:CE:74:B9:7C:B0:3D:79:B2:64:A2:C8:55:E9:33:FF
hellenicacademican dresearch instituti onsrootca2011	2018 年 6 月 18 日	FE:45:65:9B:79:03:5B:98:A1:61:B5:51:2E:AC:DA:58:09:48:22:4D
certumtrustednetwo rkca2	2018 年 6 月 18 日	D3:DD:48:3E:2B:BF:4C:05:E8:AF:10:F5:FA:76:26:CF:D3:DC:30:92

名稱	日期	SHA1 指紋
equifaxsecureca	2018 年 4 月 21 日	D2:32:09:AD:23:D3:14:23:21:74:E4:0D:7F:9D:62:13:97:86:63:3A
thawteserverca	2018 年 4 月 21 日	9F:AD:91:A6:CE:6A:C6:C5:00:47:C4:4E:C9:D4:A5:0D:92:D8:49:79
mozillacert7.pem	2018 年 6 月 18 日	AD:7E:1C:28:B0:64:EF:8F:60:03:40:20:14:C3:D0:E3:37:0E:B5:8A
affirmtrustnetworking	2018 年 6 月 18 日	29:36:21:02:8B:20:ED:02:F5:66:C5:32:D1:D6:ED:90:9F:45:00:2F
deprecateditsecca	2012 年 1 月 27 日	12:12:0B:03:0E:15:14:54:F4:DD:B3:F5:DE:13:6E:83:5A:29:72:9D
globalsignrootcar3	2018 年 6 月 18 日	D6:9B:56:11:48:F0:1C:77:C5:45:78:C1:09:26:DF:5B:85:69:76:AD
globalsignrootcar2	2018 年 6 月 18 日	75:E0:AB:B6:13:85:12:27:1C:04:F8:5F:DD:DE:38:E4:B7:24:2E:FE
quovadisrootca	2018 年 6 月 18 日	DE:3F:40:BD:50:93:D3:9B:6C:60:F6:DA:BC:07:62:01:00:89:76:C9
mozillacert31.pem	2018 年 6 月 18 日	9F:74:4E:9F:2B:4D:BA:EC:0F:31:2C:50:B6:56:3B:8E:2D:93:C3:11
entrustrootcertificationauthority	2018 年 6 月 18 日	B3:1E:B1:B7:40:E3:6C:84:02:DA:DC:37:D4:4D:F5:D4:67:49:52:F9
mozillacert23.pem	2018 年 6 月 18 日	91:C6:D6:EE:3E:8A:C8:63:84:E5:48:C2:99:29:5C:75:6C:81:7B:81
mozillacert15.pem	2018 年 6 月 18 日	74:20:74:41:72:9C:DD:92:EC:79:31:D8:23:10:8D:C2:81:92:E2:BB
verisignc2g3.pem	2018 年 6 月 18 日	61:EF:43:D7:7F:CA:D4:61:51:BC:98:E0:C3:59:12:AF:9F:EB:63:11

名稱	日期	SHA1 指紋
mozillacert93.pem	2018 年 6 月 18 日	31:F1:FD:68:22:63:20:EE:C6:3B:3F:9D: EA:4A:3E:53:7C:7C:39:17
mozillacert151.pem	2018 年 6 月 18 日	AC:ED:5F:65:53:FD:25:CE:01:5F:1F:7A: 48:3B:6A:74:9F:61:78:C6
mozillacert85.pem	2018 年 6 月 18 日	CF:9E:87:6D:D3:EB:FC:42:26:97:A3:B5: A3:7A:A0:76:A9:06:23:48
certplusclass2primaryca	2018 年 6 月 18 日	74:20:74:41:72:9C:DD:92:EC:79:31:D8: 23:10:8D:C2:81:92:E2:BB
mozillacert143.pem	2018 年 6 月 18 日	36:B1:2B:49:F9:81:9E:D7:4C:9E:BC:38: 0F:C6:56:8F:5D:AC:B2:F7
mozillacert77.pem	2018 年 6 月 18 日	13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3: 39:E2:55:76:60:9B:5C:C6
mozillacert135.pem	2018 年 6 月 18 日	62:52:DC:40:F7:11:43:A2:2F:DE:9E:F7: 34:8E:06:42:51:B1:81:18
mozillacert69.pem	2018 年 6 月 18 日	2F:78:3D:25:52:18:A7:4A:65:39:71:B5: 2C:A2:9C:45:15:6F:E9:19
mozillacert127.pem	2018 年 6 月 18 日	DE:28:F4:A4:FF:E5:B9:2F:A3:C5:03:D1: A3:49:A7:F9:96:2A:82:12
mozillacert119.pem	2018 年 6 月 18 日	75:E0:AB:B6:13:85:12:27:1C:04:F8:5F: DD:DE:38:E4:B7:24:2E:FE
geotrustprimaryca3	2018 年 4 月 21 日	03:9E:ED:B8:0B:E7:A0:3C:69:53:89:3B: 20:D2:D9:32:3A:4C:2A:FD
identrustpublicsectorrootca1	2018 年 6 月 18 日	BA:29:41:60:77:98:3F:F4:F3:EF:F2:31: 05:3B:2E:EA:6D:4D:45:FD
geotrustprimaryca2	2018 年 4 月 21 日	8D:17:84:D5:37:F3:03:7D:EC:70:FE:57: 8B:51:9A:99:E6:10:D7:B0

名稱	日期	SHA1 指紋
trustcorrootcertca2	2018 年 6 月 18 日	B8:BE:6D:CB:56:F1:55:B9:63:D4:12:CA:4E:06:34:C7:94:B2:1C:C0
mozillacert6.pem	2018 年 6 月 18 日	27:96:BA:E6:3F:18:01:E2:77:26:1B:A0:D7:77:70:02:8F:20:EE:E4
trustcorrootcertca1	2018 年 6 月 18 日	FF:BD:CD:E7:82:C8:43:5E:3C:6F:26:86:5C:CA:A8:3A:45:5B:C3:0A
networksolutionscertificat authority	2018 年 6 月 18 日	74:F8:A3:C3:EF:E7:B3:90:06:4B:83:90:3C:21:64:60:20:E5:DF:CE
twcarootcertificat ionauthority	2018 年 6 月 18 日	CF:9E:87:6D:D3:EB:FC:42:26:97:A3:B5:A3:7A:A0:76:A9:06:23:48
addtrustexternalca	2018 年 4 月 21 日	02:FA:F3:E2:91:43:54:68:60:78:57:69:4D:F5:E4:5B:68:85:18:68
verisignclass3g5ca	2018 年 4 月 21 日	4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD:56:BE:3D:9B:67:44:A5:E5
autoridaddecertifi cacionfir maprofesi onalcifa62634068	2018 年 6 月 18 日	AE:C5:FB:3F:C8:E1:BF:C4:E5:4F:03:07:5A:9A:E8:00:B7:F7:B6:FA
hellenicacademican dresearch instituti onseccrootca2015	2018 年 6 月 18 日	9F:F1:71:8D:92:D5:9A:F3:7D:74:97:B4:BC:6F:84:68:0B:BA:B6:66
verisigntsaca	2018 年 4 月 21 日	20:CE:B1:F0:F5:1C:0E:19:A9:F3:8D:B1:AA:8E:03:8C:AA:7A:C7:01
utnuserfirsthardwa reca	2018 年 4 月 21 日	04:83:ED:33:99:AC:36:08:05:87:22:ED:BC:5E:46:00:E3:BE:F9:D7

名稱	日期	SHA1 指紋
identrustcommercialrootca1	2018 年 6 月 18 日	DF:71:7E:AA:4A:D9:4E:C9:55:84:99:60:2D:48:DE:5F:BC:F0:3A:25
dtrustrootclass3ca22009	2018 年 6 月 18 日	58:E8:AB:B0:36:15:33:FB:80:F7:9B:1B:6D:29:D3:FF:8D:5F:00:F0
epkirootcertificationauthority	2018 年 6 月 18 日	67:65:0D:F1:7E:8E:7E:5B:82:40:A4:F4:56:4B:CF:E2:3D:69:C6:F0
mozillacert30.pem	2018 年 6 月 18 日	E7:B4:F6:9D:61:EC:90:69:DB:7E:90:A7:40:1A:3C:F4:7D:4F:E8:EE
teliasonerarootca1	2018 年 6 月 18 日	43:13:BB:96:F1:D5:86:9B:C1:4E:6A:92:F6:CF:F6:34:69:87:82:37
buypassclass3ca	2018 年 4 月 21 日	DA:FA:F7:FA:66:84:EC:06:8F:14:50:BD:C7:C2:81:A5:BC:A9:64:57
mozillacert22.pem	2018 年 6 月 18 日	32:3C:11:8E:1B:F7:B8:B6:52:54:E2:E2:10:0D:D6:02:90:37:F0:96
mozillacert14.pem	2018 年 6 月 18 日	5F:B7:EE:06:33:E2:59:DB:AD:0C:4C:9A:E6:D3:8F:1A:61:C7:DC:25
verisignc2g2.pem	2018 年 6 月 18 日	B3:EA:C4:47:76:C9:C8:1C:EA:F2:9D:95:B6:CC:A0:08:1B:67:EC:9D
certumca	2018 年 4 月 21 日	62:52:DC:40:F7:11:43:A2:2F:DE:9E:F7:34:8E:06:42:51:B1:81:18
mozillacert92.pem	2018 年 6 月 18 日	A3:F1:33:3F:E2:42:BF:CF:C5:D1:4E:8F:39:42:98:40:68:10:D1:A0
mozillacert150.pem	2018 年 6 月 18 日	33:9B:6B:14:50:24:9B:55:7A:01:87:72:84:D9:E0:2F:C3:D2:D8:E9
mozillacert84.pem	2018 年 6 月 18 日	D3:C0:63:F2:19:ED:07:3E:34:AD:5D:75:0B:32:76:29:FF:D5:9A:F2

名稱	日期	SHA1 指紋
ttelesecglobalrootclass3	2018 年 6 月 18 日	55:A6:72:3E:CB:F2:EC:CD:C3:23:74:70:19:9D:2A:BE:11:E3:81:D1
globalsignrootca	2018 年 6 月 18 日	B1:BC:96:8B:D4:F4:9D:62:2A:A8:9A:81:F2:15:01:52:A4:1D:82:9C
ttelesecglobalrootclass2	2018 年 6 月 18 日	59:0D:2D:7D:88:4F:40:2E:61:7E:A5:62:32:17:65:CF:17:D8:94:E9
mozillacert142.pem	2018 年 6 月 18 日	1F:49:14:F7:D8:74:95:1D:DD:AE:02:C0:BE:FD:3A:2D:82:75:51:85
mozillacert76.pem	2018 年 6 月 18 日	F9:B5:B6:32:45:5F:9C:BE:EC:57:5F:80:DC:E9:6E:2C:C7:B2:78:B7
mozillacert134.pem	2018 年 6 月 18 日	70:17:9B:86:8C:00:A4:FA:60:91:52:22:3F:9F:3E:32:BD:E0:05:62
mozillacert68.pem	2018 年 6 月 18 日	AE:C5:FB:3F:C8:E1:BF:C4:E5:4F:03:07:5A:9A:E8:00:B7:F7:B6:FA
etugracertificatio nauthority	2018 年 6 月 18 日	51:C6:E7:08:49:06:6E:F3:92:D4:5C:A0:0D:6D:A3:62:8F:C3:52:39
mozillacert126.pem	2018 年 6 月 18 日	25:01:90:19:CF:FB:D9:99:1C:B7:68:25:74:8D:94:5F:30:93:95:42
keynectisrootca	2018 年 4 月 21 日	9C:61:5C:4D:4D:85:10:3A:53:26:C2:4D:BA:EA:E4:A2:D2:D5:CC:97
mozillacert118.pem	2018 年 6 月 18 日	7E:78:4A:10:1C:82:65:CC:2D:E1:F1:6D:47:B4:40:CA:D9:0A:19:45
quovadisrootca3	2018 年 6 月 18 日	1F:49:14:F7:D8:74:95:1D:DD:AE:02:C0:BE:FD:3A:2D:82:75:51:85
quovadisrootca2	2018 年 6 月 18 日	CA:3A:FB:CF:12:40:36:4B:44:B2:16:20:88:80:48:39:19:93:7C:F7

名稱	日期	SHA1 指紋
mozillacert5.pem	2018 年 6 月 18 日	B8:01:86:D1:EB:9C:86:A5:41:04:CF:30:54:F3:4C:52:B7:E5:58:C6
verisignc1g3.pem	2018 年 6 月 18 日	20:42:85:DC:F7:EB:76:41:95:57:8E:13:6B:D4:B7:D1:E9:8E:46:A5
cybertrustglobalroot	2018 年 6 月 18 日	5F:43:E5:B1:BF:F8:78:8C:AC:1C:C7:CA:4A:9A:C6:22:2B:CC:34:C6
amzninternalinfosecag3	2015 年 2 月 27 日	B9:B1:CA:38:F7:BF:9C:D2:D4:95:E7:B6:5E:75:32:9B:A8:78:2E:F6
starfieldrootcertificateauthorityg2	2018 年 6 月 18 日	B5:1C:06:7C:EE:2B:0C:3D:F8:55:AB:2D:92:F4:FE:39:D4:E7:0F:0E
entrust2048ca	2018 年 4 月 21 日	50:30:06:09:1D:97:D4:F5:AE:39:F7:CB:E7:92:7D:7D:65:2D:34:31
swisssignsilverca2	2018 年 6 月 18 日	9B:AA:E5:9F:56:EE:21:CB:43:5A:BE:25:93:DF:A7:F0:40:D1:1D:CB
affirmtrustcommercial	2018 年 6 月 18 日	F9:B5:B6:32:45:5F:9C:BE:EC:57:5F:80:DC:E9:6E:2C:C7:B2:78:B7
certinomisrootca	2018 年 6 月 18 日	9D:70:BB:01:A5:A4:A0:18:11:2E:F7:1C:01:B9:32:C5:34:E7:88:A8
xrampglobalcaroot	2018 年 6 月 18 日	B8:01:86:D1:EB:9C:86:A5:41:04:CF:30:54:F3:4C:52:B7:E5:58:C6
secureglobalca	2018 年 6 月 18 日	3A:44:73:5A:E5:81:90:1F:24:86:61:46:1E:3B:9C:C4:5F:F5:3A:1B
swisssigngoldg2ca	2018 年 4 月 21 日	D8:C5:38:8A:B7:30:1B:1B:6E:D4:7A:E6:45:25:3A:6F:9F:1A:27:61
mozillacert21.pem	2018 年 6 月 18 日	9B:AA:E5:9F:56:EE:21:CB:43:5A:BE:25:93:DF:A7:F0:40:D1:1D:CB

名稱	日期	SHA1 指紋
mozillacert13.pem	2018 年 6 月 18 日	06:08:3F:59:3F:15:A1:04:A0:69:A4:6B: A9:03:D0:06:B7:97:09:91
verisignc2g1.pem	2018 年 6 月 18 日	67:82:AA:E0:ED:EE:E2:1A:58:39:D3:C0: CD:14:68:0A:4F:60:14:2A
mozillacert91.pem	2018 年 6 月 18 日	3B:C0:38:0B:33:C3:F6:A6:0C:86:15:22: 93:D9:DF:F5:4B:81:C0:04
oistewisekeyglobal rootgaca	2018 年 6 月 18 日	59:22:A1:E1:5A:EA:16:35:21:F8:98:39: 6A:46:46:B0:44:1B:0F:A9
mozillacert83.pem	2018 年 6 月 18 日	A0:73:E5:C5:BD:43:61:0D:86:4C:21:13: 0A:85:58:57:CC:9C:EA:46
entrustevca	2018 年 4 月 21 日	B3:1E:B1:B7:40:E3:6C:84:02:DA:DC:37: D4:4D:F5:D4:67:49:52:F9
mozillacert141.pem	2018 年 6 月 18 日	31:7A:2A:D0:7F:2B:33:5E:F5:A1:C3:4E: 4B:57:E8:B7:D8:F1:FC:A6
mozillacert75.pem	2018 年 6 月 18 日	D2:32:09:AD:23:D3:14:23:21:74:E4:0D: 7F:9D:62:13:97:86:63:3A
mozillacert133.pem	2018 年 6 月 18 日	85:B5:FF:67:9B:0C:79:96:1F:C8:6E:44: 22:00:46:13:DB:17:92:84
mozillacert67.pem	2018 年 6 月 18 日	D6:9B:56:11:48:F0:1C:77:C5:45:78:C1: 09:26:DF:5B:85:69:76:AD
mozillacert125.pem	2018 年 6 月 18 日	B3:1E:B1:B7:40:E3:6C:84:02:DA:DC:37: D4:4D:F5:D4:67:49:52:F9
mozillacert59.pem	2018 年 6 月 18 日	36:79:CA:35:66:87:72:30:4D:30:A5:FB: 87:3B:0F:A7:7B:B7:0D:54
thawtepremiumserve rca	2018 年 4 月 21 日	E0:AB:05:94:20:72:54:93:05:60:62:02: 36:70:F7:CD:2E:FC:66:66

名稱	日期	SHA1 指紋
mozillacert117.pem	2018 年 6 月 18 日	D4:DE:20:D0:5E:66:FC:53:FE:1A:50:88:2C:78:DB:28:52:CA:E4:74
utnuserfirstclientauthemailca	2018 年 4 月 21 日	B1:72:B1:A5:6D:95:F9:1F:E5:02:87:E1:4D:37:EA:6A:44:63:76:8A
entrustrootcag2	2018 年 4 月 21 日	8C:F4:27:FD:79:0C:3A:D1:66:06:8D:E8:1E:57:EF:BB:93:22:72:D4
mozillacert109.pem	2018 年 6 月 18 日	B5:61:EB:EA:A4:DE:E4:25:4B:69:1A:98:A5:57:47:C2:34:C7:D9:71
digicerttrustedrootg4	2018 年 6 月 18 日	DD:FB:16:CD:49:31:C9:73:A2:03:7D:3F:C8:3A:4D:7D:77:5D:05:E4
gdroot-g2.pem	2018 年 6 月 18 日	47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:C2:54:FD:E6:8B
comodoaaaservicesroot	2018 年 6 月 18 日	D1:EB:23:A4:6D:17:D6:8F:D9:25:64:C2:F1:F1:60:17:64:D8:E3:49
mozillacert4.pem	2018 年 6 月 18 日	E3:92:51:2F:0A:CF:F5:05:DF:F6:DE:06:7F:75:37:E1:65:EA:57:4B
verisignclass3publicprimarycertificationauthorityg5	2018 年 6 月 18 日	4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD:56:BE:3D:9B:67:44:A5:E5
chambersofcommerceroot2008	2018 年 6 月 18 日	78:6A:74:AC:76:AB:14:7F:9C:6A:30:50:BA:9E:A8:7E:FE:9A:CE:3C
verisignclass3publicprimarycertificationauthorityg4	2018 年 6 月 18 日	22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:6C:3A

名稱	日期	SHA1 指紋
verisignclass3publicprimarycertificationauthorityg3	2018 年 6 月 18 日	13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3:39:E2:55:76:60:9B:5C:C6
thawtepersonalfreemailca	2018 年 4 月 21 日	E6:18:83:AE:84:CA:C1:C1:CD:52:AD:E8:E9:25:2B:45:A6:4F:B7:E2
verisignclg2.pem	2018 年 6 月 18 日	27:3E:E1:24:57:FD:C4:F9:0C:55:E8:2B:56:16:7F:62:F5:32:E5:47
gtecybertrustglobalca	2018 年 4 月 21 日	97:81:79:50:D8:1C:96:70:CC:34:D8:09:CF:79:44:31:36:7E:F4:74
trustcenteruniversalcai	2018 年 4 月 21 日	6B:2F:34:AD:89:58:BE:62:FD:B0:6B:5C:CE:BB:9D:D9:4F:4E:39:F3
camerfirmachamberscommerceca	2018 年 4 月 21 日	6E:3A:55:A4:19:0C:19:5C:93:84:3C:C0:DB:72:2E:31:30:61:F0:B1
verisignclass1ca	2018 年 4 月 21 日	CE:6A:64:A3:09:E4:2F:BB:D9:85:1C:45:3E:64:09:EA:E8:7D:60:F1

AWS AppSync 解析程式映射範本變更日誌

Note

我們現在主要支援 APPSYNC_JS 執行期及其文件。請考慮[在此處](#)使用 APPSYNC_JS 執行期及其指南。

解析程式和函數映射範本已採取版本控制。映射範本版本，例如 2018-05-29) 規定了以下內容：

- 請求範本所提供資料來源請求組態的預期形狀
- 請求映射範本和回應映射範本的執行行為

版本呈現格式為 YYYY-MM-DD，較晚日期代表較新版本。此頁面列出目前支援的映射範本版本之間的差異 AWS AppSync。

主題

- [每種版本陣列的可用資料來源操作](#)
- [變更單位解析程式映射範本的版本](#)
- [變更函數的版本](#)
- [2018-05-29](#)
- [2017-02-28](#)

每種版本陣列的可用資料來源操作

支援的操作/版本	2017-02-28	2018-05-29
AWS Lambda 調用	是	是
AWS Lambda BatchInvoke	是	是
None Datasource	是	是
Amazon OpenSearch GET	是	是
Amazon OpenSearch POST	是	是
Amazon OpenSearch PUT	是	是
Amazon OpenSearch DELETE	是	是
Amazon OpenSearch GET	是	是
DynamoDB GetItem	是	是
DynamoDB Scan	是	是
DynamoDB Query	是	是
DynamoDB DeleteItem	是	是

支援的操作/版本	2017-02-28	2018-05-29
DynamoDB PutItem	是	是
DynamoDB BatchGetItem	否	是
DynamoDB BatchPutItem	否	是
DynamoDB BatchDeleteItem	否	是
HTTP	否	是
Amazon RDS	否	是

注意：函數目前僅支援 2018-05-29 版本。

變更單位解析程式映射範本的版本

對於單位解析程式，版本會指定為要求映射範本的內容部分。若要更新版本，只要直接將 `version` 欄位更新成新版本。

例如，若要更新 AWS Lambda 範本上的版本：

```
{
  "version": "2017-02-28",
  "operation": "Invoke",
  "payload": {
    "field": "getPost",
    "arguments": $utils.toJson($context.arguments)
  }
}
```

您需要將 2017-02-28 的版本欄位更新成 2018-05-29，如下所示：

```
{
  "version": "2018-05-29", ## Note the version
  "operation": "Invoke",
  "payload": {
    "field": "getPost",
    "arguments": $utils.toJson($context.arguments)
  }
}
```

```
}
```

變更函數的版本

對於函數，版本會透過函數物件上的 `functionVersion` 欄位指定。若要更新版本，可直接更新 `functionVersion`。注意：目前只有 2018-05-29 支援函數。

下面是可更新現有函數版本的 CLI 命令範例：

```
aws appsync update-function \  
--api-id REPLACE_WITH_API_ID \  
--function-id REPLACE_WITH_FUNCTION_ID \  
--data-source-name "PostTable" \  
--function-version "2018-05-29" \  
--request-mapping-template "{...}" \  
--response-mapping-template "$util.toJson($ctx.result)"
```

注意：建議省略要求映射範本的版本欄位，因為其不會獲得接收。如果您在函數要求映射範本內指定版本，`functionVersion` 欄位的值將覆寫此版本值。

2018-05-29

行為變更

- 如果資料來源呼叫結果是 `null`，回應映射範本將會執行。
- 如果資料來源呼叫產生錯誤，您現在要負責處理錯誤，而回應映射範本評估結果將一律置放在 GraphQL 回應 `data` 區塊中。

推理

- `null` 呼叫結果具有意義，而且在某些應用程式使用案例中，我們可能希望以自訂方式處理 `null` 結果。例如，應用程式可以檢查記錄是否存在於 Amazon DynamoDB 資料表，以便執行一些授權檢查。在這種情況下，`null` 呼叫結果就能表示使用者應未獲得授權。執行回應映射範本現在會提供引發關於未經授權錯誤的能力。這個行為能為 API 設計師提供更好的控制功能。

假設我們使用下列的回應映射範本：

```
$util.toJson($ctx.result)
```

過去使用 2017-02-28 時，如果 `$ctx.result` 傳回 Null，回應映射範本將不會執行。現在使用 2018-05-29 時，我們可以處理此案例。例如，我們可以選擇引發授權錯誤如下：

```
# throw an unauthorized error if the result is null
#if ( $util.isNull($ctx.result) )
    $util.unauthorized()
#end
$util.toJson($ctx.result)
```

注意：從資料來源發出的錯誤有時並不嚴重，或甚至早有預期，這說明了為什麼回應映射範本應該可靈活處理呼叫錯誤，並決定是否予以忽略、重新引發，或是擲出不同的錯誤。

假設我們使用下列的回應映射範本：

```
$util.toJson($ctx.result)
```

過去使用 2017-02-28 時，如果發生呼叫錯誤，回應映射範本將會進行評估，而且結果會自動放置在 GraphQL 回應的 `errors` 區塊。現在使用 2018-05-29 時，我們可以選擇如何處理錯誤、重新引發錯誤、引發不同的錯誤，或在傳回資料時附加該錯誤。

再次引發呼叫錯誤

使用下列回應範本時，我們會引發資料來源發出的相同錯誤。

```
#if ( $ctx.error )
    $util.error($ctx.error.message, $ctx.error.type)
#end
$util.toJson($ctx.result)
```

如果發生呼叫錯誤 (例如，`$ctx.error` 存在時) 回應看起來將如下所示：

```
{
  "data": {
    "getPost": null
  },
  "errors": [
    {
      "path": [
        "getPost"
      ],
```



```

        "errorType": "DynamoDB:ConditionalCheckFailedException",
        "message": "Conditional check failed exception...",
        "locations": [
            {
                "line": 5,
                "column": 5
            }
        ]
    }
}

```

引發不同的錯誤

使用下列回應範本時，我們會在處理資料來源發出錯誤之後引發自訂的錯誤。

```

#if ( $ctx.error )
    #if ( $ctx.error.type.equals("ConditionalCheckFailedException") )
        ## we choose here to change the type and message of the error for
        ConditionalCheckFailedExceptions
        $util.error("Error while updating the post, try again. Error:
$ctx.error.message", "UpdateError")
    #else
        $util.error($ctx.error.message, $ctx.error.type)
    #end
#end
$util.toJson($ctx.result)

```

如果發生呼叫錯誤 (例如，\$ctx.error 存在時) 回應看起來將如下所示：

```

{
  "data": {
    "getPost": null
  },
  "errors": [
    {
      "path": [
        "getPost"
      ],
      "errorType": "UpdateError",
      "message": "Error while updating the post, try again. Error: Conditional
check failed exception...",
    }
  ]
}

```

```

        "locations": [
          {
            "line": 5,
            "column": 5
          }
        ]
      }
    ]
  }
}

```

附加錯誤至傳回資料

使用下列回應範本時，當透過回應傳回資料時，我們會同時附加由資料來源發出的相同錯誤。這也稱為部分回應。

```

#if ( $ctx.error )
  $util.appendError($ctx.error.message, $ctx.error.type)
  #set($defaultPost = {id: "1", title: 'default post'})
  $util.toJson($defaultPost)
#else
  $util.toJson($ctx.result)
#end

```

如果發生呼叫錯誤 (例如，`$ctx.error` 存在時) 回應看起來將如下所示：

```

{
  "data": {
    "getPost": {
      "id": "1",
      "title": "A post"
    }
  },
  "errors": [
    {
      "path": [
        "getPost"
      ],
      "errorType": "ConditionalCheckFailedException",
      "message": "Conditional check failed exception...",
      "locations": [
        {
          "line": 5,

```

```

    "column": 5
  }
]
}

```

從 2017-02-28 遷移到 2018-05-29

從 2017-02-28 遷移到 2018-05-29 的過程相當簡單。變更在解析程式要求映射範本或函數版本物件上的版本欄位。不過，請注意，2018-05-29 的執行行為不同於 2017-02-28，其中變更已概述於[此處](#)。

保留 2017-02-28 到 2018-05-29 的相同執行行為

在某些情況下，在執行 2017-02-28 版本控制的範本時，可以保留與 2018-05-29 版本相同的執行行為。

範例：DynamoDB PutItem

假設使用下列 2017-02-28 DynamoDB PutItem 要求範本：

```

{
  "version" : "2017-02-28",
  "operation" : "PutItem",
  "key": {
    "foo" : ... typed value,
    "bar" : ... typed value
  },
  "attributeValues" : {
    "baz" : ... typed value
  },
  "condition" : {
    ...
  }
}

```

並使用下列回應範本：

```
$util.toJson($ctx.result)
```

遷移到 2018-05-29，這些範本變更如下：

```
{
  "version" : "2018-05-29", ## Note the new 2018-05-29 version
  "operation" : "PutItem",
  "key": {
    "foo" : ... typed value,
    "bar" : ... typed value
  },
  "attributeValues" : {
    "baz" : ... typed value
  },
  "condition" : {
    ...
  }
}
```

而且回應範本變更如下：

```
## If there is a datasource invocation error, we choose to raise the same error
## the field data will be set to null.
#if($ctx.error)
  $util.error($ctx.error.message, $ctx.error.type, $ctx.result)
#end

## If the data source invocation is null, we return null.
#if($util.isNull($ctx.result))
  #return
#end

$util.toJson($ctx.result)
```

現在，錯誤處理將由您負責，我們選擇使用 `$util.error()` 引發從 DynamoDB 傳回的相同錯誤。您可以調整這個片段，將您的映射範本轉換成 2018-05-29，請注意，如果您使用不同的回應範本，則您必須處理執行行為變更。

範例：DynamoDB GetItem

假設有下列 2017-02-28 DynamoDB GetItem 請求範本：

```
{
  "version" : "2017-02-28",
  "operation" : "GetItem",
  "key" : {
```

```
        "foo" : ... typed value,
        "bar" : ... typed value
    },
    "consistentRead" : true
}
```

並使用下列回應範本：

```
## map table attribute postId to field Post.id
$util.qr($ctx.result.put("id", $ctx.result.get("postId")))

$util.toJson($ctx.result)
```

遷移到 2018-05-29，這些範本變更如下：

```
{
  "version" : "2018-05-29", ## Note the new 2018-05-29 version
  "operation" : "GetItem",
  "key" : {
    "foo" : ... typed value,
    "bar" : ... typed value
  },
  "consistentRead" : true
}
```

而且回應範本變更如下：

```
## If there is a datasource invocation error, we choose to raise the same error
#if($ctx.error)
  $util.error($ctx.error.message, $ctx.error.type)
#end

## If the data source invocation is null, we return null.
#if($util.isNull($ctx.result))
  #return
#end

## map table attribute postId to field Post.id
$util.qr($ctx.result.put("id", $ctx.result.get("postId")))

$util.toJson($ctx.result)
```

使用 2017-02-28 版本時，如果資料來源呼叫為 `null`，表示 DynamoDB 資料表中沒有符合索引鍵的項目，而回應映射範本將不會執行。這種做法應該適用於大部分的案例，但是如果您希望 `$ctx.result` 不要是 `null`，現在您必須負責處理該案例。

2017-02-28

特性

- 如果資料來源呼叫結果是 `null`，回應映射範本將不會執行。
- 如果資料來源呼叫產生錯誤，您現在要負責處理錯誤，而回應映射範本會執行，且評估結果會置放在 GraphQL 回應 `errors.data` 區塊中。

GraphQL 類型參考

GraphQL 中的純量類型代表 GraphQL 結構描述中的基本分葉值。這些是解析為單一值的最基本資料類型。與物件類型不同，純量類型不能有子欄位。GraphQL 隨附一組預設純量類型：

- Int：帶正負號的 32 位元整數
- 浮點數：帶正負號的雙精度浮點值
- 字串：UTF-8 字元序列
- 布林值：true 或 false 值
- ID：唯一識別符，通常用於重新擷取物件或作為快取的金鑰

這些純量類型可做為結構描述中更複雜類型的建置區塊。它們用於定義包含簡單、單一值的欄位。除了這些內建純量之外，還為不同的使用案例 AWS AppSync 提供額外的純量。

GraphQL 中的界面和聯集是抽象類型，允許彈性和可擴展的結構描述設計。它們提供將相關類型分組並啟用多態查詢的機制。GraphQL 中的界面是一種抽象類型，可定義類型必須包含的一組欄位，以實作界面。它透過指定實作類型必須擁有的一組常用欄位，做為物件的合約。當您想要傳回可有多種不同類型的物件或欄位，但仍有一些保證欄位時，界面非常有用。相較之下，GraphQL 中的 Union 代表的類型可以是多種物件類型之一，但不會在這些類型之間定義任何常用欄位。當您需要傳回可具有多種類型的欄位時，聯集會很有幫助，而且這些類型不一定共用共同的欄位。界面和聯集在欄位可能傳回不同資料類型的情況下特別有用，可讓用戶端根據傳回的類型查詢特定欄位。

本節用作結構描述類型的參考。

主題

- [GraphQL 中的純量類型](#)
- [GraphQL 中的界面和聯集](#)

GraphQL 中的純量類型

GraphQL 物件類型具有名稱和欄位，而且這些欄位可以具有子欄位。最終，物件類型的欄位必須解析為純量類型，代表查詢的葉子。如需物件類型和純量的詳細資訊，請參閱 GraphQL 網站上的[結構描述和類型](#)。

除了預設的一組 GraphQL 純量之外，AWS AppSync 也可讓您使用以 AWS 字首開頭的服務定義純量。AWS AppSync 不支援建立使用者定義（自訂）純量。您必須使用預設 AWS 或純量。

您無法使用 AWS 做為自訂物件類型的字首。

下節是結構描述類型參考。

預設純量

GraphQL 定義下列預設純量：

預設純量清單

ID

物件的唯一識別符。此純量會序列化為 `String` 但並非供人閱讀。

String

UTF-8 字元序列。

Int

介於 $-(2^{31})$ 和 $2^{31}-1$ 之間的整數值。

Float

IEEE 754 浮點值。

Boolean

布林值，`true` 或 `false`。

AWS AppSync 純量

AWS AppSync 定義下列純量：

AWS AppSync 純量清單

AWSDate

格式為 `YYYY-MM-DD` 的擴充 [ISO 8601 日期](#) 字串。

AWSTime

格式為 `hh:mm:ss.sss` 的擴充 [ISO 8601 時間](#) 字串。

AWSDateTime

擴充 [ISO 8601 日期和時間](#) 字串，格式為 `YYYY-MM-DDThh:mm:ss.sssZ`。

Note

AWSDDate、AWSTime和 AWSDateTime 純量可以選擇性地包含[時區位移](#)。例如，值 1970-01-01Z、1970-01-01-07:00和 1970-01-01+05:30 都對有效AWSDDate。時區位移必須是 Z(UTC) 或以小時和分鐘為單位的位移（以及選擇性的秒數）。例如：±hh:mm:ss。時區位移中的秒欄位被視為有效，即使它不是 ISO 8601 標準的一部分。

AWSTimestamp

整數值，代表 前後的秒數1970-01-01-T00:00Z。

AWSEmail

採用 [RFC 822](#) local-part@domain-part所定義格式的電子郵件地址。

AWSJSON

JSON 字串。任何有效的 JSON 建構模組都會自動剖析並載入解析程式碼，做為映射、清單或純量值，而非字串輸入字串。未引用的字串或其他無效的 JSON 會導致 GraphQL 驗證錯誤。

AWSPhone

電話號碼。此值會儲存為字串。電話號碼可以包含空格或連字號，以分隔數字群組。沒有國家/地區代碼的電話號碼假設為遵守[北美編號計劃 \(NANP\)](#) 的美國/北美電話號碼。

AWSURL

[RFC 1738](#) 定義的 URL。例如 https://www.amazon.com/dp/B000NZW3KC/ 或 mailto:example@example.com。URLs 必須包含結構描述 (http、mailto)，且路徑部分中不能包含兩個正斜線 (//)。

AWSIPAddress

有效的 IPv4 或 IPv6 地址。IPv4 地址預期為四點符號 (123.12.34.56)。IPv6 地址預期為非括號、冒號分隔格式 (1a2b:3c4b::1234:4567)。您可以包含選用的 CIDR 尾碼 (123.45.67.89/16) 來表示子網路遮罩。

結構描述用量範例

下列範例 GraphQL 結構描述使用所有自訂純量做為「物件」，並顯示基本 put、get 和 list 操作的解析程式請求和回應範本。最後，此範例示範如何在執行查詢和變動時使用此功能。

```
type Mutation {
  putObject(
    email: AWSEmail,
    json: AWSJSON,
    date: AWSDate,
    time: AWSTime,
    datetime: AWSDateTime,
    timestamp: AWSTimestamp,
    url: AWSURL,
    phoneno: AWSPhone,
    ip: AWSIPAddress
  ): Object
}

type Object {
  id: ID!
  email: AWSEmail
  json: AWSJSON
  date: AWSDate
  time: AWSTime
  datetime: AWSDateTime
  timestamp: AWSTimestamp
  url: AWSURL
  phoneno: AWSPhone
  ip: AWSIPAddress
}

type Query {
  getObject(id: ID!): Object
  listObjects: [Object]
}

schema {
  query: Query
  mutation: Mutation
}
```

以下是 的請求範本putObject可能看起來的樣子。putObject 使用 PutItem操作來建立或更新 Amazon DynamoDB 資料表中的項目。請注意，此程式碼片段沒有已設定的 Amazon DynamoDB 資料表做為資料來源。這只是用作範例：

```
{
```

```
"version" : "2017-02-28",
"operation" : "PutItem",
"key" : {
  "id": $util.dynamodb.toDynamoDBJson($util.autoId()),
},
"attributeValues" : $util.dynamodb.toMapValuesJson($ctx.args)
}
```

的回應範本會putObject傳回結果：

```
$util.toJson($ctx.result)
```

以下是 的請求範本getObject可能看起來的樣子。getObject 使用 GetItem操作來傳回指定主索引鍵之項目的一組屬性。請注意，此程式碼片段沒有已設定的 Amazon DynamoDB 資料表做為資料來源。這只是用作範例：

```
{
  "version": "2017-02-28",
  "operation": "GetItem",
  "key": {
    "id": $util.dynamodb.toDynamoDBJson($ctx.args.id),
  }
}
```

的回應範本會getObject傳回結果：

```
$util.toJson($ctx.result)
```

以下是 的請求範本listObjects可能看起來的樣子。listObjects 使用 Scan操作來傳回一或多個項目和屬性。請注意，此程式碼片段沒有已設定的 Amazon DynamoDB 資料表做為資料來源。這只是用作範例：

```
{
  "version" : "2017-02-28",
  "operation" : "Scan",
}
```

的回應範本會listObjects傳回結果：

```
$util.toJson($ctx.result.items)
```

以下是搭配 GraphQL 查詢使用此結構描述的一些範例：

```
mutation CreateObject {
  putObject(email: "example@example.com"
    json: "{\"a\":1, \"b\":3, \"string\": 234}"
    date: "1970-01-01Z"
    time: "12:00:34."
    datetime: "1930-01-01T16:00:00-07:00"
    timestamp: -123123
    url:"https://amazon.com"
    phoneno: "+1 555 764 4377"
    ip: "127.0.0.1/8"
  ) {
    id
    email
    json
    date
    time
    datetime
    url
    timestamp
    phoneno
    ip
  }
}

query getObject {
  getObject(id:"0d97daf0-48e6-4ffc-8d48-0537e8a843d2"){
    email
    url
    timestamp
    phoneno
    ip
  }
}

query listObjects {
  listObjects {
    json
    date
    time
    datetime
  }
}
```

```
}
```

GraphQL 中的界面和聯集

GraphQL 類型系統支援[界面](#)。界面會公開特定的欄位組合，類型必須包含該組合以實作界面。

GraphQL 類型系統也支援[聯集](#)。聯集與界面相同，但它們不會定義一組常用的欄位。當可能的類型不共用邏輯階層時，使用者通常偏好使用聯集而非界面。

下節是結構描述類型參考。

介面範例

我們可以代表任何類型的活動或人員集合的Event界面。有些可能的事件類型為 Concert、Conference和 Festival。這些類型都具有常見的特性，包含名稱、事件進行的地點，以及開始和結束日期。這些類型也有差異；Conference提供發言者和研討會的清單，而 Concert具有表演樂隊。

在結構描述定義語言 (SDL) 中，Event界面定義如下：

```
interface Event {  
    id: ID!  
    name : String!  
    startsAt: String  
    endsAt: String  
    venue: Venue  
    minAgeRestriction: Int  
}
```

每個類型都會實作Event界面，如下所示：

```
type Concert implements Event {  
    id: ID!  
    name: String!  
    startsAt: String  
    endsAt: String  
    venue: Venue  
    minAgeRestriction: Int  
    performingBand: String  
}
```

```
type Festival implements Event {
  id: ID!
  name: String!
  startsAt: String
  endsAt: String
  venue: Venue
  minAgeRestriction: Int
  performers: [String]
}

type Conference implements Event {
  id: ID!
  name: String!
  startsAt: String
  endsAt: String
  venue: Venue
  minAgeRestriction: Int
  speakers: [String]
  workshops: [String]
}
```

介面可用來代表元素，而該元素可能有幾個類型。例如，我們可以搜尋在特定會場發生的所有事件。現在讓我們在結構描述上新增 `findEventsByVenue` 欄位，如下所示：

```
schema {
  query: Query
}

type Query {
  # Retrieve Events at a specific Venue
  findEventsAtVenue(venueId: ID!): [Event]
}

type Venue {
  id: ID!
  name: String
  address: String
  maxOccupancy: Int
}

type Concert implements Event {
  id: ID!
  name: String!
```

```
    startsAt: String
    endsAt: String
    venue: Venue
    minAgeRestriction: Int
    performingBand: String
  }

interface Event {
  id: ID!
  name: String!
  startsAt: String
  endsAt: String
  venue: Venue
  minAgeRestriction: Int
}

type Festival implements Event {
  id: ID!
  name: String!
  startsAt: String
  endsAt: String
  venue: Venue
  minAgeRestriction: Int
  performers: [String]
}

type Conference implements Event {
  id: ID!
  name: String!
  startsAt: String
  endsAt: String
  venue: Venue
  minAgeRestriction: Int
  speakers: [String]
  workshops: [String]
}
```

`findEventsByVenue` 會傳回的清單 `Event`。由於 GraphQL 界面欄位對於所有實作類型而言都是共通的，因此您可以在 `Event` 界面 (`id`、`name`、`startsAt`、`endsAt`、`venue` 和 `minAgeRestriction`) 選取任何欄位。此外，您可以使用 GraphQL [片段](#) 存取任何實作類型上的欄位，只是您必須指定該類型。

讓我們檢查使用 界面的 GraphQL 查詢範例。

```
query {  
  findEventsAtVenue(venueId: "Madison Square Garden") {  
    id  
    name  
    minAgeRestriction  
    startsAt  
  
    ... on Festival {  
      performers  
    }  
  
    ... on Concert {  
      performingBand  
    }  
  
    ... on Conference {  
      speakers  
      workshops  
    }  
  }  
}
```

上述查詢會產生一個結果清單，伺服器預設會依據開始日期排序事件。

```
{  
  "data": {  
    "findEventsAtVenue": [  
      {  
        "id": "Festival-2",  
        "name": "Festival 2",  
        "minAgeRestriction": 21,  
        "startsAt": "2018-10-05T14:48:00.000Z",  
        "performers": [  
          "The Singers",  
          "The Screamers"  
        ]  
      },  
      {  
        "id": "Concert-3",  
        "name": "Concert 3",  
        "minAgeRestriction": 18,  
        "startsAt": "2018-10-07T14:48:00.000Z",  
        "performingBand": "The Jumpers"  
      }  
    ]  
  }  
}
```



```

    },
    {
      "id": "Conference-4",
      "name": "Conference 4",
      "minAgeRestriction": null,
      "startsAt": "2018-10-09T14:48:00.000Z",
      "speakers": [
        "The Storytellers"
      ],
      "workshops": [
        "Writing",
        "Reading"
      ]
    }
  ]
}
}
}

```

由於結果會以單一事件集合的形式傳回，因此使用界面來表示常見特性對於排序結果非常有用。

聯集範例

如前所述，聯集不會定義一般欄位集。搜尋結果可能代表許多不同的類型。使用 Event 結構描述，您可以定義 SearchResult 聯集，如下所示：

```

type Query {
  # Retrieve Events at a specific Venue
  findEventsAtVenue(venueId: ID!): [Event]
  # Search across all content
  search(query: String!): [SearchResult]
}

union SearchResult = Conference | Festival | Concert | Venue

```

在此情況下，若要查詢聯SearchResult集上的任何欄位，您必須使用片段：

```

query {
  search(query: "Madison") {
    ... on Venue {
      id
      name
      address
    }
  }
}

```

```
... on Festival {
  id
  name
  performers
}

... on Concert {
  id
  name
  performingBand
}

... on Conference {
  speakers
  workshops
}
}
```

類型解析 in AWS AppSync

類型解析是一種機制，GraphQL 引擎會透過這個機制將解析值識別為特定物件類型。

返回聯集搜尋範例，假設查詢產生結果，結果清單中的每個項目都必須顯示為 `SearchResult` 聯集定義的可能類型之一（即 `Conference`、`Concert`、`Festival` 或 `Venue`）。

由於識別 `Festival`、`Venue` 或 `Conference` 的邏輯取決於應用程式要求，因此必須為 GraphQL 引擎提供提示，以便從原始結果中識別我們的類型。

With AWS AppSync，此提示由名為 `__typename` 的中繼欄位表示，其值對應於識別的物件類型名稱。屬於介面或聯集的傳回類型 `__typename` 需要。

類型解析度範例

讓我們重複使用之前的結構描述。您可以透過瀏覽到主控台並將以下項目新增至 Schema (結構描述) 頁面下來執行此動作：

```
schema {
  query: Query
}

type Query {
```

```
# Retrieve Events at a specific Venue
findEventsAtVenue(venueId: ID!): [Event]
# Search across all content
search(query: String!): [SearchResult]
}

union SearchResult = Conference | Festival | Concert | Venue

type Venue {
  id: ID!
  name: String!
  address: String
  maxOccupancy: Int
}

interface Event {
  id: ID!
  name: String!
  startsAt: String
  endsAt: String
  venue: Venue
  minAgeRestriction: Int
}

type Festival implements Event {
  id: ID!
  name: String!
  startsAt: String
  endsAt: String
  venue: Venue
  minAgeRestriction: Int
  performers: [String]
}

type Conference implements Event {
  id: ID!
  name: String!
  startsAt: String
  endsAt: String
  venue: Venue
  minAgeRestriction: Int
  speakers: [String]
  workshops: [String]
}
```

```
type Concert implements Event {  
  id: ID!  
  name: String!  
  startsAt: String  
  endsAt: String  
  venue: Venue  
  minAgeRestriction: Int  
  performingBand: String  
}
```

現在將解析程式附加到 `Query.search` 欄位。在 `Resolvers` 區段中，選擇連接，建立新的 `NONE` 類型的資料來源，然後命名為 `StubDataSource`。在此範例中，我們將假設從外部來源擷取結果，並將擷取結果硬式編碼至請求映射範本中。

在請求映射範本窗格中，輸入以下內容：

```
{  
  "version" : "2018-05-29",  
  "payload":  
    ## We are effectively mocking our search results for this example  
    [  
      {  
        "id": "Venue-1",  
        "name": "Venue 1",  
        "address": "2121 7th Ave, Seattle, WA 98121",  
        "maxOccupancy": 1000  
      },  
      {  
        "id": "Festival-2",  
        "name": "Festival 2",  
        "performers": ["The Singers", "The Screammers"]  
      },  
      {  
        "id": "Concert-3",  
        "name": "Concert 3",  
        "performingBand": "The Jumpers"  
      },  
      {  
        "id": "Conference-4",  
        "name": "Conference 4",  
        "speakers": ["The Storytellers"],  
        "workshops": ["Writing", "Reading"]  
      }  
    ]  
}
```

```
    }
  ]
}
```

如果應用程式傳回類型名稱做為 id 欄位的一部分，則類型解析邏輯必須剖析 id 欄位以擷取類型名稱，然後將 __typename 欄位新增至每個結果。您可以在回應映射範本中執行該邏輯，如下所示：

Note

如果您使用的是 Lambda 資料來源，您也可以在此 Lambda 函數中執行此任務。

```
#foreach ($result in $context.result)
  ## Extract type name from the id field.
  #set( $typeName = $result.id.split("-")[0] )
  #set( $ignore = $result.put("__typename", $typeName))
#end
$util.toJson($context.result)
```

執行下列查詢：

```
query {
  search(query: "Madison") {
    ... on Venue {
      id
      name
      address
    }

    ... on Festival {
      id
      name
      performers
    }

    ... on Concert {
      id
      name
      performingBand
    }
  }
}
```

```
... on Conference {  
  speakers  
  workshops  
}  
}  
}
```

此查詢會產生下列結果：

```
{  
  "data": {  
    "search": [  
      {  
        "id": "Venue-1",  
        "name": "Venue 1",  
        "address": "2121 7th Ave, Seattle, WA 98121"  
      },  
      {  
        "id": "Festival-2",  
        "name": "Festival 2",  
        "performers": [  
          "The Singers",  
          "The Screammers"  
        ]  
      },  
      {  
        "id": "Concert-3",  
        "name": "Concert 3",  
        "performingBand": "The Jumpers"  
      },  
      {  
        "speakers": [  
          "The Storytellers"  
        ],  
        "workshops": [  
          "Writing",  
          "Reading"  
        ]  
      }  
    ]  
  }  
}
```

類型解析邏輯會依據應用程式而有不同。例如，您可以有不同的識別邏輯，以檢查特定欄位或甚至結合欄位是否存在。也就是說，您可以偵測 `performers` 欄位是否存在以識別 `Festival`，或偵測 `speakers` 與 `workshops` 欄位的組合是否存在以識別 `Conference`。最終，由您決定要使用的邏輯。

故障診斷和 中的常見錯誤 AWS AppSync

本區段說明一些常見的錯誤，以及如何排除這些問題。

不正確的 DynamoDB 索引鍵映射

如果您的 GraphQL 操作傳回下列錯誤訊息，可能是因為您的請求映射範本結構不符合 Amazon DynamoDB 金鑰結構：

```
The provided key element does not match the schema (Service: AmazonDynamoDBv2; Status Code: 400; Error Code
```

例如，如果您的 DynamoDB 資料表有一個名為 的雜湊索引鍵，"id"而您的範本顯示 "PostID"，如下列範例所示，這會導致上述錯誤，因為 "id" 不符合 "PostID"。

```
{
  "version" : "2017-02-28",
  "operation" : "GetItem",
  "key" : {
    "PostID" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
  }
}
```

缺少解析程式

如果您執行查詢等 GraphQL 操作，但是獲得 null 回應，這可能是因為您並未設定解析程式。

如果匯入的結構描述定義了 `getCustomer(userId: ID!)`：欄位，但並未針對此欄位設定解析程式，則當您執行像是 `getCustomer(userId:"ID123"){...}` 的查詢時，將會獲得類似於下列的回應：

```
{
  "data": {
    "getCustomer": null
  }
}
```


映射範本錯誤

如果未正確設定映射範本，您將會收到 GraphQL 回應，其中的 `errorType` 為 `MappingTemplate`。message 欄位應該會指出映射範本的問題所在。

例如，如果請求映射範本未包含 `operation` 欄位，或是 `operation` 欄位的名稱不正確，則您將會收到類似下列的回應：

```
{
  "data": {
    "searchPosts": null
  },
  "errors": [
    {
      "path": [
        "searchPosts"
      ],
      "errorType": "MappingTemplate",
      "locations": [
        {
          "line": 2,
          "column": 3
        }
      ],
      "message": "Value for field '$[operation]' not found."
    }
  ]
}
```

不正確的傳回類型

資料來源所傳來的傳回類型，必須符合結構描述中所定義的物件類型，否則將會出現 GraphQL 錯誤，例如：

```
"errors": [
  {
    "path": [
      "posts"
    ],
    "locations": null,
```

```
    "message": "Can't resolve value (/posts) : type mismatch error, expected type LIST,
    got OBJECT"
  }
]
```

例如，下列的查詢定義可能會產生這種錯誤：

```
type Query {
  posts: [Post]
}
```

查詢式期望的結果是 [Posts] 物件的清單 (LIST)。例如，如果有使用 Node.JS 的 Lambda 函式，包含類似於下列的程式碼：

```
const result = { data: data.Items.map(item => { return item ; }) };
callback(err, result);
```

這將會丟出錯誤，因為 `result` 是一個物件。您需要將回呼變更為 `result.data`，或是將結構描述改為不傳回清單 (LIST)。

處理無效的請求

當 AWS AppSync 無法處理並傳送請求（由於不正確的資料，例如無效的語法）到欄位解析程式時，回應承載會傳回值設為 `null` 的欄位資料，`null` 以及任何相關的錯誤。

本文為英文版的機器翻譯版本，如內容有任何歧義或不一致之處，概以英文版為準。