Le migliori pratiche per l'utilizzo di AWS CDK in TypeScript per creare progetti IaC

AWS Linee guida prescrittive

AWS Linee guida prescrittive: Le migliori pratiche per l'utilizzo di AWS CDK in TypeScript per creare progetti IaC

Table of Contents

Introduzione 1
Obiettivi 1
Best practice 3
Organizzazione del codice per progetti su larga scala 3
Perché l'organizzazione del codice è importante 3
Come organizzare il codice in funzione della scalabilità 3
Esempio di organizzazione del codice 4
Sviluppo di pattern riutilizzabili 6
Abstract Factory 6
Chain of Responsibility 7
Creazione o estensione di costrutti 8
Che cos'è un costrutto 8
Quali sono i diversi tipi di costrutti 8
Come creare il proprio costrutto 9
Creazione o estensione di un costrutto L2 10
Creazione di un costrutto L3 11
Escape hatch 12
Risorse personalizzate 13
Segui le TypeScript migliori pratiche 16
Descrizione dei dati 16
Uso di enumerazioni 16
Uso di interfacce 17
Estensione delle interfacce 18
Evitare le interfacce vuote 18
Uso di factory 19
Uso della destrutturazione sulle proprietà 19
Definizione di convenzioni di denominazione standard 20
Non utilizzare la parola chiave var 20
Possibilità di utilizzare ESLint e Prettier 21
Uso di modificatori di accesso 21
Usa tipi di utilità 22
Scansione per individuare vulnerabilità di sicurezza ed errori di formattazione 23
Approcci e strumenti di sicurezza 23
Strumenti di sviluppo comuni 24

Sviluppo e perfezionamento della documentazione 24
Perché è richiesta la documentazione del codice per AWS CDK costrutti 25
Usando TypeDoc con AWS Costruisci una libreria 25
Adozione di un approccio di sviluppo basato su test 26
Test unitario 27
Test di integrazione 31
Uso del controllo di rilasci e versioni per i costrutti 31
Controllo della versione per AWS CDK 31
Deposito e imballaggio per AWS CDK costrutti 31
Construct Releasing per AWS CDK 32
Applicazione della gestione delle versioni della libreria 33
Domande frequenti 35
Quali problemi può TypeScript risolvere? 35
Perché dovrei usare? TypeScript 35
Devo usare AWS CDK o CloudFormation? 35
Cosa succede se AWS CDK non supporta una versione appena lanciata? Servizio AWS 35
Quali sono i diversi linguaggi di programmazione supportati da? AWS CDK 36
Quanto costa? AWS CDK 36
Passaggi successivi 37
Risorse 38
Cronologia dei documenti 39
..... xl

Le migliori pratiche per l'utilizzo di AWS CDK in TypeScript per creare progetti IaC

Sandeep Gawande, Mason Cahill, Sandip Gangapadhyay, Siamak Heshmati e Rajneesh Tyagi, Amazon Web Services (AWS)

Ottobre 2025 (cronologia dei documenti)

Questa guida fornisce consigli e best practice per l'utilizzo di AWS Cloud Development Kit (AWS CDK)in TypeScript per creare e implementare progetti Infrastructure as Code (IaC) su larga scala. AWS CDK È un framework per definire l'infrastruttura cloud in codice e fornire tale infrastruttura tramite. AWS CloudFormation Se non disponi di una struttura di progetto ben definita, creare e gestire una AWS CDK base di codice per progetti su larga scala può essere difficile. Per affrontare queste sfide, alcune organizzazioni utilizzano degli anti-pattern per i progetti su larga scala, ma tali pattern possono rallentare il progetto e creare altri problemi che hanno un impatto negativo sull'organizzazione. Ad esempio, gli anti-pattern possono complicare e rallentare l'onboarding degli sviluppatori, la correzione di bug e l'adozione di nuove funzionalità.

Questa guida fornisce un'alternativa all'uso degli anti-pattern e mostra come organizzare il codice a fini di scalabilità, esecuzione di test e allineamento con le best practice in materia di sicurezza. Puoi utilizzare questa guida per migliorare la qualità del codice per i tuoi progetti IaC e massimizzare l'agilità aziendale. Questa guida è destinata agli architetti, ai responsabili tecnici, agli ingegneri delle infrastrutture e a qualsiasi altro ruolo che cerchi di creare un progetto ben architettato AWS CDK per progetti su larga scala.

Obiettivi

- Costi ridotti: è possibile utilizzarli AWS CDK per progettare componenti riutilizzabili personalizzati che soddisfino i requisiti di sicurezza, conformità e governance dell'organizzazione. Puoi anche condividere facilmente i componenti all'interno dell'organizzazione, in modo da poter avviare rapidamente nuovi progetti allineati alle best practice per impostazione predefinita.
- Tempi di commercializzazione più rapidi: sfrutta le funzionalità familiari di AWS CDK per accelerare il processo di sviluppo. Ciò aumenta la riutilizzabilità per l'implementazione e riduce gli sforzi di sviluppo.
- Maggiore produttività degli sviluppatori: gli sviluppatori possono utilizzare linguaggi di programmazione familiari per definire l'infrastruttura. Questo aiuta gli sviluppatori a esprimere

e gestire AWS le risorse. Ciò può portare a una maggiore efficienza e collaborazione degli sviluppatori.

Best practice

Questa sezione fornisce una panoramica delle seguenti best practice:

- Organizzazione del codice per progetti su larga scala
- Sviluppo di pattern riutilizzabili
- Creazione o estensione di costrutti
- Segui le TypeScript migliori pratiche
- Scansione per individuare vulnerabilità di sicurezza ed errori di formattazione
- Sviluppo e perfezionamento della documentazione
- Adozione di un approccio di sviluppo basato su test
- Uso del controllo di rilasci e versioni per i costrutti
- Applicazione della gestione delle versioni della libreria

Organizzazione del codice per progetti su larga scala

Perché l'organizzazione del codice è importante

Per i AWS CDK progetti su larga scala è fondamentale avere una struttura ben definita e di alta qualità. Man mano che un progetto si amplia e il numero di funzionalità e costrutti supportati aumenta, la navigazione all'interno del codice diventa più difficile. Tale difficoltà può influire sulla produttività e rallentare l'onboarding degli sviluppatori.

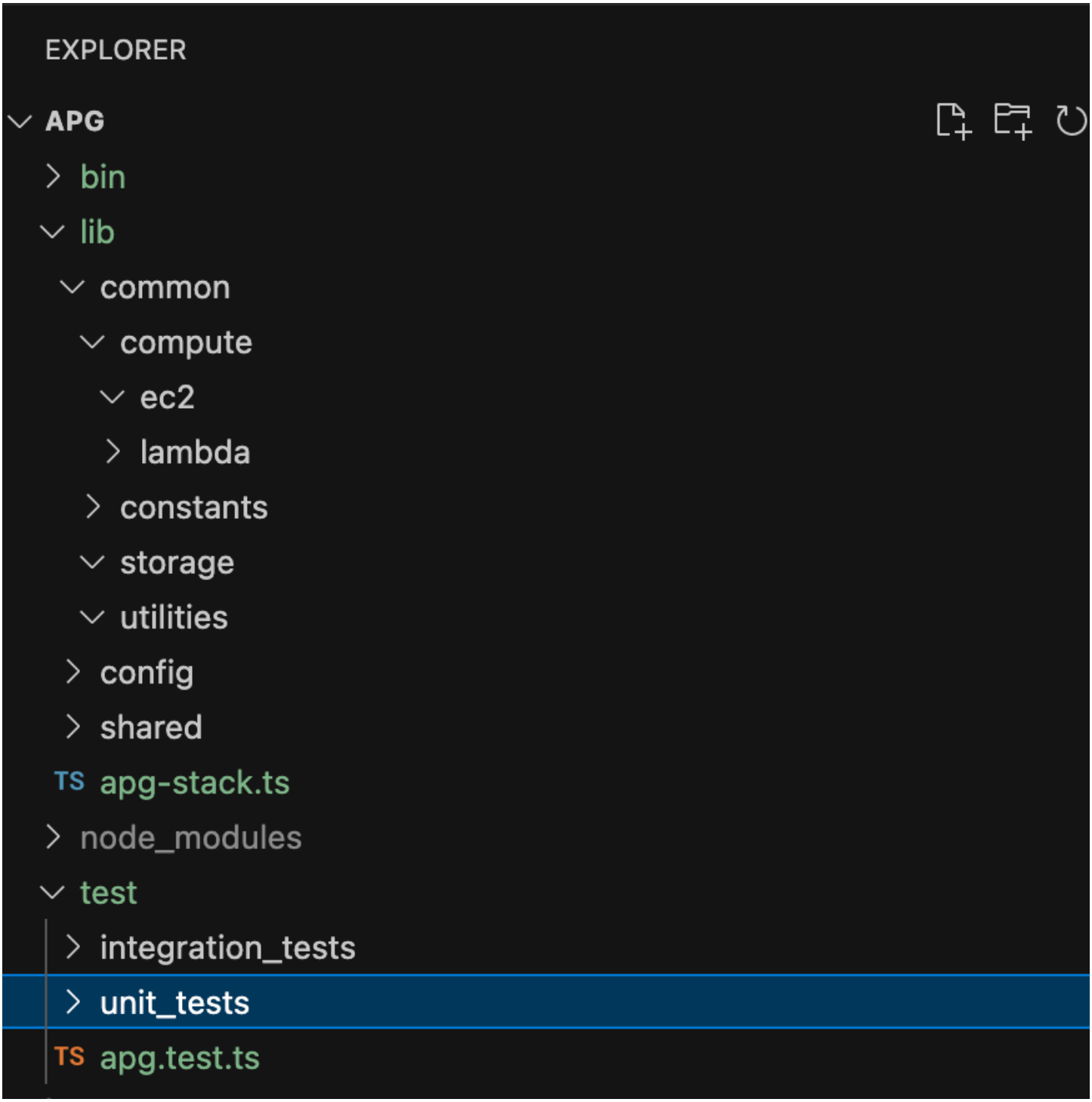
Come organizzare il codice in funzione della scalabilità

Per ottenere un livello elevato di flessibilità e leggibilità del codice, consigliamo di suddividerlo in parti logiche in base alla funzionalità. Tale divisione riflette il fatto che la maggior parte dei costrutti viene utilizzata in diversi domini aziendali. Ad esempio, sia le applicazioni frontend che quelle di backend potrebbero richiedere una AWS Lambda funzione e utilizzare lo stesso codice sorgente. Le factory possono creare oggetti senza esporre la logica di creazione al client e utilizzare un'interfaccia comune per fare riferimento agli oggetti appena creati. Puoi utilizzare una factory come pattern efficace per creare un comportamento coerente nella base di codice. Inoltre, una factory può fungere da unica fonte di verità per evitare che il codice sia ripetitivo e per agevolare la risoluzione dei problemi.

Per meglio comprendere il funzionamento delle factory, consideriamo l'esempio di un produttore di automobili. Un produttore di automobili non deve disporre delle conoscenze e delle infrastrutture necessarie per la produzione di pneumatici. Piuttosto, affida tali competenze a un produttore di pneumatici specializzato e poi si limita a ordinare gli pneumatici da quel produttore a seconda delle esigenze. Lo stesso principio si applica al codice. Ad esempio, puoi creare una factory Lambda in grado di sviluppare funzioni Lambda di alta qualità e quindi effettuare una chiamata alla factory Lambda nel codice ogni volta che devi creare una funzione Lambda. Analogamente, puoi utilizzare il medesimo processo di esternalizzazione per disaccoppiare l'applicazione e realizzare componenti modulari.

Esempio di organizzazione del codice

Il seguente progetto di TypeScript esempio, come mostrato nell'immagine seguente, include una cartella comune in cui è possibile conservare tutti i costrutti o le funzionalità comuni.



Ad esempio, la cartella compute (che si trova nella cartella common) contiene tutta la logica per diversi costrutti di calcolo. I nuovi sviluppatori possono aggiungere facilmente nuovi costrutti di calcolo senza condizionare le altre risorse. Tutti gli altri costrutti non avranno bisogno di creare nuove risorse internamente. Invece, a costrutti è sufficiente effettuare una chiamata alla factory di costrutti comune. Con la medesima procedura è possibile organizzare altri costrutti, come l'archiviazione.

Le configurazioni contengono dati basati sull'ambiente che devi disaccoppiare dalla cartella common in cui conservi la logica. Consigliamo di posizionare i dati config comuni in una cartella condivisa. Consigliamo inoltre di utilizzare la cartella utilities per svolgere tutte le funzioni helper e riordinare gli script.

Sviluppo di pattern riutilizzabili

I pattern di progettazione software sono soluzioni riutilizzabili a problemi comuni nello sviluppo di software. Fungono da guida o paradigma per aiutare gli ingegneri di software a creare prodotti che seguano le best practice. Questa sezione fornisce una panoramica di due modelli riutilizzabili che è possibile utilizzare nella propria AWS CDK codebase: il pattern Abstract Factory e il pattern Chain of Responsibility. Puoi utilizzare ciascun pattern come schema e personalizzarlo per il problema di progettazione specifico del tuo codice. Per ulteriori informazioni sui modelli di progettazione, vedete Design Patterns nella Refactoring.Guru documentazione.

Abstract Factory

Il pattern Abstract Factory fornisce interfacce per creare famiglie di oggetti correlati o dipendenti senza specificarne le classi concrete. Questo pattern è valido per i seguenti casi d'uso:

- Quando il client è indipendente dal modo in cui crei e componi gli oggetti nel sistema
- Quando il sistema è composto da più famiglie di oggetti progettate per essere utilizzate insieme
- Quando è necessario disporre di un valore di runtime per realizzare una particolare dipendenza

Per ulteriori informazioni sul pattern Abstract Factory, vedere Abstract Factory TypeScript nella Refactoring.Guru documentazione.

Il seguente esempio di codice mostra come è possibile utilizzare il pattern Abstract Factory per creare una factory di archiviazione Amazon Elastic Block Store (Amazon EBS).

```
abstract class EBSStorage {
    abstract initialize(): void;
}

class ProductEbs extends EBSStorage{
    constructor(value: String) {
        super();
        console.log(value);
```

```
    }
    initialize(): void {}
}

abstract class AbstractFactory {
    abstract createEbs(): EBSStorage
}

class EbsFactory extends AbstractFactory {
    createEbs(): ProductEbs{
        return new ProductEbs('EBS Created.')
    }
}

const ebs = new EbsFactory();
ebs.createEbs();
```

Chain of Responsibility

Chain of Responsibility è un pattern di progettazione comportamentale che consente di inoltrare una richiesta lungo la catena di potenziali gestori finché uno di essi non la gestirà. Il pattern Chain of Responsibility è valido per i seguenti casi d'uso:

- Quando più oggetti, determinati durante il runtime, sono candidati alla gestione di una richiesta
- Quando non si desidera specificare i gestori in modo esplicito nel codice
- Quando desideri inviare una richiesta a uno di più oggetti senza specificare in modo esplicito il destinatario

Per ulteriori informazioni sul modello Chain of Responsibility, consulta Chain of Responsibility TypeScript nella documentazione. Refactoring.Guru

Il codice seguente mostra un esempio di come il pattern Chain of Responsibility viene utilizzato per creare una serie di azioni necessarie per completare l'attività.

```
interface Handler {
    setNext(handler: Handler): Handler;
    handle(request: string): string;
}
abstract class AbstractHandler implements Handler
{
```

```
    private nextHandler: Handler;
    public setNext(handler: Handler): Handler {
        this.nextHandler = handler;
        return handler;
    }

    public handle(request: string): string {
        if (this.nextHandler) {
            return this.nextHandler.handle(request);
        }
        return '';
    }
}

class KMSHandler extends AbstractHandler {
    public handle(request: string): string {
        return super.handle(request);
    }
}
```

Creazione o estensione di costrutti

Che cos'è un costrutto

Un costrutto è l'elemento costitutivo di base di un' AWS CDK applicazione. Un costrutto può rappresentare una singola AWS risorsa, ad esempio un bucket Amazon Simple Storage Service (Amazon S3), oppure può essere un'astrazione di livello superiore composta da più risorse correlate. AWS I componenti di un costrutto possono includere una coda di lavoro con la capacità di elaborazione associata o un lavoro pianificato con risorse di monitoraggio e una dashboard. AWS CDK Include una raccolta di costrutti chiamata Construct Library. AWS La libreria contiene costrutti per tutti. Servizio AWS Puoi utilizzare Construct Hub per scoprire costrutti aggiuntivi forniti da AWS terze parti e dalla comunità open source. AWS CDK

Quali sono i diversi tipi di costrutti

Esistono tre diversi tipi di costrutti per: AWS CDK

- Costrutti L1: i costrutti di livello 1, o L1, sono esattamente le risorse definite da CloudFormation — né più né meno. È necessario fornire personalmente le risorse necessarie per la configurazione. Questi costrutti L1 sono molto semplici e devono essere configurati manualmente. I costrutti L1

hanno un `Cfn` prefisso e corrispondono direttamente alle specifiche. CloudFormation Servizi AWS I nuovi vengono supportati non AWS CDK appena CloudFormation viene fornito il supporto per questi servizi. CfnBucketè un buon esempio di costrutto L1. Questa classe rappresenta un bucket S3 in cui è necessario configurare esplicitamente tutte le proprietà. Ti consigliamo di utilizzare un costrutto L1 solo se non riesci a trovare il costrutto L2 o L3 corrispondente.

- Costrutti L2: i costrutti di livello 2, ovvero L2, presentano un codice boilerplate e una logica glue comuni. Questi costrutti sono dotati di comode impostazioni predefinite e riducono la quantità di conoscenze necessarie su di essi. I costrutti L2 utilizzano API basate sugli intenti per costruire le risorse e in genere incapsulano i moduli L1 corrispondenti. Un buon esempio di costrutto L2 è Bucket. Questa classe crea un bucket S3 con proprietà e metodi predefiniti come bucket.add (), che aggiunge una regola del ciclo di vita al LifeCycleRule bucket.
- Costrutti L3: un costrutto di livello 3, ovvero L3, è detto pattern. I costrutti L3 sono progettati per aiutarti a completare attività comuni in, che spesso coinvolgono più tipi di risorse. AWS Si tratta di costrutti ancora più specifici e rigidi dei costrutti L2 e servono a un caso d'uso specifico. Ad esempio, aws-ecs-patterns. ApplicationLoadBalancedFargateServiceconstruct rappresenta un'architettura che include un cluster di AWS Fargate contenitori che utilizza un Application Load Balancer. Un altro esempio è aws-apigateway. LambdaRestApicostruire. Questo rappresenta un'API Gateway Amazon API supportata da una funzione Lambda.

Man mano che i livelli dei costrutti aumentano, si formulano più ipotesi sull'uso che ne verrà fatto. Ciò consente di fornire interfacce con impostazioni predefinite più efficaci per casi d'uso altamente specifici.

Come creare il proprio costrutto

Per definire il proprio costrutto, è necessario seguire un approccio specifico. Questo perché tutti i costrutti estendono la classe `Construct`. La classe `Construct` è l'elemento base dell'albero dei costrutti. I costrutti sono implementati in classi che estendono la classe di base `Construct`. Tutti i costrutti accettano tre parametri al momento dell'inizializzazione:

- Ambito: il genitore o il proprietario di un costrutto, uno stack o un altro costrutto, che ne determina la posizione nell'albero dei costrutti. In genere devi superare `this` (o `self` in Python), che rappresenta l'oggetto corrente, per lo scope.
- id: un identificatore che deve essere univoco all'interno di tale scope. L'identificatore funge da namespace per tutto ciò che è definito all'interno del costrutto corrente e viene utilizzato per allocare identità univoche, ad esempio nomi di risorse e ID logici. CloudFormation

- Props: un insieme di proprietà che definiscono la configurazione iniziale del costrutto.

L'esempio seguente mostra come definire un costrutto.

```
import { Construct } from 'constructs';

export interface CustomProps {
  // List all the properties
  Name: string;
}
export class MyConstruct extends Construct {
  constructor(scope: Construct, id: string, props: CustomProps) {
    super(scope, id);

    // TODO
  }
}
```

Creazione o estensione di un costrutto L2

Un costrutto L2 rappresenta un «componente cloud» e incapsula tutto ciò che CloudFormation deve essere necessario per creare il componente. Un costrutto L2 può contenere una o più AWS risorse e sei libero di personalizzare tu stesso il costrutto. Il vantaggio di creare o estendere un costrutto L2 è che potete riutilizzare i componenti negli stack senza ridefinire il codice. CloudFormation È possibile semplicemente importare il costrutto come classe.

Quando esiste una relazione «is a» con un costrutto esistente, è possibile estendere un costrutto esistente per aggiungere funzionalità predefinite aggiuntive. È consigliabile riutilizzare le proprietà del costrutto L2 esistente. È possibile sovrascrivere le proprietà modificandole direttamente nel costruttore.

L'esempio seguente mostra come allinearsi alle best practice ed estendere un costrutto L2 esistente denominato `s3.Bucket`. L'estensione stabilisce proprietà predefinite, come `versioned`, `publicReadAccess`, `blockPublicAccess`, per assicurarsi che tutti gli oggetti (in questo esempio, i bucket S3) creati a partire da questo nuovo costrutto presentino sempre questi valori predefiniti impostati.

```
import * as s3 from 'aws-cdk-lib/aws-s3';
import { Construct } from 'constructs';
export class MySecureBucket extends s3.Bucket {
```

```
  constructor(scope: Construct, id: string, props?: s3.BucketProps) {

    super(scope, id, {
      ...props,
      versioned: true,
      publicReadAccess: false,
      blockPublicAccess: s3.BlockPublicAccess.BLOCK_ALL
    });
  }
}
```

Creazione di un costrutto L3

La composizione è la scelta migliore quando esiste una relazione «ha» con una composizione del costrutto esistente. Composizione significa che costruisci il tuo costrutto personale su altri costrutti esistenti. Puoi creare un pattern personalizzato per racchiudere tutte le risorse e i relativi valori predefiniti all'interno di un unico costrutto L3 di livello superiore che può essere condiviso. Il vantaggio di creare i propri costrutti (pattern) L3 è la possibilità di riutilizzare i componenti negli stack senza ridefinire il codice. È possibile semplicemente importare il costrutto come classe. Questi pattern sono progettati per aiutare gli utenti a effettuare in modo conciso il provisioning di più risorse in base a pattern comuni con una quantità limitata di conoscenze.

Il seguente esempio di codice crea un AWS CDK costrutto chiamato. `ExampleConstruct` È possibile utilizzare questo costrutto come modello per definire i propri componenti cloud.

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

export interface ExampleConstructProps {
  //insert properties you wish to expose
}

export class ExampleConstruct extends Construct {
  constructor(scope: Construct, id: string, props: ExampleConstructProps) {
    super(scope, id);
    //Insert the AWS components you wish to integrate
  }
}
```

L'esempio seguente mostra come importare il costrutto appena creato nell' AWS CDK applicazione o nello stack.

```
import { ExampleConstruct } from './lib/construct-name';
```

L'esempio seguente mostra come creare un'istanza del costrutto esteso a partire dalla classe base.

```
import { ExampleConstruct } from './lib/construct-name';

new ExampleConstruct(this, 'newConstruct', {
  //insert props which you exposed in the interface `ExampleConstructProps`
});
```

Per ulteriori informazioni, consultate AWS CDK Workshop nella documentazione del AWS CDK Workshop.

Escape hatch

È possibile utilizzare una porta di fuga in AWS CDK per salire di un livello di astrazione in modo da poter accedere al livello inferiore dei costrutti. Le botole di fuga vengono utilizzate per estendere il costrutto a funzionalità che non sono esposte nella versione corrente di ma disponibili in. AWS CloudFormation

È consigliabile utilizzare un escape hatch nei seguenti scenari:

- Una Servizio AWS funzionalità è disponibile tramite CloudFormation, ma non ci sono `Construct` costrutti corrispondenti.
- Una Servizio AWS funzionalità è disponibile tramite CloudFormation e sono presenti `Construct` dei costrutti per il servizio, ma questi non espongono ancora la funzionalità. Poiché `Construct` i costrutti vengono sviluppati «a mano», a volte possono rimanere indietro rispetto ai costrutti delle risorse. CloudFormation

L'esempio di codice seguente mostra un caso d'uso comune per l'uso di un escape hatch. In questo esempio, la funzionalità che non è ancora implementata nel costrutto di livello superiore serve ad aggiungere `httpPutResponseHopLimit` per il dimensionamento automatico di `LaunchConfiguration`.

```
const launchConfig = autoscaling.onDemandASG.node.findChild("LaunchConfig") as
 CfnLaunchConfiguration;
            launchConfig.metadataOptions = {
```

```
                    httpPutResponseHopLimit: autoscalingConfig.httpPutResponseHopLimit||
2
            }
```

L'esempio di codice precedente illustra il seguente flusso di lavoro:

1. Definisci il `AutoScalingGroup` utilizzando un costrutto L2. Il costrutto L2 non supporta l'aggiornamento di`httpPutResponseHopLimit`, quindi è necessario utilizzare una porta di fuga.
2. Utilizzate il `node.findChild()` metodo per individuare il `LaunchConfig` figlio specifico del `AutoScalingGroup` costrutto L2 e convertirlo come risorsa. `CfnLaunchConfiguration`
3. Ora puoi impostare la proprietà `launchConfig.metadataOptions` su `CfnLaunchConfiguration` L1.

Risorse personalizzate

È possibile utilizzare risorse personalizzate per scrivere una logica di provisioning personalizzata in modelli che CloudFormation viene eseguita ogni volta che si creano, aggiornano (se si modifica la risorsa personalizzata) o si eliminano gli stack. Ad esempio, è possibile utilizzare una risorsa personalizzata se si desidera includere risorse che non sono disponibili in. AWS CDK In questo modo puoi comunque gestire tutte le risorse correlate in un singolo stack.

La creazione di una risorsa personalizzata implica la scrittura di una funzione Lambda che risponda agli eventi del ciclo di vita CREATE, UPDATE e DELETE di una risorsa. Se la tua risorsa personalizzata deve effettuare solo una singola chiamata API, prendi in considerazione l'utilizzo del AwsCustomResourcecostrutto. In questo modo è possibile eseguire chiamate SDK arbitrarie durante una distribuzione. CloudFormation Altrimenti, consigliamo di scrivere una funzione Lambda personalizzata per eseguire il lavoro necessario.

Per ulteriori informazioni sulle risorse personalizzate, consulta Risorse personalizzate nella CloudFormation documentazione. Per un esempio di come utilizzare una risorsa personalizzata, consulta l'archivio Custom Resource su GitHub.

L'esempio seguente mostra come creare una classe di risorse personalizzata per avviare una funzione Lambda e CloudFormation inviare un segnale di successo o di fallimento.

```
import cdk = require('aws-cdk-lib');
import customResources = require('aws-cdk-lib/custom-resources');
import lambda = require('aws-cdk-lib/aws-lambda');
```

```
import { Construct } from 'constructs';

import fs = require('fs');

export interface MyCustomResourceProps {
  /**
   * Message to echo
   */
  message: string;
}

export class MyCustomResource extends Construct {
  public readonly response: string;

  constructor(scope: Construct, id: string, props: MyCustomResourceProps) {
    super(scope, id);

    const fn = new lambda.SingletonFunction(this, 'Singleton', {
      uuid: 'f7d4f730-4ee1-11e8-9c2d-fa7ae01bbebc',
      code: new lambda.InlineCode(fs.readFileSync('custom-resource-handler.py',
 { encoding: 'utf-8' })),
      handler: 'index.main',
      timeout: cdk.Duration.seconds(300),
      runtime: lambda.Runtime.PYTHON_3_6,
    });

    const provider = new customResources.Provider(this, 'Provider', {
      onEventHandler: fn,
    });

    const resource = new cdk.CustomResource(this, 'Resource', {
      serviceToken: provider.serviceToken,
      properties: props,
    });

    this.response = resource.getAtt('Response').toString();
    }
}
```

L'esempio seguente mostra la logica principale della risorsa personalizzata.

```
def main(event, context):
    import logging as log
```

```
    import cfnresponse
    log.getLogger().setLevel(log.INFO)

    # This needs to change if there are to be multiple resources in the same stack
    physical_id = 'TheOnlyCustomResource'

    try:
        log.info('Input event: %s', event)

        # Check if this is a Create and we're failing Creates
        if event['RequestType'] == 'Create' and
 event['ResourceProperties'].get('FailCreate', False):
            raise RuntimeError('Create failure requested')

        # Do the thing
        message = event['ResourceProperties']['Message']
        attributes = {
            'Response': 'You said "%s"' % message
        }

        cfnresponse.send(event, context, cfnresponse.SUCCESS, attributes, physical_id)
    except Exception as e:
        log.exception(e)
        # cfnresponse's error message is always "see CloudWatch"
        cfnresponse.send(event, context, cfnresponse.FAILED, {}, physical_id)
```

L'esempio seguente mostra come lo AWS CDK stack chiama la risorsa personalizzata.

```
import cdk = require('aws-cdk-lib');
import { MyCustomResource } from './my-custom-resource';

/**
 * A stack that sets up MyCustomResource and shows how to get an attribute from it
 */
class MyStack extends cdk.Stack {
  constructor(scope: cdk.App, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    const resource = new MyCustomResource(this, 'DemoResource', {
      message: 'CustomResource says hello',
    });

    // Publish the custom resource output
```

```
    new cdk.CfnOutput(this, 'ResponseMessage', {
      description: 'The message that came back from the Custom Resource',
      value: resource.response
    });
  }
}

const app = new cdk.App();
new MyStack(app, 'CustomResourceDemoStack');
app.synth();
```

Segui le TypeScript migliori pratiche

TypeScript è un linguaggio che estende le funzionalità di JavaScript. È un linguaggio fortemente tipizzato e orientato agli oggetti. È possibile TypeScript utilizzarlo per specificare i tipi di dati che vengono trasmessi all'interno del codice e ha la possibilità di segnalare errori quando i tipi non corrispondono. Questa sezione fornisce una panoramica delle TypeScript migliori pratiche.

Descrizione dei dati

È possibile TypeScript utilizzarlo per descrivere la forma degli oggetti e delle funzioni nel codice. Utilizzare il tipo `any` equivale a disattivare il controllo del tipo per una variabile. È preferibile evitare l'uso di `any` nel codice. Ecco un esempio.

```
type Result = "success" | "failure"
function verifyResult(result: Result) {
    if (result === "success") {
        console.log("Passed");
    } else {
        console.log("Failed")
    }
}
```

Uso di enumerazioni

Puoi utilizzare le enumerazioni per definire un insieme di costanti denominate e definire standard che possono essere riutilizzati nella base di codice. Consigliamo di esportare le enumerazioni una volta a livello globale e poi di consentire ad altre classi di importarle e utilizzarle. Supponete di voler creare una serie di azioni possibili per acquisire gli eventi nella vostra base di codice. TypeScript

fornisce enumerazioni sia numeriche che basate su stringhe. Nell'esempio seguente viene utilizzata un'enumerazione.

```
enum EventType {
    Create,
    Delete,
    Update
}

class InfraEvent {
    constructor(event: EventType) {
        if (event === EventType.Create) {
            // Call for other function
            console.log(`Event Captured :${event}`);
        }
    }
}

let eventSource: EventType = EventType.Create;
const eventExample = new InfraEvent(eventSource)
```

Uso di interfacce

Un'interfaccia è un contratto per la classe. Se crei un contratto, gli utenti sono tenuti a rispettarlo. Nell'esempio seguente, viene utilizzata un'interfaccia per standardizzare `props` e assicurare che i chiamanti forniscano il parametro previsto quando si utilizza questa classe.

```
import { Stack, App } from "aws-cdk-lib";
import { Construct } from "constructs";

interface BucketProps {
    name: string;
    region: string;
    encryption: boolean;

}

class S3Bucket extends Stack {
    constructor(scope: Construct, props: BucketProps) {
        super(scope);
        console.log(props.name);
```

```
    }
}
const app = App();
const myS3Bucket = new S3Bucket(app, {
    name: "amzn-s3-demo-bucket",
    region: "us-east-1",
    encryption: false
})
```

Alcune proprietà possono essere modificate solo al momento della creazione dell'oggetto. Puoi specificarlo inserendo `readonly` prima del nome della proprietà, come illustrato nell'esempio seguente.

```
interface Position {
    readonly latitude: number;
    readonly longitute: number;
}
```

Estensione delle interfacce

L'estensione delle interfacce riduce la duplicazione, perché non è necessario copiare le proprietà da un'interfaccia all'altra. Inoltre, il lettore del codice può comprendere facilmente le relazioni all'interno dell'applicazione.

```
interface BaseInterface{
   name: string;
 }
 interface EncryptedVolume extends BaseInterface{
     keyName: string;
 }
 interface UnencryptedVolume extends BaseInterface {
     tags: string[];
 }
```

Evitare le interfacce vuote

Consigliamo di evitare le interfacce vuote a causa dei potenziali rischi che creano. Nell'esempio seguente, c'è un'interfaccia vuota chiamata. `BucketProps` Gli oggetti `myS3Bucket1` e `myS3Bucket2` sono entrambi validi, ma seguono standard diversi perché l'interfaccia non applica

alcun contratto. Il codice seguente compilerà e stamperà le proprietà, ma in tal modo si introdurranno incoerenze nell'applicazione.

```
interface BucketProps {}

class S3Bucket implements BucketProps {
    constructor(props: BucketProps){
        console.log(props);
    }
}

const myS3Bucket1 = new S3Bucket({
    name: "amzn-s3-demo-bucket",
    region: "us-east-1",
    encryption: false,
});

const myS3Bucket2 = new S3Bucket({
    name: "amzn-s3-demo-bucket",
});
```

Uso di factory

In un pattern Abstract Factory, un'interfaccia è responsabile della creazione di una factory di oggetti correlati senza specificarne esplicitamente le classi. Ad esempio, puoi creare una factory Lambda per la generazione di funzioni Lambda. Invece di creare una nuova funzione Lambda all'interno del tuo costrutto, deleghi il processo di creazione alla fabbrica. Per ulteriori informazioni su questo modello di progettazione, consulta Abstract Factory TypeScript nella documentazione. Refactoring.Guru

Uso della destrutturazione sulle proprietà

La destrutturazione, introdotta in ECMAScript 6 (ES6), è una JavaScript funzionalità che offre la possibilità di estrarre più parti di dati da un array o da un oggetto e assegnarle alle proprie variabili.

```
const object = {
    objname: "obj",
    scope: "this",
};

const oName = object.objname;
const oScop = object.scope;
```

```
const { objname, scope } = object;
```

Definizione di convenzioni di denominazione standard

L'applicazione di una convenzione di denominazione mantiene coerente la base di codice e riduce il sovraccarico quando si pensa al nome da assegnare a una variabile. Consigliamo quanto segue:

- Utilizza camelCase per i nomi di variabili e funzioni.
- Utilizzate UPPER_CASE per le costanti globali per indicare chiaramente valori immutabili in fase di compilazione.
- Utilizzare per i nomi delle classi e dei nomi di interfaccia. PascalCase
- Utilizza camelCase per i membri delle interfacce.
- Utilizzare PascalCase per i nomi dei tipi e i nomi enum.
- Assegna un nome ai file con camelCase (ad esempio `ebsVolumes.tsx` o `storage.ts`)

Di seguito sono riportati alcuni esempi di queste convenzioni di denominazione consigliate:

```
// Variables and functions
const userName = 'john';
function getUserData() { }

// Global constants
const MAX_RETRY_ATTEMPTS = 3;
const API_BASE_URL = 'https://api.example.com';

// Classes and interfaces
class DatabaseConnection { }
interface UserProfile { }

// Types and enums
type ResponseStatus = 'success' | 'error';
enum HttpStatusCode { }
```

Non utilizzare la parola chiave var

L'`let`istruzione viene utilizzata per dichiarare una variabile locale in TypeScript. È simile alla `var` parola chiave, ma presenta alcune restrizioni nell'ambito rispetto alla `var` parola chiave. Una variabile

dichiarata in un blocco con `let` è disponibile per l'uso solo all'interno di quel blocco. La `var` parola chiave non può avere un ambito a blocchi, il che significa che è possibile accedervi al di fuori di un particolare blocco (rappresentato da`{}`) ma non al di fuori della funzione in cui è definita. È possibile dichiarare nuovamente e aggiornare le variabili. `var` È consigliabile evitare di utilizzare la `var` parola chiave.

Possibilità di utilizzare ESLint e Prettier

ESLint analizza staticamente il codice per individuare rapidamente i problemi. Puoi utilizzare ESLint per creare una serie di asserzioni (denominate regole lint) che definiscono come dovrebbe apparire o comportarsi il codice. ESLint offre anche suggerimenti di correzione automatica per aiutarti a migliorare il codice. Infine, ESLint consente di caricare regole lint dai plugin condivisi.

Prettier è un noto formattatore di codice che supporta una varietà di linguaggi di programmazione diversi. Puoi utilizzare Prettier per impostare lo stile del codice in modo da evitarne la formattazione manuale. Dopo l'installazione, puoi aggiornare il file `package.json` ed eseguire i comandi `npm run format` e `npm run lint`.

L'esempio seguente mostra come abilitare ESLint e il formattatore Prettier per il tuo progetto. AWS CDK

```
"scripts": {
    "build": "tsc",
    "watch": "tsc -w",
    "test": "jest",
    "cdk": "cdk",
    "lint": "eslint --ext .js,.ts .",
    "format": "prettier --ignore-path .gitignore --write '**/*.+(js|ts|json)'"
}
```

Uso di modificatori di accesso

Il modificatore privato in TypeScript limita la visibilità solo alla stessa classe. Quando aggiungi il modificatore privato a una proprietà o a un metodo, puoi accedere a tale proprietà o metodo all'interno della stessa classe.

Il modificatore pubblico consente l'accesso alle proprietà e ai metodi delle classi da tutte le posizioni. Se non specifichi alcun modificatore di accesso per proprietà e metodi, utilizzeranno il modificatore pubblico per impostazione predefinita.

Il modificatore protetto consente l'accesso alle proprietà e ai metodi di una classe all'interno della stessa classe e delle sottoclassi. Utilizzate il modificatore protetto quando pretendete di creare sottoclassi nella vostra applicazione. AWS CDK

Usa tipi di utilità

I tipi di utilità in TypeScript sono funzioni di tipo predefinito che eseguono trasformazioni e operazioni su tipi esistenti. Ciò consente di creare nuovi tipi in base ai tipi esistenti. Ad esempio, è possibile modificare o estrarre proprietà, rendere le proprietà facoltative o obbligatorie o creare versioni immutabili dei tipi. Utilizzando i tipi di utilità, è possibile definire tipi più precisi e rilevare potenziali errori in fase di compilazione.

Tipo parziale < >

`Partial`contrassegna tutti i membri di un tipo di input `Type` come facoltativi. Questa utilità restituisce un tipo che rappresenta tutti i sottoinsiemi di un determinato tipo. Di seguito è riportato un esempio di `Partial`.

```
interface Dog {
  name: string;
  age: number;
  breed: string;
  weight: number;
}

let partialDog: Partial<Dog> = {};
```

Tipo richiesto < >

`Required`fa il contrario di`Partial`. Rende tutti i membri di un tipo di input `Type` non opzionali (in altre parole, obbligatori). Di seguito è riportato un esempio di `Required`.

```
interface Dog {
  name: string;
  age: number;
  breed: string;
  weight?: number;
}
```

```
let dog: Required<Dog> = {
  name: "scruffy",
  age: 5,
  breed: "labrador",
  weight: 55 // "Required" forces weight to be defined
};
```

Scansione per individuare vulnerabilità di sicurezza ed errori di formattazione

L'Infrastruttura come codice (IaC) e l'automazione sono diventate aspetti essenziali per le imprese. Data la robustezza di IaC, hai una grande responsabilità per quanto riguarda la gestione dei rischi di sicurezza. I rischi relativi alla sicurezza comuni di IaC possono includere quanto segue:

- Over-permissive AWS Identity and Access Management privilegi (IAM)
- Gruppi di sicurezza aperti
- Risorse non crittografate
- Log di accesso non attivati

Approcci e strumenti di sicurezza

Consigliamo di implementare i seguenti approcci di sicurezza:

- Rilevamento delle vulnerabilità in fase di sviluppo: la correzione delle vulnerabilità in fase di produzione è costosa e richiede molto tempo a causa della complessità dello sviluppo e della distribuzione delle patch software. Inoltre, le vulnerabilità in fase di produzione comportano il rischio di sfruttamento. Consigliamo di utilizzare la scansione del codice sulle risorse IaC in modo da rilevare e correggere le vulnerabilità prima del rilascio in produzione.
- Conformità e riparazione automatica: AWS Config offre regole AWS gestite. Queste regole ti aiutano a far rispettare la conformità e ti consentono di tentare la riparazione automatica utilizzando l'automazione.AWS Systems Manager È inoltre possibile creare e associare documenti di automazione personalizzati utilizzando le regole. AWS Config

Strumenti di sviluppo comuni

Gli strumenti descritti in questa sezione consentono di estendere le funzionalità integrate con regole personalizzate. Ti consigliamo di allineare le regole personalizzate agli standard della tua organizzazione. Ecco alcuni strumenti di sviluppo comuni da prendere in considerazione:

- Usa cdk-nag per verificare che i costrutti all'interno di un determinato ambito siano conformi a un insieme definito di regole. Puoi anche utilizzare cdk-nag per la soppressione delle regole e la creazione di report sulla conformità. Lo strumento cdk-nag convalida i costrutti estendendo gli aspetti di. AWS CDK Per ulteriori informazioni, consulta Gestire la sicurezza e la conformità delle applicazioni con e cdk-nag nel blog AWS Cloud Development Kit (AWS CDK). AWS DevOps
- Utilizza lo strumento open source Checkov per eseguire analisi statiche sul tuo ambiente IaC. Checkov aiuta a identificare le configurazioni errate del cloud scansionando il codice dell'infrastruttura in Kubernetes, Terraform o. CloudFormation Puoi utilizzare Checkov per ottenere output in diversi formati, tra cui JSON, JUnit XML o CLI. Checkov è in grado di gestire le variabili in modo efficace creando un grafico che illustra la dipendenza di codice dinamica. Per ulteriori informazioni, consulta il repository Checkov di Bridgecrew. GitHub
- Utilizza TFlint per verificare la presenza di errori e sintassi obsoleta e agevolare l'applicazione delle best practice. Tieni presente che TFlint potrebbe non convalidare problemi specifici del provider. Per ulteriori informazioni su TFlint, consulta il repository TFlint di Terraform Linters. GitHub
- Usa Amazon Q Developer per eseguire scansioni di sicurezza. Se utilizzato in un ambiente di sviluppo integrato (IDE), Amazon Q Developer fornisce assistenza per lo sviluppo del AI-powered software. Può parlare di codice, fornire completamenti di codice in linea, generare nuovo codice, scansionare il codice per individuare eventuali vulnerabilità di sicurezza e apportare aggiornamenti e miglioramenti al codice.

Sviluppo e perfezionamento della documentazione

La documentazione è fondamentale per il successo del tuo progetto. Non solo spiega come funziona il codice, ma aiuta anche gli sviluppatori a comprendere meglio le caratteristiche e le funzionalità delle applicazioni. Lo sviluppo e il perfezionamento di documentazione di alta qualità possono rafforzare il processo di sviluppo di software, mantenere software eccellenti e favorire il trasferimento delle conoscenze tra sviluppatori.

Esistono due categorie di documentazione: la documentazione all'interno del codice e quella di supporto relativa al codice. La prima si presenta sotto forma di commenti. La seconda può consistere

in file README e documenti esterni. Non è raro che gli sviluppatori considerino la documentazione come un sovraccarico, poiché il codice stesso è facile da capire. Ciò potrebbe valere i progetti piccoli, ma la documentazione è fondamentale per i progetti su larga scala che vedono coinvolti più team.

È consigliabile che l'autore del codice scriva la documentazione poiché ne conosce bene le funzionalità. Gli sviluppatori possono avere difficoltà a gestire il sovraccarico aggiuntivo derivante dalla gestione di una documentazione di supporto a sé stante. Per superare questo ostacolo, possono aggiungere all'interno del codice commenti che possono essere estratti in modo automatico, così che tutte le versioni del codice e della documentazione siano sincronizzate.

Esistono diversi strumenti per aiutare gli sviluppatori a estrarre commenti dal codice e generare la relativa documentazione. Questa guida è considerata lo strumento preferito per i AWS CDK costrutti. TypeDoc

Perché è richiesta la documentazione del codice per AWS CDK costrutti

AWS CDK i costrutti comuni vengono creati da più team di un'organizzazione e condivisi tra diversi team per essere utilizzati. Una buona documentazione aiuta gli utenti della libreria di costrutti a integrarli facilmente e a costruire la propria infrastruttura con il minimo sforzo. Mantenere sincronizzati tutti i documenti non è semplice. Ti consigliamo di mantenere il documento all'interno del codice, che verrà estratto utilizzando la TypeDoc libreria.

Usando TypeDoc con AWS Costruisci una libreria

TypeDoc è un generatore di documenti per TypeScript. È possibile TypeDoc utilizzarlo per leggere i file TypeScript sorgente, analizzare i commenti in tali file e quindi generare un sito statico che contiene la documentazione per il codice.

Il codice seguente mostra come eseguire l'integrazione TypeDoc con la AWS Construct Library e quindi aggiungere i seguenti pacchetti nel `package.json` file in. `devDependencies`

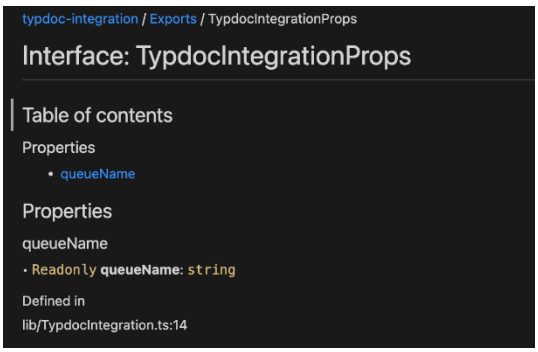
```
{

  "devDependencies": {

    "typedoc-plugin-markdown": "^3.11.7",
    "typescript": "~3.9.7"
  },

}
```

Per aggiungere `typedoc.json` nella cartella della libreria CDK, utilizza il codice seguente.

```
{
    "$schema": "https://typedoc.org/schema.json",
    "entryPoints": ["./lib"],
}
```

Per generare i file README, esegui il `npx typedoc` comando nella directory principale del progetto della libreria di AWS CDK costruzioni.

Il seguente documento di esempio è generato da. TypeDoc



Per ulteriori informazioni sulle opzioni di TypeDoc integrazione, consulta Doc Comments nella TypeDoc documentazione.

Adozione di un approccio di sviluppo basato su test

Ti consigliamo di seguire un approccio di sviluppo basato sui test (TDD) con. AWS CDK TDD è un approccio di sviluppo di software in cui si sviluppano casi di test per specificare e verificare il codice. In parole povere, per prima cosa si creano casi di test per ciascuna funzionalità e, se il test ha esito negativo, si scrive il nuovo codice per superare il test e fare in modo che sia semplice e privo di bug.

Puoi utilizzare TDD per scrivere prima il caso di test. Ciò consente di verificare l'infrastruttura con diversi vincoli di progettazione in termini di applicazione della politicy di sicurezza per le risorse e di rispetto di una convenzione di denominazione esclusiva per il progetto. L'approccio standard per testare AWS CDK le applicazioni consiste nell'utilizzare il modulo AWS CDKassertions e i framework di test più diffusi, come Jest for e/o pytest for TypeScript JavaScript Python.

Esistono due categorie di test che puoi scrivere per le tue applicazioni: AWS CDK

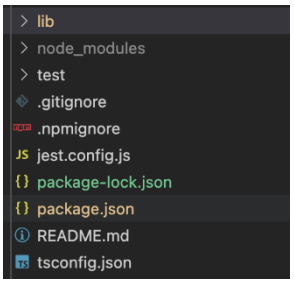
- Utilizza asserzioni dettagliate per testare un aspetto specifico del CloudFormation modello generato, ad esempio «questa risorsa ha questa proprietà con questo valore». Questi test possono

rilevare regressioni e sono utili anche quando sviluppi nuove funzionalità utilizzando TDD (scrivi prima un test, poi fallo passare scrivendo un'implementazione corretta). Fine-grained le asserzioni sono i test che scriverai di più.

- Usa i test snapshot per testare il modello sintetizzato rispetto a un CloudFormation modello di base memorizzato in precedenza. I test snapshot consentono di rifattorizzare liberamente, perché hai la certezza che il codice rifattorizzato funziona esattamente allo stesso modo dell'originale. Se le modifiche sono state intenzionali, puoi accettare una nuova base per i test futuri. Tuttavia, AWS CDK gli aggiornamenti possono anche causare modifiche ai modelli sintetizzati, quindi non puoi fare affidamento solo sulle istantanee per assicurarti che l'implementazione sia corretta.

Test unitario

Questa guida si concentra TypeScript specificamente sull'integrazione dei test unitari. Per abilitare i test, assicurati che il `package.json` file abbia le seguenti librerie: `@types/jestjest`, e `ts-jest` in`devDependencies`. Per aggiungere questi pacchetti, esegui il comando `cdk init lib --language=typescript`. Dopo aver eseguito il comando precedente, visualizzerai la struttura seguente.



Il codice seguente è un esempio di `package.json` file abilitato con la libreria Jest.

```
{
  ...
  "scripts": {
    "build": "npm run lint && tsc",
    "watch": "tsc -w",
    "test": "jest",
  },
  "devDependencies": {
    ...
    "@types/jest": "27.5.2",
    "jest": "27.5.1",
    "ts-jest": "27.1.5",
```

```
    ...
  }
}
```

Nella cartella Test, puoi scrivere il caso di test. L'esempio seguente mostra un test case per un AWS CodePipeline costrutto.

```
import { Stack } from 'aws-cdk-lib';
import { Template } from 'aws-cdk-lib/assertions';
import * as CodePipeline from 'aws-cdk-lib/aws-codepipeline';
import * as CodePipelineActions from 'aws-cdk-lib/aws-codepipeline-actions';
import { MyPipelineStack } from '../lib/my-pipeline-stack';
test('Pipeline Created with GitHub Source', () => {
  // ARRANGE
  const stack = new Stack();
  // ACT
  new MyPipelineStack(stack, 'MyTestStack');
  // ASSERT
  const template = Template.fromStack(stack);
  // Verify that the pipeline resource is created
  template.resourceCountIs('AWS::CodePipeline::Pipeline', 1);
  // Verify that the pipeline has the expected stages with GitHub source
  template.hasResourceProperties('AWS::CodePipeline::Pipeline', {
    Stages: [
      {
        Name: 'Source',
        Actions: [
          {
            Name: 'SourceAction',
            ActionTypeId: {
              Category: 'Source',
              Owner: 'ThirdParty',
              Provider: 'GitHub',
              Version: '1'
            },
            Configuration: {
              Owner: {
                'Fn::Join': [
                  '',
                  [
                    '{{resolve:secretsmanager:',
                    {
                      Ref: 'GitHubTokenSecret'
```

```
                  },
                  ':SecretString:owner}}'
                ]
              ]
            },
            Repo: {
              'Fn::Join': [
                '',
                [
                  '{{resolve:secretsmanager:',
                  {
                    Ref: 'GitHubTokenSecret'
                  },
                  ':SecretString:repo}}'
                ]
              ]
            },
            Branch: 'main',
            OAuthToken: {
              'Fn::Join': [
                '',
                [
                  '{{resolve:secretsmanager:',
                  {
                    Ref: 'GitHubTokenSecret'
                  },
                  ':SecretString:token}}'
                ]
              ]
            }
          },
          OutputArtifacts: [
            {
              Name: 'SourceOutput'
            }
          ],
          RunOrder: 1
        }
      ]
    },
    {
      Name: 'Build',
      Actions: [
```

```
          {
            Name: 'BuildAction',
            ActionTypeId: {
              Category: 'Build',
              Owner: 'AWS',
              Provider: 'CodeBuild',
              Version: '1'
            },
            InputArtifacts: [
              {
                Name: 'SourceOutput'
              }
            ],
            OutputArtifacts: [
              {
                Name: 'BuildOutput'
              }
            ],
            RunOrder: 1
          }
        ]
      }
      // Add more stage checks as needed
    ]
  });
  // Verify that a GitHub token secret is created
  template.resourceCountIs('AWS::SecretsManager::Secret', 1);
});
);
```

Per avviare un test, esegui il comando `npm run test` nel progetto. Il test restituisce i risultati seguenti.

```
PASS  test/codepipeline-module.test.ts (5.972 s)
  # Code Pipeline Created (97 ms)
Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        6.142 s, estimated 9 s
```

Per ulteriori informazioni sui casi di test, consultate Testing constructs nella AWS Cloud Development Kit (AWS CDK) Developer Guide.

Test di integrazione

I test di integrazione per AWS CDK i costrutti possono essere inclusi anche utilizzando un `integ-tests` modulo. Un test di integrazione deve essere definito come un' AWS CDK applicazione. Dovrebbe esserci una relazione uno a uno tra un test di integrazione e un' AWS CDK applicazione. Per ulteriori informazioni, visita il modulo integ-tests-alpha nell'API Reference.AWS CDK

Uso del controllo di rilasci e versioni per i costrutti

Controllo della versione per AWS CDK

AWS CDK i costrutti comuni possono essere creati da più team e condivisi all'interno di un'organizzazione per essere utilizzati. In genere, gli sviluppatori rilasciano nuove funzionalità o correzioni di bug nei loro costrutti comuni AWS CDK . Questi costrutti vengono utilizzati dalle AWS CDK applicazioni o da qualsiasi altro AWS CDK costrutto esistente come parte di una dipendenza. Per questo motivo, è fondamentale che gli sviluppatori aggiornino e rilascino i costrutti con versioni semantiche adeguate in modo indipendente. AWS CDK Le applicazioni downstream o altri AWS CDK costrutti possono aggiornare la propria dipendenza per utilizzare la versione di costruzione appena rilasciata. AWS CDK

Il controllo delle versioni semantico (Semver) è un insieme di regole, o metodo, per fornire numeri software univoci al software informatico. Le versioni sono definite come segue:

- Una versione PRINCIPALE è costituita da modifiche API incompatibili o da una modifica sostanziale.
- Una versione SECONDARIA è costituita da funzionalità aggiunte in modo retrocompatibile.
- Una versione PATCH consiste in correzioni di bug retrocompatibili..

Per ulteriori informazioni sul controllo delle versioni semantiche, vedete Semantic Versioning Specification () nella documentazione del controllo delle versioni semantiche. SemVer

Deposito e imballaggio per AWS CDK costrutti

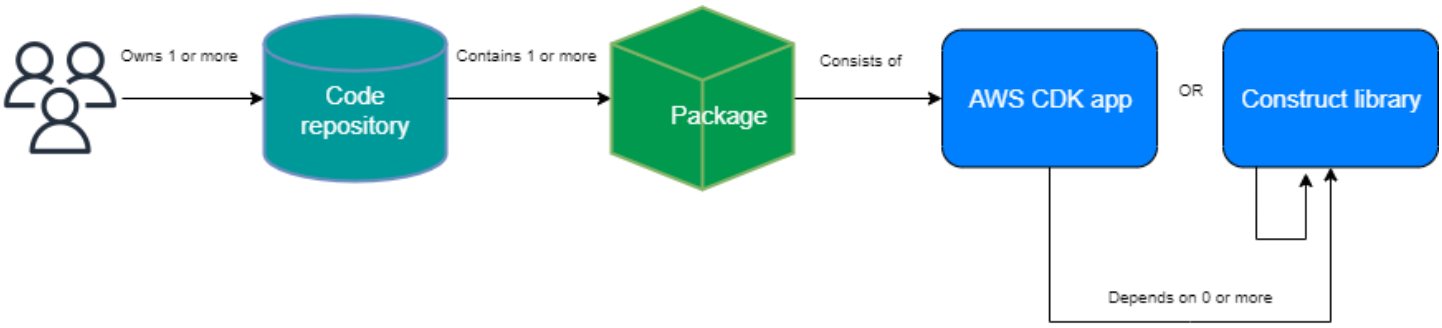
Poiché AWS CDK i costrutti sono sviluppati da team diversi e utilizzati da più AWS CDK applicazioni, è possibile utilizzare un repository separato per ogni costrutto. AWS CDK Questo può anche aiutarti ad applicare il controllo degli accessi. Ogni repository può contenere tutto il codice sorgente relativo allo stesso AWS CDK costrutto insieme a tutte le sue dipendenze. Mantenendo una singola

applicazione (ovvero un AWS CDK costrutto) in un unico repository, è possibile ridurre l'ambito di impatto delle modifiche durante la distribuzione.

AWS CDK Non solo genera CloudFormation modelli per l'implementazione dell'infrastruttura, ma raggruppa anche risorse di runtime come funzioni Lambda e immagini Docker e le distribuisce insieme all'infrastruttura. Non solo è possibile combinare il codice che definisce l'infrastruttura e il codice che implementa la logica di runtime in un unico costrutto, ma è anche una best practice. Non è necessario che questi due tipi di codice risiedano in repository o addirittura in pacchetti separati.

Per utilizzare i pacchetti oltre i confini del repository, è necessario disporre di un archivio di pacchetti privato, simile a npm o Maven Central PyPi, ma interno all'organizzazione. È inoltre necessario disporre di un processo di rilascio che crei, verifichi e pubblichi il pacchetto nel repository di pacchetti privato. Puoi creare repository privati, ad esempio PyPi server, utilizzando una macchina virtuale locale (VM) o Amazon S3. Quando progetti o crei un registro di pacchetti privato, è fondamentale considerare il rischio di interruzione del servizio a causa dell'elevata disponibilità e scalabilità. Un servizio gestito senza server ospitato nel cloud per archiviare i pacchetti può ridurre notevolmente il sovraccarico di manutenzione. Ad esempio, è possibile utilizzarlo AWS CodeArtifactper ospitare pacchetti per i linguaggi di programmazione più diffusi. È inoltre possibile CodeArtifact utilizzarlo per impostare connessioni a repository esterni e replicarle all'interno. CodeArtifact

Le dipendenze dai pacchetti nel repository dei pacchetti sono gestite dal gestore di pacchetti della tua lingua (ad esempio, npm for or applications). TypeScript JavaScript Il gestore di pacchetti si assicura che le build siano ripetibili registrando le versioni specifiche di ciascun pacchetto da cui dipende l'applicazione e quindi ti permette di aggiornare tali dipendenze in modo controllato, come illustra il diagramma seguente.

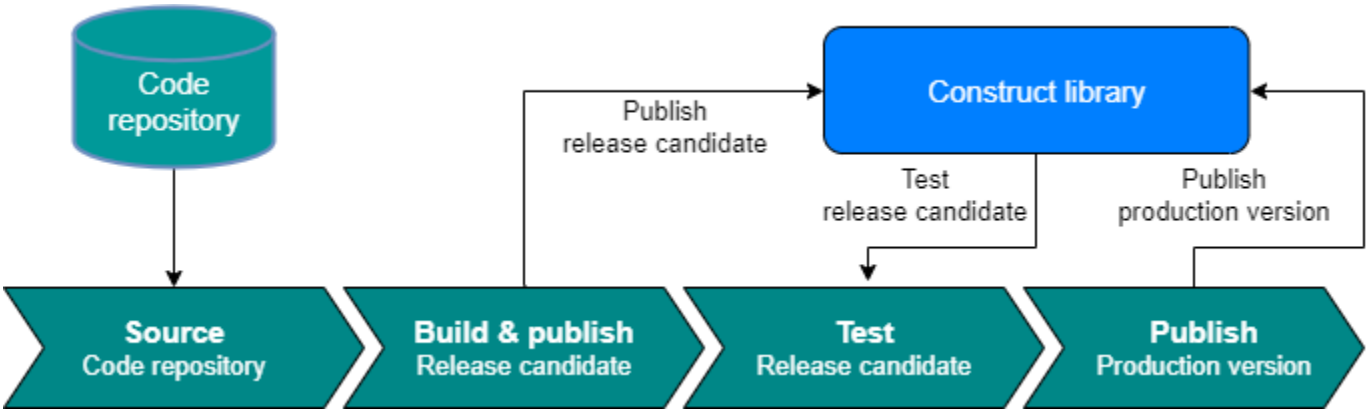


Construct Releasing per AWS CDK

Ti consigliamo di creare la tua pipeline automatizzata per creare e rilasciare nuove AWS CDK versioni di build. Se disponi di un adeguato processo di approvazione delle richieste pull, dopo aver eseguito il commit e inserito il codice di origine nel ramo principale del repository, la pipeline può

sviluppare e creare una versione release candidate. Tale versione può essere installata CodeArtifact e testata prima di rilasciare la versione pronta per la produzione. Facoltativamente, puoi testare la tua nuova AWS CDK versione di build localmente prima di unire il codice con il ramo principale. In tal modo, la pipeline esegue il rilascio della versione pronta alla produzione. Tieni presente che i costrutti e i pacchetti condivisi devono essere testati a prescindere dall'applicazione che li utilizza, come se fossero rilasciati al pubblico.

Il diagramma seguente mostra una pipeline di rilascio della versione di esempio AWS CDK .



Puoi utilizzare i seguenti comandi di esempio per creare, testare e pubblicare pacchetti npm. Innanzitutto, accedi al repository di artefatti eseguendo il comando seguente.

```
aws codeartifact login --tool npm --domain <Domain Name> --domain-owner $(aws sts get-
caller-identity --output text --query 'Account')  \
--repository <Repository Name> --region <AWS Region Name>
```

Quindi, completa questa procedura:

1. Installa i pacchetti necessari in base al file `package.json`: `npm install`
2. Crea la versione release candidate: `npm version prerelease --preid rc`
3. Crea il pacchetto npm: `npm run build`
4. Testa il pacchetto npm: `npm run test`
5. Pubblica il pacchetto npm: `npm publish`

Applicazione della gestione delle versioni della libreria

La gestione del ciclo di vita è una sfida importante quando si gestiscono basi di codice. AWS CDK Ad esempio, supponiamo di iniziare un AWS CDK progetto con la versione 1.97 e che la versione 1.169

diventi disponibile in un secondo momento. La versione 1.169 offre nuove funzionalità e correzioni di bug, ma l'infrastruttura è stata implementata utilizzando la versione precedente. Ora, con l'aumento del divario, l'aggiornamento dei costrutti diventa difficile a causa delle modifiche sostanziali che potrebbero essere introdotte nelle nuove versioni. Ciò può rappresentare un ostacolo se l'ambiente dispone di molte risorse. Lo schema introdotto in questa sezione può aiutarvi a gestire la versione della AWS CDK libreria utilizzando l'automazione. Ecco il flusso di lavoro di questo pattern:

1. Quando si avvia un nuovo prodotto CodeArtifact Service Catalog, le versioni della AWS CDK libreria e le relative dipendenze vengono archiviate nel `package.json` file.
2. Implementa una pipeline comune che tenga traccia di tutti i repository, in modo da poter applicare a questi ultimi aggiornamenti automatici se non ci sono modifiche sostanziali.
3. Una AWS CodeBuild fase verifica la presenza dell'albero delle dipendenze e cerca le ultime modifiche.
4. La pipeline crea un ramo di funzionalità e quindi esegue `cdk synth` con la nuova versione per confermare l'assenza di errori.
5. La nuova versione viene implementata nell'ambiente di test e infine esegue un test di integrazione per assicurarsi che l'implementazione sia corretta.
6. Puoi utilizzare due code Amazon Simple Queue Service (Amazon SQS) per monitorare gli stack. Gli utenti possono esaminare manualmente gli stack nella coda delle eccezioni e gestire le modifiche sostanziali. Gli elementi che superano il test di integrazione possono essere uniti e rilasciati.

Domande frequenti

Quali problemi può TypeScript risolvere?

In genere, puoi eliminare i bug nel codice scrivendo test automatici, verificando manualmente che il codice funzioni come previsto e infine chiedendo a un'altra persona di convalidare il codice. La convalida delle connessioni tra ogni parte del progetto richiede molto tempo. Per velocizzare il processo di convalida, puoi utilizzare un linguaggio a controllo tipografico, TypeScript ad esempio automatizzare la convalida del codice e fornire un feedback immediato durante lo sviluppo.

Perché dovrei usare? TypeScript

TypeScript è un linguaggio open source che semplifica il JavaScript codice, facilitandone la lettura e il debug. TypeScript fornisce inoltre strumenti JavaScript IDEs e pratiche di sviluppo altamente produttivi, come il controllo statico. Inoltre, TypeScript offre i vantaggi di ECMAScript 6 (ES6) e può aumentare la produttività. Infine, TypeScript può aiutarvi a evitare i dolorosi bug che gli sviluppatori incontrano di solito quando scrivono JavaScript per tipo controllando il codice.

Devo usare AWS CDK o CloudFormation?

Ti consigliamo di utilizzare AWS Cloud Development Kit (AWS CDK) invece di AWS CloudFormation, se la tua organizzazione ha le competenze di sviluppo necessarie per sfruttare il AWS CDK. Questo perché AWS CDK è più flessibile di CloudFormation, poiché è possibile utilizzare un linguaggio di programmazione e concetti OOP. Tieni presente che puoi utilizzare CloudFormation per creare AWS risorse in modo ordinato e prevedibile. Nel CloudFormation, le risorse vengono scritte in file di testo utilizzando il formato JSON o YAML.

Cosa succede se AWS CDK non supporta una versione appena lanciata? Servizio AWS

Puoi usare un override non elaborato o una risorsa CloudFormation personalizzata.

Quali sono i diversi linguaggi di programmazione supportati da? AWS CDK

AWS CDK è generalmente disponibile in JavaScript, Python TypeScript, Java, C# e Go (in Developer Preview).

Quanto costa? AWS CDK

Non è previsto alcun costo aggiuntivo per il AWS CDK. Paghi per le AWS risorse (come le istanze Amazon EC2 o i sistemi di bilanciamento del carico Elastic Load Balancing) che vengono create quando le AWS CDK utilizzi nello stesso modo in cui le paghi se le creassi manualmente. I prezzi sono calcolati in base all'uso effettivo. Non sono previste tariffe minime né sono richiesti impegni anticipati.

Passaggi successivi

Ti consigliamo di iniziare a creare con AWS Cloud Development Kit (AWS CDK) in TypeScript. Per ulteriori informazioni, consulta l'AWS CDK Immersion Day Workshop.

Risorse

Riferimenti

- AWS Costrutti di soluzione (AWS soluzioni)
- AWS Cloud Development Kit (AWS CDK) (GitHub)
- AWS Riferimento all'API Construct Library (documentazione diAWS CDK riferimento)
- AWS CDK Documentazione di riferimento (documentazioneAWS CDK di riferimento)
- AWS CDK Workshop di un giorno di immersione (AWS Workshop Studio)

Strumenti

- cdk-nag () GitHub
- TypeScript ESLint(documentazione) TypeScript ESLint

Guide e pattern

- AWS Modelli di Solutions Constructs (AWS documentazione)

Cronologia dei documenti

La tabella seguente descrive le modifiche significative apportate a questa guida. Per ricevere notifiche sugli aggiornamenti futuri, puoi abbonarti a un feed RSS.

Modifica	Descrizione	Data
Best practice aggiornate	Abbiamo rimosso la raccomandazione di utilizzare cfn-nag nella sezione Strumenti di sviluppo comuni. Nella sezione Definire le convenzioni di denominazione standard, abbiamo aggiunto convenzioni di denominazione standard per le costanti globali e aggiunto esempi di denominazione.	23 ottobre 2025
Codice di aggiornamento	Abbiamo aggiornato gli esempi di codice nella sezione Segui le TypeScript migliori pratiche.	16 febbraio 2024
Aggiungi sezioni	Abbiamo aggiunto le sezioni Use utility types e Integration test.	10 gennaio 2024
Aggiornamento secondario	Esempio di codice aggiornato per la creazione di un costrutto L3.	16 giugno 2023
Pubblicazione iniziale	—	8 dicembre 2022

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.