Melhores práticas para usar o AWS CDK in TypeScript para criar projetos de IaC

AWS Recomendações

AWS Recomendações: Melhores práticas para usar o AWS CDK in TypeScript para criar projetos de IaC

Table of Contents

Melhores práticas para usar o AWS CDK in TypeScript para criar projetos de IaC

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

Outubro de 2025 (histórico do documento)

Este guia fornece recomendações e melhores práticas para usar o AWS Cloud Development Kit (AWS CDK)in TypeScript para criar e implantar projetos de infraestrutura como código (IaC) em grande escala. AWS CDK É uma estrutura para definir a infraestrutura de nuvem em código e provisionar essa infraestrutura por meio dela. AWS CloudFormation Se você não tem uma estrutura de projeto bem definida, criar e gerenciar uma AWS CDK base de código para projetos de grande escala pode ser um desafio. Para lidar com esses desafios, algumas organizações usam antipadrões para projetos de grande escala, mas esses padrões podem retardar seu projeto e criar outros problemas que afetam negativamente sua organização. Por exemplo, os antipadrões podem complicar e retardar a integração de desenvolvedores, a correção de bugs e a adoção de novos recursos.

Este guia fornece uma alternativa ao uso de antipadrões e mostra como organizar seu código para escalabilidade, testes e alinhamento com as práticas recomendadas de segurança. Este guia pode ser usado para melhorar a qualidade do código para seus projetos de IaC e maximizar a agilidade de seus negócios. Este guia é destinado a arquitetos, líderes técnicos, engenheiros de infraestrutura e qualquer outra função que busque criar um AWS CDK projeto bem arquitetado para projetos de grande escala.

Objetivos

- Custos reduzidos — Você pode usar o AWS CDK para projetar seus próprios componentes reutilizáveis que atendam aos requisitos de segurança, conformidade e governança da sua organização. Você também pode compartilhar facilmente componentes em toda a sua organização a fim de iniciar rapidamente novos projetos que se alinhem às práticas recomendadas por padrão.
- Tempo de comercialização mais rápido — Aproveite os recursos familiares do AWS CDK para acelerar seu processo de desenvolvimento. Isso aumenta a reutilização para implantação e reduz os esforços de desenvolvimento.

- Maior produtividade do desenvolvedor — Os desenvolvedores podem usar linguagens de programação familiares para definir a infraestrutura. Isso ajuda os desenvolvedores a expressar e manter AWS os recursos. Isso pode levar ao aumento da eficiência e colaboração do desenvolvedor.

Práticas recomendadas

Esta seção fornece uma visão geral das seguintes práticas recomendadas:

- Organize o código para projetos de grande escala
- Desenvolva padrões reutilizáveis
- Crie ou estenda estruturas
- Siga as TypeScript melhores práticas
- Verifique se há vulnerabilidades de segurança e erros de formatação
- Desenvolva e refine a documentação
- Adote uma abordagem de desenvolvimento orientado por testes
- Use controle de versão e lançamento para estruturas
- Imponha o gerenciamento de versões da biblioteca

Organize o código para projetos de grande escala

Por que a organização do código é importante

É fundamental que AWS CDK projetos de grande escala tenham uma estrutura bem definida e de alta qualidade. À medida que um projeto cresce e seu número de recursos e estruturas compatíveis aumenta, a navegação pelo código se torna mais difícil. Essa dificuldade pode afetar a produtividade e retardar a integração do desenvolvedor.

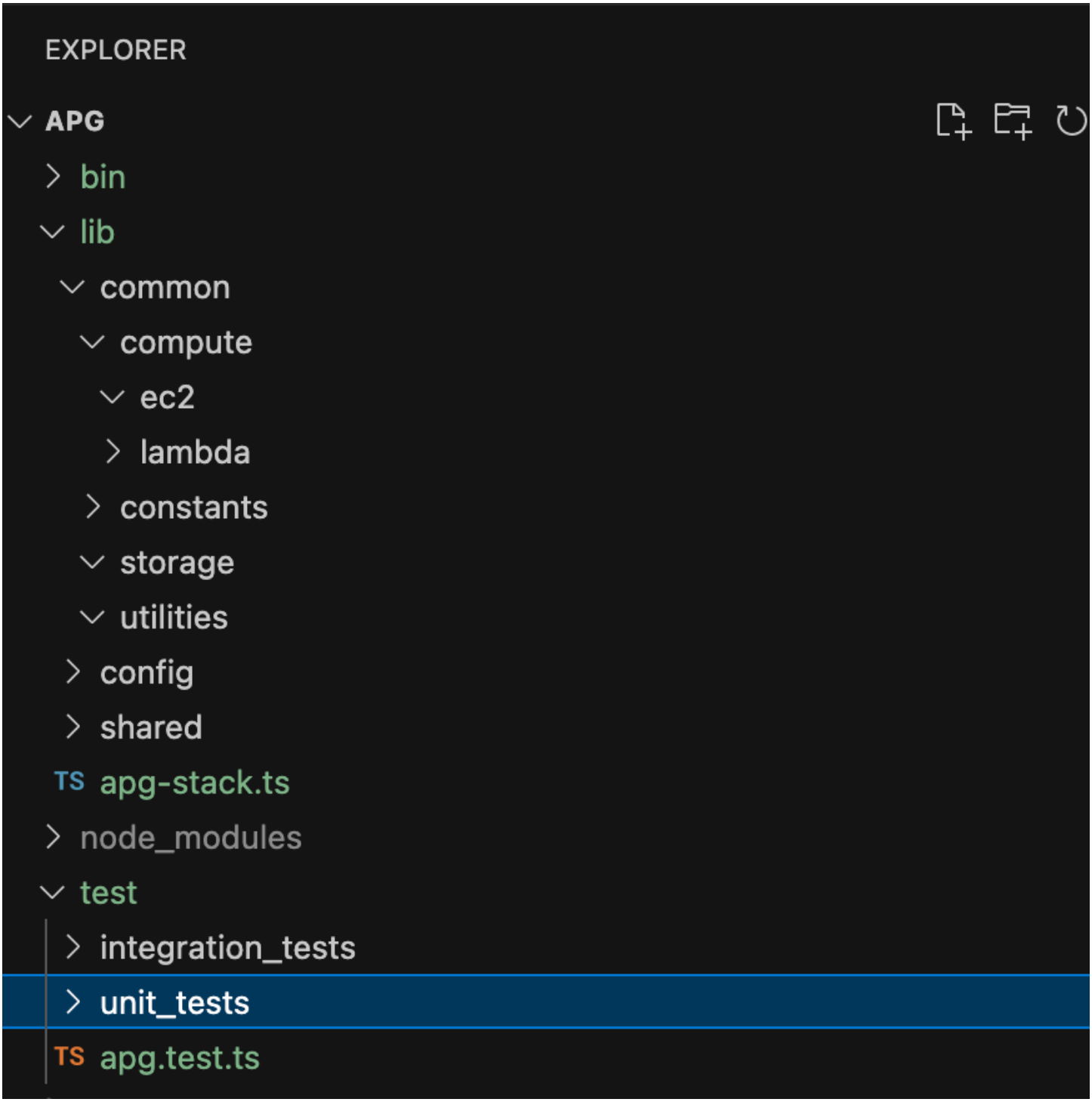
Como organizar seu código para escalar

Para alcançar um alto nível de flexibilidade e legibilidade do código, recomendamos dividir o código em partes lógicas com base na funcionalidade. Essa divisão reflete o fato de que a maioria das suas estruturas é usada em diferentes domínios de negócios. Por exemplo, seus aplicativos de front-end e back-end podem exigir uma AWS Lambda função e consumir o mesmo código-fonte. As fábricas podem criar objetos sem expor a lógica de criação ao cliente e usar uma interface comum para se referir aos objetos recém-criados. Você pode usar uma fábrica como um padrão eficaz para criar um comportamento consistente em sua base de código. Além disso, uma fábrica pode servir como uma única fonte confiável para ajudar a evitar códigos repetitivos e facilitar a solução de problemas.

Para entender melhor como as fábricas funcionam, considere o exemplo de um fabricante de automóveis. Um fabricante de automóveis não precisa ter o conhecimento e a infraestrutura necessários para fabricar pneus. Em vez disso, o fabricante do carro terceiriza essa experiência para um fabricante especializado de pneus e, em seguida, simplesmente encomenda os pneus desse fabricante conforme necessário. O mesmo princípio se aplica ao código. Por exemplo, é possível criar uma fábrica do Lambda capaz de criar funções do Lambda de alta qualidade e, em seguida, chamar a fábrica do Lambda em seu código sempre que precisar criar uma função do Lambda. Da mesma forma, é possível usar esse mesmo processo de terceirização para desacoplar sua aplicação e criar componentes modulares.

Exemplo de organização de código

O projeto de TypeScript amostra a seguir, conforme mostrado na imagem a seguir, inclui uma pasta comum na qual você pode manter todas as suas construções ou funcionalidades comuns.



Por exemplo, a pasta computar (localizada na pasta common) contém toda a lógica para diferentes estruturas de computação. Novos desenvolvedores podem adicionar facilmente novas estruturas de computação sem afetar os outros recursos. Todas as outras construções não precisarão criar novos recursos internamente. Em vez disso, essas estruturas simplesmente chamam a fábrica de estruturas comuns. Você pode organizar outras estruturas, como armazenamento, da mesma forma.

As configurações contêm dados baseados no ambiente que você deve desacoplar da pasta common em que você guarda a lógica. Recomendamos colocar seus dados de configuração comuns em uma pasta compartilhada. Também recomendamos usar a pasta utilities para servir todas as funções auxiliares e limpar scripts.

Desenvolva padrões reutilizáveis

Os padrões de design de software são soluções reutilizáveis para problemas comuns no desenvolvimento de software. Eles atuam como um guia ou paradigma para ajudar os engenheiros de software a criar produtos que sigam as práticas recomendadas. Esta seção fornece uma visão geral de dois padrões reutilizáveis que você pode usar em sua AWS CDK base de código: o padrão Abstract Factory e o padrão Chain of Responsibility. É possível usar cada padrão como um modelo e personalizá-lo para o problema de design específico no código. Para obter mais informações sobre padrões de design, consulte Padrões de design na Refactoring.Guru documentação.

Abstract Factory

O padrão Abstract Factory fornece interfaces para a criação de famílias de objetos relacionados ou dependentes sem especificar suas classes concretas. Esse padrão se aplica aos seguintes casos de uso:

- Quando o cliente é independente de como você cria e compõe os objetos no sistema
- Quando o sistema consiste em várias famílias de objetos e essas famílias são projetadas para serem usadas juntas
- Quando é necessário ter um valor em tempo de execução para construir uma dependência específica

Para obter mais informações sobre o padrão Abstract Factory, consulte Abstract Factory TypeScript na Refactoring.Guru documentação.

O exemplo de código a seguir mostra como o padrão Abstract Factory pode ser usado para criar uma fábrica de armazenamento do 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

O Chain of Responsibility é um padrão de design comportamental que permite passar uma solicitação ao longo da cadeia de manipuladores em potencial até que um deles processe a solicitação. O padrão Chain of Responsibility se aplica aos seguintes casos de uso:

- Quando vários objetos determinados em tempo de execução são candidatos para lidar com uma solicitação
- Quando você não quiser especificar manipuladores explicitamente em seu código
- Quando você deseja emitir uma solicitação para um dos vários objetos sem especificar explicitamente o receptor

Para obter mais informações sobre o padrão da Cadeia de Responsabilidade, consulte Cadeia de Responsabilidade TypeScript na Refactoring.Guru documentação.

O código a seguir mostra um exemplo de como o padrão Chain of Responsibility é usado para criar uma série de ações necessárias para concluir a tarefa.

```
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);
    }
}
```

Crie ou estenda estruturas

O que é uma estrutura

Uma construção é o alicerce básico de um AWS CDK aplicativo. Uma construção pode representar um único AWS recurso, como um bucket do Amazon Simple Storage Service (Amazon S3), ou pode ser uma abstração de nível superior que consiste em vários recursos relacionados. AWS Os componentes de uma estrutura podem incluir uma fila de trabalho com sua capacidade computacional associada ou um trabalho programado com recursos de monitoramento e um painel. AWS CDK Isso inclui uma coleção de construções chamada AWS Construct Library. A biblioteca contém construções para cada AWS service (Serviço da AWS). Você pode usar o Construct Hub para descobrir construções adicionais AWS de terceiros e da comunidade de código aberto AWS CDK .

Quais são os diferentes tipos de estruturas

Há três tipos diferentes de construções para o AWS CDK:

- Construções L1 — As construções de camada 1, ou L1, são exatamente os recursos definidos por CloudFormation —nem mais, nem menos. Você mesmo deve fornecer os recursos necessários para a configuração. Essas construções L1 são muito básicas e devem ser configuradas manualmente. As construções L1 têm um `Cfn` prefixo e correspondem diretamente às especificações. CloudFormation Novos Serviços da AWS são suportados AWS CDK assim CloudFormation que houver suporte para esses serviços. CfnBucketé um bom exemplo de uma construção L1. Essa classe representa um bucket do S3 em que você deve configurar explicitamente todas as propriedades. Recomendamos que você use apenas uma construção L1 se não conseguir encontrar a construção L2 ou L3 para ela.
- Estruturas L2: as estruturas da camada 2, ou L2, têm código boilerplate e lógica de colagem comuns. Essas construções vêm com padrões convenientes e reduzem a quantidade de conhecimento que você precisa saber sobre elas. As construções L2 usam APIs baseadas em intenção para construir seus recursos e normalmente encapsulam seus módulos L1 correspondentes. Um bom exemplo de estrutura L2 é o Bucket. Essa classe cria um bucket do S3 com propriedades e métodos padrão, como bucket.add LifeCycleRule (), que adiciona uma regra de ciclo de vida ao bucket.
- Estruturas L3: uma estrutura da camada 3, ou L3, é chamada de padrão. As construções L3 são projetadas para ajudá-lo a concluir tarefas comuns AWS, geralmente envolvendo vários tipos de recursos. Elas são ainda mais específicas e opinativas do que as estruturas L2 e servem a um caso de uso específico. Por exemplo, os aws-ecs-patterns. ApplicationLoadBalancedFargateServiceconstruct representa uma arquitetura que inclui um cluster de AWS Fargate contêineres que usa um Application Load Balancer. Outro exemplo é o aws-apigateway. LambdaRestApiconstruir. Essa estrutura representa uma API do Amazon API Gateway que é apoiada por uma função do Lambda.

À medida que os níveis de estrutura aumentam, mais suposições são feitas sobre como essas estruturas serão usadas. Isso permite que você forneça interfaces com padrões mais eficazes para casos de uso altamente específicos.

Como criar sua própria estrutura

Para definir sua própria estrutura, é necessário seguir uma abordagem específica. Isso ocorre porque todas as estruturas estendem a classe `Construct`. A classe `Construct` é o alicerce da árvore de estruturas. As estruturas são implementadas em classes que estendem a classe base `Construct`. Todas as estruturas usam três parâmetros ao serem inicializadas:

- Escopo — O pai ou proprietário de uma construção, seja uma pilha ou outra construção, que determina seu lugar na árvore de construção. Em geral, é necessário passar `this` (ou `self` em Python), que representa o objeto atual, para o escopo.
- id: um identificador que deve ser exclusivo dentro desse escopo. O identificador serve como um namespace para tudo o que está definido na construção atual e é usado para alocar identidades exclusivas, como nomes de recursos e IDs lógicos. CloudFormation
- Props — Um conjunto de propriedades que definem a configuração inicial da construção.

O exemplo a seguir mostra como definir uma estrutura.

```
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
  }
}
```

Criar ou estender uma estrutura L2

Uma construção L2 representa um “componente de nuvem” e encapsula tudo o que é CloudFormation necessário para criar o componente. Uma construção L2 pode conter um ou mais AWS recursos, e você mesmo pode personalizar a construção. A vantagem de criar ou estender uma construção L2 é que você pode reutilizar os componentes em CloudFormation pilhas sem redefinir o código. Você pode simplesmente importar a estrutura como uma classe.

Quando há uma relação “é a” com uma construção existente, você pode estender uma construção existente para adicionar recursos padrão adicionais. É uma prática recomendada reutilizar as propriedades da construção L2 existente. É possível sobrescrever propriedades modificando as propriedades diretamente no construtor.

O exemplo a seguir mostra como se alinhar às práticas recomendadas e estender uma estrutura L2 existente chamada `s3.Bucket`. A extensão estabelece propriedades padrão, como `versioned`, `publicReadAccess`, `blockPublicAccess`, para garantir que todos os objetos (neste exemplo, buckets do S3) criados a partir dessa nova estrutura tenham sempre esses valores padrão definidos.

```
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
    });
  }
}
```

Criar uma estrutura L3

A composição é a melhor escolha quando há uma relação de “tem uma” com uma composição de construção existente. Composição significa que você constrói sua própria estrutura em cima de outras estruturas existentes. Você pode criar seu próprio padrão para encapsular todos os recursos e seus valores padrão em uma única estrutura L3 de nível superior que pode ser compartilhada. A vantagem de criar suas próprias estruturas L3 (padrões) é a possibilidade de reutilizar os componentes em pilhas sem redefinir o código. Você pode simplesmente importar a estrutura como uma classe. Esses padrões são projetados para ajudar os consumidores a provisionar vários recursos com base em padrões comuns com uma quantidade limitada de conhecimento de forma concisa.

O exemplo de código a seguir cria uma AWS CDK construção chamada`ExampleConstruct`. É possível usar essa estrutura como modelo para definir seus componentes de nuvem.

```
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
  }
}
```

O exemplo a seguir mostra como importar a construção recém-criada em seu AWS CDK aplicativo ou pilha.

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

O exemplo a seguir mostra como você é possível instanciar uma instância da estrutura que foi estendida da classe base.

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

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

Para obter mais informações, consulte AWS CDK Workshop na documentação do AWS CDK Workshop.

Escotilha de emergência

Você pode usar uma saída de emergência no AWS CDK para subir um nível de abstração para poder acessar o nível inferior das construções. As escotilhas de escape são usadas para estender a construção de recursos que não estão expostos na versão atual do AWS , mas estão disponíveis em CloudFormation.

Recomenda-se usar uma escotilha de emergência nos seguintes cenários:

- Um AWS service (Serviço da AWS) recurso está disponível por meio dele CloudFormation, mas não há `Construct` construções para ele.

- Um AWS service (Serviço da AWS) recurso está disponível CloudFormation e existem `Construct` construções para o serviço, mas elas ainda não expõem o recurso. Como `Construct` as construções são desenvolvidas “à mão”, às vezes elas podem ficar aquém das construções de CloudFormation recursos.

O código de exemplo a seguir mostra um caso de uso comum para o uso de uma escotilha de emergência. Neste exemplo, a funcionalidade que ainda não foi implementada na estrutura de nível superior destina-se a adicionar `httpPutResponseHopLimit` para escalonamento automático de `LaunchConfiguration`.

```
const launchConfig = autoscaling.onDemandASG.node.findChild("LaunchConfig") as
 CfnLaunchConfiguration;
            launchConfig.metadataOptions = {
                   httpPutResponseHopLimit: autoscalingConfig.httpPutResponseHopLimit||
 2
            }
```

O exemplo de código acima mostra o seguinte fluxo de trabalho:

1. Você define o seu `AutoScalingGroup` usando uma estrutura L2. A construção L2 não suporta a atualização do`httpPutResponseHopLimit`, então você deve usar uma saída de emergência.
2. Você usa o `node.findChild()` método para localizar o `LaunchConfig` filho específico da `AutoScalingGroup` construção L2 e convertê-lo como `CfnLaunchConfiguration` recurso.
3. Agora você pode definir a propriedade `launchConfig.metadataOptions` na `CfnLaunchConfiguration` L1.

Recursos personalizados

Você pode usar recursos personalizados para escrever uma lógica de provisionamento personalizada em modelos que são CloudFormation executados sempre que você cria, atualiza (se você alterou o recurso personalizado) ou exclui pilhas. Por exemplo, você pode usar um recurso personalizado se quiser incluir recursos que não estão disponíveis no AWS CDK. Dessa forma, você ainda pode gerenciar todos os recursos relacionados em uma única pilha.

A criação de um recurso personalizado envolve escrever uma função do Lambda que responde aos eventos de ciclo de vida CREATE, UPDATE e DELETE de um recurso. Se seu recurso personalizado precisar fazer apenas uma única chamada de API, considere usar a

AwsCustomResourceconstrução. Isso possibilita realizar chamadas arbitrárias do SDK durante uma CloudFormation implantação. Caso contrário, sugerimos escrever sua própria função do Lambda para realizar o trabalho que precisa ser feito.

Para obter mais informações sobre recursos personalizados, consulte Recursos personalizados na CloudFormation documentação. Para ver um exemplo de como usar um recurso personalizado, consulte o repositório de recursos personalizados em GitHub.

O exemplo a seguir mostra como criar uma classe de recurso personalizada para iniciar uma função Lambda e CloudFormation enviar um sinal de sucesso ou falha.

```
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();
    }
}
```

O exemplo a seguir mostra a lógica principal do recurso personalizado.

```
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)
```

O exemplo a seguir mostra como a AWS CDK pilha chama o recurso personalizado.

```
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();
```

Siga as TypeScript melhores práticas

TypeScript é uma linguagem que amplia os recursos do JavaScript. É uma linguagem fortemente digitada e orientada a objetos. Você pode usar TypeScript para especificar os tipos de dados que estão sendo transmitidos em seu código e tem a capacidade de relatar erros quando os tipos não coincidem. Esta seção fornece uma visão geral das TypeScript melhores práticas.

Descreva seus dados

Você pode usar TypeScript para descrever a forma dos objetos e funções em seu código. Usar o tipo `any` é equivalente a optar por não verificar o tipo de uma variável. Recomendamos evitar usar `any`em seu código. Aqui está um exemplo.

```
type Result = "success" | "failure"
```

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

Use enumerações

É possível usar enumerações para definir um conjunto de constantes nomeadas e definir padrões que podem ser reutilizados em sua base de código. Recomendamos exportar suas enumerações uma vez em nível global e depois permitir que outras classes importem e usem as enumerações. Suponha que você queira criar um conjunto de ações possíveis para capturar os eventos em sua base de código. TypeScript fornece enumerações numéricas e baseadas em strings. O exemplo a seguir usa uma enumeração.

```
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)
```

Use interfaces do

Uma interface é um contrato para a classe. Se você criar um contrato, os usuários deverão cumpri-lo. No exemplo a seguir, uma interface é usada para padronizar `props` e garantir que os chamadores forneçam o parâmetro esperado ao usar essa 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
})
```

Algumas propriedades só podem ser modificadas quando um objeto é criado pela primeira vez. É possível especificar isso colocando `readonly` antes do nome da propriedade, conforme mostrado no exemplo a seguir.

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

Estenda interfaces

A extensão de interfaces reduz a duplicação, pois não é necessário copiar as propriedades entre as interfaces. Além disso, o leitor do seu código pode entender facilmente os relacionamentos em sua aplicação.

```
 interface BaseInterface{
```

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

Evite interfaces vazias

Recomendamos evitar interfaces vazias devido aos riscos potenciais criados por elas. No exemplo a seguir, há uma interface vazia chamada`BucketProps`. Os objetos `myS3Bucket1` e `myS3Bucket2` são ambos válidos, mas seguem padrões diferentes porque a interface não impõe nenhum contrato. O código a seguir compilará e imprimirá as propriedades, mas isso introduz inconsistências na aplicação.

```
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",
});
```

Use fábricas

Em um padrão Abstract Factory, uma interface é responsável por criar uma fábrica de objetos relacionados sem especificar explicitamente suas classes. Por exemplo, você pode criar uma fábrica do Lambda para criar funções do Lambda. Em vez de criar uma nova função Lambda em sua

construção, você está delegando o processo de criação à fábrica. Para obter mais informações sobre esse padrão de design, consulte Abstract Factory TypeScript na Refactoring.Guru documentação.

Use desestruturação em propriedades

A desestruturação, introduzida no ECMAScript 6 (ES6), é um JavaScript recurso que permite extrair vários dados de uma matriz ou objeto e atribuí-los às suas próprias variáveis.

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

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

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

Defina convenções de nomenclatura padrão

A aplicação de uma convenção de nomenclatura mantém a base de código consistente e reduz a sobrecarga representada por pensar em como nomear uma variável. Recomendamos o seguinte:

- Use camelCase para nomes de variáveis e funções.
- Use UPPER_CASE para constantes globais para indicar claramente valores imutáveis de tempo de compilação.
- Use PascalCase para nomes de classes e nomes de interface.
- Use camelCase para membros da interface.
- Use PascalCase para nomes de tipos e nomes de enumeração.
- Nomeie arquivos com camelCase (por exemplo, `ebsVolumes.tsx` ou `storage.ts`)

Veja a seguir exemplos dessas convenções de nomenclatura recomendadas:

```
// 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 { }
```

Não use a palavra-chave var

A `let` instrução é usada para declarar uma variável local em TypeScript. É semelhante à `var` palavra-chave, mas tem algumas restrições no escopo em comparação com a `var` palavra-chave. Uma variável declarada em um bloco com `let` só está disponível para uso dentro desse bloco. A `var` palavra-chave não pode ter escopo de bloco, o que significa que ela pode ser acessada fora de um bloco específico (representado por`{}`), mas não fora da função em que está definida. Você pode redeclarar e atualizar `var` variáveis. É uma prática recomendada evitar o uso da `var` palavra-chave.

Considere usar ESLint e Prettier

O ESLint analisa estaticamente seu código para encontrar problemas rapidamente. O ESLint pode ser usado para criar uma série de afirmações (chamadas regras de lint) que definem como seu código deve parecer ou se comportar. O ESLint também tem sugestões de correção automática para ajudar você a melhorar seu código. Finalmente, você pode usar o ESLint para carregar regras de lint de plug-ins compartilhados.

O Prettier é um formatador de código conhecido compatível com toda uma variedade de linguagens de programação. O Prettier pode ser usado para definir seu estilo de código a fim de evitar sua formatação manual. Após a instalação, você pode atualizar seu arquivo `package.json` e executar os comandos `npm run format` e `npm run lint`.

O exemplo a seguir mostra como habilitar o ESLint e o formatador Prettier para seu projeto. 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)'"
}
```

Use modificadores de acesso

O modificador privado em TypeScript limita a visibilidade somente para a mesma classe. Ao adicionar o modificador privado a uma propriedade ou método, é possível acessar essa propriedade ou método dentro da mesma classe.

O modificador público permite que propriedades e métodos de classe sejam acessíveis de todos os locais. Se você não especificar nenhum modificador de acesso para propriedades e métodos, eles usarão o modificador público por padrão.

O modificador protegido permite que propriedades e métodos de uma classe sejam acessíveis dentro da mesma classe e dentro de subclasses. Use o modificador protegido quando você espera criar subclasses em seu AWS CDK aplicativo.

Use tipos de utilitários

Os tipos de utilitários em TypeScript são funções de tipo predefinidas que realizam transformações e operações em tipos existentes. Isso ajuda você a criar novos tipos com base nos tipos existentes. Por exemplo, você pode alterar ou extrair propriedades, tornar as propriedades opcionais ou obrigatórias ou criar versões imutáveis de tipos. Ao usar tipos de utilitários, você pode definir tipos mais precisos e capturar possíveis erros em tempo de compilação.

<Tipo parcial >

`Partial`marca todos os membros de um tipo de entrada `Type` como opcionais. Esse utilitário retorna um tipo que representa todos os subconjuntos de um determinado tipo. Veja a seguir um exemplo de `Partial`.

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

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

<Tipo necessário >

`Required`faz o oposto de`Partial`. Isso torna todos os membros de um tipo de entrada `Type` não opcionais (em outras palavras, obrigatórios). Veja a seguir um exemplo de `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
};
```

Verifique se há vulnerabilidades de segurança e erros de formatação

A infraestrutura como código (IaC) e a automação se tornaram essenciais para as empresas. Com a IaC sendo tão robusta, você tem uma grande responsabilidade de gerenciar os riscos de segurança. Os riscos comuns de segurança da IaC podem incluir:

- Over-permissive AWS Identity and Access Management Privilégios (IAM)
- Escopo dos grupos de segurança
- Recursos não criptografados
- Logs de acesso não ativados

Abordagens e ferramentas de segurança

Recomendamos implementar as seguintes abordagens de segurança:

- Detecção de vulnerabilidades no desenvolvimento: a correção de vulnerabilidades na produção é cara e demorada devido à complexidade de desenvolver e distribuir patches de software. Além disso, as vulnerabilidades na produção causam o risco de exploração. Recomendamos usar a varredura de código em seus recursos de IaC para que as vulnerabilidades possam ser detectadas e corrigidas antes da liberação para produção.
- Conformidade e remediação automática — AWS Config oferece regras AWS gerenciadas. Essas regras ajudam a impor a conformidade e permitem que você tente a correção automática usando a automação.AWS Systems Manager Você também pode criar e associar documentos de automação personalizados usando AWS Config regras.

Ferramentas de desenvolvimento comuns

As ferramentas abordadas nesta seção ajudam você a estender suas funcionalidades integradas com suas próprias regras personalizadas. Recomendamos que você alinhe suas regras personalizadas com os padrões da sua organização. Alguns fatores comuns a considerar são:

- Use cdk-nag para validar se as construções dentro de um determinado escopo estão em conformidade com um conjunto definido de regras. Você também pode usar o cdk-nag para suprimir regras e relatórios de conformidade. A ferramenta cdk-nag valida construções estendendo aspectos no. AWS CDK Para obter mais informações, consulte Gerenciar a segurança e a conformidade de aplicativos com o AWS Cloud Development Kit (AWS CDK) e cdk-nag no blog. AWS DevOps
- Use a ferramenta de código aberto Checkov para realizar análises estáticas em seu ambiente de IaC. O Checkov ajuda a identificar configurações incorretas na nuvem escaneando seu código de infraestrutura em Kubernetes, Terraform ou. CloudFormation Você pode usar o Checkov para obter saídas em diferentes formatos, incluindo JSON, JUnit XML ou CLI. O Checkov pode lidar com variáveis de forma eficaz criando um gráfico que mostra a dependência dinâmica do código. Para obter mais informações, consulte o repositório GitHub Checkov da Bridgecrew.
- Use o TFlint para verificar erros e sintaxe obsoleta e para obter ajuda para aplicar as práticas recomendadas. Observe que o TFlint pode não validar problemas específicos de provedor. Para obter mais informações sobre o TFlint, consulte o repositório GitHub TFlint do Terraform Linters.
- Use o Amazon Q Developer para realizar verificações de segurança. Quando usado em um ambiente de desenvolvimento integrado (IDE), o Amazon Q Developer fornece assistência ao desenvolvimento de AI-powered software. Ele pode conversar sobre código, fornecer

preenchimentos de código em linha, gerar novos códigos, escanear seu código em busca de vulnerabilidades de segurança e fazer atualizações e melhorias no código.

Desenvolva e refine a documentação

A documentação é fundamental para o sucesso do seu projeto. A documentação não apenas explica como o código funciona, mas também ajuda os desenvolvedores a entender melhor os recursos e a funcionalidade das aplicações. Desenvolver e refinar documentação de alta qualidade pode fortalecer o processo de desenvolvimento de software, manter um software de alta qualidade e ajudar na transferência de conhecimento entre desenvolvedores.

Há duas categorias de documentação: documentação dentro do código e documentação de apoio sobre o código. A documentação dentro do código é fornecida na forma de comentários. A documentação de apoio sobre o código pode ser feita por arquivos README e documentos externos. Não é incomum que os desenvolvedores pensem na documentação como uma sobrecarga, pois o código em si é fácil de entender. Isso pode ser verdade para projetos pequenos, mas a documentação é crucial para projetos de grande escala em que várias equipes estão envolvidas.

É uma prática recomendada para o autor do código escrever a documentação, pois ele tem um bom entendimento de suas funcionalidades. Os desenvolvedores podem enfrentar dificuldades com a sobrecarga adicional de manter uma documentação de suporte separada. Para superar esse desafio, os desenvolvedores podem adicionar os comentários no código e esses comentários podem ser extraídos automaticamente para que todas as versões do código e da documentação sejam sincronizadas.

Existem diversas ferramentas diferentes para ajudar os desenvolvedores a extrair comentários do código e gerar a documentação para ele. Este guia se concentra em TypeDoc ser a ferramenta preferida para AWS CDK construções.

Por que a documentação do código é necessária para AWS CDK constructos

AWS CDK construções comuns são criadas por várias equipes em uma organização e compartilhadas entre diferentes equipes para consumo. Uma boa documentação ajuda os consumidores da biblioteca de estruturas a integrar facilmente estruturas e construir sua infraestrutura com o mínimo esforço. Manter todos os documentos sincronizados é uma tarefa árdua.

Recomendamos que você mantenha o documento dentro do código, que será extraído usando a TypeDoc biblioteca.

Usando TypeDoc com o AWS Construir biblioteca

TypeDoc é um gerador de documentos para TypeScript. Você pode usar TypeDoc para ler seus arquivos de TypeScript origem, analisar os comentários nesses arquivos e, em seguida, gerar um site estático que contém documentação para seu código.

O código a seguir mostra como fazer a integração TypeDoc com a AWS Construct Library e, em seguida, adicionar os seguintes pacotes ao seu `package.json` arquivo em`devDependencies`.

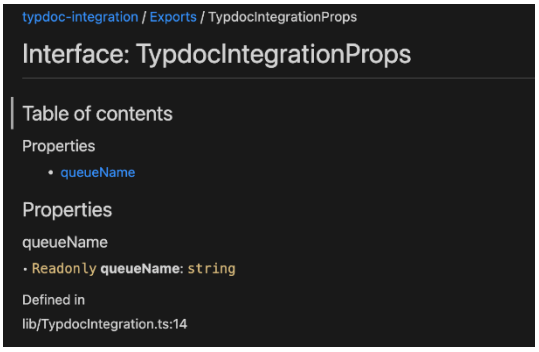
```
{

  "devDependencies": {

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

}
```

Para adicionar `typedoc.json` na pasta da biblioteca do CDK, use o código a seguir.

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

Para gerar os arquivos README, execute o `npx typedoc` comando no diretório raiz do projeto da biblioteca de AWS CDK construção.

O documento de amostra a seguir é gerado por TypeDoc.



Para obter mais informações sobre as opções de TypeDoc integração, consulte Comentários do documento na TypeDoc documentação.

Adote uma abordagem de desenvolvimento orientado por testes

Recomendamos que você siga uma abordagem de desenvolvimento orientado a testes (TDD) com o. AWS CDK O TDD é uma abordagem de desenvolvimento de software em que você desenvolve casos de teste para especificar e validar seu código. Em termos simples, primeiro você cria casos de teste para cada funcionalidade e, se o teste falhar, então você escreverá um novo código para passar no teste e tornar o código simples e livre de erros.

É possível usar o TDD para escrever o caso de teste primeiro. Isso ajuda a validar a infraestrutura com diferentes restrições de design em termos de aplicar a política de segurança para os recursos e seguir uma convenção de nomenclatura exclusiva para o projeto. A abordagem padrão para testar AWS CDK aplicativos é usar o módulo de AWS CDK asserções e estruturas de teste populares, como Jest para JavaScript e/ou pytest para TypeScript Python.

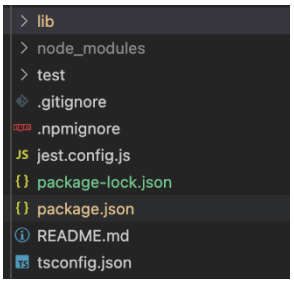
Há duas categorias de testes que você pode escrever para seus AWS CDK aplicativos:

- Use afirmações refinadas para testar um aspecto específico do CloudFormation modelo gerado, como “esse recurso tem essa propriedade com esse valor”. Esses testes podem detectar regressões e também são úteis quando você está desenvolvendo novos recursos usando o TDD (escreva um teste primeiro e depois faça com que ele passe escrevendo uma implementação correta). Fine-grained afirmações são os testes que você mais escreverá.
- Use testes instantâneos para testar o modelo sintetizado em relação a um CloudFormation modelo de linha de base armazenado anteriormente. Os testes de snapshots possibilitam refatorar livremente, pois você pode ter certeza de que o código refatorado funciona exatamente da mesma forma que o original. Se as alterações foram intencionais, é possível aceitar uma nova linha de base para futuros testes. No entanto, AWS CDK as atualizações também podem causar alterações

nos modelos sintetizados, portanto, você não pode confiar apenas em instantâneos para garantir que sua implementação esteja correta.

Teste unitário

Este guia se concentra TypeScript especificamente na integração de testes unitários. Para habilitar o teste, certifique-se de que seu `package.json` arquivo tenha as seguintes bibliotecas: `@types/jestjest`, e `ts-jest` in`devDependencies`. Para adicionar esses pacotes, execute o comando `cdk init lib --language=typescript`. É possível ver a estrutura a seguir após executar o comando anterior.



O código a seguir é um exemplo de um `package.json` arquivo habilitado com a biblioteca 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",
    ...
  }
}
```

Escreva o caso de teste na pasta Test. O exemplo a seguir mostra um caso de teste para uma AWS CodePipeline construção.

```
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);
});
);
```

Para executar um teste, execute o comando `npm run test` no projeto. A consulta retorna os resultados a seguir.

```
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
```

Para obter mais informações sobre casos de teste, consulte Construções de teste no Guia do AWS Cloud Development Kit (AWS CDK) desenvolvedor.

Teste de integração

Os testes de integração para AWS CDK construções também podem ser incluídos usando um `integ-tests` módulo. Um teste de integração deve ser definido como um AWS CDK aplicativo. Deve haver uma relação individual entre um teste de integração e um AWS CDK aplicativo. Para obter mais informações, visite o módulo integ-tests-alpha na Referência da API.AWS CDK

Use controle de versão e lançamento para estruturas

Controle de versão para o AWS CDK

AWS CDK construções comuns podem ser criadas por várias equipes e compartilhadas em uma organização para consumo. Normalmente, os desenvolvedores lançam novos recursos ou correções de erros em suas AWS CDK construções comuns. Essas construções são usadas por AWS CDK aplicativos ou quaisquer outras AWS CDK construções existentes como parte de uma dependência. Por esse motivo, é crucial que os desenvolvedores atualizem e liberem suas estruturas com versões semânticas adequadas de forma independente. AWS CDK Aplicativos downstream ou outras AWS CDK construções podem atualizar sua dependência para usar a versão de construção recém-lançada AWS CDK .

Versionamento semântico (Semver) é um conjunto de regras, ou método, para fornecer números de software exclusivos ao software de computador. As versões são definidas da seguinte forma:

- Uma versão PRINCIPAL consiste em alterações de API incompatíveis ou uma alteração significativa.
- Uma versão SECUNDÁRIA consiste em funcionalidades adicionadas de maneira compatível com versões anteriores.
- Uma versão PATCH consiste em correções de bugs compatíveis com versões anteriores.

Para obter mais informações sobre controle de versão semântica, consulte Especificação de versão semântica (SemVer) na documentação de controle de versão semântica.

Repositório e embalagem para AWS CDK constructos

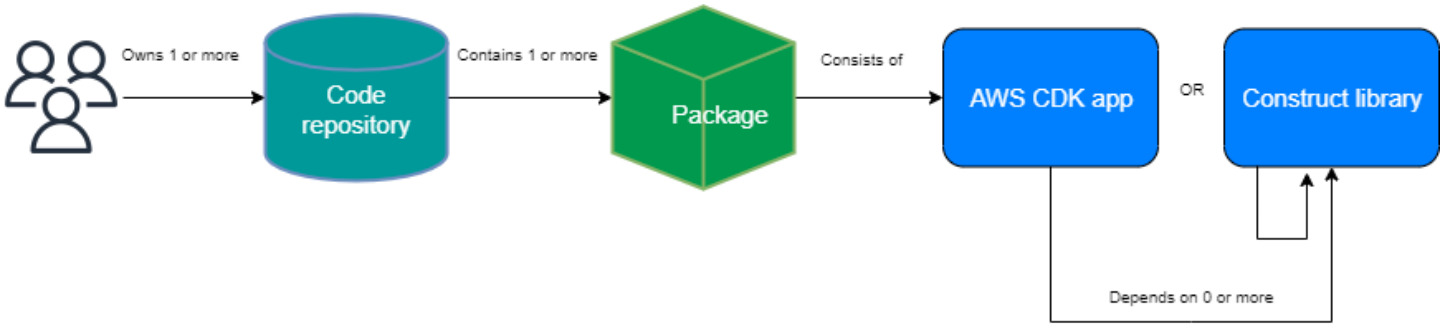
Como as AWS CDK construções são desenvolvidas por equipes diferentes e usadas por vários AWS CDK aplicativos, você pode usar um repositório separado para cada AWS CDK construção. Isso também pode ajudar na aplicação do controle de acesso. Cada repositório pode conter todo o código-fonte relacionado à mesma AWS CDK construção junto com todas as suas dependências. Ao manter um único aplicativo (ou seja, uma AWS CDK construção) em um único repositório, você pode diminuir o escopo do impacto das alterações durante a implantação.

Ele AWS CDK não apenas gera CloudFormation modelos para a implantação da infraestrutura, mas também agrupa ativos de tempo de execução, como funções Lambda e imagens do Docker, e os implanta junto com sua infraestrutura. Além de ser possível combinar o código que define

sua infraestrutura e o código que implementa sua lógica de tempo de execução em uma única construção, é uma prática recomendada. Esses dois tipos de código não precisam estar em repositórios separados ou mesmo em pacotes separados.

Para consumir pacotes além dos limites do repositório, você deve ter um repositório de pacotes privado, semelhante ao npm ou ao Maven Central PyPi, mas interno à sua organização. Também é necessário ter um processo de lançamento que compile, teste e publique o pacote no repositório de pacotes privado. Você pode criar repositórios privados, como PyPi servidores, usando uma máquina virtual (VM) local ou o Amazon S3. Ao projetar ou criar um registro de pacotes privado, é fundamental considerar o risco de interrupção do serviço devido a alta disponibilidade e escalabilidade. Um serviço gerenciado sem servidor hospedado na nuvem para armazenar pacotes pode diminuir consideravelmente a sobrecarga de manutenção. Por exemplo, você pode usar AWS CodeArtifactpara hospedar pacotes para as linguagens de programação mais populares. Você também pode usar CodeArtifact para definir conexões de repositórios externos e replicá-las internamente. CodeArtifact

As dependências dos pacotes no repositório de pacotes são gerenciadas pelo gerenciador de pacotes da sua linguagem (por exemplo, npm for TypeScript or JavaScript applications). Seu gerenciador de pacotes garante que as compilações sejam repetíveis gravando as versões específicas de cada pacote do qual a aplicação depende e, em seguida, permite que você atualize essas dependências de maneira controlada, conforme mostrado no diagrama a seguir.

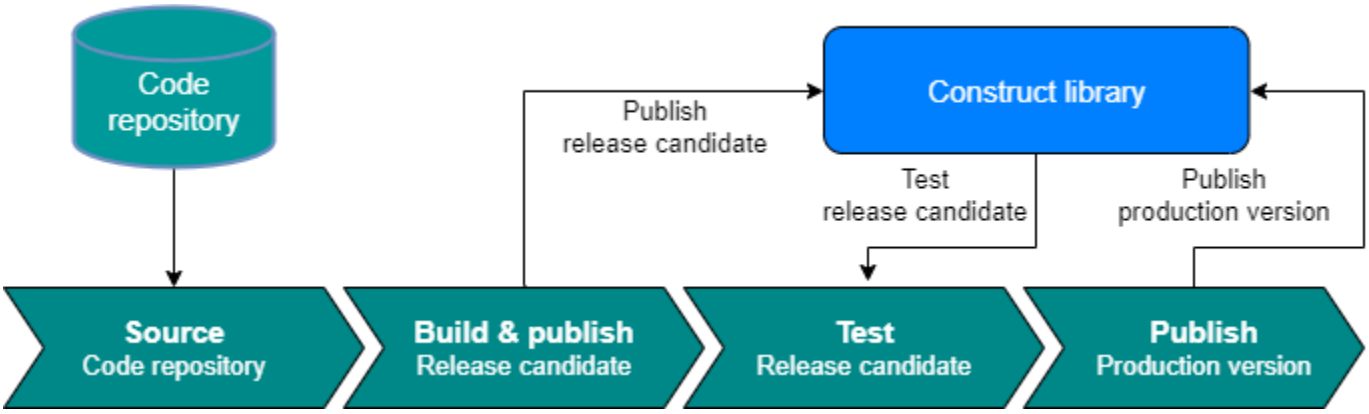


Construa a liberação para o AWS CDK

Recomendamos que você crie seu próprio pipeline automatizado para criar e lançar novas versões de AWS CDK construção. Se você implementar um processo adequado de aprovação por solicitações pull, depois de confirmar e enviar seu código-fonte para a ramificação principal do repositório, o pipeline poderá criar e construir uma versão candidata a lançamento. Essa versão pode ser enviada CodeArtifact e testada antes do lançamento da versão pronta para produção. Opcionalmente, você pode testar sua nova versão de AWS CDK construção localmente antes de

mesclar o código com a ramificação principal. Isso faz com que o pipeline lance a versão pronta para produção. Leve em consideração que estruturas e pacotes compartilhados devem ser testados independentemente da aplicação consumidora, como se estivessem sendo lançados para o público.

O diagrama a seguir mostra um exemplo de pipeline de lançamento de AWS CDK versão.



É possível usar os seguintes exemplos de comandos para compilar, testar e publicar pacotes npm. Primeiro, faça login no repositório de artefatos executando o comando a seguir.

```
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>
```

Depois, execute as etapas a seguir:

1. Instale os pacotes necessários com base no arquivo `package.json`: `npm install`
2. Crie a versão candidata ao lançamento: `npm version prerelease --preid rc`
3. Compile o pacote npm:`npm run build`
4. Teste o pacote npm:`npm run test`
5. Publique o pacote npm:`npm publish`

Imponha o gerenciamento de versões da biblioteca

O gerenciamento do ciclo de vida é um desafio significativo quando você mantém bases de AWS CDK código. Por exemplo, suponha que você inicie um AWS CDK projeto com a versão 1.97 e, em seguida, a versão 1.169 fique disponível posteriormente. A versão 1.169 oferece novos recursos e correções de erros, mas você implantou sua infraestrutura usando a versão antiga. Agora, atualizar as estruturas se torna um desafio à medida que essa lacuna aumenta devido às alterações

significativas que podem ser introduzidas em novas versões. Isso poderá ser um desafio se houver muitos recursos em seu ambiente. O padrão apresentado nesta seção pode ajudá-lo a gerenciar a versão da sua AWS CDK biblioteca usando a automação. Aqui está o fluxo de trabalho desse padrão:

1. Quando você inicia um novo produto do CodeArtifact Service Catalog, as versões da AWS CDK biblioteca e suas dependências são armazenadas no `package.json` arquivo.
2. Você implanta um pipeline comum que monitora todos os repositórios para que você possa aplicar atualizações automáticas a eles se não houver alterações significativas.
3. Um AWS CodeBuild estágio verifica a árvore de dependências e procura as alterações significativas.
4. O pipeline cria uma ramificação de recursos e, em seguida, executa `cdk synth` com a nova versão para confirmar que não há erros.
5. A nova versão é implantada no ambiente de teste e, finalmente, executa um teste de integração para garantir que a implantação esteja íntegra.
6. É possível usar duas filas do Amazon Simple Queue Service (Amazon SQS) para acompanhar as pilhas. Os usuários podem revisar as pilhas manualmente na fila de exceções e resolver as alterações mais importantes. Os itens que forem aprovados no teste de integração terão permissão para ser mesclados e lançados.

Perguntas frequentes

Quais problemas podem ser TypeScript resolvidos?

Normalmente, você pode eliminar bugs em seu código escrevendo testes automatizados, verificando manualmente se o código funciona conforme o esperado e, finalmente, fazendo com que outra pessoa valide seu código. Validar as conexões entre cada parte de um projeto é demorado. Para acelerar o processo de validação, você pode usar uma linguagem de verificação de tipo, como TypeScript para automatizar a validação do código e fornecer feedback instantâneo durante o desenvolvimento.

Por que eu deveria usar TypeScript?

TypeScript é uma linguagem de código aberto que simplifica o JavaScript código, facilitando a leitura e a depuração. TypeScript também fornece ferramentas JavaScript IDEs e práticas de desenvolvimento altamente produtivas, como verificação estática. Além disso, TypeScript oferece os benefícios do ECMAScript 6 (ES6) e pode aumentar sua produtividade. Por fim, TypeScript pode ajudá-lo a evitar bugs dolorosos que os desenvolvedores geralmente encontram ao escrever JavaScript , verificando o tipo do código.

Devo usar o AWS CDK ou CloudFormation?

Recomendamos que você use o AWS Cloud Development Kit (AWS CDK) em vez de AWS CloudFormation, se sua organização tiver a experiência em desenvolvimento para tirar proveito do AWS CDK. Isso ocorre porque AWS CDK é mais flexível do que CloudFormation, já que você pode usar uma linguagem de programação e conceitos de OOP. Lembre-se de que você pode usar CloudFormation para criar AWS recursos de maneira ordenada e previsível. Em CloudFormation, os recursos são escritos em arquivos de texto usando o formato JSON ou YAML.

E se o AWS CDK não suportar um recém-lançado AWS service (Serviço da AWS)?

Você pode usar uma substituição bruta ou um recurso CloudFormation personalizado.

Quais são as diferentes linguagens de programação suportadas pelo AWS CDK?

AWS CDK geralmente está disponível em JavaScript, Python TypeScript, Java, C# e Go (no Developer Preview).

Quanto AWS CDK custa?

Não há cobrança adicional para AWS CDK o. Você paga pelos AWS recursos (como instâncias do Amazon EC2 ou balanceadores de carga do Elastic Load Balancing) que são criados quando você os AWS CDK usa da mesma forma como se os tivesse criado manualmente. Você só paga pelo que usa à medida que usa. Não há taxas mínimas nem compromissos antecipados.

Próximas etapas

Recomendamos que você comece a construir com o AWS Cloud Development Kit (AWS CDK) in TypeScript. Para obter mais informações, consulte o Workshop do Dia AWS CDK de Imersão.

Recursos

Referências

- AWS Construções de AWS soluções (soluções)
- AWS Cloud Development Kit (AWS CDK) (GitHub)
- AWS Referência da API Construct Library (documentação de AWS CDK referência)
- AWS CDK Documentação de AWS CDK referência (documentação de referência)
- AWS CDK Workshop do Dia de Imersão (AWS Workshop Studio)

Ferramentas

- cdk-bag () GitHub
- TypeScript ESLint(TypeScript ESLint documentação)

Guias e padrões

- AWS Soluções e constrói padrões (AWS documentação)

Histórico do documento

A tabela a seguir descreve alterações significativas feitas neste guia. Se desejar receber notificações sobre futuras atualizações, inscreva-se em um feed RSS.

Alteração	Descrição	Data
Práticas recomendadas atualizadas	Removemos a recomendação de usar cfn-nag na seção Ferramentas comuns de desenvolvimento. Na seção Definir convenções de nomenclatura padrão, adicionamos convenções de nomenclatura padrão para constantes globais e adicionamos exemplos de nomenclatura.	23 de outubro de 2025
Código de atualização	Atualizamos os exemplos de código na seção Siga as TypeScript melhores práticas.	16 de fevereiro de 2024
Adicionar seções	Adicionamos as seções Usar tipos de utilitários e Teste de integração.	10 de janeiro de 2024
Atualização secundária	Exemplo de código atualizado para criar uma estrutura L3.	16 de junho de 2023
Publicação inicial	—	8 de dezembro de 2022

As traduções são geradas por tradução automática. Em caso de conflito entre o conteúdo da tradução e da versão original em inglês, a versão em inglês prevalecerá.