aws

Guida per gli sviluppatori

# AWS SDK per la crittografia del database

# AWS SDK per la crittografia del database: Guida per gli sviluppatori

# Table of Contents

# Cos'è il AWS Database Encryption SDK?

La nostra libreria di crittografia lato client è stata rinominata Database Encryption SDK. AWS Questa guida per sviluppatori fornisce ancora informazioni sul DynamoDB Encryption Client.

Il AWS Database Encryption SDK è un insieme di librerie software che consentono di includere la crittografia lato client nella progettazione del database. Il AWS Database Encryption SDK fornisce soluzioni di crittografia a livello di record. Specificate quali campi sono crittografati e quali campi sono inclusi nelle firme che garantiscono l'autenticità dei dati. La crittografia dei dati sensibili in transito e in archivio aiuta a garantire che i dati in formato testo non crittografato non siano disponibili per terze parti, tra cui. AWS Il AWS Database Encryption SDK è fornito gratuitamente con la licenza Apache 2.0.

Questa guida per sviluppatori fornisce una panoramica concettuale del AWS Database Encryption SDK, inclusa un'introduzione alla sua architettura, dettagli su come protegge i dati, su come si differenzia dalla crittografia lato server e indicazioni sulla selezione dei componenti critici per l'applicazione per aiutarti a iniziare.

Il AWS Database Encryption SDK supporta Amazon DynamoDB con crittografia a livello di attributo.

Il AWS Database Encryption SDK offre i seguenti vantaggi:

Progettato appositamente per le applicazioni di database

Non è necessario essere un esperto di crittografia per utilizzare il AWS Database Encryption SDK. Le implementazioni includono metodi di supporto progettati per funzionare con le applicazioni esistenti.

Dopo aver creato e configurato i componenti richiesti, il client di crittografia crittografa e firma in modo trasparente i record quando li aggiungi a un database e li verifica e li decrittografa quando li recuperi.

Include la funzione di firma e crittografia sicure

Il AWS Database Encryption SDK include implementazioni sicure che crittografano i valori dei campi in ogni record utilizzando una chiave di crittografia dei dati unica e quindi firmano il record per proteggerlo da modifiche non autorizzate, come l'aggiunta o l'eliminazione di campi o lo scambio di valori crittografati.

Utilizza i materiali crittografici provenienti da qualsiasi origine

Il AWS Database Encryption SDK utilizza i portachiavi per generare, crittografare e decrittografare l'esclusiva chiave di crittografia dei dati che protegge il record. I portachiavi determinano le chiavi di avvolgimento che crittografano quella chiave di dati.

È possibile utilizzare chiavi di wrapping da qualsiasi fonte, inclusi i servizi di crittografia, come () o. AWS Key Management ServiceAWS KMSAWS CloudHSM Il AWS Database Encryption SDK non richiede alcun servizio Account AWS . AWS

Support per la memorizzazione nella cache dei materiali crittografici

Il portachiavi AWS KMS Hierarchical è una soluzione di memorizzazione nella cache dei materiali crittografici che riduce il numero di AWS KMS chiamate utilizzando chiavi branch AWS KMS protette persistenti in una tabella Amazon DynamoDB e quindi memorizzando nella cache locale i materiali chiave delle branch utilizzati nelle operazioni di crittografia e decrittografia. Ti consente di proteggere i tuoi materiali crittografici con una chiave KMS di crittografia simmetrica senza chiamare ogni volta che crittografi o decrittografi un record. AWS KMS Il portachiavi AWS KMS Hierarchical è un'ottima scelta per le applicazioni che devono ridurre al minimo le chiamate a. AWS KMS

Crittografia ricercabile

È possibile progettare database in grado di eseguire ricerche nei record crittografati senza decrittografare l'intero database. A seconda del modello di minaccia e dei requisiti di interrogazione, è possibile utilizzare la crittografia ricercabile per eseguire ricerche con corrispondenza esatta o query complesse più personalizzate sul database crittografato.

Support per schemi di database multitenant

Il AWS Database Encryption SDK consente di proteggere i dati archiviati nei database con uno schema condiviso isolando ogni tenant con materiali di crittografia distinti. Se più utenti eseguono operazioni di crittografia all'interno del database, utilizza uno dei AWS KMS portachiavi per fornire a ciascun utente una chiave distinta da utilizzare nelle proprie operazioni crittografiche. Per ulteriori informazioni, consulta Lavorare con database multitenant.

Support per aggiornamenti dello schema senza interruzioni

Quando configuri il AWS Database Encryption SDK, fornisci azioni crittografiche che indicano al client quali campi crittografare e firmare, quali campi firmare (ma non crittografare) e quali ignorare. Dopo aver utilizzato AWS Database Encryption SDK per proteggere i record, puoi

comunque apportare modifiche al tuo modello di dati. Puoi aggiornare le tue azioni crittografiche, come l'aggiunta o la rimozione di campi crittografati, in un'unica implementazione.

## Sviluppato in repository open source

Il AWS Database Encryption SDK è sviluppato in repository open source su. GitHub Puoi utilizzare questi repository per visualizzare il codice, leggere e inviare problemi e trovare informazioni specifiche sulla tua implementazione.

L'SDK per la crittografia dei AWS database per DynamoDB

- Il repository aws-database-encryption-sdk-dynamodb su GitHub supporta le versioni più recenti di AWS Database Encryption SDK per DynamoDB in Java, .NET e Rust.

  Il AWS Database Encryption SDK per DynamoDB è un prodotto di Dafny, un linguaggio che supporta la verifica in cui si scrivono le specifiche, il codice per implementarle e le bozze per testarle. Il risultato è una libreria che implementa le funzionalità del AWS Database Encryption SDK per DynamoDB in un framework che garantisce la correttezza funzionale.

## Support e manutenzione

AWS Database Encryption SDK utilizza la stessa politica di manutenzione utilizzata dall' AWS SDK e dagli strumenti, comprese le fasi di controllo delle versioni e del ciclo di vita. Come best practice, ti consigliamo di utilizzare l'ultima versione disponibile di AWS Database Encryption SDK per l'implementazione del database e di eseguire l'aggiornamento non appena vengono rilasciate nuove versioni.

Per ulteriori informazioni, consulta la politica di manutenzione di AWS SDKs and Tools nella Guida di riferimento agli strumenti AWS SDKs e agli strumenti.

## Invio di feedback

Apprezziamo il tuo feedback. Se hai una domanda, un commento o un problema da segnalare, utilizza le seguenti risorse.

Se scopri una potenziale vulnerabilità di sicurezza nel AWS Database Encryption SDK, avvisa AWS la sicurezza. Non creare un problema pubblico GitHub .

Per fornire feedback su questa documentazione, utilizzare il link di feedback in qualsiasi pagina.

# AWS Concetti dell'SDK per la crittografia dei database

> La nostra libreria di crittografia lato client è stata rinominata Database Encryption SDK. AWS Questa guida per sviluppatori fornisce ancora informazioni sul DynamoDB Encryption Client.

Questo argomento spiega i concetti e la terminologia utilizzati nel AWS Database Encryption SDK.

Per informazioni su come interagiscono i componenti del AWS Database Encryption SDK, consulta. Come funziona il AWS Database Encryption SDK

Per ulteriori informazioni su AWS Database Encryption SDK, consulta i seguenti argomenti.

- Scopri come AWS Database Encryption SDK utilizza la crittografia a busta per proteggere i tuoi dati.
- Scopri gli elementi della crittografia in busta: le chiavi dati che proteggono i tuoi record e le chiavi di avvolgimento che proteggono le tue chiavi dati.
- Scopri i portachiavi che determinano le chiavi di avvolgimento da utilizzare.
- Scopri il contesto di crittografia che aggiunge integrità al tuo processo di crittografia.
- Scopri la descrizione del materiale che i metodi di crittografia aggiungono al tuo record.
- Scopri le azioni crittografiche che indicano al AWS Database Encryption SDK quali campi crittografare e firmare.

Argomenti

- Responsabile di materiali crittografici
- Crittografia simmetrica e asimmetrica
- Impegno chiave
- Firme digitali

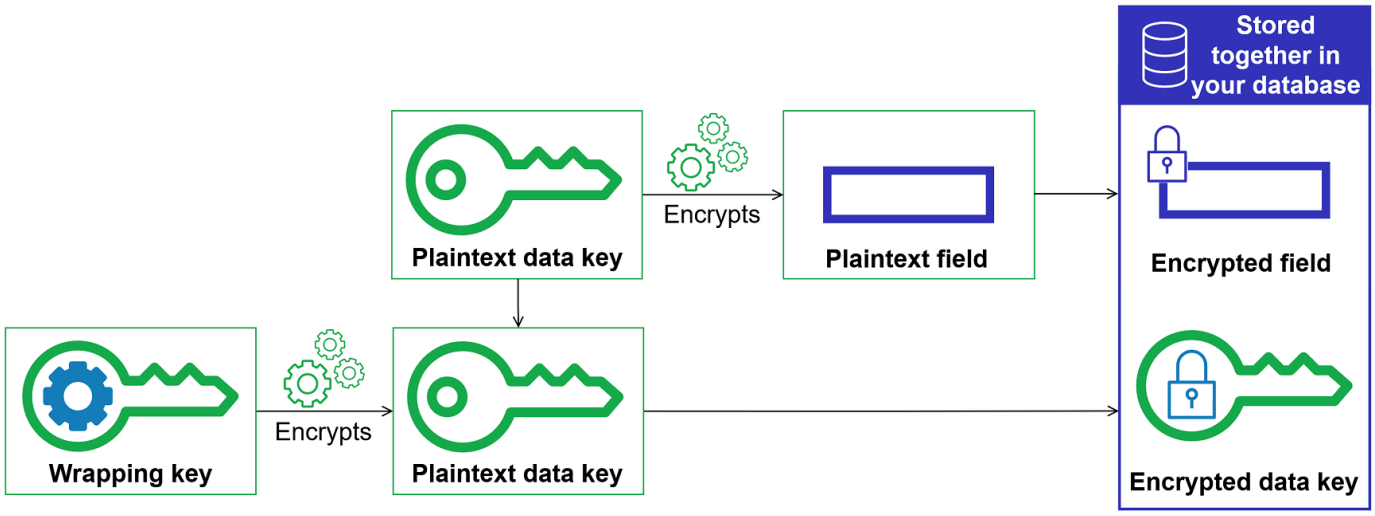
## Crittografia envelope

La sicurezza dei dati crittografati dipende in parte dalla protezione della chiave di dati che può decrittarli. Una best practice accettata per la protezione della chiave di dati è crittografarla. A tale scopo, è necessaria un'altra chiave di crittografia, nota come chiave di crittografia a chiave o chiave di wrapping. La pratica di utilizzare una chiave di wrapping per crittografare le chiavi di dati è nota come crittografia a busta.

Protezione delle chiavi dei dati

Il AWS Database Encryption SDK crittografa ogni campo con una chiave dati unica. Quindi crittografa ogni chiave di dati sotto la chiave di wrapping specificata. Memorizza le chiavi di dati crittografate nella descrizione del materiale.

Per specificare la chiave di imballaggio, si utilizza un portachiavi.




Crittografia degli stessi dati con più chiavi di wrapping

È possibile crittografare la chiave dati con più chiavi di wrapping. Potresti voler fornire chiavi di avvolgimento diverse per utenti diversi, oppure chiavi di avvolgimento di tipi diversi o in posizioni diverse. Ciascuna delle chiavi di wrapping crittografa la stessa chiave di dati. Il AWS Database

Encryption SDK memorizza tutte le chiavi di dati crittografate insieme ai campi crittografati nella descrizione del materiale.

Per decrittografare i dati, è necessario fornire almeno una chiave di wrapping in grado di decrittografare le chiavi di dati crittografate.

Abbinare i punti di forza di più algoritmi

Per crittografare i dati, per impostazione predefinita, AWS Database Encryption SDK utilizza una suite di algoritmi con crittografia simmetrica AES-GCM, una funzione di derivazione delle chiavi basata su HMAC (HKDF) e firma ECDSA. Per crittografare la chiave dati, puoi specificare un algoritmo di crittografia simmetrico o asimmetrico appropriato alla tua chiave di wrapping.

In generale, gli algoritmi di crittografia di chiavi simmetriche sono più rapidi e producono testi cifrati di dimensioni minori rispetto alla crittografia della chiave pubblica o asimmetrica. Ma gli algoritmi a chiave pubblica forniscono una separazione intrinseca dei ruoli. Per combinare i punti di forza di ciascuno, puoi crittografare la chiave dati con la crittografia a chiave pubblica.

Ti consigliamo di utilizzare uno dei AWS KMS portachiavi ogni volta che è possibile. Quando usi il AWS KMS portachiavi, puoi scegliere di combinare i punti di forza di più algoritmi specificando un RSA asimmetrico come chiave di avvolgimento. AWS KMS key Puoi anche utilizzare una chiave KMS di crittografia simmetrica.

## Chiave di dati

Una chiave dati è una chiave di crittografia che AWS Database Encryption SDK utilizza per crittografare i campi di un record contrassegnati `ENCRYPT_AND_SIGN` nelle azioni crittografiche. Ogni chiave di dati corrisponde a un array di byte conforme ai requisiti per le chiavi di crittografia. Il AWS Database Encryption SDK utilizza una chiave dati unica per crittografare ogni attributo.

Non è necessario specificare, generare, implementare, estendere, proteggere o utilizzare chiavi dati. Il AWS Database Encryption SDK funziona per te quando richiami le operazioni di crittografia e decrittografia.

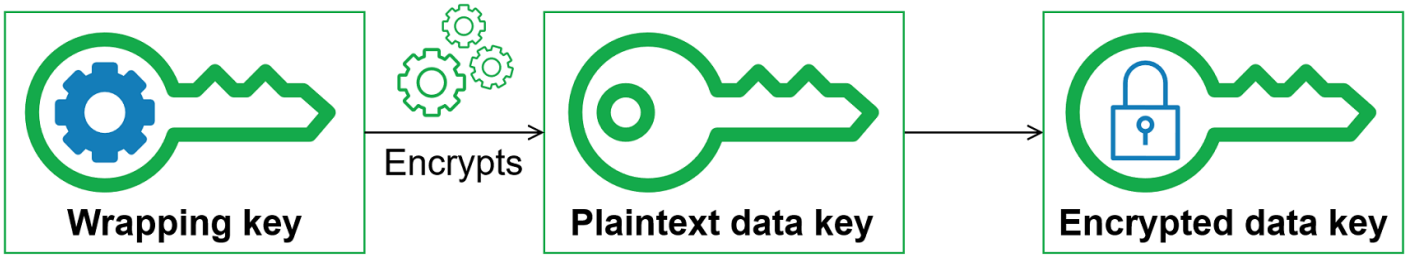
Per proteggere le chiavi dei dati, AWS Database Encryption SDK le crittografa utilizzando una o più chiavi di crittografia a chiave note come chiavi di wrapping. Dopo che AWS Database Encryption SDK ha utilizzato le chiavi di dati in testo semplice per crittografare i dati, le rimuove dalla memoria il prima possibile. Quindi memorizza la chiave di dati crittografata nella descrizione del materiale. Per informazioni dettagliate, consultare Come funziona il AWS Database Encryption SDK.

> **Tip**
>
> Nel AWS Database Encryption SDK, distinguiamo le chiavi di dati dalle chiavi di crittografia dei dati. Come best practice, tutte le suite di algoritmi supportate utilizzano una funzione di derivazione delle chiavi. La funzione di derivazione delle chiavi accetta una chiave di dati come input e restituisce le chiavi di crittografia dei dati che vengono effettivamente utilizzate per crittografare i record. Per questo motivo, abbiamo spesso detto che i dati sono crittografati "in" una chiave dei dati anziché "da" una chiave di dati.

Ogni chiave di dati crittografata include metadati, incluso l'identificatore della chiave di avvolgimento che l'ha crittografata. Questi metadati consentono al AWS Database Encryption SDK di identificare chiavi di wrapping valide durante la decrittografia.

## Chiave di avvolgimento

Una chiave di wrapping è una chiave di crittografia a chiave che AWS Database Encryption SDK utilizza per crittografare la chiave dati che crittografa i record. Ogni chiave di dati può essere crittografata con una o più chiavi di wrapping. Sei tu a determinare quali chiavi di wrapping vengono utilizzate per proteggere i tuoi dati quando configuri un portachiavi.




AWS Database Encryption SDK supporta diverse chiavi di wrapping di uso comune, come AWS Key Management Service(AWS KMS) chiavi KMS con crittografia simmetrica (incluse chiavi multiregionali AWS KMS) e chiavi RSA KMS asimmetriche, chiavi AES-GCM (Advanced Encryption Standard/ Galois Counter Mode) non elaborate e chiavi RSA non elaborate. Ti consigliamo di utilizzare le chiavi KMS ogni volta che è possibile. Per decidere quale chiave di avvolgimento utilizzare, vedi Selezione delle chiavi di avvolgimento.

Quando si utilizza la crittografia a busta, è necessario proteggere le chiavi di wrapping da accessi non autorizzati. È possibile eseguire questa operazione in uno dei seguenti modi:

- Utilizzate un servizio progettato per questo scopo, ad esempio AWS Key Management Service (AWS KMS).
- Utilizza un modulo di sicurezza hardware (HSM) come quelli offerti da AWS CloudHSM.
- Utilizzate altri strumenti e servizi di gestione delle chiavi.

Se non disponi di un sistema di gestione delle chiavi, ti consigliamo AWS KMS. Il AWS Database Encryption SDK si integra con AWS KMS per aiutarti a proteggere e utilizzare le tue chiavi di wrapping.

## Portachiavi

Per specificare le chiavi di avvolgimento utilizzate per la crittografia e la decrittografia, si utilizza un portachiavi. È possibile utilizzare i portachiavi forniti da AWS Database Encryption SDK o progettare implementazioni personalizzate.

Un keyring genera, crittografa e decritta le chiavi di dati. Genera inoltre le chiavi MAC utilizzate per calcolare i codici di autenticazione dei messaggi basati su hash (HMACs) nella firma. Quando definisci un portachiavi, puoi specificare le chiavi di avvolgimento che crittografano le tue chiavi dati. La maggior parte dei portachiavi specifica almeno una chiave di avvolgimento o un servizio che fornisce e protegge le chiavi di avvolgimento. Durante la crittografia, AWS Database Encryption SDK utilizza tutte le chiavi di wrapping specificate nel portachiavi per crittografare la chiave dati. Per informazioni sulla scelta e l'utilizzo dei portachiavi definiti da AWS Database Encryption SDK, consulta Utilizzo dei portachiavi.

## Azioni crittografiche

Le azioni crittografiche indicano al crittografo quali azioni eseguire su ogni campo di un record.

I valori delle azioni crittografiche possono essere uno dei seguenti:

- Crittografa e firma: crittografa il campo. Includi il campo crittografato nella firma.
- Solo firma: includi il campo nella firma.
- Firma e includi nel contesto di crittografia: include il campo nel contesto della firma e della crittografia.

  Per impostazione predefinita, le chiavi di partizione e ordinamento sono l'unico attributo incluso nel contesto di crittografia. Potresti prendere in considerazione la definizione di campi aggiuntivi `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` in modo che il fornitore dell'ID della chiave

di filiale per il tuo portachiavi AWS KMS gerarchico possa identificare quale chiave di filiale è necessaria per la decrittografia dal contesto di crittografia. Per ulteriori informazioni, consulta Branch Key ID supplier.

> **Note**
>
> Per utilizzare l'azione `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` crittografica, è necessario utilizzare la versione 3.3 o successiva del AWS Database Encryption SDK. Distribuisci la nuova versione a tutti i lettori prima di aggiornare il modello di dati per includere. `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`

- Non fare nulla: non crittografare o includere il campo nella firma.

Per qualsiasi campo in cui è possibile archiviare dati sensibili, utilizza Encrypt and sign. Per i valori della chiave primaria (ad esempio, una chiave di partizione e una chiave di ordinamento in una tabella DynamoDB), usa Sign only o Sign e includi nel contesto di crittografia. Se specifichi un qualsiasi attributo Sign e lo includi nel contesto di crittografia, anche gli attributi di partizione e ordinamento devono essere Sign e inclusi nel contesto di crittografia. Non è necessario specificare azioni crittografiche per la descrizione del materiale. Il AWS Database Encryption SDK firma automaticamente il campo in cui è memorizzata la descrizione del materiale.

Scegli con attenzione le tue azioni crittografiche. In caso di dubbio, usa Encrypt and sign (Crittografa e firma). Dopo aver utilizzato AWS Database Encryption SDK per proteggere i record, non è possibile modificare un `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` campo esistente `ENCRYPT_AND_SIGN` o modificare l'azione crittografica assegnata a un campo esistente. `SIGN_ONLY` `DO_NOTHING` `DO_NOTHING` Tuttavia, puoi comunque apportare altre modifiche al tuo modello di dati. Ad esempio, puoi aggiungere o rimuovere campi crittografati in un'unica distribuzione.

## Descrizione dei materiali

La descrizione del materiale funge da intestazione per un record crittografato. Quando crittografate e firmate i campi con il AWS Database Encryption SDK, il criptatore registra la descrizione del materiale mentre assembla i materiali crittografici e archivia la descrizione del materiale in un nuovo campo (`aws_dbe_head`) che il crittografo aggiunge al record.

La descrizione del materiale è una struttura di dati formattata portatile che contiene copie crittografate delle chiavi di dati e altre informazioni, come algoritmi di crittografia, contesto di crittografia e istruzioni di crittografia e firma. Il criptatore registra la descrizione del materiale mentre assembla

i materiali crittografici per la crittografia e la firma. Successivamente, quando deve assemblare materiale crittografico per verificare e decrittografare un campo, utilizza la descrizione del materiale come guida.

La memorizzazione delle chiavi dati crittografate insieme al campo crittografato semplifica l'operazione di decrittografia ed evita la necessità di archiviare e gestire le chiavi dati crittografate indipendentemente dai dati crittografati.

Per informazioni tecniche sulla descrizione del materiale, vedere. Formato della descrizione del materiale

## Contesto di crittografia

Per migliorare la sicurezza delle operazioni crittografiche, il AWS Database Encryption SDK include un contesto di crittografia in tutte le richieste di crittografia e firma di un record.

Un contesto di crittografia è un set di coppie nome-valore che contiene dati autenticati aggiuntivi arbitrari e non segreti. Il AWS Database Encryption SDK include il nome logico del database e i valori della chiave primaria (ad esempio, una chiave di partizione e una chiave di ordinamento in una tabella DynamoDB) nel contesto di crittografia. Quando si crittografa e si firma un campo, il contesto di crittografia viene associato crittograficamente al record crittografato in modo che lo stesso contesto di crittografia sia necessario per decrittografare il campo.

Se si utilizza un AWS KMS portachiavi, AWS Database Encryption SDK utilizza anche il contesto di crittografia per fornire dati autenticati aggiuntivi (AAD) nelle chiamate a cui effettua il portachiavi. AWS KMS

Ogni volta che si utilizza la suite di algoritmi predefinita, il gestore dei materiali crittografici (CMM) aggiunge una coppia nome-valore al contesto di crittografia che consiste in un nome riservato e un valore che rappresenta la chiave di `aws-crypto-public-key` verifica pubblica. La chiave di verifica pubblica è memorizzata nella descrizione del materiale.

## Responsabile di materiali crittografici

Il gestore dei materiali crittografici (CMM) assembla i materiali crittografici utilizzati per crittografare, decrittografare e firmare i dati. Ogni volta che si utilizza la suite di algoritmi predefinita, i materiali crittografici includono chiavi di dati in chiaro e crittografate, chiavi di firma simmetriche e una chiave di firma asimmetrica. Non interagisci mai direttamente con la CMM. I metodi di crittografia e decrittazione lo gestiscono per te.

Poiché la CMM funge da collegamento tra il AWS Database Encryption SDK e un portachiavi, è il punto ideale per la personalizzazione e l'estensione, ad esempio il supporto per l'applicazione delle politiche. È possibile specificare esplicitamente una CMM, ma non è obbligatorio. Quando specificate un portachiavi, AWS Database Encryption SDK crea automaticamente una CMM predefinita. La CMM predefinita ottiene i materiali di crittografia o decrittografia dal portachiavi specificato. Ciò potrebbe comportare una chiamata a un servizio crittografico, come AWS Key Management Service (AWS KMS).

## Crittografia simmetrica e asimmetrica

La crittografia simmetrica utilizza la stessa chiave per crittografare e decrittografare i dati.

La crittografia asimmetrica utilizza una coppia di chiavi di dati matematicamente correlate. Una chiave della coppia crittografa i dati; solo l'altra chiave della coppia può decrittografare i dati.

Il AWS Database Encryption SDK utilizza la crittografia a busta. Crittografa i dati con una chiave dati simmetrica. Crittografa la chiave dati simmetrica con una o più chiavi di avvolgimento simmetriche o asimmetriche. Aggiunge una descrizione del materiale al record che include almeno una copia crittografata della chiave dati.

Crittografia dei dati (crittografia simmetrica)

> Per crittografare i dati, AWS Database Encryption SDK utilizza una chiave dati simmetrica e una suite di algoritmi che include un algoritmo di crittografia simmetrica. Per decrittografare i dati, AWS Database Encryption SDK utilizza la stessa chiave di dati e la stessa suite di algoritmi.

Crittografia della chiave dati (crittografia simmetrica o asimmetrica)

> Il portachiavi fornito per un'operazione di crittografia e decrittografia determina il modo in cui la chiave dati simmetrica viene crittografata e decrittografata. Puoi scegliere un portachiavi che utilizza la crittografia simmetrica, ad esempio un portachiavi con una chiave KMS di crittografia simmetrica, o uno che utilizza la crittografia asimmetrica, come un AWS KMS portachiavi con una chiave RSA KMS asimmetrica. AWS KMS

## Impegno chiave

Il AWS Database Encryption SDK supporta Key Commitment (talvolta nota come robustezza), una proprietà di sicurezza che garantisce che ogni testo cifrato possa essere decrittografato solo in un singolo testo non crittografato. A tale scopo, Key Commitment garantisce che solo la chiave dati

che ha crittografato il record venga utilizzata per decrittografarlo. Il AWS Database Encryption SDK include un impegno fondamentale per tutte le operazioni di crittografia e decrittografia.

La maggior parte dei cifrari simmetrici moderni (incluso AES) crittografa il testo in chiaro con un'unica chiave segreta, come la chiave dati univoca utilizzata da AWS Database Encryption SDK per crittografare ogni campo di testo in chiaro contrassegnato in un record. `ENCRYPT_AND_SIGN` La decrittografia di questo record con la stessa chiave di dati restituisce un testo in chiaro identico all'originale. La decrittografia con una chiave diversa di solito non riesce. Sebbene difficile, è tecnicamente possibile decrittografare un testo cifrato con due chiavi diverse. In rari casi, è possibile trovare una chiave in grado di decrittografare parzialmente il testo cifrato in un testo semplice diverso, ma comunque comprensibile.

Il AWS Database Encryption SDK crittografa sempre ogni attributo con un'unica chiave dati. Potrebbe crittografare quella chiave dati con più chiavi di wrapping, ma le chiavi di wrapping crittografano sempre la stessa chiave dati. Tuttavia, un record crittografato sofisticato creato manualmente potrebbe effettivamente contenere diverse chiavi di dati, ognuna crittografata da una chiave di wrapping diversa. Ad esempio, se un utente decrittografa il record crittografato, restituisce 0x0 (falso) mentre un altro utente che decrittografa lo stesso record crittografato ottiene 0x1 (vero).

Per evitare questo scenario, AWS Database Encryption SDK include un impegno chiave durante la crittografia e la decrittografia. Il metodo di crittografia associa crittograficamente la chiave di dati univoca che ha prodotto il testo cifrato all'impegno chiave, un codice di autenticazione dei messaggi basato su hash (HMAC) calcolato sulla descrizione del materiale utilizzando una derivazione della chiave dati. Quindi memorizza l'impegno chiave nella descrizione del materiale. Quando decripta un record con l'impegno della chiave, AWS Database Encryption SDK verifica che la chiave dati sia l'unica chiave per quel record crittografato. Se la verifica della chiave dati non riesce, l'operazione di decrittografia ha esito negativo.

## Firme digitali

Il AWS Database Encryption SDK crittografa i dati utilizzando un algoritmo di crittografia autenticato, AES-GCM, e il processo di decrittografia verifica l'integrità e l'autenticità di un messaggio crittografato senza utilizzare una firma digitale. Tuttavia, poiché AES-GCM utilizza chiavi simmetriche, chiunque sia in grado di decrittografare la chiave dati utilizzata per decrittografare il testo cifrato potrebbe anche creare manualmente un nuovo testo cifrato crittografato, causando potenziali problemi di sicurezza. Ad esempio, se utilizzi una AWS KMS key come chiave di avvolgimento, un utente con autorizzazioni potrebbe creare testi cifrati crittografati senza chiamare. `kms:Decrypt` `kms:Encrypt`

Per evitare questo problema, la suite di algoritmi predefinita aggiunge una firma Elliptic Curve Digital Signature Algorithm (ECDSA) ai record crittografati. La suite di algoritmi predefinita crittografa i campi del record contrassegnati `ENCRYPT_AND_SIGN` utilizzando un algoritmo di crittografia autenticato, AES-GCM. Quindi, calcola sia i codici di autenticazione dei messaggi basati su hash (HMACs) che le firme ECDSA asimmetriche sui campi del record contrassegnati con, e. `ENCRYPT_AND_SIGN` `SIGN_ONLY` `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` Il processo di decrittografia utilizza le firme per verificare che un utente autorizzato abbia crittografato il record.

Quando viene utilizzata la suite di algoritmi predefinita, AWS Database Encryption SDK genera una chiave privata temporanea e una coppia di chiavi pubbliche per ogni record crittografato. Il AWS Database Encryption SDK memorizza la chiave pubblica nella descrizione del materiale e scarta la chiave privata. Ciò garantisce che nessuno possa creare un'altra firma verificabile con la chiave pubblica. L'algoritmo associa la chiave pubblica alla chiave dati crittografata come dati autenticati aggiuntivi nella descrizione del materiale, impedendo agli utenti che possono solo decrittografare i campi di alterare la chiave pubblica o influire sulla verifica della firma.

Il AWS Database Encryption SDK include sempre la verifica HMAC. Le firme digitali ECDSA sono abilitate per impostazione predefinita, ma non sono obbligatorie. Se gli utenti che crittografano i dati e gli utenti che decifrano i dati sono altrettanto affidabili, potresti prendere in considerazione l'utilizzo di una suite di algoritmi che non includa firme digitali per migliorare le tue prestazioni. Per ulteriori informazioni sulla selezione di suite di algoritmi alternative, consulta Scelta di una suite di algoritmi.

> **Note**
>
> Se un portachiavi non distingue tra criptatori e decriptatori, le firme digitali non forniscono alcun valore crittografico.

AWS KMS I portachiavi, incluso il portachiavi RSA asimmetrico, possono distinguere tra crittografatori e decryptor in base alle policy AWS KMS chiave e alle policy IAM. AWS KMS

A causa della loro natura crittografica, i seguenti portachiavi non possono distinguere tra crittografatori e decryptor:

- AWS KMS Portachiavi gerarchico
- AWS KMS Portachiavi ECDH
- Keyring non elaborato AES
- Keyring non elaborato RSA

- Portachiavi ECDH grezzo

# Come funziona il AWS Database Encryption SDK

La nostra libreria di crittografia lato client è stata rinominata Database Encryption SDK. AWS Questa guida per sviluppatori fornisce ancora informazioni sul DynamoDB Encryption Client.

Il AWS Database Encryption SDK fornisce librerie di crittografia lato client progettate specificamente per proteggere i dati archiviati nei database. Le librerie includono implementazioni sicure che puoi estendere o utilizzare senza modificarle. Per ulteriori informazioni sulla definizione e l'utilizzo di componenti personalizzati, consulta l' GitHub archivio per l'implementazione del database.

I flussi di lavoro di questa sezione spiegano come il AWS Database Encryption SDK crittografa, firma, decrittografa e verifica i dati nel database. Questi flussi di lavoro descrivono il processo di base utilizzando elementi astratti e le funzionalità predefinite. Per informazioni dettagliate su come il AWS Database Encryption SDK interagisce con l'implementazione del database, consulta l'argomento Cos'è la crittografia per il database.

Il AWS Database Encryption SDK utilizza la crittografia a busta per proteggere i dati. Ogni record è crittografato con una chiave dati unica. La chiave dati viene utilizzata per derivare una chiave di crittografia dei dati univoca per ogni campo contrassegnato `ENCRYPT_AND_SIGN` nelle azioni crittografiche. Quindi, una copia della chiave dati viene crittografata dalle chiavi di wrapping specificate. Per decrittografare il record crittografato, AWS Database Encryption SDK utilizza le chiavi di wrapping specificate per decrittografare almeno una chiave di dati crittografata. Quindi può decrittografare il testo cifrato e restituire una voce in testo semplice.

Per ulteriori informazioni sui termini utilizzati nel Database Encryption SDK, consulta AWS . AWS Concetti dell'SDK per la crittografia dei database

## Crittografa e firma

Fondamentalmente, AWS Database Encryption SDK è un crittografo dei record che crittografa, firma, verifica e decrittografa i record del database. Contiene informazioni sui tuoi record e istruzioni su quali campi crittografare e firmare. Riceve i materiali di crittografia e le istruzioni su come utilizzarli da un gestore di materiali crittografici configurato sulla base della chiave di wrapping specificata.

La procedura dettagliata seguente descrive come il AWS Database Encryption SDK crittografa e firma i dati immessi.

1. Il gestore dei materiali crittografici fornisce al AWS Database Encryption SDK chiavi di crittografia dei dati uniche: una chiave dati in chiaro, una copia della chiave dati crittografata con la chiave di wrapping specificata e una chiave MAC.

   > **Note**
   >
   > È possibile crittografare la chiave dati con più chiavi di wrapping. Ciascuna chiave di wrapping crittografa una copia separata della chiave dati. Il AWS Database Encryption SDK memorizza tutte le chiavi di dati crittografate nella descrizione del materiale. Il AWS Database Encryption SDK aggiunge un nuovo campo (`aws_dbe_head`) al record che memorizza la descrizione del materiale.
   >
   > Viene derivata una chiave MAC per ogni copia crittografata della chiave dati. Le chiavi MAC non sono memorizzate nella descrizione del materiale. Invece, il metodo decrypt utilizza le chiavi di wrapping per derivare nuovamente le chiavi MAC.

2. Il metodo di crittografia crittografa ogni campo contrassegnato come `ENCRYPT_AND_SIGN` nelle azioni crittografiche specificate.
3. Il metodo di crittografia ricava una chiave `commitKey` dalla chiave dati e la utilizza per generare un valore di impegno chiave, quindi scarta la chiave dati.
4. Il metodo di crittografia aggiunge una descrizione del materiale al record. La descrizione del materiale contiene le chiavi di dati crittografate e le altre informazioni sul record crittografato. Per un elenco completo delle informazioni incluse nella descrizione del materiale, consulta Formato della descrizione del materiale.
5. Il metodo di crittografia utilizza le chiavi MAC restituite nel passaggio 1 per calcolare i valori HMAC (Hash-Based Message Authentication Code) sulla canonicalizzazione della descrizione del materiale, del contesto di crittografia e di ogni campo contrassegnato `ENCRYPT_AND_SIGN` o `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` nelle azioni crittografiche. `SIGN_ONLY` I valori HMAC vengono memorizzati in un nuovo campo (`aws_dbe_foot`) che il metodo di crittografia aggiunge al record.
6. Il metodo di crittografia calcola una firma ECDSA sulla base della canonicalizzazione della descrizione del materiale, del contesto di crittografia e di ogni campo contrassegnato `ENCRYPT_AND_SIGN` oppure `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` e memorizza le firme ECDSA nel campo. `SIGN_ONLY` `aws_dbe_foot`

> **Note**
>
> Le firme ECDSA sono abilitate per impostazione predefinita, ma non sono obbligatorie.

7. Il metodo di crittografia archivia il record crittografato e firmato nel database

## Decrittografa e verifica

1. Il gestore dei materiali crittografici (CMM) fornisce il metodo di decrittografia con i materiali di decrittografia memorizzati nella descrizione del materiale, inclusa la chiave di dati in chiaro e la chiave MAC associata.

   - La CMM decrittografa la chiave dati crittografata con le chiavi di avvolgimento nel portachiavi specificato e restituisce la chiave di dati in testo semplice.

2. Il metodo di decrittografia confronta e verifica il valore di impegno chiave nella descrizione del materiale.
3. Il metodo di decrittografia verifica le firme nel campo della firma.

   Identifica quali campi sono contrassegnati `ENCRYPT_AND_SIGN` o `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` dall'elenco dei campi non autenticati consentiti che hai definito. `SIGN_ONLY` Il metodo di decrittografia utilizza la chiave MAC restituita nel passaggio 1 per ricalcolare e confrontare i valori HMAC per i campi contrassegnati, o. `ENCRYPT_AND_SIGN` `SIGN_ONLY` `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` Quindi, verifica le firme ECDSA utilizzando la chiave pubblica memorizzata nel contesto di crittografia.
4. Il metodo di decrittografia utilizza la chiave di dati in testo semplice per decrittografare ogni valore contrassegnato. `ENCRYPT_AND_SIGN` Il AWS Database Encryption SDK scarta quindi la chiave di dati in testo semplice.
5. Il metodo di decrittografia restituisce il record di testo in chiaro.

# Suite di algoritmi supportate nel AWS Database Encryption SDK

La nostra libreria di crittografia lato client è stata rinominata Database Encryption SDK. AWS Questa guida per sviluppatori fornisce ancora informazioni sul DynamoDB Encryption Client.

Una suite di algoritmi è una raccolta di algoritmi di crittografia e dei relativi valori. I sistemi crittografici utilizzano l'implementazione dell'algoritmo per generare il testo cifrato.

Il AWS Database Encryption SDK utilizza una suite di algoritmi per crittografare e firmare i campi del database. Tutte le suite di algoritmi supportate utilizzano l'algoritmo Advanced Encryption Standard (AES) con Galois/Counter Mode (GCM), noto come AES-GCM, per crittografare i dati grezzi. Il Database Encryption SDK supporta chiavi di crittografia a 256 bit. AWS La lunghezza del tag di autenticazione è sempre 16 byte.

AWS Suite di algoritmi SDK per la crittografia dei database

| Algoritmo | Algoritmo di crittografia | Lunghezza chiave dati (in bit) | Algoritmo di derivazione della chiave | Algoritmo di firma simmetrica | Algoritmo di firma asimmetrico | Impegno chiave |
|---|---|---|---|---|---|---|
| Predefinita | AES-GCM | 256 | HKDF con SHA-512 | HMAC-SHA-384 | ECDSA con P-384 e SHA-384 | HKDF con SHA-512 |
| AES-GCM senza firme digitali ECDSA | AES-GCM | 256 | HKDF con SHA-512 | HMAC-SHA-384 | Nessuno | HKDF con SHA-512 |

Algoritmo di crittografia

Il nome e la modalità dell'algoritmo di crittografia utilizzato. Le suite di algoritmi del AWS Database Encryption SDK utilizzano l'algoritmo Advanced Encryption Standard (AES) con Galois/Counter Mode (GCM).

Lunghezza chiave dati

La lunghezza della chiave dati in bit. Il AWS Database Encryption SDK supporta chiavi dati a 256 bit. La chiave dati viene utilizzata come input per una funzione di derivazione delle extract-and-expand chiavi basata su HMAC (HKDF). L'output dell'HKDF viene usato come chiave di crittografia dei dati nell'algoritmo di crittografia.

Algoritmo di derivazione della chiave

La funzione di derivazione delle extract-and-expand chiavi basata su HMAC (HKDF) utilizzata per derivare la chiave di crittografia dei dati. Il AWS Database Encryption SDK utilizza l'HKDF definito nella RFC 5869.

- La funzione hash utilizzata è SHA-512
- Per la fase di estrazione:
  - Non vengono utilizzati salt. Secondo la RFC, il sale è impostato su una stringa di zeri.
  - Il materiale di codifica di input è la chiave dati del portachiavi.
- Per la fase di espansione:
  - La chiave di input pseudo-casuale è l'output della fase di estrazione.
  - L'etichetta della chiave è costituita dai byte della stringa con codifica UTF-8 in ordine di byte big endian. `DERIVEKEY`
  - Le informazioni di input sono una concatenazione dell'ID dell'algoritmo e dell'etichetta della chiave (in quest'ordine).
  - La lunghezza del materiale di codifica di output è la lunghezza della chiave Data. Questo output viene usato come chiave di crittografia dei dati nell'algoritmo di crittografia.

Algoritmo di firma simmetrica

L'algoritmo Hash-Based Message Authentication Code (HMAC) utilizzato per generare una firma simmetrica. Tutte le suite di algoritmi supportate includono la verifica HMAC.

AWS Database Encryption SDK serializza la descrizione del materiale e tutti i campi contrassegnati `ENCRYPT_AND_SIGN` con, o. `SIGN_ONLY` `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` Quindi, utilizza HMAC con un algoritmo di funzione hash crittografica (SHA-384) per firmare la canonicalizzazione.

La firma HMAC simmetrica viene archiviata in un nuovo campo () `aws_dbe_foot` che Database Encryption SDK aggiunge al record. AWS

Algoritmo di firma asimmetrico

L'algoritmo di firma utilizzato per generare una firma digitale asimmetrica.

AWS Database Encryption SDK serializza la descrizione del materiale e tutti i campi contrassegnati con, o. `ENCRYPT_AND_SIGN` `SIGN_ONLY` `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` Quindi, utilizza l'Elliptic Curve Digital Signature Algorithm (ECDSA) con le seguenti specifiche per firmare la canonicalizzazione:

- La curva ellittica utilizzata è la P-384, come definita nel Digital Signature Standard (DSS) (FIPS PUB 186-4).
- La funzione hash utilizzata è SHA-384.

La firma ECDSA asimmetrica viene memorizzata sul campo insieme alla firma HMAC simmetrica. `aws_dbe_foot`

Le firme digitali ECDSA sono incluse di default, ma non sono obbligatorie.

Impegno chiave

La funzione di derivazione delle extract-and-expand chiavi (HKDF) basata su HMAC utilizzata per derivare la chiave di commit.

- La funzione hash utilizzata è SHA-512
- Per la fase di estrazione:
  - Non vengono utilizzati salt. Secondo la RFC, il sale è impostato su una stringa di zeri.
  - Il materiale di codifica di input è la chiave dati del portachiavi.
- Per la fase di espansione:
  - La chiave di input pseudo-casuale è l'output della fase di estrazione.
  - Le informazioni di input sono i byte della stringa con codifica UTF-8 in ordine di byte big endian. `COMMITKEY`
  - La lunghezza del materiale di codifica in uscita è di 256 bit. Questo output viene utilizzato come chiave di commit.

La chiave di commit calcola il record commit, un hash HMAC (Hash-Based Message Authentication Code) distinto a 256 bit, rispetto alla descrizione del materiale. Per una spiegazione tecnica dell'aggiunta dell'impegno chiave a una suite di algoritmi, vedete Key Committing in Cryptology ePrint Archive. AEADs

## Suite di algoritmi predefinita

Per impostazione predefinita, AWS Database Encryption SDK utilizza una suite di algoritmi con AES-GCM, una funzione di derivazione delle extract-and-expand chiavi basata su HMAC (HKDF), verifica HMAC, firme digitali ECDSA, key commit e una chiave di crittografia a 256 bit.

La suite di algoritmi predefinita include la verifica HMAC (firme simmetriche) e le firme digitali ECDSA (firme asimmetriche). Queste firme sono archiviate in un nuovo campo () che Database Encryption

SDK aggiunge al record. `aws_dbe_foot` AWS Le firme digitali ECDSA sono particolarmente utili quando la politica di autorizzazione consente a un set di utenti di crittografare i dati e a un gruppo diverso di utenti di decrittografare i dati.

La suite di algoritmi predefinita deriva anche da un impegno chiave: un hash HMAC che collega la chiave dei dati al record. Il valore di impegno chiave è un HMAC calcolato dalla descrizione del materiale e dalla chiave di commit. Il valore chiave di impegno viene quindi memorizzato nella descrizione del materiale. Key Commitment garantisce che ogni testo cifrato venga decrittografato in un solo testo non crittografato. Lo fanno convalidando la chiave dati utilizzata come input per l'algoritmo di crittografia. Durante la crittografia, la suite di algoritmi ricava un impegno chiave HMAC. Prima della decrittografia, convalidano che la chiave dati produca lo stesso impegno chiave HMAC. In caso contrario, la chiamata di decrittografia fallisce.

## AES-GCM senza firme digitali ECDSA

Sebbene la suite di algoritmi predefinita sia probabilmente adatta alla maggior parte delle applicazioni, è possibile scegliere una suite di algoritmi alternativa. Ad esempio, alcuni modelli di fiducia sarebbero soddisfatti da una suite di algoritmi senza firme digitali ECDSA. Utilizza questa suite solo quando gli utenti che crittografano i dati e gli utenti che decifrano i dati sono ugualmente affidabili.

Tutte le suite di algoritmi AWS Database Encryption SDK includono la verifica HMAC (firme simmetriche). L'unica differenza è che la suite di algoritmi AES-GCM senza firma digitale ECDSA è priva della firma asimmetrica che fornisce un ulteriore livello di autenticità e non ripudio.

Ad esempio, se nel portachiavi sono presenti più chiavi di avvolgimento e decifrate un record utilizzando `wrappingKeyA` la firma `wrappingKeyA` simmetrica `wrappingKeyB` HMAC per `wrappingKeyC` verificare che il record sia stato crittografato da un utente con accesso a. `wrappingKeyA` Se hai utilizzato la suite di algoritmi predefinita, HMACs forniscono la stessa verifica e inoltre utilizzano la firma digitale ECDSA per garantire che il record sia stato crittografato da un utente con autorizzazioni di crittografia per. `wrappingKeyA` `wrappingKeyA`

Per selezionare la suite di algoritmi AES-GCM senza firme digitali, includi il seguente frammento nella configurazione di crittografia.

Java

> Il seguente frammento specifica la suite di algoritmi AES-GCM senza firme digitali ECDSA. Per ulteriori informazioni, consulta the section called “Configurazione della crittografia”.

```
.algorithmSuiteId(
    DBEAlgorithmSuiteId.ALG_AES_256_GCM_HKDF_SHA512_COMMIT_KEY_SYMSIG_HMAC_SHA384)
```

C# / .NET

Il seguente frammento specifica la suite di algoritmi AES-GCM senza firme digitali ECDSA. Per ulteriori informazioni, consulta the section called “Configurazione della crittografia”.

```
AlgorithmSuiteId =
 DBEAlgorithmSuiteId.ALG_AES_256_GCM_HKDF_SHA512_COMMIT_KEY_SYMSIG_HMAC_SHA384
```

Rust

Il seguente frammento specifica la suite di algoritmi AES-GCM senza firme digitali ECDSA. Per ulteriori informazioni, consulta the section called “Configurazione della crittografia”.

```
.algorithm_suite_id(
    DbeAlgorithmSuiteId::AlgAes256GcmHkdfSha512CommitKeyEcdsaP384SymsigHmacSha384,
)
```

# Utilizzo del AWS Database Encryption SDK con AWS KMS

La nostra libreria di crittografia lato client è stata rinominata Database Encryption SDK. AWS Questa guida per sviluppatori fornisce ancora informazioni sul DynamoDB Encryption Client.

Per utilizzare AWS Database Encryption SDK, è necessario configurare un portachiavi e specificare una o più chiavi di wrapping. Se non un'infrastruttura di chiavi non è disponibile, consigliamo di utilizzare AWS Key Management Service (AWS KMS).

Il AWS Database Encryption SDK supporta due tipi di portachiavi. AWS KMS Il AWS KMS portachiavi tradizionale viene utilizzato AWS KMS keysper generare, crittografare e decrittografare le chiavi di dati. È possibile utilizzare la crittografia simmetrica (`SYMMETRIC_DEFAULT`) o le chiavi RSA KMS asimmetriche. Poiché AWS Database Encryption SDK crittografa e firma ogni record con una chiave dati unica, il AWS KMS portachiavi deve eseguire ogni operazione di crittografia e decrittografia. AWS KMS Per le applicazioni che devono ridurre al minimo il numero di chiamate a AWS KMS, AWS Database Encryption SDK supporta anche il portachiavi gerarchico.AWS KMS Il portachiavi Hierarchical è una soluzione di memorizzazione nella cache dei materiali crittografici che riduce il numero di AWS KMS chiamate utilizzando chiavi branch AWS KMS  protette persistenti in una tabella Amazon DynamoDB e quindi memorizzando nella cache locale i materiali chiave delle branch utilizzati nelle operazioni di crittografia e decrittografia. AWS KMS Consigliamo di utilizzare i portachiavi ogni volta che è possibile.

Per interagire con AWS KMS, il AWS Database Encryption SDK richiede il AWS KMS modulo di. AWS SDK per Java

Per prepararsi a utilizzare il AWS Database Encryption SDK con AWS KMS

1. Crea un Account AWS. Per ulteriori informazioni, consulta Come posso creare e attivare un nuovo account Amazon Web Services? nel AWS Knowledge Center.
2. Crea una crittografia simmetrica. AWS KMS key Per assistenza, consulta Creating Keys nella AWS Key Management Service Developer Guide.

> **Tip**
>
> Per utilizzarlo a AWS KMS key livello di codice, è necessario l'Amazon Resource Name (ARN) di. AWS KMS keyPer informazioni su come trovare l'ARN di un AWS KMS key,

consulta Finding the Key ID and ARN nella Developer Guide.AWS Key Management Service

3. Genera un ID della chiave di accesso e una chiave di accesso di sicurezza. Puoi utilizzare l'ID della chiave di accesso e la chiave di accesso segreta per un utente IAM oppure puoi utilizzarli per AWS Security Token Service creare una nuova sessione con credenziali di sicurezza temporanee che includono un ID della chiave di accesso, una chiave di accesso segreta e un token di sessione. Come best practice di sicurezza, ti consigliamo di utilizzare credenziali temporanee anziché le credenziali a lungo termine associate ai tuoi account utente o utente AWS (root) IAM.

   Per creare un utente IAM con una chiave di accesso, consulta Creating IAM Users nella IAM User Guide.

   Per generare credenziali di sicurezza temporanee, consulta Richiesta di credenziali di sicurezza temporanee nella Guida per l'utente IAM.
4. Imposta AWS le tue credenziali utilizzando le istruzioni contenute nell'ID della chiave di accesso AWS SDK per Javae nella chiave di accesso segreta che hai generato nel passaggio 3. Se hai generato credenziali temporanee, dovrai specificare anche il token di sessione.

   Questa procedura consente di AWS SDKs firmare le AWS richieste al posto tuo. Gli esempi di codice contenuti nel AWS Database Encryption SDK con cui interagisci AWS KMS presuppongono che tu abbia completato questo passaggio.

# Configurazione del Database Encryption SDK AWS

La nostra libreria di crittografia lato client è stata rinominata Database Encryption SDK. AWS Questa guida per sviluppatori fornisce ancora informazioni sul DynamoDB Encryption Client.

Il AWS Database Encryption SDK è progettato per essere facile da usare. Sebbene AWS Database Encryption SDK abbia diverse opzioni di configurazione, i valori predefiniti vengono scelti con cura per essere pratici e sicuri per la maggior parte delle applicazioni. Tuttavia, potrebbe essere necessario modificare la configurazione per migliorare le prestazioni o includere una funzionalità personalizzata nella progettazione.

Argomenti



## Selezione di un linguaggio di programmazione

Il AWS Database Encryption SDK per DynamoDB è disponibile in diversi linguaggi di programmazione. Le implementazioni del linguaggio sono progettate per essere completamente interoperabili e per offrire le stesse funzionalità, sebbene possano essere implementate in modi diversi. In genere, si utilizza la libreria compatibile con l'applicazione.

## Selezione delle chiavi di avvolgimento

Il AWS Database Encryption SDK genera una chiave dati simmetrica unica per crittografare ogni campo. Non è necessario configurare, gestire o utilizzare le chiavi dati. AWS Database Encryption SDK lo fa per te.

Tuttavia, è necessario selezionare una o più chiavi di wrapping per crittografare ogni chiave di dati. AWS Database Encryption SDK supporta AWS Key Management Service(AWS KMS) chiavi KMS di crittografia simmetrica e chiavi KMS RSA asimmetriche. Supporta anche chiavi simmetriche AES e

chiavi asimmetriche RSA fornite in diverse dimensioni. Sei responsabile della sicurezza e della durata delle tue chiavi di wrapping, quindi ti consigliamo di utilizzare una chiave di crittografia in un modulo di sicurezza hardware o in un servizio di infrastruttura chiave, ad esempio. AWS KMS

Per specificare le chiavi di avvolgimento per la crittografia e la decrittografia, si utilizza un portachiavi. A seconda del tipo di portachiavi utilizzato, è possibile specificare una chiave di avvolgimento o più chiavi di avvolgimento dello stesso tipo o di tipi diversi. Se utilizzi più chiavi di wrapping per racchiudere una chiave dati, ogni chiave di wrapping crittograferà una copia della stessa chiave dati. Le chiavi dati crittografate (una per chiave di avvolgimento) vengono memorizzate nella descrizione del materiale memorizzata accanto al campo crittografato. Per decrittografare i dati, il AWS Database Encryption SDK deve prima utilizzare una delle chiavi di wrapping per decrittografare una chiave dati crittografata.

Ti consigliamo di utilizzare uno dei portachiavi ogni volta che è possibile. AWS KMS Il AWS Database Encryption SDK fornisce il AWS KMS portachiavi e il portachiavi AWS KMS gerarchico, che riducono il numero di chiamate effettuate a. AWS KMS Per specificare un elemento AWS KMS key in un portachiavi, utilizza un identificatore di chiave supportato. AWS KMS Se si utilizza il portachiavi AWS KMS gerarchico, è necessario specificare l'ARN della chiave. Per i dettagli sugli identificatori chiave per una chiave, consulta Identificatori AWS KMS chiave nella Guida per gli sviluppatori.AWS Key Management Service

- Quando si esegue la crittografia con un AWS KMS portachiavi, è possibile specificare qualsiasi identificatore di chiave valido (ARN della chiave, nome alias, alias ARN o ID chiave) per una chiave KMS di crittografia simmetrica. Se si utilizza una chiave RSA KMS asimmetrica, è necessario specificare la chiave ARN.

  Se si specifica un nome alias o un alias ARN per una chiave KMS durante la crittografia, AWS Database Encryption SDK salva la chiave ARN attualmente associata a quell'alias; non salva l'alias. Le modifiche all'alias non influiscono sulla chiave KMS utilizzata per decrittografare le chiavi dati.
- Per impostazione predefinita, il AWS KMS portachiavi decripta i record in modalità rigorosa (dove si specificano particolari chiavi KMS). È necessario utilizzare una chiave ARN per l'identificazione AWS KMS keys per la decrittografia.

  Quando si esegue la crittografia con un AWS KMS portachiavi, AWS Database Encryption SDK memorizza l'ARN della chiave AWS KMS key nella descrizione del materiale con la chiave dati crittografata. Durante la decrittografia in modalità rigorosa, AWS Database Encryption SDK verifica che la stessa chiave ARN sia presente nel portachiavi prima di tentare di utilizzare la

chiave di wrapping per decrittografare la chiave dati crittografata. Se si utilizza un identificatore di chiave diverso, AWS Database Encryption SDK non lo riconoscerà né lo utilizzerà, anche se gli identificatori si riferiscono alla AWS KMS key stessa chiave.

- Durante la decrittografia in modalità Discovery, non viene specificata alcuna chiave di wrapping. Innanzitutto, il AWS Database Encryption SDK tenta di decrittografare il record con la chiave ARN memorizzata nella descrizione del materiale. Se ciò non funziona, AWS Database Encryption SDK chiede AWS KMS di decrittografare il record utilizzando la chiave KMS che lo ha crittografato, indipendentemente da chi possiede o ha accesso a quella chiave KMS.

Per specificare una chiave AES non elaborata o una coppia di chiavi RSA non elaborata come chiave di wrapping in un portachiavi, è necessario specificare uno spazio dei nomi e un nome. Durante la decrittografia, è necessario utilizzare lo stesso identico spazio dei nomi e lo stesso nome per ogni chiave di wrapping non elaborata utilizzata durante la crittografia. Se utilizzi un namespace o un nome diverso, AWS Database Encryption SDK non riconoscerà né utilizzerà la chiave di wrapping, anche se il materiale della chiave è lo stesso.

## Creazione di un filtro di rilevamento

Quando si decifrano dati crittografati con chiavi KMS, è consigliabile decrittografarli in modalità rigorosa, ovvero limitare le chiavi di wrapping utilizzate solo a quelle specificate dall'utente. Tuttavia, se necessario, puoi anche decrittografare in modalità di scoperta, in cui non specifichi alcuna chiave di wrapping. In questa modalità, AWS KMS puoi decrittografare la chiave dati crittografata utilizzando la chiave KMS che l'ha crittografata, indipendentemente da chi possiede o ha accesso a quella chiave KMS.

Se è necessario decrittografare in modalità di rilevamento, si consiglia di utilizzare sempre un filtro di rilevamento, che limita le chiavi KMS che possono essere utilizzate a quelle presenti in una partizione e specificata. Account AWS Il filtro di rilevamento è facoltativo, ma è una procedura consigliata.

Utilizza la tabella seguente per determinare il valore della partizione per il filtro di rilevamento.

| Regione | Partizione |
| --- | --- |
| Regioni AWS | `aws` |
| Regioni della Cina | `aws-cn` |

| Regione | Partizione |
| --- | --- |
| AWS GovCloud (US) Regions | `aws-us-gov` |

L'esempio seguente mostra come creare un filtro di rilevamento. Prima di utilizzare il codice, sostituite i valori di esempio con valori validi per la partizione Account AWS and.

Java

```
// Create the discovery filter
DiscoveryFilter discoveryFilter = DiscoveryFilter.builder()
        .partition("aws")
        .accountIds(111122223333)
        .build();
```

C# / .NET

```
var discoveryFilter = new DiscoveryFilter
{
    Partition = "aws",
    AccountIds = 111122223333
};
```

Rust

```
// Create discovery filter
let discovery_filter = DiscoveryFilter::builder()
    .partition("aws")
    .account_ids(111122223333)
    .build()?;
```

# Lavorare con database multitenant

Con AWS Database Encryption SDK, puoi configurare la crittografia lato client per i database con uno schema condiviso isolando ogni tenant con materiali di crittografia distinti. Quando prendi in considerazione un database multitenant, dedica del tempo a esaminare i requisiti di sicurezza e il modo in cui la multitenancy potrebbe influire su di essi. Ad esempio, l'utilizzo di un database

multitenant potrebbe influire sulla capacità di combinare Database Encryption SDK con un'altra soluzione di crittografia lato server AWS .

Se più utenti eseguono operazioni di crittografia all'interno del database, puoi utilizzare uno dei AWS KMS portachiavi per fornire a ciascun utente una chiave distinta da utilizzare nelle proprie operazioni crittografiche. La gestione delle chiavi dati per una soluzione di crittografia lato client multitenant può essere complicata. Ti consigliamo di organizzare i dati per tenant quando possibile. Se il tenant è identificato dai valori della chiave primaria (ad esempio, la chiave di partizione in una tabella Amazon DynamoDB), la gestione delle chiavi è più semplice.

Puoi usare il AWS KMS portachiavi per isolare ogni tenant con un portachiavi distinto e. AWS KMS AWS KMS keys In base al volume di AWS KMS chiamate effettuate per inquilino, potresti voler utilizzare il portachiavi AWS KMS gerarchico per ridurre al minimo le chiamate a. AWS KMS Il portachiavi AWS KMS Hierarchical è una soluzione di memorizzazione nella cache dei materiali crittografici che riduce il numero di AWS KMS chiamate utilizzando chiavi branch AWS KMS  protette persistenti in una tabella Amazon DynamoDB e quindi memorizzando nella cache locale i materiali chiave delle branch utilizzati nelle operazioni di crittografia e decrittografia. È necessario utilizzare il portachiavi Hierarchical per implementare la crittografia ricercabile nel database. AWS KMS

## Creazione di beacon firmati

AWS Database Encryption SDK utilizza beacon standard e beacon compositi per fornire soluzioni di crittografia ricercabili che consentono di cercare record crittografati senza decrittografare l'intero database interrogato. Tuttavia, AWS Database Encryption SDK supporta anche beacon firmati che possono essere configurati interamente a partire da campi firmati in testo semplice. I beacon firmati sono un tipo di beacon composto che indicizza ed esegue query complesse su campi e. `SIGN_ONLY` `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`

Ad esempio, se disponete di un database multitenant, potreste voler creare un beacon firmato che consenta di interrogare il database alla ricerca di record crittografati dalla chiave di un tenant specifico. Per ulteriori informazioni, consulta Interrogazione dei beacon in un database multi-tenant.

È necessario utilizzare il portachiavi AWS KMS gerarchico per creare beacon firmati.

Per configurare un beacon firmato, fornite i seguenti valori.

Java

- Configurazione del beacon composto

L'esempio seguente definisce gli elenchi delle parti firmate localmente all'interno della configurazione del beacon firmato.

```
List<CompoundBeacon> compoundBeaconList = new ArrayList<>();
CompoundBeacon exampleCompoundBeacon = CompoundBeacon.builder()
    .name("compoundBeaconName")
    .split(".")
    .signed(signedPartList)
    .constructors(constructorList)
    .build();
compoundBeaconList.add(exampleCompoundBeacon);
```

Definizione della versione del beacon

L'esempio seguente definisce gli elenchi delle parti firmate a livello globale nella versione beacon. Per ulteriori informazioni sulla definizione della versione beacon, vedete Uso dei beacon.

```
 List<BeaconVersion> beaconVersions = new ArrayList<>();
beaconVersions.add(
    BeaconVersion.builder()
        .standardBeacons(standardBeaconList)
        .compoundBeacons(compoundBeaconList)
        .signedParts(signedPartList)
        .version(1) // MUST be 1
        .keyStore(keyStore)
        .keySource(BeaconKeySource.builder()
            .single(SingleKeyStore.builder()
                .keyId(branchKeyId)
                .cacheTTL(6000)
                .build())
            .build())
        .build()
);
```

C# / .NET

Guarda l'esempio di codice completo: .cs BeaconConfig

Configurazione del beacon firmato

L'esempio seguente definisce gli elenchi delle parti firmate localmente all'interno della configurazione del beacon firmato.

```
var compoundBeaconList = new List<CompoundBeacon>();
var exampleCompoundBeacon = new CompoundBeacon
 {
    Name = "compoundBeaconName",
    Split = ".",
    Signed = signedPartList,
    Constructors = constructorList
 };
compoundBeaconList.Add(exampleCompoundBeacon);
```

Definizione della versione del beacon

L'esempio seguente definisce gli elenchi delle parti firmate a livello globale nella versione beacon. Per ulteriori informazioni sulla definizione della versione beacon, vedete Uso dei beacon.

```
var beaconVersions = new List<BeaconVersion>
{
    new BeaconVersion
    {
        StandardBeacons = standardBeaconList,
        CompoundBeacons = compoundBeaconList,
        SignedParts = signedPartsList,
        Version = 1, // MUST be 1
        KeyStore = keyStore,
        KeySource = new BeaconKeySource
        {
            Single = new SingleKeyStore
            {
                KeyId = branchKeyId,
                CacheTTL = 6000
            }
        }
    }
};
```

È possibile definire le parti firmate in elenchi definiti localmente o globalmente. Ti consigliamo di definire le parti firmate in un elenco globale nella versione beacon, quando possibile. Definendo le parti firmate a livello globale, è possibile definire ogni parte una volta e quindi riutilizzare le parti in più configurazioni beacon composte. Se intendete utilizzare una parte firmata una sola volta, potete

definirla in un elenco locale nella configurazione del beacon firmato. È possibile fare riferimento sia alle parti locali che a quelle globali nell'elenco dei costruttori.

Se definite gli elenchi di parti firmate a livello globale, dovete fornire un elenco di parti del costruttore che identifichi tutti i possibili modi in cui il beacon firmato può assemblare i campi nella configurazione del beacon.

> **Note**
>
> Per definire gli elenchi delle parti firmate a livello globale, è necessario utilizzare la versione 3.2 o successiva di Database Encryption SDK. AWS Distribuisci la nuova versione a tutti i lettori prima di definire nuove parti a livello globale.
> Non è possibile aggiornare le configurazioni dei beacon esistenti per definire elenchi di parti firmate a livello globale.

Nome del beacon

Il nome che usi quando interroghi il faro.

Il nome di un beacon firmato non può avere lo stesso nome di un campo non crittografato. Due beacon non possono avere lo stesso nome.

Carattere diviso

Il carattere usato per separare le parti che compongono il faro firmato.

Il carattere diviso non può apparire nei valori in chiaro di nessuno dei campi da cui è costruito il beacon firmato.

Elenco delle parti firmate

Identifica i campi firmati inclusi nel beacon firmato.

Ogni parte deve includere un nome, una fonte e un prefisso. L'origine è il `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` campo `SIGN_ONLY` o il campo identificato dalla parte. L'origine deve essere un nome di campo o un indice che si riferisce al valore di un campo annidato. Se il nome della parte identifica la fonte, puoi omettere la fonte e AWS Database Encryption SDK utilizzerà automaticamente il nome come fonte. Ti consigliamo di specificare l'origine come nome della parte quando possibile. Il prefisso può essere qualsiasi stringa, ma

deve essere univoco. Due parti firmate in un beacon firmato non possono avere lo stesso prefisso. Si consiglia di utilizzare un valore breve che distingua la parte dalle altre parti servite dal beacon composto.

Ti consigliamo di definire le parti firmate a livello globale quando possibile. Potresti prendere in considerazione la definizione locale di una parte firmata se intendi utilizzarla solo in un beacon composto. Una parte definita localmente non può avere lo stesso prefisso o nome di una parte definita globalmente.

Java

```
List<SignedPart> signedPartList = new ArrayList<>);
    SignedPart signedPartExample = SignedPart.builder()
        .name("signedFieldName")
        .prefix("S-")
        .build();
    signedPartList.add(signedPartExample);
```

C# / .NET

```
var signedPartsList = new List<SignedPart>
{
    new SignedPart { Name = "signedFieldName1", Prefix = "S-" },
    new SignedPart { Name = "signedFieldName2", Prefix = "SF-" }
};
```

Elenco dei costruttori (opzionale)

Identifica i costruttori che definiscono i diversi modi in cui le parti firmate possono essere assemblate dal faro firmato.

Se non specificate un elenco di costruttori, AWS Database Encryption SDK assembla il beacon firmato con il seguente costruttore predefinito.

- Tutte le parti firmate nell'ordine in cui sono state aggiunte all'elenco delle parti firmate
- Tutte le parti sono obbligatorie

Costruttori

Ogni costruttore è un elenco ordinato di parti del costruttore che definisce un modo in cui il faro firmato può essere assemblato. Le parti del costruttore vengono unite nell'ordine in cui vengono aggiunte all'elenco, con ogni parte separata dal carattere di divisione specificato.

Ogni parte del costruttore nomina una parte firmata e definisce se tale parte è obbligatoria o facoltativa all'interno del costruttore. Ad esempio, se si desidera interrogare un faro firmato su`Field1`, and `Field1.Field2``Field1.Field2.Field3`, contrassegnare e `Field3` come facoltativo `Field2` e creare un costruttore.

Ogni costruttore deve avere almeno una parte obbligatoria. Si consiglia di rendere obbligatoria la prima parte di ogni costruttore in modo da poter utilizzare l'`BEGINS_WITH`operatore nelle query.

Un costruttore ha successo se tutte le parti necessarie sono presenti nel record. Quando si scrive un nuovo record, il beacon firmato utilizza l'elenco dei costruttori per determinare se il beacon può essere assemblato in base ai valori forniti. Tenta di assemblare il beacon nell'ordine in cui i costruttori sono stati aggiunti all'elenco dei costruttori e utilizza il primo costruttore che riesce. Se nessun costruttore ha successo, il beacon non viene scritto nel record.

Tutti i lettori e gli scrittori devono specificare lo stesso ordine di costruttori per garantire che i risultati delle query siano corretti.

Utilizzate le seguenti procedure per specificare il vostro elenco di costruttori.

1. Create una parte costruttore per ogni parte firmata per definire se quella parte è necessaria o meno.

   Il nome della parte del costruttore deve essere il nome del campo firmato.

   L'esempio seguente dimostra come creare una parte costruttore per un campo firmato.

   Java

   ```
   ConstructorPart field1ConstructorPart = ConstructorPart.builder()
           .name("Field1")
           .required(true)
           .build();
   ```

   C# / .NET

   ```
   var field1ConstructorPart = new ConstructorPart { Name = "Field1", Required
    = true };
   ```

2. Create un costruttore per ogni modo possibile in cui il faro firmato può essere assemblato utilizzando le parti del costruttore create nel passaggio 1.

   Ad esempio, se si desidera eseguire un'interrogazione su `Field1.Field2.Field3` and`Field4.Field2.Field3`, è necessario creare due costruttori. `Field1`e `Field4` possono essere entrambi obbligatori perché sono definiti in due costruttori separati.

   Java

   ```
   // Create a list for Field1.Field2.Field3 queries
   List<ConstructorPart> field123ConstructorPartList = new ArrayList<>();
   field123ConstructorPartList.add(field1ConstructorPart);
   field123ConstructorPartList.add(field2ConstructorPart);
   field123ConstructorPartList.add(field3ConstructorPart);
   Constructor field123Constructor = Constructor.builder()
           .parts(field123ConstructorPartList)
           .build();
   // Create a list for Field4.Field2.Field1 queries
   List<ConstructorPart> field421ConstructorPartList = new ArrayList<>();
   field421ConstructorPartList.add(field4ConstructorPart);
   field421ConstructorPartList.add(field2ConstructorPart);
   field421ConstructorPartList.add(field1ConstructorPart);
   Constructor field421Constructor = Constructor.builder()
           .parts(field421ConstructorPartList)
           .build();
   ```

   C# / .NET

   ```
   // Create a list for Field1.Field2.Field3 queries
    var field123ConstructorPartList = new Constructor
   {
       Parts = new List<ConstructorPart> { field1ConstructorPart,
    field2ConstructorPart, field3ConstructorPart }
   };
   // Create a list for Field4.Field2.Field1 queries
   var field421ConstructorPartList = new Constructor
   {
       Parts = new List<ConstructorPart> { field4ConstructorPart,
    field2ConstructorPart, field1ConstructorPart }
   };
   ```

3. Create un elenco di costruttori che includa tutti i costruttori creati nel passaggio 2.

Java

```
List<Constructor> constructorList = new ArrayList<>();
constructorList.add(field123Constructor)
constructorList.add(field421Constructor)
```

C# / .NET

```
var constructorList = new List<Constructor>
{
    field123Constructor,
    field421Constructor
};
```

4. Specificate `constructorList` quando create il beacon firmato.

# Archivi di chiavi nel AWS Database Encryption SDK

Nel AWS Database Encryption SDK, un key store è una tabella Amazon DynamoDB che mantiene i dati gerarchici utilizzati dal portachiavi Hierarchical.AWS KMS L'archivio chiavi aiuta a ridurre il numero di chiamate necessarie per eseguire operazioni crittografiche con il portachiavi Hierarchical. AWS KMS

L'archivio chiavi persiste e gestisce le chiavi di filiale utilizzate dal portachiavi Hierarchical per eseguire la crittografia degli inviluppi e proteggere le chiavi di crittografia dei dati. Il key store memorizza la chiave di ramo attiva e tutte le versioni precedenti della chiave di filiale. La chiave di ramo attiva è la versione più recente della chiave di filiale. Il portachiavi Hierarchical utilizza una chiave di crittografia dei dati unica per ogni richiesta di crittografia e crittografa ogni chiave di crittografia dei dati con una chiave di wrapping unica derivata dalla chiave branch attiva. Il portachiavi Hierarchical dipende dalla gerarchia stabilita tra le chiavi branch attive e le relative chiavi di wrapping derivate.

## Terminologia e concetti del Key Store

Key store (Archivio chiavi)

: La tabella DynamoDB che mantiene i dati gerarchici, come le chiavi di filiale e le chiavi beacon.

Chiave principale

: Una chiave KMS con crittografia simmetrica che genera e protegge le chiavi branch e le chiavi beacon nell'archivio delle chiavi.

Chiave di filiale

: Una chiave dati che viene riutilizzata per ricavare una chiave di avvolgimento univoca per la crittografia delle buste. È possibile creare più chiavi di filiale in un unico archivio di chiavi, ma ogni chiave di ramo può avere solo una versione di chiave di ramo attiva alla volta. La chiave di ramo attiva è la versione più recente della chiave di filiale.

  Le chiavi di filiale derivano dall' AWS KMS keys uso dell'GenerateDataKeyWithoutPlaintextoperazione kms:.

Chiave di avvolgimento

: Una chiave dati unica utilizzata per crittografare la chiave di crittografia dei dati utilizzata nelle operazioni di crittografia.

Le chiavi di wrapping derivano dalle chiavi di filiale. Per ulteriori informazioni sul processo di derivazione delle chiavi, consulta Dettagli tecnici del portachiavi AWS KMS gerarchico.

Chiave di crittografia dei dati

Una chiave dati utilizzata nelle operazioni di crittografia. Il portachiavi Hierarchical utilizza una chiave di crittografia dei dati unica per ogni richiesta di crittografia.

Chiave Beacon

Una chiave dati utilizzata per generare beacon per la crittografia ricercabile. Per ulteriori informazioni, vedere Crittografia ricercabile.

# Implementazione di autorizzazioni con privilegio minimo

Quando si utilizza un archivio chiavi e portachiavi AWS KMS gerarchici, si consiglia di seguire il principio del privilegio minimo definendo i seguenti ruoli:

Amministratore del negozio di chiavi

Gli amministratori dell'archivio chiavi sono responsabili della creazione e della gestione dell'archivio chiavi e delle chiavi di filiale che esso persiste e protegge. Gli amministratori del key store devono essere gli unici utenti con autorizzazioni di scrittura per la tabella Amazon DynamoDB che funge da archivio chiavi. Dovrebbero essere gli unici utenti con accesso a operazioni amministrative privilegiate, come e. `CreateKeyVersionKey` È possibile eseguire queste operazioni solo quando si configurano staticamente le azioni dell'archivio delle chiavi.

`CreateKey`è un'operazione privilegiata che può aggiungere una nuova chiave KMS ARN alla lista delle autorizzazioni dell'archivio chiavi. Questa chiave KMS può creare nuove chiavi di filiale attive. Consigliamo di limitare l'accesso a questa operazione perché una volta aggiunta una chiave KMS all'archivio delle chiavi della filiale, non può essere eliminata.

Utente del Key Store

Nella maggior parte dei casi d'uso, l'utente dell'archivio chiavi interagisce con l'archivio chiavi solo tramite il portachiavi gerarchico mentre crittografa, decrittografa, firma e verifica i dati. Di conseguenza, necessitano solo delle autorizzazioni di lettura per la tabella Amazon DynamoDB che funge da archivio delle chiavi. Gli utenti del Key Store devono poter accedere solo alle operazioni di utilizzo che rendono possibili le operazioni crittografiche, ad `GetActiveBranchKey` esempio, e. `GetBranchKeyVersion` `GetBeaconKey` Non hanno bisogno di autorizzazioni per creare o gestire le chiavi di filiale che utilizzano.

È possibile eseguire operazioni di utilizzo quando le azioni dell'archivio chiavi sono configurate staticamente o quando sono configurate per il rilevamento. Non è possibile eseguire operazioni di amministratore (`CreateKeyeVersionKey`) quando le azioni dell'archivio chiavi sono configurate per il rilevamento.

Se l'amministratore del negozio di chiavi della filiale ha consentito l'inserimento di più chiavi KMS nell'archivio chiavi della filiale, consigliamo agli utenti dell'archivio chiavi di configurare le azioni del proprio archivio chiavi per il rilevamento in modo che il loro portachiavi gerarchico possa utilizzare più chiavi KMS.

## Creare un archivio di chiavi

Prima di poter creare chiavi di filiale o utilizzare un portachiavi AWS KMS gerarchico, devi creare il tuo key store, una tabella Amazon DynamoDB che gestisca e protegga le tue chiavi di filiale.

> **Important**
>
> Non eliminare la tabella DynamoDB che mantiene le chiavi di filiale. Se elimini questa tabella, non sarai in grado di decrittografare i dati crittografati utilizzando il portachiavi gerarchico.

Segui le procedure di creazione di una tabella nella Amazon DynamoDB Developer Guide, utilizzando i seguenti valori di stringa richiesti per la chiave di partizione e la chiave di ordinamento.

| | Chiave di partizione | Chiave di ordinamento |
|---|---|---|
| Tabella di base | `branch-key-id` | `type` |

Nome dell'archivio di chiavi logiche

Quando si assegna un nome alla tabella DynamoDB che funge da archivio chiavi, è importante considerare attentamente il nome dell'archivio di chiavi logico da specificare durante la configurazione delle azioni dell'archivio chiavi. Il nome dell'archivio logico delle chiavi funge da identificatore per l'archivio delle chiavi e non può essere modificato dopo essere stato inizialmente definito dal primo utente. È necessario specificare sempre lo stesso nome dell'archivio di chiavi logiche nelle azioni dell'archivio chiavi.

Deve esserci una one-to-one mappatura tra il nome della tabella DynamoDB e il nome dell'archivio delle chiavi logiche. Il nome dell'archivio di chiavi logiche è associato crittograficamente a tutti i dati memorizzati nella tabella per semplificare le operazioni di ripristino di DynamoDB. Sebbene il nome dell'archivio di chiavi logiche possa essere diverso dal nome della tabella DynamoDB, consigliamo vivamente di specificare il nome della tabella DynamoDB come nome dell'archivio di chiavi logiche. Nel caso in cui il nome della tabella cambi dopo il ripristino della tabella DynamoDB da un backup, il nome dell'archivio delle chiavi logiche può essere mappato al nuovo nome della tabella DynamoDB per garantire che il portachiavi Hierarchical possa ancora accedere al tuo key store.

Non includere informazioni riservate o sensibili nel nome dell'archivio di chiavi logiche. Il nome dell'archivio di chiavi logiche viene visualizzato in testo semplice negli AWS KMS CloudTrail eventi come. `tablename`

Passaggi successivi

1. the section called “Configurare le azioni del key store”
2. the section called “Crea chiavi di ramo”
3. Crea un portachiavi gerarchico AWS KMS

# Configurare le azioni del key store

Le azioni dell'archivio chiavi determinano quali operazioni possono eseguire gli utenti e in che modo il loro portachiavi AWS KMS gerarchico utilizza le chiavi KMS consentite elencate nell'archivio delle chiavi. Il AWS Database Encryption SDK supporta le seguenti configurazioni di key store action.

Statico

Quando configuri staticamente il tuo archivio chiavi, l'archivio chiavi può utilizzare solo la chiave KMS associata all'ARN della chiave KMS che fornisci `kmsConfiguration` quando configuri le azioni dell'archivio chiavi. Viene generata un'eccezione se viene rilevata una chiave KMS ARN diversa durante la creazione, il controllo delle versioni o l'ottenimento di una chiave branch.

Puoi specificare una chiave KMS multiregionale nel tuo`kmsConfiguration`, ma l'intero ARN della chiave, inclusa la regione, viene mantenuto nelle chiavi branch derivate dalla chiave KMS. Non è possibile specificare una chiave in una regione diversa, è necessario fornire esattamente la stessa chiave multiregionale affinché i valori corrispondano.

Quando configuri staticamente le azioni dell'archivio delle chiavi, puoi eseguire operazioni di utilizzo (`GetActiveBranchKey`,`GetBranchKeyVersion`,`GetBeaconKey`) e operazioni

amministrative (`CreateKeyeVersionKey`). `CreateKey`è un'operazione privilegiata che può aggiungere una nuova chiave KMS ARN alla lista delle autorizzazioni dell'archivio chiavi. Questa chiave KMS può creare nuove chiavi di filiale attive. Consigliamo di limitare l'accesso a questa operazione perché una volta aggiunta una chiave KMS all'archivio chiavi, non può essere eliminata.

Individuazione

Quando configuri le azioni dell'archivio chiavi per il rilevamento, l'archivio chiavi può utilizzare qualsiasi AWS KMS key ARN consentito nell'archivio delle chiavi. Tuttavia, viene generata un'eccezione quando viene rilevata una chiave KMS multiregionale e la regione nell'ARN della chiave non corrisponde alla regione del client utilizzato. AWS KMS

Quando si configura l'archivio delle chiavi per il rilevamento, non è possibile eseguire operazioni amministrative, come e. `CreateKey` `VersionKey` È possibile eseguire solo le operazioni di utilizzo che consentono le operazioni di crittografia, decrittografia, firma e verifica. Per ulteriori informazioni, consulta the section called “Implementazione di autorizzazioni con privilegio minimo”.

## Configura le azioni del tuo key store

Prima di configurare le azioni del tuo key store, assicurati che siano soddisfatti i seguenti prerequisiti.

- Determinate quali operazioni dovete eseguire. Per ulteriori informazioni, consulta the section called “Implementazione di autorizzazioni con privilegio minimo”.
- Scegliete il nome di un archivio di chiavi logiche

  Deve esserci una one-to-one mappatura tra il nome della tabella DynamoDB e il nome dell'archivio delle chiavi logiche. Il nome dell'archivio di chiavi logiche è associato crittograficamente a tutti i dati memorizzati nella tabella per semplificare le operazioni di ripristino di DynamoDB e non può essere modificato dopo essere stato inizialmente definito dal primo utente. È necessario specificare sempre lo stesso nome dell'archivio di chiavi logiche nelle azioni dell'archivio chiavi. Per ulteriori informazioni, consulta logical key store name.

Configurazione statica

L'esempio seguente configura staticamente le azioni di archiviazione delle chiavi. È necessario specificare il nome della tabella DynamoDB che funge da archivio chiavi, un nome logico per l'archivio chiavi e l'ARN della chiave KMS che identifica una chiave KMS di crittografia simmetrica.

> **Note**
>
> Valuta attentamente l'ARN della chiave KMS che specifichi durante la configurazione statica del servizio di archiviazione delle chiavi. L'`CreateKeyoperazione` aggiunge l'ARN della chiave KMS alla lista delle autorizzazioni dell'archivio chiavi della filiale. Una volta aggiunta una chiave KMS all'archivio delle chiavi della filiale, non può essere eliminata.

Java

```
final KeyStore keystore = KeyStore.builder().KeyStoreConfig(
                  KeyStoreConfig.builder()
                          .ddbClient(DynamoDbClient.create())
                          .ddbTableName(keyStoreName)
                          .logicalKeyStoreName(logicalKeyStoreName)
                          .kmsClient(KmsClient.create())
                          .kmsConfiguration(KMSConfiguration.builder()
                                  .kmsKeyArn(kmsKeyArn)
                                  .build())
                          .build()).build();
```

C# / .NET

```
var kmsConfig = new KMSConfiguration { KmsKeyArn = kmsKeyArn };
  var keystoreConfig = new KeyStoreConfig
  {
      KmsClient = new AmazonKeyManagementServiceClient(),
      KmsConfiguration = kmsConfig,
      DdbTableName = keyStoreName,
      DdbClient = new AmazonDynamoDBClient(),
      LogicalKeyStoreName = logicalKeyStoreName
  };
  var keystore = new KeyStore(keystoreConfig);
```

Rust

```
let sdk_config =
 aws_config::load_defaults(aws_config::BehaviorVersion::latest()).await;
let key_store_config = KeyStoreConfig::builder()
    .kms_client(aws_sdk_kms::Client::new(&sdk_config))
    .ddb_client(aws_sdk_dynamodb::Client::new(&sdk_config))
```

```
    .ddb_table_name(key_store_name)
    .logical_key_store_name(logical_key_store_name)
    .kms_configuration(KmsConfiguration::KmsKeyArn(kms_key_arn.to_string()))
    .build()?;

let keystore = keystore_client::Client::from_conf(key_store_config)?;
```

### Configurazione Discovery

L'esempio seguente configura le azioni di archiviazione delle chiavi per il rilevamento. È necessario specificare il nome della tabella DynamoDB che funge da archivio chiavi e il nome dell'archivio di chiavi logico.

Java

```
final KeyStore keystore = KeyStore.builder().KeyStoreConfig(
                KeyStoreConfig.builder()
                        .ddbClient(DynamoDbClient.create())
                        .ddbTableName(keyStoreName)
                        .logicalKeyStoreName(logicalKeyStoreName)
                        .kmsClient(KmsClient.create())
                        .kmsConfiguration(KMSConfiguration.builder()
                                .discovery(Discovery.builder().build())
                                .build())
                        .build()).build();
```

C# / .NET

```
var keystoreConfig = new KeyStoreConfig
 {
     KmsClient = new AmazonKeyManagementServiceClient(),
     KmsConfiguration = new KMSConfiguration {Discovery = new Discovery()},
     DdbTableName = keyStoreName,
     DdbClient = new AmazonDynamoDBClient(),
     LogicalKeyStoreName = logicalKeyStoreName
 };
 var keystore = new KeyStore(keystoreConfig);
```

Rust

```
let key_store_config = KeyStoreConfig::builder()
```

```
        .kms_client(kms_client)
        .ddb_client(ddb_client)
        .ddb_table_name(key_store_name)
        .logical_key_store_name(logical_key_store_name)

.kms_configuration(KmsConfiguration::Discovery(Discovery::builder().build()?))
        .build()?;
```

## Creare una chiave di ramo attiva

Una chiave branch è una chiave dati derivata da an AWS KMS key utilizzata dal portachiavi AWS KMS Hierarchical per ridurre il numero di chiamate effettuate. AWS KMS La chiave di filiale attiva è la versione più recente della chiave di filiale. Il portachiavi Hierarchical genera una chiave dati unica per ogni richiesta di crittografia e crittografa ogni chiave di dati con una chiave di wrapping unica derivata dalla chiave branch attiva.

Per creare una nuova chiave branch attiva, devi configurare staticamente le azioni del tuo key store. `CreateKey`è un'operazione privilegiata che aggiunge l'ARN della chiave KMS specificato nella configurazione delle azioni dell'archivio chiavi all'elenco delle autorizzazioni dell'archivio chiavi. Quindi, la chiave KMS viene utilizzata per generare la nuova chiave di ramo attiva. Consigliamo di limitare l'accesso a questa operazione perché una volta aggiunta una chiave KMS all'archivio chiavi, non può essere eliminata.

Ti consigliamo di utilizzare l'`CreateKey`operazione tramite l'interfaccia di KeyStore amministrazione nel piano di controllo dell'applicazione. Questo approccio è in linea con le best practice per la gestione delle chiavi.

Non create chiavi di filiale nel piano dati. Questa pratica può portare a:

- Chiamate non necessarie a AWS KMS
- Chiamate multiple simultanee verso ambienti AWS KMS ad alta concorrenza
- TransactWriteItems Chiamate multiple alla tabella DynamoDB di backup.

L'`CreateKey`operazione include un controllo delle condizioni nella `TransactWriteItems` chiamata per impedire la sovrascrittura delle chiavi di filiale esistenti. Tuttavia, la creazione di chiavi nel piano dati può comunque comportare un utilizzo inefficiente delle risorse e potenziali problemi di prestazioni.

Puoi inserire una chiave KMS nell'archivio delle chiavi oppure puoi inserire più chiavi KMS aggiornando l'ARN della chiave KMS specificato nella configurazione delle azioni dell'archivio chiavi e richiamando nuovamente. `CreateKey` Se consenti l'inserimento di più chiavi KMS, gli utenti del tuo key store devono configurare le azioni di rilevamento delle chiavi in modo che possano utilizzare tutte le chiavi consentite nell'archivio chiavi a cui hanno accesso. Per ulteriori informazioni, consulta the section called “Configurare le azioni del key store”.

Autorizzazioni richieste

Per creare chiavi branch, hai bisogno delle ReEncrypt autorizzazioni kms: GenerateDataKeyWithoutPlaintext e kms: sulla chiave KMS specificata nelle azioni del tuo key store.

Crea una chiave di filiale

La seguente operazione crea una nuova chiave di ramo attiva utilizzando la chiave KMS specificata nella configurazione delle azioni dell'archivio chiavi e aggiunge la chiave di ramo attiva alla tabella DynamoDB che funge da archivio chiavi.

Quando si chiama`CreateKey`, è possibile scegliere di specificare i seguenti valori opzionali.

- `branchKeyIdentifier`: definisce una personalizzazione`branch-key-id`.

  Per creare una personalizzazione`branch-key-id`, è necessario includere anche un contesto di crittografia aggiuntivo con il `encryptionContext` parametro.
- `encryptionContext`: definisce un set opzionale di coppie chiave-valore non segrete che fornisce dati autenticati aggiuntivi (AAD) nel contesto di crittografia incluso nella chiamata kms:. GenerateDataKeyWithoutPlaintext

  Questo contesto di crittografia aggiuntivo viene visualizzato con il prefisso. `aws-crypto-ec:`

Java

```
final Map<String, String> additionalEncryptionContext =
 Collections.singletonMap("Additional Encryption Context for",
         "custom branch key id");

  final String BranchKey = keystore.CreateKey(
         CreateKeyInput.builder()
                 .branchKeyIdentifier(custom-branch-key-id) //OPTIONAL
                 .encryptionContext(additionalEncryptionContext) //OPTIONAL
```

```
                    .build()).branchKeyIdentifier();
```

C# / .NET

```
var additionalEncryptionContext = new Dictionary<string, string>();
  additionalEncryptionContext.Add("Additional Encryption Context for", "custom
 branch key id");

  var branchKeyId = keystore.CreateKey(new CreateKeyInput
  {
      BranchKeyIdentifier = "custom-branch-key-id", // OPTIONAL
      EncryptionContext = additionalEncryptionContext // OPTIONAL
  });
```

Rust

```
let additional_encryption_context = HashMap::from([
    ("Additional Encryption Context for".to_string(), "custom branch key
 id".to_string())
]);

let branch_key_id = keystore.create_key()
    .branch_key_identifier("custom-branch-key-id") // OPTIONAL
    .encryption_context(additional_encryption_context) // OPTIONAL
    .send()
    .await?
    .branch_key_identifier
    .unwrap();
```

Innanzitutto, l'`CreateKey`operazione genera i seguenti valori.

- Un identificatore univoco universale (UUID) versione 4 per (a meno che non sia stato specificato un identificatore personalizzato). `branch-key-id` `branch-key-id`
- Un UUID versione 4 per la versione branch key
- A `timestamp` nel formato di data e ora ISO 8601 in formato UTC (Coordinated Universal Time).

Quindi, l'`CreateKey`operazione chiama kms: GenerateDataKeyWithoutPlaintext utilizzando la seguente richiesta.

```
{
```

```
    "EncryptionContext": {
       "branch-key-id" : "branch-key-id",
       "type" : "type",
       "create-time" : "timestamp",
       "logical-key-store-name" : "the logical table name for your key store",
       "kms-arn" : the KMS key ARN,
       "hierarchy-version" : "1",
       "aws-crypto-ec:contextKey": "contextValue"
    },
    "KeyId": "the KMS key ARN you specified in your key store actions",
    "NumberOfBytes": "32"
 }
```

> **Note**
>
> L'`CreateKey`operazione crea una chiave branch attiva e una chiave beacon, anche se il database non è stato configurato per la crittografia ricercabile. Entrambe le chiavi sono memorizzate nell'archivio delle chiavi. Per ulteriori informazioni, vedere Utilizzo del portachiavi gerarchico per la crittografia ricercabile.

Successivamente, l'`CreateKey`operazione chiama kms: ReEncrypt per creare un record attivo per la chiave branch aggiornando il contesto di crittografia.

Infine, l'**`CreateKey`**operazione chiama ddb: TransactWriteItems per scrivere un nuovo elemento che mantenga la chiave di ramo nella tabella creata nel passaggio 2. L'elemento ha i seguenti attributi.

```
{
      "branch-key-id" : branch-key-id,
      "type" : "branch:ACTIVE",
      "enc" : the branch key returned by the GenerateDataKeyWithoutPlaintext call,
      "version": "branch:version:the branch key version UUID",
      "create-time" : "timestamp",
      "kms-arn" : "the KMS key ARN you specified in Step 1",
      "hierarchy-version" : "1",
      "aws-crypto-ec:contextKey": "contextValue"
 }
```

# Ruota la chiave branch attiva

Può esserci una sola versione attiva per ogni chiave di ramo alla volta. In genere, ogni versione attiva della chiave di filiale viene utilizzata per soddisfare più richieste. Tuttavia, è possibile controllare la misura in cui le chiavi di ramo attive vengono riutilizzate e determinare la frequenza con cui la chiave di ramo attiva viene ruotata.

Le chiavi branch non vengono utilizzate per crittografare le chiavi di dati in testo semplice. Vengono utilizzate per derivare le chiavi di wrapping univoche che crittografano le chiavi di dati in testo non crittografato. Il processo di derivazione della chiave di wrapping produce una chiave di wrapping unica da 32 byte con 28 byte di casualità. Ciò significa che una chiave branch può derivare più di 79 ottilioni, o 2 96, chiavi di wrapping uniche prima che si verifichi l'usura crittografica. Anche se questo rischio di esaurimento è bassissimo, potrebbe essere necessario ruotare le chiavi di filiale attive a causa di regole aziendali/contrattuali o norme governative.

La versione attiva della chiave di filiale rimane attiva finché non viene ruotata. Le versioni precedenti della chiave branch attiva non verranno utilizzate per eseguire operazioni di crittografia e non potranno essere utilizzate per derivare nuove chiavi di wrapping, ma possono comunque essere interrogate e fornire chiavi di wrapping per decrittografare le chiavi di dati che crittografavano mentre erano attive.

> **Warning**
>
> L'eliminazione delle chiavi di filiale negli ambienti di test è irreversibile. Non è possibile recuperare le chiavi branch eliminate. Quando si eliminano e si ricreano le chiavi di filiale con lo stesso ID in ambienti di test, possono verificarsi i seguenti problemi:
>
> - I materiali dei test precedenti potrebbero rimanere nella cache
> - Alcuni host o thread di test potrebbero crittografare i dati utilizzando chiavi branch eliminate
> - I dati crittografati con rami eliminati non possono essere decrittografati
>
> Per evitare errori di crittografia nei test di integrazione:
>
> - Reimposta il riferimento gerarchico del portachiavi prima di creare nuove chiavi di filiale OPPURE
> - Usa una chiave IDs di filiale unica per ogni test

### Autorizzazioni richieste

Per ruotare le chiavi branch, hai bisogno delle ReEncrypt autorizzazioni kms: GenerateDataKeyWithoutPlaintext e kms: sulla chiave KMS specificata nelle azioni del tuo key store.

### Ruota una chiave branch attiva

Usa l'`VersionKey`operazione per ruotare la chiave branch attiva. Quando si ruota la chiave di ramo attiva, viene creata una nuova chiave di ramo per sostituire la versione precedente. Non `branch-key-id` cambia quando si ruota la chiave di ramo attiva. È necessario specificare la chiave `branch-key-id` che identifica la chiave di ramo attiva corrente quando si chiama. `VersionKey`

Java

```
keystore.VersionKey(
    VersionKeyInput.builder()
        .branchKeyIdentifier("branch-key-id")
        .build()
);
```

C# / .NET

```
 keystore.VersionKey(new VersionKeyInput{BranchKeyIdentifier = branchKeyId});
```

Rust

```
keystore.version_key()
        .branch_key_identifier(branch_key_id)
        .send()
        .await?;
```

# Portachiavi

La nostra libreria di crittografia lato client è stata rinominata Database Encryption SDK. AWS Questa guida per sviluppatori fornisce ancora informazioni sul DynamoDB Encryption Client.

Il AWS Database Encryption SDK utilizza i portachiavi per eseguire la crittografia delle buste. I keyring generano, crittografano e decrittano le chiavi di dati. I portachiavi determinano l'origine delle chiavi dati univoche che proteggono ogni record crittografato e delle chiavi di avvolgimento che crittografano tale chiave di dati. Puoi specificare un keyring durante la crittografia e lo stesso keyring o uno diverso durante la decrittazione.

I keyring possono essere utilizzati singolarmente o combinati in keyring multipli. Anche se la maggior parte dei keyring è in grado di generare, crittografare e decrittare le chiavi di dati, ne puoi creare uno che esegua solo una determinata operazione, ad esempio la generazione delle chiavi di dati, e utilizzarlo in combinazione con altri.

Ti consigliamo di utilizzare un portachiavi che protegga le tue chiavi di wrapping ed esegua operazioni crittografiche all'interno di un limite sicuro, come il AWS KMS portachiavi, che utilizza that never leave () unencrypted. AWS KMS keys AWS Key Management ServiceAWS KMS Puoi anche scrivere un portachiavi che utilizzi chiavi di avvolgimento archiviate nei moduli di sicurezza hardware (HSMs) o protette da altri servizi di chiavi principali.

Il portachiavi determina le chiavi di avvolgimento che proteggono le chiavi dati e, in ultima analisi, i dati. Utilizzate le chiavi di avvolgimento più sicure e pratiche per il vostro compito. Se possibile, utilizzate chiavi di wrapping protette da un modulo di sicurezza hardware (HSM) o da un'infrastruttura di gestione delle chiavi, come le chiavi KMS in AWS Key Management Service(AWS KMS) o le chiavi di crittografia in. AWS CloudHSM

AWS Database Encryption SDK offre diversi portachiavi e configurazioni di portachiavi ed è possibile creare portachiavi personalizzati. Puoi anche creare un portachiavi multiplo che includa uno o più portachiavi dello stesso tipo o di un tipo diverso.

Argomenti

- Come funzionano i keyring
- AWS KMS portachiavi
- AWS KMS Portachiavi gerarchici

- AWS KMS Portachiavi ECDH
- Keyring non elaborati AES
- Keyring non elaborato RSA
- Portachiavi ECDH grezzi
- Keyring multipli

# Come funzionano i keyring

La nostra libreria di crittografia lato client è stata rinominata Database Encryption SDK. AWS Questa guida per sviluppatori fornisce ancora informazioni sul DynamoDB Encryption Client.

Quando si crittografa e si firma un campo nel database, il Database Encryption SDK richiede al AWS portachiavi i materiali di crittografia. Il portachiavi restituisce una chiave dati in testo semplice, una copia della chiave dati crittografata da ciascuna delle chiavi di avvolgimento del portachiavi e una chiave MAC associata alla chiave dati. AWS Database Encryption SDK utilizza la chiave di testo semplice per crittografare i dati, quindi rimuove la chiave dati in chiaro dalla memoria il prima possibile. Quindi, AWS Database Encryption SDK aggiunge una descrizione del materiale che include le chiavi dei dati crittografati e altre informazioni, come le istruzioni di crittografia e firma. AWS Database Encryption SDK utilizza la chiave MAC per calcolare i codici di autenticazione dei messaggi basati su hash (HMACs) sulla canonicalizzazione della descrizione del materiale e di tutti i campi contrassegnati con o. `ENCRYPT_AND_SIGN` `SIGN_ONLY`

Quando decifri i dati, puoi utilizzare lo stesso portachiavi che hai usato per crittografare i dati o uno diverso. Per decrittografare i dati, un portachiavi di decrittografia deve avere accesso ad almeno una chiave di avvolgimento nel portachiavi di crittografia.

Il AWS Database Encryption SDK passa le chiavi dei dati crittografati dalla descrizione del materiale al portachiavi e chiede al portachiavi di decrittografarle tutte. Il keyring utilizza le chiavi di wrapping per decrittare una delle chiavi di dati crittografate e restituisce una chiave di dati di testo normale. Il AWS Database Encryption SDK utilizza la chiave dati in testo semplice per decrittografare i dati. Se nessuna delle chiavi di wrapping nel keyring è in grado di decrittare una qualsiasi delle chiavi di dati crittografate, l'operazione di decrittazione non riesce.

Puoi utilizzare un singolo keyring o combinarne più di uno dello stesso tipo o di tipi diversi in un keyring multiplo. Quando si crittografano i dati, il portachiavi multiplo restituisce una copia della

chiave dati crittografata da tutte le chiavi di avvolgimento in tutti i portachiavi che comprendono il portachiavi multiplo e una chiave MAC associata alla chiave dati. È possibile decrittografare i dati utilizzando un portachiavi con una qualsiasi delle chiavi di avvolgimento del portachiavi multiplo.

# AWS KMS portachiavi

> La nostra libreria di crittografia lato client è stata rinominata Database Encryption SDK. AWS Questa guida per sviluppatori fornisce ancora informazioni sul DynamoDB Encryption Client.

Un AWS KMS portachiavi utilizza la crittografia simmetrica o RSA asimmetrica per generare, crittografare e AWS KMS keysdecrittografare le chiavi di dati. AWS Key Management Service (AWS KMS) protegge le chiavi KMS ed esegue operazioni crittografiche entro i confini FIPS. Ti consigliamo di utilizzare un AWS KMS portachiavi o un portachiavi con proprietà di sicurezza simili, quando possibile.

Puoi anche utilizzare una chiave KMS simmetrica multiregionale in un portachiavi. AWS KMS Per ulteriori dettagli ed esempi di utilizzo di più regioni, vedere. AWS KMS keysUtilizzo di più regioni AWS KMS keys Per informazioni sulle chiavi multiregionali, consulta Uso delle chiavi multiregionali nella Guida per gli AWS Key Management Service sviluppatori.

AWS KMS i portachiavi possono includere due tipi di chiavi avvolgenti:

- Chiave generatrice: genera una chiave di dati in testo semplice e la crittografa. Un portachiavi che crittografa i dati deve avere una chiave generatrice.
- Chiavi aggiuntive: crittografa la chiave di dati in testo semplice generata dalla chiave del generatore. AWS KMS I portachiavi possono avere zero o più chiavi aggiuntive.

È necessario disporre di una chiave generatrice per crittografare i record. Quando un AWS KMS portachiavi ha una sola AWS KMS chiave, tale chiave viene utilizzata per generare e crittografare la chiave dati.

Come tutti i portachiavi, i AWS KMS portachiavi possono essere utilizzati indipendentemente o in un portachiavi multiplo con altri portachiavi dello stesso tipo o di un tipo diverso.

Argomenti

- AWS KMS Autorizzazioni richieste per i portachiavi

- Identificazione AWS KMS keys in un portachiavi AWS KMS
- Creazione di un portachiavi AWS KMS
- Utilizzo di più regioni AWS KMS keys
- Utilizzo di un portachiavi Discovery AWS KMS
- Utilizzo di un portachiavi AWS KMS Regional Discovery

## AWS KMS Autorizzazioni richieste per i portachiavi

Il AWS Database Encryption SDK non richiede Account AWS e non dipende da nessuno. Servizio AWS Tuttavia, per utilizzare un AWS KMS portachiavi, sono necessarie le seguenti autorizzazioni Account AWS minime sul AWS KMS keys portachiavi.

- Per crittografare con un AWS KMS portachiavi, è necessaria l'autorizzazione kms: GenerateDataKey sulla chiave del generatore. È necessaria l'autorizzazione KMS:Encrypt su tutte le chiavi aggiuntive nel portachiavi. AWS KMS
- Per decriptare con un AWS KMS portachiavi, è necessaria l'autorizzazione KMS:Decrypt su almeno una chiave del portachiavi. AWS KMS
- Per crittografare con un portachiavi multiplo composto da portachiavi, è necessaria l'autorizzazione kms: sulla AWS KMS chiave del generatore nel portachiavi del generatore. GenerateDataKey È necessaria l'autorizzazione KMS:Encrypt su tutte le altre chiavi in tutti gli altri portachiavi. AWS KMS
- Per crittografare con un AWS KMS portachiavi RSA asimmetrico, non è necessario kms: GenerateDataKey o KMS:Encrypt perché è necessario specificare il materiale della chiave pubblica che si desidera utilizzare per la crittografia quando si crea il portachiavi. Non viene effettuata alcuna chiamata durante la crittografia con questo portachiavi. AWS KMS Per decrittografare con un portachiavi AWS KMS RSA asimmetrico, è necessaria l'autorizzazione KMS:Decrypt.

Per informazioni dettagliate sulle autorizzazioni per, consulta Authentication and access control nella Developer Guide. AWS KMS keysAWS Key Management Service

## Identificazione AWS KMS keys in un portachiavi AWS KMS

Un AWS KMS portachiavi può includerne uno o più. AWS KMS keys Per specificare un elemento AWS KMS key in un AWS KMS portachiavi, utilizzate un identificatore di AWS KMS chiave supportato. Gli identificatori di chiave che è possibile utilizzare per identificare un elemento AWS

KMS key in un portachiavi variano a seconda dell'operazione e dell'implementazione del linguaggio. Per informazioni dettagliate sugli identificatori chiave di an AWS KMS key, consulta Key Identifiers nella Developer Guide.AWS Key Management Service

Come procedura consigliata, utilizzate l'identificatore di chiave più specifico e pratico per la vostra attività.

- Per crittografare con un AWS KMS portachiavi, puoi utilizzare un ID chiave, un ARN di chiave, un nome alias o un alias ARN per crittografare i dati.

  > **Note**
  >
  > Se si specifica un nome alias o un alias ARN per una chiave KMS in un portachiavi di crittografia, l'operazione di crittografia salva la chiave ARN attualmente associata all'alias nei metadati della chiave dati crittografata. Non salva l'alias. Le modifiche all'alias non influiscono sulla chiave KMS utilizzata per decrittografare le chiavi di dati crittografate.

- Per decrittografare con un AWS KMS portachiavi, è necessario utilizzare una chiave ARN per l'identificazione. AWS KMS keys Per informazioni dettagliate, consultare Selezione delle chiavi di avvolgimento.
- In un keyring utilizzato per la crittografia e la decrittazione devi utilizzare un ARN di chiave per identificare le AWS KMS keys.

Durante la decrittografia, AWS Database Encryption SDK cerca nel portachiavi una soluzione in AWS KMS key grado di decrittografare una delle AWS KMS chiavi di dati crittografate. In particolare, AWS Database Encryption SDK utilizza lo schema seguente per ogni chiave di dati crittografata nella descrizione del materiale.

- Il AWS Database Encryption SDK ottiene l'ARN della AWS KMS key chiave che ha crittografato la chiave dati dai metadati della descrizione del materiale.
- Il AWS Database Encryption SDK cerca nel portachiavi di decrittografia un ARN con AWS KMS key una chiave corrispondente.
- Se trova un ARN AWS KMS key con una chiave ARN corrispondente nel portachiavi, AWS Database Encryption SDK chiede di utilizzare la chiave KMS per AWS KMS decrittografare la chiave dati crittografata.
- In caso contrario, passa alla chiave di dati crittografata successiva, se presente.

# Creazione di un portachiavi AWS KMS

È possibile configurare ogni AWS KMS portachiavi con uno AWS KMS key o più portachiavi nello stesso e AWS KMS keys in modo diverso Account AWS . Regioni AWS AWS KMS key Deve essere una chiave di crittografia simmetrica (`SYMMETRIC_DEFAULT`) o una chiave RSA KMS asimmetrica. È inoltre possibile utilizzare una chiave KMS multiregionale con crittografia simmetrica. È possibile utilizzare uno o più AWS KMS portachiavi in un portachiavi multiplo.

È possibile creare un AWS KMS portachiavi che crittografa e decrittografa i dati oppure creare AWS KMS portachiavi specifici per crittografare o decrittografare. Quando si crea un AWS KMS portachiavi per crittografare i dati, è necessario specificare una chiave generatrice, AWS KMS key che viene utilizzata per generare una chiave di dati in testo semplice e crittografarla. La chiave dati non è matematicamente correlata alla chiave KMS. Quindi, se lo desideri, puoi specificarne altre AWS KMS keys che crittografano la stessa chiave di dati in testo normale. Per decrittografare un campo crittografato protetto da questo portachiavi, il portachiavi di decrittografia utilizzato deve includere almeno uno dei valori definiti nel portachiavi, altrimenti no. AWS KMS keys AWS KMS keys(Un AWS KMS portachiavi senza è noto come portachiavi Discovery. AWS KMS keys )AWS KMS

Tutte le chiavi di inserimento in un portachiavi crittografico o in un portachiavi multiplo devono essere in grado di crittografare la chiave dati. Se una chiave di wrapping non riesce a crittografare, il metodo di crittografia fallisce. Di conseguenza, il chiamante deve disporre delle autorizzazioni necessarie per tutte le chiavi del portachiavi. Se si utilizza un portachiavi Discovery per crittografare i dati, da solo o in un portachiavi multiplo, l'operazione di crittografia non riesce.

Gli esempi seguenti utilizzano il `CreateAwsKmsMrkMultiKeyring` metodo per creare un AWS KMS portachiavi con una chiave KMS di crittografia simmetrica. Il `CreateAwsKmsMrkMultiKeyring` metodo crea automaticamente il AWS KMS client e garantisce che il portachiavi gestisca correttamente sia le chiavi a regione singola che a più regioni. Questi esempi utilizzano una chiave ARNs per identificare le chiavi KMS. Per maggiori dettagli, consulta Identificazione AWS KMS keys in un portachiavi AWS KMS.

Java

```
final MaterialProviders matProv = MaterialProviders.builder()
        .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
        .build();
final CreateAwsKmsMrkMultiKeyringInput keyringInput =
 CreateAwsKmsMrkMultiKeyringInput.builder()
        .generator(kmsKeyArn)
```

```
        .build();
final IKeyring kmsKeyring = matProv.CreateAwsKmsMrkMultiKeyring(keyringInput);
```

C# / .NET

```
var matProv = new MaterialProviders(new MaterialProvidersConfig());
var createAwsKmsMrkMultiKeyringInput = new CreateAwsKmsMrkMultiKeyringInput
{
    Generator = kmsKeyArn
};
var awsKmsMrkMultiKeyring =
 matProv.CreateAwsKmsMrkMultiKeyring(createAwsKmsMrkMultiKeyringInput);
```

Rust

```
let provider_config = MaterialProvidersConfig::builder().build()?;
let mat_prov = client::Client::from_conf(provider_config)?;
let kms_keyring = mat_prov
    .create_aws_kms_mrk_multi_keyring()
    .generator(kms_key_id)
    .send()
    .await?;
```

Gli esempi seguenti utilizzano il `CreateAwsKmsRsaKeyring` metodo per creare un AWS KMS portachiavi con una chiave RSA KMS asimmetrica. Per creare un portachiavi RSA asimmetrico, fornisci i seguenti valori. AWS KMS

- `kmsClient`: crea un nuovo client AWS KMS
- `kmsKeyID`: la chiave ARN che identifica la tua chiave RSA KMS asimmetrica
- `publicKey`: a ByteBuffer di un file PEM con codifica UTF-8 che rappresenta la chiave pubblica della chiave a cui hai passato `kmsKeyID`
- `encryptionAlgorithm`: l'algoritmo di crittografia deve essere o `RSAES_OAEP_SHA_256` `RSAES_OAEP_SHA_1`

Java

```
final MaterialProviders matProv = MaterialProviders.builder()
   .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
```

```
    .build();
final CreateAwsKmsRsaKeyringInput createAwsKmsRsaKeyringInput =
    CreateAwsKmsRsaKeyringInput.builder()
        .kmsClient(KmsClient.create())
        .kmsKeyId(rsaKMSKeyArn)
        .publicKey(publicKey)
        .encryptionAlgorithm(EncryptionAlgorithmSpec.RSAES_OAEP_SHA_256)
        .build();
IKeyring awsKmsRsaKeyring =
 matProv.CreateAwsKmsRsaKeyring(createAwsKmsRsaKeyringInput);
```

C# / .NET

```
var matProv = new MaterialProviders(new MaterialProvidersConfig());
var createAwsKmsRsaKeyringInput = new CreateAwsKmsRsaKeyringInput
{
    KmsClient = new AmazonKeyManagementServiceClient(),
    KmsKeyId = rsaKMSKeyArn,
    PublicKey = publicKey,
    EncryptionAlgorithm = EncryptionAlgorithmSpec.RSAES_OAEP_SHA_256
};
IKeyring awsKmsRsaKeyring =
 matProv.CreateAwsKmsRsaKeyring(createAwsKmsRsaKeyringInput);
```

Rust

```
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;
let sdk_config =
 aws_config::load_defaults(aws_config::BehaviorVersion::latest()).await;
let kms_rsa_keyring = mpl
    .create_aws_kms_rsa_keyring()
    .kms_key_id(rsa_kms_key_arn)
    .public_key(public_key)

 .encryption_algorithm(aws_sdk_kms::types::EncryptionAlgorithmSpec::RsaesOaepSha256)
    .kms_client(aws_sdk_kms::Client::new(&sdk_config))
    .send()
    .await?;
```

# Utilizzo di più regioni AWS KMS keys

È possibile utilizzare più regioni AWS KMS keys come chiavi di wrapping nel AWS Database Encryption SDK. Se si esegue la crittografia con una chiave multiregionale in una Regione AWS, è possibile decrittografare utilizzando una chiave multiregionale correlata in un'altra. Regione AWS

Le chiavi KMS multiregionali sono un insieme di chiavi diverse Regioni AWS che hanno lo stesso AWS KMS keys materiale chiave e lo stesso ID di chiave. Puoi usare queste chiavi correlate come se fossero la stessa chiave in diverse regioni. Le chiavi multiregionali supportano scenari di disaster recovery e backup comuni che richiedono la crittografia in una regione e la decrittografia in un'altra regione senza effettuare una chiamata interregionale a. AWS KMSPer informazioni sulle chiavi multiregionali, consulta Using Multiregion Keys nella Developer Guide.AWS Key Management Service

Per supportare le chiavi multiregionali, il AWS Database Encryption SDK include i portachiavi. AWS KMS multi-Region-aware Il `CreateAwsKmsMrkMultiKeyring` metodo supporta sia chiavi a regione singola che a più regioni.

- Per le chiavi a regione singola, il multi-Region-aware simbolo si comporta esattamente come il portachiavi a regione singola. AWS KMS Tenta di decrittografare il testo cifrato solo con la chiave a regione singola che ha crittografato i dati. Per semplificare la tua esperienza AWS KMS con il portachiavi, ti consigliamo di utilizzare questo `CreateAwsKmsMrkMultiKeyring` metodo ogni volta che utilizzi una chiave KMS con crittografia simmetrica.
- Per le chiavi multiregionali, il multi-Region-aware simbolo tenta di decrittografare il testo cifrato con la stessa chiave multiregionale che ha crittografato i dati o con la relativa chiave multiregionale nella regione specificata.

Nei multi-Region-aware portachiavi che utilizzano più di una chiave KMS, puoi specificare più chiavi singole e multiregionali. Tuttavia, puoi specificare solo una chiave per ogni set di chiavi multiregionali correlate. Se specificate più di un identificatore di chiave con lo stesso ID chiave, la chiamata al costruttore ha esito negativo.

Gli esempi seguenti creano un AWS KMS portachiavi con una chiave KMS multiregionale. Gli esempi specificano una chiave multiregionale come chiave del generatore e una chiave a regione singola come chiave secondaria.

Java

```
final MaterialProviders matProv = MaterialProviders.builder()
        .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
        .build();
final CreateAwsKmsMrkMultiKeyringInput createAwsKmsMrkMultiKeyringInput =
    CreateAwsKmsMrkMultiKeyringInput.builder()
            .generator(multiRegionKeyArn)
            .kmsKeyIds(Collections.singletonList(kmsKeyArn))
            .build();
IKeyring awsKmsMrkMultiKeyring =
 matProv.CreateAwsKmsMrkMultiKeyring(createAwsKmsMrkMultiKeyringInput);
```

C# / .NET

```
var matProv = new MaterialProviders(new MaterialProvidersConfig());
var createAwsKmsMrkMultiKeyringInput = new CreateAwsKmsMrkMultiKeyringInput
{
    Generator = multiRegionKeyArn,
    KmsKeyIds = new List<String> { kmsKeyArn }
};
var awsKmsMrkMultiKeyring =
 matProv.CreateAwsKmsMrkMultiKeyring(createAwsKmsMrkMultiKeyringInput);
```

Rust

```
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;

let aws_kms_mrk_multi_keyring = mpl
    .create_aws_kms_mrk_multi_keyring()
    .generator(multiRegion_key_arn)
    .kms_key_ids(vec![key_arn.to_string()])
    .send()
    .await?;
```

Quando si utilizzano AWS KMS portachiavi multiregione, è possibile decrittografare il testo cifrato in modalità rigorosa o in modalità discovery. Per decrittografare il testo cifrato in modalità rigorosa, istanzia il simbolo multi-Region-aware con la chiave ARN della chiave multiregione correlata nella regione in cui stai decrittografando il testo cifrato. Se si specifica la chiave ARN di una chiave multiregionale correlata in una regione diversa (ad esempio, la regione in cui il record è stato

crittografato), il multi-Region-aware simbolo effettuerà una chiamata interregionale a tale scopo. AWS KMS key

Quando si decrittografa in modalità rigorosa, il multi-Region-aware simbolo richiede una chiave ARN. Accetta solo una chiave ARN da ogni set di chiavi multiregione correlate.

È inoltre possibile decrittografare in modalità di scoperta con chiavi multiregionali. AWS KMS Durante la decrittografia in modalità di rilevamento, non ne viene specificata alcuna. AWS KMS keys(Per informazioni sui portachiavi Single-Region AWS KMS Discovery, consulta.) Utilizzo di un portachiavi Discovery AWS KMS

Se è stata crittografata con una chiave multiregionale, il multi-Region-aware simbolo in modalità di individuazione tenterà di decrittografare utilizzando una chiave multiregionale correlata nella regione locale. Se non ne esiste nessuna, la chiamata ha esito negativo. In modalità di individuazione, AWS Database Encryption SDK non tenterà di effettuare una chiamata interregionale per la chiave multiregionale utilizzata per la crittografia.

## Utilizzo di un portachiavi Discovery AWS KMS

Durante la decrittografia, è consigliabile specificare le chiavi di wrapping che il AWS Database Encryption SDK può utilizzare. Per seguire questa procedura ottimale, utilizza un portachiavi di AWS KMS decrittografia che limiti le chiavi di AWS KMS wrapping a quelle specificate. Tuttavia, puoi anche creare un portachiavi AWS KMS Discovery, ovvero un AWS KMS portachiavi che non specifichi alcuna chiave di avvolgimento.

Il AWS Database Encryption SDK fornisce un portachiavi di AWS KMS rilevamento standard e un portachiavi di rilevamento per chiavi multiregionali. AWS KMS Per informazioni sull'utilizzo di chiavi multiregionali con AWS Database Encryption SDK, consulta. Utilizzo di più regioni AWS KMS keys

Poiché non specifica alcuna chiave di wrapping, un portachiavi Discovery non può crittografare i dati. Se si utilizza un portachiavi Discovery per crittografare i dati, da solo o in un portachiavi multiplo, l'operazione di crittografia non riesce.

Durante la decrittografia, un portachiavi Discovery consente al AWS Database Encryption SDK di chiedere AWS KMS di decrittografare qualsiasi chiave di dati crittografata utilizzando quella chiave crittografata, indipendentemente da chi la possiede o ha accesso a AWS KMS key tale chiave. AWS KMS key La chiamata ha esito positivo solo quando il chiamante dispone dell'autorizzazione per. `kms:Decrypt` AWS KMS key

> **Important**
>
> Se includi un portachiavi AWS KMS Discovery in un portachiavi multiplo di decrittografia, il portachiavi Discovery ha la precedenza su tutte le restrizioni relative alle chiavi KMS specificate dagli altri portachiavi del portachiavi multiplo. Il portachiavi multiplo si comporta come il portachiavi meno restrittivo. Se si utilizza un portachiavi Discovery per crittografare i dati, da solo o in un portachiavi multiplo, l'operazione di crittografia non riesce

Il AWS Database Encryption SDK fornisce un portachiavi di rilevamento per comodità. AWS KMS ma, se possibile, consigliamo di utilizzare un keyring di portata più limitata per i motivi seguenti.

- Autenticità: un portachiavi AWS KMS Discovery può utilizzare qualsiasi AWS KMS key chiave utilizzata per crittografare una chiave di dati nella descrizione del materiale, purché il chiamante sia autorizzato a utilizzarla per la decrittografia. AWS KMS key Questo potrebbe non essere quello AWS KMS key che il chiamante intende utilizzare. Ad esempio, una delle chiavi di dati crittografate potrebbe essere stata crittografata con un metodo meno sicuro AWS KMS key che chiunque può utilizzare.
- Latenza e prestazioni: un portachiavi AWS KMS Discovery potrebbe essere sensibilmente più lento rispetto ad altri portachiavi perché AWS Database Encryption SDK tenta di decrittografare tutte le chiavi di dati crittografate, comprese quelle crittografate AWS KMS keys in altre regioni, Account AWS e AWS KMS keys che il chiamante non è autorizzato a utilizzare per la decrittografia.

Se utilizzi un portachiavi di rilevamento, ti consigliamo di utilizzare un filtro di rilevamento per limitare le chiavi KMS che possono essere utilizzate a quelle presenti in partizioni Account AWS e specifiche. Per informazioni su come trovare l'ID e la partizione dell'account, consulta I tuoi Account AWS identificatori e il formato ARN in. Riferimenti generali di AWS

I seguenti esempi di codice creano un'istanza di un portachiavi di AWS KMS rilevamento con un filtro di rilevamento che limita le chiavi KMS che AWS Database Encryption SDK può utilizzare a quelle presenti nella partizione e nell'account di esempio. `aws` `111122223333`

Prima di utilizzare questo codice, sostituisci i valori di esempio Account AWS e di partizione con valori validi per la tua partizione and. Account AWS Se le tue chiavi KMS si trovano nelle regioni della Cina, usa il valore della `aws-cn` partizione. Se le tue chiavi KMS sono inserite AWS GovCloud (US) Regions, usa il valore della `aws-us-gov` partizione. Per tutti gli altri Regioni AWS, usa il valore della `aws` partizione.

Java

```
// Create discovery filter
DiscoveryFilter discoveryFilter = DiscoveryFilter.builder()
        .partition("aws")
        .accountIds(111122223333)
        .build();
// Create the discovery keyring
CreateAwsKmsMrkDiscoveryMultiKeyringInput createAwsKmsMrkDiscoveryMultiKeyringInput
 = CreateAwsKmsMrkDiscoveryMultiKeyringInput.builder()
        .discoveryFilter(discoveryFilter)
        .build();
IKeyring decryptKeyring =
 matProv.CreateAwsKmsMrkDiscoveryMultiKeyring(createAwsKmsMrkDiscoveryMultiKeyringInput);
```

C# / .NET

```
// Create discovery filter
var discoveryFilter = new DiscoveryFilter
{
    Partition = "aws",
    AccountIds = 111122223333
};
// Create the discovery keyring
var createAwsKmsMrkDiscoveryMultiKeyringInput = new
 CreateAwsKmsMrkDiscoveryMultiKeyringInput
{
    DiscoveryFilter = discoveryFilter
};
var decryptKeyring =
 matProv.CreateAwsKmsMrkDiscoveryMultiKeyring(createAwsKmsMrkDiscoveryMultiKeyringInput);
```

Rust

```
// Create discovery filter
let discovery_filter = DiscoveryFilter::builder()
    .partition("aws")
    .account_ids(111122223333)
    .build()?;

// Create the discovery keyring
let decrypt_keyring = mpl
    .create_aws_kms_mrk_discovery_multi_keyring()
```

```
    .discovery_filter(discovery_filter)
    .send()
    .await?;
```

## Utilizzo di un portachiavi AWS KMS Regional Discovery

Un portachiavi AWS KMS Regional Discovery è un portachiavi che non specifica le ARNs chiavi KMS. Al contrario, consente al AWS Database Encryption SDK di decrittografare utilizzando solo le chiavi KMS in particolare. Regioni AWS

Quando si esegue la decrittografia con un keyring di rilevamento AWS KMS regionale, AWS Database Encryption SDK decrittografa qualsiasi chiave di dati crittografata che è stata crittografata con un valore specificato. AWS KMS key Regione AWS Per avere successo, il chiamante deve disporre dell'`kms:Decrypt`autorizzazione su almeno una delle chiavi di dati specificate che hanno crittografato una AWS KMS keys chiave dati. Regione AWS

Come altri portachiavi Discovery, il portachiavi Discovery regionale non ha alcun effetto sulla crittografia. Funziona solo quando si decifrano campi crittografati. Se si utilizza un portachiavi Regional Discovery in un portachiavi multiplo utilizzato per la crittografia e la decrittografia, è efficace solo durante la decrittografia. Se si utilizza un portachiavi di rilevamento multiregionale per crittografare i dati, da solo o in un portachiavi multiregionale, l'operazione di crittografia non riesce.

> **Important**
>
> Se includi un portachiavi di rilevamento AWS KMS regionale in un portachiavi multiplo di decrittografia, il portachiavi di rilevamento regionale ha la precedenza su tutte le restrizioni relative alle chiavi KMS specificate dagli altri portachiavi del portachiavi multiplo. Il portachiavi multiplo si comporta come il portachiavi meno restrittivo. Un portachiavi AWS KMS Discovery non ha alcun effetto sulla crittografia se utilizzato da solo o in un portachiavi multiplo.

Il portachiavi di rilevamento regionale nel AWS Database Encryption SDK tenta di decrittografare solo con chiavi KMS nella regione specificata. Quando si utilizza un portachiavi di rilevamento, si configura la regione sul client. AWS KMS Queste implementazioni di AWS Database Encryption SDK non filtrano le chiavi KMS per regione, ma AWS KMS falliranno una richiesta di decrittografia per le chiavi KMS al di fuori della regione specificata.

Se utilizzi un portachiavi di rilevamento, ti consigliamo di utilizzare un filtro di rilevamento per limitare le chiavi KMS utilizzate nella decrittografia a quelle presenti nelle partizioni e nelle partizioni specificate. Account AWS

Ad esempio, il codice seguente crea un portachiavi di rilevamento AWS KMS regionale con un filtro di rilevamento. Questo portachiavi limita l'SDK di crittografia del AWS database alle chiavi KMS nell'account 111122223333 nella regione Stati Uniti occidentali (Oregon) (us-west-2).

Java

```
// Create the discovery filter
DiscoveryFilter discoveryFilter = DiscoveryFilter.builder()
        .partition("aws")
        .accountIds(111122223333)
        .build();
// Create the discovery keyring
CreateAwsKmsMrkDiscoveryMultiKeyringInput createAwsKmsMrkDiscoveryMultiKeyringInput
 = CreateAwsKmsMrkDiscoveryMultiKeyringInput.builder()
        .discoveryFilter(discoveryFilter)
        .regions("us-west-2")
        .build();
IKeyring decryptKeyring =
 matProv.CreateAwsKmsMrkDiscoveryMultiKeyring(createAwsKmsMrkDiscoveryMultiKeyringInput);
```

C# / .NET

```
// Create discovery filter
var discoveryFilter = new DiscoveryFilter
{
    Partition = "aws",
    AccountIds = 111122223333
};
// Create the discovery keyring
var createAwsKmsMrkDiscoveryMultiKeyringInput = new
 CreateAwsKmsMrkDiscoveryMultiKeyringInput
{
    DiscoveryFilter = discoveryFilter,
    Regions = us-west-2
};
var decryptKeyring =
 matProv.CreateAwsKmsMrkDiscoveryMultiKeyring(createAwsKmsMrkDiscoveryMultiKeyringInput);
```

Rust

```
// Create discovery filter
let discovery_filter = DiscoveryFilter::builder()
    .partition("aws")
    .account_ids(111122223333)
    .build()?;

// Create the discovery keyring
let decrypt_keyring = mpl
    .create_aws_kms_mrk_discovery_multi_keyring()
    .discovery_filter(discovery_filter)
    .regions(us-west-2)
    .send()
    .await?;
```

# AWS KMS Portachiavi gerarchici

La nostra libreria di crittografia lato client è stata rinominata Database Encryption SDK. AWS Questa guida per sviluppatori fornisce ancora informazioni sul DynamoDB Encryption Client.

> **Note**
>
> A partire dal 24 luglio 2023, le chiavi di filiale create durante l'anteprima per sviluppatori non sono supportate. Crea nuove chiavi branch per continuare a utilizzare l'archivio di chiavi creato durante l'anteprima per sviluppatori.

Con il portachiavi AWS KMS Hierarchical, puoi proteggere i tuoi materiali crittografici con una chiave KMS a crittografia simmetrica senza chiamare AWS KMS ogni volta che crittografi o decrittografi un record. È una buona scelta per le applicazioni che devono ridurre al minimo le chiamate e le applicazioni che possono riutilizzare alcuni materiali crittografici AWS KMS senza violare i requisiti di sicurezza.

Il portachiavi Hierarchical è una soluzione di memorizzazione nella cache dei materiali crittografici che riduce il numero di AWS KMS chiamate utilizzando chiavi branch AWS KMS  protette persistenti in una tabella Amazon DynamoDB e quindi memorizzando nella cache locale i materiali chiave

delle branch utilizzati nelle operazioni di crittografia e decrittografia. La tabella DynamoDB funge da archivio di chiavi che gestisce e protegge le chiavi delle filiali. Memorizza la chiave di ramo attiva e tutte le versioni precedenti della chiave di ramo. La chiave di ramo attiva è la versione più recente della chiave di filiale. Il portachiavi Hierarchical utilizza una chiave di crittografia dei dati unica per ogni richiesta di crittografia e crittografa ogni chiave di crittografia dei dati con una chiave di wrapping unica derivata dalla chiave branch attiva. Il portachiavi Hierarchical dipende dalla gerarchia stabilita tra le chiavi branch attive e le relative chiavi di wrapping derivate.

Il portachiavi Hierarchical utilizza in genere ogni versione della chiave branch per soddisfare più richieste. Tuttavia, puoi controllare la misura in cui le chiavi di ramo attive vengono riutilizzate e determinare la frequenza con cui la chiave di ramo attiva viene ruotata. La versione attiva della chiave di ramo rimane attiva finché non viene ruotata. Le versioni precedenti della chiave di ramo attiva non verranno utilizzate per eseguire operazioni di crittografia, ma potranno comunque essere interrogate e utilizzate nelle operazioni di decrittografia.

Quando si crea un'istanza del portachiavi Hierarchical, viene creata una cache locale. Si specifica un limite di cache che definisce la quantità massima di tempo in cui i materiali chiave del branch vengono archiviati nella cache locale prima che scadano e vengano rimossi dalla cache. Il portachiavi Hierarchical effettua una AWS KMS chiamata per decrittografare la chiave del ramo e assemblare i materiali delle chiavi del ramo la prima volta che a viene specificato in un'operazione. `branch-key-id` I materiali delle chiavi di filiale vengono quindi archiviati nella cache locale e riutilizzati per tutte le operazioni di crittografia e decrittografia che lo specificano fino alla scadenza del limite di cache. `branch-key-id` La memorizzazione dei materiali chiave della filiale nella cache locale riduce le chiamate. AWS KMS Ad esempio, si consideri un limite di cache di 15 minuti. Se si eseguono 10.000 operazioni di crittografia entro tale limite di cache, il AWS KMS portachiavi tradizionale dovrebbe effettuare 10.000 AWS KMS chiamate per soddisfare 10.000 operazioni di crittografia. Se ne hai uno attivo`branch-key-id`, il portachiavi Hierarchical deve effettuare solo una AWS KMS chiamata per soddisfare 10.000 operazioni di crittografia.

La cache locale separa i materiali di crittografia dai materiali di decrittografia. I materiali di crittografia vengono assemblati a partire dalla chiave branch attiva e riutilizzati per tutte le operazioni di crittografia fino alla scadenza del limite della cache. I materiali di decrittografia vengono assemblati a partire dall'ID e dalla versione della chiave di filiale identificati nei metadati del campo crittografato e vengono riutilizzati per tutte le operazioni di decrittografia relative all'ID e alla versione della chiave di filiale fino alla scadenza del limite della cache. La cache locale può memorizzare più versioni della stessa chiave di ramo contemporaneamente. Quando la cache locale è configurata per utilizzare a branch key ID supplier, può anche archiviare i materiali chiave delle branch provenienti da più chiavi di branch attive contemporaneamente.

> **Note**
>
> Tutte le menzioni del portachiavi gerarchico nel AWS Database Encryption SDK si riferiscono al portachiavi gerarchico. AWS KMS

Argomenti



## Come funziona

Le seguenti procedure dettagliate descrivono come il portachiavi Hierarchical assembla i materiali di crittografia e decrittografia e le diverse chiamate che il portachiavi effettua per le operazioni di crittografia e decrittografia. Per i dettagli tecnici sulla derivazione delle chiavi di wrapping e sui processi di crittografia delle chiavi di dati in chiaro, consulta Dettagli tecnici del portachiavi gerarchico.AWS KMS

Crittografa e firma

La procedura dettagliata seguente descrive come il portachiavi Hierarchical assembla i materiali di crittografia e ricava una chiave di avvolgimento unica.

1. Il metodo di crittografia richiede al portachiavi Hierarchical i materiali di crittografia. Il portachiavi genera una chiave di dati in testo semplice, quindi verifica se nella cache locale sono presenti materiali chiave branch validi per generare la chiave di wrapping. Se sono presenti materiali validi per le chiavi di filiale, il portachiavi passa alla Fase 4.
2. Se non ci sono materiali validi per le chiavi di ramo, il portachiavi Hierarchical interroga l'archivio delle chiavi per la chiave di ramo attiva.

    a. Il key store chiama AWS KMS per decrittografare la chiave branch attiva e restituisce la chiave branch attiva in testo semplice. I dati che identificano la chiave di ramo attiva

vengono serializzati per fornire dati autenticati aggiuntivi (AAD) nella chiamata di decrittografia a. AWS KMS

b. L'archivio chiavi restituisce la chiave di ramo in testo semplice e i dati che la identificano, ad esempio la versione della chiave di filiale.

3. Il portachiavi Hierarchical assembla i materiali chiave del ramo (la chiave di ramo in testo semplice e la versione della chiave di ramo) e ne archivia una copia nella cache locale.
4. Il portachiavi Hierarchical ricava una chiave di avvolgimento unica dalla chiave branch in testo semplice e un sale casuale a 16 byte. Utilizza la chiave di wrapping derivata per crittografare una copia della chiave di dati in testo non crittografato.

Il metodo di crittografia utilizza i materiali di crittografia per crittografare e firmare il record. Per ulteriori informazioni su come i record vengono crittografati e firmati nel AWS Database Encryption SDK, consulta Encrypt and sign.

Decrittografa e verifica

La procedura dettagliata seguente descrive come il portachiavi gerarchico assembla i materiali di decrittografia e decrittografa la chiave di dati crittografata.

1. Il metodo di decrittografia identifica la chiave di dati crittografata dal campo di descrizione del materiale del record crittografato e la passa al portachiavi gerarchico.
2. Il portachiavi Hierarchical deserializza i dati che identificano la chiave dati crittografata, inclusa la versione della chiave branch, il sale da 16 byte e altre informazioni che descrivono come è stata crittografata la chiave dati.

   Per ulteriori informazioni, consulta AWS KMS Dettagli tecnici del portachiavi gerarchico.
3. Il portachiavi Hierarchical verifica se nella cache locale sono presenti materiali chiave di filiale validi che corrispondono alla versione della chiave di filiale identificata nel passaggio 2. Se sono presenti materiali validi per le chiavi di filiale, il portachiavi passa alla Fase 6.
4. Se non ci sono materiali validi per le chiavi di ramo, il portachiavi Hierarchical interroga l'archivio delle chiavi per la chiave di filiale che corrisponde alla versione della chiave di filiale identificata nello Step 2.

   a. L'archivio chiavi chiama AWS KMS per decrittografare la chiave di ramo e restituisce la chiave di ramo attiva in testo semplice. I dati che identificano la chiave di ramo attiva vengono serializzati per fornire dati autenticati aggiuntivi (AAD) nella chiamata di decrittografia a. AWS KMS

b. L'archivio chiavi restituisce la chiave di ramo in testo semplice e i dati che la identificano, ad esempio la versione della chiave di filiale.

5. Il portachiavi Hierarchical assembla i materiali chiave del ramo (la chiave di ramo in testo semplice e la versione della chiave di ramo) e ne archivia una copia nella cache locale.
6. Il portachiavi Hierarchical utilizza i materiali delle chiavi branch assemblate e il sale da 16 byte identificato nella fase 2 per riprodurre la chiave di avvolgimento univoca che crittografava la chiave dati.
7. Il portachiavi Hierarchical utilizza la chiave di wrapping riprodotta per decrittografare la chiave dati e restituisce la chiave dati in testo semplice.

Il metodo di decrittografia utilizza i materiali di decrittografia e la chiave di dati in testo semplice per decrittografare e verificare il record. Per ulteriori informazioni su come i record vengono decrittografati e verificati nel Database Encryption SDK, consulta Decriptare e verificare. AWS

## Prerequisiti

Prima di creare e utilizzare un portachiavi gerarchico, assicurati che siano soddisfatti i seguenti prerequisiti.

- Tu o il tuo amministratore dell'archivio chiavi avete creato un archivio chiavi e creato almeno una chiave di ramo attiva.
- Hai configurato le azioni del tuo archivio chiavi.

> **Note**
>
> Il modo in cui configuri le azioni del tuo archivio chiavi determina quali operazioni puoi eseguire e quali chiavi KMS possono essere utilizzate dal portachiavi Hierarchical. Per ulteriori informazioni, consulta Key store actions.

- Disponi delle AWS KMS autorizzazioni necessarie per accedere e utilizzare le chiavi del key store e del branch. Per ulteriori informazioni, consulta the section called “Autorizzazioni richieste”.
- Hai esaminato i tipi di cache supportati e configurato il tipo di cache più adatto alle tue esigenze. Per ulteriori informazioni, consulta the section called “Scegli una cache”

## Autorizzazioni richieste

Il AWS Database Encryption SDK non richiede Account AWS e non dipende da nessuno Servizio AWS. Tuttavia, per utilizzare un portachiavi gerarchico, sono necessarie le seguenti autorizzazioni Account AWS minime per le AWS KMS key crittografie simmetriche presenti nell'archivio delle chiavi.

- Per crittografare e decrittografare i dati con il portachiavi Hierarchical, è necessario KMS:Decrypt.
- Per creare e ruotare le chiavi branch, hai bisogno di kms: e kms:. GenerateDataKeyWithoutPlaintext ReEncrypt

Per ulteriori informazioni sul controllo dell'accesso alle chiavi di filiale e all'archivio delle chiavi, consulta. the section called “Implementazione di autorizzazioni con privilegio minimo”

## Scegli una cache

Il portachiavi gerarchico riduce il numero di chiamate effettuate AWS KMS memorizzando localmente nella cache i materiali chiave della filiale utilizzati nelle operazioni di crittografia e decrittografia. Prima di creare il tuo portachiavi Hierarchical, devi decidere che tipo di cache vuoi usare. È possibile utilizzare la cache predefinita o personalizzarla in base alle proprie esigenze.

Il portachiavi Hierarchical supporta i seguenti tipi di cache:

- the section called “Cache predefinita”
- the section called “MultiThreaded cache”
- the section called “StormTracking cache”
- the section called “Cache condivisa”

### Cache predefinita

Per la maggior parte degli utenti, la cache predefinita soddisfa i requisiti di threading. La cache predefinita è progettata per supportare ambienti con molti multithread. Quando una voce relativa ai materiali delle chiavi di branch scade, la cache predefinita impedisce la chiamata di più thread AWS KMS notificando a un thread che la voce relativa ai materiali della chiave di branch sta per scadere con 10 secondi di anticipo. Ciò garantisce che solo un thread invii una richiesta di aggiornamento della cache AWS KMS .

Il valore predefinito e le StormTracking cache supportano lo stesso modello di threading, ma è sufficiente specificare la capacità di ingresso per utilizzare la cache predefinita. Per personalizzazioni più granulari della cache, usa. the section called “StormTracking cache”

A meno che non si desideri personalizzare il numero di voci relative ai materiali chiave del ramo che possono essere archiviate nella cache locale, non è necessario specificare un tipo di cache quando si crea il portachiavi Hierarchical. Se non si specifica un tipo di cache, il portachiavi Hierarchical utilizza il tipo di cache predefinito e imposta la capacità di immissione su 1000.

Per personalizzare la cache predefinita, specificare i seguenti valori:

- Capacità di ingresso: limita il numero di voci relative ai materiali chiave della filiale che possono essere archiviate nella cache locale.

Java

```
.cache(CacheType.builder()
        .Default(DefaultCache.builder()
        .entryCapacity(100)
        .build())
```

C# / .NET

```
CacheType defaultCache = new CacheType
{
    Default = new DefaultCache{EntryCapacity = 100}
};
```

Rust

```
let cache: CacheType = CacheType::Default(
    DefaultCache::builder()
        .entry_capacity(100)
        .build()?,
);
```

## MultiThreaded cache

La MultiThreaded cache è sicura da usare in ambienti multithread, ma non fornisce alcuna funzionalità per ridurre al minimo AWS KMS le chiamate Amazon DynamoDB. Di conseguenza,

quando scade l'immissione di materiali chiave in una filiale, tutti i thread verranno avvisati contemporaneamente. Ciò può comportare più AWS KMS chiamate per aggiornare la cache.

Per utilizzare la MultiThreaded cache, specificate i seguenti valori:

- Capacità di ingresso: limita il numero di voci relative ai materiali chiave della filiale che possono essere archiviate nella cache locale.
- Entry Poting Tail Size: definisce il numero di elementi da potare se viene raggiunta la capacità di ingresso.

Java

```
.cache(CacheType.builder()
        .MultiThreaded(MultiThreadedCache.builder()
        .entryCapacity(100)
        .entryPruningTailSize(1)
        .build())
```

C# / .NET

```
CacheType multithreadedCache = new CacheType
{
    MultiThreaded = new MultiThreadedCache
    {
        EntryCapacity = 100,
        EntryPruningTailSize = 1
    }
};
```

Rust

```
CacheType::MultiThreaded(
            MultiThreadedCache::builder()
                    .entry_capacity(100)
                    .entry_pruning_tail_size(1)
                    .build()?)
```

## StormTracking cache

La StormTracking cache è progettata per supportare ambienti con molti multithread. Quando una voce relativa ai materiali della chiave di filiale scade, la StormTracking cache impedisce la chiamata di più thread AWS KMS notificando in anticipo a un thread che la voce relativa ai materiali chiave della branch sta per scadere. Ciò garantisce che solo un thread invii una richiesta di aggiornamento della cache AWS KMS .

Per utilizzare la StormTracking cache, specificate i seguenti valori:

- Capacità di ingresso: limita il numero di voci relative ai materiali chiave della filiale che possono essere archiviate nella cache locale.

  Valore predefinito: 1000 voci
- Dimensione della coda di potatura di base: definisce il numero di materiali chiave del ramo da potare alla volta.

  Valore predefinito: 1 voce
- Periodo di tolleranza: definisce il numero di secondi prima della scadenza in cui viene effettuato un tentativo di aggiornare i materiali chiave della filiale.

  Valore predefinito: 10 secondi
- Intervallo di grazia: definisce il numero di secondi tra i tentativi di aggiornamento dei materiali chiave del ramo.

  Valore predefinito: 1 secondi
- Fan out: definisce il numero di tentativi simultanei che è possibile effettuare per aggiornare i materiali chiave della filiale.

  Valore predefinito: 20 tentativi
- In flight time to live (TTL): definisce il numero di secondi che mancano al timeout di un tentativo di aggiornamento dei materiali chiave della filiale. Ogni volta che la cache ritorna `NoSuchEntry` in risposta a una`GetCacheEntry`, quella chiave di ramo viene considerata in esecuzione finché la stessa chiave non viene scritta con una `PutCache` voce.

  Valore predefinito: 10 secondi
- Sospensione: definisce il numero di secondi in cui un thread deve essere sospeso se `fanOut` viene superato il limite.

Valore predefinito: 20 millisecondi

### Java

```
.cache(CacheType.builder()
        .StormTracking(StormTrackingCache.builder()
        .entryCapacity(100)
        .entryPruningTailSize(1)
        .gracePeriod(10)
        .graceInterval(1)
        .fanOut(20)
        .inFlightTTL(10)
        .sleepMilli(20)
        .build())
```

### C# / .NET

```
CacheType stormTrackingCache = new CacheType
{
    StormTracking = new StormTrackingCache
    {
        EntryCapacity = 100,
        EntryPruningTailSize = 1,
        FanOut = 20,
        GraceInterval = 1,
        GracePeriod = 10,
        InFlightTTL = 10,
        SleepMilli = 20
    }
};
```

### Rust

```
CacheType::StormTracking(
                StormTrackingCache::builder()
                    .entry_capacity(100)
                    .entry_pruning_tail_size(1)
                    .grace_period(10)
                    .grace_interval(1)
                    .fan_out(20)
                    .in_flight_ttl(10)
```

```
                .sleep_milli(20)
                .build()?)
```

## Cache condivisa

Per impostazione predefinita, il portachiavi Hierarchical crea una nuova cache locale ogni volta che si crea un'istanza del portachiavi. Tuttavia, la cache condivisa può aiutare a risparmiare memoria consentendoti di condividere una cache tra più portachiavi gerarchici. Anziché creare una nuova cache di materiali crittografici per ogni portachiavi gerarchico istanziato, la cache condivisa archivia solo una cache in memoria, che può essere utilizzata da tutti i portachiavi gerarchici che vi fanno riferimento. La cache condivisa aiuta a ottimizzare l'utilizzo della memoria evitando la duplicazione di materiali crittografici tra portachiavi. I portachiavi gerarchici possono invece accedere alla stessa cache sottostante, riducendo l'ingombro complessivo della memoria.

Quando crei la cache condivisa, definisci comunque il tipo di cache. È possibile specificare un the section called “Cache predefinita”the section called “MultiThreaded cache”, o the section called “StormTracking cache” come tipo di cache o sostituire qualsiasi cache personalizzata compatibile.

### Partizioni

Più portachiavi gerarchici possono utilizzare un'unica cache condivisa. Quando si crea un portachiavi gerarchico con una cache condivisa, è possibile definire un ID di partizione opzionale. L'ID di partizione distingue quale portachiavi gerarchico sta scrivendo nella cache. Se due portachiavi gerarchici fanno riferimento allo stesso ID di partizione e allo stesso ID di chiave di filialelogical key store name, i due portachiavi condivideranno le stesse voci della cache. Se si creano due portachiavi gerarchici con la stessa cache condivisa, ma una partizione diversa IDs, ogni portachiavi accederà alle voci della cache solo dalla propria partizione designata all'interno della cache condivisa. Le partizioni agiscono come divisioni logiche all'interno della cache condivisa, consentendo a ciascun portachiavi gerarchico di funzionare indipendentemente sulla propria partizione designata, senza interferire con i dati memorizzati nell'altra partizione.

Se si intende riutilizzare o condividere le voci della cache in una partizione, è necessario definire il proprio ID di partizione. Quando passate l'ID della partizione al portachiavi Hierarchical, il portachiavi può riutilizzare le voci della cache che sono già presenti nella cache condivisa, anziché dover recuperare e autorizzare nuovamente i materiali delle chiavi della branch. Se non si specifica un ID di partizione, un ID di partizione univoco viene assegnato automaticamente al portachiavi ogni volta che si crea un'istanza del portachiavi Hierarchical.

Le seguenti procedure mostrano come creare una cache condivisa con il tipo di cache predefinito e passarla a un portachiavi gerarchico.

1. Crea un `CryptographicMaterialsCache` (CMC) utilizzando la Material Providers Library (MPL).

   Java

```
// Instantiate the MPL
final MaterialProviders matProv =
    MaterialProviders.builder()
        .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
        .build();

// Create a CacheType object for the Default cache
final CacheType cache =
    CacheType.builder()
        .Default(DefaultCache.builder().entryCapacity(100).build())
        .build();

// Create a CMC using the default cache
final CreateCryptographicMaterialsCacheInput cryptographicMaterialsCacheInput =
    CreateCryptographicMaterialsCacheInput.builder()
        .cache(cache)
        .build();

final ICryptographicMaterialsCache sharedCryptographicMaterialsCache =
    matProv.CreateCryptographicMaterialsCache(cryptographicMaterialsCacheInput);
```

   C# / .NET

```
// Instantiate the MPL
var materialProviders = new MaterialProviders(new MaterialProvidersConfig());

// Create a CacheType object for the Default cache
var cache = new CacheType { Default = new DefaultCache{EntryCapacity = 100} };

// Create a CMC using the default cache
var cryptographicMaterialsCacheInput = new
 CreateCryptographicMaterialsCacheInput {Cache = cache};
```

```
var sharedCryptographicMaterialsCache =
 materialProviders.CreateCryptographicMaterialsCache(cryptographicMaterialsCacheInput);
```

Rust

```
// Instantiate the MPL
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;

// Create a CacheType object for the default cache
let cache: CacheType = CacheType::Default(
    DefaultCache::builder()
        .entry_capacity(100)
        .build()?,
);

// Create a CMC using the default cache
let shared_cryptographic_materials_cache: CryptographicMaterialsCacheRef = mpl.
    create_cryptographic_materials_cache()
    .cache(cache)
    .send()
    .await?;
```

2. Crea un `CacheType` oggetto per la cache condivisa.

   Passa `sharedCryptographicMaterialsCache` il file creato nel passaggio 1 al nuovo `CacheType` oggetto.

   Java

   ```
   // Create a CacheType object for the sharedCryptographicMaterialsCache
   final CacheType sharedCache =
       CacheType.builder()
           .Shared(sharedCryptographicMaterialsCache)
           .build();
   ```

   C# / .NET

   ```
   // Create a CacheType object for the sharedCryptographicMaterialsCache
   var sharedCache = new CacheType { Shared = sharedCryptographicMaterialsCache };
   ```

Rust

```
// Create a CacheType object for the shared_cryptographic_materials_cache
let shared_cache: CacheType =
 CacheType::Shared(shared_cryptographic_materials_cache);
```

3. Passa l'`sharedCache`oggetto dallo Step 2 al tuo portachiavi gerarchico.

Quando crei un portachiavi gerarchico con una cache condivisa, puoi facoltativamente definire un portachiavi gerarchico `partitionID` per condividere le voci della cache su più portachiavi gerarchici. Se non si specifica un ID di partizione, il portachiavi Hierarchical assegna automaticamente al portachiavi un ID di partizione univoco.

> **Note**
>
> I portachiavi gerarchici condivideranno le stesse voci della cache in una cache condivisa se crei due o più portachiavi che fanno riferimento allo stesso ID di partizione e allo stesso ID di chiave di filiale. logical key store name Se non desideri che più portachiavi condividano le stesse voci della cache, devi utilizzare un ID di partizione univoco per ogni portachiavi gerarchico.

L'esempio seguente crea un portachiavi Hierarchical con un branch key ID supplier limite di cache di 600 secondi. Per ulteriori informazioni sui valori definiti nella seguente configurazione del portachiavi gerarchico, vedere. the section called “Crea un portachiavi gerarchico”

Java

```
// Create the Hierarchical keyring
final CreateAwsKmsHierarchicalKeyringInput keyringInput =
    CreateAwsKmsHierarchicalKeyringInput.builder()
        .keyStore(keystore)
        .branchKeyIdSupplier(branchKeyIdSupplier)
        .ttlSeconds(600)
        .cache(sharedCache)
        .partitionID(partitionID)
        .build();
```

```
final IKeyring hierarchicalKeyring =
 matProv.CreateAwsKmsHierarchicalKeyring(keyringInput);
```

C# / .NET

```
// Create the Hierarchical keyring
var createKeyringInput = new CreateAwsKmsHierarchicalKeyringInput
{
   KeyStore = keystore,
   BranchKeyIdSupplier = branchKeyIdSupplier,
   Cache = sharedCache,
   TtlSeconds = 600,
   PartitionId = partitionID
};
var keyring =
 materialProviders.CreateAwsKmsHierarchicalKeyring(createKeyringInput);
```

Rust

```
// Create the Hierarchical keyring
let keyring1 = mpl
    .create_aws_kms_hierarchical_keyring()
    .key_store(key_store1)
    .branch_key_id(branch_key_id.clone())
    // CryptographicMaterialsCacheRef is an Rc (Reference Counted), so if you
 clone it to
    // pass it to different Hierarchical Keyrings, it will still point to the
 same
    // underlying cache, and increment the reference count accordingly.
    .cache(shared_cache.clone())
    .ttl_seconds(600)
    .partition_id(partition_id.clone())
    .send()
    .await?;
```

## Crea un portachiavi gerarchico

Per creare un portachiavi gerarchico, è necessario fornire i seguenti valori:

- Il nome di un archivio di chiavi

Il nome della tabella DynamoDB che tu o il tuo amministratore del key store avete creato per fungere da archivio chiavi.

•

Un limite di durata della cache (TTL)

La quantità di tempo, in secondi, durante la quale una chiave di filiale deve essere inserita nella cache locale può essere utilizzata prima della scadenza. Il limite di cache TTL determina la frequenza con cui il client chiama AWS KMS per autorizzare l'uso delle chiavi della filiale. Questo valore deve essere maggiore di zero. Dopo la scadenza del limite di cache TTL, la voce non viene mai fornita e verrà rimossa dalla cache locale.

- Un identificatore di chiave di filiale

  Puoi configurare staticamente il codice `branch-key-id` che identifica una singola chiave di filiale attiva nel tuo archivio di chiavi o fornire un fornitore di ID per le chiavi di filiale.

  Il fornitore di ID della chiave di filiale utilizza i campi memorizzati nel contesto di crittografia per determinare quale chiave di filiale è necessaria per decrittografare un record. Per impostazione predefinita, nel contesto di crittografia sono incluse solo le chiavi di partizione e ordinamento. Tuttavia, è possibile utilizzare l'[azione `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` crittografica] per includere campi aggiuntivi nel contesto di crittografia.

  Consigliamo vivamente di utilizzare un fornitore di ID di chiavi di filiale per database multitenant in cui ogni tenant ha la propria chiave di filiale. Puoi utilizzare il fornitore di ID delle chiavi di filiale per creare un nome descrittivo per la tua chiave IDs di filiale in modo da facilitare il riconoscimento dell'ID corretto della chiave di filiale per un tenant specifico. Ad esempio, il nome descrittivo consente di fare riferimento a una chiave di filiale come `tenant1` invece di `b3f61619-4d35-48ad-a275-050f87e15122`.

  Per le operazioni di decrittografia, è possibile configurare staticamente un singolo portachiavi gerarchico per limitare la decrittografia a un singolo tenant, oppure è possibile utilizzare il fornitore di ID della chiave di filiale per identificare quale tenant è responsabile della decrittografia di un record.

- (Facoltativo) Una cache

Se desideri personalizzare il tipo di cache o il numero di voci relative ai materiali chiave della filiale che possono essere archiviate nella cache locale, specifica il tipo di cache e la capacità di accesso quando inizializzi il portachiavi.

Il portachiavi Hierarchical supporta i seguenti tipi di cache: predefinita, MultiThreaded e condivisa. StormTracking Per ulteriori informazioni ed esempi che dimostrano come definire ogni tipo di cache, vedere. the section called “Scegli una cache”

Se non si specifica una cache, il portachiavi gerarchico utilizza automaticamente il tipo di cache predefinito e imposta la capacità di ingresso su 1000.

- (Facoltativo) Un ID di partizione

  Se si specifica ilthe section called “Cache condivisa”, è possibile definire facoltativamente un ID di partizione. L'ID di partizione distingue quale portachiavi Hierarchical sta scrivendo nella cache. Se si intende riutilizzare o condividere le voci della cache in una partizione, è necessario definire il proprio ID di partizione. È possibile specificare qualsiasi stringa per l'ID della partizione. Se non si specifica un ID di partizione, al portachiavi viene assegnato automaticamente un ID di partizione univoco al momento della creazione.

  Per ulteriori informazioni, consulta Partitions.

  > **Note**
  >
  > I portachiavi gerarchici condivideranno le stesse voci della cache in una cache condivisa se crei due o più portachiavi che fanno riferimento allo stesso ID di partizione e allo stesso ID di chiave di filiale. logical key store name Se non desideri che più portachiavi condividano le stesse voci della cache, devi utilizzare un ID di partizione univoco per ogni portachiavi gerarchico.

- (Facoltativo) Un elenco di token di concessione

  Se controlli l'accesso alla chiave KMS nel tuo portachiavi gerarchico con le concessioni, devi fornire tutti i token di concessione necessari quando inizializzi il portachiavi.

### Crea un portachiavi gerarchico con un ID di chiave branch statico

Gli esempi seguenti mostrano come creare un portachiavi gerarchico con un ID di chiave branch staticothe section called “Cache predefinita”, the e un limite di cache TTL di 600 secondi.

### Java

```
final MaterialProviders matProv = MaterialProviders.builder()
        .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
        .build();
final CreateAwsKmsHierarchicalKeyringInput keyringInput =
 CreateAwsKmsHierarchicalKeyringInput.builder()
        .keyStore(branchKeyStoreName)
        .branchKeyId(branch-key-id)
        .ttlSeconds(600)
        .build();
final Keyring hierarchicalKeyring =
 matProv.CreateAwsKmsHierarchicalKeyring(keyringInput);
```

### C# / .NET

```
var matProv = new MaterialProviders(new MaterialProvidersConfig());
var keyringInput = new CreateAwsKmsHierarchicalKeyringInput
{
   KeyStore = keystore,
   BranchKeyIdSupplier = branchKeyIdSupplier,
   TtlSeconds = 600
};
var hierarchicalKeyring = matProv.CreateAwsKmsHierarchicalKeyring(keyringInput);
```

### Rust

```
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;

let hierarchical_keyring = mpl
    .create_aws_kms_hierarchical_keyring()
    .branch_key_id(branch_key_id)
    .key_store(branch_key_store_name)
    .ttl_seconds(600)
    .send()
    .await?;
```

Crea un portachiavi gerarchico con una chiave di filiale (ID fornitore).

Le seguenti procedure mostrano come creare un portachiavi gerarchico con un fornitore di ID di chiave di filiale.

1. Crea un fornitore di ID chiave di filiale

   L'esempio seguente crea nomi descrittivi per le due chiavi di ramo create nel passaggio 1 e chiama `CreateDynamoDbEncryptionBranchKeyIdSupplier` a creare un fornitore di ID di chiavi di filiale con il client AWS Database Encryption SDK per DynamoDB.

   Java

```
// Create friendly names for each branch-key-id
class ExampleBranchKeyIdSupplier implements IDynamoDbKeyBranchKeyIdSupplier {
    private static String branchKeyIdForTenant1;
    private static String branchKeyIdForTenant2;

    public ExampleBranchKeyIdSupplier(String tenant1Id, String tenant2Id) {
        this.branchKeyIdForTenant1 = tenant1Id;
        this.branchKeyIdForTenant2 = tenant2Id;
    }
// Create the branch key ID supplier
final DynamoDbEncryption ddbEnc = DynamoDbEncryption.builder()
        .DynamoDbEncryptionConfig(DynamoDbEncryptionConfig.builder().build())
        .build();
final BranchKeyIdSupplier branchKeyIdSupplier =
 ddbEnc.CreateDynamoDbEncryptionBranchKeyIdSupplier(
    CreateDynamoDbEncryptionBranchKeyIdSupplierInput.builder()
            .ddbKeyBranchKeyIdSupplier(new ExampleBranchKeyIdSupplier(branch-
key-ID-tenant1, branch-key-ID-tenant2))
            .build()).branchKeyIdSupplier();
```

   C# / .NET

```
// Create friendly names for each branch-key-id
 class ExampleBranchKeyIdSupplier : DynamoDbKeyBranchKeyIdSupplierBase {
    private String _branchKeyIdForTenant1;
    private String _branchKeyIdForTenant2;

    public ExampleBranchKeyIdSupplier(String tenant1Id, String tenant2Id) {
        this._branchKeyIdForTenant1 = tenant1Id;
```

```
        this._branchKeyIdForTenant2 = tenant2Id;
    }
// Create the branch key ID supplier
var ddbEnc = new DynamoDbEncryption(new DynamoDbEncryptionConfig());
var branchKeyIdSupplier = ddbEnc.CreateDynamoDbEncryptionBranchKeyIdSupplier(
    new CreateDynamoDbEncryptionBranchKeyIdSupplierInput
    {
        DdbKeyBranchKeyIdSupplier = new ExampleBranchKeyIdSupplier(branch-key-
ID-tenant1, branch-key-ID-tenant2)
    }).BranchKeyIdSupplier;
```

Rust

```
// Create friendly names for each branch_key_id
pub struct ExampleBranchKeyIdSupplier {
    branch_key_id_for_tenant1: String,
    branch_key_id_for_tenant2: String,
}

impl ExampleBranchKeyIdSupplier {
    pub fn new(tenant1_id: &str, tenant2_id: &str) -> Self {
        Self {
            branch_key_id_for_tenant1: tenant1_id.to_string(),
            branch_key_id_for_tenant2: tenant2_id.to_string(),
        }
    }
}

// Create the branch key ID supplier
let dbesdk_config = DynamoDbEncryptionConfig::builder().build()?;
let dbesdk = dbesdk_client::Client::from_conf(dbesdk_config)?;
let supplier = ExampleBranchKeyIdSupplier::new(tenant1_branch_key_id,
 tenant2_branch_key_id);

let branch_key_id_supplier = dbesdk
    .create_dynamo_db_encryption_branch_key_id_supplier()
    .ddb_key_branch_key_id_supplier(supplier)
    .send()
    .await?
    .branch_key_id_supplier
    .unwrap();
```

2. Crea un portachiavi gerarchico

I seguenti esempi inizializzano un portachiavi gerarchico con il branch key ID supplier creato nel passaggio 1, un TLL limite di cache di 600 secondi e una dimensione massima della cache di 1000.

Java

```
final MaterialProviders matProv = MaterialProviders.builder()
        .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
        .build();
final CreateAwsKmsHierarchicalKeyringInput keyringInput =
 CreateAwsKmsHierarchicalKeyringInput.builder()
        .keyStore(keystore)
        .branchKeyIdSupplier(branchKeyIdSupplier)
        .ttlSeconds(600)
        .cache(CacheType.builder() //OPTIONAL
                .Default(DefaultCache.builder()
                .entryCapacity(100)
                .build())
        .build();
final Keyring hierarchicalKeyring =
 matProv.CreateAwsKmsHierarchicalKeyring(keyringInput);
```

C# / .NET

```
var matProv = new MaterialProviders(new MaterialProvidersConfig());
var keyringInput = new CreateAwsKmsHierarchicalKeyringInput
{
   KeyStore = keystore,
   BranchKeyIdSupplier = branchKeyIdSupplier,
   TtlSeconds = 600,
   Cache = new CacheType
   {
        Default = new DefaultCache { EntryCapacity = 100 }
   }
};
var hierarchicalKeyring = matProv.CreateAwsKmsHierarchicalKeyring(keyringInput);
```

Rust

```
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;
```

```
let hierarchical_keyring = mpl
    .create_aws_kms_hierarchical_keyring()
    .branch_key_id_supplier(branch_key_id_supplier)
    .key_store(key_store)
    .ttl_seconds(600)
    .send()
    .await?;
```

## Utilizzo del portachiavi gerarchico per una crittografia ricercabile

La crittografia ricercabile consente di cercare record crittografati senza decrittografare l'intero database. Ciò si ottiene indicizzando il valore in chiaro di un campo crittografato con un beacon. Per implementare la crittografia ricercabile, è necessario utilizzare un portachiavi gerarchico.

L'*`CreateKey`*operazione di archiviazione delle chiavi genera sia una chiave branch che una chiave beacon. La chiave branch viene utilizzata nelle operazioni di crittografia e decrittografia dei record. La chiave beacon viene utilizzata per generare beacon.

La chiave branch e la chiave beacon sono protette dallo stesso AWS KMS key che hai specificato durante la creazione del servizio di archiviazione delle chiavi. Dopo che l'`CreateKey`operazione chiama AWS KMS per generare la chiave branch, chiama kms: GenerateDataKeyWithoutPlaintext una seconda volta per generare la chiave beacon utilizzando la seguente richiesta.

```
{
   "EncryptionContext": {
      "branch-key-id" : "branch-key-id",
      "type" : type,
      "create-time" : "timestamp",
      "logical-key-store-name" : "the logical table name for your key store",
      "kms-arn" : the KMS key ARN,
      "hierarchy-version" : 1
   },
   "KeyId": "the KMS key ARN",
   "NumberOfBytes": "32"
}
```

Dopo aver generato entrambe le chiavi, l'`CreateKey`operazione chiama ddb: TransactWriteItems per scrivere due nuovi elementi che mantengano la chiave branch e la chiave beacon nell'archivio delle chiavi della filiale.

Quando configuri un beacon standard, AWS Database Encryption SDK interroga l'archivio delle chiavi per la chiave beacon. Quindi, utilizza una funzione di derivazione delle extract-and-expand chiavi basata su HMAC (HKDF) per combinare la chiave beacon con il nome del beacon standard per creare la chiave HMAC per un determinato beacon.

A differenza delle chiavi branch, esiste una sola versione di chiave beacon per ogni chiave. `branch-key-id` La chiave beacon non viene mai ruotata.

### Definizione della fonte della chiave del beacon

Quando si definisce la versione beacon per i beacon standard e compositi, è necessario identificare la chiave beacon e definire un TTL (cache limit time to live) per i materiali chiave del beacon. I materiali delle chiavi beacon vengono archiviati in una cache locale separata dalle chiavi branch. Il seguente frammento mostra come definire un database single-tenant. `keySource` Identifica la tua chiave beacon in base alla quale è associata. `branch-key-id`

Java

```
keySource(BeaconKeySource.builder()
        .single(SingleKeyStore.builder()
                .keyId(branch-key-id)
                .cacheTTL(6000)
                .build())
        .build())
```

C# / .NET

```
KeySource = new BeaconKeySource
{
    Single = new SingleKeyStore
    {
       KeyId = branch-key-id,
       CacheTTL = 6000
    }
}
```

Rust

```
 .key_source(BeaconKeySource::Single(
    SingleKeyStore::builder()
        // `keyId` references a beacon key.
```

```
        // For every branch key we create in the keystore,
        // we also create a beacon key.
        // This beacon key is not the same as the branch key,
        // but is created with the same ID as the branch key.
        .key_id(branch_key_id)
        .cache_ttl(6000)
        .build()?,
))
```

Definizione della fonte del beacon in un database multitenant

Se si dispone di un database multitenant, è necessario specificare i seguenti valori durante la configurazione di. `keySource`

- keyFieldName

  Definisce il nome del campo che memorizza la chiave beacon `branch-key-id` associata alla chiave beacon utilizzata per generare i beacon per un determinato tenant. `keyFieldName`Può essere una stringa qualsiasi, ma deve essere univoca per tutti gli altri campi del database. Quando si scrivono nuovi record nel database, la chiave `branch-key-id` che identifica la chiave beacon utilizzata per generare i beacon per quel record viene memorizzata in questo campo. È necessario includere questo campo nelle query del beacon e identificare i materiali chiave del beacon appropriati necessari per ricalcolare il beacon. Per ulteriori informazioni, consulta Interrogazione dei beacon in un database multi-tenant.
- CacheTTL

  La quantità di tempo, in secondi, necessaria per l'immissione di una chiave beacon nella cache locale del beacon prima della scadenza. Questo valore deve essere maggiore di zero. Quando il limite di cache TTL scade, la voce viene rimossa dalla cache locale.
- (Facoltativo) Una cache

  Se desideri personalizzare il tipo di cache o il numero di voci relative ai materiali chiave della filiale che possono essere archiviate nella cache locale, specifica il tipo di cache e la capacità di accesso quando inizializzi il portachiavi.

  Il portachiavi Hierarchical supporta i seguenti tipi di cache: predefinita, MultiThreaded e condivisa. StormTracking Per ulteriori informazioni ed esempi che dimostrano come definire ogni tipo di cache, vedere. the section called “Scegli una cache”

Se non si specifica una cache, il portachiavi gerarchico utilizza automaticamente il tipo di cache predefinito e imposta la capacità di ingresso su 1000.

L'esempio seguente crea un portachiavi gerarchico con una chiave di filiale (ID provider), un limite di cache (TLL) di 600 secondi e una capacità di immissione di 1000.

Java

```
final MaterialProviders matProv = MaterialProviders.builder()
        .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
        .build();
final CreateAwsKmsHierarchicalKeyringInput keyringInput =
 CreateAwsKmsHierarchicalKeyringInput.builder()
        .keyStore(branchKeyStoreName)
        .branchKeyIdSupplier(branchKeyIdSupplier)
        .ttlSeconds(600)
        .cache(CacheType.builder() //OPTIONAL
                .Default(DefaultCache.builder()
                        .entryCapacity(1000)
                        .build())
                .build();
final IKeyring hierarchicalKeyring =
 matProv.CreateAwsKmsHierarchicalKeyring(keyringInput);
```

C# / .NET

```
var matProv = new MaterialProviders(new MaterialProvidersConfig());
var keyringInput = new CreateAwsKmsHierarchicalKeyringInput
{
   KeyStore = keystore,
   BranchKeyIdSupplier = branchKeyIdSupplier,
   TtlSeconds = 600,
   Cache = new CacheType
   {
        Default = new DefaultCache { EntryCapacity = 1000 }
   }
};
var hierarchicalKeyring = matProv.CreateAwsKmsHierarchicalKeyring(keyringInput);
```

Rust

```
let provider_config = MaterialProvidersConfig::builder().build()?;
```

```
    let mat_prov = client::Client::from_conf(provider_config)?;
    let kms_keyring = mat_prov
        .create_aws_kms_hierarchical_keyring()
        .branch_key_id(branch_key_id)
        .key_store(key_store)
        .ttl_seconds(600)
        .send()
        .await?;
```

# AWS KMS Portachiavi ECDH

La nostra libreria di crittografia lato client è stata rinominata Database Encryption SDK. AWS Questa guida per sviluppatori fornisce ancora informazioni sul DynamoDB Encryption Client.

**Important**

Il portachiavi AWS KMS ECDH è disponibile solo con la versione 1.5.0 o successiva della Material Providers Library.

Un portachiavi AWS KMS ECDH utilizza un accordo di chiave asimmetrico per ricavare una chiave di avvolgimento simmetrica AWS KMS keyscondivisa tra due parti. Innanzitutto, il portachiavi utilizza l'algoritmo di accordo delle chiavi Elliptic Curve Diffie-Hellman (ECDH) per ricavare un segreto condiviso dalla chiave privata nella coppia di chiavi KMS del mittente e dalla chiave pubblica del destinatario. Quindi, il portachiavi utilizza il segreto condiviso per derivare la chiave di wrapping condivisa che protegge le chiavi di crittografia dei dati. La funzione di derivazione delle chiavi utilizzata da AWS Database Encryption SDK (`KDF_CTR_HMAC_SHA384`) per derivare la chiave di wrapping condivisa è conforme alle raccomandazioni del NIST per la derivazione delle chiavi.

La funzione di derivazione delle chiavi restituisce 64 byte di materiale di codifica. Per garantire che entrambe le parti utilizzino il materiale di codifica corretto, AWS Database Encryption SDK utilizza i primi 32 byte come chiave di commit e gli ultimi 32 byte come chiave di wrapping condivisa. In caso di decrittografia, se il portachiavi non è in grado di riprodurre la stessa chiave di impegno e la stessa chiave di wrapping condivisa memorizzate nel campo di descrizione del materiale del record crittografato, l'operazione ha esito negativo. Ad esempio, se si crittografa un record con un portachiavi configurato con la chiave privata di Alice e la chiave pubblica di Bob, un portachiavi

configurato con la chiave privata di Bob e la chiave pubblica di Alice riprodurrà la stessa chiave di impegno e la stessa chiave di wrapping condivisa e sarà in grado di decrittografare il record. Se la chiave pubblica di Bob non proviene da una coppia di chiavi KMS, Bob può creare un portachiavi ECDH Raw per decrittografare il record.

Il portachiavi AWS KMS ECDH crittografa i record con una chiave simmetrica utilizzando AES-GCM. La chiave dati viene quindi crittografata in busta con la chiave di wrapping condivisa derivata utilizzando AES-GCM. Ogni portachiavi AWS KMS ECDH può avere solo una chiave di avvolgimento condivisa, ma è possibile includere più portachiavi AWS KMS ECDH, da soli o con altri portachiavi, in un portachiavi multiplo.

Argomenti

- AWS KMS Autorizzazioni richieste per i portachiavi ECDH
- AWS KMS Creazione di un portachiavi ECDH
- Creazione di un portachiavi ECDH Discovery AWS KMS

## AWS KMS Autorizzazioni richieste per i portachiavi ECDH

Il AWS Database Encryption SDK non richiede un AWS account e non dipende da alcun servizio. AWS Tuttavia, per utilizzare un portachiavi AWS KMS ECDH, è necessario disporre di un AWS account e delle seguenti autorizzazioni minime presenti nel portachiavi. AWS KMS keys Le autorizzazioni variano in base allo schema di accordi chiave utilizzato.

- Per crittografare e decrittografare i record utilizzando lo schema di accordo `KmsPrivateKeyToStaticPublicKey` chiave, sono necessari kms: GetPublicKey e kms: DeriveSharedSecret sulla coppia di chiavi KMS asimmetrica del mittente. Se fornisci direttamente la chiave pubblica con codifica DER del mittente quando crei un'istanza del tuo portachiavi, hai solo bisogno dell'DeriveSharedSecretautorizzazione kms: sulla coppia di chiavi KMS asimmetrica del mittente.
- Per decrittografare i record utilizzando lo schema di accordo `KmsPublicKeyDiscovery` chiave, sono necessarie le GetPublicKey autorizzazioni kms: DeriveSharedSecret e kms: sulla coppia di chiavi KMS asimmetrica specificata.

# AWS KMS Creazione di un portachiavi ECDH

Per creare un portachiavi AWS KMS ECDH che crittografa e decrittografa i dati, è necessario utilizzare lo schema degli accordi chiave. `KmsPrivateKeyToStaticPublicKey` Per inizializzare un portachiavi AWS KMS ECDH con lo schema degli accordi chiave, fornisci i seguenti valori: `KmsPrivateKeyToStaticPublicKey`

- ID del mittente AWS KMS key

  Deve identificare una coppia di chiavi KMS a curva ellittica (ECC) asimmetrica consigliata dal NIST con un valore di. `KeyUsage` `KEY_AGREEMENT` La chiave privata del mittente viene utilizzata per derivare il segreto condiviso.

- (Facoltativo) Chiave pubblica del mittente

  Deve essere una chiave pubblica X.509 con codifica DER, nota anche come `SubjectPublicKeyInfo` (SPKI), come definita in RFC 5280.

  L' AWS KMS GetPublicKeyoperazione restituisce la chiave pubblica di una coppia di chiavi KMS asimmetrica nel formato codificato DER richiesto.

  Per ridurre il numero di AWS KMS chiamate effettuate dal portachiavi, puoi fornire direttamente la chiave pubblica del mittente. Se non viene fornito alcun valore per la chiave pubblica del mittente, il portachiavi chiama AWS KMS per recuperare la chiave pubblica del mittente.

- Chiave pubblica del destinatario

  È necessario fornire la chiave pubblica X.509 con codifica DER del destinatario, nota anche come `SubjectPublicKeyInfo` (SPKI), come definita in RFC 5280.

  L' AWS KMS GetPublicKeyoperazione restituisce la chiave pubblica di una coppia di chiavi KMS asimmetrica nel formato codificato DER richiesto.

- Specificazione della curva

  Identifica la specifica della curva ellittica nelle coppie di chiavi specificate. Entrambe le coppie di chiavi del mittente e del destinatario devono avere la stessa specifica di curva.

  Valori validi: `ECC_NIST_P256`, `ECC_NIS_P384`, `ECC_NIST_P512`

- (Facoltativo) Un elenco di token di concessione

Se controlli l'accesso alla chiave KMS nel tuo portachiavi AWS KMS ECDH con le sovvenzioni, devi fornire tutti i token di concessione necessari quando inizializzi il portachiavi.

C# / .NET

L'esempio seguente crea un portachiavi AWS KMS ECDH con la chiave KMS del mittente, la chiave pubblica del mittente e la chiave pubblica del destinatario. Questo esempio utilizza il `senderPublicKey` parametro opzionale per fornire la chiave pubblica del mittente. Se non fornisci la chiave pubblica del mittente, il portachiavi chiama AWS KMS per recuperare la chiave pubblica del mittente. Entrambe le coppie di chiavi del mittente e del destinatario sono pronte. `ECC_NIST_P256`

```
// Instantiate material providers
var materialProviders = new MaterialProviders(new MaterialProvidersConfig());

// Must be DER-encoded X.509 public keys
var BobPublicKey = new MemoryStream(new byte[] { });
var AlicePublicKey = new MemoryStream(new byte[] { });

// Create the AWS KMS ECDH static keyring
var staticConfiguration = new KmsEcdhStaticConfigurations
{
    KmsPrivateKeyToStaticPublicKey = new KmsPrivateKeyToStaticPublicKeyInput
    {
        SenderKmsIdentifier = "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab",
        SenderPublicKey = BobPublicKey,
        RecipientPublicKey = AlicePublicKey
    }
};

var createKeyringInput = new CreateAwsKmsEcdhKeyringInput
{
    CurveSpec = ECDHCurveSpec.ECC_NIST_P256,
    KmsClient = new AmazonKeyManagementServiceClient(),
    KeyAgreementScheme = staticConfiguration
};

var keyring = materialProviders.CreateAwsKmsEcdhKeyring(createKeyringInput);
```

Java

L'esempio seguente crea un portachiavi AWS KMS ECDH con la chiave KMS del mittente, la chiave pubblica del mittente e la chiave pubblica del destinatario. Questo esempio utilizza il `senderPublicKey` parametro opzionale per fornire la chiave pubblica del mittente. Se non fornisci la chiave pubblica del mittente, il portachiavi chiama AWS KMS per recuperare la chiave pubblica del mittente. Entrambe le coppie di chiavi del mittente e del destinatario sono pronte. `ECC_NIST_P256`

```
// Retrieve public keys
// Must be DER-encoded X.509 public keys
ByteBuffer BobPublicKey = getPublicKeyBytes("arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab");
        ByteBuffer AlicePublicKey = getPublicKeyBytes("arn:aws:kms:us-
west-2:111122223333:key/0987dcba-09fe-87dc-65ba-ab0987654321");

// Create the AWS KMS ECDH static keyring
        final CreateAwsKmsEcdhKeyringInput senderKeyringInput =
          CreateAwsKmsEcdhKeyringInput.builder()
            .kmsClient(KmsClient.create())
            .curveSpec(ECDHCurveSpec.ECC_NIST_P256)
            .KeyAgreementScheme(
              KmsEcdhStaticConfigurations.builder()
                .KmsPrivateKeyToStaticPublicKey(
                  KmsPrivateKeyToStaticPublicKeyInput.builder()
                    .senderKmsIdentifier("arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab")
                    .senderPublicKey(BobPublicKey)
                    .recipientPublicKey(AlicePublicKey)
                    .build()).build()).build();
```

Rust

L'esempio seguente crea un portachiavi AWS KMS ECDH con la chiave KMS del mittente, la chiave pubblica del mittente e la chiave pubblica del destinatario. Questo esempio utilizza il `sender_public_key` parametro opzionale per fornire la chiave pubblica del mittente. Se non fornisci la chiave pubblica del mittente, il portachiavi chiama AWS KMS per recuperare la chiave pubblica del mittente.

```
// Retrieve public keys
// Must be DER-encoded X.509 keys
```

```
let public_key_file_content_sender =
 std::fs::read_to_string(Path::new(EXAMPLE_KMS_ECC_PUBLIC_KEY_FILENAME_SENDER))?;
let parsed_public_key_file_content_sender = parse(public_key_file_content_sender)?;
let public_key_sender_utf8_bytes = parsed_public_key_file_content_sender.contents();

let public_key_file_content_recipient =
 std::fs::read_to_string(Path::new(EXAMPLE_KMS_ECC_PUBLIC_KEY_FILENAME_RECIPIENT))?;
let parsed_public_key_file_content_recipient =
 parse(public_key_file_content_recipient)?;
let public_key_recipient_utf8_bytes =
 parsed_public_key_file_content_recipient.contents();

// Create KmsPrivateKeyToStaticPublicKeyInput
let kms_ecdh_static_configuration_input =
    KmsPrivateKeyToStaticPublicKeyInput::builder()
        .sender_kms_identifier(arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab)
        // Must be a UTF8 DER-encoded X.509 public key
        .sender_public_key(public_key_sender_utf8_bytes)
        // Must be a UTF8 DER-encoded X.509 public key
        .recipient_public_key(public_key_recipient_utf8_bytes)
        .build()?;

let kms_ecdh_static_configuration =
 KmsEcdhStaticConfigurations::KmsPrivateKeyToStaticPublicKey(kms_ecdh_static_configuration_i

// Instantiate the material providers library
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;

// Create AWS KMS ECDH keyring
let kms_ecdh_keyring = mpl
    .create_aws_kms_ecdh_keyring()
    .kms_client(kms_client)
    .curve_spec(ecdh_curve_spec)
    .key_agreement_scheme(kms_ecdh_static_configuration)
    .send()
    .await?;
```

# Creazione di un portachiavi ECDH Discovery AWS KMS

Durante la decrittografia, è consigliabile specificare le chiavi che il AWS Database Encryption SDK può utilizzare. Per seguire questa best practice, utilizzate un portachiavi AWS KMS ECDH con lo schema degli accordi chiave. `KmsPrivateKeyToStaticPublicKey` Tuttavia, puoi anche creare un portachiavi AWS KMS ECDH discovery, ovvero un portachiavi AWS KMS ECDH in grado di decrittografare qualsiasi record in cui la chiave pubblica della coppia di chiavi KMS specificata corrisponda alla chiave pubblica del destinatario memorizzata nel campo di descrizione del materiale del record crittografato.

> **Important**
>
> Quando si decifrano i record utilizzando lo schema degli accordi `KmsPublicKeyDiscovery` chiave, si accettano tutte le chiavi pubbliche, indipendentemente dal proprietario.

Per inizializzare un portachiavi AWS KMS ECDH con lo schema degli accordi `KmsPublicKeyDiscovery` chiave, fornite i seguenti valori:

- ID del destinatario AWS KMS key

  Deve identificare una coppia di chiavi KMS a curva ellittica (ECC) asimmetrica consigliata dal NIST con un valore di. `KeyUsage` `KEY_AGREEMENT`
- Specificazione della curva

  Identifica la specifica della curva ellittica nella coppia di chiavi KMS del destinatario.

  Valori validi: `ECC_NIST_P256`, `ECC_NIS_P384`, `ECC_NIST_P512`
- (Facoltativo) Un elenco di token di concessione

  Se controlli l'accesso alla chiave KMS nel tuo portachiavi AWS KMS ECDH con le sovvenzioni, devi fornire tutti i token di concessione necessari quando inizializzi il portachiavi.

C# / .NET

L'esempio seguente crea un portachiavi AWS KMS ECDH discovery con una coppia di chiavi KMS sulla curva. `ECC_NIST_P256` È necessario disporre delle DeriveSharedSecret autorizzazioni kms: GetPublicKey e kms: sulla coppia di key pair KMS specificata. Questo portachiavi può decrittografare qualsiasi record in cui la chiave pubblica della coppia di chiavi

KMS specificata corrisponde alla chiave pubblica del destinatario memorizzata nel campo di descrizione del materiale del record crittografato.

```
// Instantiate material providers
var materialProviders = new MaterialProviders(new MaterialProvidersConfig());

// Create the AWS KMS ECDH discovery keyring
var discoveryConfiguration = new KmsEcdhStaticConfigurations
{
    KmsPublicKeyDiscovery = new KmsPublicKeyDiscoveryInput
    {
        RecipientKmsIdentifier = "arn:aws:kms:us-
west-2:111122223333:key/0987dcba-09fe-87dc-65ba-ab0987654321"
    }

};
var createKeyringInput = new CreateAwsKmsEcdhKeyringInput
{
    CurveSpec = ECDHCurveSpec.ECC_NIST_P256,
    KmsClient = new AmazonKeyManagementServiceClient(),
    KeyAgreementScheme = discoveryConfiguration
};
var keyring = materialProviders.CreateAwsKmsEcdhKeyring(createKeyringInput);
```

Java

L'esempio seguente crea un portachiavi AWS KMS ECDH discovery con una coppia di chiavi KMS sulla curva. `ECC_NIST_P256` È necessario disporre delle DeriveSharedSecret autorizzazioni kms: GetPublicKey e kms: sulla coppia di key pair KMS specificata. Questo portachiavi può decrittografare qualsiasi record in cui la chiave pubblica della coppia di chiavi KMS specificata corrisponde alla chiave pubblica del destinatario memorizzata nel campo di descrizione del materiale del record crittografato.

```
// Create the AWS KMS ECDH discovery keyring
final CreateAwsKmsEcdhKeyringInput recipientKeyringInput =
  CreateAwsKmsEcdhKeyringInput.builder()
    .kmsClient(KmsClient.create())
    .curveSpec(ECDHCurveSpec.ECC_NIST_P256)
    .KeyAgreementScheme(
      KmsEcdhStaticConfigurations.builder()
        .KmsPublicKeyDiscovery(
          KmsPublicKeyDiscoveryInput.builder()
```

```
            .recipientKmsIdentifier("arn:aws:kms:us-
west-2:111122223333:key/0987dcba-09fe-87dc-65ba-ab0987654321").build()
        ).build())
    .build();
```

Rust

```
// Create KmsPublicKeyDiscoveryInput
let kms_ecdh_discovery_static_configuration_input =
    KmsPublicKeyDiscoveryInput::builder()
        .recipient_kms_identifier(ecc_recipient_key_arn)
        .build()?;

let kms_ecdh_discovery_static_configuration =
 KmsEcdhStaticConfigurations::KmsPublicKeyDiscovery(kms_ecdh_discovery_static_configuration_

// Instantiate the material providers library
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;

// Create AWS KMS ECDH discovery keyring
let kms_ecdh_discovery_keyring = mpl
    .create_aws_kms_ecdh_keyring()
    .kms_client(kms_client.clone())
    .curve_spec(ecdh_curve_spec)
    .key_agreement_scheme(kms_ecdh_discovery_static_configuration)
    .send()
    .await?;
```

# Keyring non elaborati AES

La nostra libreria di crittografia lato client è stata rinominata AWS Database Encryption SDK. Questa guida per sviluppatori fornisce ancora informazioni sul DynamoDB Encryption Client.

Il AWS Database Encryption SDK consente di utilizzare una chiave simmetrica AES fornita come chiave di wrapping per proteggere la chiave dati. È necessario generare, archiviare e proteggere il materiale chiave, preferibilmente in un modulo di sicurezza hardware (HSM) o in un sistema di gestione delle chiavi. Usa un portachiavi Raw AES quando devi fornire la chiave di wrapping e crittografare le chiavi dati localmente o offline.

Il portachiavi Raw AES crittografa i dati utilizzando l'algoritmo AES-GCM e una chiave di wrapping specificata come array di byte. È possibile specificare solo una chiave di avvolgimento in ogni portachiavi Raw AES, ma è possibile includere più portachiavi Raw AES, da soli o con altri portachiavi, in un portachiavi multiplo.

Namespace e nomi chiave

Per identificare la chiave AES in un portachiavi, il portachiavi Raw AES utilizza uno spazio dei nomi e un nome chiave forniti dall'utente. Questi valori non sono segreti. Appaiono in testo semplice nella descrizione del materiale che AWS Database Encryption SDK aggiunge al record. Si consiglia di utilizzare uno spazio dei nomi delle chiavi (HSM o sistema di gestione delle chiavi) e un nome di chiave che identifichi la chiave AES in quel sistema.

> **Note**
>
> Lo spazio dei nomi e il nome della chiave sono equivalenti ai campi Provider ID (o Provider) e Key ID di. `JceMasterKey`

Se si creano portachiavi diversi per crittografare e decrittografare un determinato campo, lo spazio dei nomi e i valori dei nomi sono fondamentali. Se lo spazio dei nomi della chiave e il nome della chiave nel portachiavi di decrittografia non corrispondono esattamente, con distinzione tra maiuscole e minuscole, per lo spazio dei nomi della chiave e il nome della chiave nel portachiavi di crittografia, il portachiavi di decrittografia non viene utilizzato, anche se i byte del materiale della chiave sono identici.

Ad esempio, è possibile definire un portachiavi Raw AES con lo spazio dei nomi e il nome della chiave. `HSM_01` `AES_256_012` Quindi, usi quel portachiavi per crittografare alcuni dati. Per decrittografare quei dati, crea un portachiavi Raw AES con lo stesso spazio dei nomi delle chiavi, nome chiave e materiale chiave.

I seguenti esempi mostrano come creare un portachiavi Raw AES. La `AESWrappingKey` variabile rappresenta il materiale chiave fornito.

Java

```
final CreateRawAesKeyringInput keyringInput = CreateRawAesKeyringInput.builder()
        .keyName("AES_256_012")
        .keyNamespace("HSM_01")
        .wrappingKey(AESWrappingKey)
```

```
        .wrappingAlg(AesWrappingAlg.ALG_AES256_GCM_IV12_TAG16)
        .build();
final MaterialProviders matProv = MaterialProviders.builder()
        .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
        .build();
IKeyring rawAesKeyring = matProv.CreateRawAesKeyring(keyringInput);
```

### C# / .NET

```
var keyNamespace = "HSM_01";
var keyName = "AES_256_012";

// This example uses the key generator in Bouncy Castle to generate the key
 material.
// In production, use key material from a secure source.
var aesWrappingKey = new
 MemoryStream(GeneratorUtilities.GetKeyGenerator("AES256").GenerateKey());

// Create the keyring
var keyringInput = new CreateRawAesKeyringInput
{
    KeyNamespace = keyNamespace,
    KeyName = keyName,
    WrappingKey = AESWrappingKey,
    WrappingAlg = AesWrappingAlg.ALG_AES256_GCM_IV12_TAG16
};

var matProv = new MaterialProviders(new MaterialProvidersConfig());
IKeyring rawAesKeyring = matProv.CreateRawAesKeyring(keyringInput);
```

### Rust

```
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;
let raw_aes_keyring = mpl
    .create_raw_aes_keyring()
    .key_name("AES_256_012")
    .key_namespace("HSM_01")
    .wrapping_key(aes_key_bytes)
    .wrapping_alg(AesWrappingAlg::AlgAes256GcmIv12Tag16)
    .send()
    .await?;
```

# Keyring non elaborato RSA

La nostra libreria di crittografia lato client è stata rinominata AWS Database Encryption SDK. Questa guida per sviluppatori fornisce ancora informazioni sul DynamoDB Encryption Client.

Il portachiavi Raw RSA esegue la crittografia e la decrittografia asimmetriche delle chiavi di dati nella memoria locale con una chiave pubblica e privata RSA fornita dall'utente. È necessario generare, archiviare e proteggere la chiave privata, preferibilmente in un modulo di sicurezza hardware (HSM) o in un sistema di gestione delle chiavi. La funzione di crittografia consente di crittografare la chiave di dati nella chiave pubblica RSA. La funzione di decrittazione consente di decrittare la chiave di dati con la chiave privata. Puoi scegliere tra i diverse modalità di padding RSA.

Un keyring non elaborato RSA che esegue crittografia e decrittazione deve includere una coppia di chiavi pubblica e privata asimmetriche. Tuttavia, è possibile crittografare i dati con un portachiavi Raw RSA che ha solo una chiave pubblica e decrittografare i dati con un portachiavi Raw RSA che ha solo una chiave privata. Puoi includere qualsiasi portachiavi Raw RSA in un portachiavi multiplo. Se configuri un portachiavi Raw RSA con una chiave pubblica e una privata, assicurati che facciano parte della stessa coppia di chiavi.

Il portachiavi Raw RSA è equivalente e interagisce con il JceMasterKeyin SDK di crittografia AWS per Java quando viene utilizzato con chiavi di crittografia asimmetrica RSA.

> **Note**
>
> Il portachiavi Raw RSA non supporta le chiavi KMS asimmetriche. Per utilizzare chiavi RSA KMS asimmetriche, crea un portachiavi.AWS KMS

### Namespace e nomi

Per identificare il materiale chiave RSA in un portachiavi, il portachiavi Raw RSA utilizza uno spazio dei nomi e un nome chiave forniti dall'utente. Questi valori non sono segreti. Appaiono in testo semplice nella descrizione del materiale che AWS Database Encryption SDK aggiunge al record. Ti consigliamo di utilizzare lo spazio dei nomi e il nome della chiave che identificano la coppia di chiavi RSA (o la relativa chiave privata) nel tuo HSM o sistema di gestione delle chiavi.

> **Note**
>
> Lo spazio dei nomi e il nome della chiave sono equivalenti ai campi Provider ID (o Provider) e Key ID di. `JceMasterKey`

Se si creano portachiavi diversi per crittografare e decrittografare un determinato record, lo spazio dei nomi e i valori dei nomi sono fondamentali. Se lo spazio dei nomi della chiave e il nome della chiave nel portachiavi di decrittografia non corrispondono esattamente, con distinzione tra maiuscole e minuscole, per lo spazio dei nomi della chiave e il nome della chiave nel portachiavi di crittografia, il portachiavi di decrittografia non viene utilizzato, anche se le chiavi appartengono alla stessa coppia di chiavi.

Lo spazio dei nomi e il nome chiave del materiale chiave nei portachiavi di crittografia e decrittografia devono essere gli stessi indipendentemente dal fatto che il portachiavi contenga la chiave pubblica RSA, la chiave privata RSA o entrambe le chiavi della coppia di chiavi. Ad esempio, supponiamo di crittografare i dati con un portachiavi RSA Raw per una chiave pubblica RSA con spazio dei nomi e nome della chiave. `HSM_01` `RSA_2048_06` Per decrittografare quei dati, costruisci un portachiavi Raw RSA con la chiave privata (o coppia di chiavi) e lo stesso namespace e nome della chiave.

Modalità di imbottitura

È necessario specificare una modalità di riempimento per i portachiavi Raw RSA utilizzati per la crittografia e la decrittografia oppure utilizzare funzionalità dell'implementazione del linguaggio che la specificano automaticamente.

AWS Encryption SDK Supporta le seguenti modalità di riempimento, soggette ai vincoli di ogni lingua. Consigliamo una modalità di riempimento OAEP, in particolare OAEP con SHA-256 e con imbottitura SHA-256. MGF1 La modalità padding è supportata solo per la compatibilità con le versioni precedenti. PKCS1

- OAEP con SHA-1 e con imbottitura SHA-1 MGF1
- OAEP con SHA-256 e con imbottitura SHA-256 MGF1
- OAEP con SHA-384 e con imbottitura SHA-384 MGF1
- OAEP con SHA-512 e MGF1 con imbottitura SHA-512
- PKCS1 Imbottitura v1.5

Il seguente esempio Java mostra come creare un portachiavi Raw RSA con la chiave pubblica e privata di una coppia di chiavi RSA e l'OAEP con SHA-256 e con modalità padding SHA-256. MGF1 Le variabili and rappresentano il materiale chiave fornito. `RSAPublicKey` `RSAPrivateKey`

Java

```
final CreateRawRsaKeyringInput keyringInput = CreateRawRsaKeyringInput.builder()
        .keyName("RSA_2048_06")
        .keyNamespace("HSM_01")
        .paddingScheme(PaddingScheme.OAEP_SHA256_MGF1)
        .publicKey(RSAPublicKey)
        .privateKey(RSAPrivateKey)
        .build();
final MaterialProviders matProv = MaterialProviders.builder()
        .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
        .build();
IKeyring rawRsaKeyring = matProv.CreateRawRsaKeyring(keyringInput);
```

C# / .NET

```
var keyNamespace = "HSM_01";
var keyName = "RSA_2048_06";

// Get public and private keys from PEM files
var publicKey = new
 MemoryStream(System.IO.File.ReadAllBytes("RSAKeyringExamplePublicKey.pem"));
var privateKey = new
 MemoryStream(System.IO.File.ReadAllBytes("RSAKeyringExamplePrivateKey.pem"));

// Create the keyring input
var keyringInput = new CreateRawRsaKeyringInput
{
    KeyNamespace = keyNamespace,
    KeyName = keyName,
    PaddingScheme = PaddingScheme.OAEP_SHA512_MGF1,
    PublicKey = publicKey,
    PrivateKey = privateKey
};

// Create the keyring
var matProv = new MaterialProviders(new MaterialProvidersConfig());
var rawRsaKeyring = matProv.CreateRawRsaKeyring(keyringInput);
```

Rust

```
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;
let raw_rsa_keyring = mpl
    .create_raw_rsa_keyring()
    .key_name("RSA_2048_06")
    .key_namespace("HSM_01")
    .padding_scheme(PaddingScheme::OaepSha256Mgf1)
    .public_key(RSA_public_key)
    .private_key(RSA_private_key)
    .send()
    .await?;
```

# Portachiavi ECDH grezzi

La nostra libreria di crittografia lato client è stata rinominata Database Encryption SDK. AWS Questa guida per sviluppatori fornisce ancora informazioni sul DynamoDB Encryption Client.

**Important**

Il portachiavi Raw ECDH è disponibile solo con la versione 1.5.0 della Material Providers Library.

Il portachiavi Raw ECDH utilizza le coppie di chiavi pubblico-private a curva ellittica fornite dall'utente per ricavare una chiave di wrapping condivisa tra due parti. Innanzitutto, il portachiavi ricava un segreto condiviso utilizzando la chiave privata del mittente, la chiave pubblica del destinatario e l'algoritmo di accordo delle chiavi Elliptic Curve Diffie-Hellman (ECDH). Quindi, il portachiavi utilizza il segreto condiviso per derivare la chiave di avvolgimento condivisa che protegge le chiavi di crittografia dei dati. La funzione di derivazione delle chiavi utilizzata da AWS Database Encryption SDK (KDF_CTR_HMAC_SHA384) per derivare la chiave di wrapping condivisa è conforme alle raccomandazioni del NIST per la derivazione delle chiavi.

La funzione di derivazione delle chiavi restituisce 64 byte di materiale di codifica. Per garantire che entrambe le parti utilizzino il materiale di codifica corretto, AWS Database Encryption SDK utilizza i primi 32 byte come chiave di commit e gli ultimi 32 byte come chiave di wrapping condivisa. In

caso di decrittografia, se il portachiavi non è in grado di riprodurre la stessa chiave di impegno e la stessa chiave di wrapping condivisa memorizzate nel campo di descrizione del materiale del record crittografato, l'operazione ha esito negativo. Ad esempio, se si crittografa un record con un portachiavi configurato con la chiave privata di Alice e la chiave pubblica di Bob, un portachiavi configurato con la chiave privata di Bob e la chiave pubblica di Alice riprodurrà la stessa chiave di impegno e la stessa chiave di wrapping condivisa e sarà in grado di decrittografare il record. Se la chiave pubblica di Bob proviene da una AWS KMS key coppia, allora Bob può creare un AWS KMS portachiavi ECDH per decrittografare il record.

Il portachiavi Raw ECDH crittografa i record con una chiave simmetrica utilizzando AES-GCM. La chiave dati viene quindi crittografata in busta con la chiave di wrapping condivisa derivata utilizzando AES-GCM. Ogni portachiavi Raw ECDH può avere solo una chiave di avvolgimento condivisa, ma è possibile includere più portachiavi Raw ECDH, da soli o con altri portachiavi, in un portachiavi multiplo.

L'utente è responsabile della generazione, dell'archiviazione e della protezione delle chiavi private, preferibilmente in un modulo di sicurezza hardware (HSM) o in un sistema di gestione delle chiavi. Le coppie di chiavi del mittente e del destinatario devono trovarsi sulla stessa curva ellittica. Il AWS Database Encryption SDK supporta le seguenti specifiche della curva ellittica:

- `ECC_NIST_P256`
- `ECC_NIST_P384`
- `ECC_NIST_P512`

## Creazione di un portachiavi ECDH non elaborato

Il portachiavi Raw ECDH supporta tre schemi di accordi chiave:, e. `RawPrivateKeyToStaticPublicKey` `EphemeralPrivateKeyToStaticPublicKey` `PublicKeyDiscovery` Lo schema di accordo chiave selezionato determina le operazioni crittografiche che è possibile eseguire e il modo in cui vengono assemblati i materiali di codifica.

Argomenti

- RawPrivateKeyToStaticPublicKey
- EphemeralPrivateKeyToStaticPublicKey
- PublicKeyDiscovery

### RawPrivateKeyToStaticPublicKey

Utilizza lo schema degli accordi `RawPrivateKeyToStaticPublicKey` chiave per configurare staticamente la chiave privata del mittente e la chiave pubblica del destinatario nel portachiavi. Questo schema di accordo chiave può crittografare e decrittografare i record.

Per inizializzare un portachiavi Raw ECDH con lo schema degli accordi `RawPrivateKeyToStaticPublicKey` chiave, fornite i seguenti valori:

- Chiave privata del mittente

  È necessario fornire la chiave privata con codifica PEM del mittente ( PrivateKeyInfo strutture PKCS #8), come definita in RFC 5958.

- Chiave pubblica del destinatario

  È necessario fornire la chiave pubblica X.509 con codifica DER del destinatario, nota anche come `SubjectPublicKeyInfo` (SPKI), come definita in RFC 5280.

  È possibile specificare la chiave pubblica di una coppia di chiavi KMS con accordo di chiave asimmetrico o la chiave pubblica da una coppia di chiavi generata all'esterno di. AWS

- Specificazione della curva

  Identifica la specifica della curva ellittica nelle coppie di chiavi specificate. Entrambe le coppie di chiavi del mittente e del destinatario devono avere la stessa specifica di curva.

  Valori validi: `ECC_NIST_P256`, `ECC_NIS_P384`, `ECC_NIST_P512`

C# / .NET

```
// Instantiate material providers
var materialProviders = new MaterialProviders(new MaterialProvidersConfig());
     var BobPrivateKey = new MemoryStream(new byte[] { });
     var AlicePublicKey = new MemoryStream(new byte[] { });

     // Create the Raw ECDH static keyring
     var staticConfiguration = new RawEcdhStaticConfigurations()
     {
      RawPrivateKeyToStaticPublicKey = new RawPrivateKeyToStaticPublicKeyInput
      {
       SenderStaticPrivateKey = BobPrivateKey,
       RecipientPublicKey = AlicePublicKey
```

```
     }
    };

    var createKeyringInput = new CreateRawEcdhKeyringInput()
    {
     CurveSpec = ECDHCurveSpec.ECC_NIST_P256,
     KeyAgreementScheme = staticConfiguration
    };

    var keyring = materialProviders.CreateRawEcdhKeyring(createKeyringInput);
```

Java

Il seguente esempio di Java utilizza lo schema di accordo delle `RawPrivateKeyToStaticPublicKey` chiavi per configurare staticamente la chiave privata del mittente e la chiave pubblica del destinatario. Entrambe le coppie di chiavi sono sulla `ECC_NIST_P256` curva.

```
private static void StaticRawKeyring() {
    // Instantiate material providers
    final MaterialProviders materialProviders =
      MaterialProviders.builder()
        .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
        .build();

    KeyPair senderKeys = GetRawEccKey();
    KeyPair recipient = GetRawEccKey();

    // Create the Raw ECDH static keyring
    final CreateRawEcdhKeyringInput rawKeyringInput =
      CreateRawEcdhKeyringInput.builder()
        .curveSpec(ECDHCurveSpec.ECC_NIST_P256)
        .KeyAgreementScheme(
          RawEcdhStaticConfigurations.builder()
            .RawPrivateKeyToStaticPublicKey(
                RawPrivateKeyToStaticPublicKeyInput.builder()
                  // Must be a PEM-encoded private key

 .senderStaticPrivateKey(ByteBuffer.wrap(senderKeys.getPrivate().getEncoded()))
                  // Must be a DER-encoded X.509 public key

 .recipientPublicKey(ByteBuffer.wrap(recipient.getPublic().getEncoded()))
                  .build()
```

```
            )
            .build()
        ).build();

    final IKeyring staticKeyring =
 materialProviders.CreateRawEcdhKeyring(rawKeyringInput);
}
```

Rust

Il seguente esempio di Python utilizza lo schema di accordo delle `raw_ecdh_static_configuration` chiavi per configurare staticamente la chiave privata del mittente e la chiave pubblica del destinatario. Entrambe le coppie di chiavi devono trovarsi sulla stessa curva.

```
// Create keyring input
let raw_ecdh_static_configuration_input =
    RawPrivateKeyToStaticPublicKeyInput::builder()
        // Must be a UTF8 PEM-encoded private key
        .sender_static_private_key(private_key_sender_utf8_bytes)
        // Must be a UTF8 DER-encoded X.509 public key
        .recipient_public_key(public_key_recipient_utf8_bytes)
        .build()?;

let raw_ecdh_static_configuration =
 RawEcdhStaticConfigurations::RawPrivateKeyToStaticPublicKey(raw_ecdh_static_configuration_i

// Instantiate the material providers library
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;

// Create raw ECDH static keyring
let raw_ecdh_keyring = mpl
    .create_raw_ecdh_keyring()
    .curve_spec(ecdh_curve_spec)
    .key_agreement_scheme(raw_ecdh_static_configuration)
    .send()
    .await?;
```

### EphemeralPrivateKeyToStaticPublicKey

I portachiavi configurati con lo schema `EphemeralPrivateKeyToStaticPublicKey` key agreement creano una nuova coppia di chiavi localmente e derivano una chiave di wrapping condivisa unica per ogni chiamata crittografata.

Questo schema di accordo chiave può solo crittografare i record. Per decrittografare i record crittografati con lo schema del contratto di `EphemeralPrivateKeyToStaticPublicKey` chiave, è necessario utilizzare uno schema di accordo di chiave di rilevamento configurato con la chiave pubblica dello stesso destinatario. Per decrittografare, è possibile utilizzare un portachiavi ECDH non elaborato con l'algoritmo di accordo `PublicKeyDiscovery`chiave oppure, se la chiave pubblica del destinatario proviene da una coppia di chiavi KMS con accordo di chiave asimmetrico, è possibile AWS KMS utilizzare un portachiavi ECDH con lo schema di accordo chiave. KmsPublicKeyDiscovery

Per inizializzare un portachiavi ECDH non elaborato con lo schema di accordo chiave, fornisci i seguenti valori: `EphemeralPrivateKeyToStaticPublicKey`

- Chiave pubblica del destinatario

  È necessario fornire la chiave pubblica X.509 con codifica DER del destinatario, nota anche come `SubjectPublicKeyInfo` (SPKI), come definita in RFC 5280.

  È possibile specificare la chiave pubblica di una coppia di chiavi KMS con accordo di chiave asimmetrico o la chiave pubblica da una coppia di chiavi generata all'esterno di. AWS
- Specificazione della curva

  Identifica la specifica della curva ellittica nella chiave pubblica specificata.

  In encrypt, il portachiavi crea una nuova coppia di chiavi sulla curva specificata e utilizza la nuova chiave privata e la chiave pubblica specificata per derivare una chiave di wrapping condivisa.

  Valori validi: `ECC_NIST_P256`, `ECC_NIS_P384`, `ECC_NIST_P512`

C# / .NET

L'esempio seguente crea un portachiavi ECDH non elaborato con lo schema di accordo delle chiavi. `EphemeralPrivateKeyToStaticPublicKey` Su encrypt, il portachiavi creerà una nuova coppia di chiavi localmente sulla curva specificata`ECC_NIST_P256`.

```
// Instantiate material providers
```

```
var materialProviders = new MaterialProviders(new MaterialProvidersConfig());
     var AlicePublicKey = new MemoryStream(new byte[] { });

     // Create the Raw ECDH ephemeral keyring
     var ephemeralConfiguration = new RawEcdhStaticConfigurations()
     {
      EphemeralPrivateKeyToStaticPublicKey = new
 EphemeralPrivateKeyToStaticPublicKeyInput
      {
       RecipientPublicKey = AlicePublicKey
      }
     };

     var createKeyringInput = new CreateRawEcdhKeyringInput()
     {
      CurveSpec = ECDHCurveSpec.ECC_NIST_P256,
      KeyAgreementScheme = ephemeralConfiguration
     };

     var keyring = materialProviders.CreateRawEcdhKeyring(createKeyringInput);
```

Java

L'esempio seguente crea un portachiavi Raw ECDH con lo schema di accordo chiave. `EphemeralPrivateKeyToStaticPublicKey` Su encrypt, il portachiavi creerà una nuova coppia di chiavi localmente sulla curva specificata`ECC_NIST_P256`.

```
private static void EphemeralRawEcdhKeyring() {
    // Instantiate material providers
    final MaterialProviders materialProviders =
      MaterialProviders.builder()
        .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
        .build();

    ByteBuffer recipientPublicKey = getPublicKeyBytes();

    // Create the Raw ECDH ephemeral keyring
    final CreateRawEcdhKeyringInput ephemeralInput =
      CreateRawEcdhKeyringInput.builder()
        .curveSpec(ECDHCurveSpec.ECC_NIST_P256)
        .KeyAgreementScheme(
          RawEcdhStaticConfigurations.builder()
            .EphemeralPrivateKeyToStaticPublicKey(
```

```
                EphemeralPrivateKeyToStaticPublicKeyInput.builder()
                  .recipientPublicKey(recipientPublicKey)
                  .build()
              )
              .build()
        ).build();

    final IKeyring ephemeralKeyring =
 materialProviders.CreateRawEcdhKeyring(ephemeralInput);
}
```

Rust

L'esempio seguente crea un portachiavi Raw ECDH con lo schema di accordo chiave. `ephemeral_raw_ecdh_static_configuration` Su encrypt, il portachiavi creerà una nuova coppia di chiavi localmente sulla curva specificata.

```
// Create EphemeralPrivateKeyToStaticPublicKeyInput
let ephemeral_raw_ecdh_static_configuration_input =
    EphemeralPrivateKeyToStaticPublicKeyInput::builder()
        // Must be a UTF8 DER-encoded X.509 public key
        .recipient_public_key(public_key_recipient_utf8_bytes)
        .build()?;

let ephemeral_raw_ecdh_static_configuration =

 RawEcdhStaticConfigurations::EphemeralPrivateKeyToStaticPublicKey(ephemeral_raw_ecdh_static

// Instantiate the material providers library
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;

// Create raw ECDH ephemeral private key keyring
let ephemeral_raw_ecdh_keyring = mpl
    .create_raw_ecdh_keyring()
    .curve_spec(ecdh_curve_spec)
    .key_agreement_scheme(ephemeral_raw_ecdh_static_configuration)
    .send()
    .await?;
```

### PublicKeyDiscovery

Durante la decrittografia, è consigliabile specificare le chiavi di wrapping che il AWS Database Encryption SDK può utilizzare. Per seguire questa best practice, utilizzate un portachiavi ECDH che specifichi sia la chiave privata del mittente che la chiave pubblica del destinatario. Tuttavia, puoi anche creare un portachiavi Raw ECDH Discovery, ovvero un portachiavi ECDH non elaborato in grado di decrittografare qualsiasi record in cui la chiave pubblica della chiave specificata corrisponda alla chiave pubblica del destinatario memorizzata nel campo di descrizione del materiale del record crittografato. Questo schema di accordi chiave può solo decrittografare i record.

> **Important**
>
> Quando si decrittografano i record utilizzando lo schema del contratto di `PublicKeyDiscovery` chiave, si accettano tutte le chiavi pubbliche, indipendentemente dal proprietario.

Per inizializzare un portachiavi Raw ECDH con lo schema di accordo `PublicKeyDiscovery` chiave, fornite i seguenti valori:

- Chiave privata statica del destinatario

  È necessario fornire la chiave privata con codifica PEM del destinatario ( PrivateKeyInfo strutture PKCS #8), come definita in RFC 5958.

- Specificazione della curva

  Identifica la specifica della curva ellittica nella chiave privata specificata. Entrambe le coppie di chiavi del mittente e del destinatario devono avere la stessa specifica di curva.

  Valori validi: `ECC_NIST_P256`, `ECC_NIS_P384`, `ECC_NIST_P512`

C# / .NET

L'esempio seguente crea un portachiavi Raw ECDH con lo schema di accordo chiave. `PublicKeyDiscovery` Questo portachiavi può decrittografare qualsiasi record in cui la chiave pubblica della chiave privata specificata corrisponda alla chiave pubblica del destinatario memorizzata nel campo di descrizione del materiale del record crittografato.

```
// Instantiate material providers
```

```
var materialProviders = new MaterialProviders(new MaterialProvidersConfig());
     var AlicePrivateKey = new MemoryStream(new byte[] { });

     // Create the Raw ECDH discovery keyring
     var discoveryConfiguration = new RawEcdhStaticConfigurations()
     {
      PublicKeyDiscovery = new PublicKeyDiscoveryInput
      {
       RecipientStaticPrivateKey = AlicePrivateKey
      }
     };

     var createKeyringInput = new CreateRawEcdhKeyringInput()
     {
      CurveSpec = ECDHCurveSpec.ECC_NIST_P256,
      KeyAgreementScheme = discoveryConfiguration
     };

     var keyring = materialProviders.CreateRawEcdhKeyring(createKeyringInput);
```

Java

L'esempio seguente crea un portachiavi ECDH non elaborato con lo schema di accordo delle chiavi. `PublicKeyDiscovery` Questo portachiavi può decrittografare qualsiasi record in cui la chiave pubblica della chiave privata specificata corrisponda alla chiave pubblica del destinatario memorizzata nel campo di descrizione del materiale del record crittografato.

```
private static void RawEcdhDiscovery() {
    // Instantiate material providers
    final MaterialProviders materialProviders =
      MaterialProviders.builder()
        .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
        .build();

    KeyPair recipient = GetRawEccKey();

    // Create the Raw ECDH discovery keyring
    final CreateRawEcdhKeyringInput rawKeyringInput =
      CreateRawEcdhKeyringInput.builder()
        .curveSpec(ECDHCurveSpec.ECC_NIST_P256)
        .KeyAgreementScheme(
          RawEcdhStaticConfigurations.builder()
            .PublicKeyDiscovery(
```

```
              PublicKeyDiscoveryInput.builder()
                // Must be a PEM-encoded private key

 .recipientStaticPrivateKey(ByteBuffer.wrap(sender.getPrivate().getEncoded()))
                .build()
            )
            .build()
        ).build();

    final IKeyring publicKeyDiscovery  =
 materialProviders.CreateRawEcdhKeyring(rawKeyringInput);
}
```

Rust

L'esempio seguente crea un portachiavi ECDH non elaborato con lo schema di accordo delle chiavi. `discovery_raw_ecdh_static_configuration` Questo portachiavi può decrittografare qualsiasi messaggio in cui la chiave pubblica della chiave privata specificata corrisponda alla chiave pubblica del destinatario memorizzata nel testo cifrato del messaggio.

```
// Create PublicKeyDiscoveryInput
let discovery_raw_ecdh_static_configuration_input =
    PublicKeyDiscoveryInput::builder()
        // Must be a UTF8 PEM-encoded private key
        .recipient_static_private_key(private_key_recipient_utf8_bytes)
        .build()?;

let discovery_raw_ecdh_static_configuration =

 RawEcdhStaticConfigurations::PublicKeyDiscovery(discovery_raw_ecdh_static_configuration_inp

// Create raw ECDH discovery private key keyring
let discovery_raw_ecdh_keyring = mpl
    .create_raw_ecdh_keyring()
    .curve_spec(ecdh_curve_spec)
    .key_agreement_scheme(discovery_raw_ecdh_static_configuration)
    .send()
    .await?;
```

# Keyring multipli

La nostra libreria di crittografia lato client è stata rinominata AWS Database Encryption SDK. Questa guida per sviluppatori fornisce ancora informazioni sul DynamoDB Encryption Client.

È possibile combinare più keyring in un keyring multiplo. Un keyring multiplo è composto da uno o più keyring dello stesso tipo o di tipi diversi. Il risultato è analogo a quello ottenuto utilizzando diversi keyring in serie. Quando utilizzi un keyring multiplo per crittografare i dati, questi possono essere decrittati con le chiavi di wrapping contenute in qualsiasi keyring.

Quando crei un keyring multiplo per crittografare i dati, uno dei keyring viene designato come keyring generatore, tutti gli altri keyring sono i keyring figlio. che si occupa di generare e crittografare la chiave di dati di testo normale. Quindi, tutte le chiavi di wrapping in tutti i keyring figlio crittografano la stessa chiave di dati di testo normale. Il keyring multiplo restituisce la chiave di dati di testo normale e una chiave di dati crittografata per ciascuna chiave di wrapping nel keyring multiplo. Se il portachiavi del generatore è un portachiavi KMS, la chiave del generatore nel AWS KMS portachiavi genera e crittografa la chiave in testo semplice. Quindi, tutte le chiavi aggiuntive AWS KMS keys presenti nel portachiavi e tutte le AWS KMS chiavi inserite in tutti i portachiavi secondari del portachiavi multiplo crittografano la stessa chiave in chiaro.

Durante la decrittografia, AWS Database Encryption SDK utilizza i portachiavi per cercare di decrittografare una delle chiavi di dati crittografate. I keyring sono chiamati nell'ordine in cui sono specificati nel keyring multiplo. L'elaborazione si interrompe non appena una chiave in qualsiasi keyring può decrittare una chiave di dati crittografata.

Per creare un keyring multiplo, crea prima un'istanza dei keyring figlio. In questo esempio, utilizziamo un AWS KMS portachiavi e un portachiavi Raw AES, ma puoi combinare tutti i portachiavi supportati in un portachiavi multiplo.

Java

```
// 1. Create the raw AES keyring.
final MaterialProviders matProv = MaterialProviders.builder()
        .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
        .build();
final CreateRawAesKeyringInput createRawAesKeyringInput =
 CreateRawAesKeyringInput.builder()
        .keyName("AES_256_012")
```

```
        .keyNamespace("HSM_01")
        .wrappingKey(AESWrappingKey)
        .wrappingAlg(AesWrappingAlg.ALG_AES256_GCM_IV12_TAG16)
        .build();
IKeyring rawAesKeyring = matProv.CreateRawAesKeyring(createRawAesKeyringInput);

// 2. Create the AWS KMS keyring.
final CreateAwsKmsMrkMultiKeyringInput createAwsKmsMrkMultiKeyringInput =
 CreateAwsKmsMrkMultiKeyringInput.builder()
        .generator(kmsKeyArn)
        .build();
IKeyring awsKmsMrkMultiKeyring =
 matProv.CreateAwsKmsMrkMultiKeyring(createAwsKmsMrkMultiKeyringInput);
```

## C# / .NET

```
// 1. Create the raw AES keyring.
var keyNamespace = "HSM_01";
var keyName = "AES_256_012";

var matProv = new MaterialProviders(new MaterialProvidersConfig());
var createRawAesKeyringInput = new CreateRawAesKeyringInput
{
    KeyName = "keyName",
    KeyNamespace = "myNamespaces",
    WrappingKey = AESWrappingKey,
    WrappingAlg = AesWrappingAlg.ALG_AES256_GCM_IV12_TAG16
};
var rawAesKeyring = matProv.CreateRawAesKeyring(createRawAesKeyringInput);

// 2. Create the AWS KMS keyring.
//    We create a MRK multi keyring, as this interface also supports
//    single-region KMS keys,
//    and creates the KMS client for us automatically.
var createAwsKmsMrkMultiKeyringInput = new CreateAwsKmsMrkMultiKeyringInput
{
    Generator = keyArn
};
var awsKmsMrkMultiKeyring =
 matProv.CreateAwsKmsMrkMultiKeyring(createAwsKmsMrkMultiKeyringInput);
```

Rust

```
// 1. Create the raw AES keyring
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;

let raw_aes_keyring = mpl
    .create_raw_aes_keyring()
    .key_name("AES_256_012")
    .key_namespace("HSM_01")
    .wrapping_key(aes_key_bytes)
    .wrapping_alg(AesWrappingAlg::AlgAes256GcmIv12Tag16)
    .send()
    .await?;

// 2. Create the AWS KMS keyring
let aws_kms_mrk_multi_keyring = mpl
    .create_aws_kms_mrk_multi_keyring()
    .generator(key_arn)
    .send()
    .await?;
```

Crea quindi il keyring multiplo e specifica il keyring generatore, se presente. In questo esempio, creiamo un portachiavi multiplo in cui il portachiavi è il portachiavi del generatore e il AWS KMS portachiavi AES è il portachiavi per bambini.

Java

Il `CreateMultiKeyringInput` costruttore Java consente di definire un portachiavi del generatore e dei portachiavi secondari. L'`createMultiKeyringInput`oggetto risultante è immutabile.

```
final CreateMultiKeyringInput createMultiKeyringInput =
 CreateMultiKeyringInput.builder()
        .generator(awsKmsMrkMultiKeyring)
        .childKeyrings(Collections.singletonList(rawAesKeyring))
        .build();
IKeyring multiKeyring = matProv.CreateMultiKeyring(createMultiKeyringInput);
```

C# / .NET

Il `CreateMultiKeyringInput` costruttore.NET consente di definire un portachiavi del generatore e dei portachiavi secondari. L'`CreateMultiKeyringInput`oggetto risultante è immutabile.

```
var createMultiKeyringInput = new CreateMultiKeyringInput
{
    Generator = awsKmsMrkMultiKeyring,
    ChildKeyrings = new List<IKeyring> { rawAesKeyring }
};
var multiKeyring = matProv.CreateMultiKeyring(createMultiKeyringInput);
```

Rust

```
let multi_keyring = mpl
    .create_multi_keyring()
    .generator(aws_kms_mrk_multi_keyring)
    .child_keyrings(vec![raw_aes_keyring.clone()])
    .send()
    .await?;
```

Ora puoi utilizzare il keyring multiplo per crittografare e decrittare i dati.

# Crittografia ricercabile

La nostra libreria di crittografia lato client è stata rinominata Database Encryption SDK. AWS Questa guida per sviluppatori fornisce ancora informazioni sul DynamoDB Encryption Client.

La crittografia ricercabile consente di cercare record crittografati senza decrittografare l'intero database. Questa operazione viene eseguita utilizzando i beacon, che creano una mappa tra il valore di testo in chiaro scritto in un campo e il valore crittografato effettivamente memorizzato nel database. Il AWS Database Encryption SDK memorizza il beacon in un nuovo campo che aggiunge al record. A seconda del tipo di beacon utilizzato, è possibile eseguire ricerche con corrispondenza esatta o query complesse più personalizzate sui dati crittografati.

> **Note**
>
> La crittografia ricercabile nel AWS Database Encryption SDK è diversa dalla crittografia simmetrica ricercabile definita nella ricerca accademica, come la crittografia simmetrica ricercabile.

Un beacon è un tag HMAC (Hash-Based Message Authentication Code) troncato che crea una mappa tra il testo semplice e i valori crittografati di un campo. Quando scrivi un nuovo valore in un campo crittografato configurato per la crittografia ricercabile, AWS Database Encryption SDK calcola un HMAC sul valore di testo in chiaro. Questo output HMAC corrisponde uno a uno (1:1) per il valore in chiaro di quel campo. L'output HMAC viene troncato in modo che più valori di testo in chiaro distinti vengano mappati allo stesso tag HMAC troncato. Questi falsi positivi limitano la capacità di un utente non autorizzato di identificare informazioni distintive sul valore del testo in chiaro. Quando si esegue una query su un beacon, il AWS Database Encryption SDK filtra automaticamente questi falsi positivi e restituisce il risultato in testo non crittografato della query.

Il numero medio di falsi positivi generati per ogni beacon è determinato dalla lunghezza del beacon rimanente dopo il troncamento. Per informazioni sulla determinazione della lunghezza del beacon appropriata per l'implementazione, consulta Determinazione della lunghezza del beacon.

> **Note**
>
> La crittografia ricercabile è progettata per essere implementata in nuovi database non popolati. Qualsiasi beacon configurato in un database esistente mapperà solo i nuovi record caricati nel database, non è possibile per un beacon mappare i dati esistenti.

Argomenti



# I beacon sono adatti al mio set di dati?

L'uso dei beacon per eseguire query su dati crittografati riduce i costi prestazionali associati ai database crittografati lato client. Quando si utilizzano i beacon, esiste un compromesso intrinseco tra l'efficienza delle query e la quantità di informazioni rivelate sulla distribuzione dei dati. Il beacon non altera lo stato crittografato del campo. Quando si crittografa e si firma un campo con AWS Database Encryption SDK, il valore in chiaro del campo non viene mai esposto al database. Il database memorizza il valore crittografato e randomizzato del campo.

I beacon vengono memorizzati insieme ai campi crittografati da cui vengono calcolati. Ciò significa che anche se un utente non autorizzato non è in grado di visualizzare i valori in chiaro di un campo crittografato, potrebbe essere in grado di eseguire analisi statistiche sui beacon per saperne di più sulla distribuzione del set di dati e, in casi estremi, identificare i valori di testo in chiaro a cui un beacon è mappato. Il modo in cui configuri i beacon può mitigare questi rischi. In particolare, la scelta della giusta lunghezza del beacon può aiutarti a preservare la riservatezza del tuo set di dati.

Sicurezza vs. prestazioni

- Più breve è la lunghezza del faro, maggiore è la sicurezza.
- Maggiore è la lunghezza del faro, maggiori sono le prestazioni preservate.

La crittografia ricercabile potrebbe non essere in grado di fornire i livelli desiderati di prestazioni e sicurezza per tutti i set di dati. Esamina il modello di minaccia, i requisiti di sicurezza e le esigenze prestazionali prima di configurare qualsiasi beacon.

Prendi in considerazione i seguenti requisiti di unicità del set di dati per determinare se la crittografia ricercabile è adatta al tuo set di dati.

Distribuzione

La quantità di sicurezza garantita da un beacon dipende dalla distribuzione del set di dati. Quando configuri un campo crittografato per una crittografia ricercabile, AWS Database Encryption SDK calcola un HMAC sui valori di testo in chiaro scritti in quel campo. Tutti i beacon calcolati per un determinato campo vengono calcolati utilizzando la stessa chiave, ad eccezione dei database multitenant che utilizzano una chiave distinta per ogni tenant. Ciò significa che se lo stesso valore di testo in chiaro viene scritto più volte nel campo, viene creato lo stesso tag HMAC per ogni istanza di quel valore di testo in chiaro.

Dovresti evitare di creare beacon a partire da campi che contengono valori molto comuni. Ad esempio, si consideri un database che memorizza l'indirizzo di tutti i residenti dello stato dell'Illinois. Se costruisci un faro utilizzando il `City` campo criptato, il beacon calcolato su «Chicago» sarà sovrarappresentato a causa della grande percentuale della popolazione dell'Illinois che vive a Chicago. Anche se un utente non autorizzato può leggere solo i valori crittografati e i valori del beacon, potrebbe essere in grado di identificare quali record contengono dati relativi ai residenti di Chicago se il beacon mantiene questa distribuzione. Per ridurre al minimo la quantità di informazioni distintive rivelate sulla distribuzione, è necessario troncare sufficientemente il beacon. La lunghezza del beacon necessaria per nascondere questa distribuzione irregolare comporta costi prestazionali significativi che potrebbero non soddisfare le esigenze dell'applicazione.

È necessario analizzare attentamente la distribuzione del set di dati per determinare in che misura i beacon devono essere troncati. La lunghezza del beacon rimanente dopo il troncamento è direttamente correlata alla quantità di informazioni statistiche che è possibile identificare sulla distribuzione. Potrebbe essere necessario scegliere beacon di lunghezza inferiore per ridurre al minimo la quantità di informazioni distintive rivelate sul set di dati.

In casi estremi, non è possibile calcolare la lunghezza del beacon per un set di dati distribuito in modo non uniforme che bilanci efficacemente prestazioni e sicurezza. Ad esempio, non si dovrebbe costruire un faro partendo da un campo che memorizza il risultato di un test medico per una malattia rara. Poiché si prevede che `NEGATIVE` i risultati siano significativamente più diffusi all'interno del set di dati, i `POSITIVE` risultati possono essere facilmente identificati in base alla loro rarità. È molto difficile nascondere la distribuzione quando il campo ha solo due valori possibili. Se si utilizza una lunghezza del beacon sufficientemente breve da nascondere la distribuzione, tutti i valori in testo semplice vengono mappati allo stesso tag HMAC. Se si utilizza

un beacon di lunghezza maggiore, è ovvio quali beacon vengono mappati su valori in chiaro. `POSITIVE`

Correlazione

Si consiglia vivamente di evitare di creare beacon distinti a partire da campi con valori correlati. I beacon creati da campi correlati richiedono beacon di lunghezza inferiore per ridurre al minimo in misura sufficiente la quantità di informazioni rivelate sulla distribuzione di ciascun set di dati a un utente non autorizzato. È necessario analizzare attentamente il set di dati, compresa l'entropia e la distribuzione congiunta dei valori correlati, per determinare in che misura i beacon devono essere troncati. Se la lunghezza del beacon risultante non soddisfa le esigenze prestazionali, i beacon potrebbero non essere adatti al set di dati.

Ad esempio, non dovreste creare due beacon `City` e due `ZIPCode` campi separati perché il codice postale sarà probabilmente associato a una sola città. In genere, i falsi positivi generati da un beacon limitano la capacità di un utente non autorizzato di identificare le informazioni distintive sul set di dati. Tuttavia, la correlazione tra i `ZIPCode` campi `City` e significa che un utente non autorizzato può identificare facilmente quali risultati sono falsi positivi e distinguere i diversi codici postali.

È inoltre consigliabile evitare di creare beacon a partire da campi che contengono gli stessi valori in chiaro. Ad esempio, non dovreste costruire un beacon partendo da `preferredPhone` campi `mobilePhone` e perché probabilmente contengono gli stessi valori. Se costruisci beacon distinti da entrambi i campi, AWS Database Encryption SDK crea i beacon per ogni campo con chiavi diverse. Ciò si traduce in due tag HMAC diversi per lo stesso valore di testo in chiaro. È improbabile che i due beacon distinti abbiano gli stessi falsi positivi e un utente non autorizzato potrebbe essere in grado di distinguere numeri di telefono diversi.

Anche se il set di dati contiene campi correlati o ha una distribuzione non uniforme, è possibile creare beacon che preservino la riservatezza del set di dati utilizzando beacon di lunghezza inferiore. Tuttavia, la lunghezza del beacon non garantisce che ogni valore univoco nel set di dati produca una serie di falsi positivi che riducono efficacemente al minimo la quantità di informazioni distintive rivelate sul set di dati. La lunghezza del beacon stima solo il numero medio di falsi positivi prodotti. Più il set di dati è distribuito in modo non uniforme, minore è la lunghezza del faro nel determinare il numero medio di falsi positivi prodotti.

Valuta attentamente la distribuzione dei campi da cui costruisci i beacon e valuta quanto ti servirà per troncare la lunghezza del beacon per soddisfare i tuoi requisiti di sicurezza. Gli argomenti seguenti di

questo capitolo presuppongono che i beacon siano distribuiti uniformemente e non contengano dati correlati.

## Scenario di crittografia ricercabile

L'esempio seguente illustra una semplice soluzione di crittografia ricercabile. In applicazione, i campi di esempio utilizzati in questo esempio potrebbero non soddisfare le raccomandazioni sull'unicità di distribuzione e correlazione per i beacon. Puoi usare questo esempio come riferimento mentre leggi i concetti di crittografia ricercabile in questo capitolo.

Prendiamo in considerazione un database denominato `Employees` che tiene traccia dei dati dei dipendenti di un'azienda. Ogni record del database contiene campi denominati EmployeeID LastNameFirstName, e Address. Ogni campo del `Employees` database è identificato dalla chiave primaria. `EmployeeID`

Di seguito è riportato un esempio di record di testo in chiaro nel database.

```
{
    "EmployeeID": 101,
    "LastName": "Jones",
    "FirstName": "Mary",
    "Address": {
                "Street": "123 Main",
                "City": "Anytown",
                "State": "OH",
                "ZIPCode": 12345
    }
}
```

Se hai contrassegnato i `FirstName` campi `LastName` and come `ENCRYPT_AND_SIGN` nelle tue azioni crittografiche, i valori in questi campi vengono crittografati localmente prima di essere caricati nel database. I dati crittografati caricati sono completamente randomizzati, il database non riconosce questi dati come protetti. Rileva solo le immissioni di dati tipiche. Ciò significa che il record effettivamente archiviato nel database potrebbe avere il seguente aspetto.

```
{
    "PersonID": 101,
    "LastName": "1d76e94a2063578637d51371b363c9682bad926cbd",
    "FirstName": "21d6d54b0aaabc411e9f9b34b6d53aa4ef3b0a35",
    "Address": {
                "Street": "123 Main",
```

```
                "City": "Anytown",
                "State": "OH",
                "ZIPCode": 12345
    }
}
```

Se devi interrogare il database per verificare le corrispondenze esatte sul `LastName` campo, configura un beacon standard denominato LastNameper mappare i valori in chiaro scritti nel `LastName` campo ai valori crittografati memorizzati nel database.

Questo beacon esegue il calcolo in base ai valori in chiaro presenti nel HMACs campo. `LastName` Ogni output HMAC viene troncato in modo che non corrisponda più esattamente al valore del testo in chiaro. Ad esempio, l'hash completo e l'hash troncato per potrebbero essere simili ai seguenti. `Jones`

Hash completo

`2aa4e9b404c68182562b6ec761fcca5306de527826a69468885e59dc36d0c3f824bdd44cab45526f`

Hash troncato

`b35099d408c833`

Dopo aver configurato il beacon standard, è possibile eseguire ricerche di uguaglianza sul campo. `LastName` Ad esempio, se desideri cercare`Jones`, usa il LastNamebeacon per eseguire la seguente query.

```
LastName = Jones
```

AWS Database Encryption SDK filtra automaticamente i falsi positivi e restituisce il risultato in testo non crittografato della query.

# Fari

> La nostra libreria di crittografia lato client è stata rinominata Database Encryption SDK. AWS Questa guida per sviluppatori fornisce ancora informazioni sul DynamoDB Encryption Client.

Un beacon è un tag HMAC (Hash-Based Message Authentication Code) troncato che crea una mappa tra il valore in chiaro scritto in un campo e il valore crittografato effettivamente memorizzato nel database. Il beacon non altera lo stato crittografato del campo. Il beacon calcola un HMAC

sul valore di testo in chiaro del campo e lo memorizza insieme al valore crittografato. Questo output HMAC corrisponde uno a uno (1:1) per il valore in chiaro di quel campo. L'output HMAC viene troncato in modo che più valori di testo in chiaro distinti vengano mappati allo stesso tag HMAC troncato. Questi falsi positivi limitano la capacità di un utente non autorizzato di identificare informazioni distintive sul valore del testo in chiaro.

I beacon possono essere creati solo a partire da campi contrassegnati o nelle azioni `ENCRYPT_AND_SIGN` crittografiche`SIGN_ONLY`. `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` Il beacon stesso non è firmato o crittografato. Non è possibile costruire un beacon con campi contrassegnati. `DO_NOTHING`

Il tipo di beacon configurato determina il tipo di interrogazioni che è possibile eseguire. Esistono due tipi di beacon che supportano la crittografia ricercabile. I beacon standard eseguono ricerche di uguaglianza. I beacon composti combinano stringhe letterali in chiaro e beacon standard per eseguire operazioni complesse sul database. Dopo aver configurato i beacon, è necessario configurare un indice secondario per ogni beacon prima di poter eseguire la ricerca nei campi crittografati. Per ulteriori informazioni, consulta Configurazione degli indici secondari con beacon.

Argomenti

- Beacon standard
- Beacon composti

## Beacon standard

I beacon standard sono il modo più semplice per implementare la crittografia ricercabile nel database. Possono eseguire ricerche di uguaglianza solo per un singolo campo crittografato o virtuale. Per informazioni su come configurare i beacon standard, vedere Configurazione dei beacon standard.

Il campo da cui è costruito un beacon standard è chiamato beacon source. Identifica la posizione dei dati che il beacon deve mappare. La fonte del beacon può essere un campo crittografato o un campo virtuale. La fonte del beacon in ogni beacon standard deve essere unica. Non è possibile configurare due beacon con la stessa sorgente beacon.

I beacon standard possono essere utilizzati per eseguire ricerche di uguaglianza per un campo crittografato o virtuale. In alternativa, possono essere utilizzati per costruire beacon composti per eseguire operazioni di database più complesse. Per aiutarvi a organizzare e gestire i beacon

standard, il AWS Database Encryption SDK fornisce i seguenti stili di beacon opzionali che definiscono l'uso previsto di un beacon standard. Per ulteriori informazioni, consulta Definizione degli stili di beacon.

È possibile creare un beacon standard che esegue ricerche di uguaglianza per un singolo campo crittografato oppure creare un beacon standard che esegue ricerche di uguaglianza sulla concatenazione di più campi e creando un campo virtuale. `ENCRYPT_AND_SIGN` `SIGN_ONLY` `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`

### Campi virtuali

Un campo virtuale è un campo concettuale costruito da uno o più campi di origine. La creazione di un campo virtuale non comporta la scrittura di un nuovo campo nel record. Il campo virtuale non è archiviato in modo esplicito nel database. Viene utilizzato nella configurazione standard del beacon per fornire al beacon istruzioni su come identificare un segmento specifico di un campo o concatenare più campi all'interno di un record per eseguire una query specifica. Un campo virtuale richiede almeno un campo crittografato.

> **Note**
>
> L'esempio seguente illustra i tipi di trasformazioni e interrogazioni che è possibile eseguire con un campo virtuale. In applicazione, i campi di esempio utilizzati in questo esempio potrebbero non soddisfare le raccomandazioni sull'unicità della distribuzione e della correlazione per i beacon.

Ad esempio, se si desidera eseguire ricerche di uguaglianza sulla concatenazione di `LastName` campi `FirstName` and, è possibile creare uno dei seguenti campi virtuali.

- Un `NameTag` campo virtuale, composto dalla prima lettera del `FirstName` campo, seguito dal campo, tutto in `LastName` minuscolo. Questo campo virtuale consente di eseguire interrogazioni. `NameTag=mjones`
- Un `LastFirst` campo virtuale, costruito a partire dal `LastName` campo, seguito dal `FirstName` campo. Questo campo virtuale consente di eseguire interrogazioni`LastFirst=JonesMary`.

Oppure, se desideri eseguire ricerche di uguaglianza su un segmento specifico di un campo crittografato, crea un campo virtuale che identifichi il segmento su cui desideri eseguire la query.

Ad esempio, se desideri interrogare un `IPAddress` campo crittografato utilizzando i primi tre segmenti dell'indirizzo IP, crea il seguente campo virtuale.

- Un `IPSegment` campo virtuale, costruito da`Segments('.', 0, 3)`. Questo campo virtuale consente di eseguire interrogazioni`IPSegment=192.0.2`. La query restituisce tutti i record con un `IPAddress` valore che inizia con «192.0.2".

I campi virtuali devono essere univoci. Non è possibile creare due campi virtuali a partire dagli stessi campi sorgente.

Per informazioni sulla configurazione dei campi virtuali e dei beacon che li utilizzano, consulta Creazione di un campo virtuale.

## Beacon composti

I beacon composti creano indici che migliorano le prestazioni delle query e consentono di eseguire operazioni di database più complesse. È possibile utilizzare i beacon composti per combinare stringhe di testo in chiaro letterali e beacon standard per eseguire query complesse su record crittografati, ad esempio interrogare due diversi tipi di record da un singolo indice o interrogare una combinazione di campi con una chiave di ordinamento. Per altri esempi di soluzioni beacon composte, consulta Scegliere un tipo di beacon.

I beacon composti possono essere costruiti a partire da beacon standard o da una combinazione di beacon standard e campi firmati. Sono costituiti da un elenco di parti. Tutti i beacon composti devono includere un elenco di parti crittografate che identifichi i `ENCRYPT_AND_SIGN` campi inclusi nel beacon. Ogni `ENCRYPT_AND_SIGN` campo deve essere identificato da un beacon standard. I beacon composti più complessi potrebbero includere anche un elenco di parti firmate che identificano il testo in chiaro `SIGN_ONLY` o i `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` campi inclusi nel beacon e un elenco di parti del costruttore che identificano tutti i possibili modi in cui il beacon composto può assemblare i campi.

> **Note**
>
> Il AWS Database Encryption SDK supporta anche beacon firmati che possono essere configurati interamente da testo in chiaro e campi. `SIGN_ONLY` `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` I beacon firmati sono un tipo di beacon composto che indicizza ed esegue query complesse su campi firmati, ma non crittografati. Per ulteriori informazioni, consulta Creazione di beacon firmati.

Per informazioni sulla configurazione dei beacon composti, consultate Configurazione dei beacon composti.

Il modo in cui configuri il beacon composto determina i tipi di query che può eseguire. Ad esempio, è possibile rendere opzionali alcune parti crittografate e firmate per consentire una maggiore flessibilità nelle query. Per ulteriori informazioni sui tipi di interrogazioni che i beacon composti possono eseguire, vedere. Interrogazione dei beacon

# Fari di pianificazione

> La nostra libreria di crittografia lato client è stata rinominata Database Encryption SDK. AWS Questa guida per sviluppatori fornisce ancora informazioni sul DynamoDB Encryption Client.

I beacon sono progettati per essere implementati in nuovi database non popolati. Qualsiasi beacon configurato in un database esistente mapperà solo i nuovi record scritti nel database. I beacon vengono calcolati in base al valore in chiaro di un campo, una volta crittografato il campo non è più possibile per il beacon di mappare i dati esistenti. Dopo aver scritto nuovi record con un beacon, non è possibile aggiornare la configurazione del beacon. Tuttavia, puoi aggiungere nuovi beacon per i nuovi campi che aggiungi al tuo record.

Per implementare la crittografia ricercabile, è necessario utilizzare il portachiavi AWS KMS gerarchico per generare, crittografare e decrittografare le chiavi dati utilizzate per proteggere i record. Per ulteriori informazioni, consulta Utilizzo del portachiavi gerarchico per una crittografia ricercabile.

Prima di poter configurare i beacon per la crittografia ricercabile, è necessario esaminare i requisiti di crittografia, i modelli di accesso al database e il modello di minaccia per determinare la soluzione migliore per il database.

Il tipo di beacon configurato determina il tipo di query che è possibile eseguire. La lunghezza del beacon specificata nella configurazione standard del beacon determina il numero previsto di falsi positivi prodotti per un determinato beacon. Consigliamo vivamente di identificare e pianificare i tipi di interrogazioni da eseguire prima di configurare i beacon. Dopo aver utilizzato un beacon, la configurazione non può essere aggiornata.

Si consiglia vivamente di esaminare e completare le seguenti attività prima di configurare qualsiasi beacon.

- Determina se i beacon sono adatti al tuo set di dati

- Scegli un tipo di faro
- Scegli la lunghezza del faro
- Scegli il nome del faro

Ricorda i seguenti requisiti di unicità dei beacon quando pianifichi la soluzione di crittografia ricercabile per il tuo database.

- Ogni beacon standard deve avere una fonte di beacon unica

  Non è possibile creare più beacon standard a partire dallo stesso campo criptato o virtuale.

  Tuttavia, è possibile utilizzare un singolo beacon standard per costruire più beacon composti.
- Evitate di creare un campo virtuale con campi sorgente che si sovrappongono ai beacon standard esistenti

  La creazione di un beacon standard a partire da un campo virtuale che contiene un campo sorgente utilizzato per creare un altro beacon standard può ridurre la sicurezza di entrambi i beacon.

  Per ulteriori informazioni, consulta Considerazioni sulla sicurezza per i campi virtuali.

## Considerazioni per i database multitenant

Per interrogare i beacon configurati in un database multitenant, è necessario includere nella query il campo che memorizza il record `branch-key-id` associato al tenant che ha crittografato il record. Questo campo viene definito quando si definisce la fonte della chiave beacon. Affinché la query abbia esito positivo, il valore in questo campo deve identificare i materiali chiave del beacon appropriati necessari per ricalcolare il beacon.

Prima di configurare i beacon, è necessario decidere in che modo intendi includerli nelle query. `branch-key-id` Per ulteriori informazioni sui diversi modi in cui puoi includerli `branch-key-id` nelle tue query, consulta. Interrogazione dei beacon in un database multi-tenant

## Scelta del tipo di faro

> La nostra libreria di crittografia lato client è stata rinominata Database Encryption SDK. AWS Questa guida per sviluppatori fornisce ancora informazioni sul DynamoDB Encryption Client.

Con la crittografia ricercabile, puoi cercare record crittografati mappando i valori in chiaro in un campo crittografato con un beacon. Il tipo di beacon configurato determina il tipo di interrogazioni che è possibile eseguire.

Consigliamo vivamente di identificare e pianificare i tipi di query da eseguire prima di configurare i beacon. Dopo aver configurato i beacon, è necessario configurare un indice secondario per ogni beacon prima di poter eseguire la ricerca nei campi crittografati. Per ulteriori informazioni, consulta Configurazione degli indici secondari con beacon.

I beacon creano una mappa tra il valore in chiaro scritto in un campo e il valore crittografato effettivamente memorizzato nel database. Non è possibile confrontare i valori di due beacon standard, anche se contengono lo stesso testo in chiaro sottostante. I due beacon standard produrranno due tag HMAC diversi per gli stessi valori di testo in chiaro. Di conseguenza, i beacon standard non possono eseguire le seguenti interrogazioni.

- *beacon1* = *beacon2*
- *beacon1* IN (*beacon2*)
- *value* IN (*beacon1*, *beacon2*, ...)
- CONTAINS(*beacon1*, *beacon2*)

È possibile eseguire le query precedenti solo se si confrontano le parti firmate dei beacon composti, ad eccezione dell'CONTAINSoperatore, che è possibile utilizzare con i beacon composti per identificare l'intero valore di un campo crittografato o firmato contenuto nel beacon assemblato. Quando si confrontano parti firmate, è possibile includere facoltativamente il prefisso di una parte crittografata, ma non il valore crittografato di un campo. Per ulteriori informazioni sui tipi di interrogazioni che i beacon standard e composti possono eseguire, vedete Interrogazione dei beacon.

Prendi in considerazione le seguenti soluzioni di crittografia ricercabili quando esamini i modelli di accesso al database. Gli esempi seguenti definiscono quale beacon configurare per soddisfare diversi requisiti di crittografia e interrogazione.

## Beacon standard

I beacon standard possono eseguire solo ricerche di uguaglianza. È possibile utilizzare i beacon standard per eseguire le seguenti interrogazioni.

### Interroga un singolo campo crittografato

Se desideri identificare i record che contengono un valore specifico per un campo crittografato, crea un beacon standard.

#### Esempi

Per l'esempio seguente, si consideri un database denominato `UnitInspection` che tiene traccia dei dati di ispezione per un impianto di produzione. Ogni record del database contiene campi denominati `work_id`inspection_date,`inspector_id_last4`, e`unit`. L'ID ispettore completo è un numero compreso tra 0 e 99.999.999. Tuttavia, per garantire che il set di dati sia distribuito uniformemente, memorizza `inspector_id_last4` solo le ultime quattro cifre dell'ID dell'ispettore. Ogni campo del database è identificato dalla chiave primaria. `work_id` I `unit` campi `inspector_id_last4` e sono contrassegnati `ENCRYPT_AND_SIGN` nelle azioni crittografiche.

Di seguito è riportato un esempio di voce in chiaro nel database. `UnitInspection`

```
{
    "work_id": "1c7fcff3-6e74-41a8-b7f7-925dc039830b",
    "inspection_date": 2023-06-07,
    "inspector_id_last4": 8744,
    "unit": 229304973450
}
```

#### Interroga un singolo campo crittografato in un record

> Se il `inspector_id_last4` campo deve essere crittografato, ma è comunque necessario interrogarlo per verificare le corrispondenze esatte, costruisci un beacon standard a partire dal campo. `inspector_id_last4` Quindi, utilizzate il beacon standard per creare un indice secondario. È possibile utilizzare questo indice secondario per eseguire query sul campo `inspector_id_last4` crittografato.

Per informazioni sulla configurazione dei beacon standard, vedere Configurazione dei beacon standard.

#### Interroga un campo virtuale

Un campo virtuale è un campo concettuale costruito da uno o più campi di origine. Se desideri eseguire ricerche di uguaglianza per un segmento specifico di un campo crittografato o eseguire

ricerche di uguaglianza sulla concatenazione di più campi, costruisci un beacon standard da un campo virtuale. Tutti i campi virtuali devono includere almeno un campo sorgente crittografato.

Esempi

Gli esempi seguenti creano campi virtuali per il `Employees` database. Di seguito è riportato un esempio di record di testo in chiaro nel `Employees` database.

```
{
    "EmployeeID": 101,
    "SSN": 000-00-0000,
    "LastName": "Jones",
    "FirstName": "Mary",
    "Address": {
                "Street": "123 Main",
                "City": "Anytown",
                "State": "OH",
                "ZIPCode": 12345
    }
}
```

Interroga un segmento di un campo crittografato

Per questo esempio, il `SSN` campo è crittografato.

Se desideri interrogare il `SSN` campo utilizzando le ultime quattro cifre di un numero di previdenza sociale, crea un campo virtuale che identifichi il segmento che intendi interrogare.

Un `Last4SSN` campo virtuale, costruito da, `Suffix(4)` consente di eseguire interrogazioni. `Last4SSN=0000` Usa questo campo virtuale per costruire un beacon standard. Quindi, usa il beacon standard per creare un indice secondario. È possibile utilizzare questo indice secondario per eseguire interrogazioni sul campo virtuale. Questa query restituisce tutti i record con un `SSN` valore che termina con le ultime quattro cifre specificate.

Interroga la concatenazione di più campi

> **Note**
>
> L'esempio seguente illustra i tipi di trasformazioni e interrogazioni che è possibile eseguire con un campo virtuale. In applicazione, i campi di esempio utilizzati in questo esempio

potrebbero non soddisfare le raccomandazioni sull'unicità della distribuzione e della correlazione per i beacon.

Se desideri eseguire ricerche di uguaglianza su una concatenazione di `LastName` campi `FirstName` and, puoi creare un `NameTag` campo virtuale, composto dalla prima lettera del campo, seguito dal `FirstName` campo, tutto in minuscolo. `LastName` Usa questo campo virtuale per costruire un beacon standard. Quindi, usa il beacon standard per creare un indice secondario. È possibile utilizzare questo indice secondario per eseguire interrogazioni `NameTag=mjones` sul campo virtuale.

Almeno uno dei campi di origine deve essere crittografato. `FirstName`O `LastName` potrebbero essere crittografati oppure potrebbero essere crittografati entrambi. Tutti i campi sorgente in testo semplice devono essere contrassegnati come `SIGN_ONLY` o `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` nelle azioni crittografiche.

Per informazioni sulla configurazione dei campi virtuali e dei beacon che li utilizzano, consulta Creazione di un campo virtuale.

## Beacon composti

I beacon composti creano un indice a partire da stringhe letterali in chiaro e beacon standard per eseguire operazioni complesse sul database. È possibile utilizzare i beacon composti per eseguire le seguenti interrogazioni.

Interroga una combinazione di campi crittografati su un singolo indice

Se devi interrogare una combinazione di campi crittografati su un singolo indice, crea un beacon composto che combini i singoli beacon standard costruiti per ogni campo crittografato per formare un unico indice.

Dopo aver configurato il beacon composto, è possibile creare un indice secondario che lo specifichi come chiave di partizione per eseguire query con corrispondenza esatta o con una chiave di ordinamento per eseguire query più complesse. Gli indici secondari che specificano il beacon composto come chiave di ordinamento possono eseguire query di corrispondenza esatta e query complesse più personalizzate.

Esempi

Per gli esempi seguenti, prendete in considerazione un database denominato `UnitInspection` che tiene traccia dei dati di ispezione per un impianto di produzione. Ogni record del database contiene campi denominati `work_id`inspection_date,`inspector_id_last4`, e`unit`. L'ID ispettore completo è un numero compreso tra 0 e 99.999.999. Tuttavia, per garantire che il set di dati sia distribuito uniformemente, memorizza `inspector_id_last4` solo le ultime quattro cifre dell'ID dell'ispettore. Ogni campo del database è identificato dalla chiave primaria. `work_id` I `unit` campi `inspector_id_last4` e sono contrassegnati `ENCRYPT_AND_SIGN` nelle azioni crittografiche.

Di seguito è riportato un esempio di voce in chiaro nel database. `UnitInspection`

```
{
    "work_id": "1c7fcff3-6e74-41a8-b7f7-925dc039830b",
    "inspection_date": 2023-06-07,
    "inspector_id_last4": 8744,
    "unit": 229304973450
}
```

Esegue ricerche di uguaglianza su una combinazione di campi crittografati

Se desideri interrogare il `UnitInspection` database per verificare le corrispondenze esatte`inspector_id_last4.unit`, crea innanzitutto beacon standard distinti per i `inspector_id_last4` campi and. `unit` Quindi, create un faro composto dai due beacon standard.

Dopo aver configurato il beacon composto, create un indice secondario che specifichi il beacon composto come chiave di partizione. Utilizzate questo indice secondario per ricercare le corrispondenze esatte su. `inspector_id_last4.unit` Ad esempio, è possibile interrogare questo faro per trovare un elenco di ispezioni eseguite da un ispettore per una determinata unità.

Esegui interrogazioni complesse su una combinazione di campi crittografati

Se desideri interrogare il `UnitInspection` database su `inspector_id_last4` and`inspector_id_last4.unit`, crea innanzitutto beacon standard distinti per i `inspector_id_last4` campi and. `unit` Quindi, create un beacon composto dai due beacon standard.

Dopo aver configurato il beacon composto, create un indice secondario che specifichi il beacon composto come chiave di ordinamento. Utilizzate questo indice secondario per interrogare il `UnitInspection` database alla ricerca di voci che iniziano con un determinato ispettore o

interrogare il database per ottenere un elenco di tutte le unità all'interno di uno specifico intervallo di ID di unità che sono state ispezionate da un determinato ispettore. Puoi anche eseguire ricerche con corrispondenze esatte su. `inspector_id_last4.unit`

Per informazioni sulla configurazione dei beacon composti, consulta Configurazione dei beacon composti.

Interroga una combinazione di campi crittografati e di testo semplice su un singolo indice

Se devi interrogare una combinazione di campi crittografati e di testo in chiaro su un singolo indice, crea un beacon composto che combini singoli beacon standard e campi di testo in chiaro per formare un unico indice. I campi di testo in chiaro utilizzati per costruire il beacon composto devono essere contrassegnati o nelle azioni crittografiche. `SIGN_ONLY`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT

Dopo aver configurato il beacon composto, è possibile creare un indice secondario che specifichi il beacon composto come chiave di partizione per eseguire query con corrispondenza esatta o con una chiave di ordinamento per eseguire query più complesse. Gli indici secondari che specificano il beacon composto come chiave di ordinamento possono eseguire query di corrispondenza esatta e query complesse più personalizzate.

Esempi

Per gli esempi seguenti, prendete in considerazione un database denominato `UnitInspection` che tiene traccia dei dati di ispezione per un impianto di produzione. Ogni record del database contiene campi denominati `work_id`inspection_date,`inspector_id_last4`, e`unit`. L'ID ispettore completo è un numero compreso tra 0 e 99.999.999. Tuttavia, per garantire che il set di dati sia distribuito uniformemente, memorizza `inspector_id_last4` solo le ultime quattro cifre dell'ID dell'ispettore. Ogni campo del database è identificato dalla chiave primaria. `work_id` I `unit` campi `inspector_id_last4` e sono contrassegnati `ENCRYPT_AND_SIGN` nelle azioni crittografiche.

Di seguito è riportato un esempio di voce in chiaro nel database. `UnitInspection`

```
{
    "work_id": "1c7fcff3-6e74-41a8-b7f7-925dc039830b",
    "inspection_date": 2023-06-07,
    "inspector_id_last4": 8744,
    "unit": 229304973450
}
```

Esegue ricerche di uguaglianza su una combinazione di campi

Se desideri interrogare il `UnitInspection` database per verificare le ispezioni condotte da un ispettore specifico in una data specifica, crea innanzitutto un beacon standard per il campo. `inspector_id_last4` Il `inspector_id_last4` campo è contrassegnato `ENCRYPT_AND_SIGN` nelle azioni crittografiche. Tutte le parti crittografate richiedono un proprio beacon standard. Il `inspection_date` campo è contrassegnato `SIGN_ONLY` e non richiede un beacon standard. Quindi, create un faro composto dal `inspection_date` campo e dal faro standard. `inspector_id_last4`

Dopo aver configurato il beacon composto, create un indice secondario che specifichi il beacon composto come chiave di partizione. Utilizzate questo indice secondario per interrogare i database alla ricerca di record con corrispondenze esatte con una determinata data di ispezione e ispezione. Ad esempio, è possibile interrogare il database per ottenere un elenco di tutte le ispezioni `8744` condotte dall'ispettore il cui ID termina in una data specifica.

Esegui interrogazioni complesse su una combinazione di campi

Se desideri interrogare il database per le ispezioni condotte all'interno di un `inspection_date` intervallo o interrogare il database per le ispezioni `inspection_date` condotte su un particolare vincolo `inspector_id_last4` o`inspector_id_last4.unit`, crea prima beacon standard distinti per i campi and. `inspector_id_last4` `unit` Quindi, create un beacon composto dal campo di testo in chiaro e dai due beacon standard. `inspection_date`

Dopo aver configurato il beacon composto, create un indice secondario che specifichi il beacon composto come chiave di ordinamento. Utilizzate questo indice secondario per eseguire interrogazioni relative alle ispezioni condotte in date specifiche da un ispettore specifico. Ad esempio, è possibile interrogare il database per ottenere un elenco di tutte le unità ispezionate nella stessa data. In alternativa, è possibile interrogare il database per ottenere un elenco di tutte le ispezioni eseguite su un'unità specifica tra un determinato intervallo di date di ispezione.

Per informazioni sulla configurazione dei beacon compositi, consulta Configurazione dei beacon compositi.

## Scelta della lunghezza del faro

La nostra libreria di crittografia lato client è stata rinominata Database Encryption SDK. AWS Questa guida per sviluppatori fornisce ancora informazioni sul DynamoDB Encryption Client.

Quando scrivi un nuovo valore in un campo crittografato configurato per la crittografia ricercabile, AWS Database Encryption SDK calcola un HMAC sul valore di testo in chiaro. Questo output HMAC corrisponde uno a uno (1:1) per il valore in chiaro di quel campo. L'output HMAC viene troncato in modo che più valori di testo in chiaro distinti vengano mappati allo stesso tag HMAC troncato. Queste collisioni, o falsi positivi, limitano la capacità di un utente non autorizzato di identificare informazioni distintive sul valore del testo in chiaro.

Il numero medio di falsi positivi generati per ogni beacon è determinato dalla lunghezza del beacon rimanente dopo il troncamento. È necessario definire la lunghezza del beacon solo quando si configurano i beacon standard. I beacon composti utilizzano le lunghezze dei beacon standard da cui sono costruiti.

Il beacon non altera lo stato crittografato del campo. Tuttavia, quando si utilizzano i beacon, esiste un compromesso intrinseco tra l'efficienza delle query e la quantità di informazioni rivelate sulla distribuzione dei dati.

L'obiettivo della crittografia ricercabile è ridurre i costi prestazionali associati ai database crittografati lato client utilizzando i beacon per eseguire query su dati crittografati. I beacon vengono archiviati insieme ai campi crittografati da cui vengono calcolati. Ciò significa che possono rivelare informazioni distintive sulla distribuzione del set di dati. In casi estremi, un utente non autorizzato potrebbe essere in grado di analizzare le informazioni rivelate sulla distribuzione e utilizzarle per identificare il valore di testo in chiaro di un campo. La scelta della giusta lunghezza del beacon può aiutare a mitigare questi rischi e preservare la riservatezza della distribuzione.

Esamina il tuo modello di minaccia per determinare il livello di sicurezza di cui hai bisogno. Ad esempio, maggiore è il numero di persone che hanno accesso al database, ma non dovrebbero avere accesso ai dati in chiaro, più potreste voler proteggere la riservatezza della distribuzione del set di dati. Per aumentare la riservatezza, un beacon deve generare più falsi positivi. Una maggiore riservatezza si traduce in una riduzione delle prestazioni delle query.

Sicurezza vs. prestazioni

- Una lunghezza del faro troppo lunga produce un numero insufficiente di falsi positivi e potrebbe rivelare informazioni distintive sulla distribuzione del set di dati.
- Una lunghezza del beacon troppo corta produce troppi falsi positivi e aumenta il costo delle prestazioni delle query perché richiede una scansione più ampia del database.

Nel determinare la lunghezza del beacon appropriata per la soluzione, è necessario trovare una lunghezza che preservi adeguatamente la sicurezza dei dati senza influire sulle prestazioni delle

query più del necessario. Il livello di sicurezza garantito da un beacon dipende dalla distribuzione del set di dati e dalla correlazione dei campi da cui sono costruiti i beacon. Gli argomenti seguenti presuppongono che i beacon siano distribuiti uniformemente e non contengano dati correlati.

Argomenti



## Calcolo della lunghezza del faro

La lunghezza del beacon è definita in bit e si riferisce al numero di bit del tag HMAC che vengono conservati dopo il troncamento. La lunghezza del beacon consigliata varia in base alla distribuzione del set di dati, alla presenza di valori correlati e ai requisiti specifici di sicurezza e prestazioni. Se il set di dati è distribuito in modo uniforme, è possibile utilizzare le seguenti equazioni e procedure per identificare la lunghezza del beacon ottimale per l'implementazione. Queste equazioni stimano solo il numero medio di falsi positivi che il beacon produrrà, ma non garantiscono che ogni valore univoco del set di dati produca un numero specifico di falsi positivi.

> **Note**
>
> L'efficacia di queste equazioni dipende dalla distribuzione del set di dati. Se il set di dati non è distribuito uniformemente, vedi. I beacon sono adatti al mio set di dati?
> In generale, più il set di dati è lontano da una distribuzione uniforme, più è necessario ridurre la lunghezza del beacon.

1.

   Stima la popolazione

   La popolazione è il numero previsto di valori univoci nel campo da cui è costruito il beacon standard, non è il numero totale previsto di valori memorizzati nel campo. Ad esempio, si consideri un `Room` campo crittografato che identifica la sede delle riunioni dei dipendenti. Il `Room` campo dovrebbe memorizzare 100.000 valori totali, ma ci sono solo 50 sale diverse che i dipendenti possono prenotare per le riunioni. Ciò significa che la popolazione è 50 perché nel `Room` campo possono essere memorizzati solo 50 valori univoci possibili.

> **Note**
>
> Se il beacon standard è costruito a partire da un campo virtuale, la popolazione utilizzata per calcolare la lunghezza del beacon è il numero di combinazioni uniche create dal campo virtuale.

Quando stimate la popolazione, assicuratevi di considerare la crescita prevista del set di dati. Dopo aver scritto nuovi record con il beacon, non è possibile aggiornare la lunghezza del beacon. Esamina il tuo modello di minaccia e tutte le soluzioni di database esistenti per creare una stima del numero di valori univoci che ti aspetti che questo campo memorizzi nei prossimi cinque anni.

Non è necessario che la tua popolazione sia precisa. Innanzitutto, identifica il numero di valori univoci nel tuo database corrente o stima il numero di valori univoci che prevedi di archiviare nel primo anno. Successivamente, utilizza le seguenti domande per determinare la crescita prevista di valori univoci nei prossimi cinque anni.

- Prevedi che i valori univoci si moltiplichino per 10?
- Ti aspetti che i valori univoci si moltiplichino per 100?
- Ti aspetti che i valori univoci si moltiplichino per 1000?

La differenza tra 50.000 e 60.000 valori univoci non è significativa ed entrambi daranno come risultato la stessa lunghezza del beacon consigliata. Tuttavia, la differenza tra 50.000 e 500.000 valori univoci influirà in modo significativo sulla lunghezza del beacon consigliata.

Prendi in considerazione la possibilità di esaminare i dati pubblici sulla frequenza dei tipi di dati più comuni, come codici postali o cognomi. Ad esempio, ci sono 41.707 codici postali negli Stati Uniti d'America. La popolazione che utilizzi deve essere proporzionale al tuo database. Se il `ZIPCode` campo del database include dati provenienti da tutti gli Stati Uniti d'America, è possibile definire la popolazione come 41.707, anche se al momento `ZIPCode` il campo non ha 41.707 valori univoci. Se il `ZIPCode` campo del database include solo dati di un singolo stato e includerà sempre e solo i dati di un singolo stato, puoi definire la tua popolazione come il numero totale di codici postali in quello stato anziché 41.704.

2. Calcola l'intervallo consigliato per il numero previsto di collisioni

Per determinare la lunghezza del faro appropriata per un determinato campo, è necessario innanzitutto identificare un intervallo appropriato per il numero previsto di collisioni. Il numero previsto di collisioni rappresenta il numero medio previsto di valori di testo in chiaro univoci mappati a un particolare tag HMAC. Il numero previsto di falsi positivi per un valore di testo in chiaro univoco è inferiore di uno rispetto al numero previsto di collisioni.

È consigliabile che il numero previsto di collisioni sia maggiore o uguale a due e inferiore alla radice quadrata della popolazione. Le seguenti equazioni funzionano solo se la popolazione ha 16 o più valori univoci.

```
2 ≤ number of collisions < √(Population)
```

Se il numero di collisioni è inferiore a due, il faro produrrà troppo pochi falsi positivi. Consigliamo due come numero minimo di collisioni previste perché significa che, in media, ogni valore univoco nel campo genererà almeno un falso positivo mappando un altro valore univoco.

3. Calcola l'intervallo consigliato per le lunghezze dei beacon

   Dopo aver identificato il numero minimo e massimo di collisioni previste, utilizzate la seguente equazione per identificare un intervallo di lunghezze di beacon appropriate.

   $$\text{number of collisions} = \text{Population} * 2^{-(\text{beacon length})}$$

   Innanzitutto, stabilite la lunghezza del faro in cui il numero di collisioni previste è uguale a due (il numero minimo consigliato di collisioni previste).

   $$2 = \text{Population} * 2^{-(\text{beacon length})}$$

   Quindi, stabilite la lunghezza del faro in modo che il numero previsto di collisioni sia uguale alla radice quadrata della popolazione (il numero massimo consigliato di collisioni previste).

   $$\sqrt{(\text{Population})} = \text{Population} * 2^{-(\text{beacon length})}$$

   Consigliamo di arrotondare l'output prodotto da questa equazione alla lunghezza del faro più corta. Ad esempio, se l'equazione produce una lunghezza del faro di 15,6, si consiglia di arrotondare tale valore per difetto a 15 bit anziché arrotondarlo a 16 bit.

4. Scegliete la lunghezza del faro

Queste equazioni identificano solo l'intervallo consigliato di lunghezze dei beacon per il tuo campo. Ti consigliamo di utilizzare un beacon di lunghezza inferiore per preservare la sicurezza del set di dati quando possibile. Tuttavia, la lunghezza del beacon effettivamente utilizzata è determinata dal modello di minaccia in uso. Considerate i vostri requisiti prestazionali quando esaminate il modello di minaccia per determinare la lunghezza del beacon migliore per il vostro campo.

L'utilizzo di un beacon di lunghezza inferiore riduce le prestazioni delle query, mentre l'utilizzo di un beacon di lunghezza maggiore riduce la sicurezza. In generale, se il set di dati è distribuito in modo non uniforme o se si creano beacon distinti a partire da campi correlati, è necessario utilizzare beacon di lunghezza inferiore per ridurre al minimo la quantità di informazioni rivelate sulla distribuzione dei set di dati.

Se esamini il tuo modello di minaccia e decidi che qualsiasi informazione distintiva rivelata sulla distribuzione di un campo non rappresenta una minaccia per la tua sicurezza generale, puoi scegliere di utilizzare una lunghezza del beacon superiore all'intervallo consigliato che hai calcolato. Ad esempio, se l'intervallo consigliato di lunghezze dei beacon per un campo è compreso tra 9 e 16 bit, è possibile scegliere di utilizzare una lunghezza del beacon di 24 bit per evitare perdite di prestazioni.

Scegliete con attenzione la lunghezza del vostro faro. Dopo aver scritto nuovi record con il beacon, non è possibile aggiornare la lunghezza del beacon.

## Esempio

Prendiamo in considerazione un database che contrassegna il `unit` campo come `ENCRYPT_AND_SIGN` nelle azioni crittografiche. Per configurare un beacon standard per il `unit` campo, dobbiamo determinare il numero previsto di falsi positivi e la lunghezza del beacon per il campo. `unit`

1. Stima la popolazione

   Dopo aver esaminato il nostro modello di minaccia e l'attuale soluzione di database, prevediamo che alla fine il `unit` campo avrà 100.000 valori univoci.

   Ciò significa che Popolazione = 100.000.
2. Calcola l'intervallo consigliato per il numero previsto di collisioni.

Per questo esempio, il numero previsto di collisioni deve essere compreso tra 2 e 316.

```
2 ≤ number of collisions < √(Population)
```

a.
```
2 ≤ number of collisions < √(100,000)
```

b.
```
2 ≤ number of collisions < 316
```

3. Calcola l'intervallo consigliato per la lunghezza del faro.

   Per questo esempio, la lunghezza del faro deve essere compresa tra 9 e 16 bit.

   $$\text{number of collisions} = \text{Population} * 2^{-(\text{beacon length})}$$

   a. Calcola la lunghezza del faro in cui il numero previsto di collisioni è uguale al minimo identificato nella Fase 2.

      $$2 = 100{,}000 * 2^{-(\text{beacon length})}$$

      Lunghezza del faro = 15,6 o 15 bit

   b. Calcola la lunghezza del faro in cui il numero previsto di collisioni è uguale al massimo identificato nella Fase 2.

      $$316 = 100{,}000 * 2^{-(\text{beacon length})}$$

      Lunghezza del faro = 8,3 o 8 bit

4. Determinate la lunghezza del beacon appropriata per i vostri requisiti di sicurezza e prestazioni.

   Per ogni bit inferiore a 15, il costo delle prestazioni e la sicurezza raddoppiano.

   - 16 bit
     - In media, ogni valore univoco verrà mappato a 1,5 altre unità.
     - Sicurezza: due record con lo stesso tag HMAC troncato hanno il 66% di probabilità che abbiano lo stesso valore di testo in chiaro.
     - Prestazioni: una query recupererà 15 record ogni 10 record effettivamente richiesti.
   - 14 bit

- In media, ogni valore univoco verrà mappato a 6,1 altre unità.
- Sicurezza: due record con lo stesso tag HMAC troncato hanno il 33% di probabilità che abbiano lo stesso valore di testo in chiaro.
- Prestazioni: una query recupererà 30 record ogni 10 record effettivamente richiesti.

## Scelta del nome del faro

> La nostra libreria di crittografia lato client è stata rinominata Database Encryption SDK. AWS Questa guida per sviluppatori fornisce ancora informazioni sul DynamoDB Encryption Client.

Ogni beacon è identificato da un nome univoco. Una volta configurato un beacon, il nome del beacon è il nome utilizzato quando si esegue una query su un campo crittografato. Il nome di un beacon può avere lo stesso nome di un campo crittografato o di un campo virtuale, ma non può avere lo stesso nome di un campo non crittografato. Due beacon diversi non possono avere lo stesso nome beacon.

Per esempi che dimostrano come denominare e configurare i beacon, vedere Configurazione dei beacon.

Denominazione dei beacon standard

Quando si assegnano nomi ai beacon standard, si consiglia vivamente di convertire il nome del beacon nella fonte del beacon, quando possibile. Ciò significa che il nome del beacon e il nome del campo criptato o virtuale da cui è costruito il beacon standard sono gli stessi. Ad esempio, se state creando un beacon standard per un campo criptato denominato`LastName`, dovrebbe esserlo anche il nome del beacon. `LastName`

Se il nome del beacon è lo stesso della fonte del beacon, puoi omettere la fonte del beacon dalla configurazione e AWS Database Encryption SDK utilizzerà automaticamente il nome del beacon come fonte del beacon.

# Configurazione dei beacon

> La nostra libreria di crittografia lato client è stata rinominata Database Encryption SDK. AWS Questa guida per sviluppatori fornisce ancora informazioni sul DynamoDB Encryption Client.

Esistono due tipi di beacon che supportano la crittografia ricercabile. I beacon standard eseguono ricerche di uguaglianza. Sono il modo più semplice per implementare la crittografia ricercabile nel database. I beacon composti combinano stringhe letterali in chiaro e beacon standard per eseguire query più complesse.

I beacon sono progettati per essere implementati in nuovi database non popolati. Qualsiasi beacon configurato in un database esistente mapperà solo i nuovi record scritti nel database. I beacon vengono calcolati in base al valore in chiaro di un campo, una volta crittografato il campo non è più possibile per il beacon di mappare i dati esistenti. Dopo aver scritto nuovi record con un beacon, non è possibile aggiornare la configurazione del beacon. Tuttavia, puoi aggiungere nuovi beacon per i nuovi campi che aggiungi al tuo record.

Dopo aver determinato i modelli di accesso, la configurazione dei beacon dovrebbe essere il secondo passaggio dell'implementazione del database. Quindi, dopo aver configurato tutti i beacon, è necessario creare un portachiavi AWS KMS gerarchico, definire la versione del beacon, configurare un indice secondario per ogni beacon, definire le azioni crittografiche e configurare il database e il client Database Encryption SDK. AWS Per ulteriori informazioni, consulta Utilizzo dei beacon.

Per semplificare la definizione della versione beacon, consigliamo di creare elenchi per beacon standard e composti. Aggiungi ogni beacon che crei al rispettivo elenco di beacon standard o compositi man mano che li configuri.

Argomenti

- Configurazione dei beacon standard
- Configurazione dei beacon composti
- Configurazioni di esempio

## Configurazione dei beacon standard

I beacon standard sono il modo più semplice per implementare la crittografia ricercabile nel database. Possono eseguire ricerche di uguaglianza solo per un singolo campo crittografato o virtuale.

### Esempio di sintassi di configurazione

Java

```
List<StandardBeacon> standardBeaconList = new ArrayList<>();
StandardBeacon exampleStandardBeacon = StandardBeacon.builder()
    .name("beaconName")
```

```
    .length(beaconLengthInBits)
    .build();
standardBeaconList.add(exampleStandardBeacon);
```

C# / .NET

```
var standardBeaconList = new List<StandardBeacon>();
StandardBeacon exampleStandardBeacon = new StandardBeacon
  {
    Name = "beaconName",
    Length = 10
  };
standardBeaconList.Add(exampleStandardBeacon);
```

Rust

```
let standard_beacon_list = vec![

 StandardBeacon::builder().name("beacon_name").length(beacon_length_in_bits).build()?,
```

Per configurare un beacon standard, fornite i seguenti valori.

Nome del beacon

Il nome che usi per interrogare un campo crittografato.

Il nome di un beacon può avere lo stesso nome di un campo crittografato o di un campo virtuale, ma non può avere lo stesso nome di un campo non crittografato. Se possibile, consigliamo vivamente di utilizzare il nome del campo crittografato o del campo virtuale da cui è costruito il beacon standard. Due beacon diversi non possono avere lo stesso nome di beacon. Per informazioni su come determinare il nome del beacon migliore per l'implementazione, consulta Scelta del nome del beacon.

Lunghezza del faro

Il numero di bit del valore hash del beacon che vengono conservati dopo il troncamento.

La lunghezza del faro determina il numero medio di falsi positivi prodotti da un determinato faro. Per ulteriori informazioni e assistenza nella determinazione della lunghezza del beacon appropriata per l'implementazione, vedere Determinazione della lunghezza del beacon.

Fonte del beacon (opzionale)

Il campo da cui è costruito un beacon standard.

La fonte del beacon deve essere un nome di campo o un indice che si riferisce al valore di un campo annidato. Quando il nome del beacon è lo stesso della fonte del beacon, puoi omettere la fonte del beacon dalla configurazione e AWS Database Encryption SDK utilizzerà automaticamente il nome del beacon come fonte del beacon.

## Creazione di un campo virtuale

Per creare un campo virtuale, è necessario fornire un nome per il campo virtuale e un elenco dei campi di origine. L'ordine in cui vengono aggiunti i campi di origine all'elenco delle parti virtuali determina l'ordine in cui vengono concatenati per creare il campo virtuale. L'esempio seguente concatena due campi di origine nella loro interezza per creare un campo virtuale.

> **Note**
>
> Ti consigliamo di verificare che i campi virtuali producano il risultato previsto prima di popolare il database. Per ulteriori informazioni, consulta Testing beacon outputs.

Java

Vedi l'esempio di codice completo: .java VirtualBeaconSearchableEncryptionExample

```
List<VirtualPart> virtualPartList = new ArrayList<>();
    virtualPartList.add(sourceField1);
    virtualPartList.add(sourceField2);

VirtualField virtualFieldName = VirtualField.builder()
    .name("virtualFieldName")
    .parts(virtualPartList)
    .build();

List<VirtualField> virtualFieldList = new ArrayList<>();
    virtualFieldList.add(virtualFieldName);
```

C# / .NET

Guarda l'esempio di codice completo: .cs VirtualBeaconSearchableEncryptionExample

```
var virtualPartList = new List<VirtualPart> { sourceField1, sourceField2 };

var virtualFieldName = new VirtualField
{
    Name = "virtualFieldName",
    Parts = virtualPartList
};

var virtualFieldList = new List<VirtualField> { virtualFieldName };
```

Rust

Guarda l'esempio di codice completo: virtual_beacon_searchable_encryption.rs

```
let virtual_part_list = vec![source_field_one, source_field_two];

let state_and_has_test_result_field = VirtualField::builder()
    .name("virtual_field_name")
    .parts(virtual_part_list)
    .build()?;

let virtual_field_list = vec![virtual_field_name];
```

Per creare un campo virtuale con un segmento specifico di un campo sorgente, è necessario definire tale trasformazione prima di aggiungere il campo di origine all'elenco delle parti virtuali.

Considerazioni sulla sicurezza per i campi virtuali

I beacon non alterano lo stato crittografato del campo. Tuttavia, quando si utilizzano i beacon, esiste un compromesso intrinseco tra l'efficienza delle query e la quantità di informazioni rivelate sulla distribuzione dei dati. Il modo in cui configuri il beacon determina il livello di sicurezza che viene mantenuto da quel beacon.

Evitate di creare un campo virtuale con campi sorgente che si sovrappongono ai beacon standard esistenti. La creazione di campi virtuali che includono un campo sorgente che è già stato utilizzato per creare un beacon standard può ridurre il livello di sicurezza di entrambi i beacon. La misura in cui la sicurezza viene ridotta dipende dal livello di entropia aggiunto dai campi sorgente aggiuntivi. Il livello di entropia è determinato dalla distribuzione di valori univoci nel campo sorgente aggiuntivo e dal numero di bit con cui il campo sorgente aggiuntivo contribuisce alla dimensione complessiva del campo virtuale.

È possibile utilizzare la popolazione e la lunghezza del beacon per determinare se i campi di origine di un campo virtuale preservano la sicurezza del set di dati. La popolazione è il numero previsto di valori univoci in un campo. Non è necessario che la tua popolazione sia precisa. Per informazioni sulla stima della popolazione di un campo, vedi Stima della popolazione.

Considera il seguente esempio quando esamini la sicurezza dei tuoi campi virtuali.

- Beacon1 è costruito da. `FieldA` `FieldA`ha una popolazione superiore a $2^{(\text{lunghezza Beacon1})}$.
- Beacon2 è costruito da`VirtualField`, che è costruito da,, e. `FieldA` `FieldB` `FieldC` `FieldD` Insieme, **`FieldBFieldC`**, e **`FieldD`** hanno una popolazione superiore a 2 N

Beacon2 preserva la sicurezza sia di Beacon1 che di Beacon2 se le seguenti affermazioni sono vere:

```
N ≥ (Beacon1 length)/2
```

e

```
N ≥ (Beacon2 length)/2
```

## Definizione degli stili dei beacon

I beacon standard possono essere utilizzati per eseguire ricerche di uguaglianza per un campo crittografato o virtuale. In alternativa, possono essere utilizzati per costruire beacon composti per eseguire operazioni di database più complesse. Per aiutarvi a organizzare e gestire i beacon standard, il AWS Database Encryption SDK fornisce i seguenti stili di beacon opzionali che definiscono l'uso previsto di un beacon standard.

> **Note**
>
> Per definire gli stili dei beacon, è necessario utilizzare la versione 3.2 o successiva di Database Encryption SDK. AWS Distribuisci la nuova versione a tutti i lettori prima di aggiungere stili beacon alle configurazioni dei beacon.

PartOnly

Un beacon standard definito come `PartOnly` può essere utilizzato solo per definire una parte crittografata di un beacon composto. Non è possibile interrogare direttamente un beacon standard. `PartOnly`

Java

```
List<StandardBeacon> standardBeaconList = new ArrayList<>();
StandardBeacon exampleStandardBeacon = StandardBeacon.builder()
    .name("beaconName")
    .length(beaconLengthInBits)
    .style(
        BeaconStyle.builder()
           .partOnly(PartOnly.builder().build())
        .build()
    )
    .build();
standardBeaconList.add(exampleStandardBeacon);
```

C# /.NET

```
new StandardBeacon
{
    Name = "beaconName",
    Length = beaconLengthInBits,
    Style = new BeaconStyle
    {
        PartOnly = new PartOnly()
    }
}
```

Rust

```
StandardBeacon::builder()
    .name("beacon_name")
    .length(beacon_length_in_bits)
    .style(BeaconStyle::PartOnly(PartOnly::builder().build()?))
    .build()?
```

Shared

Per impostazione predefinita, ogni beacon standard genera una chiave HMAC unica per il calcolo del beacon. Di conseguenza, non è possibile eseguire una ricerca di uguaglianza sui campi crittografati utilizzando due beacon standard separati. Un beacon standard definito come `Shared` utilizza la chiave HMAC di un altro beacon standard per i suoi calcoli.

Ad esempio, se devi confrontare `beacon1` campi con `beacon2` campi, definiscilo `beacon2` come un `Shared` beacon che utilizza la chiave HMAC di per i suoi calcoli. `beacon1`

> **Note**
>
> Considerate le vostre esigenze di sicurezza e prestazioni prima di configurare qualsiasi beacon. `Shared` `Shared`di beacon potrebbero aumentare la quantità di informazioni statistiche identificabili sulla distribuzione del set di dati. Ad esempio, potrebbero rivelare quali campi condivisi contengono lo stesso valore di testo in chiaro.

Java

```
List<StandardBeacon> standardBeaconList = new ArrayList<>();
StandardBeacon exampleStandardBeacon = StandardBeacon.builder()
    .name("beacon2")
    .length(beaconLengthInBits)
    .style(
        BeaconStyle.builder()
           .shared(Shared.builder().other("beacon1").build())
        .build()
    )
    .build();
standardBeaconList.add(exampleStandardBeacon);
```

C# /.NET

```
new StandardBeacon
{
    Name = "beacon2",
    Length = beaconLengthInBits,
    Style = new BeaconStyle
    {
        Shared = new Shared { Other = "beacon1" }
```

```
    }
}
```

Rust

```
StandardBeacon::builder()
    .name("beacon2")
    .length(beacon_length_in_bits)
    .style(BeaconStyle::Shared(
       Shared::builder().other("beacon1").build()?,
    ))
    .build()?
```

### AsSet

Per impostazione predefinita, se il valore di un campo è un set, AWS Database Encryption SDK calcola un singolo beacon standard per il set. Di conseguenza, non è possibile eseguire la query `CONTAINS(a, :value)` dove si *a* trova un campo crittografato. Un beacon standard definito come `AsSet` calcola i singoli valori del beacon standard per ogni singolo elemento del set e memorizza il valore del beacon nell'elemento come set. Ciò consente al AWS Database Encryption SDK di eseguire la query. `CONTAINS(a, :value)`

Per definire un beacon `AsSet` standard, gli elementi del set devono appartenere alla stessa popolazione in modo che possano utilizzare tutti la stessa lunghezza del beacon. Il set beacon potrebbe contenere un numero inferiore di elementi rispetto al set di testo in chiaro in caso di collisioni durante il calcolo dei valori del beacon.

> **Note**
>
> Considerate le vostre esigenze di sicurezza e prestazioni prima di configurare qualsiasi beacon. `AsSet` `AsSeti` beacon potrebbero aumentare la quantità di informazioni statistiche identificabili sulla distribuzione del set di dati. Ad esempio, potrebbero rivelare la dimensione del set di testo in chiaro.

Java

```
List<StandardBeacon> standardBeaconList = new ArrayList<>();
StandardBeacon exampleStandardBeacon = StandardBeacon.builder()
```

```
    .name("beaconName")
    .length(beaconLengthInBits)
    .style(
        BeaconStyle.builder()
           .asSet(AsSet.builder().build())
        .build()
    )
    .build();
standardBeaconList.add(exampleStandardBeacon);
```

C# /.NET

```
new StandardBeacon
{
    Name = "beaconName",
    Length = beaconLengthInBits,
    Style = new BeaconStyle
    {
        AsSet = new AsSet()
    }
}
```

Rust

```
StandardBeacon::builder()
    .name("beacon_name")
    .length(beacon_length_in_bits)
    .style(BeaconStyle::AsSet(AsSet::builder().build()?))
    .build()?
```

### SharedSet

Un beacon standard definito come `SharedSet` combina le `AsSet` funzioni `Shared` and in modo da poter eseguire ricerche di uguaglianza sui valori crittografati di un set e di un campo. Ciò consente al AWS Database Encryption SDK di eseguire la query `CONTAINS(a, b)` in cui si *a* trova un set crittografato e *b* un campo crittografato.

> **Note**
>
> Considerate le vostre esigenze di sicurezza e prestazioni prima di configurare qualsiasi `Shared` beacon. `SharedSeti` beacon potrebbero aumentare la quantità di informazioni

statistiche identificabili sulla distribuzione del set di dati. Ad esempio, potrebbero rivelare la dimensione del set di testo in chiaro o quali campi condivisi contengono lo stesso valore di testo in chiaro.

Java

```
List<StandardBeacon> standardBeaconList = new ArrayList<>();
StandardBeacon exampleStandardBeacon = StandardBeacon.builder()
    .name("beacon2")
    .length(beaconLengthInBits)
    .style(
        BeaconStyle.builder()
           .sharedSet(SharedSet.builder().other("beacon1").build())
        .build()
    )
    .build();
standardBeaconList.add(exampleStandardBeacon);
```

C# /.NET

```
new StandardBeacon
{
    Name = "beacon2",
    Length = beaconLengthInBits,
    Style = new BeaconStyle
    {
        SharedSet = new SharedSet { Other = "beacon1" }
    }
}
```

Rust

```
StandardBeacon::builder()
    .name("beacon2")
    .length(beacon_length_in_bits)
    .style(BeaconStyle::SharedSet(
        SharedSet::builder().other("beacon1").build()?,
    ))
    .build()?
```

# Configurazione dei beacon composti

I beacon composti combinano stringhe letterali in chiaro e beacon standard per eseguire operazioni complesse sul database, come interrogare due diversi tipi di record da un singolo indice o interrogare una combinazione di campi con una chiave di ordinamento. I beacon composti possono essere costruiti da, e campi. `ENCRYPT_AND_SIGN` `SIGN_ONLY` `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` È necessario creare un beacon standard per ogni campo crittografato incluso nel beacon composto.

> **Note**
>
> Ti consigliamo di verificare che i beacon composti producano il risultato previsto prima di popolare il database. Per ulteriori informazioni, consulta Testing beacon outputs.

## Esempio di sintassi di configurazione

Java

Configurazione del beacon composto

L'esempio seguente definisce gli elenchi di parti crittografati e firmati localmente all'interno della configurazione del beacon composto.

```
List<CompoundBeacon> compoundBeaconList = new ArrayList<>();
CompoundBeacon exampleCompoundBeacon = CompoundBeacon.builder()
    .name("compoundBeaconName")
    .split(".")
    .encrypted(encryptedPartList)
    .signed(signedPartList)
    .constructors(constructorList)
    .build();
compoundBeaconList.add(exampleCompoundBeacon);
```

Definizione della versione del beacon

L'esempio seguente definisce gli elenchi di parti crittografati e firmati a livello globale nella versione beacon. Per ulteriori informazioni sulla definizione della versione beacon, vedete Uso dei beacon.

```
List<BeaconVersion> beaconVersions = new ArrayList<>();
```

```
beaconVersions.add(
    BeaconVersion.builder()
        .standardBeacons(standardBeaconList)
        .compoundBeacons(compoundBeaconList)
        .encryptedParts(encryptedPartList)
        .signedParts(signedPartList)
        .version(1) // MUST be 1
        .keyStore(keyStore)
        .keySource(BeaconKeySource.builder()
            .single(SingleKeyStore.builder()
                .keyId(branchKeyId)
                .cacheTTL(6000)
                .build())
            .build())
        .build()
);
```

C# / .NET

Guarda l'esempio di codice completo: .cs BeaconConfig

Configurazione del beacon composto

L'esempio seguente definisce gli elenchi di parti crittografati e firmati localmente all'interno della configurazione del beacon composto.

```
var compoundBeaconList = new List<CompoundBeacon>();
var exampleCompoundBeacon = new CompoundBeacon
 {
    Name = "compoundBeaconName",
    Split = ".",
    Encrypted = encryptedPartList,
    Signed = signedPartList,
    Constructors = constructorList
 };
compoundBeaconList.Add(exampleCompoundBeacon);
```

Definizione della versione del beacon

L'esempio seguente definisce gli elenchi di parti crittografati e firmati a livello globale nella versione beacon. Per ulteriori informazioni sulla definizione della versione beacon, vedete Uso dei beacon.

```
var beaconVersions = new List<BeaconVersion>
{
    new BeaconVersion
    {
        StandardBeacons = standardBeaconList,
        CompoundBeacons = compoundBeaconList,
        EncryptedParts = encryptedPartsList,
        SignedParts = signedPartsList,
        Version = 1, // MUST be 1
        KeyStore = keyStore,
        KeySource = new BeaconKeySource
        {
            Single = new SingleKeyStore
            {
                KeyId = branchKeyId,
                CacheTTL = 6000
            }
        }
    }
};
```

Rust

Guardate l'esempio completo di codice: beacon_config.rs

Configurazione del beacon composto

L'esempio seguente definisce gli elenchi di parti crittografati e firmati localmente all'interno della configurazione del beacon composto.

```
let compound_beacon_list = vec![
    CompoundBeacon::builder()
        .name("compound_beacon_name")
        .split(".")
        .encrypted(encrypted_parts_list)
        .signed(signed_parts_list)
        .constructors(constructor_list)
        .build()?
```

Definizione della versione del beacon

L'esempio seguente definisce gli elenchi di parti crittografati e firmati a livello globale nella versione beacon. Per ulteriori informazioni sulla definizione della versione beacon, vedete Uso dei beacon.

```
let beacon_versions = BeaconVersion::builder()
    .standard_beacons(standard_beacon_list)
    .compound_beacons(compound_beacon_list)
    .encrypted_parts(encrypted_parts_list)
    .signed_parts(signed_parts_list)
    .version(1) // MUST be 1
    .key_store(key_store.clone())
    .key_source(BeaconKeySource::Single(
        SingleKeyStore::builder()
            .key_id(branch_key_id)
            .cache_ttl(6000)
            .build()?,
    ))
    .build()?;
let beacon_versions = vec![beacon_versions];
```

È possibile definire le parti crittografate e le parti firmate in elenchi definiti localmente o globalmente. Ti consigliamo di definire le parti crittografate e firmate in un elenco globale nella versione beacon, quando possibile. Definendo le parti crittografate e firmate a livello globale, è possibile definire ogni parte una volta e quindi riutilizzarle in più configurazioni beacon composte. Se intendete utilizzare una parte crittografata o firmata una sola volta, potete definirla in un elenco locale nella configurazione del beacon composto. È possibile fare riferimento sia alle parti locali che a quelle globali nell'elenco dei costruttori.

Se definite gli elenchi di parti crittografati e firmati a livello globale, dovete fornire un elenco di parti del costruttore che identifichi tutti i modi possibili in cui il beacon composto può assemblare i campi nella configurazione del beacon composto.

> **Note**
>
> Per definire elenchi di parti crittografati e firmati a livello globale, è necessario utilizzare la versione 3.2 o successiva di Database Encryption SDK. AWS Distribuisci la nuova versione a tutti i lettori prima di definire nuove parti a livello globale.
> Non è possibile aggiornare le configurazioni dei beacon esistenti per definire elenchi di parti crittografati e firmati a livello globale.

Per configurare un beacon composto, fornite i seguenti valori.

Nome del beacon

Il nome che usi per interrogare un campo crittografato.

Il nome di un beacon può avere lo stesso nome di un campo crittografato o di un campo virtuale, ma non può avere lo stesso nome di un campo non crittografato. Due beacon non possono avere lo stesso nome di beacon. Per informazioni su come determinare il nome del beacon migliore per l'implementazione, consulta Scelta del nome del beacon.

Carattere diviso

Il personaggio usato per separare le parti che compongono il faro composto.

Il carattere diviso non può apparire nei valori in chiaro di nessuno dei campi da cui è costruito il beacon composto.

Elenco delle parti crittografato

Identifica i `ENCRYPT_AND_SIGN` campi inclusi nel beacon composto.

Ogni parte deve includere un nome e un prefisso. Il nome della parte deve essere il nome del beacon standard costruito a partire dal campo crittografato. Il prefisso può essere qualsiasi stringa, ma deve essere univoco. Una parte crittografata non può avere lo stesso prefisso di una parte firmata. Si consiglia di utilizzare un valore breve che distingua la parte dalle altre parti servite dal beacon composto.

Ti consigliamo di definire le parti crittografate a livello globale quando possibile. Potresti prendere in considerazione la definizione locale di una parte crittografata se intendi utilizzarla solo in un beacon composto. Una parte crittografata definita localmente non può avere lo stesso prefisso o nome di una parte crittografata definita a livello globale.

Java

```
List<EncryptedPart> encryptedPartList = new ArrayList<>);
EncryptedPart encryptedPartExample = EncryptedPart.builder()
    .name("standardBeaconName")
    .prefix("E-")
    .build();
encryptedPartList.add(encryptedPartExample);
```

C# / .NET

```
var encryptedPartList = new List<EncryptedPart>();
var encryptedPartExample = new EncryptedPart
 {
    Name = "compoundBeaconName",
    Prefix = "E-"
 };
encryptedPartList.Add(encryptedPartExample);
```

Rust

```
let encrypted_parts_list = vec![
    EncryptedPart::builder()
        .name("standard_beacon_name")
        .prefix("E-")
        .build()?
];
```

Elenco delle parti firmate

Identifica i campi firmati inclusi nel beacon composto.

> **Note**
>
> Le parti firmate sono opzionali. È possibile configurare un beacon composto che non faccia riferimento a parti firmate.

Ogni parte deve includere un nome, una fonte e un prefisso. L'origine è il `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` campo `SIGN_ONLY` o il campo identificato dalla parte. L'origine deve essere un nome di campo o un indice che si riferisce al valore di un campo annidato. Se il nome della parte identifica la fonte, puoi omettere la fonte e AWS Database Encryption SDK utilizzerà automaticamente il nome come fonte. Ti consigliamo di specificare l'origine come nome della parte quando possibile. Il prefisso può essere qualsiasi stringa, ma deve essere univoco. Una parte firmata non può avere lo stesso prefisso di una parte crittografata. Si consiglia di utilizzare un valore breve che distingua la parte dalle altre parti servite dal beacon composto.

Ti consigliamo di definire le parti firmate a livello globale quando possibile. Potresti prendere in considerazione la definizione locale di una parte firmata se intendi utilizzarla solo in un beacon composto. Una parte firmata definita localmente non può avere lo stesso prefisso o nome di una parte firmata definita a livello globale.

Java

```
List<SignedPart> signedPartList = new ArrayList<>);
SignedPart signedPartExample = SignedPart.builder()
    .name("signedFieldName")
    .prefix("S-")
    .build();
signedPartList.add(signedPartExample);
```

C# / .NET

```
var signedPartsList = new List<SignedPart>
{
    new SignedPart { Name = "signedFieldName1", Prefix = "S-" },
    new SignedPart { Name = "signedFieldName2", Prefix = "SF-" }
};
```

Rust

```
let signed_parts_list = vec![
    SignedPart::builder()
        .name("signed_field_name_1")
        .prefix("S-")
        .build()?,
   SignedPart::builder()
        .name("signed_field_name_2")
        .prefix("SF-")
        .build()?,
];
```

Elenco dei costruttori

Identifica i costruttori che definiscono i diversi modi in cui le parti crittografate e firmate possono essere assemblate dal beacon composto. È possibile fare riferimento sia alle parti locali che a quelle globali nell'elenco dei costruttori.

Se costruite il vostro beacon composto da parti crittografate e firmate definite a livello globale, dovete fornire un elenco di costruttori.

Se non utilizzate parti crittografate o firmate definite a livello globale per costruire il beacon composto, l'elenco dei costruttori è facoltativo. Se non specificate un elenco di costruttori, AWS Database Encryption SDK assembla il beacon composto con il seguente costruttore predefinito.

- Tutte le parti firmate nell'ordine in cui sono state aggiunte all'elenco delle parti firmate
- Tutte le parti crittografate nell'ordine in cui sono state aggiunte all'elenco delle parti crittografate
- Tutte le parti sono obbligatorie

Costruttori

Ogni costruttore è un elenco ordinato di parti del costruttore che definisce un modo in cui il faro composto può essere assemblato. Le parti del costruttore vengono unite nell'ordine in cui vengono aggiunte all'elenco, con ogni parte separata dal carattere di divisione specificato.

Ogni parte del costruttore nomina una parte crittografata o firmata e definisce se tale parte è obbligatoria o facoltativa all'interno del costruttore. Ad esempio, se si desidera interrogare un beacon composto su`Field1`, e `Field1.Field2``Field1.Field2.Field3`, contrassegnarlo `Field3` come facoltativo `Field2` e creare un costruttore.

Ogni costruttore deve avere almeno una parte obbligatoria. Si consiglia di rendere obbligatoria la prima parte di ogni costruttore in modo da poter utilizzare l'`BEGINS_WITH`operatore nelle query.

Un costruttore ha successo se tutte le parti necessarie sono presenti nel record. Quando si scrive un nuovo record, il beacon composto utilizza l'elenco dei costruttori per determinare se il beacon può essere assemblato in base ai valori forniti. Tenta di assemblare il beacon nell'ordine in cui i costruttori sono stati aggiunti all'elenco dei costruttori e utilizza il primo costruttore che riesce. Se nessun costruttore ha successo, il beacon non viene scritto nel record.

Tutti i lettori e gli scrittori devono specificare lo stesso ordine di costruttori per garantire che i risultati delle query siano corretti.

Utilizzate le seguenti procedure per specificare il vostro elenco di costruttori.

1. Create una parte costruttore per ogni parte crittografata e parte firmata per definire se quella parte è necessaria o meno.

Il nome della parte del costruttore deve essere il nome del beacon standard o del campo firmato che rappresenta.

Java

```
ConstructorPart field1ConstructorPart = ConstructorPart.builder()
        .name("Field1")
        .required(true)
        .build();
```

C# / .NET

```
var field1ConstructorPart = new ConstructorPart { Name = "Field1", Required
 = true };
```

Rust

```
let field_1_constructor_part = ConstructorPart::builder()
    .name("field_1")
    .required(true)
    .build()?;
```

2. Create un costruttore per ogni possibile modo in cui il beacon composto può essere assemblato utilizzando le parti del costruttore create nel passaggio 1.

   Ad esempio, se si desidera eseguire un'interrogazione su `Field1.Field2.Field3` and`Field4.Field2.Field3`, è necessario creare due costruttori. `Field1`e `Field4` possono essere entrambi obbligatori perché sono definiti in due costruttori separati.

   Java

```
// Create a list for Field1.Field2.Field3 queries
List<ConstructorPart> field123ConstructorPartList = new ArrayList<>();
field123ConstructorPartList.add(field1ConstructorPart);
field123ConstructorPartList.add(field2ConstructorPart);
field123ConstructorPartList.add(field3ConstructorPart);
Constructor field123Constructor = Constructor.builder()
        .parts(field123ConstructorPartList)
        .build();
// Create a list for Field4.Field2.Field1 queries
List<ConstructorPart> field421ConstructorPartList = new ArrayList<>();
```

```
field421ConstructorPartList.add(field4ConstructorPart);
field421ConstructorPartList.add(field2ConstructorPart);
field421ConstructorPartList.add(field1ConstructorPart);
Constructor field421Constructor = Constructor.builder()
        .parts(field421ConstructorPartList)
        .build();
```

C# / .NET

```
// Create a list for Field1.Field2.Field3 queries
 var field123ConstructorPartList = new Constructor
{
    Parts = new List<ConstructorPart> { field1ConstructorPart,
 field2ConstructorPart, field3ConstructorPart }
};
// Create a list for Field4.Field2.Field1 queries
var field421ConstructorPartList = new Constructor
{
    Parts = new List<ConstructorPart> { field4ConstructorPart,
 field2ConstructorPart, field1ConstructorPart }
};
```

Rust

```
// Create a list for field1.field2.field3 queries
let field1_field2_field3_constructor = Constructor::builder()
    .parts(vec![
        field1_constructor_part,
        field2_constroctor_part.clone(),
        field3_constructor_part,
    ])
    .build()?;

// Create a list for field4.field2.field1 queries
let field4_field2_field1_constructor = Constructor::builder()
    .parts(vec![
        field4_constructor_part,
        field2_constroctor_part.clone(),
        field1_constructor_part,
    ])
    .build()?;
```

3. Create un elenco di costruttori che includa tutti i costruttori creati nel passaggio 2.

Java

```
List<Constructor> constructorList = new ArrayList<>();
constructorList.add(field123Constructor)
constructorList.add(field421Constructor)
```

C# / .NET

```
var constructorList = new List<Constructor>
{
    field123Constructor,
    field421Constructor
};
```

Rust

```
let constructor_list = vec![
    field1_field2_field3_constructor,
    field4_field2_field1_constructor,
];
```

4. Specificate `constructorList` quando create il beacon composto.

## Configurazioni di esempio

La nostra libreria di crittografia lato client è stata rinominata Database Encryption SDK. AWS Questa guida per sviluppatori fornisce ancora informazioni sul DynamoDB Encryption Client.

Gli esempi seguenti mostrano come configurare beacon standard e composti. Le seguenti configurazioni non forniscono le lunghezze dei beacon. Per informazioni sulla determinazione della lunghezza del beacon appropriata per la configurazione, consulta Scelta della lunghezza del beacon.

Per vedere esempi di codice completi che dimostrano come configurare e utilizzare i beacon, consulta gli esempi di crittografia ricercabile in Java, .NET e Rust nel repository -dynamodb on. aws-database-encryption-sdk GitHub

Argomenti

- Beacon standard

- Fari composti

## Beacon standard

Se desideri interrogare il `inspector_id_last4` campo per verificare le corrispondenze esatte, crea un beacon standard utilizzando la seguente configurazione.

Java

```
List<StandardBeacon> standardBeaconList = new ArrayList<>();
StandardBeacon exampleStandardBeacon = StandardBeacon.builder()
    .name("inspector_id_last4")
    .length(beaconLengthInBits)
    .build();
standardBeaconList.add(exampleStandardBeacon);
```

C# / .NET

```
var standardBeaconList = new List<StandardBeacon>>);
StandardBeacon exampleStandardBeacon = new StandardBeacon
  {
    Name = "inspector_id_last4",
    Length = 10
  };
standardBeaconList.Add(exampleStandardBeacon);
```

Rust

```
let last4_beacon = StandardBeacon::builder()
    .name("inspector_id_last4")
    .length(10)
    .build()?;

let unit_beacon = StandardBeacon::builder().name("unit").length(30).build()?;

let standard_beacon_list = vec![last4_beacon, unit_beacon];
```

## Fari composti

Se vuoi interrogare il `UnitInspection` database su `inspector_id_last4` and`inspector_id_last4.unit`, crea un beacon composto con la seguente configurazione. Questo beacon composto richiede solo parti criptate.

Java

```
// 1. Create standard beacons for the inspector_id_last4 and unit fields.
List<StandardBeacon> standardBeaconList = new ArrayList<>);
StandardBeacon inspectorBeacon = StandardBeacon.builder()
    .name("inspector_id_last4")
    .length(beaconLengthInBits)
    .build();
standardBeaconList.add(inspectorBeacon);

StandardBeacon unitBeacon = StandardBeacon.builder()
    .name("unit")
    .length(beaconLengthInBits)
    .build();
standardBeaconList.add(unitBeacon);

// 2. Define the encrypted parts.
List<EncryptedPart> encryptedPartList = new ArrayList<>);

// Each encrypted part needs a name and prefix
// The name must be the name of the standard beacon
// The prefix must be unique
// For this example we use the prefix "I-" for "inspector_id_last4"
// and "U-" for "unit"
EncryptedPart encryptedPartInspector = EncryptedPart.builder()
    .name("inspector_id_last4")
    .prefix("I-")
    .build();
encryptedPartList.add(encryptedPartInspector);

EncryptedPart encryptedPartUnit = EncryptedPart.builder()
    .name("unit")
    .prefix("U-")
    .build();
encryptedPartList.add(encryptedPartUnit);

// 3. Create the compound beacon.
```

```
// This compound beacon only requires a name, split character,
// and list of encrypted parts
CompoundBeacon inspectorUnitBeacon = CompoundBeacon.builder()
    .name("inspectorUnitBeacon")
    .split(".")
    .sensitive(encryptedPartList)
    .build();
```

### C# / .NET

```
// 1. Create standard beacons for the inspector_id_last4 and unit fields.
StandardBeacon inspectorBeacon = new StandardBeacon
 {
   Name = "inspector_id_last4",
   Length = 10
 };
standardBeaconList.Add(inspectorBeacon);
StandardBeacon unitBeacon = new StandardBeacon
 {
    Name = "unit",
    Length = 30
 };
standardBeaconList.Add(unitBeacon);

// 2. Define the encrypted parts.
var last4EncryptedPart = new EncryptedPart

// Each encrypted part needs a name and prefix
// The name must be the name of the standard beacon
// The prefix must be unique
// For this example we use the prefix "I-" for "inspector_id_last4"
// and "U-" for "unit"
var last4EncryptedPart = new EncryptedPart
 {
   Name = "inspector_id_last4",
   Prefix = "I-"
 };
encryptedPartList.Add(last4EncryptedPart);

var unitEncryptedPart = new EncryptedPart
 {
   Name = "unit",
   Prefix = "U-"
```

```
 };
encryptedPartList.Add(unitEncryptedPart);

// 3. Create the compound beacon.
// This compound beacon only requires a name, split character,
// and list of encrypted parts
var compoundBeaconList = new List<CompoundBeacon>>);
var inspectorCompoundBeacon = new CompoundBeacon
  {
      Name = "inspector_id_last4",
      Split = ".",
      Encrypted = encryptedPartList
  };
compoundBeaconList.Add(inspectorCompoundBeacon);
```

### Rust

```
// 1. Create standard beacons for the inspector_id_last4 and unit fields.
let last4_beacon = StandardBeacon::builder()
    .name("inspector_id_last4")
    .length(10)
    .build()?;

let unit_beacon = StandardBeacon::builder().name("unit").length(30).build()?;

let standard_beacon_list = vec![last4_beacon, unit_beacon];

// 2. Define the encrypted parts.
// The name must be the name of the standard beacon
// The prefix must be unique
// For this example we use the prefix "I-" for "inspector_id_last4"
// and "U-" for "unit"
let encrypted_parts_list = vec![
    EncryptedPart::builder()
        .name("inspector_id_last4")
        .prefix("I-")
        .build()?,
    EncryptedPart::builder().name("unit").prefix("U-").build()?,
];

// 3. Create the compound beacon
// This compound beacon only requires a name, split character,
// and list of encrypted parts
```

```
let compound_beacon_list = vec![CompoundBeacon::builder()
    .name("last4UnitCompound")
    .split(".")
    .encrypted(encrypted_parts_list)
    .build()?];
```

# Utilizzo dei beacon

La nostra libreria di crittografia lato client è stata rinominata Database Encryption SDK. AWS Questa guida per sviluppatori fornisce ancora informazioni sul DynamoDB Encryption Client.

I beacon consentono di cercare record crittografati senza decrittografare l'intero database su cui viene eseguita la query. I beacon sono progettati per essere implementati in nuovi database non popolati. Qualsiasi beacon configurato in un database esistente mapperà solo i nuovi record scritti nel database. I beacon vengono calcolati in base al valore in chiaro di un campo, una volta crittografato il campo non è più possibile per il beacon di mappare i dati esistenti. Dopo aver scritto nuovi record con un beacon, non è possibile aggiornare la configurazione del beacon. Tuttavia, puoi aggiungere nuovi beacon per i nuovi campi che aggiungi al tuo record.

Dopo aver configurato i beacon, è necessario completare i seguenti passaggi prima di iniziare a popolare il database ed eseguire query sui beacon.

1. AWS KMS Crea un portachiavi gerarchico

   Per utilizzare la crittografia ricercabile, è necessario utilizzare il portachiavi AWS KMS gerarchico per generare, crittografare e decrittografare le chiavi dati utilizzate per proteggere i record.

   Dopo aver configurato i beacon, assemblate i prerequisiti del portachiavi gerarchico e create il portachiavi gerarchico.

   Per maggiori dettagli sul motivo per cui è necessario il portachiavi gerarchico, consulta Utilizzo del portachiavi gerarchico per la crittografia ricercabile.

2. 

   Definisci la versione del beacon

Specificate il vostro `keyStorekeySource`, un elenco di tutti i beacon standard che avete configurato, un elenco di tutti i beacon composti che avete configurato, un elenco di parti crittografate, un elenco di parti firmate e una versione del beacon. È necessario specificare la versione beacon. 1 Per indicazioni sulla definizione del tuo`keySource`, vedi. Definizione della fonte della chiave del beacon

Il seguente esempio di Java definisce la versione beacon per un database a tenant singolo. Per informazioni sulla definizione della versione beacon per un database multitenant, vedere Crittografia ricercabile per database multitenant.

Java

```
 List<BeaconVersion> beaconVersions = new ArrayList<>();
beaconVersions.add(
    BeaconVersion.builder()
        .standardBeacons(standardBeaconList)
        .compoundBeacons(compoundBeaconList)
        .encryptedParts(encryptedPartsList)
        .signedParts(signedPartsList)
        .version(1) // MUST be 1
        .keyStore(keyStore)
        .keySource(BeaconKeySource.builder()
            .single(SingleKeyStore.builder()
                .keyId(branchKeyId)
                .cacheTTL(6000)
                .build())
            .build())
        .build()
);
```

C# / .NET

```
var beaconVersions = new List<BeaconVersion>
{
    new BeaconVersion
    {
        StandardBeacons = standardBeaconList,
        CompoundBeacons = compoundBeaconList,
        EncryptedParts = encryptedPartsList,
        SignedParts = signedPartsList,
        Version = 1, // MUST be 1
```

```
        KeyStore = branchKeyStoreName,
        KeySource = new BeaconKeySource
        {
            Single = new SingleKeyStore
            {
                KeyId = branch-key-id,
                CacheTTL = 6000
            }
        }
    }
};
```

Rust

```
let beacon_version = BeaconVersion::builder()
    .standard_beacons(standard_beacon_list)
    .compound_beacons(compound_beacon_list)
    .version(1) // MUST be 1
    .key_store(key_store.clone())
    .key_source(BeaconKeySource::Single(
        SingleKeyStore::builder()
            // `keyId` references a beacon key.
            // For every branch key we create in the keystore,
            // we also create a beacon key.
            // This beacon key is not the same as the branch key,
            // but is created with the same ID as the branch key.
            .key_id(branch_key_id)
            .cache_ttl(6000)
            .build()?,
    ))
    .build()?;
let beacon_versions = vec![beacon_version];
```

3. Configura gli indici secondari

   Dopo aver configurato i beacon, è necessario configurare un indice secondario che rifletta ogni beacon prima di poter eseguire la ricerca nei campi crittografati. Per ulteriori informazioni, consulta Configurazione degli indici secondari con beacon.

4. Definisci le tue azioni crittografiche

Tutti i campi utilizzati per costruire un beacon standard devono essere contrassegnati. `ENCRYPT_AND_SIGN` Tutti gli altri campi utilizzati per costruire i beacon devono essere contrassegnati o. `SIGN_ONLY` `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`

5. Configurare un client AWS Database Encryption SDK

   Per configurare un client AWS Database Encryption SDK che protegge gli elementi della tabella DynamoDB, consulta Libreria di crittografia lato client Java per DynamoDB.

## Interrogazione dei beacon

Il tipo di beacon configurato determina il tipo di interrogazioni che è possibile eseguire. I beacon standard utilizzano espressioni di filtro per eseguire ricerche di uguaglianza. I beacon composti combinano stringhe letterali in chiaro e beacon standard per eseguire query complesse. Quando si interrogano dati crittografati, si esegue la ricerca in base al nome del beacon.

Non è possibile confrontare i valori di due beacon standard, anche se contengono lo stesso testo in chiaro sottostante. I due beacon standard produrranno due tag HMAC diversi per gli stessi valori di testo in chiaro. Di conseguenza, i beacon standard non possono eseguire le seguenti interrogazioni.

- *beacon1* = *beacon2*
- *beacon1* IN (*beacon2*)
- *value* IN (*beacon1*, *beacon2*, ...)
- CONTAINS(*beacon1*, *beacon2*)

I beacon composti possono eseguire le seguenti interrogazioni.

- `BEGINS_WITH(`*a*`)`, dove *a* riflette l'intero valore del campo con cui inizia il faro composto assemblato. Non è possibile utilizzare l'`BEGINS_WITH`operatore per identificare un valore che inizia con una particolare sottostringa. Tuttavia, è possibile utilizzare`BEGINS_WITH(`*S_*`)`, where *S_* riflette il prefisso per una parte con cui inizia il faro composto assemblato.
- `CONTAINS(`*a*`)`, dove *a* riflette l'intero valore di un campo contenuto nel beacon composto assemblato. Non è possibile utilizzare l'`CONTAINS`operatore per identificare un record che contiene una particolare sottostringa o un valore all'interno di un set.

  Ad esempio, non è possibile eseguire un'interrogazione `CONTAINS(`*path*`, "`*a*`"` in cui *a* rifletta il valore in un set.

- È possibile confrontare parti firmate di beacon composti. Quando si confrontano parti firmate, è possibile aggiungere facoltativamente il prefisso di una parte crittografata a una o più parti firmate, ma non è possibile includere il valore di un campo crittografato in nessuna query.

  Ad esempio, è possibile confrontare parti firmate ed eseguire una query su o. `signedField1 = signedField2 value IN (signedField1, signedField2, ...)`

  È inoltre possibile confrontare le parti firmate e il prefisso di una parte crittografata mediante interrogazione su`signedField1.A_ = signedField2.B_`.
- `field BETWEEN a AND b`, dove `a` e `b` sono le parti firmate. È possibile aggiungere facoltativamente il prefisso di una parte crittografata a una o più parti firmate, ma non è possibile includere il valore di un campo crittografato in nessuna query.

È necessario includere il prefisso per ogni parte inclusa in una query su un beacon composto. Ad esempio, se avete creato un beacon composto a partire da due campi `encryptedField` e `compoundBeaconsignedField`, dovete includere i prefissi configurati per queste due parti quando interrogate il beacon.

```
compoundBeacon = E_encryptedFieldValue.S_signedFieldValue
```

## Crittografia ricercabile per database multitenant

> La nostra libreria di crittografia lato client è stata rinominata Database Encryption SDK. AWS Questa guida per sviluppatori fornisce ancora informazioni sul DynamoDB Encryption Client.

Per implementare la crittografia ricercabile nel database, è necessario utilizzare un portachiavi gerarchico AWS KMS . Il portachiavi AWS KMS gerarchico genera, crittografa e decrittografa le chiavi dati utilizzate per proteggere i record. Crea anche la chiave beacon utilizzata per generare beacon. Quando si utilizza il portachiavi AWS KMS Hierarchical con database multitenant, esistono una chiave branch e una chiave beacon distinte per ogni tenant. Per interrogare i dati crittografati in un database multitenant, è necessario identificare i materiali chiave del beacon utilizzati per generare il beacon su cui si sta interrogando. Per ulteriori informazioni, consulta the section called “Utilizzo del portachiavi gerarchico per una crittografia ricercabile”.

Quando definisci la versione beacon per un database multitenant, specifica un elenco di tutti i beacon standard che hai configurato, un elenco di tutti i beacon composti che hai configurato, una versione

beacon e un. `keySource` È necessario definire la fonte della chiave beacon come e includere a) un `MultiKeyStore` tempo di permanenza della cache per la cache della chiave beacon locale e la dimensione massima della cache per la cache locale delle chiavi beacon. `keyFieldName`

Se hai configurato dei beacon firmati, questi devono essere inclusi nel tuo. `compoundBeaconList` I beacon firmati sono un tipo di beacon composto che indicizza ed esegue interrogazioni complesse su campi e. `SIGN_ONLY` `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`

Java

```
List<BeaconVersion> beaconVersions = new ArrayList<>();
    beaconVersions.add(
        BeaconVersion.builder()
                .standardBeacons(standardBeaconList)
                .compoundBeacons(compoundBeaconList)
                .version(1) // MUST be 1
                .keyStore(branchKeyStoreName)
                .keySource(BeaconKeySource.builder()
                        .multi(MultiKeyStore.builder()
                                .keyFieldName(keyField)
                                .cacheTTL(6000)
                                .maxCacheSize(10)
                        .build())
                .build())
        .build()
    );
```

C# / .NET

```
var beaconVersions = new List<BeaconVersion>
{
    new BeaconVersion
    {
        StandardBeacons = standardBeaconList,
        CompoundBeacons = compoundBeaconList,
        EncryptedParts = encryptedPartsList,
        SignedParts = signedPartsList,
        Version = 1, // MUST be 1
        KeyStore = branchKeyStoreName,
        KeySource = new BeaconKeySource
        {
            Multi = new MultiKeyStore
```

```
            {
                KeyId = branch-key-id,
                CacheTTL = 6000,
                MaxCacheSize = 10
            }
        }
    }
};
```

Rust

```
let beacon_version = BeaconVersion::builder()
    .standard_beacons(standard_beacon_list)
    .compound_beacons(compound_beacon_list)
    .version(1) // MUST be 1
    .key_store(key_store.clone())
    .key_source(BeaconKeySource::Multi(
        MultiKeyStore::builder()
            // `keyId` references a beacon key.
            // For every branch key we create in the keystore,
            // we also create a beacon key.
            // This beacon key is not the same as the branch key,
            // but is created with the same ID as the branch key.
            .key_id(branch_key_id)
            .cache_ttl(6000)
            .max_cache_size(10)
            .build()?,
    ))
    .build()?;
let beacon_versions = vec![beacon_version];
```

keyFieldName

`keyFieldName`definisce il nome del campo che memorizza la chiave beacon `branch-key-id` associata alla chiave beacon utilizzata per generare i beacon per un determinato tenant.

Quando si scrivono nuovi record nel database, la chiave `branch-key-id` che identifica la chiave beacon utilizzata per generare i beacon per quel record viene memorizzata in questo campo.

Per impostazione predefinita, `keyField` è un campo concettuale che non è memorizzato in modo esplicito nel database. Il AWS Database Encryption SDK identifica la `branch-key-id`chiave dei dati crittografati nella descrizione del materiale e memorizza il valore concettuale `keyField`

a cui potete fare riferimento nei beacon composti e nei beacon firmati. Poiché la descrizione del materiale è firmata, la parte concettuale è considerata una parte firmata. `keyField`

Puoi anche includerlo `keyField` nelle tue azioni crittografiche come `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` campo `SIGN_ONLY` or per memorizzare esplicitamente il campo nel tuo database. Se si esegue questa operazione, è necessario includerli manualmente `keyField` ogni volta che si scrive un record `branch-key-id` nel database.

## Interrogazione dei beacon in un database multi-tenant

Per interrogare un beacon, è necessario includere `keyField` nella query i materiali chiave del beacon appropriati necessari per ricalcolare il beacon. È necessario specificare la chiave `branch-key-id` associata alla chiave beacon utilizzata per generare i beacon per un record. Non è possibile specificare il nome descrittivo che identifica il fornitore dell'ID della chiave di filiale di `branch-key-id` un tenant. Puoi includerli `keyField` nelle tue domande nei seguenti modi.

Fari composti

Che li memorizziate esplicitamente o meno `keyField` nei vostri archivi, potete includerli `keyField` direttamente nei vostri beacon compositi come parte firmata. La parte `keyField` firmata deve essere obbligatoria.

Ad esempio, se si desidera creare un faro composto da due campi `encryptedField` e `compoundBeaconsignedField`, è necessario includerlo anche `keyField` come parte firmata. Ciò consente di eseguire la seguente interrogazione su. `compoundBeacon`

```
compoundBeacon = E_encryptedFieldValue.S_signedFieldValue.K_branch-key-id
```

Fari firmati

Il AWS Database Encryption SDK utilizza beacon standard e compositi per fornire soluzioni di crittografia ricercabili. Questi beacon devono includere almeno un campo crittografato. Tuttavia, AWS Database Encryption SDK supporta anche beacon firmati che possono essere configurati interamente da testo e campi in chiaro. `SIGN_ONLY` `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`

I beacon firmati possono essere costruiti da un'unica parte. Indipendentemente dal fatto che lo memorizziate esplicitamente `keyField` nei vostri archivi o meno, potete costruire un beacon firmato da `keyField` e utilizzarlo per creare query composte che combinano una query sul

beacon `keyField` firmato con una query su uno degli altri beacon. Ad esempio, è possibile eseguire la seguente query.

```
keyField = K_branch-key-id AND compoundBeacon =
 E_encryptedFieldValue.S_signedFieldValue
```

Per informazioni sulla configurazione dei beacon firmati, vedere Creazione di beacon firmati

Effettua una query direttamente su **`keyField`**

Se hai specificato il `keyField` nelle tue azioni crittografiche e lo hai archiviato in modo esplicito nel tuo record, puoi creare una query composta che combina una query sul tuo beacon con una query su. `keyField` Puoi scegliere di interrogare direttamente su `keyField` se desideri interrogare un beacon standard. Ad esempio, è possibile eseguire la seguente query.

```
keyField = branch-key-id AND standardBeacon = S_standardBeaconValue
```

# AWS SDK di crittografia del database per DynamoDB

La nostra libreria di crittografia lato client è stata rinominata Database Encryption SDK. AWS Questa guida per sviluppatori fornisce ancora informazioni sul DynamoDB Encryption Client.

Il AWS Database Encryption SDK per DynamoDB è una libreria software che consente di includere la crittografia lato client nella progettazione di Amazon DynamoDB. Il AWS Database Encryption SDK per DynamoDB fornisce la crittografia a livello di attributo e consente di specificare quali elementi crittografare e quali elementi includere nelle firme che garantiscono l'autenticità dei dati. La crittografia dei dati sensibili in transito e in archivio aiuta a garantire che i dati in chiaro non siano disponibili per terze parti, tra cui. AWS

> **Note**
>
> Il AWS Database Encryption SDK non supporta PartiQL.

In DynamoDB, una tabella è una raccolta di elementi. E ogni item è una raccolta di attributi. Ogni attributo ha un nome e un valore. Il AWS Database Encryption SDK per DynamoDB crittografa i valori degli attributi. Quindi calcola una firma sugli attributi. È necessario specificare quali valori degli attributi crittografare e quali includere nella firma nelle azioni crittografiche.

Gli argomenti di questo capitolo forniscono una panoramica del AWS Database Encryption SDK per DynamoDB, inclusi i campi crittografati, indicazioni sull'installazione e la configurazione dei client ed esempi Java per aiutarti a iniziare.

Argomenti

# Crittografia lato client e lato server

La nostra libreria di crittografia lato client è stata rinominata AWS Database Encryption SDK. Questa guida per sviluppatori fornisce ancora informazioni sul DynamoDB Encryption Client.

Il AWS Database Encryption SDK per DynamoDB supporta la crittografia lato client, che consente di crittografare i dati della tabella prima di inviarli al database. Tuttavia, DynamoDB fornisce una funzionalità di crittografia a riposo lato server che crittografa in modo trasparente la tabella quando viene mantenuta su disco e la decrittografa quando si accede alla tabella.

La scelta degli strumenti dipende dal livello di riservatezza dei dati e dai requisiti di sicurezza dell'applicazione. È possibile utilizzare sia il AWS Database Encryption SDK per DynamoDB sia la crittografia a riposo. Quando invii elementi crittografati e firmati a DynamoDB, DynamoDB non riconosce gli elementi come protetti. Rileva soltanto gli item tipici della tabella con valori di attributo binari.

Crittografia dei dati inattivi lato server

DynamoDB supporta la crittografia a riposo, una funzionalità di crittografia lato server in cui DynamoDB crittografa in modo trasparente le tabelle per te quando la tabella viene mantenuta su disco e le decrittografa quando accedi ai dati della tabella.

Quando si utilizza un AWS SDK per interagire con DynamoDB, per impostazione predefinita, i dati vengono crittografati in transito tramite una connessione HTTPS, decrittografati sull'endpoint DynamoDB e quindi ricrittografati prima di essere archiviati in DynamoDB.

- Crittografia per impostazione predefinita. DynamoDB crittografa e decrittografa in modo trasparente tutte le tabelle quando vengono scritte. Non c'è alcuna opzione per abilitare o disabilitare la crittografia dei dati inattivi.
- DynamoDB crea e gestisce le chiavi crittografiche.La chiave univoca per ogni tabella è protetta da un codice AWS KMS keyche non lascia mai AWS Key Management Service()AWS KMS non crittografata. Per impostazione predefinita, DynamoDB utilizza Chiave di proprietà di AWSun account del servizio DynamoDB, ma puoi scegliere Chiave gestita da AWSuna chiave o una chiave gestita dal cliente nel tuo account per proteggere alcune o tutte le tabelle.
- Tutti i dati delle tabelle sono crittografati su disco.Quando una tabella crittografata viene salvata su disco, DynamoDB crittografa tutti i dati della tabella, inclusa la chiave primaria e gli indici secondari

locali e globali. Se la tabella dispone di chiavi di ordinamento, alcune di queste che contrassegnano i limiti dell'intervallo sono archiviate come testo non crittografato nei metadati della tabella.

- Anche gli oggetti correlati alle tabelle sono crittografati. La crittografia a riposo protegge i flussi, le tabelle globali e i backup di DynamoDB ogni volta che vengono scritti su supporti durevoli.
- I tuoi articoli vengono decriptati quando accedi ad essi.Quando accedi alla tabella, DynamoDB decrittografa la parte della tabella che include l'elemento di destinazione e ti restituisce l'elemento in testo normale.

### AWS SDK di crittografia del database per DynamoDB

La crittografia lato client fornisce end-to-end protezione per i dati, in transito e a riposo, dalla loro origine allo storage in DynamoDB. I tuoi dati in chiaro non vengono mai esposti a terze parti, inclusi. AWS Puoi utilizzare AWS Database Encryption SDK per DynamoDB con nuove tabelle DynamoDB oppure puoi migrare le tabelle Amazon DynamoDB esistenti alla versione più recente di Database Encryption SDK per DynamoDB. AWS

- I dati sono protetti sia quando sono in transito sia quando sono inattivi. Non è mai esposto a terze parti, inclusi. AWS
- Puoi firmare gli item della tabella. Puoi indirizzare il AWS Database Encryption SDK per DynamoDB a calcolare una firma su tutto o parte di un elemento della tabella, inclusi gli attributi della chiave primaria. Tramite le firme, puoi rilevare modifiche non autorizzate all'item nel suo insieme, tra cui l'aggiunta o l'eliminazione di attributi o lo scambio dei valori di attributo.
- Puoi determinare come proteggere i tuoi dati selezionando un portachiavi. Il portachiavi determina le chiavi di avvolgimento che proteggono le chiavi dati e, in ultima analisi, i dati. Utilizzate le chiavi di avvolgimento più sicure e pratiche per il vostro compito.
- Il AWS Database Encryption SDK per DynamoDB non crittografa l'intera tabella. Sei tu a scegliere quali attributi crittografare nei tuoi articoli. Il AWS Database Encryption SDK per DynamoDB non crittografa un intero elemento. Non crittografa i nomi di attributo o i nomi o i valori degli attributi della chiave primaria (chiave di partizione e chiave di ordinamento).

### AWS Encryption SDK

Se stai crittografando i dati archiviati in DynamoDB, ti consigliamo AWS il Database Encryption SDK per DynamoDB.

AWS Encryption SDK è una libreria di crittografia lato client che ti consente di crittografare e decrittografare i dati generici. Anche se è in grado di proteggere qualsiasi tipo di dati, non è stato

progettato per funzionare con i dati strutturati, come i record di database. A differenza del AWS Database Encryption SDK per DynamoDB, non può fornire AWS Encryption SDK il controllo dell'integrità a livello di elemento e non ha alcuna logica per riconoscere gli attributi o impedire la crittografia delle chiavi primarie.

Se utilizzi il AWS Encryption SDK per crittografare qualsiasi elemento della tabella, ricorda che non è compatibile con il AWS Database Encryption SDK per DynamoDB. Non puoi utilizzare due librerie diverse per la crittografia e la decrittografia.

## Quali campi sono crittografati e firmati?

> La nostra libreria di crittografia lato client è stata rinominata AWS Database Encryption SDK. Questa guida per sviluppatori fornisce ancora informazioni sul DynamoDB Encryption Client.

Il AWS Database Encryption SDK per DynamoDB è una libreria di crittografia lato client progettata appositamente per le applicazioni Amazon DynamoDB. Amazon DynamoDB archivia i dati in tabelle, che sono una raccolta di elementi. E ogni item è una raccolta di attributi. Ogni attributo ha un nome e un valore. Il AWS Database Encryption SDK per DynamoDB crittografa i valori degli attributi. Quindi calcola una firma sugli attributi. Puoi specificare quali valori degli attributi crittografare e quali includere nelle firma.

La crittografia protegge la riservatezza del valore degli attributi. La firma assicura l'integrità di tutti gli attributi firmati e la loro relazione reciproca e fornisce l'autenticazione. Consente di rilevare modifiche non autorizzate all'item nel suo insieme, come l'aggiunta o l'eliminazione di attributi o la sostituzione di un valore crittografato con un altro.

In un elemento crittografato, alcuni dati rimangono in testo semplice, tra cui il nome della tabella, tutti i nomi degli attributi, i valori degli attributi che non vengono crittografati, i nomi e i valori degli attributi della chiave primaria (chiave di partizione e chiave di ordinamento) e i tipi di attributo. Non archiviare dati sensibili in questi campi.

Per ulteriori informazioni su come funziona AWS Database Encryption SDK per DynamoDB, consulta. Come funziona il AWS Database Encryption SDK

> **Note**
>
> Tutte le menzioni delle azioni degli attributi negli argomentiAWS Database Encryption SDK per DynamoDB si riferiscono alle azioni crittografiche.

Argomenti



## Crittografia dei valori degli attributi

Il AWS Database Encryption SDK per DynamoDB crittografa i valori (ma non il nome o il tipo di attributo) degli attributi specificati. Per determinare quali valori attributo vengono crittografati, utilizza le operazioni di attributo.

Ad esempio, questo item include gli attributi `example` e `test`.

```
'example': 'data',
'test': 'test-value',
...
```

Se effettui la crittografia dell'attributo `example`, ma non dell'attributo `test`, i risultati saranno simili a quelli riportati di seguito. Il valore dell'attributo `example` sono dati binari e non una stringa.

```
'example': Binary(b"'b\x933\x9a+s\xf1\xd6a\xc5\xd5\x1aZ\xed\xd6\xce\xe9X\xf0T\xcb\x9fY
\x9f\xf3\xc9C\x83\r\xbb\\"),
'test': 'test-value'
...
```

Gli attributi della chiave primaria, chiave di partizione e chiave di ordinamento, di ogni elemento devono rimanere in testo semplice perché DynamoDB li utilizza per trovare l'elemento nella tabella. Devono essere firmati, ma non crittografati.

Il AWS Database Encryption SDK per DynamoDB identifica automaticamente gli attributi della chiave primaria e garantisce che i loro valori siano firmati, ma non crittografati. Se individui la tua chiave primaria e tenti di crittografarla, il client genera un'eccezione.

Il client memorizza la descrizione del materiale in un nuovo attributo (`aws_dbe_head`) che aggiunge all'articolo. La descrizione del materiale descrive come l'articolo è stato crittografato e firmato. Il client utilizza l'informazione per verificare e decrittografare l'item. Il campo che memorizza la descrizione del materiale non è crittografato.

### Firma dell'item

Dopo aver crittografato i valori degli attributi specificati, AWS Database Encryption SDK per DynamoDB calcola i codici di autenticazione dei messaggi basati su hash (HMACs) e una firma digitale sulla canonicalizzazione della descrizione del materiale, del contesto di crittografia e di ogni campo contrassegnato o nelle azioni degli attributi. `ENCRYPT_AND_SIGNSIGN_ONLYSIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` Le firme ECDSA sono abilitate per impostazione predefinita, ma non sono obbligatorie. Il client memorizza le firme HMACs and in un nuovo attributo (`aws_dbe_foot`) che aggiunge all'elemento.

## Crittografia ricercabile in DynamoDB

Per configurare le tabelle Amazon DynamoDB per una crittografia ricercabile, devi utilizzare AWS KMS il portachiavi gerarchico per generare, crittografare e decrittografare le chiavi dati utilizzate per proteggere i tuoi articoli. È inoltre necessario includere la configurazione di crittografia nella tabella. `SearchConfig`

> **Note**
>
> Se utilizzi la libreria di crittografia lato client Java per DynamoDB, devi utilizzare l'API di basso livello AWS Database Encryption SDK for DynamoDB per DynamoDB per crittografare, firmare, verificare e decrittografare gli elementi della tabella. Il DynamoDB Enhanced Client e il `DynamoDBItemEncryptor` livello inferiore non supportano la crittografia ricercabile.

Argomenti

- Configurazione degli indici secondari con beacon
- Test delle uscite dei beacon

# Configurazione degli indici secondari con beacon

Dopo aver configurato i beacon, è necessario configurare un indice secondario che rifletta ogni beacon prima di poter eseguire la ricerca negli attributi crittografati.

Quando configurate un beacon standard o composto, AWS Database Encryption SDK aggiunge il `aws_dbe_b_` prefisso al nome del beacon in modo che il server possa identificare facilmente i beacon. Ad esempio, se si assegna un nome a un beacon composto, in realtà il nome completo del beacon è. `compoundBeacon` `aws_dbe_b_compoundBeacon` Se si desidera configurare indici secondari che includano un beacon standard o composto, è necessario includere il prefisso quando si identifica il nome del `aws_dbe_b_` beacon.

Chiavi di partizione e ordinamento

Non è possibile crittografare i valori della chiave primaria. Le chiavi di partizione e di ordinamento devono essere firmate. I valori della chiave primaria non possono essere un beacon standard o composto.

I valori della chiave primaria devono essere`SIGN_ONLY`, a meno che non si specifichi alcun `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` attributo, anche gli attributi di partizione e ordinamento. `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`

I valori della chiave primaria possono essere beacon firmati. Se hai configurato beacon firmati distinti per ciascuno dei valori della chiave primaria, devi specificare il nome dell'attributo che identifica il valore della chiave primaria come nome del beacon firmato. Tuttavia, AWS Database Encryption SDK non aggiunge il prefisso ai beacon firmati. `aws_dbe_b_` Anche se hai configurato beacon firmati distinti per i valori della chiave primaria, devi specificare i nomi degli attributi per i valori della chiave primaria solo quando configuri un indice secondario.

Indici secondari locali

La chiave di ordinamento per un indice secondario locale può essere un beacon.

Se si specifica un beacon per la chiave di ordinamento, il tipo deve essere String. Se specificate un beacon standard o composto per la chiave di ordinamento, dovete includere il `aws_dbe_b_` prefisso quando specificate il nome del beacon. Se specificate un faro firmato, specificate il nome del beacon senza alcun prefisso.

Indici secondari globali

Le chiavi di partizione e di ordinamento per un indice secondario globale possono essere entrambe beacon.

Se si specifica un beacon per la chiave di partizione o di ordinamento, il tipo deve essere String. Se specificate un beacon standard o composto per la chiave di ordinamento, dovete includere il `aws_dbe_b_` prefisso quando specificate il nome del beacon. Se specificate un faro firmato, specificate il nome del beacon senza alcun prefisso.

Proiezioni di attributi

Una proiezione è l'insieme di attributi copiato da una tabella in un indice secondario. La chiave di partizione e la chiave di ordinamento della tabella vengono sempre proiettati nell'indice; è possibile proiettare altri attributi per supportare i requisiti di query dell'applicazione. DynamoDB offre tre diverse opzioni per le proiezioni `KEYS_ONLY` degli attributi:, e. `INCLUDE` `ALL`

Se si utilizza la proiezione dell'attributo INCLUDE per cercare su un beacon, è necessario specificare i nomi di tutti gli attributi da cui è costruito il beacon e il nome del beacon con il prefisso. `aws_dbe_b_` Ad esempio, se avete configurato un faro composto, from, and`compoundBeacon`, dovete specificare`field1`, `field2``field3`, e nella proiezione. `aws_dbe_b_compoundBeacon` `field1` `field2` `field3`

Un indice secondario globale può utilizzare solo gli attributi specificati esplicitamente nella proiezione, ma un indice secondario locale può utilizzare qualsiasi attributo.

## Test delle uscite dei beacon

Se hai configurato beacon compositi o costruito i beacon utilizzando campi virtuali, ti consigliamo di verificare che questi beacon producano l'output previsto prima di popolare la tabella DynamoDB.

Il AWS Database Encryption SDK fornisce il `DynamoDbEncryptionTransforms` servizio per aiutarti a risolvere i problemi relativi agli output dei campi virtuali e dei beacon compositi.

Test dei campi virtuali

Il seguente frammento crea elementi di test, definisce il `DynamoDbEncryptionTransforms` servizio con la configurazione di crittografia della tabella DynamoDB e dimostra come `ResolveAttributes` utilizzarlo per verificare che il campo virtuale produca l'output previsto.

Java

Guarda l'esempio di codice completo: .java VirtualBeaconSearchableEncryptionExample

```
// Create test items
final PutItemRequest itemWithHasTestResultPutRequest = PutItemRequest.builder()
```

```
    .tableName(ddbTableName)
    .item(itemWithHasTestResult)
    .build();

final PutItemResponse itemWithHasTestResultPutResponse =
 ddb.putItem(itemWithHasTestResultPutRequest);

final PutItemRequest itemWithNoHasTestResultPutRequest = PutItemRequest.builder()
    .tableName(ddbTableName)
    .item(itemWithNoHasTestResult)
    .build();

final PutItemResponse itemWithNoHasTestResultPutResponse =
 ddb.putItem(itemWithNoHasTestResultPutRequest);

// Define the DynamoDbEncryptionTransforms service
final DynamoDbEncryptionTransforms trans = DynamoDbEncryptionTransforms.builder()
    .DynamoDbTablesEncryptionConfig(encryptionConfig).build();

// Verify configuration
final ResolveAttributesInput resolveInput = ResolveAttributesInput.builder()
    .TableName(ddbTableName)
    .Item(itemWithHasTestResult)
    .Version(1)
    .build();
final ResolveAttributesOutput resolveOutput = trans.ResolveAttributes(resolveInput);

// Verify that VirtualFields has the expected value
Map<String, String> vf = new HashMap<>();
vf.put("stateAndHasTestResult", "CAt");
assert resolveOutput.VirtualFields().equals(vf);
```

C# / .NET

Guarda l'esempio di codice completo: VirtualBeaconSearchableEncryptionExample.cs.

```
 // Create item with hasTestResult=true
var itemWithHasTestResult = new Dictionary<String, AttributeValue>
{
    ["customer_id"] = new AttributeValue("ABC-123"),
    ["create_time"] = new AttributeValue { N = "1681495205" },
    ["state"] = new AttributeValue("CA"),
    ["hasTestResult"] = new AttributeValue { BOOL = true }
};
```

```
// Create item with hasTestResult=false
var itemWithNoHasTestResult = new Dictionary<String, AttributeValue>
{
    ["customer_id"] = new AttributeValue("DEF-456"),
    ["create_time"] = new AttributeValue { N = "1681495205" },
    ["state"] = new AttributeValue("CA"),
    ["hasTestResult"] = new AttributeValue { BOOL = false }
};

// Define the DynamoDbEncryptionTransforms service
var trans = new DynamoDbEncryptionTransforms(encryptionConfig);

// Verify configuration
var resolveInput = new ResolveAttributesInput
{
    TableName = ddbTableName,
    Item = itemWithHasTestResult,
    Version = 1
};
var resolveOutput = trans.ResolveAttributes(resolveInput);

// Verify that VirtualFields has the expected value
Debug.Assert(resolveOutput.VirtualFields.Count == 1);
Debug.Assert(resolveOutput.VirtualFields["stateAndHasTestResult"] == "CAt");
```

Rust

Guarda l'esempio di codice completo: virtual_beacon_searchable_encryption.rs.

```
// Create item with hasTestResult=true
let item_with_has_test_result = HashMap::from([
    (
        "customer_id".to_string(),
        AttributeValue::S("ABC-123".to_string()),
    ),
    (
        "create_time".to_string(),
        AttributeValue::N("1681495205".to_string()),
    ),
    ("state".to_string(), AttributeValue::S("CA".to_string())),
    ("hasTestResult".to_string(), AttributeValue::Bool(true)),
]);
```

```
// Create item with hasTestResult=false
let item_with_no_has_test_result = HashMap::from([
    (
        "customer_id".to_string(),
        AttributeValue::S("DEF-456".to_string()),
    ),
    (
        "create_time".to_string(),
        AttributeValue::N("1681495205".to_string()),
    ),
    ("state".to_string(), AttributeValue::S("CA".to_string())),
    ("hasTestResult".to_string(), AttributeValue::Bool(false)),
]);

// Define the transform service
let trans = transform_client::Client::from_conf(encryption_config.clone())?;

// Verify the configuration
let resolve_output = trans
    .resolve_attributes()
    .table_name(ddb_table_name)
    .item(item_with_has_test_result.clone())
    .version(1)
    .send()
    .await?;

// Verify that VirtualFields has the expected value
let virtual_fields = resolve_output.virtual_fields.unwrap();
assert_eq!(virtual_fields.len(), 1);
assert_eq!(virtual_fields["stateAndHasTestResult"], "CAt");
```

### Test dei beacon composti

Il seguente frammento crea un elemento di test, definisce il `DynamoDbEncryptionTransforms` servizio con la configurazione di crittografia della tabella DynamoDB e dimostra come `ResolveAttributes` utilizzarlo per verificare che il beacon composto produca l'output previsto.

Java

Guarda l'esempio di codice completo: .java CompoundBeaconSearchableEncryptionExample

```
// Create an item with both attributes used in the compound beacon.
```

```
final HashMap<String, AttributeValue> item = new HashMap<>();
item.put("work_id", AttributeValue.builder().s("9ce39272-8068-4efd-a211-
cd162ad65d4c").build());
item.put("inspection_date", AttributeValue.builder().s("2023-06-13").build());
item.put("inspector_id_last4", AttributeValue.builder().s("5678").build());
item.put("unit", AttributeValue.builder().s("011899988199").build());

// Define the DynamoDbEncryptionTransforms service
final DynamoDbEncryptionTransforms trans = DynamoDbEncryptionTransforms.builder()
    .DynamoDbTablesEncryptionConfig(encryptionConfig).build();

// Verify configuration
final ResolveAttributesInput resolveInput = ResolveAttributesInput.builder()
    .TableName(ddbTableName)
    .Item(item)
    .Version(1)
    .build();

final ResolveAttributesOutput resolveOutput = trans.ResolveAttributes(resolveInput);

// Verify that CompoundBeacons has the expected value
Map<String, String> cbs = new HashMap<>();
cbs.put("last4UnitCompound", "L-5678.U-011899988199");
assert resolveOutput.CompoundBeacons().equals(cbs);
// Note : the compound beacon actually stored in the table is not
 "L-5678.U-011899988199"
// but rather something like "L-abc.U-123", as both parts are EncryptedParts
// and therefore the text is replaced by the associated beacon
```

C# / .NET

Guarda l'esempio di codice completo: .cs CompoundBeaconSearchableEncryptionExample

```
// Create an item with both attributes used in the compound beacon
var item = new Dictionary<String, AttributeValue>
{
    ["work_id"] = new AttributeValue("9ce39272-8068-4efd-a211-cd162ad65d4c"),
    ["inspection_date"] = new AttributeValue("2023-06-13"),
    ["inspector_id_last4"] = new AttributeValue("5678"),
    ["unit"] = new AttributeValue("011899988199")
};

// Define the DynamoDbEncryptionTransforms service
var trans = new DynamoDbEncryptionTransforms(encryptionConfig);
```

```
// Verify configuration
var resolveInput = new ResolveAttributesInput
{
    TableName = ddbTableName,
    Item = item,
    Version = 1
};
var resolveOutput = trans.ResolveAttributes(resolveInput);


// Verify that CompoundBeacons has the expected value
Debug.Assert(resolveOutput.CompoundBeacons.Count == 1);
Debug.Assert(resolveOutput.CompoundBeacons["last4UnitCompound"] ==
 "L-5678.U-011899988199");
// Note : the compound beacon actually stored in the table is not
 "L-5678.U-011899988199"
// but rather something like "L-abc.U-123", as both parts are EncryptedParts
// and therefore the text is replaced by the associated beacon
```

Rust

Guarda l'esempio di codice completo: compound_beacon_searchable_encryption.rs

```
// Create an item with both attributes used in the compound beacon
let item = HashMap::from([
    (
        "work_id".to_string(),
        AttributeValue::S("9ce39272-8068-4efd-a211-cd162ad65d4c".to_string()),
    ),
    (
        "inspection_date".to_string(),
        AttributeValue::S("2023-06-13".to_string()),
    ),
    (
        "inspector_id_last4".to_string(),
        AttributeValue::S("5678".to_string()),
    ),
    (
        "unit".to_string(),
        AttributeValue::S("011899988199".to_string()),
    ),
]);
```

```
// Define the transforms service
let trans = transform_client::Client::from_conf(encryption_config.clone())?;

// Verify configuration
let resolve_output = trans
    .resolve_attributes()
    .table_name(ddb_table_name)
    .item(item.clone())
    .version(1)
    .send()
    .await?;

// Verify that CompoundBeacons has the expected value
Dlet compound_beacons = resolve_output.compound_beacons.unwrap();
assert_eq!(compound_beacons.len(), 1);
assert_eq!(
    compound_beacons["last4UnitCompound"],
    "L-5678.U-011899988199"
);
// but rather something like "L-abc.U-123", as both parts are EncryptedParts
// and therefore the text is replaced by the associated beacon
```

# Aggiornamento del modello di dati

La nostra libreria di crittografia lato client è stata rinominata Database Encryption SDK. AWS Questa guida per sviluppatori fornisce ancora informazioni sul DynamoDB Encryption Client.

Quando configuri il AWS Database Encryption SDK per DynamoDB, fornisci azioni relative agli attributi. Su encrypt, AWS Database Encryption SDK utilizza le azioni degli attributi per identificare quali attributi crittografare e firmare, quali attributi firmare (ma non crittografare) e quali ignorare. È inoltre possibile definire gli attributi non firmati consentiti per indicare in modo esplicito al client quali attributi sono esclusi dalle firme. In fase di decrittografia, AWS Database Encryption SDK utilizza gli attributi non firmati consentiti definiti dall'utente per identificare gli attributi non inclusi nelle firme. Le azioni relative agli attributi non vengono salvate nell'elemento crittografato e AWS Database Encryption SDK non aggiorna automaticamente le azioni relative agli attributi.

Scegli attentamente le operazioni di attributo. In caso di dubbio, usa Encrypt and sign (Crittografa e firma). Dopo aver utilizzato il AWS Database Encryption SDK per proteggere

gli elementi, non è possibile modificare un attributo esistente `ENCRYPT_AND_SIGN` o un `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` attributo a. `SIGN_ONLY` `DO_NOTHING` Tuttavia, puoi apportare in sicurezza le seguenti modifiche.

- Aggiungi nuovi `ENCRYPT_AND_SIGN`SIGN_ONLY attributi e `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`
- Rimuovi gli attributi esistenti
- Modificate un `ENCRYPT_AND_SIGN` attributo esistente in `SIGN_ONLY` o `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`
- Modificate un `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` attributo esistente `SIGN_ONLY` o in `ENCRYPT_AND_SIGN`
- Aggiungere un nuovo `DO_NOTHING` attributo
- Modificare un `SIGN_ONLY` attributo esistente in `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`
- Modificate un `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` attributo esistente in `SIGN_ONLY`

Considerazioni sulla crittografia ricercabile

Prima di aggiornare il modello di dati, valuta attentamente in che modo gli aggiornamenti potrebbero influire sui beacon che hai creato a partire dagli attributi. Dopo aver scritto nuovi record con un beacon, non è possibile aggiornare la configurazione del beacon. Non è possibile aggiornare le azioni relative agli attributi associati agli attributi utilizzati per costruire i beacon. Se rimuovi un attributo esistente e il beacon associato, non sarai in grado di interrogare i record esistenti utilizzando quel beacon. È possibile creare nuovi beacon per i nuovi campi che si aggiungono al record, ma non è possibile aggiornare i beacon esistenti per includere il nuovo campo.

Considerazioni sugli attributi **`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`**

Per impostazione predefinita, le chiavi di partizione e ordinamento sono l'unico attributo incluso nel contesto di crittografia. Potresti prendere in considerazione la definizione di campi aggiuntivi `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` in modo che il fornitore dell'ID della chiave di filiale per il tuo portachiavi AWS KMS gerarchico possa identificare quale chiave di filiale è necessaria per la decrittografia dal contesto di crittografia. Per ulteriori informazioni, consulta Branch Key ID supplier. Se si specificano `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` degli attributi, devono esserlo `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` anche gli attributi di partizione e ordinamento.

> **Note**
>
> Per utilizzare l'azione `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` crittografica, è necessario utilizzare la versione 3.3 o successiva di AWS Database Encryption SDK. Distribuisci la nuova versione a tutti i lettori prima di aggiornare il modello di dati per includere. `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`

## Aggiungi nuovi **`ENCRYPT_AND_SIGNSIGN_ONLY`** attributi e **`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`**

Per aggiungere un nuovo `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` attributo `ENCRYPT_AND_SIGNSIGN_ONLY`, o, definisci il nuovo attributo nelle azioni relative agli attributi.

Non è possibile rimuovere un `DO_NOTHING` attributo esistente e aggiungerlo nuovamente come `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` attributo `ENCRYPT_AND_SIGNSIGN_ONLY`,, o.

Utilizzo di una classe di dati annotata

Se hai definito le azioni degli attributi con a`TableSchema`, aggiungi il nuovo attributo alla tua classe di dati annotata. Se non specificate un'annotazione relativa all'azione degli attributi per il nuovo attributo, il client crittograferà e firmerà il nuovo attributo per impostazione predefinita (a meno che l'attributo non faccia parte della chiave primaria). Se si desidera firmare solo il nuovo attributo, è necessario aggiungere il nuovo attributo con l'annotazione `@DynamoDBEncryptionSignOnly` o`@DynamoDBEncryptionSignAndIncludeInEncryptionContext`.

Utilizzo di un modello a oggetti

Se avete definito manualmente le azioni degli attributi, aggiungete il nuovo attributo alle azioni degli attributi nel modello a oggetti e specificate `ENCRYPT_AND_SIGNSIGN_ONLY`, o `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` come azione di attributo.

## Rimuovi gli attributi esistenti

Se decidi di non aver più bisogno di un attributo, puoi smettere di scrivere dati su quell'attributo oppure puoi rimuoverlo formalmente dalle tue azioni relative agli attributi. Quando smetti di scrivere nuovi dati su un attributo, l'attributo viene ancora visualizzato nelle tue azioni relative agli attributi. Ciò può essere utile se è necessario ricominciare a utilizzare l'attributo in futuro. La rimozione formale

dell'attributo dalle azioni relative agli attributi non lo rimuove dal set di dati. Il set di dati conterrà comunque elementi che includono quell'attributo.

Per rimuovere formalmente un `DO_NOTHING` attributo`ENCRYPT_AND_SIGN`,, o esistente `SIGN_ONLY``SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`, aggiorna le azioni relative agli attributi.

Se rimuovi un `DO_NOTHING` attributo, non devi rimuovere quell'attributo dagli attributi non firmati consentiti. Anche se non state più scrivendo nuovi valori per quell'attributo, il client deve comunque sapere che l'attributo non è firmato per leggere gli elementi esistenti che lo contengono.

Utilizzo di una classe di dati annotata

Se hai definito le azioni degli attributi con `aTableSchema`, rimuovi l'attributo dalla classe di dati annotata.

Utilizzo di un modello a oggetti

Se avete definito manualmente le azioni relative agli attributi, rimuovete l'attributo dalle azioni degli attributi nel modello a oggetti.

## Modificate un **`ENCRYPT_AND_SIGN`** attributo esistente in **`SIGN_ONLY`** o **`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`**

Per modificare un `ENCRYPT_AND_SIGN` attributo esistente in `SIGN_ONLY` o`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`, è necessario aggiornare le azioni relative agli attributi. Dopo aver distribuito l'aggiornamento, il client sarà in grado di verificare e decrittografare i valori esistenti scritti nell'attributo, ma firmerà solo i nuovi valori scritti nell'attributo.

> **Note**
>
> Valuta attentamente i requisiti di sicurezza prima di modificare un `ENCRYPT_AND_SIGN` attributo esistente in o. `SIGN_ONLY` `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` Qualsiasi attributo in grado di memorizzare dati sensibili deve essere crittografato.

Utilizzo di una classe di dati annotata

Se hai definito le azioni degli attributi con aTableSchema, aggiorna l'attributo esistente per includere l'@DynamoDBEncryptionSignAndIncludeInEncryptionContextannotazione @DynamoDBEncryptionSignOnly o nella classe di dati annotata.

Utilizzo di un modello a oggetti

Se avete definito manualmente le azioni relative agli attributi, aggiornate l'azione dell'attributo associata all'attributo esistente da SIGN_ONLY o SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT nel modello ENCRYPT_AND_SIGN a oggetti.

## Modificate un **SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT** attributo esistente **SIGN_ONLY** o in **ENCRYPT_AND_SIGN**

Per modificare un SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT attributo esistente SIGN_ONLY o inENCRYPT_AND_SIGN, è necessario aggiornare le azioni relative agli attributi. Dopo aver distribuito l'aggiornamento, il client sarà in grado di verificare i valori esistenti scritti nell'attributo e crittograferà e firmerà i nuovi valori scritti nell'attributo.

Utilizzo di una classe di dati annotata

Se hai definito le azioni degli attributi con aTableSchema, rimuovi l'@DynamoDBEncryptionSignAndIncludeInEncryptionContextannotazione @DynamoDBEncryptionSignOnly or dall'attributo esistente.

Utilizzo di un modello a oggetti

Se avete definito manualmente le azioni relative agli attributi, aggiornate l'azione dell'attributo associata all'attributo from SIGN_ONLY o SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT to ENCRYPT_AND_SIGN nel modello a oggetti.

## Aggiungere un nuovo **DO_NOTHING** attributo

Per ridurre il rischio di errori durante l'aggiunta di un nuovo DO_NOTHING attributo, consigliamo di specificare un prefisso distinto quando si assegnano i nomi DO_NOTHING agli attributi e quindi di utilizzare tale prefisso per definire gli attributi non firmati consentiti.

Non è possibile rimuovere un SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT attributo o esistente ENCRYPT_AND_SIGN dalla classe di dati annotata e quindi aggiungere nuovamente l'attributo come attributo. SIGN_ONLY DO_NOTHING È possibile aggiungere solo DO_NOTHING attributi completamente nuovi.

I passaggi da eseguire per aggiungere un nuovo DO_NOTHING attributo dipendono dal fatto che gli attributi non firmati consentiti siano stati definiti esplicitamente in un elenco o con un prefisso.

Utilizzo di un prefisso consentito per gli attributi non firmati

Se hai definito le azioni degli attributi con aTableSchema, aggiungi il nuovo DO_NOTHING attributo alla classe di dati annotata con l'annotazione. @DynamoDBEncryptionDoNothing Se hai definito manualmente le azioni relative agli attributi, aggiorna le azioni degli attributi per includere il nuovo attributo. Assicurati di configurare in modo esplicito il nuovo attributo con l'azione dell'DO_NOTHINGattributo. È necessario includere lo stesso prefisso distinto nel nome del nuovo attributo.

Utilizzo di un elenco di attributi non firmati consentiti

1. Aggiungi il nuovo DO_NOTHING attributo all'elenco degli attributi non firmati consentiti e distribuisci l'elenco aggiornato.
2. Implementa la modifica dalla Fase 1.

   Non è possibile passare alla Fase 3 finché la modifica non si è propagata a tutti gli host che devono leggere questi dati.
3. Aggiungi il nuovo DO_NOTHING attributo alle tue azioni relative agli attributi.

   a. Se hai definito le azioni degli attributi con aTableSchema, aggiungi il nuovo DO_NOTHING attributo alla classe di dati annotata con l'@DynamoDBEncryptionDoNothingannotazione.

   b. Se hai definito manualmente le azioni relative agli attributi, aggiorna le azioni degli attributi per includere il nuovo attributo. Assicurati di configurare in modo esplicito il nuovo attributo con l'azione dell'DO_NOTHINGattributo.
4. Implementa la modifica dalla Fase 3.

## Modificare un **SIGN_ONLY** attributo esistente in **SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT**

Per modificare un SIGN_ONLY attributo esistente inSIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT, è necessario aggiornare le azioni relative agli attributi. Dopo aver distribuito l'aggiornamento, il client sarà in grado di verificare i valori esistenti scritti nell'attributo e continuerà a firmare nuovi valori scritti nell'attributo. I nuovi valori scritti nell'attributo verranno inclusi nel contesto di crittografia.

Se si specificano SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT degli attributi, devono esserlo SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT anche gli attributi di partizione e ordinamento.

Utilizzo di una classe di dati annotata

Se hai definito le azioni degli attributi con aTableSchema, aggiorna l'azione dell'attributo associata all'attributo da @DynamoDBEncryptionSignOnly a@DynamoDBEncryptionSignAndIncludeInEncryptionContext.

Utilizzo di un modello a oggetti

Se avete definito manualmente le azioni relative agli attributi, aggiornate l'azione dell'attributo associata all'attributo da SIGN_ONLY a SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT nel modello a oggetti.

## Modificate un **SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT** attributo esistente in **SIGN_ONLY**

Per modificare un SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT attributo esistente inSIGN_ONLY, è necessario aggiornare le azioni relative agli attributi. Dopo aver distribuito l'aggiornamento, il client sarà in grado di verificare i valori esistenti scritti nell'attributo e continuerà a firmare nuovi valori scritti nell'attributo. I nuovi valori scritti nell'attributo non verranno inclusi nel contesto di crittografia.

Prima di modificare un SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT attributo esistente inSIGN_ONLY, valuta attentamente in che modo gli aggiornamenti potrebbero influire sulla funzionalità del fornitore dell'ID della chiave di filiale.

Utilizzo di una classe di dati annotata

Se hai definito le azioni degli attributi con aTableSchema, aggiorna l'azione dell'attributo associata all'attributo da @DynamoDBEncryptionSignAndIncludeInEncryptionContext a@DynamoDBEncryptionSignOnly.

Utilizzo di un modello a oggetti

Se avete definito manualmente le azioni relative agli attributi, aggiornate l'azione dell'attributo associata all'attributo da SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT a SIGN_ONLY nel modello a oggetti.

# AWS Database Encryption SDK per DynamoDB (linguaggi di programmazione disponibili)

Il AWS Database Encryption SDK per DynamoDB è disponibile per i seguenti linguaggi di programmazione. Anche se ogni linguaggio ha delle librerie specifiche, le implementazioni risultanti sono interoperabili. È possibile crittografare con un'implementazione di una lingua e decrittografare con un'altra. L'interoperabilità potrebbe essere soggetta a vincoli linguistici. In tal caso, questi vincoli sono descritti nell'argomento relativo all'implementazione della lingua.

Argomenti



## Java

La nostra libreria di crittografia lato client è stata rinominata Database Encryption SDK. AWS Questa guida per sviluppatori fornisce ancora informazioni sul DynamoDB Encryption Client.

Questo argomento spiega come installare e utilizzare la versione 3. x della libreria di crittografia lato client Java per DynamoDB. Per i dettagli sulla programmazione con AWS Database Encryption SDK per DynamoDB, consulta gli esempi Java nel aws-database-encryption-sdk repository -dynamodb su. GitHub

> **Note**
>
> I seguenti argomenti si concentrano sulla versione 3. x della libreria di crittografia lato client Java per DynamoDB.
> La nostra libreria di crittografia lato client è stata rinominata Database Encryption SDK. AWS Il AWS Database Encryption SDK continua a supportare le versioni precedenti di DynamoDB Encryption Client.

Argomenti

## Prerequisiti

Prima di installare la versione 3. x della libreria di crittografia lato client Java per DynamoDB, assicurati di avere i seguenti prerequisiti.

Un ambiente di sviluppo Java

È necessario Java 8 o versioni successive. Nel sito Web di Oracle, accedi alla pagina Java SE Download, quindi scarica e installa Java SE Development Kit (JDK).

Se utilizzi Oracle JDK, devi scaricare e installare anche Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files.

AWS SDK for Java 2.x

Il AWS Database Encryption SDK per DynamoDB richiede il modulo DynamoDB Enhanced Client di. AWS SDK for Java 2.x Puoi installare l'intero SDK o solo questo modulo.

Per informazioni sull'aggiornamento della versione di AWS SDK per Java, consulta Migrazione dalla versione 1.x alla 2.x di. AWS SDK per Java

AWS SDK per Java È disponibile tramite Apache Maven. È possibile dichiarare una dipendenza per l'intero modulo o solo per il modulo AWS SDK per Java. `dynamodb-enhanced`

Installa AWS SDK per Java utilizzando Apache Maven

- Per importare l'intero AWS SDK per Java come dipendenza, dichiaralo nel file `pom.xml`.
- Per creare una dipendenza solo per il modulo Amazon DynamoDB AWS SDK per Java in, segui le istruzioni per specificare moduli particolari. Imposta il `groupId` to e il to. `software.amazon.awssdk` `artifactID` `dynamodb-enhanced`

> **Note**
>
> Se si utilizza il AWS KMS portachiavi o il portachiavi AWS KMS gerarchico, è inoltre necessario creare una dipendenza per il modulo. AWS KMS Imposta il to e il `groupId` to. `software.amazon.awssdk` `artifactID` `kms`

## Installazione

È possibile installare la versione 3. x della libreria di crittografia lato client Java per DynamoDB nei seguenti modi.

### Utilizzo di Apache Maven

Il client di crittografia Amazon DynamoDB per Java è disponibile tramite Apache Maven con la seguente definizione di dipendenza.

```
<dependency>
  <groupId>software.amazon.cryptography</groupId>
  <artifactId>aws-database-encryption-sdk-dynamodb</artifactId>
  <version>version-number</version>
</dependency>
```

### Usare Gradle Kotlin

Puoi usare Gradle per dichiarare una dipendenza da Amazon DynamoDB Encryption Client for Java aggiungendo quanto segue alla sezione delle dipendenze del tuo progetto Gradle.

```
implementation("software.amazon.cryptography:aws-database-encryption-sdk-
dynamodb:version-number")
```

### Manualmente

Per installare la libreria di crittografia lato client Java per DynamoDB, clona o scarica il repository -dynamodb. aws-database-encryption-sdk GitHub

Dopo aver installato l'SDK, inizia a guardare il codice di esempio in questa guida e gli esempi Java nel repository -dynamodb su. aws-database-encryption-sdk GitHub

# Utilizzo della libreria di crittografia lato client Java per DynamoDB

> La nostra libreria di crittografia lato client è stata rinominata Database Encryption SDK. AWS Questa guida per sviluppatori fornisce ancora informazioni sul DynamoDB Encryption Client.

Questo argomento spiega alcune delle funzioni e delle classi di supporto della versione 3. x della libreria di crittografia lato client Java per DynamoDB.

Per i dettagli sulla programmazione con la libreria di crittografia lato client Java per DynamoDB, consulta gli esempi Java, gli esempi Java nel repository -dynamodb su. aws-database-encryption-sdk GitHub

Argomenti

- Componenti di crittografia dell'item
- Azioni relative agli attributi nel AWS Database Encryption SDK per DynamoDB
- Configurazione della crittografia nel AWS Database Encryption SDK per DynamoDB
- Aggiornamento degli AWS elementi con Database Encryption SDK
- Decrittografia dei set firmati

Componenti di crittografia dell'item

Fondamentalmente, il AWS Database Encryption SDK per DynamoDB è un cifratore di elementi. È possibile utilizzare la versione 3. x della libreria di crittografia lato client Java per DynamoDB per crittografare, firmare, verificare e decrittografare gli elementi della tabella DynamoDB nei seguenti modi.

Il client avanzato per DynamoDB

> È possibile configurare il DynamoDB Enhanced Client per crittografare e firmare automaticamente `DynamoDbEncryptionInterceptor` gli elementi lato client con le richieste DynamoDB. `PutItem` Con DynamoDB Enhanced Client, puoi definire le azioni degli attributi utilizzando una classe di dati annotata. Consigliamo di utilizzare il DynamoDB Enhanced Client ogni volta che è possibile.
>
> Il DynamoDB Enhanced Client non supporta la crittografia ricercabile.

> **Note**
>
> Il AWS Database Encryption SDK non supporta le annotazioni sugli attributi annidati.

L'API DynamoDB di basso livello

Puoi configurare l'API DynamoDB di basso livello per crittografare e firmare automaticamente gli elementi lato client con `DynamoDbEncryptionInterceptor` le tue richieste DynamoDB. `PutItem`

È necessario utilizzare l'API DynamoDB di basso livello per utilizzare la crittografia ricercabile.

Il livello inferiore **`DynamoDbItemEncryptor`**

Il livello inferiore crittografa e firma o decrittografa e verifica `DynamoDbItemEncryptor` direttamente gli elementi della tabella senza chiamare DynamoDB. Non crea DynamoDB o `PutItem` richieste`GetItem`. Ad esempio, puoi utilizzare il livello inferiore per `DynamoDbItemEncryptor` decrittografare e verificare direttamente un elemento DynamoDB che hai già recuperato.

Il livello inferiore non supporta la crittografia ricercabile. `DynamoDbItemEncryptor`

Azioni relative agli attributi nel AWS Database Encryption SDK per DynamoDB

Le azioni relative agli attributi determinano quali valori degli attributi sono crittografati e firmati, quali sono solo firmati, quali sono firmati e inclusi nel contesto di crittografia e quali vengono ignorati.

> **Note**
>
> Per utilizzare l'azione `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` crittografica, è necessario utilizzare la versione 3.3 o successiva di AWS Database Encryption SDK. Distribuisci la nuova versione a tutti i lettori prima di aggiornare il modello di dati per includere. `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`

Se utilizzi l'API DynamoDB di basso livello o il `DynamoDbItemEncryptor` livello inferiore, devi definire manualmente le azioni degli attributi. Se si utilizza il DynamoDB Enhanced Client, è possibile definire manualmente le azioni relative agli attributi oppure utilizzare una classe di dati annotata per generare un. `TableSchema` Per semplificare il processo di configurazione, consigliamo di utilizzare

una classe di dati annotata. Quando utilizzate una classe di dati annotata, dovete modellare l'oggetto una sola volta.

> **Note**
>
> Dopo aver definito le azioni relative agli attributi, è necessario definire quali attributi sono esclusi dalle firme. Per semplificare l'aggiunta di nuovi attributi non firmati in futuro, consigliamo di scegliere un prefisso distinto (ad esempio "`:`«) per identificare gli attributi non firmati. Includi questo prefisso nel nome dell'attributo per tutti gli attributi contrassegnati durante `DO_NOTHING` la definizione dello schema DynamoDB e delle azioni degli attributi.

Utilizza una classe di dati annotata

Utilizza una classe di dati annotata per specificare le azioni degli attributi con DynamoDB Enhanced Client e. `DynamoDbEncryptionInterceptor` Il AWS Database Encryption SDK per DynamoDB utilizza le annotazioni standard degli attributi DynamoDB che definiscono il tipo di attributo per determinare come proteggere un attributo. Per impostazione predefinita, tutti gli attributi sono crittografati e firmati, tranne le chiavi primarie, che sono firmate ma non crittografate.

> **Note**
>
> Per utilizzare l'azione `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` crittografica, è necessario utilizzare la versione 3.3 o successiva del Database Encryption SDK. AWS Distribuisci la nuova versione a tutti i lettori prima di aggiornare il modello di dati per includere. `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`

SimpleClassConsulta.java nel repository aws-database-encryption-sdk -dynamodb su GitHub per ulteriori indicazioni sulle annotazioni di DynamoDB Enhanced Client.

Per impostazione predefinita, gli attributi della chiave primaria sono firmati ma non crittografati (`SIGN_ONLY`) e tutti gli altri attributi sono crittografati e firmati (). `ENCRYPT_AND_SIGN` Se si definiscono gli attributi come`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`, allora anche gli attributi di partizione e ordinamento devono esserlo`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`. Per specificare le eccezioni, utilizzate le annotazioni di crittografia definite nella libreria di crittografia lato client Java per DynamoDB. Ad esempio, se desideri che un particolare

attributo venga firmato, utilizza l'annotazione. `@DynamoDbEncryptionSignOnly` Se desideri che un particolare attributo venga firmato e incluso nel contesto di crittografia, usa il`@DynamoDbEncryptionSignAndIncludeInEncryptionContext`. Se vuoi che un particolare attributo non sia né firmato né crittografato (`DO_NOTHING`), usa l'`@DynamoDbEncryptionDoNothing`annotazione.

> **Note**
>
> Il AWS Database Encryption SDK non supporta le annotazioni sugli attributi annidati.

L'esempio seguente mostra le annotazioni utilizzate per definire e attribuire `ENCRYPT_AND_SIGN` le `SIGN_ONLY` azioni. `DO_NOTHING` Per un esempio che mostra le annotazioni utilizzate per definire`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`, vedete SimpleClass 4.java.

```
@DynamoDbBean
public class SimpleClass {

    private String partitionKey;
    private int sortKey;
    private String attribute1;
    private String attribute2;
    private String attribute3;

    @DynamoDbPartitionKey
    @DynamoDbAttribute(value = "partition_key")
    public String getPartitionKey() {
        return this.partitionKey;
    }

    public void setPartitionKey(String partitionKey) {
        this.partitionKey = partitionKey;
    }

    @DynamoDbSortKey
    @DynamoDbAttribute(value = "sort_key")
    public int getSortKey() {
        return this.sortKey;
    }

    public void setSortKey(int sortKey) {
        this.sortKey = sortKey;
```

```
    }

    public String getAttribute1() {
        return this.attribute1;
    }

    public void setAttribute1(String attribute1) {
        this.attribute1 = attribute1;
    }

    @DynamoDbEncryptionSignOnly
    public String getAttribute2() {
        return this.attribute2;
    }

    public void setAttribute2(String attribute2) {
        this.attribute2 = attribute2;
    }

    @DynamoDbEncryptionDoNothing
    public String getAttribute3() {
        return this.attribute3;
    }

    @DynamoDbAttribute(value = ":attribute3")
    public void setAttribute3(String attribute3) {
        this.attribute3 = attribute3;
    }

}
```

Usa la tua classe di dati annotata per creare il file `TableSchema` come mostrato nel seguente frammento.

```
final TableSchema<SimpleClass> tableSchema = TableSchema.fromBean(SimpleClass.class);
```

Definisci manualmente le azioni degli attributi

Per specificare manualmente le azioni degli attributi, create un `Map` oggetto in cui le coppie nome-valore rappresentino i nomi degli attributi e le azioni specificate.

Specificate `ENCRYPT_AND_SIGN` di crittografare e firmare un attributo.
`SIGN_ONLY`Specificare di firmare, ma non crittografare, un attributo.

`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`Specificare di firmare un attributo e di includerlo nel contesto di crittografia. Non è possibile crittografare un attributo senza firmarlo.
`DO_NOTHING`Specificare di ignorare un attributo.

Gli attributi di partizione e ordinamento devono essere `SIGN_ONLY` o`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`. Se si definiscono gli attributi come`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`, allora anche gli attributi di partizione e ordinamento devono essere uguali. `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`

> **Note**
>
> Per utilizzare l'azione `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` crittografica, è necessario utilizzare la versione 3.3 o successiva del AWS Database Encryption SDK. Distribuisci la nuova versione a tutti i lettori prima di aggiornare il modello di dati per includere. `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`

```
final Map<String, CryptoAction> attributeActionsOnEncrypt = new HashMap<>();
// The partition attribute must be signed
attributeActionsOnEncrypt.put("partition_key",
 CryptoAction.SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT);
// The sort attribute must be signed
attributeActionsOnEncrypt.put("sort_key",
 CryptoAction.SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT);
attributeActionsOnEncrypt.put("attribute1", CryptoAction.ENCRYPT_AND_SIGN);
attributeActionsOnEncrypt.put("attribute2", CryptoAction.SIGN_ONLY);
attributeActionsOnEncrypt.put("attribute3",
 CryptoAction.SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT);
attributeActionsOnEncrypt.put(":attribute4", CryptoAction.DO_NOTHING);
```

Configurazione della crittografia nel AWS Database Encryption SDK per DynamoDB

Quando si utilizza AWS Database Encryption SDK, è necessario definire in modo esplicito una configurazione di crittografia per la tabella DynamoDB. I valori richiesti nella configurazione di crittografia dipendono dal fatto che le azioni degli attributi siano state definite manualmente o con una classe di dati annotata.

Il seguente frammento definisce una configurazione di crittografia delle tabelle DynamoDB utilizzando il DynamoDB Enhanced Client e gli attributi non firmati consentiti definiti da un prefisso `TableSchema`distinto.

```
final Map<String, DynamoDbEnhancedTableEncryptionConfig> tableConfigs = new
 HashMap<>();
tableConfigs.put(ddbTableName,
        DynamoDbEnhancedTableEncryptionConfig.builder()
            .logicalTableName(ddbTableName)
            .keyring(kmsKeyring)
            .allowedUnsignedAttributePrefix(unsignedAttrPrefix)
            .schemaOnEncrypt(tableSchema)
            // Optional: only required if you use beacons
            .search(SearchConfig.builder()
                    .writeVersion(1) // MUST be 1
                    .versions(beaconVersions)
                    .build())
            .build());
```

Nome della tabella logica

Un nome di tabella logica per la tabella DynamoDB.

Il nome della tabella logica è associato crittograficamente a tutti i dati memorizzati nella tabella per semplificare le operazioni di ripristino di DynamoDB. Consigliamo vivamente di specificare il nome della tabella DynamoDB come nome della tabella logica quando si definisce per la prima volta la configurazione di crittografia. È necessario specificare sempre lo stesso nome di tabella logica. Affinché la decrittografia abbia esito positivo, il nome della tabella logica deve corrispondere al nome specificato nella crittografia. Nel caso in cui il nome della tabella DynamoDB cambi dopo il ripristino della tabella DynamoDB da un backup, il nome della tabella logica assicura che l'operazione di decrittografia riconosca ancora la tabella.

Attributi non firmati consentiti

Gli attributi contrassegnati `DO_NOTHING` nelle azioni relative agli attributi.

Gli attributi non firmati consentiti indicano al client quali attributi sono esclusi dalle firme. Il client presume che tutti gli altri attributi siano inclusi nella firma. Quindi, durante la decrittografia di un record, il client determina quali attributi deve verificare e quali ignorare tra gli attributi non firmati consentiti specificati. Non è possibile rimuovere un attributo dagli attributi non firmati consentiti.

È possibile definire gli attributi non firmati consentiti in modo esplicito creando un array che elenca tutti gli attributi. `DO_NOTHING` È inoltre possibile specificare un prefisso distinto quando si assegnano nomi `DO_NOTHING` agli attributi e utilizzare il prefisso per indicare al client quali attributi non sono firmati. Consigliamo vivamente di specificare un prefisso distinto perché

semplifica il processo di aggiunta di un nuovo `DO_NOTHING` attributo in futuro. Per ulteriori informazioni, consulta Aggiornamento del modello di dati.

Se non si specifica un prefisso per tutti gli `DO_NOTHING` attributi, è possibile configurare un `allowedUnsignedAttributes` array che elenchi in modo esplicito tutti gli attributi che il client dovrebbe aspettarsi che non siano firmati quando li incontra durante la decrittografia. È necessario definire in modo esplicito gli attributi non firmati consentiti solo se assolutamente necessario.

Configurazione della ricerca (opzionale)

`SearchConfig`definisce la versione del beacon.

È `SearchConfig` necessario specificare il per utilizzare la crittografia ricercabile o i beacon firmati.

Algorithm Suite (opzionale)

`algorithmSuiteId`Definisce la suite di algoritmi utilizzata da AWS Database Encryption SDK.

A meno che non si specifichi esplicitamente una suite di algoritmi alternativa, AWS Database Encryption SDK utilizza la suite di algoritmi predefinita. La suite di algoritmi predefinita utilizza l'algoritmo AES-GCM con derivazione delle chiavi, firme digitali e impegno delle chiavi. Sebbene la suite di algoritmi predefinita sia probabilmente adatta alla maggior parte delle applicazioni, è possibile scegliere una suite di algoritmi alternativa. Ad esempio, alcuni modelli di fiducia sarebbero soddisfatti da una suite di algoritmi senza firme digitali. Per informazioni sulle suite di algoritmi supportate da AWS Database Encryption SDK, consulta. Suite di algoritmi supportate nel AWS Database Encryption SDK

Per selezionare la suite di algoritmi AES-GCM senza firme digitali ECDSA, includi il seguente frammento nella configurazione di crittografia delle tabelle.

```
.algorithmSuiteId(
    DBEAlgorithmSuiteId.ALG_AES_256_GCM_HKDF_SHA512_COMMIT_KEY_SYMSIG_HMAC_SHA384)
```

Aggiornamento degli AWS elementi con Database Encryption SDK

Il AWS Database Encryption SDK non supporta ddb: UpdateItem per gli elementi che sono stati crittografati o firmati. Per aggiornare un elemento crittografato o firmato, è necessario utilizzare ddb:. PutItem Quando specifichi la stessa chiave primaria di un elemento esistente nella tua

`PutItem` richiesta, il nuovo elemento sostituisce completamente l'elemento esistente. Puoi anche usare CLOBBER per cancellare e sostituire tutti gli attributi al momento del salvataggio dopo aver aggiornato i tuoi articoli.

Decrittografia dei set firmati

Nelle versioni 3.0.0 e 3.1.0 del AWS Database Encryption SDK, se si definisce un attributo di tipo set come`SIGN_ONLY`, i valori del set vengono canonicalizzati nell'ordine in cui vengono forniti. DynamoDB non mantiene l'ordine dei set. Di conseguenza, c'è la possibilità che la convalida della firma dell'elemento che contiene il set abbia esito negativo. La convalida della firma ha esito negativo quando i valori del set vengono restituiti in un ordine diverso da quello fornito al AWS Database Encryption SDK, anche se gli attributi del set contengono gli stessi valori.

> **Note**
>
> Le versioni 3.1.1 e successive del AWS Database Encryption SDK canonicalizzano i valori di tutti gli attributi dei tipi di set, in modo che i valori vengano letti nello stesso ordine in cui sono stati scritti in DynamoDB.

Se la convalida della firma fallisce, l'operazione di decrittografia fallisce e restituisce il seguente messaggio di errore.

```
software.amazon.cryptography.dbencryptionsdk.structuredencryption.model. StructuredEncrypti
onException: Nessun tag del destinatario corrispondente.
```

Se ricevete il messaggio di errore riportato sopra e ritenete che l'elemento che state cercando di decrittografare includa un set firmato utilizzando la versione 3.0.0 o 3.1.0, consultate la DecryptWithPermutedirectory del repository aws-database-encryption-sdk -dynamodb-java su per i dettagli su come convalidare correttamente il set. GitHub

## Esempi di Java

La nostra libreria di crittografia lato client è stata rinominata Database Encryption SDK. AWS Questa guida per sviluppatori fornisce ancora informazioni sul DynamoDB Encryption Client.

Gli esempi seguenti mostrano come utilizzare la libreria di crittografia lato client Java per DynamoDB per proteggere gli elementi della tabella nell'applicazione. Puoi trovare altri esempi (e contribuire con i tuoi) negli esempi Java nel repository -dynamodb su. aws-database-encryption-sdk GitHub

Gli esempi seguenti mostrano come configurare la libreria di crittografia lato client Java per DynamoDB in una nuova tabella Amazon DynamoDB non popolata. Se desideri configurare le tabelle Amazon DynamoDB esistenti per la crittografia lato client, consulta. Aggiungi la versione 3.x a una tabella esistente

Argomenti

- Utilizzo del client avanzato DynamoDB
- Utilizzo dell'API DynamoDB di basso livello
- Utilizzo del livello inferiore DynamoDbItemEncryptor

Utilizzo del client avanzato DynamoDB

L'esempio seguente mostra come utilizzare il DynamoDB Enhanced Client `DynamoDbEncryptionInterceptor` e AWS KMS un portachiavi per crittografare gli elementi della tabella DynamoDB come parte delle chiamate API DynamoDB.

Puoi utilizzare qualsiasi portachiavi supportato con DynamoDB Enhanced Client, ma consigliamo di utilizzare uno dei AWS KMS portachiavi quando possibile.

> **Note**
>
> Il DynamoDB Enhanced Client non supporta la crittografia ricercabile. Utilizza l'`DynamoDbEncryptionInterceptor`API DynamoDB di basso livello per utilizzare la crittografia ricercabile.

Guarda l'esempio di codice completo: .java EnhancedPutGetExample

Fase 1: Creare il portachiavi AWS KMS

L'esempio seguente utilizza la creazione `CreateAwsKmsMrkMultiKeyring` di un AWS KMS portachiavi con una chiave KMS di crittografia simmetrica. Il `CreateAwsKmsMrkMultiKeyring` metodo garantisce che il portachiavi gestisca correttamente sia le chiavi a regione singola che quelle a più regioni.

```
final MaterialProviders matProv = MaterialProviders.builder()
        .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
        .build();
final CreateAwsKmsMrkMultiKeyringInput keyringInput =
 CreateAwsKmsMrkMultiKeyringInput.builder()
        .generator(kmsKeyId)
        .build();
final IKeyring kmsKeyring = matProv.CreateAwsKmsMrkMultiKeyring(keyringInput);
```

Passaggio 2: creare uno schema tabellare dalla classe di dati annotata

L'esempio seguente utilizza la classe di dati annotati per creare il. `TableSchema`

Questo esempio presuppone che la classe di dati annotata e le azioni degli attributi siano state definite utilizzando .java. SimpleClass Per ulteriori informazioni sull'annotazione delle azioni relative agli attributi, consulta. Utilizza una classe di dati annotata

> **Note**
>
> Il AWS Database Encryption SDK non supporta le annotazioni sugli attributi annidati.

```
final TableSchema<SimpleClass> schemaOnEncrypt =
 TableSchema.fromBean(SimpleClass.class);
```

Fase 3: Definire quali attributi sono esclusi dalle firme

L'esempio seguente presuppone che tutti `DO_NOTHING` gli attributi condividano il prefisso distinto `":`«e utilizza il prefisso per definire gli attributi non firmati consentiti. Il client presuppone che qualsiasi nome di attributo con il prefisso `":"` sia escluso dalle firme. Per ulteriori informazioni, consulta Allowed unsigned attributes.

```
final String unsignedAttrPrefix = ":";
```

Fase 4: Creare la configurazione di crittografia

L'esempio seguente definisce una `tableConfigs` mappa che rappresenta la configurazione di crittografia per la tabella DynamoDB.

Questo esempio specifica il nome della tabella DynamoDB come nome della tabella logica. Consigliamo vivamente di specificare il nome della tabella DynamoDB come nome della tabella

logica quando si definisce per la prima volta la configurazione di crittografia. Per ulteriori informazioni, consulta Configurazione della crittografia nel AWS Database Encryption SDK per DynamoDB.

> **Note**
>
> Per utilizzare la crittografia ricercabile o i beacon firmati, è necessario includerli anche nella configurazione di crittografia. `SearchConfig`

```
final Map<String, DynamoDbEnhancedTableEncryptionConfig> tableConfigs = new
 HashMap<>();
tableConfigs.put(ddbTableName,
    DynamoDbEnhancedTableEncryptionConfig.builder()
        .logicalTableName(ddbTableName)
        .keyring(kmsKeyring)
        .allowedUnsignedAttributePrefix(unsignedAttrPrefix)
        .schemaOnEncrypt(tableSchema)
        .build());
```

Fase 5: Crea il **`DynamoDbEncryptionInterceptor`**

L'esempio seguente ne crea uno nuovo `DynamoDbEncryptionInterceptor` con il `tableConfigs` passo 4.

```
final DynamoDbEncryptionInterceptor interceptor =
    DynamoDbEnhancedClientEncryption.CreateDynamoDbEncryptionInterceptor(
        CreateDynamoDbEncryptionInterceptorInput.builder()
            .tableEncryptionConfigs(tableConfigs)
            .build()
    );
```

Fase 6: Creare un nuovo client AWS SDK DynamoDB

L'esempio seguente crea un nuovo client AWS SDK DynamoDB utilizzando **`interceptor`** lo Step 5.

```
final DynamoDbClient ddb = DynamoDbClient.builder()
        .overrideConfiguration(
                ClientOverrideConfiguration.builder()
```

```
                        .addExecutionInterceptor(interceptor)
                        .build())
        .build();
```

Fase 7: Creare il DynamoDB Enhanced Client e creare una tabella

L'esempio seguente crea il DynamoDB Enhanced Client utilizzando il client DynamoDB AWS SDK creato nel passaggio 6 e crea una tabella utilizzando la classe di dati annotati.

```
final DynamoDbEnhancedClient enhancedClient = DynamoDbEnhancedClient.builder()
        .dynamoDbClient(ddb)
        .build();
final DynamoDbTable<SimpleClass> table = enhancedClient.table(ddbTableName,
 tableSchema);
```

Fase 8: Crittografare e firmare un elemento della tabella

L'esempio seguente inserisce un elemento nella tabella DynamoDB utilizzando il DynamoDB Enhanced Client. L'elemento viene crittografato e firmato lato client prima di essere inviato a DynamoDB.

```
final SimpleClass item = new SimpleClass();
item.setPartitionKey("EnhancedPutGetExample");
item.setSortKey(0);
item.setAttribute1("encrypt and sign me!");
item.setAttribute2("sign me!");
item.setAttribute3("ignore me!");

table.putItem(item);
```

Utilizzo dell'API DynamoDB di basso livello

L'esempio seguente mostra come utilizzare l'API DynamoDB di basso livello con AWS KMS un portachiavi per crittografare e firmare automaticamente gli elementi lato client con le richieste DynamoDB. `PutItem`

Puoi utilizzare qualsiasi portachiavi supportato, ma ti consigliamo di utilizzare uno dei portachiavi AWS KMS quando possibile.

Guarda l'esempio di codice completo: .java BasicPutGetExample

### Fase 1: Creare il portachiavi AWS KMS

L'esempio seguente utilizza la creazione `CreateAwsKmsMrkMultiKeyring` di un AWS KMS portachiavi con una chiave KMS di crittografia simmetrica. Il `CreateAwsKmsMrkMultiKeyring` metodo garantisce che il portachiavi gestisca correttamente sia le chiavi a regione singola che quelle a più regioni.

```
final MaterialProviders matProv = MaterialProviders.builder()
         .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
         .build();
final CreateAwsKmsMrkMultiKeyringInput keyringInput =
 CreateAwsKmsMrkMultiKeyringInput.builder()
        .generator(kmsKeyId)
        .build();
final IKeyring kmsKeyring = matProv.CreateAwsKmsMrkMultiKeyring(keyringInput);
```

### Passaggio 2: configura le azioni relative agli attributi

L'esempio seguente definisce una `attributeActionsOnEncrypt` mappa che rappresenta azioni di esempio relative agli attributi per un elemento della tabella.

> **Note**
>
> L'esempio seguente non definisce alcun attributo come`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`. Se si specifica un `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` attributo, devono esserlo `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` anche gli attributi di partizione e ordinamento.

```
final Map<String, CryptoAction> attributeActionsOnEncrypt = new HashMap<>();
// The partition attribute must be SIGN_ONLY
attributeActionsOnEncrypt.put("partition_key", CryptoAction.SIGN_ONLY);
// The sort attribute must be SIGN_ONLY
attributeActionsOnEncrypt.put("sort_key", CryptoAction.SIGN_ONLY);
attributeActionsOnEncrypt.put("attribute1", CryptoAction.ENCRYPT_AND_SIGN);
attributeActionsOnEncrypt.put("attribute2", CryptoAction.SIGN_ONLY);
attributeActionsOnEncrypt.put(":attribute3", CryptoAction.DO_NOTHING);
```

Fase 3: Definire quali attributi sono esclusi dalle firme

L'esempio seguente presuppone che tutti `DO_NOTHING` gli attributi condividano il prefisso distinto "`:`«e utilizza il prefisso per definire gli attributi non firmati consentiti. Il client presuppone che qualsiasi nome di attributo con il prefisso "`:`" sia escluso dalle firme. Per ulteriori informazioni, consulta Allowed unsigned attributes.

```
final String unsignedAttrPrefix = ":";
```

Fase 4: Definire la configurazione di crittografia delle tabelle DynamoDB

L'esempio seguente definisce una `tableConfigs` mappa che rappresenta la configurazione di crittografia per questa tabella DynamoDB.

Questo esempio specifica il nome della tabella DynamoDB come nome della tabella logica. Consigliamo vivamente di specificare il nome della tabella DynamoDB come nome della tabella logica quando si definisce per la prima volta la configurazione di crittografia. Per ulteriori informazioni, consulta Configurazione della crittografia nel AWS Database Encryption SDK per DynamoDB.

> **Note**
>
> Per utilizzare la crittografia ricercabile o i beacon firmati, è necessario includerli anche nella configurazione di crittografia. `SearchConfig`

```
final Map<String, DynamoDbTableEncryptionConfig> tableConfigs = new HashMap<>();
final DynamoDbTableEncryptionConfig config = DynamoDbTableEncryptionConfig.builder()
        .logicalTableName(ddbTableName)
        .partitionKeyName("partition_key")
        .sortKeyName("sort_key")
        .attributeActionsOnEncrypt(attributeActionsOnEncrypt)
        .keyring(kmsKeyring)
        .allowedUnsignedAttributePrefix(unsignedAttrPrefix)
        .build();
tableConfigs.put(ddbTableName, config);
```

Fase 5: Creare il **`DynamoDbEncryptionInterceptor`**

L'esempio seguente crea l'`DynamoDbEncryptionInterceptor`utilizzo del `tableConfigs` dal passaggio 4.

```
DynamoDbEncryptionInterceptor interceptor = DynamoDbEncryptionInterceptor.builder()
        .config(DynamoDbTablesEncryptionConfig.builder()
                .tableEncryptionConfigs(tableConfigs)
                .build())
        .build();
```

Fase 6: Creare un nuovo client AWS SDK DynamoDB

L'esempio seguente crea un nuovo client AWS SDK DynamoDB utilizzando **`interceptor`** lo Step 5.

```
final DynamoDbClient ddb = DynamoDbClient.builder()
        .overrideConfiguration(
                ClientOverrideConfiguration.builder()
                        .addExecutionInterceptor(interceptor)
                        .build())
        .build();
```

Fase 7: Crittografare e firmare un elemento della tabella DynamoDB

L'esempio seguente definisce una `item` mappa che rappresenta un elemento della tabella di esempio e inserisce l'elemento nella tabella DynamoDB. L'elemento viene crittografato e firmato lato client prima di essere inviato a DynamoDB.

```
final HashMap<String, AttributeValue> item = new HashMap<>();
item.put("partition_key", AttributeValue.builder().s("BasicPutGetExample").build());
item.put("sort_key", AttributeValue.builder().n("0").build());
item.put("attribute1", AttributeValue.builder().s("encrypt and sign me!").build());
item.put("attribute2", AttributeValue.builder().s("sign me!").build());
item.put(":attribute3", AttributeValue.builder().s("ignore me!").build());

final PutItemRequest putRequest = PutItemRequest.builder()
        .tableName(ddbTableName)
        .item(item)
        .build();

final PutItemResponse putResponse = ddb.putItem(putRequest);
```

### Utilizzo del livello inferiore DynamoDbItemEncryptor

L'esempio seguente mostra come utilizzare il livello inferiore `DynamoDbItemEncryptor` con un AWS KMS portachiavi per crittografare e firmare direttamente gli elementi della tabella. Non `DynamoDbItemEncryptor` inserisce l'elemento nella tabella DynamoDB.

Puoi utilizzare qualsiasi portachiavi supportato con DynamoDB Enhanced Client, ma consigliamo di utilizzare uno dei AWS KMS portachiavi quando possibile.

> **Note**
>
> Il livello inferiore `DynamoDbItemEncryptor` non supporta la crittografia ricercabile. Utilizza l'`DynamoDbEncryptionInterceptor`API DynamoDB di basso livello per utilizzare la crittografia ricercabile.

Guarda l'esempio di codice completo: .java ItemEncryptDecryptExample

#### Fase 1: Creare il portachiavi AWS KMS

L'esempio seguente utilizza la creazione `CreateAwsKmsMrkMultiKeyring` di un AWS KMS portachiavi con una chiave KMS di crittografia simmetrica. Il `CreateAwsKmsMrkMultiKeyring` metodo garantisce che il portachiavi gestisca correttamente sia le chiavi a regione singola che quelle a più regioni.

```
final MaterialProviders matProv = MaterialProviders.builder()
         .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
         .build();
final CreateAwsKmsMrkMultiKeyringInput keyringInput =
 CreateAwsKmsMrkMultiKeyringInput.builder()
        .generator(kmsKeyId)
        .build();
final IKeyring kmsKeyring = matProv.CreateAwsKmsMrkMultiKeyring(keyringInput);
```

#### Passaggio 2: configura le azioni relative agli attributi

L'esempio seguente definisce una `attributeActionsOnEncrypt` mappa che rappresenta azioni di esempio relative agli attributi per un elemento della tabella.

> **Note**
>
> L'esempio seguente non definisce alcun attributo come`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`. Se si specifica un `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` attributo, devono esserlo `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` anche gli attributi di partizione e ordinamento.

```
final Map<String, CryptoAction> attributeActionsOnEncrypt = new HashMap<>();
// The partition attribute must be SIGN_ONLY
attributeActionsOnEncrypt.put("partition_key", CryptoAction.SIGN_ONLY);
// The sort attribute must be SIGN_ONLY
attributeActionsOnEncrypt.put("sort_key", CryptoAction.SIGN_ONLY);
attributeActionsOnEncrypt.put("attribute1", CryptoAction.ENCRYPT_AND_SIGN);
attributeActionsOnEncrypt.put("attribute2", CryptoAction.SIGN_ONLY);
attributeActionsOnEncrypt.put(":attribute3", CryptoAction.DO_NOTHING);
```

Fase 3: Definire quali attributi sono esclusi dalle firme

L'esempio seguente presuppone che tutti `DO_NOTHING` gli attributi condividano il prefisso distinto "`:`«e utilizza il prefisso per definire gli attributi non firmati consentiti. Il client presuppone che qualsiasi nome di attributo con il prefisso "`:`" sia escluso dalle firme. Per ulteriori informazioni, consulta Allowed unsigned attributes.

```
final String unsignedAttrPrefix = ":";
```

Fase 4: Definire la configurazione **`DynamoDbItemEncryptor`**

L'esempio seguente definisce la configurazione per`DynamoDbItemEncryptor`.

Questo esempio specifica il nome della tabella DynamoDB come nome della tabella logica. Consigliamo vivamente di specificare il nome della tabella DynamoDB come nome della tabella logica quando si definisce per la prima volta la configurazione di crittografia. Per ulteriori informazioni, consulta Configurazione della crittografia nel AWS Database Encryption SDK per DynamoDB.

```
final DynamoDbItemEncryptorConfig config = DynamoDbItemEncryptorConfig.builder()
        .logicalTableName(ddbTableName)
```

```
        .partitionKeyName("partition_key")
        .sortKeyName("sort_key")
        .attributeActionsOnEncrypt(attributeActionsOnEncrypt)
        .keyring(kmsKeyring)
        .allowedUnsignedAttributePrefix(unsignedAttrPrefix)
        .build();
```

Fase 5: Creare il **`DynamoDbItemEncryptor`**

L'esempio seguente ne crea uno nuovo `DynamoDbItemEncryptor` utilizzando il `config` tratto dal passaggio 4.

```
final DynamoDbItemEncryptor itemEncryptor = DynamoDbItemEncryptor.builder()
        .DynamoDbItemEncryptorConfig(config)
        .build();
```

Fase 6: Crittografare e firmare direttamente un elemento della tabella

L'esempio seguente crittografa e firma direttamente un elemento utilizzando il. `DynamoDbItemEncryptor` Non `DynamoDbItemEncryptor` inserisce l'elemento nella tabella DynamoDB.

```
final Map<String, AttributeValue> originalItem = new HashMap<>();
originalItem.put("partition_key",
 AttributeValue.builder().s("ItemEncryptDecryptExample").build());
originalItem.put("sort_key", AttributeValue.builder().n("0").build());
originalItem.put("attribute1", AttributeValue.builder().s("encrypt and sign
 me!").build());
originalItem.put("attribute2", AttributeValue.builder().s("sign me!").build());
originalItem.put(":attribute3", AttributeValue.builder().s("ignore me!").build());

final Map<String, AttributeValue> encryptedItem = itemEncryptor.EncryptItem(
        EncryptItemInput.builder()
                .plaintextItem(originalItem)
                .build()
).encryptedItem();
```

# Configurare una tabella DynamoDB esistente per utilizzare AWS il Database Encryption SDK per DynamoDB

La nostra libreria di crittografia lato client è stata rinominata Database Encryption SDK. AWS Questa guida per sviluppatori fornisce ancora informazioni sul DynamoDB Encryption Client.

Con la versione 3. x della libreria di crittografia lato client Java per DynamoDB, puoi configurare le tabelle Amazon DynamoDB esistenti per la crittografia lato client. Questo argomento fornisce indicazioni sui tre passaggi da eseguire per aggiungere la versione 3. x su una tabella DynamoDB esistente e popolata.

Prerequisiti

Versione 3. x della libreria di crittografia lato client Java per DynamoDB richiede il DynamoDB Enhanced Client fornito in. AWS SDK for Java 2.x Se usi ancora Dynamo DBMapper, devi migrare per utilizzare il DynamoDB AWS SDK for Java 2.x Enhanced Client.

Segui le istruzioni per la migrazione dalla versione 1.x alla 2.x di. AWS SDK per Java

Quindi, segui le istruzioni per iniziare a utilizzare l'API DynamoDB Enhanced Client.

Prima di configurare la tabella per utilizzare la libreria di crittografia lato client Java per DynamoDB, è necessario generare una classe di dati `TableSchema`con annotazioni e creare un client avanzato.

Fase 1: Prepararsi a leggere e scrivere elementi crittografati

Completa i seguenti passaggi per preparare il client AWS Database Encryption SDK per leggere e scrivere elementi crittografati. Dopo aver implementato le seguenti modifiche, il client continuerà a leggere e scrivere elementi in testo non crittografato. Non crittograferà né firmerà i nuovi elementi scritti nella tabella, ma sarà in grado di decrittografare gli elementi crittografati non appena vengono visualizzati. Queste modifiche preparano il client a iniziare a crittografare nuovi elementi. Le seguenti modifiche devono essere implementate su ciascun lettore prima di procedere al passaggio successivo.

1. Definite le azioni relative agli attributi

   Aggiorna la tua classe di dati annotata per includere azioni di attributo che definiscono quali valori degli attributi saranno crittografati e firmati, quali saranno solo firmati e quali verranno ignorati.

Consulta il SimpleClassfile.java nel repository aws-database-encryption-sdk -dynamodb GitHub per ulteriori informazioni sulle annotazioni di DynamoDB Enhanced Client.

Per impostazione predefinita, gli attributi della chiave primaria sono firmati ma non crittografati (`SIGN_ONLY`) e tutti gli altri attributi sono crittografati e firmati (). `ENCRYPT_AND_SIGN` Per specificare le eccezioni, utilizzate le annotazioni di crittografia definite nella libreria di crittografia lato client Java per DynamoDB. Ad esempio, se vuoi che un particolare attributo sia solo segno, usa l'annotazione. `@DynamoDbEncryptionSignOnly` Se desideri che un particolare attributo sia firmato e incluso nel contesto di crittografia, usa l'`@DynamoDbEncryptionSignAndIncludeInEncryptionContext`annotazione. Se vuoi che un particolare attributo non sia né firmato né crittografato (`DO_NOTHING`), usa l'`@DynamoDbEncryptionDoNothing`annotazione.

> **Note**
>
> Se specificate degli `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` attributi, devono esserlo anche gli attributi di partizione e ordinamento. `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` Per un esempio che mostra le annotazioni utilizzate per definire`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`, vedete SimpleClass 4.java.

Per esempio annotazioni, vedere. Utilizza una classe di dati annotata

2. Definire quali attributi verranno esclusi dalle firme

   L'esempio seguente presuppone che tutti `DO_NOTHING` gli attributi condividano il prefisso distinto "`:`«e utilizza il prefisso per definire gli attributi non firmati consentiti. Il client presumerà che qualsiasi nome di attributo con il prefisso "`:`" sia escluso dalle firme. Per ulteriori informazioni, consulta Allowed unsigned attributes.

   ```
   final String unsignedAttrPrefix = ":";
   ```

3. Crea un portachiavi

   L'esempio seguente crea un AWS KMS portachiavi. Il AWS KMS portachiavi utilizza la crittografia simmetrica o RSA asimmetrica per generare, crittografare e AWS KMS keys decrittografare le chiavi di dati.

Questo esempio utilizza per creare un portachiavi con una chiave KMS di crittografia `CreateMrkMultiKeyring` simmetrica. AWS KMS Il `CreateAwsKmsMrkMultiKeyring` metodo garantisce che il portachiavi gestisca correttamente sia le chiavi a regione singola che quelle a più regioni.

```
final MaterialProviders matProv = MaterialProviders.builder()
        .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
        .build();
final CreateAwsKmsMrkMultiKeyringInput keyringInput =
 CreateAwsKmsMrkMultiKeyringInput.builder()
        .generator(kmsKeyId)
        .build();
final IKeyring kmsKeyring = matProv.CreateAwsKmsMrkMultiKeyring(keyringInput);
```

4. Definire la configurazione di crittografia delle tabelle DynamoDB

L'esempio seguente definisce una `tableConfigs` mappa che rappresenta la configurazione di crittografia per questa tabella DynamoDB.

Questo esempio specifica il nome della tabella DynamoDB come nome della tabella logica. Consigliamo vivamente di specificare il nome della tabella DynamoDB come nome della tabella logica quando si definisce per la prima volta la configurazione di crittografia. Per ulteriori informazioni, consulta Configurazione della crittografia nel AWS Database Encryption SDK per DynamoDB.

È necessario specificare `FORCE_WRITE_PLAINTEXT_ALLOW_READ_PLAINTEXT` come override in testo semplice. Questa politica continua a leggere e scrivere elementi in testo non crittografato, legge gli elementi crittografati e prepara il client alla scrittura di elementi crittografati.

```
final Map<String, DynamoDbTableEncryptionConfig> tableConfigs = new HashMap<>();
final DynamoDbTableEncryptionConfig config = DynamoDbTableEncryptionConfig.builder()
        .logicalTableName(ddbTableName)
        .partitionKeyName("partition_key")
        .sortKeyName("sort_key")
        .schemaOnEncrypt(tableSchema)
        .keyring(kmsKeyring)
        .allowedUnsignedAttributePrefix(unsignedAttrPrefix)

 .plaintextOverride(PlaintextOverride.FORCE_WRITE_PLAINTEXT_ALLOW_READ_PLAINTEXT)
        .build();
tableConfigs.put(ddbTableName, config);
```

5. Creazione del **`DynamoDbEncryptionInterceptor`**

L'esempio seguente crea l'**`DynamoDbEncryptionInterceptor`**utilizzo del dal passaggio 3. **`tableConfigs`**

```
DynamoDbEncryptionInterceptor interceptor = DynamoDbEncryptionInterceptor.builder()
        .config(DynamoDbTablesEncryptionConfig.builder()
                .tableEncryptionConfigs(tableConfigs)
                .build())
        .build();
```

Fase 2: Scrivere elementi crittografati e firmati

Aggiorna la politica del testo in chiaro nella tua `DynamoDbEncryptionInterceptor` configurazione per consentire al client di scrivere elementi crittografati e firmati. Dopo aver implementato la seguente modifica, il client crittograferà e firmerà i nuovi elementi in base alle azioni degli attributi configurate nel passaggio 1. Il client sarà in grado di leggere elementi in testo semplice e elementi crittografati e firmati.

Prima di procedere alla Fase 3, è necessario crittografare e firmare tutti gli elementi di testo in chiaro esistenti nella tabella. Non è possibile eseguire alcuna metrica o query per crittografare rapidamente gli elementi di testo in chiaro esistenti. Utilizzate il processo più adatto al vostro sistema. Ad esempio, è possibile utilizzare un processo asincrono che scansiona lentamente la tabella e quindi riscrive gli elementi utilizzando le azioni degli attributi e la configurazione di crittografia definite. Per identificare gli elementi di testo in chiaro nella tabella, consigliamo di cercare tutti gli elementi che non contengono gli `aws_dbe_foot` attributi `aws_dbe_head` e che AWS Database Encryption SDK aggiunge agli elementi quando sono crittografati e firmati.

L'esempio seguente aggiorna la configurazione di crittografia della tabella dal passaggio 1. È necessario aggiornare l'override del testo in chiaro con. `FORBID_WRITE_PLAINTEXT_ALLOW_READ_PLAINTEXT` Questo criterio continua a leggere gli elementi di testo in chiaro, ma legge e scrive anche gli elementi crittografati. Creane uno nuovo `DynamoDbEncryptionInterceptor` utilizzando quello aggiornato. `tableConfigs`

```
final Map<String, DynamoDbTableEncryptionConfig> tableConfigs = new HashMap<>();
final DynamoDbTableEncryptionConfig config = DynamoDbTableEncryptionConfig.builder()
        .logicalTableName(ddbTableName)
        .partitionKeyName("partition_key")
        .sortKeyName("sort_key")
```

```
        .schemaOnEncrypt(tableSchema)
        .keyring(kmsKeyring)
        .allowedUnsignedAttributePrefix(unsignedAttrPrefix)

 .plaintextOverride(PlaintextOverride.FORBID_WRITE_PLAINTEXT_ALLOW_READ_PLAINTEXT)
        .build();
tableConfigs.put(ddbTableName, config);
```

Passaggio 3: Leggi solo gli elementi crittografati e firmati

Dopo aver crittografato e firmato tutti gli articoli, aggiorna l'override del testo in chiaro nella `DynamoDbEncryptionInterceptor` configurazione per consentire al client di leggere e scrivere solo gli elementi crittografati e firmati. Dopo aver implementato la seguente modifica, il client crittograferà e firmerà i nuovi elementi in base alle azioni degli attributi configurate nel passaggio 1. Il client sarà in grado di leggere solo gli elementi crittografati e firmati.

L'esempio seguente aggiorna la configurazione di crittografia delle tabelle dal passaggio 2. È possibile aggiornare l'override del testo in chiaro con `FORBID_WRITE_PLAINTEXT_FORBID_READ_PLAINTEXT` o rimuovere la politica del testo in chiaro dalla configurazione. Per impostazione predefinita, il client legge e scrive solo elementi crittografati e firmati. Creane uno nuovo `DynamoDbEncryptionInterceptor` utilizzando quello aggiornato`tableConfigs`.

```
final Map<String, DynamoDbTableEncryptionConfig> tableConfigs = new HashMap<>();
final DynamoDbTableEncryptionConfig config = DynamoDbTableEncryptionConfig.builder()
        .logicalTableName(ddbTableName)
        .partitionKeyName("partition_key")
        .sortKeyName("sort_key")
        .schemaOnEncrypt(tableSchema)
        .keyring(kmsKeyring)
        .allowedUnsignedAttributePrefix(unsignedAttrPrefix)
        // Optional: you can also remove the plaintext policy from your configuration

 .plaintextOverride(PlaintextOverride.FORBID_WRITE_PLAINTEXT_FORBID_READ_PLAINTEXT)
        .build();
tableConfigs.put(ddbTableName, config);
```

# Esegui la migrazione alla versione 3.x della libreria di crittografia lato client Java per DynamoDB

La nostra libreria di crittografia lato client è stata rinominata AWS Database Encryption SDK. Questa guida per sviluppatori fornisce ancora informazioni sul DynamoDB Encryption Client.

Versione 3. x della libreria di crittografia lato client Java per DynamoDB è una riscrittura importante della 2. x codice base. Include molti aggiornamenti, come un nuovo formato di dati strutturati, un supporto multitenancy migliorato, modifiche dello schema senza interruzioni e supporto per la crittografia ricercabile. Questo argomento fornisce indicazioni su come migrare il codice alla versione 3. x.

Migrazione dalla versione 1.x alla 2.x

Migrare alla versione 2. x prima di migrare alla versione 3. x. Versione 2. x ha cambiato il simbolo del provider più recente da `MostRecentProvider` a`CachingMostRecentProvider`. Se attualmente utilizzi la versione 1. x della libreria di crittografia lato client Java per DynamoDB con il `MostRecentProvider` simbolo, è necessario aggiornare il nome del simbolo nel codice su. `CachingMostRecentProvider` Per ulteriori informazioni, consulta Updates to the Most Recent Provider.

Migrazione dalla versione 2.x alla 3.x

Le seguenti procedure descrivono come migrare il codice dalla versione 2. x alla versione 3. x della libreria di crittografia lato client Java per DynamoDB.

Fase 1: Preparati a leggere gli articoli nel nuovo formato

Completa i seguenti passaggi per preparare il client AWS Database Encryption SDK alla lettura degli elementi nel nuovo formato. Dopo aver implementato le seguenti modifiche, il client continuerà a comportarsi nello stesso modo in cui si comportava nella versione 2. x. Il tuo client continuerà a leggere e scrivere elementi nella versione 2. formato x, ma queste modifiche preparano il client a leggere gli elementi nel nuovo formato.

Aggiorna il tuo AWS SDK per Java alla versione 2.x

Versione 3. x della libreria di crittografia lato client Java per DynamoDB richiede DynamoDB Enhanced Client. Il DynamoDB Enhanced Client sostituisce il DBMapper Dynamo utilizzato nelle

versioni precedenti. Per utilizzare il client avanzato, è necessario utilizzare il. AWS SDK for Java 2.x

Segui le istruzioni per la migrazione dalla versione 1.x alla 2.x di. AWS SDK per Java

Per ulteriori informazioni sui AWS SDK for Java 2.x moduli richiesti, vedere. Prerequisiti

Configura il tuo client per leggere gli elementi crittografati dalle versioni precedenti

Le seguenti procedure forniscono una panoramica dei passaggi illustrati nell'esempio di codice riportato di seguito.

1. Crea un portachiavi.

   I portachiavi e i gestori di materiali crittografici sostituiscono i fornitori di materiali crittografici utilizzati nelle versioni precedenti della libreria di crittografia lato client Java per DynamoDB.

   > **Important**
   >
   > Le chiavi di wrapping specificate durante la creazione di un portachiavi devono essere le stesse chiavi di wrapping utilizzate con il fornitore di materiali crittografici nella versione 2. x.

2. Crea uno schema tabellare sulla tua classe annotata.

   Questo passaggio definisce le azioni degli attributi che verranno utilizzate quando inizierai a scrivere elementi nel nuovo formato.

   Per indicazioni sull'uso del nuovo DynamoDB Enhanced Client, consulta Generate a nella AWS SDK per Java Developer *TableSchema* Guide.

   L'esempio seguente presuppone che tu abbia aggiornato la tua classe annotata dalla versione 2. x utilizzando le nuove annotazioni sulle azioni degli attributi. Per ulteriori indicazioni sull'annotazione delle azioni relative agli attributi, vedere. Utilizza una classe di dati annotata

   > **Note**
   >
   > Se si specificano degli `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` attributi, devono esserlo anche gli attributi di partizione e ordinamento. `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` Per un esempio che mostra le

annotazioni utilizzate per definire`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`, vedete SimpleClass 4.java.

3. Definire quali attributi sono esclusi dalla firma.
4. Configura una mappa esplicita delle azioni degli attributi configurate nella classe modellata della versione 2.x.

   Questo passaggio definisce le azioni degli attributi utilizzate per scrivere gli elementi nel vecchio formato.
5. Configura quello `DynamoDBEncryptor` che hai usato nella versione 2. x della libreria di crittografia lato client Java per DynamoDB.
6. Configura il comportamento precedente.
7. Creare un `DynamoDbEncryptionInterceptor`.
8. Crea un nuovo client AWS SDK DynamoDB.
9. Crea `DynamoDBEnhancedClient` e crea una tabella con la tua classe modellata.

   Per ulteriori informazioni sul DynamoDB Enhanced Client, consulta creare un client avanzato.

```
public class MigrationExampleStep1 {

    public static void MigrationStep1(String kmsKeyId, String ddbTableName, int
 sortReadValue) {
        // 1. Create a Keyring.
        //    This example creates an AWS KMS Keyring that specifies the
        //    same kmsKeyId previously used in the version 2.x configuration.
        //    It uses the 'CreateMrkMultiKeyring' method to create the
        //    keyring, so that the keyring can correctly handle both single
        //    region and Multi-Region KMS Keys.
        //    Note that this example uses the AWS SDK for Java v2 KMS client.
        final MaterialProviders matProv = MaterialProviders.builder()
                .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
                .build();
        final CreateAwsKmsMrkMultiKeyringInput keyringInput =
 CreateAwsKmsMrkMultiKeyringInput.builder()
                .generator(kmsKeyId)
                .build();
        final IKeyring kmsKeyring = matProv.CreateAwsKmsMrkMultiKeyring(keyringInput);

        // 2. Create a Table Schema over your annotated class.
```

```
        //    For guidance on using the new attribute actions
        //    annotations, see SimpleClass.java in the
        //    aws-database-encryption-sdk-dynamodb GitHub repository.
        //    All primary key attributes must be signed but not encrypted
        //    and by default all non-primary key attributes
        //    are encrypted and signed (ENCRYPT_AND_SIGN).
        //    If you want a particular non-primary key attribute to be signed but
        //    not encrypted, use the 'DynamoDbEncryptionSignOnly' annotation.
        //    If you want a particular attribute to be neither signed nor encrypted
        //    (DO_NOTHING), use the 'DynamoDbEncryptionDoNothing' annotation.
        final TableSchema<SimpleClass> schemaOnEncrypt =
TableSchema.fromBean(SimpleClass.class);

        // 3. Define which attributes the client should expect to be excluded
        //    from the signature when reading items.
        //    This value represents all unsigned attributes across the entire
        //    dataset.
        final List<String> allowedUnsignedAttributes = Arrays.asList("attribute3");

        // 4. Configure an explicit map of the attribute actions configured
        //    in your version 2.x modeled class.
        final Map<String, CryptoAction> legacyActions = new HashMap<>();
        legacyActions.put("partition_key", CryptoAction.SIGN_ONLY);
        legacyActions.put("sort_key", CryptoAction.SIGN_ONLY);
        legacyActions.put("attribute1", CryptoAction.ENCRYPT_AND_SIGN);
        legacyActions.put("attribute2", CryptoAction.SIGN_ONLY);
        legacyActions.put("attribute3", CryptoAction.DO_NOTHING);

        // 5. Configure the DynamoDBEncryptor that you used in version 2.x.
        final AWSKMS kmsClient = AWSKMSClientBuilder.defaultClient();
        final DirectKmsMaterialProvider cmp = new DirectKmsMaterialProvider(kmsClient,
kmsKeyId);
        final DynamoDBEncryptor oldEncryptor = DynamoDBEncryptor.getInstance(cmp);

        // 6. Configure the legacy behavior.
        //    Input the DynamoDBEncryptor and attribute actions created in
        //    the previous steps. For Legacy Policy, use
        //    'FORCE_LEGACY_ENCRYPT_ALLOW_LEGACY_DECRYPT'. This policy continues to
read
        //    and write items using the old format, but will be able to read
        //    items written in the new format as soon as they appear.
        final LegacyOverride legacyOverride = LegacyOverride
                .builder()
                .encryptor(oldEncryptor)
```

```
                .policy(LegacyPolicy.FORCE_LEGACY_ENCRYPT_ALLOW_LEGACY_DECRYPT)
                .attributeActionsOnEncrypt(legacyActions)
                .build();

        // 7. Create a DynamoDbEncryptionInterceptor with the above configuration.
        final Map<String, DynamoDbEnhancedTableEncryptionConfig> tableConfigs = new
 HashMap<>();
        tableConfigs.put(ddbTableName,
                DynamoDbEnhancedTableEncryptionConfig.builder()
                        .logicalTableName(ddbTableName)
                        .keyring(kmsKeyring)
                        .allowedUnsignedAttributes(allowedUnsignedAttributes)
                        .schemaOnEncrypt(tableSchema)
                        .legacyOverride(legacyOverride)
                        .build());
        final DynamoDbEncryptionInterceptor interceptor =
                DynamoDbEnhancedClientEncryption.CreateDynamoDbEncryptionInterceptor(
                        CreateDynamoDbEncryptionInterceptorInput.builder()
                                .tableEncryptionConfigs(tableConfigs)
                                .build()
                );

        // 8. Create a new AWS SDK DynamoDb client using the
        //    interceptor from Step 7.
        final DynamoDbClient ddb = DynamoDbClient.builder()
                .overrideConfiguration(
                        ClientOverrideConfiguration.builder()
                                .addExecutionInterceptor(interceptor)
                                .build())
                .build();

        // 9. Create the DynamoDbEnhancedClient using the AWS SDK DynamoDb client
        //    created in Step 8, and create a table with your modeled class.
        final DynamoDbEnhancedClient enhancedClient = DynamoDbEnhancedClient.builder()
                .dynamoDbClient(ddb)
                .build();
        final DynamoDbTable<SimpleClass> table = enhancedClient.table(ddbTableName,
 tableSchema);
    }
}
```

Fase 2: Scrivi gli elementi nel nuovo formato

Dopo aver distribuito le modifiche dal Passaggio 1 a tutti i lettori, completa i passaggi seguenti per configurare il client AWS Database Encryption SDK per scrivere elementi nel nuovo formato. Dopo aver implementato le seguenti modifiche, il client continuerà a leggere gli elementi nel vecchio formato e inizierà a scrivere e leggere gli elementi nel nuovo formato.

Le seguenti procedure forniscono una panoramica dei passaggi illustrati nell'esempio di codice riportato di seguito.

1. Continua a configurare il portachiavi, lo schema della tabella`allowedUnsignedAttributes`, le azioni degli attributi legacy e `DynamoDBEncryptor` come hai fatto nel passaggio 1.
2. Aggiorna il tuo comportamento precedente per scrivere solo nuovi elementi utilizzando il nuovo formato.
3. Creazione di una `DynamoDbEncryptionInterceptor`
4. Crea un nuovo client AWS SDK DynamoDB.
5. Crea `DynamoDBEnhancedClient` e crea una tabella con la tua classe modellata.

   Per ulteriori informazioni sul DynamoDB Enhanced Client, consulta creare un client avanzato.

```
public class MigrationExampleStep2 {

    public static void MigrationStep2(String kmsKeyId, String ddbTableName, int
 sortReadValue) {
        // 1. Continue to configure your keyring, table schema, legacy
        //    attribute actions, allowedUnsignedAttributes, and
        //    DynamoDBEncryptor as you did in Step 1.
        final MaterialProviders matProv = MaterialProviders.builder()
                .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
                .build();
        final CreateAwsKmsMrkMultiKeyringInput keyringInput =
 CreateAwsKmsMrkMultiKeyringInput.builder()
                .generator(kmsKeyId)
                .build();
        final IKeyring kmsKeyring = matProv.CreateAwsKmsMrkMultiKeyring(keyringInput);

        final TableSchema<SimpleClass> schemaOnEncrypt =
 TableSchema.fromBean(SimpleClass.class);

        final List<String> allowedUnsignedAttributes = Arrays.asList("attribute3");
```

```
        final Map<String, CryptoAction> legacyActions = new HashMap<>();
        legacyActions.put("partition_key", CryptoAction.SIGN_ONLY);
        legacyActions.put("sort_key", CryptoAction.SIGN_ONLY);
        legacyActions.put("attribute1", CryptoAction.ENCRYPT_AND_SIGN);
        legacyActions.put("attribute2", CryptoAction.SIGN_ONLY);
        legacyActions.put("attribute3", CryptoAction.DO_NOTHING);

        final AWSKMS kmsClient = AWSKMSClientBuilder.defaultClient();
        final DirectKmsMaterialProvider cmp = new DirectKmsMaterialProvider(kmsClient,
kmsKeyId);
        final DynamoDBEncryptor oldEncryptor = DynamoDBEncryptor.getInstance(cmp);

        // 2. Update your legacy behavior to only write new items using the new
        //    format.
        //    For Legacy Policy, use 'FORBID_LEGACY_ENCRYPT_ALLOW_LEGACY_DECRYPT'. This
policy
        //    continues to read items in both formats, but will only write items
        //    using the new format.
        final LegacyOverride legacyOverride = LegacyOverride
                .builder()
                .encryptor(oldEncryptor)
                .policy(LegacyPolicy.FORBID_LEGACY_ENCRYPT_ALLOW_LEGACY_DECRYPT)
                .attributeActionsOnEncrypt(legacyActions)
                .build();

        // 3. Create a DynamoDbEncryptionInterceptor with the above configuration.
        final Map<String, DynamoDbEnhancedTableEncryptionConfig> tableConfigs = new
HashMap<>();
        tableConfigs.put(ddbTableName,
                DynamoDbEnhancedTableEncryptionConfig.builder()
                        .logicalTableName(ddbTableName)
                        .keyring(kmsKeyring)
                        .allowedUnsignedAttributes(allowedUnsignedAttributes)
                        .schemaOnEncrypt(tableSchema)
                        .legacyOverride(legacyOverride)
                        .build());
        final DynamoDbEncryptionInterceptor interceptor =
                DynamoDbEnhancedClientEncryption.CreateDynamoDbEncryptionInterceptor(
                        CreateDynamoDbEncryptionInterceptorInput.builder()
                                .tableEncryptionConfigs(tableConfigs)
                                .build()
                );
```

```
        // 4. Create a new AWS SDK DynamoDb client using the
        //    interceptor from Step 3.
        final DynamoDbClient ddb = DynamoDbClient.builder()
                .overrideConfiguration(
                        ClientOverrideConfiguration.builder()
                                .addExecutionInterceptor(interceptor)
                                .build())
                .build();

        // 5. Create the DynamoDbEnhancedClient using the AWS SDK DynamoDb Client
 created
        //    in Step 4, and create a table with your modeled class.
        final DynamoDbEnhancedClient enhancedClient = DynamoDbEnhancedClient.builder()
                .dynamoDbClient(ddb)
                .build();
        final DynamoDbTable<SimpleClass> table = enhancedClient.table(ddbTableName,
 tableSchema);
    }
}
```

Dopo aver implementato le modifiche della Fase 2, è necessario crittografare nuovamente tutti i vecchi elementi della tabella con il nuovo formato prima di poter continuare con la Fase 3. Non è possibile eseguire alcuna metrica o query per crittografare rapidamente gli elementi esistenti. Utilizzate il processo più adatto al vostro sistema. Ad esempio, è possibile utilizzare un processo asincrono che scansiona lentamente la tabella e quindi riscrive gli elementi utilizzando le nuove azioni degli attributi e la configurazione di crittografia definite.

Fase 3. Leggi e scrivi solo elementi nel nuovo formato

Dopo aver ricrittografato tutti gli elementi della tabella con il nuovo formato, puoi rimuovere il comportamento precedente dalla tua configurazione. Completa i seguenti passaggi per configurare il client in modo che legga e scriva solo elementi nel nuovo formato.

Le seguenti procedure forniscono una panoramica dei passaggi illustrati nell'esempio di codice riportato di seguito.

1. Continua a configurare il portachiavi, lo schema della tabella e `allowedUnsignedAttributes` come hai fatto nel passaggio 1. Rimuovi le azioni degli attributi precedenti `DynamoDBEncryptor` dalla tua configurazione.
2. Creare un `DynamoDbEncryptionInterceptor`.
3. Crea un nuovo client AWS SDK DynamoDB.

4. Crea `DynamoDBEnhancedClient` e crea una tabella con la tua classe modellata.

   Per ulteriori informazioni sul DynamoDB Enhanced Client, consulta creare un client avanzato.

```
public class MigrationExampleStep3 {

    public static void MigrationStep3(String kmsKeyId, String ddbTableName, int
 sortReadValue) {
        // 1. Continue to configure your keyring, table schema,
        //    and allowedUnsignedAttributes as you did in Step 1.
        //    Do not include the configurations for the DynamoDBEncryptor or
        //    the legacy attribute actions.
        final MaterialProviders matProv = MaterialProviders.builder()
                .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
                .build();
        final CreateAwsKmsMrkMultiKeyringInput keyringInput =
 CreateAwsKmsMrkMultiKeyringInput.builder()
                .generator(kmsKeyId)
                .build();
        final IKeyring kmsKeyring = matProv.CreateAwsKmsMrkMultiKeyring(keyringInput);

        final TableSchema<SimpleClass> schemaOnEncrypt =
 TableSchema.fromBean(SimpleClass.class);

        final List<String> allowedUnsignedAttributes = Arrays.asList("attribute3");


        // 3. Create a DynamoDbEncryptionInterceptor with the above configuration.
        //    Do not configure any legacy behavior.
        final Map<String, DynamoDbEnhancedTableEncryptionConfig> tableConfigs = new
 HashMap<>();
        tableConfigs.put(ddbTableName,
                DynamoDbEnhancedTableEncryptionConfig.builder()
                        .logicalTableName(ddbTableName)
                        .keyring(kmsKeyring)
                        .allowedUnsignedAttributes(allowedUnsignedAttributes)
                        .schemaOnEncrypt(tableSchema)
                        .build());
        final DynamoDbEncryptionInterceptor interceptor =
                DynamoDbEnhancedClientEncryption.CreateDynamoDbEncryptionInterceptor(
                        CreateDynamoDbEncryptionInterceptorInput.builder()
                                .tableEncryptionConfigs(tableConfigs)
                                .build()
```

```
                );

        // 4. Create a new AWS SDK DynamoDb client using the
        //    interceptor from Step 3.
        final DynamoDbClient ddb = DynamoDbClient.builder()
                .overrideConfiguration(
                        ClientOverrideConfiguration.builder()
                                .addExecutionInterceptor(interceptor)
                                .build())
                .build();

        // 5. Create the DynamoDbEnhancedClient using the AWS SDK Client
        //    created in Step 4, and create a table with your modeled class.
        final DynamoDbEnhancedClient enhancedClient = DynamoDbEnhancedClient.builder()
                .dynamoDbClient(ddb)
                .build();
        final DynamoDbTable<SimpleClass> table = enhancedClient.table(ddbTableName,
 tableSchema);
    }
}
```

## .NET

Questo argomento spiega come installare e utilizzare la versione 3. x della libreria di crittografia lato client.NET per DynamoDB. Per i dettagli sulla programmazione con AWS Database Encryption SDK per DynamoDB, consulta gli esempi.NET nel aws-database-encryption-sdk repository -dynamodb su. GitHub

La libreria di crittografia lato client.NET per DynamoDB è destinata agli sviluppatori che scrivono applicazioni in C# e altri linguaggi di programmazione.NET. ed è supportata su Windows, macOS e Linux.

Tutte le implementazioni del linguaggio di programmazione del AWS Database Encryption SDK per DynamoDB sono interoperabili. Tuttavia, non SDK per .NET supporta valori vuoti per i tipi di dati di elenchi o mappe. Ciò significa che se si utilizza la libreria di crittografia lato client Java per DynamoDB per scrivere un elemento che contiene valori vuoti per un tipo di dati di elenco o mappa, non è possibile decrittografare e leggere tale elemento utilizzando la libreria di crittografia lato client.NET per DynamoDB.

Argomenti

- Installazione della libreria di crittografia lato client.NET per DynamoDB

- Esecuzione del debug con.NET
- Utilizzo della libreria di crittografia lato client.NET per DynamoDB
- esempi.NET
- Configurare una tabella DynamoDB esistente per utilizzare AWS il Database Encryption SDK per DynamoDB

## Installazione della libreria di crittografia lato client.NET per DynamoDB

La libreria di crittografia lato client.NET per DynamoDB è disponibile come AWS.Cryptography. DbEncryptionSDK. DynamoDbpacchetto in NuGet. Per informazioni dettagliate sull'installazione e la creazione della libreria, consultate il file.NET README.md nel repository -dynamodb. aws-database-encryption-sdk La libreria di crittografia lato client.NET per DynamoDB richiede le chiavi SDK per .NET anche se non si utilizzano (). AWS Key Management Service AWS KMS SDK per .NET Viene installato con il pacchetto. NuGet

Versione 3. x della libreria di crittografia lato client.NET per DynamoDB supporta .NET 6.0 e.NET Framework net48 e versioni successive.

## Esecuzione del debug con.NET

La libreria di crittografia lato client.NET per DynamoDB non genera alcun registro. Le eccezioni nella libreria di crittografia lato client.NET per DynamoDB generano un messaggio di eccezione, ma nessuna traccia dello stack.

Per aiutarti a eseguire il debug, assicurati di abilitare l'accesso a. SDK per .NET I log e i messaggi di errore di SDK per .NET possono aiutarti a distinguere gli errori derivanti SDK per .NET da quelli presenti nella libreria di crittografia lato client.NET per DynamoDB. Per informazioni sulla SDK per .NET registrazione, consulta la Guida per gli sviluppatori. AWSLoggingAWS SDK per .NET (Per vedere l'argomento, espandi la sezione Apri per visualizzare il contenuto di.NET Framework).

## Utilizzo della libreria di crittografia lato client.NET per DynamoDB

Questo argomento spiega alcune delle funzioni e delle classi di supporto della versione 3. x della libreria di crittografia lato client.NET per DynamoDB.

Per i dettagli sulla programmazione con la libreria di crittografia lato client.NET per DynamoDB, consulta gli esempi.NET nel repository -dynamodb su. aws-database-encryption-sdk GitHub

Argomenti

- Componenti di crittografia dell'item
- Azioni relative agli attributi nel AWS Database Encryption SDK per DynamoDB
- Configurazione della crittografia nel AWS Database Encryption SDK per DynamoDB
- Aggiornamento degli AWS elementi con Database Encryption SDK

Componenti di crittografia dell'item

Fondamentalmente, il AWS Database Encryption SDK per DynamoDB è un cifratore di elementi. È possibile utilizzare la versione 3. x della libreria di crittografia lato client.NET per DynamoDB per crittografare, firmare, verificare e decrittografare gli elementi della tabella DynamoDB nei seguenti modi.

L'SDK di crittografia del AWS database di basso livello per l'API DynamoDB

È possibile utilizzare la configurazione di crittografia delle tabelle per creare un client DynamoDB che crittografa e firma automaticamente gli elementi lato client con le richieste DynamoDB. `PutItem` È possibile utilizzare questo client direttamente oppure è possibile creare un modello di documento o un modello di persistenza degli oggetti.

È necessario utilizzare l'API AWS Database Encryption SDK di basso livello per DynamoDB per utilizzare la crittografia ricercabile.

Il livello inferiore **`DynamoDbItemEncryptor`**

Il livello inferiore crittografa e firma o decrittografa e verifica `DynamoDbItemEncryptor` direttamente gli elementi della tabella senza chiamare DynamoDB. Non crea DynamoDB o `PutItem` richieste`GetItem`. Ad esempio, puoi utilizzare il livello inferiore per `DynamoDbItemEncryptor` decrittografare e verificare direttamente un elemento DynamoDB che hai già recuperato. Se si utilizza il livello inferiore`DynamoDbItemEncryptor`, si consiglia di utilizzare il modello di programmazione di basso livello SDK per .NET fornito per comunicare con DynamoDB.

Il livello inferiore non supporta la crittografia ricercabile`DynamoDbItemEncryptor`.

Azioni relative agli attributi nel AWS Database Encryption SDK per DynamoDB

Le azioni relative agli attributi determinano quali valori degli attributi sono crittografati e firmati, quali sono solo firmati, quali sono firmati e inclusi nel contesto di crittografia e quali vengono ignorati.

Per specificare le azioni relative agli attributi con il client.NET, definisci manualmente le azioni relative agli attributi utilizzando un modello a oggetti. Specificate le azioni relative agli attributi creando un `Dictionary` oggetto in cui le coppie nome-valore rappresentano i nomi degli attributi e le azioni specificate.

Specificate `ENCRYPT_AND_SIGN` di crittografare e firmare un attributo.
`SIGN_ONLY`Specificare di firmare, ma non crittografare, un attributo.
`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`Specificare di firmare un attributo e di includerlo nel contesto di crittografia. Non è possibile crittografare un attributo senza firmarlo.
`DO_NOTHING`Specificare di ignorare un attributo.

Gli attributi di partizione e ordinamento devono essere `SIGN_ONLY` o`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`. Se si definiscono gli attributi come`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`, allora anche gli attributi di partizione e ordinamento devono essere uguali. `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`

> **Note**
>
> Dopo aver definito le azioni relative agli attributi, è necessario definire quali attributi sono esclusi dalle firme. Per semplificare l'aggiunta di nuovi attributi non firmati in futuro, consigliamo di scegliere un prefisso distinto (ad esempio "`:`«) per identificare gli attributi non firmati. Includi questo prefisso nel nome dell'attributo per tutti gli attributi contrassegnati durante `DO_NOTHING` la definizione dello schema DynamoDB e delle azioni degli attributi.

Il seguente modello a oggetti mostra come specificare `ENCRYPT_AND_SIGN` `SIGN_ONLY`e `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`, e `DO_NOTHING` attribuire azioni con il client.NET. Questo esempio utilizza il prefisso "`:`" per identificare `DO_NOTHING` gli attributi.

> **Note**
>
> Per utilizzare l'azione `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` crittografica, è necessario utilizzare la versione 3.3 o successiva di AWS Database Encryption SDK. Distribuisci la nuova versione a tutti i lettori prima di aggiornare il modello di dati per includere. `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`

```
var attributeActionsOnEncrypt = new Dictionary<string, CryptoAction>
```

```
{
    ["partition_key"] = CryptoAction.SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT, // The
 partition attribute must be signed
    ["sort_key"] = CryptoAction.SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT, // The sort
 attribute must be signed
    ["attribute1"] = CryptoAction.ENCRYPT_AND_SIGN,
    ["attribute2"] = CryptoAction.SIGN_ONLY,
    ["attribute3"] = CryptoAction.SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT,
    [":attribute4"] = CryptoAction.DO_NOTHING
};
```

Configurazione della crittografia nel AWS Database Encryption SDK per DynamoDB

Quando si utilizza AWS Database Encryption SDK, è necessario definire in modo esplicito una configurazione di crittografia per la tabella DynamoDB. I valori richiesti nella configurazione di crittografia dipendono dal fatto che le azioni degli attributi siano state definite manualmente o con una classe di dati annotata.

Il seguente frammento definisce una configurazione di crittografia delle tabelle DynamoDB utilizzando l'API AWS Database Encryption SDK di basso livello per DynamoDB e gli attributi non firmati consentiti definiti da un prefisso distinto.

```
Dictionary<String, DynamoDbTableEncryptionConfig> tableConfigs =
    new Dictionary<String, DynamoDbTableEncryptionConfig>();
DynamoDbTableEncryptionConfig config = new DynamoDbTableEncryptionConfig
{
    LogicalTableName = ddbTableName,
    PartitionKeyName = "partition_key",
    SortKeyName = "sort_key",
    AttributeActionsOnEncrypt = attributeActionsOnEncrypt,
    Keyring = kmsKeyring,
    AllowedUnsignedAttributePrefix = unsignAttrPrefix,
    // Optional: SearchConfig only required if you use beacons
    Search = new SearchConfig
    {
        WriteVersion = 1, // MUST be 1
        Versions = beaconVersions
    }
};
tableConfigs.Add(ddbTableName, config);
```

Nome della tabella logica

Un nome di tabella logica per la tabella DynamoDB.

Il nome della tabella logica è associato crittograficamente a tutti i dati memorizzati nella tabella per semplificare le operazioni di ripristino di DynamoDB. Consigliamo vivamente di specificare il nome della tabella DynamoDB come nome della tabella logica quando si definisce per la prima volta la configurazione di crittografia. È necessario specificare sempre lo stesso nome di tabella logica. Affinché la decrittografia abbia esito positivo, il nome della tabella logica deve corrispondere al nome specificato nella crittografia. Nel caso in cui il nome della tabella DynamoDB cambi dopo il ripristino della tabella DynamoDB da un backup, il nome della tabella logica assicura che l'operazione di decrittografia riconosca ancora la tabella.

Attributi non firmati consentiti

Gli attributi contrassegnati `DO_NOTHING` nelle azioni relative agli attributi.

Gli attributi non firmati consentiti indicano al client quali attributi sono esclusi dalle firme. Il client presume che tutti gli altri attributi siano inclusi nella firma. Quindi, durante la decrittografia di un record, il client determina quali attributi deve verificare e quali ignorare tra gli attributi non firmati consentiti specificati. Non è possibile rimuovere un attributo dagli attributi non firmati consentiti.

È possibile definire gli attributi non firmati consentiti in modo esplicito creando un array che elenca tutti gli attributi. `DO_NOTHING` È inoltre possibile specificare un prefisso distinto quando si assegnano nomi `DO_NOTHING` agli attributi e utilizzare il prefisso per indicare al client quali attributi non sono firmati. Consigliamo vivamente di specificare un prefisso distinto perché semplifica il processo di aggiunta di un nuovo `DO_NOTHING` attributo in futuro. Per ulteriori informazioni, consulta Aggiornamento del modello di dati.

Se non si specifica un prefisso per tutti gli `DO_NOTHING` attributi, è possibile configurare un `allowedUnsignedAttributes` array che elenchi in modo esplicito tutti gli attributi che il client dovrebbe aspettarsi che non siano firmati quando li incontra durante la decrittografia. È necessario definire in modo esplicito gli attributi non firmati consentiti solo se assolutamente necessario.

Configurazione della ricerca (opzionale)

`SearchConfig`definisce la versione del beacon.

È `SearchConfig` necessario specificare il per utilizzare la crittografia ricercabile o i beacon firmati.

Algorithm Suite (opzionale)

`algorithmSuiteId`Definisce la suite di algoritmi utilizzata da AWS Database Encryption SDK.

A meno che non si specifichi esplicitamente una suite di algoritmi alternativa, AWS Database Encryption SDK utilizza la suite di algoritmi predefinita. La suite di algoritmi predefinita utilizza l'algoritmo AES-GCM con derivazione delle chiavi, firme digitali e impegno delle chiavi. Sebbene la suite di algoritmi predefinita sia probabilmente adatta alla maggior parte delle applicazioni, è possibile scegliere una suite di algoritmi alternativa. Ad esempio, alcuni modelli di fiducia sarebbero soddisfatti da una suite di algoritmi senza firme digitali. Per informazioni sulle suite di algoritmi supportate da AWS Database Encryption SDK, consulta. Suite di algoritmi supportate nel AWS Database Encryption SDK

Per selezionare la suite di algoritmi AES-GCM senza firme digitali ECDSA, includi il seguente frammento nella configurazione di crittografia delle tabelle.

```
AlgorithmSuiteId =
 DBEAlgorithmSuiteId.ALG_AES_256_GCM_HKDF_SHA512_COMMIT_KEY_SYMSIG_HMAC_SHA384
```

Aggiornamento degli AWS elementi con Database Encryption SDK

Il AWS Database Encryption SDK non supporta ddb: UpdateItem per gli elementi che includono attributi crittografati o firmati. Per aggiornare un attributo crittografato o firmato, è necessario utilizzare ddb:. PutItem Quando specifichi la stessa chiave primaria di un elemento esistente nella tua `PutItem` richiesta, il nuovo elemento sostituisce completamente l'elemento esistente. Puoi anche usare CLOBBER per cancellare e sostituire tutti gli attributi al momento del salvataggio dopo aver aggiornato i tuoi articoli.

## esempi.NET

Gli esempi seguenti mostrano come utilizzare la libreria di crittografia lato client.NET per DynamoDB per proteggere gli elementi della tabella nell'applicazione. Per trovare altri esempi (e contribuire con i tuoi), consulta gli esempi.NET nel repository -dynamodb su. aws-database-encryption-sdk GitHub

Gli esempi seguenti mostrano come configurare la libreria di crittografia lato client.NET per DynamoDB in una nuova tabella Amazon DynamoDB non popolata. Se desideri configurare le tabelle Amazon DynamoDB esistenti per la crittografia lato client, consulta. Aggiungi la versione 3.x a una tabella esistente

Argomenti



## Utilizzo dell'SDK di basso livello AWS Database Encryption per DynamoDB API

L'esempio seguente mostra come utilizzare l'API AWS Database Encryption SDK di basso livello per DynamoDB con un AWS KMS portachiavi per crittografare e firmare automaticamente gli elementi lato client con le richieste DynamoDB. `PutItem`

Puoi utilizzare qualsiasi portachiavi supportato, ma ti consigliamo di utilizzare uno dei portachiavi quando possibile. AWS KMS

Guarda l'esempio di codice completo: .cs BasicPutGetExample

### Fase 1: Creare il portachiavi AWS KMS

L'esempio seguente utilizza la creazione `CreateAwsKmsMrkMultiKeyring` di un AWS KMS portachiavi con una chiave KMS di crittografia simmetrica. Il `CreateAwsKmsMrkMultiKeyring` metodo garantisce che il portachiavi gestisca correttamente sia le chiavi a regione singola che quelle a più regioni.

```
var matProv = new MaterialProviders(new MaterialProvidersConfig());
var keyringInput = new CreateAwsKmsMrkMultiKeyringInput { Generator = kmsKeyId };
var kmsKeyring = matProv.CreateAwsKmsMrkMultiKeyring(keyringInput);
```

### Passaggio 2: configura le azioni relative agli attributi

L'esempio seguente definisce un `attributeActionsOnEncrypt` dizionario che rappresenta azioni di esempio relative agli attributi per un elemento della tabella.

> **Note**
>
> L'esempio seguente non definisce alcun attributo come`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`. Se si specifica un `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` attributo, devono esserlo `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` anche gli attributi di partizione e ordinamento.

```
var attributeActionsOnEncrypt = new Dictionary<string, CryptoAction>
{
    ["partition_key"] = CryptoAction.SIGN_ONLY, // The partition attribute must be
 SIGN_ONLY
    ["sort_key"] = CryptoAction.SIGN_ONLY, // The sort attribute must be SIGN_ONLY
    ["attribute1"] = CryptoAction.ENCRYPT_AND_SIGN,
    ["attribute2"] = CryptoAction.SIGN_ONLY,
    [":attribute3"] = CryptoAction.DO_NOTHING
};
```

Fase 3: Definire quali attributi sono esclusi dalle firme

L'esempio seguente presuppone che tutti `DO_NOTHING` gli attributi condividano il prefisso distinto "`:`«e utilizza il prefisso per definire gli attributi non firmati consentiti. Il client presuppone che qualsiasi nome di attributo con il prefisso "`:`" sia escluso dalle firme. Per ulteriori informazioni, consulta Allowed unsigned attributes.

```
const String unsignAttrPrefix = ":";
```

Fase 4: Definire la configurazione di crittografia delle tabelle DynamoDB

L'esempio seguente definisce una `tableConfigs` mappa che rappresenta la configurazione di crittografia per questa tabella DynamoDB.

Questo esempio specifica il nome della tabella DynamoDB come nome della tabella logica. Consigliamo vivamente di specificare il nome della tabella DynamoDB come nome della tabella logica quando si definisce per la prima volta la configurazione di crittografia. Per ulteriori informazioni, consulta Configurazione della crittografia nel AWS Database Encryption SDK per DynamoDB.

> **Note**
>
> Per utilizzare la crittografia ricercabile o i beacon firmati, è necessario includerli anche nella configurazione di crittografia. `SearchConfig`

```
Dictionary<String, DynamoDbTableEncryptionConfig> tableConfigs =
    new Dictionary<String, DynamoDbTableEncryptionConfig>();
DynamoDbTableEncryptionConfig config = new DynamoDbTableEncryptionConfig
```

```
{
    LogicalTableName = ddbTableName,
    PartitionKeyName = "partition_key",
    SortKeyName = "sort_key",
    AttributeActionsOnEncrypt = attributeActionsOnEncrypt,
    Keyring = kmsKeyring,
    AllowedUnsignedAttributePrefix = unsignAttrPrefix
};
tableConfigs.Add(ddbTableName, config);
```

Fase 5: Creare un nuovo client AWS SDK DynamoDB

L'esempio seguente crea un nuovo client AWS SDK DynamoDB utilizzando **`TableEncryptionConfigs`** lo Step 4.

```
var ddb = new Client.DynamoDbClient(
    new DynamoDbTablesEncryptionConfig { TableEncryptionConfigs = tableConfigs });
```

Fase 6: Crittografare e firmare un elemento della tabella DynamoDB

L'esempio seguente definisce un `item` dizionario che rappresenta un elemento di tabella di esempio e inserisce l'elemento nella tabella DynamoDB. L'elemento viene crittografato e firmato lato client prima di essere inviato a DynamoDB.

```
var item = new Dictionary<String, AttributeValue>
{
    ["partition_key"] = new AttributeValue("BasicPutGetExample"),
    ["sort_key"] = new AttributeValue { N = "0" },
    ["attribute1"] = new AttributeValue("encrypt and sign me!"),
    ["attribute2"] = new AttributeValue("sign me!"),
    [":attribute3"] = new AttributeValue("ignore me!")
};

PutItemRequest putRequest = new PutItemRequest
{
    TableName = ddbTableName,
    Item = item
};

PutItemResponse putResponse = await ddb.PutItemAsync(putRequest);
```

### Utilizzo del livello inferiore **`DynamoDbItemEncryptor`**

L'esempio seguente mostra come utilizzare il livello inferiore `DynamoDbItemEncryptor` con un AWS KMS portachiavi per crittografare e firmare direttamente gli elementi della tabella. Non `DynamoDbItemEncryptor` inserisce l'elemento nella tabella DynamoDB.

Puoi utilizzare qualsiasi portachiavi supportato con DynamoDB Enhanced Client, ma consigliamo di utilizzare uno dei AWS KMS portachiavi quando possibile.

> **Note**
>
> Il livello inferiore `DynamoDbItemEncryptor` non supporta la crittografia ricercabile. Utilizza l'API AWS Database Encryption SDK di basso livello per DynamoDB per utilizzare la crittografia ricercabile.

Guarda l'esempio di codice completo: .cs ItemEncryptDecryptExample

#### Fase 1: Creare il portachiavi AWS KMS

L'esempio seguente utilizza la creazione `CreateAwsKmsMrkMultiKeyring` di un AWS KMS portachiavi con una chiave KMS di crittografia simmetrica. Il `CreateAwsKmsMrkMultiKeyring` metodo garantisce che il portachiavi gestisca correttamente sia le chiavi a regione singola che quelle a più regioni.

```
var matProv = new MaterialProviders(new MaterialProvidersConfig());
var keyringInput = new CreateAwsKmsMrkMultiKeyringInput { Generator = kmsKeyId };
var kmsKeyring = matProv.CreateAwsKmsMrkMultiKeyring(keyringInput);
```

#### Passaggio 2: configura le azioni relative agli attributi

L'esempio seguente definisce un `attributeActionsOnEncrypt` dizionario che rappresenta azioni di esempio relative agli attributi per un elemento della tabella.

> **Note**
>
> L'esempio seguente non definisce alcun attributo come`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`. Se si specifica un `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` attributo, devono esserlo

`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` anche gli attributi di partizione e ordinamento.

```
var attributeActionsOnEncrypt = new Dictionary<String, CryptoAction>
{
    ["partition_key"] = CryptoAction.SIGN_ONLY, // The partition attribute must be
 SIGN_ONLY
    ["sort_key"] = CryptoAction.SIGN_ONLY, // The sort attribute must be SIGN_ONLY
    ["attribute1"] = CryptoAction.ENCRYPT_AND_SIGN,
    ["attribute2"] = CryptoAction.SIGN_ONLY,
    [":attribute3"] = CryptoAction.DO_NOTHING
};
```

Fase 3: Definire quali attributi sono esclusi dalle firme

L'esempio seguente presuppone che tutti `DO_NOTHING` gli attributi condividano il prefisso distinto "`:`«e utilizza il prefisso per definire gli attributi non firmati consentiti. Il client presuppone che qualsiasi nome di attributo con il prefisso "`:`" sia escluso dalle firme. Per ulteriori informazioni, consulta Allowed unsigned attributes.

```
String unsignAttrPrefix = ":";
```

Fase 4: Definire la configurazione **`DynamoDbItemEncryptor`**

L'esempio seguente definisce la configurazione per`DynamoDbItemEncryptor`.

Questo esempio specifica il nome della tabella DynamoDB come nome della tabella logica. Consigliamo vivamente di specificare il nome della tabella DynamoDB come nome della tabella logica quando si definisce per la prima volta la configurazione di crittografia. Per ulteriori informazioni, consulta Configurazione della crittografia nel AWS Database Encryption SDK per DynamoDB.

```
var config = new DynamoDbItemEncryptorConfig
{
    LogicalTableName = ddbTableName,
    PartitionKeyName = "partition_key",
    SortKeyName = "sort_key",
    AttributeActionsOnEncrypt = attributeActionsOnEncrypt,
    Keyring = kmsKeyring,
    AllowedUnsignedAttributePrefix = unsignAttrPrefix
```

```
};
```

Fase 5: Creare il **`DynamoDbItemEncryptor`**

L'esempio seguente ne crea uno nuovo `DynamoDbItemEncryptor` utilizzando il `config` tratto dal passaggio 4.

```
var itemEncryptor = new DynamoDbItemEncryptor(config);
```

Fase 6: Crittografare e firmare direttamente un elemento della tabella

L'esempio seguente crittografa e firma direttamente un elemento utilizzando il. `DynamoDbItemEncryptor` Non `DynamoDbItemEncryptor` inserisce l'elemento nella tabella DynamoDB.

```
var originalItem = new Dictionary<String, AttributeValue>
{
    ["partition_key"] = new AttributeValue("ItemEncryptDecryptExample"),
    ["sort_key"] = new AttributeValue { N = "0" },
    ["attribute1"] = new AttributeValue("encrypt and sign me!"),
    ["attribute2"] = new AttributeValue("sign me!"),
    [":attribute3"] = new AttributeValue("ignore me!")
};

var encryptedItem = itemEncryptor.EncryptItem(
    new EncryptItemInput { PlaintextItem = originalItem }
).EncryptedItem;
```

## Configurare una tabella DynamoDB esistente per utilizzare AWS il Database Encryption SDK per DynamoDB

Con la versione 3. x della libreria di crittografia lato client.NET per DynamoDB, puoi configurare le tabelle Amazon DynamoDB esistenti per la crittografia lato client. Questo argomento fornisce indicazioni sui tre passaggi da eseguire per aggiungere la versione 3. x su una tabella DynamoDB esistente e popolata.

Fase 1: Prepararsi a leggere e scrivere elementi crittografati

Completa i seguenti passaggi per preparare il client AWS Database Encryption SDK per leggere e scrivere elementi crittografati. Dopo aver implementato le seguenti modifiche, il client continuerà a

leggere e scrivere elementi in testo non crittografato. Non crittograferà né firmerà i nuovi elementi scritti nella tabella, ma sarà in grado di decrittografare gli elementi crittografati non appena vengono visualizzati. Queste modifiche preparano il client a iniziare a crittografare nuovi elementi. Le seguenti modifiche devono essere implementate su ciascun lettore prima di procedere al passaggio successivo.

1. Definite le azioni relative agli attributi

   Create un modello a oggetti per definire quali valori degli attributi verranno crittografati e firmati, quali verranno solo firmati e quali ignorati.

   Per impostazione predefinita, gli attributi della chiave primaria sono firmati ma non crittografati (`SIGN_ONLY`) e tutti gli altri attributi sono crittografati e firmati (`ENCRYPT_AND_SIGN`).

   `ENCRYPT_AND_SIGN`Specificare di crittografare e firmare un attributo.
   `SIGN_ONLY`Specificare di firmare, ma non crittografare, un attributo.
   `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`Specificare se firmare e attribuire e includerlo nel contesto di crittografia. Non è possibile crittografare un attributo senza firmarlo.
   `DO_NOTHING`Specificare di ignorare un attributo. Per ulteriori informazioni, consulta Azioni relative agli attributi nel AWS Database Encryption SDK per DynamoDB.

   > **Note**
   >
   > Se specificate degli `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` attributi, devono esserlo `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` anche gli attributi di partizione e ordinamento.

```
var attributeActionsOnEncrypt = new Dictionary<string, CryptoAction>
{
    ["partition_key"] = CryptoAction.SIGN_ONLY, // The partition attribute must be
 SIGN_ONLY
    ["sort_key"] = CryptoAction.SIGN_ONLY, // The sort attribute must be SIGN_ONLY
    ["attribute1"] = CryptoAction.ENCRYPT_AND_SIGN,
    ["attribute2"] = CryptoAction.SIGN_ONLY,
    [":attribute3"] = CryptoAction.DO_NOTHING
};
```

2. Definite quali attributi verranno esclusi dalle firme

   L'esempio seguente presuppone che tutti `DO_NOTHING` gli attributi condividano il prefisso distinto "`:`«e utilizza il prefisso per definire gli attributi non firmati consentiti. Il client presumerà che qualsiasi nome di attributo con il prefisso "`:`" sia escluso dalle firme. Per ulteriori informazioni, consulta Allowed unsigned attributes.

   ```
   const String unsignAttrPrefix = ":";
   ```

3. Crea un portachiavi

   L'esempio seguente crea un AWS KMS portachiavi. Il AWS KMS portachiavi utilizza la crittografia simmetrica o RSA asimmetrica per generare, crittografare e AWS KMS keys decrittografare le chiavi di dati.

   Questo esempio utilizza per creare un portachiavi con una chiave KMS di crittografia `CreateMrkMultiKeyring` simmetrica. AWS KMS Il `CreateAwsKmsMrkMultiKeyring` metodo garantisce che il portachiavi gestisca correttamente sia le chiavi a regione singola che quelle a più regioni.

   ```
   var matProv = new MaterialProviders(new MaterialProvidersConfig());
   var keyringInput = new CreateAwsKmsMrkMultiKeyringInput { Generator = kmsKeyId };
   var kmsKeyring = matProv.CreateAwsKmsMrkMultiKeyring(keyringInput);
   ```

4. Definire la configurazione di crittografia delle tabelle DynamoDB

   L'esempio seguente definisce una `tableConfigs` mappa che rappresenta la configurazione di crittografia per questa tabella DynamoDB.

   Questo esempio specifica il nome della tabella DynamoDB come nome della tabella logica. Consigliamo vivamente di specificare il nome della tabella DynamoDB come nome della tabella logica quando si definisce per la prima volta la configurazione di crittografia.

   È necessario specificare `FORCE_WRITE_PLAINTEXT_ALLOW_READ_PLAINTEXT` come override in testo semplice. Questa politica continua a leggere e scrivere elementi in testo non crittografato, legge elementi crittografati e prepara il client a scrivere elementi crittografati.

   Per ulteriori informazioni sui valori inclusi nella configurazione di crittografia delle tabelle, vedere. Configurazione della crittografia nel AWS Database Encryption SDK per DynamoDB

   ```
   Dictionary<String, DynamoDbTableEncryptionConfig> tableConfigs =
   ```

```
    new Dictionary<String, DynamoDbTableEncryptionConfig>();
DynamoDbTableEncryptionConfig config = new DynamoDbTableEncryptionConfig
{
    LogicalTableName = ddbTableName,
    PartitionKeyName = "partition_key",
    SortKeyName = "sort_key",
    AttributeActionsOnEncrypt = attributeActionsOnEncrypt,
    Keyring = kmsKeyring,
    AllowedUnsignedAttributePrefix = unsignAttrPrefix,
    PlaintextOverride = FORCE_WRITE_PLAINTEXT_ALLOW_READ_PLAINTEXT
};
tableConfigs.Add(ddbTableName, config);
```

5. Crea un nuovo client AWS SDK DynamoDB

   L'esempio seguente crea un nuovo client AWS SDK DynamoDB utilizzando **`TableEncryptionConfigs`** lo Step 4.

```
var ddb = new Client.DynamoDbClient(
    new DynamoDbTablesEncryptionConfig { TableEncryptionConfigs = tableConfigs });
```

Fase 2: Scrivere elementi crittografati e firmati

Aggiorna la politica di testo in chiaro nella configurazione di crittografia delle tabelle per consentire al client di scrivere elementi crittografati e firmati. Dopo aver implementato la seguente modifica, il client crittograferà e firmerà i nuovi elementi in base alle azioni degli attributi configurate nel passaggio 1. Il client sarà in grado di leggere elementi in testo semplice e elementi crittografati e firmati.

Prima di procedere alla Fase 3, è necessario crittografare e firmare tutti gli elementi di testo in chiaro esistenti nella tabella. Non è possibile eseguire alcuna metrica o query per crittografare rapidamente gli elementi di testo in chiaro esistenti. Utilizzate il processo più adatto al vostro sistema. Ad esempio, è possibile utilizzare un processo asincrono che scansiona lentamente la tabella e quindi riscrive gli elementi utilizzando le azioni degli attributi e la configurazione di crittografia definite. Per identificare gli elementi di testo in chiaro nella tabella, consigliamo di cercare tutti gli elementi che non contengono gli `aws_dbe_foot` attributi `aws_dbe_head` e che AWS Database Encryption SDK aggiunge agli elementi quando sono crittografati e firmati.

L'esempio seguente aggiorna la configurazione di crittografia della tabella dal passaggio 1. È necessario aggiornare l'override del testo in chiaro con. `FORBID_WRITE_PLAINTEXT_ALLOW_READ_PLAINTEXT` Questo criterio continua a leggere gli

elementi di testo in chiaro, ma legge e scrive anche gli elementi crittografati. Crea un nuovo client AWS SDK DynamoDB utilizzando l'aggiornamento. `TableEncryptionConfigs`

```
Dictionary<String, DynamoDbTableEncryptionConfig> tableConfigs =
    new Dictionary<String, DynamoDbTableEncryptionConfig>();
DynamoDbTableEncryptionConfig config = new DynamoDbTableEncryptionConfig
{
    LogicalTableName = ddbTableName,
    PartitionKeyName = "partition_key",
    SortKeyName = "sort_key",
    AttributeActionsOnEncrypt = attributeActionsOnEncrypt,
    Keyring = kmsKeyring,
    AllowedUnsignedAttributePrefix = unsignAttrPrefix,
    PlaintextOverride = FORBID_WRITE_PLAINTEXT_ALLOW_READ_PLAINTEXT
};
tableConfigs.Add(ddbTableName, config);
```

Passaggio 3: Leggi solo gli elementi crittografati e firmati

Dopo aver crittografato e firmato tutti gli elementi, aggiorna l'override del testo in chiaro nella configurazione di crittografia delle tabelle per consentire al client solo di leggere e scrivere elementi crittografati e firmati. Dopo aver implementato la seguente modifica, il client crittograferà e firmerà i nuovi elementi in base alle azioni degli attributi configurate nel passaggio 1. Il client sarà in grado di leggere solo gli elementi crittografati e firmati.

L'esempio seguente aggiorna la configurazione di crittografia delle tabelle dal passaggio 2. È possibile aggiornare l'override del testo in chiaro con `FORBID_WRITE_PLAINTEXT_FORBID_READ_PLAINTEXT` o rimuovere la politica del testo in chiaro dalla configurazione. Per impostazione predefinita, il client legge e scrive solo elementi crittografati e firmati. Crea un nuovo client AWS SDK DynamoDB utilizzando l'aggiornamento. `TableEncryptionConfigs`

```
Dictionary<String, DynamoDbTableEncryptionConfig> tableConfigs =
    new Dictionary<String, DynamoDbTableEncryptionConfig>();
DynamoDbTableEncryptionConfig config = new DynamoDbTableEncryptionConfig
{
    LogicalTableName = ddbTableName,
    PartitionKeyName = "partition_key",
    SortKeyName = "sort_key",
    AttributeActionsOnEncrypt = attributeActionsOnEncrypt,
    Keyring = kmsKeyring,
```

```
    AllowedUnsignedAttributePrefix = unsignAttrPrefix,
    // Optional: you can also remove the plaintext policy from your configuration
    PlaintextOverride = FORBID_WRITE_PLAINTEXT_FORBID_READ_PLAINTEXT
};
tableConfigs.Add(ddbTableName, config);
```

# Rust

Questo argomento spiega come installare e utilizzare la versione 1. x della libreria di crittografia lato client Rust per DynamoDB. Per i dettagli sulla programmazione con AWS Database Encryption SDK per DynamoDB, consulta gli esempi di Rust nel aws-database-encryption-sdk repository -dynamodb su. GitHub

Tutte le implementazioni del linguaggio di programmazione del AWS Database Encryption SDK per DynamoDB sono interoperabili.

Argomenti



## Prerequisiti

Prima di installare la libreria di crittografia lato client Rust per DynamoDB, assicurati di avere i seguenti prerequisiti.

Installa Rust and Cargo

Installa l'attuale versione stabile di Rust usando rustup.

Per ulteriori informazioni sul download e l'installazione di rustup, consulta le procedure di installazione in The Cargo Book.

## Installazione

La libreria di crittografia lato client Rust per DynamoDB è disponibile come cassa su Crates.io. aws-db-esdk Per i dettagli sull'installazione e la creazione della libreria, consultate il file README.md nel repository -dynamodb. aws-database-encryption-sdk GitHub

Manualmente

Per installare la libreria di crittografia lato client Rust per DynamoDB, clona o scarica il repository -dynamodb. aws-database-encryption-sdk GitHub

Per installare la versione più recente

Esegui il seguente comando Cargo nella directory del tuo progetto:

```
cargo add aws-db-esdk
```

Oppure aggiungi la seguente riga al tuo Cargo.toml:

```
aws-db-esdk = "<version>"
```

## Utilizzo della libreria di crittografia lato client Rust per DynamoDB

Questo argomento spiega alcune delle funzioni e delle classi di supporto della versione 1. x della libreria di crittografia lato client Rust per DynamoDB.

Per i dettagli sulla programmazione con la libreria di crittografia lato client Rust per DynamoDB, consulta gli esempi di Rust nel repository -dynamodb su. aws-database-encryption-sdk GitHub

Argomenti

- Componenti di crittografia dell'item
- Azioni relative agli attributi nel AWS Database Encryption SDK per DynamoDB
- Configurazione della crittografia nel AWS Database Encryption SDK per DynamoDB
- Aggiornamento degli AWS elementi con Database Encryption SDK

### Componenti di crittografia dell'item

Fondamentalmente, il AWS Database Encryption SDK per DynamoDB è un cifratore di elementi. È possibile utilizzare la versione 1. x della libreria di crittografia lato client Rust per DynamoDB per crittografare, firmare, verificare e decrittografare gli elementi della tabella DynamoDB nei seguenti modi.

L'SDK di crittografia del AWS database di basso livello per l'API DynamoDB

È possibile utilizzare la configurazione di crittografia delle tabelle per creare un client DynamoDB che crittografa e firma automaticamente gli elementi lato client con le richieste DynamoDB. `PutItem`

È necessario utilizzare l'API AWS Database Encryption SDK di basso livello per DynamoDB per utilizzare la crittografia ricercabile.

Per un esempio che dimostra come utilizzare l'SDK di basso livello di AWS Database Encryption SDK per DynamoDB, consulta basic_get_put_example.rs nel repository -dynamodb su. aws-database-encryption-sdk GitHub

Il livello inferiore **`DynamoDbItemEncryptor`**

Il livello inferiore crittografa e firma o decrittografa e verifica `DynamoDbItemEncryptor` direttamente gli elementi della tabella senza chiamare DynamoDB. Non crea DynamoDB o `PutItem` richieste`GetItem`. Ad esempio, puoi utilizzare il livello inferiore per `DynamoDbItemEncryptor` decrittografare e verificare direttamente un elemento DynamoDB che hai già recuperato.

Il livello inferiore non supporta la crittografia ricercabile. `DynamoDbItemEncryptor`

Per un esempio che dimostra come utilizzare il livello inferiore, consulta item_encrypt_decrypt.rs nel repository `DynamoDbItemEncryptor` -dynamodb su. aws-database-encryption-sdk GitHub

Azioni relative agli attributi nel AWS Database Encryption SDK per DynamoDB

Le azioni relative agli attributi determinano quali valori degli attributi sono crittografati e firmati, quali sono solo firmati, quali sono firmati e inclusi nel contesto di crittografia e quali vengono ignorati.

Per specificare le azioni relative agli attributi con il client Rust, definisci manualmente le azioni relative agli attributi utilizzando un modello a oggetti. Specificate le azioni degli attributi creando un `HashMap` oggetto in cui le coppie nome-valore rappresentano i nomi degli attributi e le azioni specificate.

Specificate `ENCRYPT_AND_SIGN` di crittografare e firmare un attributo.
`SIGN_ONLY`Specificare di firmare, ma non crittografare, un attributo.
`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`Specificare di firmare un attributo e di includerlo nel contesto di crittografia. Non è possibile crittografare un attributo senza firmarlo.
`DO_NOTHING`Specificare di ignorare un attributo.

Gli attributi di partizione e ordinamento devono essere `SIGN_ONLY` o`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`. Se si definiscono gli attributi come`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`, allora anche gli attributi di partizione e ordinamento devono essere uguali. `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`

> **Note**
>
> Dopo aver definito le azioni relative agli attributi, è necessario definire quali attributi sono esclusi dalle firme. Per semplificare l'aggiunta di nuovi attributi non firmati in futuro, consigliamo di scegliere un prefisso distinto (ad esempio "`:`«) per identificare gli attributi non firmati. Includi questo prefisso nel nome dell'attributo per tutti gli attributi contrassegnati durante `DO_NOTHING` la definizione dello schema DynamoDB e delle azioni degli attributi.

Il seguente modello a oggetti mostra come specificare`ENCRYPT_AND_SIGN`, `SIGN_ONLY`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT, e `DO_NOTHING` attribuire azioni con il client Rust. Questo esempio utilizza il prefisso "`:`" per identificare `DO_NOTHING` gli attributi.

```
let attribute_actions_on_encrypt = HashMap::from([
    ("partition_key".to_string(), CryptoAction::SignOnly),
    ("sort_key".to_string(), CryptoAction::SignOnly),
    ("attribute1".to_string(), CryptoAction::EncryptAndSign),
    ("attribute2".to_string(), CryptoAction::SignOnly),
    (":attribute3".to_string(), CryptoAction::DoNothing),
]);
```

Configurazione della crittografia nel AWS Database Encryption SDK per DynamoDB

Quando si utilizza AWS Database Encryption SDK, è necessario definire in modo esplicito una configurazione di crittografia per la tabella DynamoDB. I valori richiesti nella configurazione di crittografia dipendono dal fatto che le azioni degli attributi siano state definite manualmente o con una classe di dati annotata.

Il seguente frammento definisce una configurazione di crittografia delle tabelle DynamoDB utilizzando l'API AWS Database Encryption SDK di basso livello per DynamoDB e gli attributi non firmati consentiti definiti da un prefisso distinto.

```
let table_config = DynamoDbTableEncryptionConfig::builder()
    .logical_table_name(ddb_table_name)
```

```
    .partition_key_name("partition_key")
    .sort_key_name("sort_key")
    .attribute_actions_on_encrypt(attribute_actions_on_encrypt)
    .keyring(kms_keyring)
    .allowed_unsigned_attribute_prefix(UNSIGNED_ATTR_PREFIX)
    // Specifying an algorithm suite is optional
    .algorithm_suite_id(
        DbeAlgorithmSuiteId::AlgAes256GcmHkdfSha512CommitKeyEcdsaP384SymsigHmacSha384,
    )
    .build()?;

let table_configs = DynamoDbTablesEncryptionConfig::builder()
    .table_encryption_configs(HashMap::from([(ddb_table_name.to_string(),
 table_config)]))
    .build()?;
```

Nome della tabella logica

Un nome di tabella logica per la tabella DynamoDB.

Il nome della tabella logica è associato crittograficamente a tutti i dati memorizzati nella tabella per semplificare le operazioni di ripristino di DynamoDB. Consigliamo vivamente di specificare il nome della tabella DynamoDB come nome della tabella logica quando si definisce per la prima volta la configurazione di crittografia. È necessario specificare sempre lo stesso nome di tabella logica. Affinché la decrittografia abbia esito positivo, il nome della tabella logica deve corrispondere al nome specificato nella crittografia. Nel caso in cui il nome della tabella DynamoDB cambi dopo il ripristino della tabella DynamoDB da un backup, il nome della tabella logica assicura che l'operazione di decrittografia riconosca ancora la tabella.

Attributi non firmati consentiti

Gli attributi contrassegnati `DO_NOTHING` nelle azioni relative agli attributi.

Gli attributi non firmati consentiti indicano al client quali attributi sono esclusi dalle firme. Il client presume che tutti gli altri attributi siano inclusi nella firma. Quindi, durante la decrittografia di un record, il client determina quali attributi deve verificare e quali ignorare tra gli attributi non firmati consentiti specificati. Non è possibile rimuovere un attributo dagli attributi non firmati consentiti.

È possibile definire gli attributi non firmati consentiti in modo esplicito creando un array che elenca tutti gli attributi. `DO_NOTHING` È inoltre possibile specificare un prefisso distinto quando si assegnano nomi `DO_NOTHING` agli attributi e utilizzare il prefisso per indicare al client quali attributi non sono firmati. Consigliamo vivamente di specificare un prefisso distinto perché

semplifica il processo di aggiunta di un nuovo `DO_NOTHING` attributo in futuro. Per ulteriori informazioni, consulta Aggiornamento del modello di dati.

Se non si specifica un prefisso per tutti gli `DO_NOTHING` attributi, è possibile configurare un `allowedUnsignedAttributes` array che elenchi in modo esplicito tutti gli attributi che il client dovrebbe aspettarsi che non siano firmati quando li incontra durante la decrittografia. È necessario definire in modo esplicito gli attributi non firmati consentiti solo se assolutamente necessario.

Configurazione della ricerca (opzionale)

`SearchConfig`definisce la versione del beacon.

È `SearchConfig` necessario specificare il per utilizzare la crittografia ricercabile o i beacon firmati.

Algorithm Suite (opzionale)

`algorithmSuiteId`Definisce la suite di algoritmi utilizzata da AWS Database Encryption SDK.

A meno che non si specifichi esplicitamente una suite di algoritmi alternativa, AWS Database Encryption SDK utilizza la suite di algoritmi predefinita. La suite di algoritmi predefinita utilizza l'algoritmo AES-GCM con derivazione delle chiavi, firme digitali e impegno delle chiavi. Sebbene la suite di algoritmi predefinita sia probabilmente adatta alla maggior parte delle applicazioni, è possibile scegliere una suite di algoritmi alternativa. Ad esempio, alcuni modelli di fiducia sarebbero soddisfatti da una suite di algoritmi senza firme digitali. Per informazioni sulle suite di algoritmi supportate da AWS Database Encryption SDK, consulta. Suite di algoritmi supportate nel AWS Database Encryption SDK

Per selezionare la suite di algoritmi AES-GCM senza firme digitali ECDSA, includi il seguente frammento nella configurazione di crittografia delle tabelle.

```
.algorithm_suite_id(
    DbeAlgorithmSuiteId::AlgAes256GcmHkdfSha512CommitKeyEcdsaP384SymsigHmacSha384,
)
```

Aggiornamento degli AWS elementi con Database Encryption SDK

Il AWS Database Encryption SDK non supporta ddb: UpdateItem per gli elementi che includono attributi crittografati o firmati. Per aggiornare un attributo crittografato o firmato, è necessario

utilizzare ddb:. PutItem Quando specifichi la stessa chiave primaria di un elemento esistente nella tua `PutItem` richiesta, il nuovo elemento sostituisce completamente l'elemento esistente.

# Client di crittografia DynamoDB legacy

Il 9 giugno 2023, la nostra libreria di crittografia lato client è stata rinominata Database Encryption SDK. AWS Il AWS Database Encryption SDK continua a supportare le versioni precedenti di DynamoDB Encryption Client. Per ulteriori informazioni sulle diverse parti della libreria di crittografia lato client che sono state modificate con la ridenominazione, consulta. Rinomina del client di crittografia Amazon DynamoDB

Per migrare alla versione più recente della libreria di crittografia lato client Java per DynamoDB, vedere. Migrare alla versione 3.x

Argomenti

- AWS Supporto della versione di Database Encryption SDK per DynamoDB
- Come funziona il client di crittografia DynamoDB
- Concetti del client di crittografia Amazon DynamoDB
- Fornitore di materiali crittografici
- Linguaggi di programmazione disponibili per Amazon DynamoDB Encryption Client
- Modifica del modello di dati
- Risoluzione dei problemi nell'applicazione DynamoDB Encryption Client

## AWS Supporto della versione di Database Encryption SDK per DynamoDB

Gli argomenti del capitolo Legacy forniscono informazioni sulle versioni 1. x —2. x del DynamoDB Encryption Client for Java e versioni 1. x —3. x del client di crittografia DynamoDB per Python.

La tabella seguente elenca le lingue e le versioni che supportano la crittografia lato client in Amazon DynamoDB.

| Linguaggio di programma zione | Versione | Fase del ciclo di vita della versione principale dell'SDK |
|---|---|---|
| Java | Versioni 1. x | End-of-Support fase, in vigore da luglio 2022 |

| Linguaggio di programmazione | Versione | Fase del ciclo di vita della versione principale dell'SDK |
|---|---|---|
| Java | Versioni 2. x | Disponibilità generale (GA) |
| Java | Versione 3. x | Disponibilità generale (GA) |
| Python | Versioni 1. x | End-of-Support fase, in vigore da luglio 2022 |
| Python | Versioni 2. x | End-of-Support fase, in vigore da luglio 2022 |
| Python | Versioni 3. x | Disponibilità generale (GA) |

## Come funziona il client di crittografia DynamoDB

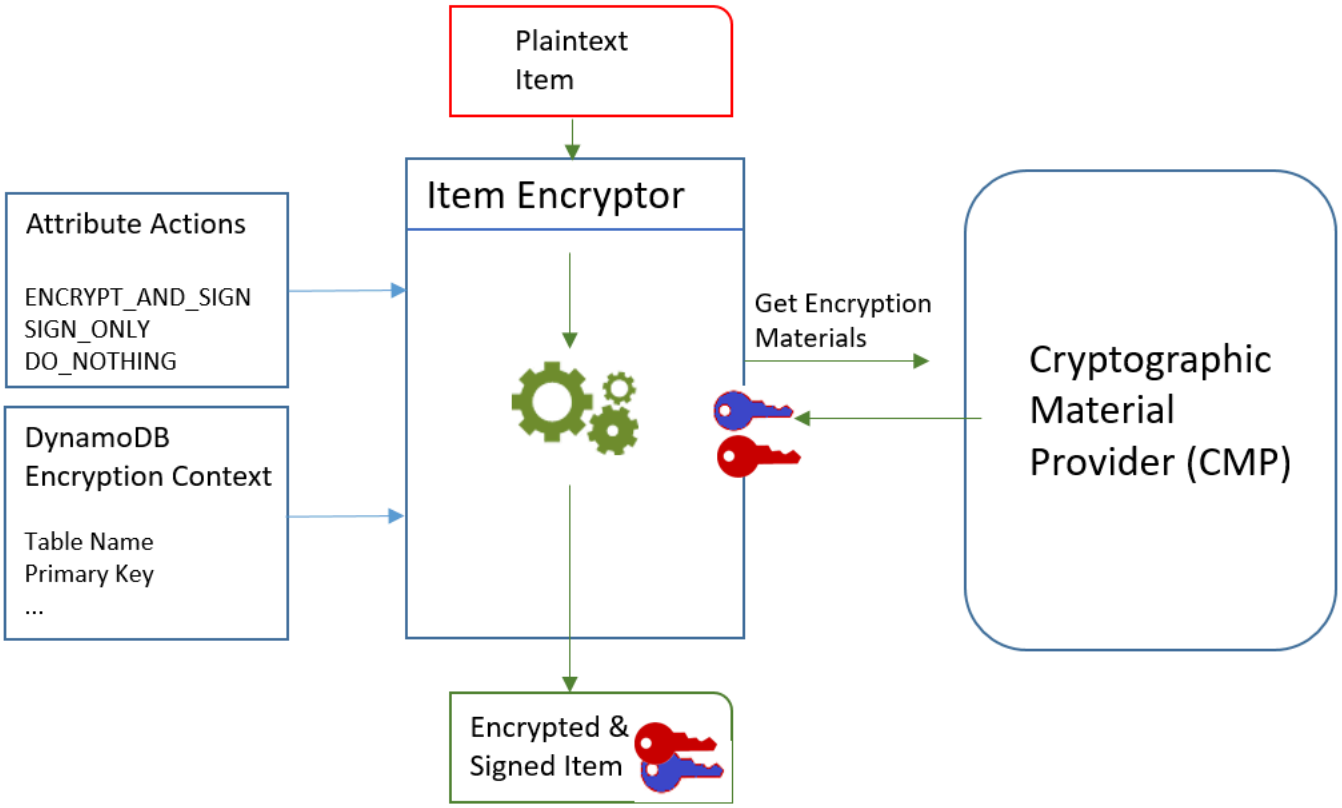
> **Note**
>
> La nostra libreria di crittografia lato client è stata rinominata Database Encryption SDK. AWS Il seguente argomento fornisce informazioni sulle versioni 1. x —2. x del DynamoDB Encryption Client for Java e versioni 1. x —3. x del client di crittografia DynamoDB per Python. Per ulteriori informazioni, consulta AWS Database Encryption SDK per il supporto della versione DynamoDB.

Il DynamoDB Encryption Client è progettato specificamente per proteggere i dati archiviati in DynamoDB. Le librerie includono implementazioni sicure che puoi estendere o utilizzare senza modificarle. Inoltre, la maggior parte degli elementi sono rappresentati da elementi astratti per consentirti di creare e utilizzare componenti personalizzati compatibili.

Crittografia e firma degli item della tabella

Alla base del DynamoDB Encryption Client c'è uno strumento di crittografia degli elementi che crittografa, firma, verifica e decrittografa gli elementi della tabella. Carica informazioni sugli elementi della tabella e istruzioni su quali item criptare e firmare. Ottiene i materiali di crittografia e le istruzioni su come utilizzarli da un provider di materiali crittografici da te selezionato e configurato.

Il seguente diagramma mostra una vista generale di questo processo:




Per crittografare e firmare un elemento della tabella, il client di crittografia DynamoDB necessita di:

- Informazioni sulla tabella. Ottiene informazioni sulla tabella da un contesto di crittografia DynamoDB fornito dall'utente. Alcuni helper ottengono le informazioni richieste da DynamoDB e creano automaticamente il contesto di crittografia DynamoDB.

  > **Note**
  >
  > Il contesto di crittografia DynamoDB nel DynamoDB Encryption Client non è correlato al contesto di crittografia in () e in. AWS Key Management Service AWS KMS AWS Encryption SDK

- Informazioni su quali attributi crittografare e firmare. Ottiene queste informazioni dalle operazioni di attributo da te fornite.
- Materiali di crittografia, incluse le chiavi di crittografia e di firma. Li ottiene da un provider di materiali crittografici (CMP) da te selezionato e configurato.

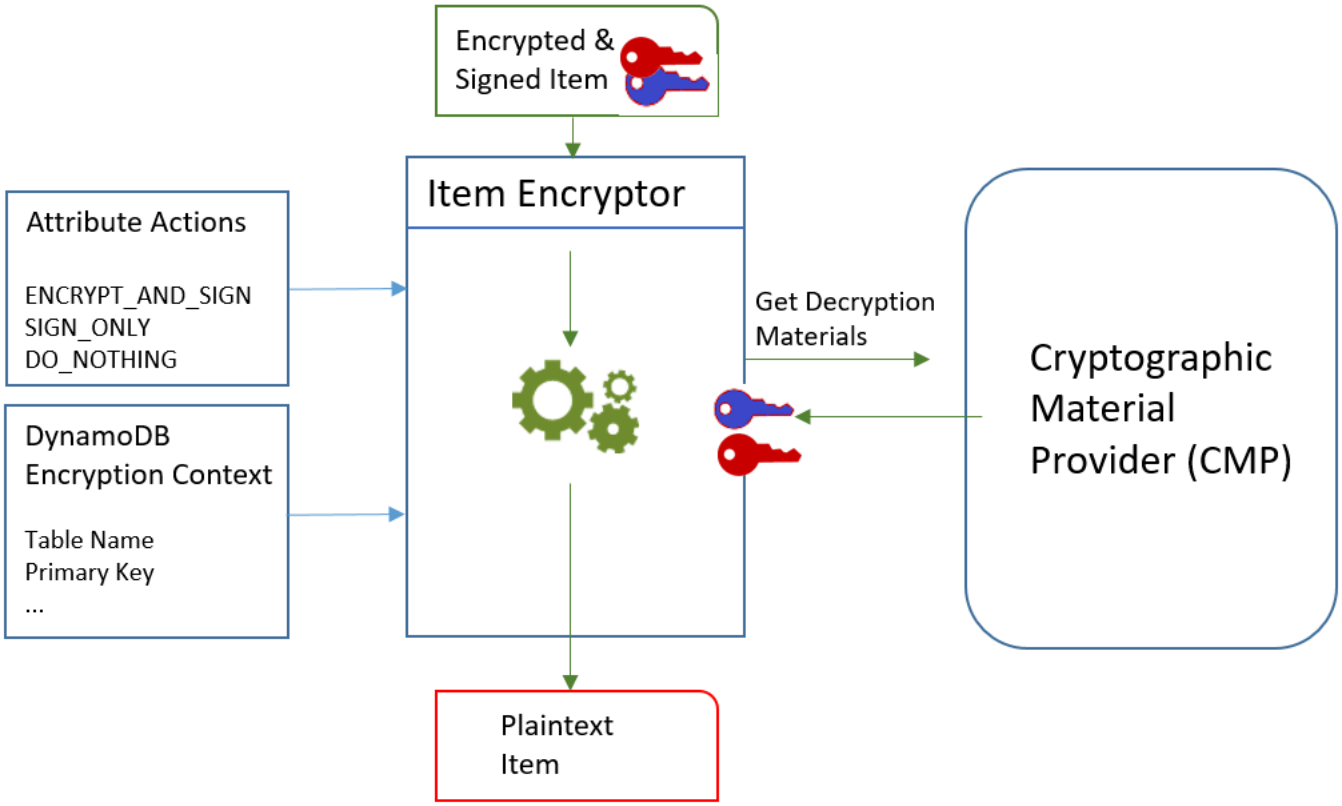
- Istruzioni per la crittografia e la firma dell'item. Il CMP aggiunge istruzioni per l'utilizzo dei materiali di crittografia, inclusi gli algoritmi di crittografia e firma, alla descrizione dei materiali effettivi.

Il componente di crittografia dell'item utilizza tutti questi elementi per crittografare e firmare l'item. Il componente di crittografia dell'item, inoltre, aggiunge due attributi all'item: un attributo di descrizione del materiale, contenente le istruzioni per la crittografia e la firma (la descrizione dei materiali effettivi), e un attributo che contiene la firma. Puoi interagire direttamente con il componente di crittografia dell'item o utilizzare le funzionalità helper che interagiscono con il componente di crittografia dell'item per implementare un comportamento predefinito sicuro.

Il risultato è un item DynamoDB contenente dati crittografati e firmati.

Verifica e decrittografia degli item della tabella

Anche questi componenti operano insieme per verificare e decrittografare l'item, come mostrato nel diagramma riportato di seguito.



Per verificare e decrittografare un elemento, il client di crittografia DynamoDB necessita degli stessi componenti, componenti con la stessa configurazione o componenti progettati appositamente per decrittografare gli elementi, come segue:

- Informazioni sulla tabella dal contesto di crittografia DynamoDB.
- Quali attributi verificare e decrittografare. ottenute dalle operazioni di attributo.
- Materiali di decrittografia, incluse le chiavi di verifica e decrittografia, ottenute dal CMP da te selezionato e configurato.

  L'item crittografato non include alcun record del CMP che è stato utilizzato per crittografarlo. Devi fornire lo stesso CMP, un CMP con la stessa configurazione o un CMP che è stato designato per decrittografare gli item.
- Informazioni sui modi in cui l'item è stato crittografato e firmato, inclusi gli algoritmi di crittografia e firma. Il client le ottiene dall'attributo di descrizione del materiale nell'item.

Il componente di crittografia dell'item utilizza tutti questi elementi per verificare e decrittografare l'item. Inoltre, rimuove gli attributi di descrizione del materiale e di firma. Il risultato è un elemento DynamoDB in testo semplice.

## Concetti del client di crittografia Amazon DynamoDB

> **Note**
>
> La nostra libreria di crittografia lato client è stata rinominata Database Encryption SDK. AWS Il seguente argomento fornisce informazioni sulle versioni 1. x —2. x del DynamoDB Encryption Client for Java e versioni 1. x —3. x del client di crittografia DynamoDB per Python. Per ulteriori informazioni, consulta AWS Database Encryption SDK per il supporto della versione DynamoDB.

Questo argomento spiega i concetti e la terminologia utilizzati in Amazon DynamoDB Encryption Client.

Per scoprire come interagiscono i componenti del DynamoDB Encryption Client, consulta. Come funziona il client di crittografia DynamoDB

Argomenti

- Provider di materiali crittografici
- Componenti di crittografia dell'item
- Operazioni di attributo
- Descrizione dei materiali
- Contesto di crittografia DynamoDB
- Archivio provider

## Provider di materiali crittografici

Quando si implementa il DynamoDB Encryption Client, una delle prime attività è selezionare un fornitore di materiali crittografici (CMP) (noto anche come fornitore di materiali di crittografia). La scelta determina gran parte del processo di implementazione rimanente.

Un CMP raccoglie, assembla e restituisce i materiali crittografici che il componente di crittografia dell'item utilizza per crittografare e firmare gli item della tabella. Il CMP determina gli algoritmi di crittografia da utilizzare e come generare e proteggere le chiavi di crittografia e firma.

Il CMP interagisce con il componente di crittografia dell'item. Il componente di crittografia dell'item richiede i materiali di crittografia e decrittografia al CMP e il CMP li restituisce al componente di crittografia dell'item. Quindi, il componente di crittografia dell'item utilizza i materiali crittografici per crittografare e firmare l'item o per verificarlo e decrittografarlo.

Il CMP viene specificato quando si configura il client. È possibile creare una CMP personalizzata compatibile o utilizzare una delle tante presenti nella libreria. CMPs La maggior parte CMPs sono disponibili per più linguaggi di programmazione.

## Componenti di crittografia dell'item

L'item encryptor è un componente di livello inferiore che esegue operazioni crittografiche per il DynamoDB Encryptor Client. Richiede materiali crittografici a un CMP, quindi utilizza i materiali che il CMP restituisce per crittografare e firmare l'item della tabella o per verificarlo e decrittografarlo.

Puoi interagire direttamente con il componente di crittografia dell'item o utilizzare gli helper forniti dalla libreria. Ad esempio, il client di crittografia DynamoDB per Java include `AttributeEncryptor` una classe di supporto che è possibile utilizzare con, anziché interagire direttamente con `DynamoDBMapper` l'crittografo degli elementi. `DynamoDBEncryptor` La libreria Python include le classi helper `EncryptedTable`, `EncryptedClient` ed `EncryptedResource` che interagiscono con il componente di crittografia dell'item per te.

## Operazioni di attributo

Le operazioni di attributo comunicano al componente di crittografia dell'item quali operazioni effettuare su ciascun attributo dell'item.

È possibile utilizzare i seguenti valori per le operazioni di attributo:

- Crittografa e firma: crittografa il valore dell'attributo. Includi l'attributo (nome e valore) nella firma dell'item.
- Solo firma: include l'attributo nella firma dell'articolo.
- Non fare nulla: non crittografare o firmare l'attributo.

Utilizza Encrypt and sign (Crittografa e firma) per tutti gli attributi in grado di memorizzare dati sensibili. Per gli attributi delle chiavi primarie (chiave di partizione e chiave di ordinamento), utilizza Sign only (Firma soltanto). L'attributo di descrizione del materiale e l'attributo di firma non sono firmati o crittografati. Non è necessario specificare operazioni di attributo per questi attributi.

Scegli attentamente le operazioni di attributo. In caso di dubbio, usa Encrypt and sign (Crittografa e firma). Dopo aver utilizzato il DynamoDB Encryption Client per proteggere gli elementi della tabella, non è possibile modificare l'azione per un attributo senza rischiare un errore di convalida della firma. Per informazioni dettagliate, consultare Modifica del modello di dati.

> **Warning**
>
> Non crittografare gli attributi che vengono usati per la chiave primaria. Devono rimanere in testo semplice in modo che DynamoDB possa trovare l'elemento senza eseguire una scansione completa della tabella.

Se il contesto di crittografia DynamoDB identifica gli attributi della chiave primaria, il client genererà un errore se si tenta di crittografarli.

La tecnica da utilizzare per specificare le operazioni di attributo è diversa per ogni linguaggio di programmazione. Potrebbe anche essere specifica per le classi helper che utilizzi.

Per ulteriori informazioni, consulta la documentazione relativa al tuo linguaggio di programmazione.

- Python

- Java

## Descrizione dei materiali

La descrizione dei materiali per un item della tabella crittografato comprende informazioni, come gli algoritmi di crittografia, sui modi in cui l'item della tabella viene crittografato e firmato. Il CMP registra la descrizione dei materiali poiché assembla i materiali crittografici per la crittografia e la firma. Successivamente, quando deve assemblare i materiali crittografici per verificare e decrittografare l'item, utilizza la descrizione dei materiali come guida.

Nel DynamoDB Encryption Client, la descrizione del materiale si riferisce a tre elementi correlati:

Descrizione dei materiali richiesti

Alcuni fornitori di materiali crittografici (CMPs) consentono di specificare opzioni avanzate, come un algoritmo di crittografia. Per indicare le tue scelte, aggiungi coppie nome-valore alla proprietà di descrizione del materiale del contesto di crittografia DynamoDB nella tua richiesta di crittografia di un elemento della tabella. Questo elemento è noto come descrizione dei materiali richiesti. I valori validi nella descrizione dei materiali richiesti sono definiti dal CMP scelto.

> **Note**
>
> Poiché la descrizione dei materiali può sovrascrivere i valori predefiniti, ti consigliamo di omettere la descrizione dei materiali richiesti se non hai un valido motivo per utilizzarla.

Descrizione dei materiali effettivi

La descrizione del materiale restituita dai provider di materiali crittografici (CMPs) è nota come descrizione effettiva del materiale. Descrive i valori effettivi che il CMP ha utilizzato quando ha assemblato i materiali crittografici. In genere comprende l'eventuale descrizione dei materiali richiesti, con aggiunte e modifiche.

Attributo di descrizione del materiale

Il client salva la descrizione dei materiali effettivi nell'attributo di descrizione del materiale dell'item crittografato. Il nome dell'attributo di descrizione del materiale è `amzn-ddb-map-desc` e il suo valore è la descrizione dei materiali effettivi. Il client utilizza i valori dell'attributo di descrizione del materiale per verificare e decrittografare l'item.

## Contesto di crittografia DynamoDB

Il contesto di crittografia DynamoDB fornisce informazioni sulla tabella e sull'elemento al fornitore di materiali crittografici (CMP). Nelle implementazioni avanzate, il contesto di crittografia DynamoDB può includere una descrizione del materiale richiesta.

Quando si crittografano gli elementi della tabella, il contesto di crittografia DynamoDB è associato crittograficamente ai valori degli attributi crittografati. Quando si decrittografa, se il contesto di crittografia DynamoDB non corrisponde esattamente, con distinzione tra maiuscole e minuscole, al contesto di crittografia DynamoDB utilizzato per crittografare, l'operazione di decrittografia fallisce. Se interagisci direttamente con l'item encryptor, devi fornire un contesto di crittografia DynamoDB quando chiami un metodo di crittografia o decrittografia. La maggior parte degli helper crea automaticamente il contesto di crittografia DynamoDB.

> **Note**
>
> Il contesto di crittografia DynamoDB nel DynamoDB Encryption Client non è correlato al contesto di crittografia in () e in. AWS Key Management Service AWS KMS AWS Encryption SDK

Il contesto di crittografia DynamoDB può includere i seguenti campi. Tutti i campi e i valori sono facoltativi.

- Nome tabella
- Nome della chiave di partizione
- Nome della chiave di ordinamento
- Coppie nome/valore degli attributi
- Descrizione dei materiali richiesti

## Archivio provider

Un provider store è un componente che restituisce materiali crittografici providers (). CMPs Il provider store può creare CMPs o ottenere i file da un'altra fonte, ad esempio un altro provider store. Il provider store salva le versioni di CMPs ciò che crea in una memoria persistente in cui ogni CMP archiviato è identificato dal nome del materiale del richiedente e dal numero di versione.

Il provider più recente del client di crittografia DynamoDB lo ottiene CMPs da un archivio provider, ma è possibile utilizzare l'archivio provider per CMPs rifornire qualsiasi componente. Ogni provider più recente è associato a un provider store, ma un provider store può CMPs rifornire più richiedenti su più host.

Il provider store crea nuove versioni di CMPs on demand e restituisce versioni nuove ed esistenti. Restituisce inoltre l'ultimo numero di versione per un dato nome di materiale. Questo consente al richiedente di sapere quando l'archivio provider dispone di una nuova versione del suo CMP che è possibile richiedere.

Il client di crittografia DynamoDB include MetaStoreun, ovvero un provider store che crea CMPs Wrapped con chiavi archiviate in DynamoDB e crittografate utilizzando un client di crittografia DynamoDB interno.

Ulteriori informazioni:

- Archivio provider: Java, Python
- MetaStore: Java, Python

## Fornitore di materiali crittografici

> **Note**
>
> La nostra libreria di crittografia lato client è stata rinominata Database Encryption SDK. AWS Il seguente argomento fornisce informazioni sulle versioni 1. x —2. x del DynamoDB Encryption Client for Java e versioni 1. x —3. x del client di crittografia DynamoDB per Python. Per ulteriori informazioni, consulta AWS Database Encryption SDK per il supporto della versione DynamoDB.

Una delle decisioni più importanti da prendere quando si utilizza il DynamoDB Encryption Client è la selezione di un fornitore di materiali crittografici (CMP). Il CMP raccoglie e restituisce i materiali crittografici al componente di crittografia dell'item. Determina inoltre la modalità di generazione delle chiavi di crittografia e di firma, gli algoritmi di crittografia e firma utilizzati e se i nuovi materiali di chiave devono essere creati o riutilizzati per ciascun item.

Puoi scegliere una CMP tra le implementazioni fornite nelle librerie DynamoDB Encryption Client o creare una CMP personalizzata compatibile. La scelta del CMP potrebbe inoltre dipendere dal linguaggio di programmazione utilizzato.

Questo argomento descrive le più comuni CMPs e offre alcuni consigli per aiutarvi a scegliere quella migliore per la vostra applicazione.

Provider di materiali KMS diretto

Direct KMS Materials Provider protegge gli elementi della tabella in AWS KMS keymodo che non rimangano mai AWS Key Management Service(AWS KMS) non crittografati. Non è necessario che l'applicazione generi o gestisca i materiali crittografici. Poiché utilizza il AWS KMS key per generare chiavi di crittografia e firma uniche per ogni elemento, questo provider chiama AWS KMS ogni volta che crittografa o decrittografa un elemento.

Se utilizzate AWS KMS e una sola AWS KMS chiamata per transazione è pratica per la vostra applicazione, questo provider è una buona scelta.

Per informazioni dettagliate, consultare Provider di materiali KMS diretto.

Provider di materiali di sottoposti a wrapping (CMP di sottoposti a wrapping)

Il Wrapped Materials Provider (Wrapped CMP) consente di generare e gestire le chiavi di wrapping e firma al di fuori del DynamoDB Encryption Client.

Il CMP di sottoposti a wrapping genera una chiave di crittografia univoca per ciascun item. Utilizza quindi le chiavi di firma e di crittografia (o di annullamento della crittografia) da te fornite. In questo modo, puoi determinare la modalità di generazione delle chiavi di crittografia e di firma e se sono univoche o se vengono riutilizzate per ciascun item. Wrapped CMP è un'alternativa sicura al Direct KMS Provider per applicazioni che non utilizzano e possono gestire in sicurezza materiali crittografici. AWS KMS

Per informazioni dettagliate, consultare Provider di materiali sottoposti a wrapping.

Provider più recente

Il provider più recente è un provider di materiali crittografici (CMP) progettato per funzionare con gli archivi del provider. Proviene CMPs dall'archivio del provider e ottiene i materiali crittografici che restituisce da. CMPs Il Provider più recente in genere utilizza ciascun CMP per soddisfare più richieste di materiali crittografici, ma puoi utilizzare le funzioni dell'archivio del provider per controllare in quale misura vengono riutilizzati i materiali, determinare la frequenza di rotazione del CMP e persino modificare il tipo di CMP utilizzato senza cambiare il Provider più recente.

Puoi utilizzare il Provider più recente con qualsiasi archivio del provider compatibile. Il client di crittografia DynamoDB include MetaStore un, che è un provider store che restituisce Wrapped. CMPs

Il Provider più recente rappresenta una scelta adeguata per le applicazioni che devono ridurre al minimo le chiamate alla relativa origine crittografica e per le applicazioni che possono riutilizzare alcuni materiali crittografici senza violare i requisiti di sicurezza. Ad esempio, consente di proteggere i materiali crittografici utilizzando un comando AWS KMS keyin AWS Key Management Service(AWS KMS) senza chiamare AWS KMS ogni volta che si crittografa o decrittografa un elemento.

Per informazioni dettagliate, consultare Provider più recente.

Provider di materiali statici

Lo Static Materials Provider è progettato per test, proof-of-concept dimostrazioni e compatibilità con le versioni precedenti. Non genera nessun materiale crittografico univoco per gli item. Restituisce le stesse chiavi di crittografia e di firma da te fornite che vengono utilizzate direttamente per crittografare, decrittografare e firmare gli item della tabella.

> **Note**
>
> Il Provider di statici asimmetrico nella libreria Java non è un provider di statici. Fornisce soltanto costruttori alternativi per il CMP di sottoposti a wrapping. Puoi utilizzarlo nell'ambiente di produzione, ma ti consigliamo di utilizzare direttamente il CMP di sottoposti a wrapping ogni qualvolta sia possibile.

Argomenti

- Provider di materiali KMS diretto
- Provider di materiali sottoposti a wrapping
- Provider più recente
- Provider di materiali statici

## Provider di materiali KMS diretto

> **Note**
>
> La nostra libreria di crittografia lato client è stata rinominata Database Encryption SDK. AWS Il seguente argomento fornisce informazioni sulle versioni 1. x —2. x del DynamoDB Encryption Client for Java e versioni 1. x —3. x del client di crittografia DynamoDB per Python. Per ulteriori informazioni, consulta AWS Database Encryption SDK per il supporto della versione DynamoDB.

Direct KMS Materials Provider (Direct KMS Provider) protegge gli elementi della tabella in modo AWS KMS keyche non rimangano mai () non crittografati. AWS Key Management ServiceAWS KMS Questo provider di materiali crittografici restituisce una chiave di crittografia e una chiave di firma univoche per ogni item della tabella. A tal fine, chiama AWS KMS ogni volta che si crittografa o decrittografa un elemento.

Se stai elaborando elementi DynamoDB ad alta frequenza e su larga scala, potresti superare i limiti, causando ritardi AWS KMS requests-per-secondnell'elaborazione. Se devi superare un limite, crea un caso nel Centro.Supporto AWS Potresti anche prendere in considerazione l'utilizzo di un fornitore di materiali crittografici con un riutilizzo limitato delle chiavi, come il Most Recent Provider.

Per utilizzare Direct KMS Provider, il chiamante deve disporre di almeno uno AWS KMS key e del permesso di chiamare GenerateDataKeye decriptare le operazioni su. Account AWS AWS KMS key AWS KMS key Deve essere una chiave di crittografia simmetrica; il DynamoDB Encryption Client non supporta la crittografia asimmetrica. Se utilizzi una tabella globale DynamoDB, potresti voler specificare AWS KMS una chiave multiregione. Per informazioni dettagliate, consultare Come utilizzarlo.

> **Note**
>
> Quando si utilizza Direct KMS Provider, i nomi e i valori degli attributi della chiave primaria vengono visualizzati in testo semplice nel contesto di AWS KMS crittografia e nei registri delle operazioni correlate. AWS CloudTrail AWS KMS Tuttavia, il DynamoDB Encryption Client non espone mai il testo in chiaro di alcun valore di attributo crittografato.

Il Direct KMS Provider è uno dei numerosi fornitori di materiali crittografici (CMPs) supportati dal DynamoDB Encryption Client. Per informazioni sull'altro, consulta. CMPs Fornitore di materiali crittografici

Per il codice di esempio, consulta:

- Java: AwsKmsEncryptedItem
- Python:, aws-kms-encrypted-tableaws-kms-encrypted-item

Argomenti

- Come utilizzarlo
- Come funziona

Come utilizzarlo

Per creare un Direct KMS Provider, utilizza il parametro key ID per specificare una chiave KMS di crittografia simmetrica nel tuo account. Il valore del parametro key ID può essere l'ID chiave, l'ARN della chiave, il nome dell'alias o l'alias ARN di. AWS KMS keyPer i dettagli sugli identificatori chiave, consulta Identificatori chiave nella Guida per gli sviluppatori.AWS Key Management Service

Il provider Direct KMS richiede una chiave KMS di crittografia simmetrica. Non è possibile utilizzare una chiave KMS asimmetrica. Tuttavia, puoi utilizzare una chiave KMS multiregionale, una chiave KMS con materiale chiave importato o una chiave KMS in un archivio di chiavi personalizzato. È necessario disporre dell'autorizzazione kms: GenerateDataKey e KMS:Decrypt sulla chiave KMS. Pertanto, è necessario utilizzare una chiave gestita dal cliente, non una chiave KMS gestita o di proprietà. AWS AWS

Il client di crittografia DynamoDB per Python determina la regione per la AWS KMS chiamata dalla regione nel valore del parametro ID chiave, se ne include una. Altrimenti, utilizza la Regione nel AWS KMS client, se ne specifichi una, o la Regione che configuri in. AWS SDK per Python (Boto3) Per informazioni sulla selezione della regione in Python, consulta Configuration in the AWS SDK for Python (Boto3) API Reference.

Il client di crittografia DynamoDB per Java determina la regione per la AWS KMS chiamata dalla regione del client, se AWS KMS il client specificato include una regione. Altrimenti, utilizza la regione configurata in. AWS SDK per Java Per informazioni sulla selezione della regione in AWS SDK per Java, consulta la Regione AWS selezione nella Guida per gli AWS SDK per Java sviluppatori.

Java

```
// Replace the example key ARN and Region with valid values for your application
final String keyArn = 'arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab'
final String region = 'us-west-2'

final AWSKMS kms = AWSKMSClientBuilder.standard().withRegion(region).build();
final DirectKmsMaterialProvider cmp = new DirectKmsMaterialProvider(kms, keyArn);
```

Python

L'esempio seguente utilizza la chiave ARN per specificare. AWS KMS key Se l'identificatore della chiave non include un Regione AWS, il client di crittografia DynamoDB ottiene la regione dalla sessione Botocore configurata, se presente, o dalle impostazioni predefinite di Boto.

```
# Replace the example key ID with a valid value
kms_key = 'arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab'
kms_cmp = AwsKmsCryptographicMaterialsProvider(key_id=kms_key)
```

Se utilizzi tabelle globali di Amazon DynamoDB, ti consigliamo di crittografare i dati con una chiave multiregionale. AWS KMS Le chiavi multiregionali sono disponibili AWS KMS keys in diversi formati e possono essere utilizzate in Regioni AWS modo intercambiabile perché hanno lo stesso ID di chiave e lo stesso materiale chiave. Per i dettagli, consulta Uso delle chiavi multiregionali nella Guida per gli sviluppatori.AWS Key Management Service

> **Note**
>
> Se si utilizza la versione 2017.11.29 delle tabelle globali, è necessario impostare le azioni degli attributi in modo che i campi di replica riservati non siano crittografati o firmati. Per informazioni dettagliate, consultare Problemi con le tabelle globali delle versioni precedenti.

Per utilizzare una chiave multiregione con il DynamoDB Encryption Client, crea una chiave multiregione e replicala nelle regioni in cui viene eseguita l'applicazione. Quindi configura il provider Direct KMS per utilizzare la chiave multiregione nella regione in cui il DynamoDB Encryption Client chiama. AWS KMS

L'esempio seguente configura il DynamoDB Encryption Client per crittografare i dati nella regione Stati Uniti orientali (Virginia settentrionale) (us-east-1) e decrittografarli nella regione Stati Uniti occidentali (Oregon) (us-west-2) utilizzando una chiave multiregionale.

Java

In questo esempio, il client di crittografia DynamoDB ottiene la regione per la AWS KMS chiamata dalla regione del client. AWS KMS Il `keyArn` valore identifica una chiave multiregionale nella stessa regione.

```
// Encrypt in us-east-1

// Replace the example key ARN and Region with valid values for your application
final String usEastKey = 'arn:aws:kms:us-east-1:111122223333:key/
mrk-1234abcd12ab34cd56ef1234567890ab'
final String region = 'us-east-1'

final AWSKMS kms = AWSKMSClientBuilder.standard().withRegion(region).build();
final DirectKmsMaterialProvider cmp = new DirectKmsMaterialProvider(kms, usEastKey);
```

```
// Decrypt in us-west-2

// Replace the example key ARN and Region with valid values for your application
final String usWestKey = 'arn:aws:kms:us-west-2:111122223333:key/
mrk-1234abcd12ab34cd56ef1234567890ab'
final String region = 'us-west-2'

final AWSKMS kms = AWSKMSClientBuilder.standard().withRegion(region).build();
final DirectKmsMaterialProvider cmp = new DirectKmsMaterialProvider(kms, usWestKey);
```

Python

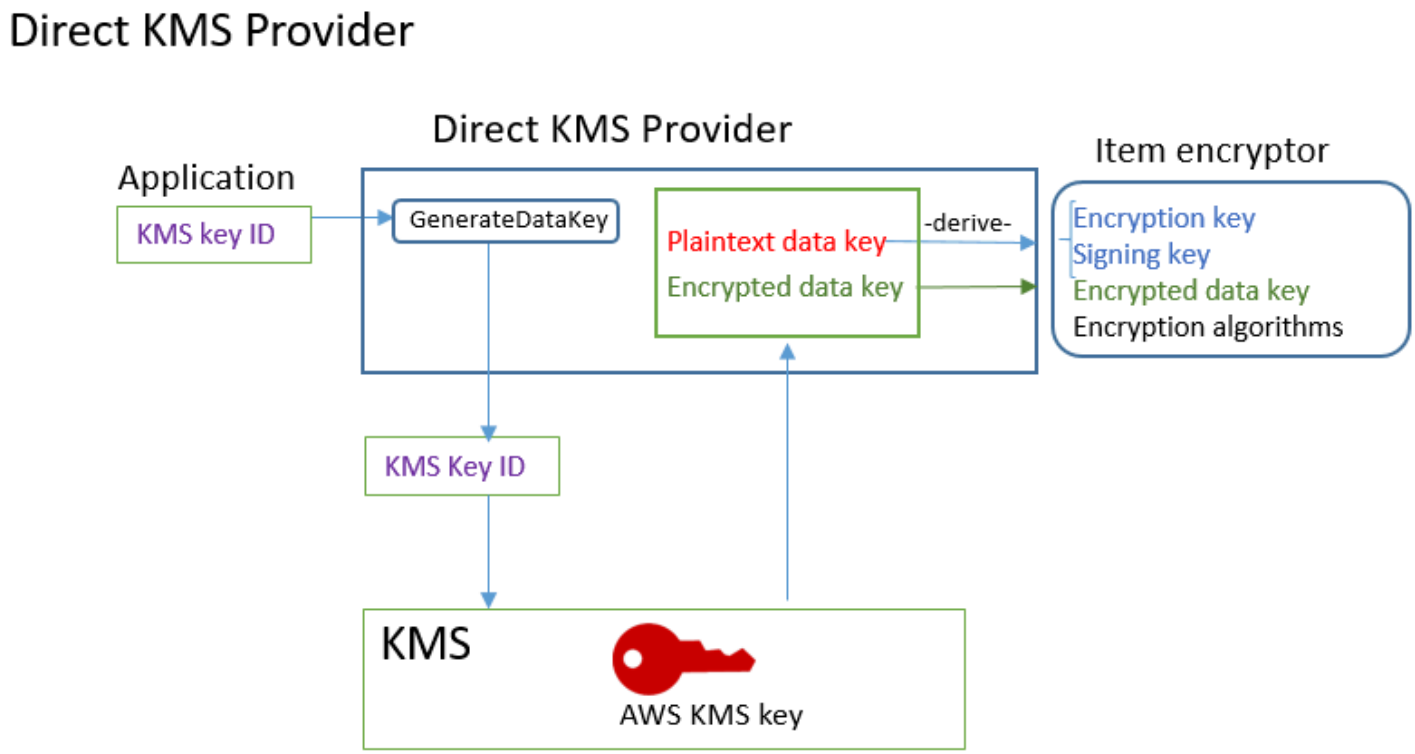
In questo esempio, il client di crittografia DynamoDB ottiene la regione per la AWS KMS chiamata dalla regione nella chiave ARN.

```
# Encrypt in us-east-1

# Replace the example key ID with a valid value
us_east_key = 'arn:aws:kms:us-east-1:111122223333:key/
mrk-1234abcd12ab34cd56ef1234567890ab'
kms_cmp = AwsKmsCryptographicMaterialsProvider(key_id=us_east_key)
```

```
# Decrypt in us-west-2

# Replace the example key ID with a valid value
us_west_key = 'arn:aws:kms:us-west-2:111122223333:key/
mrk-1234abcd12ab34cd56ef1234567890ab'
kms_cmp = AwsKmsCryptographicMaterialsProvider(key_id=us_west_key)
```

Come funziona

Il provider Direct KMS restituisce chiavi di crittografia e firma protette da una AWS KMS key password specificata dall'utente, come illustrato nel diagramma seguente.




- Per generare materiali di crittografia, il provider Direct KMS richiede di AWS KMS generare una chiave dati univoca per ogni elemento utilizzando una AWS KMS key chiave specificata dall'utente. Deriva le chiavi di crittografia e di firma dell'item dalla copia di testo non crittografato della chiave di dati e le restituisce, insieme alla chiave di dati crittografata, archiviata nell'attributo di descrizione del materiale dell'item.

  Il componente di crittografia dell'item utilizza le chiavi di crittografia e di firma e le rimuove dalla memoria il prima possibile. Nell'item crittografato viene salvata soltanto la copia crittografata della chiave di dati da cui queste chiavi sono state derivate.

- Per generare materiali di decrittografia, il provider Direct KMS chiede di AWS KMS decrittografare la chiave dati crittografata. Quindi, deriva le chiavi di verifica e di firma dalla chiave di dati con testo non crittografato e le restituisce al componente di crittografia dell'item.

  Il componente di crittografia dell'item effettua la verifica dell'item e, in assenza di errori, decrittografa i valori crittografati. Quindi, rimuove le chiavi dalla memoria il prima possibile.

### Ottenere materiali di crittografia

Questa sezione descrive nei dettagli gli input, gli output e l'elaborazione del provider KMS diretto al momento della ricezione di una richiesta di materiali di crittografia dal componente di crittografia dell'item.

Input (dall'applicazione)

- L'ID della chiave di un. AWS KMS key

Input (dal componente di crittografia dell'item)

- Contesto di crittografia DynamoDB

Output (sul componente di crittografia dell'item)

- Chiave di crittografia (testo non crittografato)
- Chiave di firma
- Nella descrizione dei materiali effettivi: questi valori vengono salvati nell'attributo di descrizione del materiale aggiunto all'item dal client.
  - amzn-ddb-env-key: chiave dati con codifica Base64 crittografata da AWS KMS key
  - amzn-ddb-env-alg: algoritmo di crittografia, per impostazione predefinita AES/256
  - amzn-ddb-sig-alg: algoritmo di firma, per impostazione predefinita, Hmac/256 SHA256
  - amzn-ddb-wrap-alg: km

Processing

1. Il provider Direct KMS invia AWS KMS una richiesta per utilizzare quanto specificato AWS KMS key per generare una chiave dati univoca per l'articolo. L'operazione restituisce una chiave di testo

non crittografato e una copia che viene crittografata con la AWS KMS key. Quest'ultima è nota come materiale di chiave iniziale.

La richiesta include i valori seguenti sotto forma di testo non crittografato nel contesto di crittografia di AWS KMS. Questi valori non segreti sono crittograficamente legati all'oggetto crittografato; pertanto, per effettuare la decrittografia è necessario lo stesso contesto di crittografia. È possibile utilizzare questi valori per identificare la chiamata AWS KMS ai AWS CloudTrail log.

- amzn-ddb-env-alg — Algoritmo di crittografia, di default AES/256
- amzn-ddb-sig-alg — Algoritmo di firma, di default Hmac/256 SHA256
- (Opzionale) — aws-kms-table *`table name`*
- (Facoltativo) *`partition key name`* — *`partition key value`* (i valori binari sono codificati in Base64)
- (Facoltativo) *`sort key name`* — (i valori binari sono codificati in *`sort key value`* Base64)

Il provider Direct KMS ottiene i valori per il contesto di AWS KMS crittografia dal contesto di crittografia DynamoDB per l'elemento. Se il contesto di crittografia DynamoDB non include un valore, ad esempio il nome della tabella, quella coppia nome-valore viene omessa dal contesto di crittografia. AWS KMS

2. Il Provider KMS diretto deriva la chiave di crittografia e la chiave di firma simmetriche dalla chiave di dati. Per impostazione predefinita, utilizza Secure Hash Algorithm (SHA) 256 e la Key Derivation Function RFC5869 basata su HMAC per derivare una chiave di crittografia simmetrica AES a 256 bit e una chiave di firma HMAC-SHA-256 a 256 bit.
3. Il Provider KMS diretto restituisce l'output al componente di crittografia dell'item.
4. Il componente di crittografia dell'item utilizza la chiave di crittografia per crittografare gli attributi specificati e la chiave di firma per firmarli, tramite gli algoritmi specificati nella descrizione dei materiali effettivi. Rimuove le chiavi di testo non crittografato dalla memoria il prima possibile.

### Ottenere materiali di decrittografia

Questa sezione descrive nei dettagli gli input, gli output e l'elaborazione del provider KMS diretto al momento della ricezione di una richiesta di materiali di decrittografia dal componente di crittografia dell'item.

Input (dall'applicazione)

- L'ID della chiave di un. AWS KMS key

Il valore dell'ID chiave può essere l'ID della chiave, l'ARN della chiave, il nome dell'alias o l'alias ARN di. AWS KMS keyTutti i valori che non sono inclusi nell'ID della chiave, come la regione, devono essere disponibili nel profilo denominato.AWS L'ARN chiave fornisce tutti i valori necessari. AWS KMS

Input (dal componente di crittografia dell'item)

- Una copia del contesto di crittografia DynamoDB che contiene il contenuto dell'attributo di descrizione del materiale.

Output (sul componente di crittografia dell'item)

- Chiave di crittografia (testo non crittografato)
- Chiave di firma

Processing

1. Il provider Direct KMS ottiene la chiave dei dati crittografati dall'attributo di descrizione del materiale nell'elemento crittografato.
2. Chiede AWS KMS di utilizzare quanto specificato AWS KMS key per decrittografare la chiave dati crittografata. L'operazione restituisce una chiave di testo non crittografato.

   Questa richiesta deve utilizzare lo stesso contesto di crittografia di AWS KMS utilizzato per generare e crittografare la chiave di dati.

   - aws-kms-table – *`table name`*
   - *`partition key name`*— *`partition key value`* (i valori binari sono codificati in Base64)
   - (Facoltativo) *`sort key name`* — (i valori binari sono codificati in *`sort key value`* Base64)
   - amzn-ddb-env-alg — Algoritmo di crittografia, di default AES/256
   - amzn-ddb-sig-alg — Algoritmo di firma, di default Hmac/256 SHA256
3. Il provider Direct KMS utilizza Secure Hash Algorithm (SHA) 256 e la funzione di derivazione delle chiavi RFC5869 basata su HMAC per derivare una chiave di crittografia simmetrica AES a 256 bit e una chiave di firma HMAC-SHA-256 a 256 bit dalla chiave dati.
4. Il Provider KMS diretto restituisce l'output al componente di crittografia dell'item.

5. Il componente di crittografia dell'item utilizza la chiave di firma per verificare l'item. Se l'operazione riesce, utilizza la chiave di crittografia simmetrica per decrittografare i valori di attributo crittografati. Queste operazioni utilizzano gli algoritmi di crittografia e di firma specificati nella descrizione dei materiali effettivi. Il componente di crittografia dell'item rimuove le chiavi di testo non crittografato dalla memoria il prima possibile.

## Provider di materiali sottoposti a wrapping

> **Note**
>
> La nostra libreria di crittografia lato client è stata rinominata Database Encryption SDK. AWS Il seguente argomento fornisce informazioni sulle versioni 1. x —2. x del DynamoDB Encryption Client for Java e versioni 1. x —3. x del client di crittografia DynamoDB per Python. Per ulteriori informazioni, consulta AWS Database Encryption SDK per il supporto della versione DynamoDB.

Il Wrapped Materials Provider (Wrapped CMP) consente di utilizzare chiavi di wrapping e firma da qualsiasi fonte con il DynamoDB Encryption Client. La Wrapped CMP non dipende da alcun servizio. AWS Tuttavia, devi generare e gestire le chiavi di wrapping e firma all'esterno del client, nonché fornire le chiavi corrette per verificare e decrittografare l'item.

Il CMP di sottoposti a wrapping genera una chiave di crittografia item univoca per ciascun item. Esegue il wrapping della chiave di crittografia dell'item con la chiave di wrapping che hai fornito e salva la chiave di crittografia dell'item sottoposta a wrapping nell'attributo di descrizione del materiale dell'item. Poiché sei tu a fornire le chiavi di wrapping e di firma, puoi capire come sono generate le chiavi di wrapping e firma e se sono univoche per ciascun item o riutilizzate.

Il CMP di sottoposti a wrapping è un'implementazione sicura e un'ottima scelta per le applicazioni in grado di gestire i materiali crittografici.

Wrapped CMP è uno dei numerosi fornitori di materiali crittografici (CMPs) supportati dal DynamoDB Encryption Client. Per informazioni sull'altro, vedere. CMPs Fornitore di materiali crittografici

Per il codice di esempio, consulta:

- Java: AsymmetricEncryptedItem
- Python:, wrapped-rsa-encrypted-tablewrapped-symmetric-encrypted-table

Argomenti



## Come utilizzarlo

Per creare un CMP di sottoposti a wrapping, specifica una chiave di wrapping (necessaria per la crittografia), una chiave di annullamento del wrapping (necessaria per la decrittografia) e una chiave di firma. Le chiavi devono essere fornite al momento di crittografare e decrittografare gli item.

Le chiavi di wrapping, annullamento del wrapping e firma possono essere chiavi simmetriche o coppie di chiavi asimmetriche.

Java

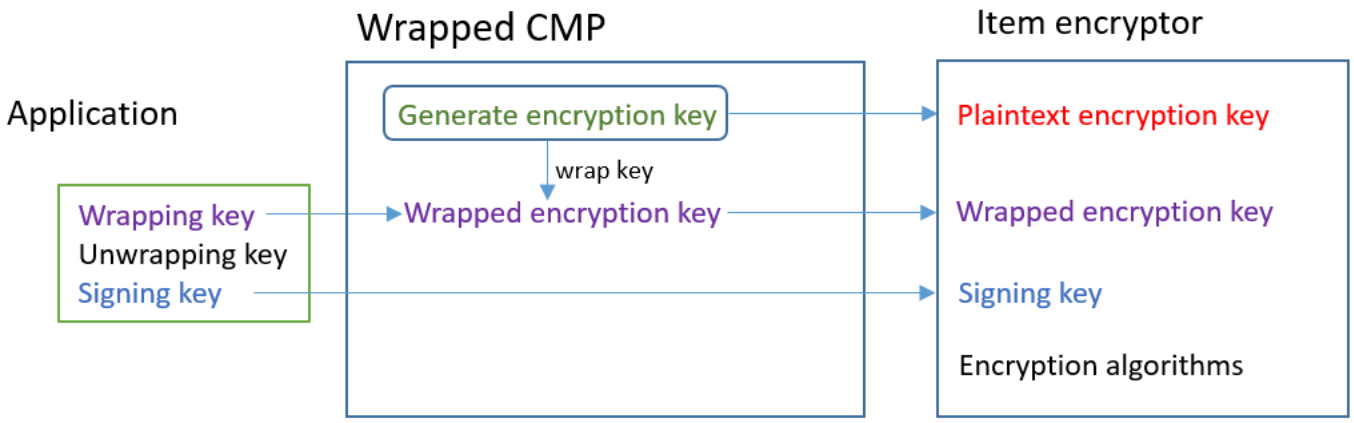
```
// This example uses asymmetric wrapping and signing key pairs
final KeyPair wrappingKeys = ...
final KeyPair signingKeys = ...

final WrappedMaterialsProvider cmp =
    new WrappedMaterialsProvider(wrappingKeys.getPublic(),
                                 wrappingKeys.getPrivate(),
                                 signingKeys);
```

Python

```
# This example uses symmetric wrapping and signing keys
wrapping_key = ...
signing_key  = ...

wrapped_cmp = WrappedCryptographicMaterialsProvider(
    wrapping_key=wrapping_key,
    unwrapping_key=wrapping_key,
    signing_key=signing_key
)
```

Come funziona

Il CMP di sottoposti a wrapping genera una nuova chiave di crittografia item per ciascun item. Utilizza le chiavi di wrapping, annullamento del wrapping e firma da te fornite, come mostrato nel diagramma mostrato di seguito.




Ottenere materiali di crittografia

In questa sezione vengono descritti nei dettagli gli input, gli output e l'elaborazione eseguita dal provider di materiali sottoposti a wrapping (CMP di sottoposti a wrapping) al momento della ricezione di una richiesta di materiali di crittografia.

Input (dall'applicazione)

- Chiave di wrapping: una chiave AES simmetrica o una chiave pubblica RSA. Necessaria se alcuni valori degli attributi sono crittografati. In caso contrario, è facoltativa e viene ignorata.
- Chiave di annullamento del wrapping: facoltativa e ignorata.
- Chiave di firma

Input (dal componente di crittografia dell'item)

- Contesto di crittografia DynamoDB

Output (sul componente di crittografia dell'item)

- Chiave di crittografia dell'item di testo normale
- Chiave di firma (invariata)

- Descrizione dei materiali effettivi: questi valori vengono salvati nell'attributo di descrizione del materiale che il client aggiunge all'item.
  - `amzn-ddb-env-key`: chiave di crittografia item sottoposta a wrapping Base64-encoded
  - `amzn-ddb-env-alg`: algoritmo di crittografia utilizzato per crittografare l'item. Il valore predefinito è AES-256-CBC.
  - `amzn-ddb-wrap-alg`: l'algoritmo di wrapping che il CMP di sottoposti a wrapping ha utilizzato per eseguire il wrapping della chiave di crittografia item. Se la chiave di wrapping è una chiave AES, viene sottoposta a wrapping utilizzando `AES-Keywrap` senza riempimenti come indicato in RFC 3394. Se la chiave di wrapping è una chiave RSA, la chiave viene crittografata utilizzando RSA OAEP con padding. MGF1

Processing

La crittografia di un item richiede una chiave di wrapping e una chiave di firma. La chiave di annullamento del wrapping è facoltativa e viene ignorata.

1. Il CMP di sottoposti a wrapping genera una chiave di crittografia item simmetrica univoca per l'item della tabella.
2. Utilizza la chiave di wrapping da te specificata per eseguire il wrapping della chiave di crittografia item. Quindi, la rimuove dalla memoria il prima possibile.
3. Restituisce la chiave di crittografia dell'item in testo normale, la chiave di firma da te fornita e una descrizione dei materiali effettivi che include la chiave di crittografia dell'item sottoposta a wrapping e gli algoritmi di crittografia e wrapping.
4. Il componente di crittografia dell'item utilizza la chiave di crittografia testo normale per crittografare l'item. Utilizza la chiave di firma da te fornita per firmare l'item. Quindi, rimuove le chiavi di testo normale dalla memoria il prima possibile. Copia i campi della descrizione dei materiali effettivi, inclusa la chiave di crittografia sottoposta a wrapping (`amzn-ddb-env-key`) nell'attributo di descrizione del materiale dell'item.

Ottenere materiali di decrittografia

In questa sezione vengono descritti nei dettagli gli input, gli output e l'elaborazione eseguita dal provider di materiali sottoposti a wrapping (CMP di sottoposti a wrapping) al momento della ricezione di una richiesta di materiali di decrittografia.

Input (dall'applicazione)

- Chiave wrapping: facoltativa e ignorata.
- Chiave di annullamento del wrapping: la stessa chiave AES simmetrica o la chiave privata RSA corrispondente alla chiave pubblica RSA utilizzata per la crittografia. Necessaria se alcuni valori degli attributi sono crittografati. In caso contrario, è facoltativa e viene ignorata.
- Chiave di firma

Input (dal componente di crittografia dell'item)

- Una copia del contesto di crittografia DynamoDB che contiene il contenuto dell'attributo di descrizione del materiale.

Output (sul componente di crittografia dell'item)

- Chiave di crittografia dell'item di testo normale
- Chiave di firma (invariata)

Processing

La decrittografia di un item richiede una chiave di annullamento del wrapping e una chiave di firma. La chiave di wrapping è facoltativa e viene ignorata.

1. Il CMP di sottoposti a wrapping ottiene la chiave di crittografia item sottoposta a wrapping dall'attributo di descrizione del materiale dell'item.
2. Utilizza la chiave e l'algoritmo di annullamento del wrapping per annullare il wrapping della chiave di crittografia item.
3. Restituisce la chiave di crittografia item di testo normale, la chiave di firma e gli algoritmi di crittografia e firma al componente di crittografia dell'item.
4. Il componente di crittografia dell'item utilizza la chiave di firma per verificare l'item. Se la verifica riesce, utilizza la chiave di crittografia item per decrittografare l'item. Quindi, rimuove le chiavi di testo normale dalla memoria il prima possibile.

## Provider più recente

> **Note**
>
> La nostra libreria di crittografia lato client è stata rinominata Database Encryption SDK. AWS Il seguente argomento fornisce informazioni sulle versioni 1. x —2. x del DynamoDB Encryption Client for Java e versioni 1. x —3. x del client di crittografia DynamoDB per Python. Per ulteriori informazioni, consulta AWS Database Encryption SDK per il supporto della versione DynamoDB.

Il provider più recente è un provider di materiali crittografici (CMP) progettato per funzionare con gli archivi del provider. Viene CMPs dall'archivio del provider e ottiene i materiali crittografici restituiti da. CMPs In genere utilizza ciascun CMP per soddisfare più richieste di materiali crittografici. Ma puoi utilizzare le funzioni del suo archivio provider per controllare in quale misura i materiali vengono riutilizzati, stabilire la frequenza di rotazione del CMP e persino cambiare il tipo di CMP utilizzato senza cambiare il provider più recente.

> **Note**
>
> Il codice associato al `MostRecentProvider` simbolo del provider più recente potrebbe archiviare materiali crittografici in memoria per tutta la durata del processo. Potrebbe consentire a un chiamante di utilizzare chiavi che non è più autorizzato a utilizzare. Il `MostRecentProvider` simbolo è obsoleto nelle versioni precedenti supportate del DynamoDB Encryption Client e rimosso dalla versione 2.0.0. Viene sostituito dal simbolo. `CachingMostRecentProvider` Per informazioni dettagliate, consultare Aggiornamenti al provider più recente.

Il provider più recente è una buona scelta per le applicazioni che devono ridurre al minimo le chiamate all'archivio provider e all'origine crittografica e per le applicazioni che possono riutilizzare alcuni materiali crittografici senza violare i requisiti di sicurezza. Ad esempio, ti consente di proteggere i tuoi materiali crittografici con un AWS KMS keyin AWS Key Management Service(AWS KMS) senza chiamare AWS KMS ogni volta che crittografi o decrittografi un elemento.

Il provider store scelto determina il tipo di provider utilizzato dal provider più recente e la frequenza con CMPs cui riceve una nuova CMP. Puoi utilizzare qualsiasi archivio provider compatibile con il provider più recente, inclusi quelli personalizzati che hai progettato.

Il client di crittografia DynamoDB include MetaStoreun client che crea e restituisce Wrapped Materials Providers (Wrapped). CMPs MetaStore Salva più versioni di Wrapped CMPs che genera in una tabella DynamoDB interna e le protegge con la crittografia lato client tramite un'istanza interna del DynamoDB Encryption Client.

Puoi configurarlo MetaStore per utilizzare qualsiasi tipo di CMP interno per proteggere i materiali nella tabella, incluso un Direct KMS Provider che genera materiali crittografici protetti dall'utente AWS KMS key, un CMP Wrapped che utilizza le chiavi di wrapping e firma fornite dall'utente o un CMP personalizzato compatibile progettato da te.

Per il codice di esempio, consulta:

- Java: MostRecentEncryptedItem
- Python: most_recent_provider_encrypted_table

Argomenti

- Come utilizzarlo
- Come funziona
- Aggiornamenti al provider più recente

Come utilizzarlo

Per creare un provider più recente devi creare e configurare un archivio provider e quindi creare un provider più recente che lo utilizzi.

Gli esempi seguenti mostrano come creare un provider più recente che utilizza a MetaStore e protegge le versioni nella sua tabella DynamoDB interna con materiali crittografici provenienti da un provider Direct KMS. Questi esempi utilizzano il simbolo. `CachingMostRecentProvider`

Ogni provider più recente ha un nome che lo identifica CMPs nella MetaStore tabella, un'impostazione time-to-live(TTL) e un'impostazione della dimensione della cache che determina il numero di voci che la cache può contenere. Questi esempi impostano la dimensione della cache su 1000 voci e un TTL di 60 secondi.

Java

```
// Set the name for MetaStore's internal table
final String keyTableName = 'metaStoreTable'
```

```
// Set the Region and AWS KMS key
final String region = 'us-west-2'
final String keyArn = 'arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab'

// Set the TTL and cache size
final long ttlInMillis = 60000;
final long cacheSize = 1000;

// Name that identifies the MetaStore's CMPs in the provider store
final String materialName = 'testMRP'

// Create an internal DynamoDB client for the MetaStore
final AmazonDynamoDB ddb =
 AmazonDynamoDBClientBuilder.standard().withRegion(region).build();

// Create an internal Direct KMS Provider for the MetaStore
final AWSKMS kms = AWSKMSClientBuilder.standard().withRegion(region).build();
final DirectKmsMaterialProvider kmsProv = new DirectKmsMaterialProvider(kms,
 keyArn);

// Create an item encryptor for the MetaStore,
// including the Direct KMS Provider
final DynamoDBEncryptor keyEncryptor = DynamoDBEncryptor.getInstance(kmsProv);

// Create the MetaStore
final MetaStore metaStore = new MetaStore(ddb, keyTableName, keyEncryptor);

//Create the Most Recent Provider
final CachingMostRecentProvider cmp = new CachingMostRecentProvider(metaStore,
 materialName, ttlInMillis, cacheSize);
```

### Python

```
# Designate an AWS KMS key
kms_key_id = 'arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab'

# Set the name for MetaStore's internal table
meta_table_name = 'metaStoreTable'

# Name that identifies the MetaStore's CMPs in the provider store
```

```
material_name = 'testMRP'

# Create an internal DynamoDB table resource for the MetaStore
meta_table = boto3.resource('dynamodb').Table(meta_table_name)

# Create an internal Direct KMS Provider for the MetaStore
kms_cmp = AwsKmsCryptographicMaterialsProvider(key_id=kms_key_id)

# Create the MetaStore with the Direct KMS Provider
meta_store = MetaStore(
    table=meta_table,
    materials_provider=kms_cmp
)

# Create a Most Recent Provider using the MetaStore
#    Sets the TTL (in seconds) and cache size (# entries)
most_recent_cmp = MostRecentProvider(
    provider_store=meta_store,
    material_name=material_name,
    version_ttl=60.0,
    cache_size=1000
)
```

Come funziona

Il provider più recente riceve CMPs da un provider store. Quindi utilizza il CMP per generare i materiali crittografici che restituisce al componente di crittografia dell'item.

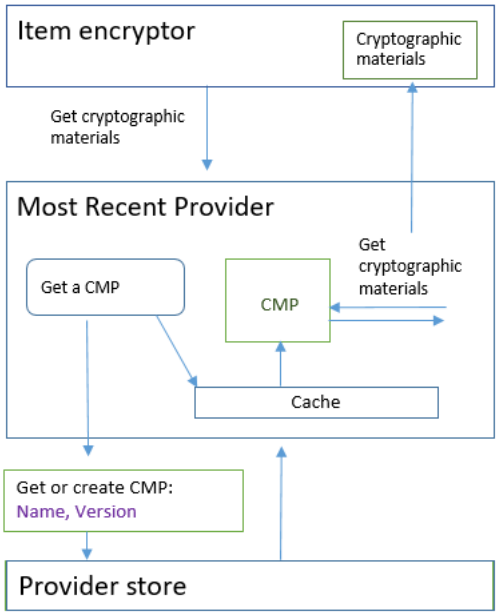
Informazioni sul provider più recente

Il provider più recente ottiene un provider di materiali crittografici (CMP) da un archivio provider. Utilizza quindi il CMP per generare i materiali crittografici che restituisce. Ogni provider più recente è associato a un provider store, ma un provider store può CMPs rifornire più provider su più host.

Il provider più recente può utilizzare qualsiasi CMP compatibile di qualsiasi archivio provider. Richiede materiali di crittografia o decrittografia dal CMP e restituisce l'output all'item encryptor. Non esegue alcuna operazione crittografica.

Per richiedere un CMP al suo archivio provider, il provider più recente fornisce il nome del materiale e la versione di un CMP esistente che desidera utilizzare. Per i materiali di crittografia, il provider più recente richiede sempre la versione massima ("più recente"). Per i materiali di decrittografia, richiede

la versione del CMP che è stata utilizzata per creare i materiali di crittografia, come mostrato nel diagramma seguente.




Il provider più recente salva le versioni restituite dall'archivio del CMPs provider in una cache locale LRU (Least Recently Used) in memoria. La cache consente al provider più recente di ottenere ciò di CMPs cui ha bisogno senza chiamare l'archivio del provider per ogni articolo. Puoi cancellare la cache on-demand.

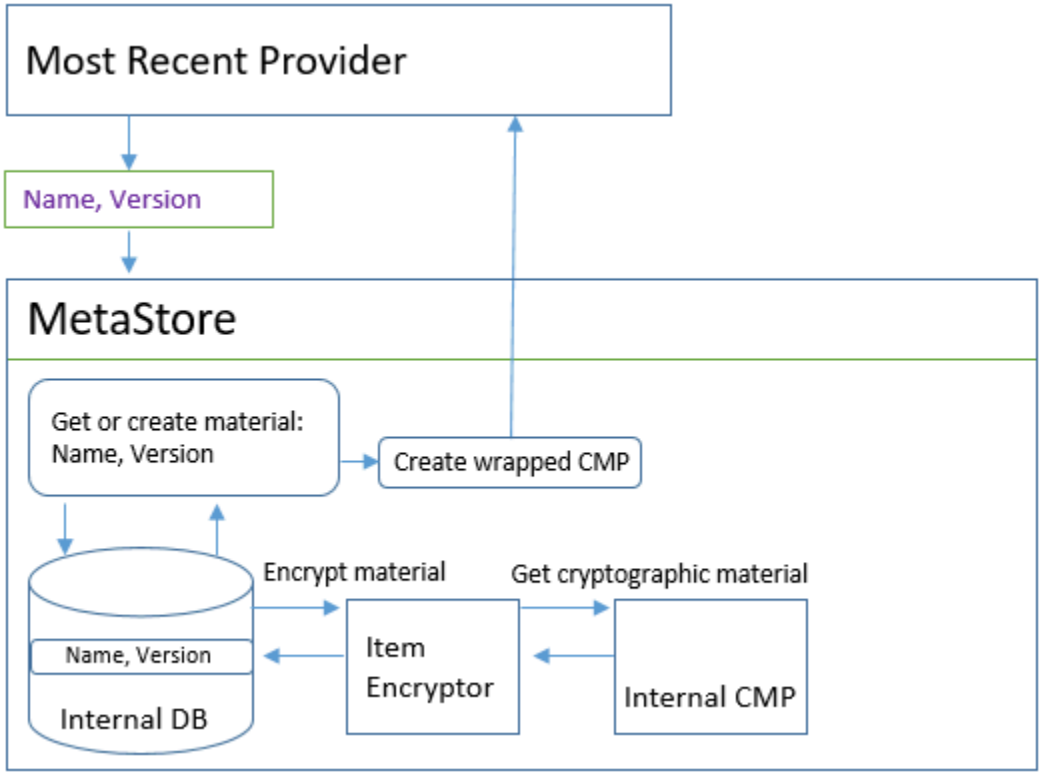
Il provider più recente utilizza un time-to-livevalore configurabile che è possibile regolare in base alle caratteristiche dell'applicazione.

Informazioni su MetaStore

Puoi utilizzare un provider più recente con qualsiasi archivio provider, anche un archivio provider personalizzato compatibile. Il DynamoDB Encryption Client include MetaStore un'implementazione sicura che è possibile configurare e personalizzare.

A MetaStoreè un provider store che crea e restituisce Wrapped configurati con la chiave di CMPs wrapping, la chiave di unwrapping e la chiave di firma richieste da Wrapped. CMPs A MetaStore è un'opzione sicura per un provider più recente perché Wrapped genera CMPs sempre chiavi di crittografia degli elementi uniche per ogni articolo. Vengono riutilizzate solo la chiave di wrapping che protegge la chiave di crittografia degli item e le chiavi di firma.

Il diagramma seguente mostra i componenti di MetaStore e come interagisce con il provider più recente.




MetaStore genera i Wrapped CMPs e poi li archivia (in forma crittografata) in una tabella DynamoDB interna. La chiave di partizione è il nome del materiale del provider più recente; la chiave di ordinamento è il numero di versione. I materiali nella tabella sono protetti da un client di crittografia DynamoDB interno, che include un item encryptor e un provider interno di materiali crittografici (CMP).

Puoi utilizzare qualsiasi tipo di CMP interno al tuo sito MetaStore, incluso un Direct KMS Provider, un CMP Wrapped con materiali crittografici da te fornito o un CMP personalizzato compatibile. Se il tuo CMP interno MetaStore è un Direct KMS Provider, le tue chiavi di wrapping e firma riutilizzabili sono protette da un in (). AWS KMS keyAWS Key Management ServiceAWS KMS Le MetaStore chiamate AWS KMS ogni volta che aggiunge una nuova versione CMP alla sua tabella interna o ottiene una versione CMP dalla sua tabella interna.

Impostazione di un valore time-to-live

È possibile impostare un valore time-to-live (TTL) per ogni provider più recente creato. In generale, utilizzate il valore TTL più basso che sia pratico per la vostra applicazione.

L'uso del valore TTL viene modificato nel `CachingMostRecentProvider` simbolo del provider più recente.

> **Note**
>
> Il `MostRecentProvider` simbolo del provider più recente è obsoleto nelle versioni precedenti supportate del DynamoDB Encryption Client e rimosso dalla versione 2.0.0. Viene sostituito dal simbolo. `CachingMostRecentProvider` Ti consigliamo di aggiornare il codice il prima possibile. Per informazioni dettagliate, consultare Aggiornamenti al provider più recente.

**`CachingMostRecentProvider`**

`CachingMostRecentProvider`Utilizza il valore TTL in due modi diversi.

- Il TTL determina la frequenza con cui il provider più recente verifica la presenza di una nuova versione del CMP nell'archivio del provider. Se è disponibile una nuova versione, il provider più recente sostituisce la propria CMP e aggiorna i materiali crittografici. Altrimenti, continua a utilizzare i suoi attuali materiali CMP e crittografici.
- Il TTL determina per quanto tempo CMPs è possibile utilizzare la cache. Prima di utilizzare una CMP memorizzata nella cache per la crittografia, il Most Recent Provider valuta il tempo trascorso nella cache. Se il tempo della cache CMP supera il TTL, la CMP viene rimossa dalla cache e il provider più recente riceve una nuova versione CMP dall'archivio del provider.

**`MostRecentProvider`**

Nel`MostRecentProvider`, il TTL determina la frequenza con cui il provider più recente verifica la presenza di una nuova versione del CMP nell'archivio del provider. Se è disponibile una nuova versione, il provider più recente sostituisce la propria CMP e aggiorna i materiali crittografici. Altrimenti, continua a utilizzare i suoi attuali materiali CMP e crittografici.

Il TTL non determina la frequenza con cui viene creata una nuova versione CMP. È possibile creare nuove versioni CMP ruotando i materiali crittografici.

Un valore TTL ideale varia a seconda dell'applicazione e dei suoi obiettivi di latenza e disponibilità. Un TTL inferiore migliora il profilo di sicurezza riducendo il tempo di archiviazione dei materiali crittografici in memoria. Inoltre, un TTL inferiore aggiorna le informazioni critiche con maggiore frequenza. Ad esempio, se il CMP interno è un Direct KMS Provider, verifica più frequentemente che il chiamante sia ancora autorizzato a utilizzare un. AWS KMS key

Tuttavia, se il TTL è troppo breve, le chiamate frequenti all'archivio del provider possono aumentare i costi e far sì che l'archivio del provider limiti le richieste provenienti dall'applicazione e da altre applicazioni che condividono l'account di servizio. Potresti anche trarre vantaggio dal coordinamento del TTL con la velocità di rotazione dei materiali crittografici.

Durante i test, variate il TTL e le dimensioni della cache in base ai diversi carichi di lavoro fino a trovare una configurazione adatta alla vostra applicazione e ai vostri standard di sicurezza e prestazioni.

Rotazione dei materiali crittografici

Quando un Most Recent Provider necessita di materiali di crittografia, utilizza sempre la versione più recente della sua CMP di cui è a conoscenza. La frequenza con cui verifica la presenza di una versione più recente è determinata dal valore time-to-live(TTL) impostato quando si configura il provider più recente.

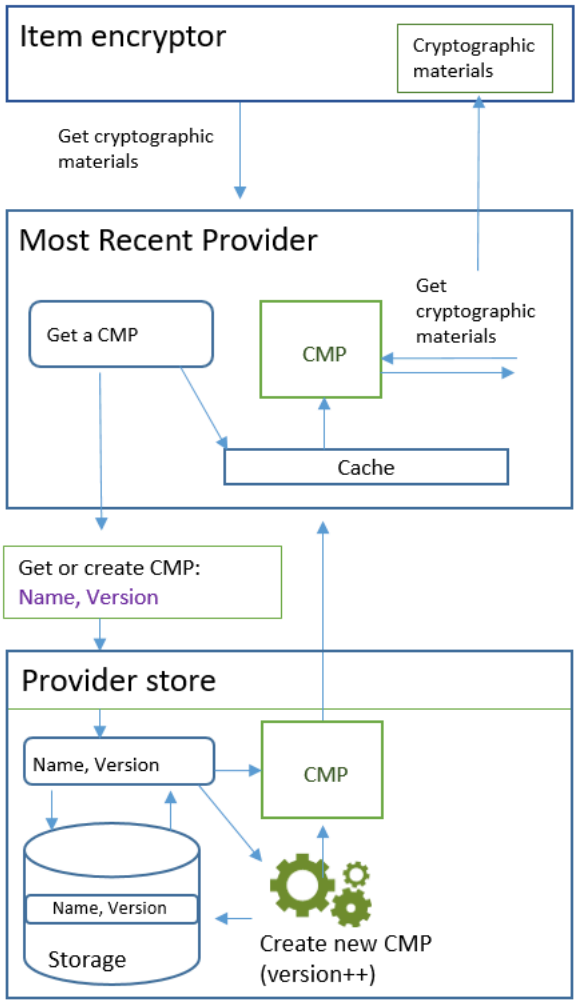
Quando il TTL scade, il provider più recente verifica la presenza di una versione più recente del CMP nell'archivio del provider. Se disponibile, il provider più recente la ottiene e sostituisce la CMP nella sua cache. Utilizza questo CMP e il relativo materiale crittografico finché non scopre che Provider Store ha una versione più recente.

Per indicare all'archivio provider di creare una nuova versione di un CMP per un provider più recente, richiama l'operazione Create New Provider (Crea nuovo provider) dell'archivio provider con il nome del materiale del provider più recente. L'archivio provider crea un nuovo CMP e salva una copia crittografata nel suo storage interno con un numero di versione superiore: Restituisce anche un CMP, ma puoi ignorarlo. Di conseguenza, la volta successiva che il provider più recente richiede al provider store il numero massimo di versione CMPs, ottiene il nuovo numero di versione più recente e lo utilizza nelle richieste successive allo store per verificare se è stata creata una nuova versione del CMP.

Puoi programmare le chiamate Create New Provider (Crea nuovo provider) sulla base del tempo, del numero di item o di attributi elaborati o su qualsiasi altro parametro rilevante per la tua applicazione.

Ottenere materiali di crittografia

Il provider più recente utilizza il seguente processo, mostrato in questo diagramma, per ottenere i materiali di crittografia che restituisce al componente di crittografia dell'item. L'output dipende dal tipo di CMP restituito dall'archivio provider. Il provider più recente può utilizzare qualsiasi archivio provider compatibile, incluso MetaStore quello incluso nel DynamoDB Encryption Client.




Quando si crea un provider più recente utilizzando il `CachingMostRecentProvider`simbolo, si specifica un archivio provider, un nome per il provider più recente e un valore time-to-live(TTL). È inoltre possibile specificare facoltativamente una dimensione della cache, che determina il numero massimo di materiali crittografici che possono esistere nella cache.

Quando il componente di crittografia dell'item chiede al provider più recente i materiali di crittografia, il provider più recente inizia a cercare nella sua cache la versione più recente del suo CMP.

- Se trova la versione CMP più recente nella cache e la CMP non ha superato il valore TTL, il provider più recente utilizza la CMP per generare materiali di crittografia. Quindi restituisce i materiali di crittografia al componente di crittografia dell'item. Questa operazione non richiede una chiamata dell'archivio provider.
- Se la versione più recente della CMP non è presente nella cache o se è presente nella cache ma ha superato il valore TTL, il provider più recente richiede una CMP dall'archivio del provider. La

richiesta include il nome del materiale del provider più recente e il numero della versione massima che conosce.

1. L'archivio provider restituisce un CMP dal suo storage persistente. Se il provider store è un MetaStore, ottiene un Wrapped CMP crittografato dalla tabella DynamoDB interna utilizzando il nome del materiale del provider più recente come chiave di partizione e il numero di versione come chiave di ordinamento. MetaStore Utilizza il criptatore interno degli elementi e la CMP interna per decrittografare la Wrapped CMP. Quindi restituisce il CMP come testo normale al provider più recente. Se il CMP interno è un provider KMS diretto, questa fase prevede una chiamata a AWS Key Management Service (AWS KMS).
2. Il CMP aggiunge il campo `amzn-ddb-meta-id` alla descrizione dei materiali effettivi. Il suo valore è il nome del materiale e la versione del CMP nella sua tabella interna. L'archivio provider restituisce al provider più recente il CMP come testo normale.
3. Il provider più recente archivia il CMP nella cache.
4. Il provider più recente utilizza il CMP per generare i materiali di crittografia. Quindi restituisce i materiali di crittografia al componente di crittografia dell'item.

Ottenere materiali di decrittografia

Quando il componente di crittografia dell'item chiede al provider più recente i materiali di decrittografia, il provider più recente utilizza il seguente processo per ottenerli e restituirli.

1. Il provider più recente chiede all'archivio provider il numero di versione dei materiali crittografici utilizzati per crittografare l'item. Passa la descrizione dei materiali effettivi dall'attributo di descrizione del materiale dell'item.
2. L'archivio provider riceve il numero di versione del CMP di crittografia dal campo `amzn-ddb-meta-id` nella descrizione dei materiali effettivi e lo restituisce al provider più recente.
3. Il provider più recente ricerca nella sua cache la versione del CMP utilizzata per crittografare e firmare l'item.

- Se rileva che la versione corrispondente della CMP è nella sua cache e la CMP non ha superato il valore time-to-live (TTL), il provider più recente utilizza la CMP per generare materiali di decrittografia. Quindi restituisce i materiali di decrittografia al componente di crittografia dell'item. Questa operazione non richiede una chiamata dell'archivio provider o a qualsiasi altro CMP.

- Se la versione corrispondente della CMP non è presente nella cache o se la cache AWS KMS key ha superato il valore TTL, il provider più recente richiede una CMP dal proprio provider store. Nella richiesta invia il nome del materiale e il numero di versione del suo CMP di crittografia.

  1. L'archivio provider ricerca nello storage persistente il CMP utilizzando il nome del provider più recente come chiave di partizione e il numero di versione come chiave di ordinamento.

     - Se il nome e il numero di versione non sono nello storage persistente, l'archivio provider rileva un'eccezione. Se l'archivio provider è stato utilizzato per generare il CMP, il CMP dovrebbe essere archiviato nel suo storage persistente, a meno che non sia stato volutamente eliminato.
     - Se il CMP con il numero di versione e il nome corrispondenti sono disponibili nello storage persistente dell'archivio provider, quest'ultimo restituisce il CMP specificato al provider più recente.

       Se il provider store è un MetaStore, ottiene il CMP crittografato dalla tabella DynamoDB. Quindi utilizza i materiali crittografici dal suo CMP interno per decrittografare il CMP crittografato prima di restituire il CMP al provider più recente. Se il CMP interno è un provider KMS diretto, questa fase prevede una chiamata a AWS Key Management Service (AWS KMS).

  2. Il provider più recente archivia il CMP nella cache.
  3. Il provider più recente utilizza il CMP per generare i materiali di decrittografia. Quindi restituisce i materiali di decrittografia al componente di crittografia dell'item.

Aggiornamenti al provider più recente

Il simbolo del provider più recente viene modificato da `MostRecentProvider` a `CachingMostRecentProvider`.

> **Note**
>
> Il `MostRecentProvider` simbolo, che rappresenta il provider più recente, è obsoleto nella versione 1.15 del DynamoDB Encryption Client for Java e nella versione 1.3 del DynamoDB Encryption Client for Python e rimosso dalle versioni 2.0.0 del DynamoDB Encryption Client in entrambe le implementazioni linguistiche. Utilizzate invece il. `CachingMostRecentProvider`

`CachingMostRecentProvider`Implementa le seguenti modifiche:

- Rimuove `CachingMostRecentProvider` periodicamente i materiali crittografici dalla memoria quando la loro permanenza in memoria supera il valore configurato time-to-live (TTL).

  `MostRecentProvider`Potrebbero archiviare materiali crittografici in memoria per tutta la durata del processo. Di conseguenza, il provider più recente potrebbe non essere a conoscenza delle modifiche alle autorizzazioni. Potrebbe utilizzare le chiavi di crittografia dopo la revoca delle autorizzazioni del chiamante per utilizzarle.

  Se non riesci ad eseguire l'aggiornamento a questa nuova versione, puoi ottenere un effetto simile chiamando periodicamente il `clear()` metodo nella cache. Questo metodo svuota manualmente il contenuto della cache e richiede al Most Recent Provider di richiedere una nuova CMP e nuovi materiali crittografici.
- Include `CachingMostRecentProvider` anche un'impostazione della dimensione della cache che consente un maggiore controllo sulla cache.

Per eseguire l'aggiornamento a`CachingMostRecentProvider`, è necessario modificare il nome del simbolo nel codice. Sotto tutti gli altri aspetti, `CachingMostRecentProvider` è completamente retrocompatibile con. `MostRecentProvider` Non è necessario crittografare nuovamente gli elementi della tabella.

Tuttavia, `CachingMostRecentProvider` genera più chiamate all'infrastruttura chiave sottostante. Chiama l'archivio del provider almeno una volta in ogni intervallo time-to-live (TTL). Le applicazioni con numerose applicazioni attive CMPs (a causa della frequente rotazione) o le applicazioni con flotte di grandi dimensioni sono più suscettibili a questo cambiamento.

Prima di rilasciare il codice aggiornato, testalo accuratamente per assicurarti che le chiamate più frequenti non danneggino l'applicazione o causino limitazioni da parte dei servizi da cui dipende il tuo provider, come AWS Key Management Service () o AWS KMS Amazon DynamoDB. Per mitigare eventuali problemi di prestazioni, regola la dimensione e la dimensione della cache in `CachingMostRecentProvider` base alle time-to-live caratteristiche prestazionali osservate. Per le linee guida, consulta Impostazione di un valore time-to-live.

## Provider di materiali statici

> **Note**
>
> La nostra libreria di crittografia lato client è stata rinominata Database Encryption SDK. AWS Il seguente argomento fornisce informazioni sulle versioni 1. x —2. x del DynamoDB Encryption Client for Java e versioni 1. x —3. x del client di crittografia DynamoDB per Python. Per ulteriori informazioni, consulta AWS Database Encryption SDK per il supporto della versione DynamoDB.

Lo Static Materials Provider (Static CMP) è un fornitore di materiali crittografici (CMP) molto semplice destinato a test, proof-of-concept dimostrazioni e compatibilità con le versioni precedenti.

Per utilizzare il CMP di statici per crittografare un item della tabella, è necessario fornire una chiave di crittografia simmetrica AES e una chiave di firma o una coppia di chiavi. Devi fornire le stesse chiavi per decrittografare l'item crittografato. Il CMP di statici non esegue alcuna operazione di crittografia. Passa invece al componente di crittografia dell'item le chiavi di crittografia da te fornite senza modificarle. Il componente di crittografia dell'item crittografa direttamente gli item con la chiave di crittografia. Quindi, utilizza direttamente la chiave di firma per firmarli.

Poiché il CMP di statici non genera alcun materiale crittografico univoco, tutti gli item della tabella che hai elaborato sono crittografati con la stessa chiave di crittografia e firmati con la stessa chiave di firma. Quando utilizzi la stessa chiave per crittografare i valori degli attributi in numerosi item o utilizzi la stessa chiave o coppia di chiavi per firmare tutti gli item, rischi di superare i limiti crittografici delle chiavi.

> **Note**
>
> Il Provider di statici asimmetrico nella libreria Java non è un provider di statici. Fornisce soltanto costruttori alternativi per il CMP di sottoposti a wrapping. Può essere utilizzato per la produzione senza alcun rischio per la sicurezza, tuttavia dovresti utilizzare direttamente il CMP di sottoposti a wrapping ogni qualvolta sia possibile.

Static CMP è uno dei numerosi fornitori di materiali crittografici (CMPs) supportati dal DynamoDB Encryption Client. Per informazioni sull'altro, vedere. CMPs Fornitore di materiali crittografici

Per il codice di esempio, consulta:

- Java: SymmetricEncryptedItem

Argomenti

- Come utilizzarlo
- Come funziona

### Come utilizzarlo

Per creare un provider di statici, fornisci una chiave di crittografia o una coppia di chiavi e una chiave di firma o una coppia di chiavi. Devi fornire materiali chiave per crittografare e decrittografare gli item della tabella.

Java

```
// To encrypt
SecretKey cek = ...;        // Encryption key
SecretKey macKey =  ...;    // Signing key
EncryptionMaterialsProvider provider = new SymmetricStaticProvider(cek, macKey);

// To decrypt
SecretKey cek = ...;        // Encryption key
SecretKey macKey =  ...;    // Verification key
EncryptionMaterialsProvider provider = new SymmetricStaticProvider(cek, macKey);
```

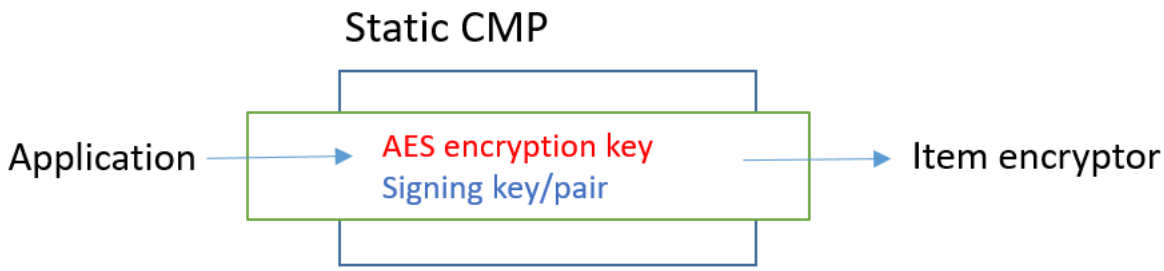
Python

```
# You can provide encryption materials, decryption materials, or both
encrypt_keys = EncryptionMaterials(
    encryption_key = ...,
    signing_key = ...
)

decrypt_keys = DecryptionMaterials(
    decryption_key = ...,
    verification_key = ...
)

static_cmp = StaticCryptographicMaterialsProvider(
    encryption_materials=encrypt_keys
    decryption_materials=decrypt_keys
```

```
)
```

Come funziona

Il provider di statici passa le chiavi di crittografia e firma che hai fornito al componente di crittografia dell'item, dove vengono direttamente utilizzate per crittografare e firmare gli item della tabella. Vengono utilizzate le stesse chiavi per tutti gli item, a meno che tu non fornisca chiavi diverse per ciascun item.




Ottenere materiali di crittografia

In questa sezione vengono descritti nei dettagli gli input, gli output e l'elaborazione eseguita dal provider di materiali statici (CMP di statici) al momento della ricezione di una richiesta di materiali di crittografia.

Input (dall'applicazione)

- Una chiave di crittografia: deve essere una chiave simmetrica, ad esempio una chiave Advanced Encryption Standard (AES).
- Una chiave di firma: può essere una chiave simmetrica o una coppia di chiavi asimmetrica.

Input (dal componente di crittografia dell'item)

- Contesto di crittografia DynamoDB

Output (sul componente di crittografia dell'item)

- La chiave di crittografia passata come input.
- La chiave di firma passata come input.

- Descrizione dei materiali effettivi: l'eventuale descrizione dei materiali richiesti invariata.

Ottenere materiali di decrittografia

In questa sezione vengono descritti nei dettagli gli input, gli output e l'elaborazione eseguita dal provider di materiali statici (CMP di statici) al momento della ricezione di una richiesta di materiali di decrittografia.

Sebbene i metodi per ottenere i materiali di crittografia e ottenere i materiali di decrittografia siano separati, il comportamento è lo stesso.

Input (dall'applicazione)

- Una chiave di crittografia: deve essere una chiave simmetrica, ad esempio una chiave Advanced Encryption Standard (AES).
- Una chiave di firma: può essere una chiave simmetrica o una coppia di chiavi asimmetrica.

Input (dal componente di crittografia dell'item)

- Contesto di crittografia DynamoDB (non utilizzato)

Output (sul componente di crittografia dell'item)

- La chiave di crittografia passata come input.
- La chiave di firma passata come input.

## Linguaggi di programmazione disponibili per Amazon DynamoDB Encryption Client

> **Note**
>
> La nostra libreria di crittografia lato client è stata rinominata Database Encryption SDK. AWS Il seguente argomento fornisce informazioni sulle versioni 1. x —2. x del DynamoDB Encryption Client for Java e versioni 1. x —3. x del client di crittografia DynamoDB per Python. Per ulteriori informazioni, consulta AWS Database Encryption SDK per il supporto della versione DynamoDB.

Il client di crittografia Amazon DynamoDB è disponibile per i seguenti linguaggi di programmazione. Anche se ogni linguaggio ha delle librerie specifiche, le implementazioni risultanti sono interoperabili. Ad esempio, puoi crittografare (e firmare) un item con il client Java e decrittografarlo con il client Python.

Per maggiori informazioni, consulta l'argomento corrispondente.

Argomenti

- Client di crittografia Amazon DynamoDB per Java
- Client di crittografia DynamoDB per Python

## Client di crittografia Amazon DynamoDB per Java

> **Note**
>
> La nostra libreria di crittografia lato client è stata rinominata Database Encryption SDK. AWS Il seguente argomento fornisce informazioni sulle versioni 1. x —2. x del DynamoDB Encryption Client for Java e versioni 1. x —3. x del client di crittografia DynamoDB per Python. Per ulteriori informazioni, consulta AWS Database Encryption SDK per il supporto della versione DynamoDB.

Questo argomento spiega come installare e utilizzare il client di crittografia Amazon DynamoDB per Java. Per i dettagli sulla programmazione con il DynamoDB Encryption Client, consulta gli esempi Java, gli esempi nel repository GitHub su e Javadoc per aws-dynamodb-encryption-java il DynamoDB Encryption Client.

> **Note**
>
> Versioni 1. x. x del DynamoDB Encryption Client for Java entreranno in vigore gradualmente end-of-support a luglio 2022. Effettua l'upgrade a una versione più recente il prima possibile.

Argomenti

- Prerequisiti
- Installazione

- Utilizzo del client di crittografia DynamoDB per Java
- Codice di esempio per il client di crittografia DynamoDB per Java

### Prerequisiti

Prima di installare Amazon DynamoDB Encryption Client for Java, assicurati di avere i seguenti prerequisiti.

#### Un ambiente di sviluppo Java

È necessario Java 8 o versioni successive. Nel sito Web di Oracle, accedi alla pagina Java SE Download, quindi scarica e installa Java SE Development Kit (JDK).

Se utilizzi Oracle JDK, devi scaricare e installare anche Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files.

#### AWS SDK per Java

Il DynamoDB Encryption Client richiede il modulo DynamoDB di DynamoDB anche se AWS SDK per Java l'applicazione non interagisce con DynamoDB. Puoi installare l'intero SDK o solo questo modulo. Se utilizzi Maven, aggiungi `aws-java-sdk-dynamodb` al file `pom.xml`.

Per ulteriori informazioni sull'installazione e la configurazione di, vedere. AWS SDK per JavaAWS SDK per Java

### Installazione

Puoi installare Amazon DynamoDB Encryption Client for Java nei seguenti modi.

#### Manualmente

Per installare Amazon DynamoDB Encryption Client for Java, clona o scarica il repository. aws-dynamodb-encryption-java GitHub

#### Utilizzo di Apache Maven

Il client di crittografia Amazon DynamoDB per Java è disponibile tramite Apache Maven con la seguente definizione di dipendenza.

```
<dependency>
```

```
  <groupId>com.amazonaws</groupId>
  <artifactId>aws-dynamodb-encryption-java</artifactId>
  <version>version-number</version>
</dependency>
```

Dopo aver installato l'SDK, inizia esaminando il codice di esempio in questa guida e il client di crittografia DynamoDB Javadoc attivo. GitHub

Utilizzo del client di crittografia DynamoDB per Java

> **Note**
>
> La nostra libreria di crittografia lato client è stata rinominata Database Encryption SDK. AWS Il seguente argomento fornisce informazioni sulle versioni 1. x —2. x del DynamoDB Encryption Client for Java e versioni 1. x —3. x del client di crittografia DynamoDB per Python. Per ulteriori informazioni, consulta AWS Database Encryption SDK per il supporto della versione DynamoDB.

Questo argomento spiega alcune funzionalità del DynamoDB Encryption Client in Java che potrebbero non essere presenti in altre implementazioni del linguaggio di programmazione.

Per i dettagli sulla programmazione con il DynamoDB Encryption Client, consulta gli esempi Java, gli esempi in the GitHub on e Javadoc per `aws-dynamodb-encryption-java repository` il DynamoDB Encryption Client.

Argomenti

- Item encryptors: e Dynamo AttributeEncryptor DBEncryptor
- Configurazione del comportamento di salvataggio
- Operazioni di attributo in Java
- Sovrascrivere i nomi delle tabelle

Item encryptors: e Dynamo AttributeEncryptor DBEncryptor

Il DynamoDB Encryption Client in Java dispone di due crittografi di elementi: Dynamo di livello inferiore e. DBEncryptor AttributeEncryptor

`AttributeEncryptor`È una classe di supporto che consente di utilizzare Dynamo AWS SDK per Java con il `DynamoDB Encryptor` client DBMapper di crittografia DynamoDB. Quando utilizzi `AttributeEncryptor` con `DynamoDBMapper`, crittografa e firma in modo trasparente gli item quando li salvi. Inoltre, verifica e decrittografa in modo trasparente gli item quando li carichi.

Configurazione del comportamento di salvataggio

È possibile utilizzare `AttributeEncryptor` and `DynamoDBMapper` per aggiungere o sostituire gli elementi della tabella con attributi solo firmati o crittografati e firmati. Per queste attività, ti consigliamo di configurare il servizio per utilizzare il comportamento di salvataggio PUT, come illustrato nell'esempio seguente. In caso contrario, potresti non riuscire a decrittografare i dati.

```
DynamoDBMapperConfig mapperConfig =
 DynamoDBMapperConfig.builder().withSaveBehavior(SaveBehavior.PUT).build();
DynamoDBMapper mapper = new DynamoDBMapper(ddb, mapperConfig, new
 AttributeEncryptor(encryptor));
```

Se si utilizza il comportamento di salvataggio predefinito, che aggiorna solo gli attributi modellati nell'elemento della tabella, gli attributi non modellati non vengono inclusi nella firma e non vengono modificati dalle scritture della tabella. Di conseguenza, nelle letture successive di tutti gli attributi, la firma non verrà convalidata, poiché non include attributi non modellati.

Puoi inoltre utilizzare il comportamento di salvataggio `CLOBBER`. Questo comportamento di salvataggio è identico al comportamento di salvataggio PUT, ma disabilita il blocco ottimistico e sovrascrive l'item nella tabella.

Per evitare errori di firma, il client di crittografia DynamoDB genera un'eccezione di runtime se una viene utilizzata con `AttributeEncryptor` un file che non è configurato con `DynamoDBMapper` un comportamento di salvataggio di o. `CLOBBER` PUT

Per vedere questo codice usato in un esempio, vedi Usare Dynamo DBMapper e l'AwsKmsEncryptedObjectesempio.java nel repository in. `aws-dynamodb-encryption-java` GitHub

Operazioni di attributo in Java

Le operazioni di attributo determinano quali valori attributo sono crittografati e firmati, quali solo firmati e quali ignorati. Il metodo utilizzato per specificare le azioni degli attributi dipende dal fatto che si utilizzi il comando `DynamoDBMapper` and o il `AttributeEncryptor` Dynamo di livello inferiore. DBEncryptor

> **Important**
>
> Dopo aver utilizzato le azioni degli attributi per crittografare gli elementi della tabella, l'aggiunta o la rimozione di attributi dal modello di dati potrebbe causare un errore di convalida della firma che impedisce di decrittografare i dati. Per una spiegazione dettagliata, consulta Modifica del modello di dati.

Azioni relative agli attributi per la Dynamo DBMapper

Quando utilizzi `DynamoDBMapper` e `AttributeEncryptor`, devi utilizzare le annotazioni per specificare le operazioni di attributo. Il DynamoDB Encryption Client utilizza le annotazioni standard degli attributi DynamoDB che definiscono il tipo di attributo per determinare come proteggere un attributo. Per impostazione predefinita, tutti gli attributi sono crittografati e firmati, tranne le chiavi primarie, che sono firmate ma non crittografate.

> **Note**
>
> Non crittografate il valore degli attributi con l'annotazione @Dynamo DBVersion Attribute, anche se potete (e dovreste) firmarli. In caso contrario, le condizioni che utilizzano questo valore potrebbero avere effetti imprevisti.

```
// Attributes are encrypted and signed
@DynamoDBAttribute(attributeName="Description")

// Partition keys are signed but not encrypted
@DynamoDBHashKey(attributeName="Title")

// Sort keys are signed but not encrypted
@DynamoDBRangeKey(attributeName="Author")
```

Per specificare le eccezioni, utilizzate le annotazioni di crittografia definite nel DynamoDB Encryption Client for Java. Se le specifichi a livello di classe, diventano il valore predefinito per la classe.

```
// Sign only
@DoNotEncrypt

// Do nothing; not encrypted or signed
```

```
@DoNotTouch
```

Ad esempio, queste annotazioni firmano ma non crittografano l'attributo `PublicationYear` e non crittografano né firmano il valore attributo `ISBN`.

```
// Sign only (override the default)
@DoNotEncrypt
@DynamoDBAttribute(attributeName="PublicationYear")

// Do nothing (override the default)
@DoNotTouch
@DynamoDBAttribute(attributeName="ISBN")
```

Azioni relative agli attributi per Dynamo DBEncryptor

Per specificare le azioni degli attributi quando utilizzate DBEncryptor direttamente Dynamo, create un `HashMap` oggetto in cui le coppie nome-valore rappresentino i nomi degli attributi e le azioni specificate.

I valori sono validi per le operazioni di attributo definite nel tipo enumerato `EncryptionFlags`. Puoi utilizzare `ENCRYPT` e `SIGN` insieme o solo `SIGN` o ometterle entrambe. Tuttavia, se si utilizza `ENCRYPT` da solo, il client di crittografia DynamoDB genera un errore. Non puoi crittografare un attributo non firmato.

```
ENCRYPT
SIGN
```

> **Warning**
>
> Non crittografare gli attributi che vengono usati per la chiave primaria. Devono rimanere in testo semplice in modo che DynamoDB possa trovare l'elemento senza eseguire una scansione completa della tabella.

Se si specifica una chiave primaria nel contesto di crittografia e quindi si specifica `ENCRYPT` nell'azione di attributo per uno degli attributi della chiave primaria, il client di crittografia DynamoDB genera un'eccezione.

Ad esempio, il seguente codice Java crea un codice `actions` HashMap che crittografa e firma tutti gli attributi dell'elemento. `record` Le eccezioni sono la chiave di partizione e gli attributi della

chiave di ordinamento, che sono firmati ma non crittografati, e l'attributo `test`, che non è firmato o crittografato.

```
final EnumSet<EncryptionFlags> signOnly = EnumSet.of(EncryptionFlags.SIGN);
final EnumSet<EncryptionFlags> encryptAndSign = EnumSet.of(EncryptionFlags.ENCRYPT,
 EncryptionFlags.SIGN);
final Map<String, Set<EncryptionFlags>> actions = new HashMap<>();

for (final String attributeName : record.keySet()) {
  switch (attributeName) {
    case partitionKeyName: // no break; falls through to next case
    case sortKeyName:
      // Partition and sort keys must not be encrypted, but should be signed
      actions.put(attributeName, signOnly);
      break;
    case "test":
      // Don't encrypt or sign
      break;
    default:
      // Encrypt and sign everything else
      actions.put(attributeName, encryptAndSign);
      break;
  }
}
```

Successivamente, quando chiami il metodo encryptRecord del componente `DynamoDBEncryptor`, devi specificare la mappa come valore del parametro `attributeFlags`. Ad esempio, questa chiamata a `encryptRecord` utilizza la mappa `actions`.

```
// Encrypt the plaintext record
final Map<String, AttributeValue> encrypted_record = encryptor.encryptRecord(record,
 actions, encryptionContext);
```

Sovrascrivere i nomi delle tabelle

Nel DynamoDB Encryption Client, il nome della tabella DynamoDB è un elemento del contesto di crittografia DynamoDB che viene passato ai metodi di crittografia e decrittografia. Quando si crittografano o si firmano gli elementi della tabella, il contesto di crittografia DynamoDB, incluso il nome della tabella, è associato crittograficamente al testo cifrato. Se il contesto di crittografia DynamoDB passato al metodo decrypt non corrisponde al contesto di crittografia DynamoDB passato al metodo encrypt, l'operazione di decrittografia ha esito negativo.

Occasionalmente, il nome di una tabella cambia, ad esempio quando si esegue il backup di una tabella o si esegue un ripristino. point-in-time Quando si decrittografa o si verifica la firma di questi elementi, è necessario passare nello stesso contesto di crittografia DynamoDB utilizzato per crittografare e firmare gli elementi, incluso il nome della tabella originale. Il nome della tabella corrente non è necessario.

Quando si utilizzaDynamoDBEncryptor, si assembla manualmente il contesto di crittografia DynamoDB. Tuttavia, se si utilizza ilDynamoDBMapper, AttributeEncryptor crea automaticamente il contesto di crittografia DynamoDB, incluso il nome della tabella corrente. Per comunicare a AttributeEncryptor di creare un contesto di crittografia con un nome di tabella diverso, utilizza EncryptionContextOverrideOperator.

Ad esempio, il codice seguente crea istanze del provider di materiali crittografici (CMP) e di DynamoDBEncryptor. Quindi chiama il metodo setEncryptionContextOverrideOperator di DynamoDBEncryptor. Utilizza l'operatore overrideEncryptionContextTableName, che sovrascrive il nome di una tabella. Quando è configurato in questo modo, AttributeEncryptor crea un contesto di crittografia DynamoDB che newTableName include al posto di. oldTableName Per un esempio completo, consulta EncryptionContextOverridesWithDynamo DBMapper .java.

```
final DirectKmsMaterialProvider cmp = new DirectKmsMaterialProvider(kms, keyArn);
final DynamoDBEncryptor encryptor = DynamoDBEncryptor.getInstance(cmp);

encryptor.setEncryptionContextOverrideOperator(EncryptionContextOperators.overrideEncryptionCon
                oldTableName, newTableName));
```

Quando chiami il metodo di caricamento di DynamoDBMapper, che esegue la decrittografia e la verifica dell'item, devi specificare il nome della tabella originale.

```
mapper.load(itemClass, DynamoDBMapperConfig.builder()

 .withTableNameOverride(DynamoDBMapperConfig.TableNameOverride.withTableNameReplacement(oldTabl
                .build());
```

Puoi anche utilizzare l'operatore overrideEncryptionContextTableNameUsingMap, che sovrascrive più nomi di tabella.

Gli operatori che sovrascrivono i nomi di tabella vengono in genere utilizzati per la decrittografia dei dati e la verifica delle firme. Tuttavia, è possibile utilizzarli per impostare il nome della tabella nel contesto di crittografia DynamoDB su un valore diverso durante la crittografia e la firma.

Non utilizzare operatori che sovrascrivono i nomi di tabella se utilizzi `DynamoDBEncryptor`. Crea invece un contesto di crittografia con il nome della tabella originale e invialo al metodo di decrittografia.

Codice di esempio per il client di crittografia DynamoDB per Java

> **Note**
>
> La nostra libreria di crittografia lato client è stata rinominata Database Encryption SDK. AWS Il seguente argomento fornisce informazioni sulle versioni 1. x —2. x del DynamoDB Encryption Client for Java e versioni 1. x —3. x del client di crittografia DynamoDB per Python. Per ulteriori informazioni, consulta AWS Database Encryption SDK per il supporto della versione DynamoDB.

Gli esempi seguenti mostrano come utilizzare il DynamoDB Encryption Client for Java per proteggere gli elementi della tabella DynamoDB nell'applicazione. Puoi trovare altri esempi (e contribuire con i tuoi) nella directory examples del repository su. aws-dynamodb-encryption-java GitHub

Argomenti

- Usare la Dynamo DBEncryptor
- Usare Dynamo DBMapper

Usare la Dynamo DBEncryptor

Questo esempio mostra come utilizzare Dynamo di livello inferiore con Direct KMS DBEncryptor Provider. Il provider Direct KMS genera e protegge i suoi materiali crittografici con un AWS KMS keyin () specificato dall'utente. AWS Key Management Service AWS KMS

Puoi utilizzare qualsiasi fornitore di materiali crittografici (CMP) compatibile con`DynamoDBEncryptor`, e puoi utilizzare Direct KMS Provider con and. `DynamoDBMapper` AttributeEncryptor

Guarda l'esempio di codice completo: .java AwsKmsEncryptedItem

Fase 1: creazione del provider KMS diretto

Crea un'istanza del AWS KMS client con la regione specificata. Quindi, usa l'istanza del client per creare un'istanza del Direct KMS Provider con la tua preferita AWS KMS key.

Questo esempio utilizza Amazon Resource Name (ARN) per identificare AWS KMS key, ma puoi utilizzare qualsiasi identificatore di chiave valido.

```
final String keyArn = "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab";
final String region = "us-west-2";

final AWSKMS kms = AWSKMSClientBuilder.standard().withRegion(region).build();
final DirectKmsMaterialProvider cmp = new DirectKmsMaterialProvider(kms, keyArn);
```

Fase 2: creazione di un item

Questo esempio definisce un elemento `record` HashMap che rappresenta un elemento di tabella di esempio.

```
final String partitionKeyName = "partition_attribute";
final String sortKeyName = "sort_attribute";

final Map<String, AttributeValue> record = new HashMap<>();
record.put(partitionKeyName, new AttributeValue().withS("value1"));
record.put(sortKeyName, new AttributeValue().withN("55"));
record.put("example", new AttributeValue().withS("data"));
record.put("numbers", new AttributeValue().withN("99"));
record.put("binary", new AttributeValue().withB(ByteBuffer.wrap(new byte[]{0x00,
 0x01, 0x02})));
record.put("test", new AttributeValue().withS("test-value"));
```

Fase 3: Creare una dinamo DBEncryptor

Crea un'istanza del componente `DynamoDBEncryptor` con il provider KMS diretto.

```
final DynamoDBEncryptor encryptor = DynamoDBEncryptor.getInstance(cmp);
```

Fase 4: Creare un contesto di crittografia DynamoDB

Il contesto di crittografia DynamoDB contiene informazioni sulla struttura della tabella e su come viene crittografata e firmata. Se utilizzi il componente `DynamoDBMapper`, il componente `AttributeEncryptor` crea il contesto di crittografia per tuo conto.

```
final String tableName = "testTable";
```

```
final EncryptionContext encryptionContext = new EncryptionContext.Builder()
    .withTableName(tableName)
    .withHashKeyName(partitionKeyName)
    .withRangeKeyName(sortKeyName)
    .build();
```

Fase 5: creazione dell'oggetto delle operazioni di attributo.

Le operazioni di attributo determinano quali attributi dell'item sono crittografati e firmati, quali solo firmati e quali non sono né crittografati né firmati.

In Java, per specificare le azioni relative agli attributi, si crea una serie HashMap di coppie di nomi e `EncryptionFlags` valori degli attributi.

Ad esempio, il codice Java seguente crea un codice `actions` HashMap che crittografa e firma tutti gli attributi dell'`record`elemento, ad eccezione della chiave di partizione e degli attributi della chiave di ordinamento, che sono firmati ma non crittografati, e dell'`test`attributo, che non è firmato o crittografato.

```
final EnumSet<EncryptionFlags> signOnly = EnumSet.of(EncryptionFlags.SIGN);
final EnumSet<EncryptionFlags> encryptAndSign = EnumSet.of(EncryptionFlags.ENCRYPT,
 EncryptionFlags.SIGN);
final Map<String, Set<EncryptionFlags>> actions = new HashMap<>();

for (final String attributeName : record.keySet()) {
  switch (attributeName) {
    case partitionKeyName: // fall through to the next case
    case sortKeyName:
      // Partition and sort keys must not be encrypted, but should be signed
      actions.put(attributeName, signOnly);
      break;
    case "test":
      // Neither encrypted nor signed
      break;
    default:
      // Encrypt and sign all other attributes
      actions.put(attributeName, encryptAndSign);
      break;
  }
}
```

Fase 6: crittografia e firma dell'item

Per crittografare e firmare l'item della tabella, chiama il metodo `encryptRecord` nell'istanza del componente `DynamoDBEncryptor`. Specifica l'item della tabella (`record`), le operazioni di attributo (`actions`) e il contesto di crittografia (`encryptionContext`).

```
final Map<String, AttributeValue> encrypted_record = encryptor.encryptRecord(record,
 actions, encryptionContext);
```

Fase 7: Inserire l'elemento nella tabella DynamoDB

Infine, inserisci l'elemento crittografato e firmato nella tabella DynamoDB.

```
final AmazonDynamoDB ddb = AmazonDynamoDBClientBuilder.defaultClient();
ddb.putItem(tableName, encrypted_record);
```

Usare Dynamo DBMapper

L'esempio seguente mostra come utilizzare la classe helper DynamoDB mapper con Direct KMS Provider. Il provider Direct KMS genera e protegge i suoi materiali crittografici con un AWS KMS keyin () specificato dall'utente. AWS Key Management Service AWS KMS

Puoi usare qualunque provider di materiali crittografici (CMP) compatibile insieme al mappatore `DynamoDBMapper` e puoi utilizzare il provider KMS diretto con il componente `DynamoDBEncryptor` di livello inferiore.

Guarda l'esempio di codice completo: .java AwsKmsEncryptedObject

Fase 1: creazione del provider KMS diretto

Crea un'istanza del AWS KMS client con la regione specificata. Quindi, usa l'istanza del client per creare un'istanza del Direct KMS Provider con la tua preferita AWS KMS key.

Questo esempio utilizza Amazon Resource Name (ARN) per identificare AWS KMS key, ma puoi utilizzare qualsiasi identificatore di chiave valido.

```
final String keyArn = "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab";
final String region = "us-west-2";
```

```
final AWSKMS kms = AWSKMSClientBuilder.standard().withRegion(region).build();
final DirectKmsMaterialProvider cmp = new DirectKmsMaterialProvider(kms, keyArn);
```

Fase 2: Creare DynamoDB Encryptor e Dynamo DBMapper

Utilizza il provider Direct KMS creato nel passaggio precedente per creare un'istanza di DynamoDB Encryptor. È necessario creare un'istanza del DynamoDB Encryptor di livello inferiore per utilizzare DynamoDB Mapper.

Successivamente, crea un'istanza del database DynamoDB e una configurazione mapper e usale per creare un'istanza del DynamoDB Mapper.

> **Important**
>
> Quando utilizzi `DynamoDBMapper` per aggiungere o modificare item firmati (oppure crittografati e firmati), configuralo per utilizzare un comportamento di salvataggio, ad esempio PUT, che includa tutti gli attributi, come mostrato nel seguente esempio. In caso contrario, potresti non riuscire a decrittografare i dati.

```
final DynamoDBEncryptor encryptor = DynamoDBEncryptor.getInstance(cmp)
final AmazonDynamoDB ddb =
 AmazonDynamoDBClientBuilder.standard().withRegion(region).build();

DynamoDBMapperConfig mapperConfig =
 DynamoDBMapperConfig.builder().withSaveBehavior(SaveBehavior.PUT).build();
DynamoDBMapper mapper = new DynamoDBMapper(ddb, mapperConfig, new
 AttributeEncryptor(encryptor));
```

Fase 3: Definizione della tabella DynamoDB

Successivamente, definisci la tua tabella DynamoDB. Per specificare le operazioni di attributo, utilizza le annotazioni. Questo esempio crea una tabella DynamoDB e `DataPoJo` una classe che rappresenta gli elementi della tabella`ExampleTable`.

In questo esempio, gli attributi della chiave primaria saranno firmati ma non crittografati. Ciò vale per l'attributo `partition_attribute`, che viene annotato con `@DynamoDBHashKey`, e per l'attributo `sort_attribute`, che viene annotato con `@DynamoDBRangeKey`.

Gli attributi annotati con `@DynamoDBAttribute`, ad esempio `some numbers`, saranno crittografati e firmati. Le eccezioni sono gli attributi che utilizzano le annotazioni di crittografia `@DoNotEncrypt` (solo firma) o `@DoNotTouch` (non crittografare o firmare) definite dal DynamoDB Encryption Client. Ad esempio, poiché l'attributo `leave me` ha un'annotazione `@DoNotTouch`, non sarà crittografato né firmato.

```
@DynamoDBTable(tableName = "ExampleTable")
public static final class DataPoJo {
  private String partitionAttribute;
  private int sortAttribute;
  private String example;
  private long someNumbers;
  private byte[] someBinary;
  private String leaveMe;

  @DynamoDBHashKey(attributeName = "partition_attribute")
  public String getPartitionAttribute() {
    return partitionAttribute;
  }

  public void setPartitionAttribute(String partitionAttribute) {
    this.partitionAttribute = partitionAttribute;
  }

  @DynamoDBRangeKey(attributeName = "sort_attribute")
  public int getSortAttribute() {
    return sortAttribute;
  }

  public void setSortAttribute(int sortAttribute) {
    this.sortAttribute = sortAttribute;
  }

  @DynamoDBAttribute(attributeName = "example")
  public String getExample() {
    return example;
  }

  public void setExample(String example) {
    this.example = example;
  }

  @DynamoDBAttribute(attributeName = "some numbers")
```

```
  public long getSomeNumbers() {
    return someNumbers;
  }

  public void setSomeNumbers(long someNumbers) {
    this.someNumbers = someNumbers;
  }

  @DynamoDBAttribute(attributeName = "and some binary")
  public byte[] getSomeBinary() {
    return someBinary;
  }

  public void setSomeBinary(byte[] someBinary) {
    this.someBinary = someBinary;
  }

  @DynamoDBAttribute(attributeName = "leave me")
  @DoNotTouch
  public String getLeaveMe() {
    return leaveMe;
  }

  public void setLeaveMe(String leaveMe) {
    this.leaveMe = leaveMe;
  }

  @Override
  public String toString() {
    return "DataPoJo [partitionAttribute=" + partitionAttribute + ", sortAttribute="
        + sortAttribute + ", example=" + example + ", someNumbers=" + someNumbers
        + ", someBinary=" + Arrays.toString(someBinary) + ", leaveMe=" + leaveMe +
 "]";
  }
}
```

Fase 4: crittografia e salvataggio di un item della tabella

Ora, quando si crea un elemento della tabella e si utilizza DynamoDB Mapper per salvarlo, l'elemento viene automaticamente crittografato e firmato prima di essere aggiunto alla tabella.

In questo esempio viene definito un item della tabella denominato `record`. Prima che venga salvato nella tabella, i suoi attributi vengono crittografati e firmati in base alle annotazioni

nella classe `DataPoJo`. In questo caso, tutti gli attributi eccetto `PartitionAttribute`, `SortAttribute` e `LeaveMe` sono crittografati e firmati. `PartitionAttribute` e `SortAttributes` sono solo firmati. L'attributo `LeaveMe` non è crittografato né firmato.

Per crittografare e firmare l'item `record` e aggiungerlo alla tabella `ExampleTable`, chiama il metodo `save` della classe `DynamoDBMapper`. Poiché DynamoDB Mapper è configurato per utilizzare PUT il comportamento di salvataggio, l'elemento sostituisce qualsiasi elemento con le stesse chiavi primarie, anziché aggiornarlo. In questo modo le firme corrispondono ed è possibile decrittografare l'item quando si ottiene dalla tabella.

```
DataPoJo record = new DataPoJo();
record.setPartitionAttribute("is this");
record.setSortAttribute(55);
record.setExample("data");
record.setSomeNumbers(99);
record.setSomeBinary(new byte[]{0x00, 0x01, 0x02});
record.setLeaveMe("alone");

mapper.save(record);
```

## Client di crittografia DynamoDB per Python

> **Note**
>
> La nostra libreria di crittografia lato client è stata rinominata Database Encryption SDK. AWS Il seguente argomento fornisce informazioni sulle versioni 1. x —2. x del DynamoDB Encryption Client for Java e versioni 1. x —3. x del client di crittografia DynamoDB per Python. Per ulteriori informazioni, consulta AWS Database Encryption SDK per il supporto della versione DynamoDB.

Questo argomento spiega come installare e utilizzare il client di crittografia DynamoDB per Python. Puoi trovare il codice nel aws-dynamodb-encryption-pythonrepository su GitHub, incluso codice di esempio completo e testato per aiutarti a iniziare.

> **Note**
>
> Versioni 1. x. x e 2. x. x del DynamoDB Encryption Client for Python entreranno in vigore a luglio 2022. end-of-support Effettua l'upgrade a una versione più recente il prima possibile.

Argomenti



## Prerequisiti

Prima di installare il client di crittografia Amazon DynamoDB per Python, assicurati di avere i seguenti prerequisiti.

Una versione supportata di Python

Python 3.8 o versioni successive è richiesto dalle versioni 3.3.0 e successive di Amazon DynamoDB Encryption Client for Python. Per scaricare Python, consulta la pagina relativa ai download di Python.

Le versioni precedenti di Amazon DynamoDB Encryption Client for Python supportano Python 2.7 e Python 3.4 e versioni successive, ma consigliamo di utilizzare la versione più recente di DynamoDB Encryption Client.

Lo strumento di installazione pip per Python

Python 3.6 e versioni successive includono pip, anche se potresti volerlo aggiornare. Per ulteriori informazioni sull'aggiornamento o sull'installazione di pip, consulta la sezione relativa all'installazione nella documentazione su pip.

## Installazione

Usa pip per installare il client di crittografia Amazon DynamoDB per Python, come mostrato negli esempi seguenti.

Per installare la versione più recente

```
pip install dynamodb-encryption-sdk
```

Per ulteriori dettagli sull'utilizzo di pip per installare e aggiornare pacchetti, consulta la sezione relativa all'installazione dei pacchetti.

Il DynamoDB Encryption Client richiede la libreria di crittografia su tutte le piattaforme. Tutte le versioni di pip installano e creano la libreria di crittografia su Windows. pip 8.1 e le versioni successive installano e creano la libreria di crittografia su Linux. Se utilizzi una versione precedente di pip e il tuo ambiente Linux non possiede gli strumenti necessari per creare la libreria di crittografia, devi installarli. Per ulteriori informazioni, consulta la sezione relativa alla creazione di una crittografia in Linux.

È possibile scaricare l'ultima versione di sviluppo del DynamoDB Encryption Client dal aws-dynamodb-encryption-pythonrepository in poi. GitHub

Dopo aver installato il DynamoDB Encryption Client, inizia a guardare l'esempio di codice Python in questa guida.

Utilizzo del client di crittografia DynamoDB per Python

> **Note**
>
> La nostra libreria di crittografia lato client è stata rinominata Database Encryption SDK. AWS Il seguente argomento fornisce informazioni sulle versioni 1. x —2. x del DynamoDB Encryption Client for Java e versioni 1. x —3. x del client di crittografia DynamoDB per Python. Per ulteriori informazioni, consulta AWS Database Encryption SDK per il supporto della versione DynamoDB.

Questo argomento spiega alcune funzionalità del DynamoDB Encryption Client for Python che potrebbero non essere presenti in altre implementazioni del linguaggio di programmazione. Queste funzionalità sono progettate per semplificare l'utilizzo del DynamoDB Encryption Client nel modo più sicuro. Ti consigliamo di utilizzarle a meno che il tuo caso d'uso non sia insolito.

Per i dettagli sulla programmazione con il DynamoDB Encryption Client, consulta gli esempi in Python in questa guida, gli esempi nel repository GitHub su e la documentazione Python per aws-dynamodb-encryption-python il DynamoDB Encryption Client.

Argomenti

- Classi helper del client
- TableInfo classe
- Operazioni di attributo in Python

## Classi helper del client

Il client di crittografia DynamoDB per Python include diverse classi di supporto client che rispecchiano le classi Boto 3 per DynamoDB. Queste classi di supporto sono progettate per semplificare l'aggiunta di crittografia e firma all'applicazione DynamoDB esistente ed evitare i problemi più comuni, come segue:

- Impedite di crittografare la chiave primaria del vostro elemento, aggiungendo un'azione di override per la chiave primaria all'oggetto o generando un'eccezione se l'AttributeActionsoggetto dice esplicitamente al client di crittografare `AttributeActions` la chiave primaria. Se l'azione predefinita nell'oggetto `AttributeActions` è `DO_NOTHING`, le classi helper del client utilizzano tale azione per la chiave primaria. Altrimenti, utilizzano `SIGN_ONLY`.
- Crea un TableInfo oggetto e popola il contesto di crittografia DynamoDB in base a una chiamata a DynamoDB. Questo aiuta a garantire che il contesto di crittografia DynamoDB sia accurato e che il client possa identificare la chiave primaria.
- Supporta metodi, come `put_item` and`get_item`, che crittografano e decrittografano in modo trasparente gli elementi della tabella quando scrivi o leggi da una tabella DynamoDB. L'unico metodo non supportato è `update_item`.

Puoi utilizzare le classi helper del client al posto dell'interazione diretta con il componente di crittografia dell'item di livello inferiore. Utilizza queste classi a meno che non sia necessario impostare opzioni avanzate nel componente di crittografia dell'item.

Le classi helper del client includono:

- EncryptedTableper le applicazioni che utilizzano la risorsa Table in DynamoDB per elaborare una tabella alla volta.
- EncryptedResourceper le applicazioni che utilizzano la classe Service Resource in DynamoDB per l'elaborazione in batch.
- EncryptedClientper applicazioni che utilizzano il client di livello inferiore in DynamoDB.

Per utilizzare le classi client helper, il chiamante deve avere l'autorizzazione a chiamare l'operazione DynamoDB sulla tabella di destinazione DescribeTable.

TableInfo classe

La TableInfoclasse è una classe di supporto che rappresenta una tabella DynamoDB, completa di campi per la chiave primaria e gli indici secondari. Ti consente di ottenere informazioni precise e in tempo reale sulla tabella.

Se utilizzi una classe helper del client, questa crea e utilizza un oggetto `TableInfo` per tuo conto. Altrimenti, puoi crearne uno esplicitamente. Per vedere un esempio, consulta Utilizzo del componente di crittografia dell'item.

Quando si chiama il `refresh_indexed_attributes` metodo su un `TableInfo` oggetto, popola i valori delle proprietà dell'oggetto chiamando l'operazione DynamoDB DescribeTable. L'esecuzione di query sulla tabella è molto più affidabile rispetto all'impostazione come hardcoded dei nomi di indice. La `TableInfo` classe include anche una `encryption_context_values` proprietà che fornisce i valori richiesti per il contesto di crittografia DynamoDB.

Per utilizzare il `refresh_indexed_attributes` metodo, il chiamante deve avere il permesso di chiamare l'operazione DescribeTableDynamoDB sulla tabella di destinazione.

Operazioni di attributo in Python

Le operazioni di attributo comunicano al componente di crittografia dell'item quali operazioni effettuare su ciascun attributo dell'item. Per specificare le operazioni di attributo in Python, creare un oggetto `AttributeActions` con un'operazione predefinita ed eventuali eccezioni per determinati attributi. I valori validi vengono definiti nel tipo enumerato `CryptoAction`.

> **Important**
>
> Dopo aver utilizzato le azioni degli attributi per crittografare gli elementi della tabella, l'aggiunta o la rimozione di attributi dal modello di dati potrebbe causare un errore di convalida della firma che impedisce di decrittografare i dati. Per una spiegazione dettagliata, consulta Modifica del modello di dati.

```
DO_NOTHING = 0
SIGN_ONLY = 1
ENCRYPT_AND_SIGN = 2
```

Ad esempio, questo oggetto `AttributeActions` stabilisce l'operazione `ENCRYPT_AND_SIGN` come predefinita per tutti gli attributi e specifica le eccezioni per gli attributi `ISBN` e `PublicationYear`.

```
actions = AttributeActions(
    default_action=CryptoAction.ENCRYPT_AND_SIGN,
    attribute_actions={
        'ISBN': CryptoAction.DO_NOTHING,
        'PublicationYear': CryptoAction.SIGN_ONLY
    }
)
```

Se utilizzi una classe helper del client, non è necessario specificare un'operazione di attributo per gli attributi della chiave primaria. Le classi helper del client impediscono la crittografia della chiave primaria.

Se non utilizzi una classe helper del client e l'operazione predefinita è `ENCRYPT_AND_SIGN`, devi specificare un'operazione per la chiave primaria. L'operazione consigliata per le chiavi primarie è `SIGN_ONLY`. Per semplificare la procedura, utilizza il metodo `set_index_keys`, che utilizza l'operazione SIGN_ONLY per le chiavi primarie o l'operazione DO_NOTHING quando questa è impostata come operazione predefinita.

> **Warning**
>
> Non crittografare gli attributi che vengono usati per la chiave primaria. Devono rimanere in testo semplice in modo che DynamoDB possa trovare l'elemento senza eseguire una scansione completa della tabella.

```
actions = AttributeActions(
    default_action=CryptoAction.ENCRYPT_AND_SIGN,
)
actions.set_index_keys(*table_info.protected_index_keys())
```

Codice di esempio per il client di crittografia DynamoDB per Python

> **Note**
>
> La nostra libreria di crittografia lato client è stata rinominata Database Encryption SDK. AWS Il seguente argomento fornisce informazioni sulle versioni 1. x —2. x del DynamoDB Encryption Client for Java e versioni 1. x —3. x del client di crittografia DynamoDB per

Python. Per ulteriori informazioni, consulta AWS Database Encryption SDK per il supporto della versione DynamoDB.

Gli esempi seguenti mostrano come utilizzare il DynamoDB Encryption Client for Python per proteggere i dati DynamoDB nella tua applicazione. Puoi trovare altri esempi (e contribuire con i tuoi) nella directory examples del repository su. aws-dynamodb-encryption-python GitHub

Argomenti

- Usa la classe EncryptedTable client helper
- Utilizzo del componente di crittografia dell'item

Usa la classe EncryptedTable client helper

L'esempio seguente mostra come utilizzare il provider KMS diretto con la `EncryptedTable` classe helper del client. Questo esempio utilizza lo stesso provider di materiali crittografici dell'esempio Utilizzo del componente di crittografia dell'item seguente. Tuttavia, utilizza la classe `EncryptedTable` invece di interagire direttamente con il componente di crittografia dell'item di livello inferiore.

Confrontando questi esempi, puoi visualizzare il lavoro che la classe helper del client esegue per tuo conto. Ciò include la creazione del contesto di crittografia DynamoDB e la verifica che gli attributi della chiave primaria siano sempre firmati, ma mai crittografati. Per creare il contesto di crittografia e scoprire la chiave primaria, le classi client helper chiamano l'operazione DynamoDB DescribeTable. Per eseguire questo codice, devi disporre dell'autorizzazione per chiamare questa operazione.

Consulta l'esempio di codice completo: aws_kms_encrypted_table.py

Fase 1: creazione della tabella

Inizia creando un'istanza di una tabella DynamoDB standard con il nome della tabella.

```
table_name='test-table'
table = boto3.resource('dynamodb').Table(table_name)
```

Fase 2: creazione di un provider di materiali crittografici

Crea un'istanza del provider di materiali crittografici (cryptographic materials provider, CMP) selezionato.

Questo esempio utilizza il provider KMS diretto, ma puoi utilizzare qualunque CMP compatibile. Per creare un provider Direct KMS, specifica un. AWS KMS key Questo esempio utilizza l'Amazon Resource Name (ARN) di AWS KMS key, ma puoi utilizzare qualsiasi identificatore di chiave valido.

```
kms_key_id='arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab'
kms_cmp = AwsKmsCryptographicMaterialsProvider(key_id=kms_key_id)
```

Fase 3: creazione dell'oggetto delle operazioni di attributo.

Le operazioni di attributo comunicano al componente di crittografia dell'item quali operazioni effettuare su ciascun attributo dell'item. L'oggetto `AttributeActions` in questo esempio crittografa e firma tutti gli item tranne l'attributo `test`, che viene ignorato.

Non devi specificare operazioni di attributo per gli attributi della chiave primaria quando utilizzi una classe helper del client. La classe `EncryptedTable` firma gli attributi della chiave primaria, ma non li crittografa mai.

```
actions = AttributeActions(
    default_action=CryptoAction.ENCRYPT_AND_SIGN,
    attribute_actions={'test': CryptoAction.DO_NOTHING}
)
```

Fase 4: creazione della tabella crittografata

Crea la tabella criptata utilizzando la tabella standard, il provider KMS diretto e le operazioni di attributo. Questa fase completa la configurazione.

```
encrypted_table = EncryptedTable(
    table=table,
    materials_provider=kms_cmp,
    attribute_actions=actions
)
```

Fase 5: inserimento dell'item non crittografato nella tabella

Quando si chiama il `put_item` metodo su`encrypted_table`, gli elementi della tabella vengono crittografati, firmati e aggiunti in modo trasparente alla tabella DynamoDB.

Come prima cosa, definisci l'item della tabella.

```
plaintext_item = {
    'partition_attribute': 'value1',
    'sort_attribute': 55
    'example': 'data',
    'numbers': 99,
    'binary': Binary(b'\x00\x01\x02'),
    'test': 'test-value'
}
```

Inseriscilo quindi nella tabella.

```
encrypted_table.put_item(Item=plaintext_item)
```

Per ottenere l'elemento dalla tabella DynamoDB nella sua forma crittografata, chiamate `get_item` il metodo sull'oggetto. `table` Per ottenere l'item decrittografato, chiama il metodo `get_item` nell'oggetto `encrypted_table`.

Utilizzo del componente di crittografia dell'item

Questo esempio mostra come interagire direttamente con l'elemento encryptor nel DynamoDB Encryptor Client durante la crittografia degli elementi della tabella, invece di utilizzare le classi client helper che interagiscono con l'crittografo degli elementi per te.

Quando si utilizza questa tecnica, si creano manualmente il contesto di crittografia DynamoDB e l'oggetto `CryptoConfig` di configurazione (). Inoltre, si crittografano gli elementi in una chiamata e li si inserisce nella tabella DynamoDB in una chiamata separata. Ciò consente di personalizzare `put_item` le chiamate e utilizzare il DynamoDB Encryption Client per crittografare e firmare dati strutturati che non vengono mai inviati a DynamoDB.

Questo esempio utilizza il provider KMS diretto, ma puoi utilizzare qualunque CMP compatibile.

Consulta l'esempio di codice completo: aws_kms_encrypted_item.py

Fase 1: creazione della tabella

Inizia creando un'istanza di una risorsa di tabella DynamoDB standard con il nome della tabella.

```
table_name='test-table'
table = boto3.resource('dynamodb').Table(table_name)
```

Fase 2: creazione di un provider di materiali crittografici

Crea un'istanza del provider di materiali crittografici (cryptographic materials provider, CMP) selezionato.

Questo esempio utilizza il provider KMS diretto, ma puoi utilizzare qualunque CMP compatibile. Per creare un provider Direct KMS, specifica un. AWS KMS key Questo esempio utilizza l'Amazon Resource Name (ARN) di AWS KMS key, ma puoi utilizzare qualsiasi identificatore di chiave valido.

```
kms_key_id='arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab'
kms_cmp = AwsKmsCryptographicMaterialsProvider(key_id=kms_key_id)
```

Passaggio 3: usa la classe helper TableInfo

Per ottenere informazioni sulla tabella da DynamoDB, create un'istanza della TableInfoclasse helper. Quando lavori direttamente con il componente di crittografia dell'item, devi creare un'istanza `TableInfo` e chiamarne i metodi. Le classi helper del client eseguono questa operazione per tuo conto.

Il `refresh_indexed_attributes` metodo `TableInfo` utilizza l'operazione DescribeTableDynamoDB per ottenere informazioni accurate e in tempo reale sulla tabella. Queste comprendono la chiave primaria e gli indici secondari locali e globali. L'intermediario deve disporre dell'autorizzazione a chiamare `DescribeTable`.

```
table_info = TableInfo(name=table_name)
table_info.refresh_indexed_attributes(table.meta.client)
```

Fase 4: Creare il contesto di crittografia DynamoDB

Il contesto di crittografia DynamoDB contiene informazioni sulla struttura della tabella e su come viene crittografata e firmata. Questo esempio crea un contesto di crittografia DynamoDB in modo esplicito, poiché interagisce con l'elemento encryptor. Le classi client helper creano automaticamente il contesto di crittografia DynamoDB.

Per ottenere la chiave di partizione e la chiave di ordinamento, puoi utilizzare le proprietà della classe helper. TableInfo

```
index_key = {
    'partition_attribute': 'value1',
```

```
    'sort_attribute': 55
}

encryption_context = EncryptionContext(
    table_name=table_name,
    partition_key_name=table_info.primary_index.partition,
    sort_key_name=table_info.primary_index.sort,
    attributes=dict_to_ddb(index_key)
)
```

Fase 5: creazione dell'oggetto delle operazioni di attributo.

Le operazioni di attributo comunicano al componente di crittografia dell'item quali operazioni effettuare su ciascun attributo dell'item. L'oggetto `AttributeActions` in questo esempio crittografa e firma tutti gli item tranne gli attributi della chiave primaria, che vengono firmati ma non crittografati, e l'attributo `test`, che viene ignorato.

Quando interagisci direttamente con il componente di crittografia dell'item e l'operazione predefinita è `ENCRYPT_AND_SIGN`, devi specificare un'operazione alternativa per la chiave primaria. Puoi utilizzare il metodo `set_index_keys`, che utilizza `SIGN_ONLY` per la chiave primaria o `DO_NOTHING` se questa è l'operazione predefinita.

Per specificare la chiave primaria, questo esempio utilizza le chiavi di indice nell'TableInfooggetto, che viene popolato da una chiamata a DynamoDB. Questa tecnica è più sicura rispetto all'impostazione come hardcoded dei nomi della chiave primaria.

```
actions = AttributeActions(
    default_action=CryptoAction.ENCRYPT_AND_SIGN,
    attribute_actions={'test': CryptoAction.DO_NOTHING}
)
actions.set_index_keys(*table_info.protected_index_keys())
```

Fase 6: creazione della configurazione dell'item

Per configurare il client di crittografia DynamoDB, utilizza gli oggetti appena creati in CryptoConfiguna configurazione per l'elemento della tabella. Le classi client helper lo creano per te. CryptoConfig

```
crypto_config = CryptoConfig(
    materials_provider=kms_cmp,
    encryption_context=encryption_context,
    attribute_actions=actions
```

```
)
```

Fase 7: crittografia dell'item

Questo passaggio crittografa e firma l'elemento, ma non lo inserisce nella tabella DynamoDB.

Quando si utilizza una classe client helper, gli elementi vengono crittografati e firmati in modo trasparente e quindi aggiunti alla tabella DynamoDB quando si chiama il `put_item` metodo della classe helper. Quando utilizzi direttamente il componente di crittografia dell'item, le operazioni di crittografia e di inserimento sono indipendenti.

Crea prima un item non crittografato.

```
plaintext_item = {
    'partition_attribute': 'value1',
    'sort_key': 55,
    'example': 'data',
    'numbers': 99,
    'binary': Binary(b'\x00\x01\x02'),
    'test': 'test-value'
}
```

Poi, crittografalo e firmalo. Il metodo `encrypt_python_item` richiede l'oggetto di configurazione `CryptoConfig`.

```
encrypted_item = encrypt_python_item(plaintext_item, crypto_config)
```

Fase 8: inserimento dell'item nella tabella

Questo passaggio inserisce l'elemento crittografato e firmato nella tabella DynamoDB.

```
table.put_item(Item=encrypted_item)
```

Per visualizzare l'item crittografato, chiama il metodo `get_item` nell'oggetto `table` originale, invece che nell'oggetto `encrypted_table`. Il metodo ottiene l'item dalla tabella DynamoDB senza verificarlo né decrittografarlo.

```
encrypted_item = table.get_item(Key=partition_key)['Item']
```

L'immagine seguente mostra parte di un esempio di item della tabella crittografato e firmato.

I valori dell'attributo crittografato sono dati binari. I nomi e i valori degli attributi della chiave primaria (`partition_attribute` e `sort_attribute`) e l'attributo `test` restano non crittografati. L'output mostra anche l'attributo che contiene la firma (`*amzn-ddb-map-sig*`) e l'attributo di descrizione del materiale (`*amzn-ddb-map-desc*`).

```
{
    '*amzn-ddb-map-desc*': Binary(b'\x00\x00\x00\x00\x00\x00\x10amzn-ddb-env-alg\
\x00\x00\x00\xe0AQEBAHhA84wnXjEJdBbBBylRUFcZZK2j7xwh6UyLoL28nQ
+0FAAAAH4wfAYJKoZIhvcNAQcGoG8wbQIBADBoBgkqhkiG9w0BBwEwHgYJYIZIAWUDBAEuMBEEDPeFBydmoJD
izYl0R0C4M7wAK6E1/N/bgTmHI=\x00\x00\x00\x17amzn-ddb-map-signingAlg\x00\x00\x00\nHmacS
\x00\x00\x00\x11/CBC/PKCS5Padding\x00\x00\x00\x10amzn-ddb-sig-alg\x00\x00\x00\x0eHmac
\x00\x00\x00\x0faws-kms-ec-attr\x00\x00\x00\x06*keys*'),
    '*amzn-ddb-map-sig*': Binary(b"\xd3\xc6\xc7\n\xb7#\x13\xd1Y\xea\xe4.|^\xbd\xdf\xe
    'binary': Binary(b'!"\xc5\x92\xd7\x13\x1d\xe8Bs\x9b\x7f\xa8\x8e\x9c\xcf\x10\x1e\x
    'example': Binary(b"'b\x933\x9a+s\xf1\xd6a\xc5\xd5\x1aZ\xed\xd6\xce\xe9X\xf0T\xcb
    'numbers': Binary(b'\xd5\xa0\\d\xcc\x85\xf5\x1e\xb9-f!\xb9\xb8\x8a\x1aT\xbaq\xf7\
    'partition_attribute': 'value1',
    'sort_attribute': 55,
    'test': 'test-value'
}
```

## Modifica del modello di dati

> **Note**
>
> La nostra libreria di crittografia lato client è stata rinominata Database Encryption SDK. AWS Il seguente argomento fornisce informazioni sulle versioni 1. x —2. x del DynamoDB Encryption Client for Java e versioni 1. x —3. x del client di crittografia DynamoDB per Python. Per ulteriori informazioni, consulta AWS Database Encryption SDK per il supporto della versione DynamoDB.

Ogni volta che si crittografa o decrittografa un elemento, è necessario fornire azioni di attributo che indichino al DynamoDB Encryption Client quali attributi crittografare e firmare, quali attributi firmare (ma non crittografare) e quali ignorare. Le azioni degli attributi non vengono salvate nell'elemento crittografato e il DynamoDB Encryption Client non aggiorna automaticamente le azioni degli attributi.

> **Important**
>
> Il DynamoDB Encryption Client non supporta la crittografia dei dati delle tabelle DynamoDB esistenti e non crittografati.

Ogni volta che modifichi il modello di dati, ovvero quando aggiungi o rimuovi attributi dagli item della tabella, rischi di incorrere un errore. Se le operazioni di attributo specificate non valgono per tutti gli attributi nell'item, l'item potrebbe non essere crittografato e firmato secondo le tue intenzioni. E, cosa più importante, se le operazioni di attributo che fornisci quando decrittografi un item sono diverse da quelle che hai fornito quando lo hai crittografato, la verifica della firma potrebbe non andare a buon fine.

Ad esempio, se le operazioni di attributo usate per crittografare l'item gli comunicano di firmare l'attributo `test`, la firma nell'item comprenderà l'attributo `test`. Ma se le operazioni di attributo usate per decrittografare l'item non valgono per l'attributo `test`, la verifica non andrà a buon fine, perché il client proverà a verificare una firma che non comprende l'attributo `test`.

Questo è un problema particolare quando più applicazioni leggono e scrivono gli stessi elementi di DynamoDB, perché il DynamoDB Encryption Client deve calcolare la stessa firma per gli elementi in tutte le applicazioni. È anche un problema per qualsiasi applicazione distribuita perché le modifiche nelle operazioni di attributo devono propagarsi a tutti gli host. Anche se alle tabelle DynamoDB accede un solo host in un unico processo, stabilire una procedura basata sulle best practice aiuterà a prevenire gli errori nel caso in cui il progetto dovesse diventare più complesso.

Per evitare errori di convalida delle firme che impediscono la lettura degli item della tabella, utilizza le istruzioni riportate di seguito.

- Aggiungere un attributo: se il nuovo attributo modifica le azioni dell'attributo, implementa completamente la modifica dell'azione dell'attributo prima di includere il nuovo attributo in un elemento.
- Rimozione di un attributo: se smetti di utilizzare un attributo nei tuoi articoli, non modificare le azioni relative agli attributi.
- Modifica dell'azione: dopo aver utilizzato una configurazione delle azioni degli attributi per crittografare gli elementi della tabella, non è possibile modificare in modo sicuro l'azione predefinita o l'azione per un attributo esistente senza crittografare nuovamente ogni elemento della tabella.

Gli errori di convalida delle firme possono essere estremamente difficili da risolvere, quindi l'approccio migliore è prevenirli.

Argomenti

- Aggiunta di un attributo
- Rimozione di un attributo

## Aggiunta di un attributo

Quando aggiungi un nuovo attributo agli item della tabella, potrebbe essere necessario modificare le operazioni di attributo. Per evitare errori di convalida delle firme, ti consigliamo di implementare questa modifica in un processo a due fasi. Verifica che la prima fase sia completata prima di iniziare la seconda fase.

1. Modifica le operazioni di attributo in tutte le applicazioni che leggono o scrivono nella tabella. Distribuisci queste modifiche e conferma che l'aggiornamento è stato propagato a tutti gli host di destinazione.
2. Scrivi i valori nel nuovo attributo negli item della tabella.

Questo approccio in due fasi garantisce che tutte le applicazioni e gli host abbiano le stesse operazioni di attributo e calcolerà la stessa firma prima che qualsiasi elemento incontri il nuovo attributo. Ciò è importante anche quando l'operazione di attributo è Non fare nulla (non crittografare o firmare), perché l'impostazione predefinita per alcuni sistemi di crittografia è crittografare e firmare.

Negli esempi seguenti viene illustrato il codice per la prima fase di questo processo. Viene aggiunto un nuovo attributo dell'item, `link`, che memorizza un collegamento a un altro item della tabella. Poiché questo collegamento deve rimanere in testo normale, l'esempio assegna l'operazione di sola firma. Dopo aver distribuito completamente questa modifica e aver verificato che tutte le applicazioni e gli host abbiano le nuove operazioni di attributo, puoi iniziare a utilizzare l'attributo `link` negli item della tabella.

Java DynamoDB Mapper

Quando utilizzi `DynamoDB Mapper` e `AttributeEncryptor`, per impostazione predefinita, tutti gli attributi sono crittografati e firmati, tranne le chiavi primarie, che sono firmate ma non crittografate. Per specificare un'operazione di sola firma, utilizza l'annotazione `@DoNotEncrypt`.

In questo esempio viene utilizzata l'annotazione `@DoNotEncrypt` per il nuovo attributo `link`.

```
@DynamoDBTable(tableName = "ExampleTable")
public static final class DataPoJo {
  private String partitionAttribute;
  private int sortAttribute;
  private String link;

  @DynamoDBHashKey(attributeName = "partition_attribute")
```

```
  public String getPartitionAttribute() {
    return partitionAttribute;
  }

  public void setPartitionAttribute(String partitionAttribute) {
    this.partitionAttribute = partitionAttribute;
  }

  @DynamoDBRangeKey(attributeName = "sort_attribute")
  public int getSortAttribute() {
    return sortAttribute;
  }

  public void setSortAttribute(int sortAttribute) {
    this.sortAttribute = sortAttribute;
  }

  @DynamoDBAttribute(attributeName = "link")
  @DoNotEncrypt
  public String getLink() {
    return link;
  }

  public void setLink(String link) {
    this.link = link;
  }

  @Override
  public String toString() {
    return "DataPoJo [partitionAttribute=" + partitionAttribute + ",
        sortAttribute=" + sortAttribute + ",
        link=" + link + "]";
  }
}
```

Java DynamoDB encryptor

Nel criptatore DynamoDB di livello inferiore, è necessario impostare azioni per ogni attributo. In questo esempio viene utilizzata un'istruzione switch in cui l'impostazione predefinita è `encryptAndSign` e vengono specificate eccezioni per la chiave di partizione, la chiave di ordinamento e il nuovo attributo `link`. In questo esempio, se il codice attributo di collegamento non è stato distribuito completamente prima dell'utilizzo, l'attributo di collegamento verrà crittografato e firmato da alcune applicazioni, mentre verrà solo firmato da altre applicazioni.

```
for (final String attributeName : record.keySet()) {
    switch (attributeName) {
        case partitionKeyName:
            // fall through to the next case
        case sortKeyName:
            // partition and sort keys must be signed, but not encrypted
            actions.put(attributeName, signOnly);
            break;
        case "link":
            // only signed
            actions.put(attributeName, signOnly);
            break;
        default:
            // Encrypt and sign all other attributes
            actions.put(attributeName, encryptAndSign);
            break;
    }
}
```

Python

Nel DynamoDB Encryption Client for Python, è possibile specificare un'azione predefinita per tutti gli attributi e quindi specificare le eccezioni.

Se utilizzi una classe helper del client Python, non devi specificare un'operazione di attributo per gli attributi della chiave primaria. Le classi helper del client impediscono la crittografia della chiave primaria. Tuttavia, se utilizzi una classe helper del client, devi impostare l'operazione SIGN_ONLY sulla chiave di partizione e la chiave di ordinamento. Se esegui accidentalmente la crittografia della partizione o della chiave di ordinamento, non potrai recuperare i dati senza una scansione completa della tabella.

In questo esempio viene specificata un'eccezione per il nuovo attributo `link`, che ottiene l'operazione `SIGN_ONLY`.

```
actions = AttributeActions(
    default_action=CryptoAction.ENCRYPT_AND_SIGN,
    attribute_actions={
      'example': CryptoAction.DO_NOTHING,
      'link': CryptoAction.SIGN_ONLY
    }
)
```

### Rimozione di un attributo

Se non è più necessario un attributo negli elementi che sono stati crittografati con il DynamoDB Encryption Client, è possibile smettere di utilizzare l'attributo. Tuttavia, non eliminare o modificare l'operazione per tale attributo. In tal caso, se viene riscontrato un item con tale attributo, la firma calcolata per l'item non corrisponderà alla firma originale e la convalida della firma avrà esito negativo.

Anche se potresti essere tentato di rimuovere tutte le tracce dell'attributo dal tuo codice, aggiungi un commento che indica che l'item non è più utilizzato invece di eliminarlo. Anche se esegui una scansione completa della tabella per eliminare tutte le istanze dell'attributo, un item crittografato con tale attributo potrebbe essere memorizzato nella cache o in fase di elaborazione in qualche punto della configurazione.

## Risoluzione dei problemi nell'applicazione DynamoDB Encryption Client

> **Note**
>
> La nostra libreria di crittografia lato client è stata rinominata Database Encryption SDK. AWS Il seguente argomento fornisce informazioni sulle versioni 1. x —2. x del DynamoDB Encryption Client for Java e versioni 1. x —3. x del client di crittografia DynamoDB per Python. Per ulteriori informazioni, consulta AWS Database Encryption SDK per il supporto della versione DynamoDB.

Questa sezione descrive i problemi che si possono incontrare durante l'utilizzo del DynamoDB Encryption Client e offre suggerimenti per risolverli.

Per fornire feedback sul DynamoDB Encryption Client, segnala un problema nel aws-dynamodb-encryption-javarepository or. aws-dynamodb-encryption-python GitHub

Per fornire feedback su questa documentazione, utilizzare il link di feedback in qualsiasi pagina.

Argomenti

- Accesso negato
- La verifica della firma non va a buon fine
- Problemi con le tabelle globali delle versioni precedenti
- Scarse prestazioni del provider più recente

## Accesso negato

Problema: la tua applicazione non può accedere a una risorsa necessaria.

Suggerimento: scopri le autorizzazioni richieste e aggiungile al contesto di sicurezza in cui opera la tua applicazione.

Dettagli

Per eseguire un'applicazione che utilizza la libreria DynamoDB Encryption Client, il chiamante deve disporre dell'autorizzazione per utilizzarne i componenti. In caso contrario, le applicazioni non potranno accedere agli elementi richiesti.

- Il client di crittografia DynamoDB non richiede un account Amazon Web Services AWS() né dipende da alcun servizio. AWS Tuttavia, se l'applicazione utilizza l'account AWS, è necessario disporre di un utente Account AWS and che disponga dell'autorizzazione per utilizzare l'account.
- Il client di crittografia DynamoDB non richiede Amazon DynamoDB. Tuttavia, se l'applicazione che utilizza il client crea tabelle DynamoDB, inserisce elementi in una tabella o ottiene elementi da una tabella, il chiamante deve avere l'autorizzazione per utilizzare le operazioni DynamoDB richieste nel tuo. Account AWS Per i dettagli, consulta gli argomenti sul controllo degli accessi nella Amazon DynamoDB Developer Guide.
- Se l'applicazione utilizza una classe client helper nel client di crittografia DynamoDB per Python, il chiamante deve avere l'autorizzazione per chiamare l'operazione DynamoDB. DescribeTable
- Il client di crittografia DynamoDB non AWS Key Management Service richiede ().AWS KMSTuttavia, se l'applicazione utilizza un Direct KMS Materials Provider o utilizza un fornitorepiù recente con un provider store che utilizza Direct KMS Materials Provider AWS KMS, il chiamante deve essere autorizzato a utilizzare le AWS KMSGenerateDataKeyoperazioni e Decrypt.

## La verifica della firma non va a buon fine

Problema: un item non può essere decrittografato perché la verifica della firma non va a buon fine. L'item potrebbe anche non essere crittografato e firmato secondo le tue intenzioni.

Suggerimento: assicurati che tutte le operazioni di attributo valgano per tutti gli attributi dell'item. Quando decrittografi un item, assicurati di fornire operazioni di attributo che corrispondano a quelle utilizzate per crittografare l'item.

Dettagli

Le azioni relative agli attributi fornite indicano al DynamoDB Encryption Client quali attributi crittografare e firmare, quali attributi firmare (ma non crittografare) e quali ignorare.

Se le operazioni di attributo specificate non valgono per tutti gli attributi nell'item, l'item potrebbe non essere crittografato e firmato secondo le tue intenzioni. Se le operazioni di attributo che fornisci quando decrittografi un item sono diverse da quelle che hai fornito quando lo hai crittografato, la verifica della firma potrebbe non andare a buon fine. Questo è problema tipico delle applicazioni distribuite, in cui le nuove operazioni di attributo non sono state propagate a tutti gli host.

Gli errori di convalida delle firme sono difficili da risolvere. Per aiutare a prevenirli, adotta ulteriori precauzioni quando modifichi il modello di dati. Per informazioni dettagliate, consultare Modifica del modello di dati.

## Problemi con le tabelle globali delle versioni precedenti

Problema: gli elementi in una versione precedente della tabella globale di Amazon DynamoDB non possono essere decrittografati perché la verifica della firma non riesce.

Suggerimento: imposta le azioni relative agli attributi in modo che i campi di replica riservati non siano crittografati o firmati.

Dettagli

È possibile utilizzare il DynamoDB Encryption Client con le tabelle globali DynamoDB. Ti consigliamo di utilizzare tabelle globali con una chiave KMS multiregionale e di replicare la chiave KMS in tutti i luoghi in cui viene replicata la tabella globale. Regioni AWS

A partire dalla versione 2019.11.21 delle tabelle globali, puoi utilizzare le tabelle globali con il DynamoDB Encryption Client senza alcuna configurazione speciale. Tuttavia, se si utilizza la versione 2017.11.29 delle tabelle globali, è necessario assicurarsi che i campi di replica riservati non siano crittografati o firmati.

Se si utilizza la versione delle tabelle globali 2017.11.29, è necessario impostare le azioni degli attributi per i seguenti attributi `DO_NOTHING` in `@DoNotTouch`Java o Python.

- `aws:rep:deleting`
- `aws:rep:updatetime`
- `aws:rep:updateregion`

Se si utilizza un'altra versione delle tabelle globali, non è richiesta alcuna azione.

## Scarse prestazioni del provider più recente

Problema: l'applicazione è meno reattiva, soprattutto dopo l'aggiornamento a una versione più recente del DynamoDB Encryption Client.

Suggerimento: modifica il valore e la dimensione della cache. time-to-live

Dettagli

The Most Recent Provider è progettato per migliorare le prestazioni delle applicazioni che utilizzano il DynamoDB Encryption Client consentendo un riutilizzo limitato dei materiali crittografici. Quando configuri il provider più recente per la tua applicazione, devi bilanciare il miglioramento delle prestazioni con i problemi di sicurezza derivanti dalla memorizzazione nella cache e dal riutilizzo.

Nelle versioni più recenti di DynamoDB Encryption Client, time-to-live il valore (TTL) determina per quanto tempo possono essere utilizzati i provider di materiale crittografico memorizzati nella cache (). CMPs Il TTL determina anche la frequenza con cui il provider più recente verifica la presenza di una nuova versione della CMP.

Se il TTL è troppo lungo, l'applicazione potrebbe violare le regole aziendali o gli standard di sicurezza. Se il TTL è troppo breve, le chiamate frequenti al provider store possono far sì che quest'ultimo limiti le richieste provenienti dall'applicazione e da altre applicazioni che condividono l'account di servizio. Per risolvere questo problema, regolate il TTL e le dimensioni della cache a un valore che soddisfi gli obiettivi di latenza e disponibilità e sia conforme ai vostri standard di sicurezza. Per informazioni dettagliate, consultare Impostazione di un valore time-to-live.

# Rinomina del client di crittografia Amazon DynamoDB

La nostra libreria di crittografia lato client è stata rinominata Database Encryption SDK. AWS Questa guida per sviluppatori fornisce ancora informazioni sul DynamoDB Encryption Client.

Il 9 giugno 2023, la nostra libreria di crittografia lato client è stata rinominata Database Encryption SDK. AWS Il AWS Database Encryption SDK è compatibile con Amazon DynamoDB. Può decrittografare e leggere gli elementi crittografati dal client di crittografia DynamoDB legacy. Per ulteriori informazioni sulle versioni precedenti di DynamoDB Encryption Client, vedere. AWS Supporto della versione di Database Encryption SDK per DynamoDB

Il AWS Database Encryption SDK fornisce la versione 3. x della libreria di crittografia lato client Java per DynamoDB, che è una riscrittura importante del DynamoDB Encryption Client for Java. Include molti aggiornamenti, come un nuovo formato di dati strutturati, un supporto multitenancy migliorato, modifiche dello schema senza interruzioni e supporto per la crittografia ricercabile.

Per ulteriori informazioni sulle nuove funzionalità introdotte con AWS Database Encryption SDK, consulta i seguenti argomenti.

Crittografia ricercabile

È possibile progettare database in grado di cercare record crittografati senza decriptare l'intero database. A seconda del modello di minaccia e dei requisiti di interrogazione, è possibile utilizzare la crittografia ricercabile per eseguire ricerche con corrispondenza esatta o query complesse più personalizzate sui record crittografati.

Portachiavi

Il AWS Database Encryption SDK utilizza i portachiavi per eseguire la crittografia delle buste. I portachiavi generano, crittografano e decrittografano le chiavi di dati che proteggono i tuoi record. Il AWS Database Encryption SDK supporta AWS KMS portachiavi che utilizzano la crittografia simmetrica o RSA asimmetrica per proteggere le chiavi dei dati e i portachiavi AWS KMS gerarchici che consentono di AWS KMS keysproteggere i materiali crittografici con una chiave KMS con crittografia simmetrica senza chiamare ogni volta che si crittografa o decrittografa un record. AWS KMS Puoi anche specificare il materiale per le tue chiavi con portachiavi Raw AES e portachiavi Raw RSA.

## Modifiche allo schema senza interruzioni

Quando configuri il AWS Database Encryption SDK, fornisci azioni crittografiche che indicano al client quali campi crittografare e firmare, quali campi firmare (ma non crittografare) e quali ignorare. Dopo aver utilizzato AWS Database Encryption SDK per proteggere i record, puoi comunque apportare modifiche al tuo modello di dati. Puoi aggiornare le tue azioni crittografiche, come l'aggiunta o la rimozione di campi crittografati, in un'unica implementazione.

## Configurazione delle tabelle DynamoDB esistenti per la crittografia lato client

Le versioni precedenti di DynamoDB Encryption Client sono state progettate per essere implementate in nuove tabelle non popolate. Con il AWS Database Encryption SDK per DynamoDB, puoi migrare le tabelle Amazon DynamoDB esistenti alla versione 3. x della libreria di crittografia lato client Java per DynamoDB.

# Riferimento

La nostra libreria di crittografia lato client è stata rinominata AWS Database Encryption SDK. Questa guida per sviluppatori fornisce ancora informazioni sul DynamoDB Encryption Client.

I seguenti argomenti forniscono dettagli tecnici per il AWS Database Encryption SDK.

## Formato della descrizione del materiale

La descrizione del materiale funge da intestazione per un record crittografato. Quando crittografate e firmate i campi con il AWS Database Encryption SDK, il criptatore registra la descrizione del materiale mentre assembla i materiali crittografici e memorizza la descrizione del materiale in un nuovo campo (`aws_dbe_head`) che il crittografo aggiunge al record. La descrizione del materiale è una struttura di dati formattata portatile che contiene la chiave di dati crittografata e informazioni su come il record è stato crittografato e firmato. La tabella seguente descrive i valori che costituiscono la descrizione del materiale. I byte vengono aggiunti nell'ordine mostrato.

| Valore | Lunghezza in byte |
|---|---|
| Version | 1 |
| Signatures Enabled | 1 |
| Record ID | 32 |
| Encrypt Legend | Variabile |
| Encryption Context Length | 2 |
| ??? | Variabile |
| Encrypted Data Key Count | 1 |
| Encrypted Data Keys | Variabile |
| Record Commitment | 1 |

Versione

La versione del formato di questo `aws_dbe_head` campo.

Firme abilitate

Codifica se le firme digitali ECDSA sono abilitate per questo record.

| Valore in byte | Significato |
|---|---|
| `0x01` | Firme digitali ECDSA abilitate (impostazione predefinita) |
| `0x00` | Firme digitali ECDSA disattivate |

ID di registrazione

Un valore a 256 bit generato casualmente che identifica il record. L'ID del record:

- Identifica in modo univoco il record crittografato.
- Associa la descrizione del materiale al record crittografato.

Crittografa la leggenda

Una descrizione serializzata di quali campi autenticati sono stati crittografati. La Encrypt Legend viene utilizzata per determinare quali campi il metodo di decrittografia deve tentare di decrittografare.

| Valore in byte | Significato |
|---|---|
| `0x65` | `ENCRYPT_AND_SIGN` |
| `0x73` | `SIGN_ONLY` |

Encrypt Legend è serializzato come segue:

1. Lessicograficamente in base alla sequenza di byte che rappresenta il loro percorso canonico.
2. Per ogni campo, nell'ordine, aggiungi uno dei valori di byte specificati sopra per indicare se quel campo deve essere crittografato.

Lunghezza del contesto di crittografia

La lunghezza del contesto di crittografia. È un valore a 2 byte interpretato come un numero intero senza segno a 16 bit. La lunghezza massima è di 65.535 byte.

Contesto di crittografia

Un insieme di coppie nome-valore che contengono dati autenticati aggiuntivi arbitrari e non segreti.

Quando le firme digitali ECDSA sono abilitate, il contesto di crittografia contiene la coppia chiave-valore. `{"aws-crypto-footer-ecdsa-key": Qtxt}` `Qtxt`rappresenta il punto della curva ellittica Q compresso secondo SEC 1 versione 2.0 e quindi codificato in base 64.

Conteggio delle chiavi di dati crittografati

Il numero di chiavi di dati crittografati. È un valore di 1 byte interpretato come un numero intero senza segno a 8 bit che specifica il numero di chiavi dati crittografate. Il numero massimo di chiavi dati crittografate in ogni record è 255.

Chiavi dati crittografate

Sequenza di chiavi di dati crittografati. La lunghezza della sequenza è determinata dal numero di chiavi di dati crittografati e dalla lunghezza di ciascuna. La sequenza contiene almeno una chiave di dati crittografati.

La tabella seguente descrive i campi che costituiscono ogni chiave di dati crittografati. I byte vengono aggiunti nell'ordine mostrato.

Struttura chiave dati crittografati

| Campo | Lunghezza in byte |
| --- | --- |
| Key Provider ID Length | 2 |
| Key Provider ID | Variabile. Pari al valore specificato nei 2 byte precedenti (lunghezza ID provider chiave). |
| Key Provider Information Length | 2 |
| Key Provider Information | Variabile. Pari al valore specificato nei 2 byte precedenti (lunghezza informazione provider chiave). |

| Campo | Lunghezza in byte |
| --- | --- |
| Encrypted Data Key Length | 2 |
| Encrypted Data Key | Variabile. Pari al valore specificato nei 2 byte precedenti (lunghezza chiave dati crittografati). |

Lunghezza dell'ID del provider della chiave

Lunghezza dell'identificatore del provider della chiave. Si tratta di un valore di 2 byte interpretato come un numero intero senza segno a 16 bit che specifica il numero di byte che contengono l'ID del provider della chiave.

ID del fornitore di chiavi

Identificatore del provider della chiave. Viene utilizzato per indicare il provider della chiave dei dati crittografati ed è destinato a essere ampliabile.

Lunghezza delle informazioni chiave del fornitore

Lunghezza delle informazioni del provider della chiave. Si tratta di un valore di 2 byte interpretato come un numero intero senza segno a 16 bit che specifica il numero di byte che contengono le informazioni del provider della chiave.

Informazioni chiave sul fornitore

Informazioni provider chiave. Dipende dal provider di chiavi.

Quando utilizzi un AWS KMS portachiavi, questo valore contiene l'Amazon Resource Name (ARN) di. AWS KMS key

Lunghezza della chiave di dati crittografati

La lunghezza della chiave di dati crittografati. Si tratta di un valore di 2 byte interpretato come un numero intero senza segno a 16 bit che specifica il numero di byte che contengono la chiave di dati crittografati.

Chiave dati crittografata

Chiave di dati crittografati. È la chiave dati crittografata dal fornitore della chiave.

Impegno da record

Un hash distinto del codice di autenticazione dei messaggi basato su hash (HMAC) a 256 bit calcolato su tutti i precedenti byte di descrizione del materiale utilizzando la chiave di commit.

# AWS KMS Dettagli tecnici del portachiavi gerarchico

Il portachiavi AWS KMS Hierarchical utilizza una chiave dati univoca per crittografare ogni campo e crittografa ogni chiave di dati con una chiave di avvolgimento unica derivata da una chiave branch attiva. Utilizza una derivazione della chiave in modalità contatore con una funzione pseudocasuale con HMAC SHA-256 per derivare la chiave di wrapping a 32 byte con i seguenti input.

- Un sale casuale da 16 byte
- La chiave branch attiva
- Il valore codificato UTF-8 per l'identificatore del provider di chiavi "» aws-kms-hierarchy

Il portachiavi Hierarchical utilizza la chiave di wrapping derivata per crittografare una copia della chiave dati in chiaro utilizzando AES-GCM-256 con un tag di autenticazione a 16 byte e i seguenti input.

- La chiave di wrapping derivata viene utilizzata come chiave di crittografia AES-GCM
- La chiave dati viene utilizzata come messaggio AES-GCM
- Un vettore di inizializzazione casuale (IV) a 12 byte viene utilizzato come AES-GCM IV
- Dati autenticati aggiuntivi (AAD) contenenti i seguenti valori serializzati.

| Valore | Lunghezza in byte | Interpretato come |
|---|---|---|
| "aws-kms-hierarchy" | 17 | codificato UTF-8 |
| L'identificatore della chiave di filiale | Variabile | codificato UTF-8 |
| La versione Branch Key | 16 | codificato UTF-8 |
| Contesto di crittografia | Variabile | coppie chiave-valore codificate in UTF-8 |

# Cronologia dei documenti per la AWS Database Encryption SDK Developer Guide

La tabella seguente descrive le modifiche significative apportate a questa documentazione. Oltre a queste modifiche maggiori, aggiorniamo la documentazione di frequente per migliorare le descrizioni e gli esempi e per dar spazio al feedback inviatoci. Per ricevere una notifica sulle modifiche rilevanti, iscriversi al feed RSS.

| Modifica | Descrizione | Data |
| --- | --- | --- |
| Nuova caratteristica | È stata aggiunta la documentazione per il portachiavi AWS KMS ECDH e il portachiavi Raw ECDH. | 17 giugno 2024 |
| Versione General Availability (GA) | Presentazione del supporto per la libreria di crittografia lato client.NET per DynamoDB. | 17 gennaio 2024 |
| Versione General Availability (GA) | Documentazione aggiornata per la versione GA della versione 3. x della libreria di crittografia lato client Java per DynamoDB.<br><br>**Warning**<br>Le chiavi branch create durante la versione di anteprima per sviluppatori non sono più supportate. | 24 luglio 2023 |
| Rebranding del client di crittografia DynamoDB | La libreria di crittografia lato client viene rinominata | 9 giugno 2023 |

| | | |
|---|---|---|
| | Database Encryption SDK. AWS | |
| Versione di anteprima | Documentazione aggiunta e aggiornata per la versione 3. x della libreria di crittografia lato client Java per DynamoDB, che include un nuovo formato di dati strutturati, supporto multitenancy migliorato, modifiche dello schema senza interruzioni e supporto per la crittografia ricercabile. | 9 giugno 2023 |
| Modifica della documentazione | Sostituisci il AWS Key Management Service termine chiave master del cliente (CMK) con chiave KMS. AWS KMS key | 30 agosto 2021 |
| Nuova caratteristica | È stato aggiunto il supporto per le AWS Key Managemen t Service chiavi AWS KMS multiregionali (). Le chiavi multiregionali sono AWS KMS chiavi diverse Regioni AWS che possono essere utilizzat e in modo intercambiabile perché hanno lo stesso ID chiave e lo stesso materiale chiave. | 8 giugno 2021 |
| Nuovo esempio | Aggiunto un esempio di utilizzo di Dynamo DBMapper in Java. | 6 settembre 2018 |
| Supporto Python | Aggiunto il supporto per Python, oltre a Java. | 2 maggio 2018 |

| | | |
|---|---|---|
| Versione iniziale | Versione iniziale di questa documentazione. | 2 maggio 2018 |

Le traduzioni sono generate tramite traduzione automatica. In caso di conflitto tra il contenuto di una traduzione e la versione originale in Inglese, quest'ultima prevarrà.