Mejores prácticas para usar la información AWS CDK TypeScript para crear proyectos de IaC

AWS Orientación prescriptiva

AWS Orientación prescriptiva: Mejores prácticas para usar la información AWS CDK TypeScript para crear proyectos de IaC

Table of Contents

Mejores prácticas para usar la información AWS CDK TypeScript para crear proyectos de IaC

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

Octubre de 2025 (historial del documento)

En esta guía se proporcionan recomendaciones y prácticas recomendadas para utilizarla AWS Cloud Development Kit (AWS CDK)en TypeScript la creación e implementación de proyectos de infraestructura como código (IaC) a gran escala. AWS CDK Se trata de un marco para definir la infraestructura de nube en el código y aprovisionarla mediante ella. AWS CloudFormation Si no tiene una estructura de proyecto bien definida, crear y administrar una AWS CDK base de código para proyectos a gran escala puede ser un desafío. A fin de hacer frente a estos desafíos, algunas organizaciones utilizan antipatrones para proyectos a gran escala, pero estos patrones pueden retrasar el proyecto y crear otros problemas que afecten de forma negativa a la organización. Por ejemplo, los antipatrones pueden complicar y retrasar la incorporación de los desarrolladores, la corrección de errores y la adopción de características nuevas.

En esta guía, se ofrece una alternativa al uso de antipatrones y se muestra cómo organizar el código para garantizar la escalabilidad, las pruebas y la alineación con las prácticas recomendadas de seguridad. Puede utilizar esta guía para mejorar la calidad del código de sus proyectos de IaC y maximizar la agilidad de su empresa. Esta guía está dirigida a arquitectos, directores técnicos, ingenieros de infraestructura y cualquier otra persona que busque crear un proyecto bien diseñado para proyectos a gran escala. AWS CDK

Objetivos

- Costes reducidos: puede utilizarla AWS CDK para diseñar sus propios componentes reutilizables que cumplan con los requisitos de seguridad, cumplimiento y gobierno de su organización. También puede compartir con facilidad los componentes de su organización, de modo que pueda iniciar con rapidez proyectos nuevos que se ajusten a las prácticas recomendadas de forma predeterminada.
- Lanzamiento al mercado más rápido: aproveche las funciones conocidas AWS CDK para acelerar su proceso de desarrollo. Esto aumenta la capacidad de reutilización para la implementación y reduce los esfuerzos de desarrollo.

- Mayor productividad de los desarrolladores: los desarrolladores pueden usar lenguajes de programación conocidos para definir la infraestructura. Esto ayuda a los desarrolladores a expresar y mantener AWS los recursos. Esto puede llevar a un aumento de la eficiencia y la colaboración de los desarrolladores.

Prácticas recomendadas

En esta sección, se brinda un resumen de las siguientes prácticas recomendadas:

- Organizar el código para proyectos a gran escala
- Desarrollar patrones reutilizables
- Crear o ampliar constructos
- Siga las TypeScript mejores prácticas
- Buscar vulnerabilidades de seguridad y errores de formato
- Desarrollar y perfeccionar la documentación
- Adoptar un enfoque de desarrollo basado en pruebas
- Utilizar el control de versiones y lanzamiento para los constructos
- Aplicar la administración de versiones de bibliotecas

Organizar el código para proyectos a gran escala

Por qué es importante la organización del código

Es fundamental que los AWS CDK proyectos a gran escala tengan una estructura bien definida y de alta calidad. A medida que un proyecto se hace más grande y aumenta la cantidad de características y constructos compatibles, la navegación por el código se hace más difícil. Esta dificultad puede afectar a la productividad y retrasar la incorporación de los desarrolladores.

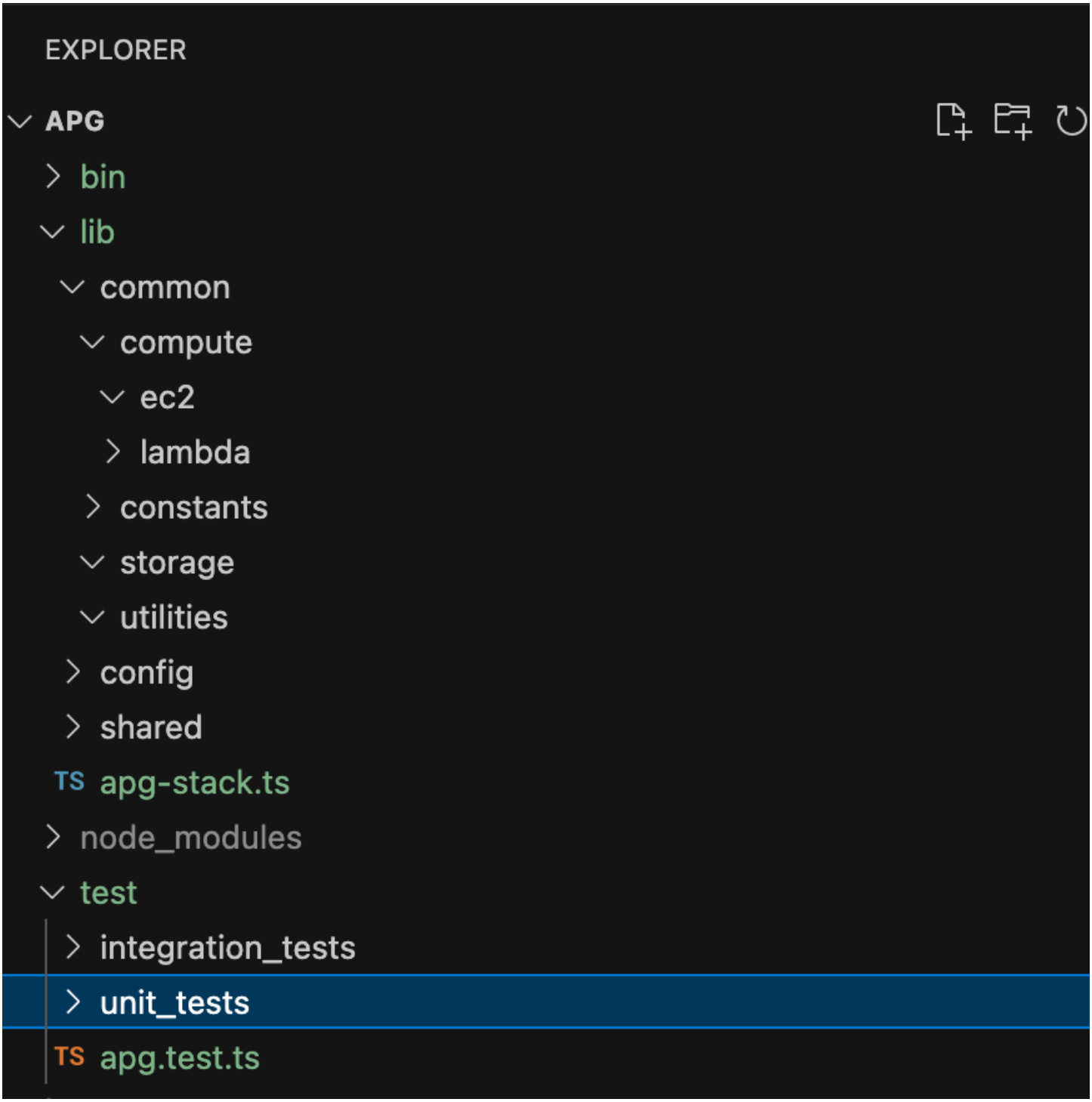
Cómo organizar el código para el escalado

Para lograr un alto nivel de flexibilidad y legibilidad del código, le recomendamos que lo divida en partes lógicas de acuerdo con la funcionalidad. Esta división refleja el hecho de que la mayoría de sus constructos se utilizan en diferentes dominios empresariales. Por ejemplo, tanto las aplicaciones de frontend como las de backend pueden requerir una AWS Lambda función y consumir el mismo código fuente. Las fábricas pueden crear objetos sin exponer la lógica de creación al cliente y utilizar una interfaz común para hacer referencia a los objetos que se crearon recientemente. Puede utilizar una fábrica como patrón eficaz para crear un comportamiento coherente en la base del código. Además, una fábrica puede servir como fuente única de información fiable para ayudarlo a evitar la repetición de códigos y facilitar la solución de problemas.

Para comprender mejor cómo funcionan las fábricas, consideremos el ejemplo de un fabricante de automóviles. Un fabricante de automóviles no necesita tener los conocimientos y la infraestructura necesarios para fabricar neumáticos. El fabricante de automóviles subcontrata a un fabricante de neumáticos especializado en esa materia y, luego, simplemente solicita los neumáticos a ese fabricante según sea necesario. El mismo principio se aplica al código. Por ejemplo, puede crear una fábrica de Lambda que sea capaz de crear funciones de Lambda de alta calidad y, a continuación, llamar a la fábrica de Lambda en el código siempre que necesite crear una función de Lambda. De igual modo, puede utilizar este mismo proceso de subcontratación para desvincular la aplicación y crear componentes modulares.

Organización del código de ejemplo

El siguiente proyecto de ejemplo, tal y como se TypeScript muestra en la imagen siguiente, incluye una carpeta común en la que puede guardar todas sus construcciones o funcionalidades comunes.



Por ejemplo, la carpeta computación (que se encuentra en la carpeta común) contiene toda la lógica de los diferentes constructos de computación. Los desarrolladores nuevos pueden agregar con facilidad constructos de computación nuevos sin afectar al resto de los recursos. Todas las demás construcciones no necesitarán crear nuevos recursos internamente. En su lugar, estos constructos

simplemente llaman a la fábrica de constructos comunes. Puede organizar otros constructos, como el almacenamiento, de la misma manera.

Las configuraciones contienen datos basados en el entorno que debe desacoplar de la carpeta común donde guarda la lógica. Le recomendamos que coloque sus datos de configuración comunes en una carpeta compartida. También recomendamos que utilice la carpeta de utilidades para ejecutar todas las funciones de ayuda y limpiar los scripts.

Desarrollar patrones reutilizables

Los patrones de diseño de software son soluciones reutilizables para problemas comunes en el desarrollo de software. Actúan como una guía o un paradigma para ayudar a los ingenieros de software a crear productos que sigan las prácticas recomendadas. Esta sección proporciona una descripción general de dos patrones reutilizables que puedes usar en tu AWS CDK base de código: el patrón Abstract Factory y el patrón Chain of Responsibility. Puede utilizar cada patrón como esquema y personalizarlo para el problema de diseño concreto de su código. Para obtener más información sobre los patrones de diseño, consulta Patrones de diseño en la Refactoring.Guru documentación.

Fábrica abstracta

El patrón Fábrica abstracta brinda interfaces para crear familias de objetos relacionados o dependientes sin especificar sus clases concretas. Este patrón se aplica a los siguientes casos de uso:

- Cuando el cliente es independiente de la forma en que se crean y componen los objetos del sistema
- Cuando el sistema consta de varias familias de objetos y estas familias se han diseñado para utilizarse juntas
- Cuando debe tener un valor de tiempo de ejecución para crear una dependencia determinada

Para obtener más información sobre el patrón Abstract Factory, consulte Abstract Factory TypeScript en la Refactoring.Guru documentación.

En el siguiente ejemplo de código, se muestra cómo utilizar el patrón Fábrica abstracta para crear una fábrica de almacenamiento de 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();
```

Cadena de responsabilidades

La Cadena de responsabilidades es un patrón de diseño de comportamientos que permite transmitir una solicitud a lo largo de la cadena de posibles controladores hasta que uno de ellos gestione la solicitud. El patrón Cadena de responsabilidades se aplica a los siguientes casos de uso:

- Cuando varios objetos, determinados en el tiempo de ejecución, son candidatos para gestionar una solicitud
- Cuando no desee especificar controladores de forma explícita en su código
- Cuando quiere enviar una solicitud a uno de varios objetos sin especificar el receptor de forma explícita

Para obtener más información sobre el patrón de cadena de responsabilidad, consulte Cadena de responsabilidad TypeScript en la Refactoring.Guru documentación.

En el siguiente código, se muestra un ejemplo de cómo se utiliza el patrón Cadena de responsabilidades a fin de crear una serie de acciones necesarias para completar la tarea.

```
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);
    }
}
```

Crear o ampliar constructos

Qué es un constructo

Una construcción es el componente básico de una AWS CDK aplicación. Una construcción puede representar un único AWS recurso, como un bucket de Amazon Simple Storage Service (Amazon S3), o puede ser una abstracción de nivel superior que consta de varios recursos relacionados. AWS Los componentes de un constructo pueden incluir una cola de trabajadores con su capacidad de computación asociada o un trabajo programado con recursos de monitoreo y un panel. AWS CDK Incluye un conjunto de construcciones denominado Construct Library. AWS La biblioteca contiene

componentes fijos para cada. Servicio de AWS Puedes usar el Construct Hub para descubrir otras construcciones de terceros y de AWS la comunidad de código abierto AWS CDK .

Cuáles son los diferentes tipos de constructos

Existen tres tipos diferentes de construcciones para: AWS CDK

- Construcciones de L1: las construcciones de capa 1, o L1, son exactamente los recursos definidos por CloudFormation, ni más ni menos. Debe brindar los recursos necesarios para la configuración. Estas construcciones de L1 son muy básicas y se deben configurar manualmente. Las construcciones L1 tienen un `Cfn` prefijo y se corresponden directamente con las especificaciones. CloudFormation Servicios de AWS Los nuevos se admiten tan pronto AWS CDK como se admita este CloudFormation tipo de servicios. CfnBuckettes un buen ejemplo de construcción L1. Esta clase representa un bucket de S3 en el que se deben configurar todas las propiedades de forma explícita. Te recomendamos que solo utilices una construcción L1 si no puedes encontrar la construcción L2 o L3 para ella.
- Constructos de C2: los constructos de capa 2, o C2, tienen un código repetitivo y una lógica de enlace comunes. Estos constructos vienen con cómodos valores predeterminados y reducen la cantidad de conocimientos que necesita saber sobre ellos. Las construcciones de nivel 2 utilizan API basadas en la intención para crear sus recursos y, por lo general, encapsulan sus módulos de nivel 1 correspondientes. Un buen ejemplo de un constructo C2 es el Bucket. Esta clase crea un depósito de S3 con propiedades y métodos predeterminados, como bucket.add LifeCycleRule (), que añade una regla de ciclo de vida al depósito.
- Constructo de C3: un constructo de capa 3, o C3, se denomina patrón. Las estructuras L3 están diseñadas para ayudarle a completar tareas comunes en varios tipos de recursos AWS, que suelen implicar varios tipos de recursos. Son incluso más específicos y obstinados que los constructos de C2 y sirven para un caso de uso específico. Por ejemplo, los aws-ecs-patterns. ApplicationLoadBalancedFargateServiceconstruct representa una arquitectura que incluye un clúster de AWS Fargate contenedores que utiliza un Application Load Balancer. Otro ejemplo es el aws-apigateway. LambdaRestApiconstruir. Este constructo representa una API de Amazon API Gateway que se encuentra respaldada por una función de Lambda.

A medida que los niveles de constructo aumentan, se hacen más suposiciones sobre cómo se utilizarán los constructos. Esto le permite proporcionar interfaces con valores predeterminados más efectivos para casos de uso muy específicos.

Cómo crear su propio constructo

Para definir su propio constructo, debe seguir un enfoque específico. Esto se debe a que todos los constructos amplían la clase `Construct`. La clase `Construct` es el componente básico del árbol de constructo. Los constructos se implementan en clases que amplían la clase base `Construct`. Todos los constructos toman tres parámetros cuando se inicializan:

- Alcance: el elemento principal o propietario de un componente fijo, ya sea una pila u otro componente fijo, que determina su lugar en el árbol de componentes fijos. Por lo general, debe pasar `this` (o `self` en Python), que representa el objeto actual, para el ámbito.
- ID: un identificador que debe ser único en este alcance. El identificador sirve como espacio de nombres para todo lo que se define en la construcción actual y se utiliza para asignar identidades únicas, como nombres de recursos e identificadores CloudFormation lógicos.
- Accesorios: conjunto de propiedades que definen la configuración inicial de la construcción.

En el siguiente ejemplo, se muestra cómo definir un constructo.

```
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
  }
}
```

Crear o ampliar un constructo de C2

Una construcción L2 representa un «componente de nube» y encapsula todo lo CloudFormation necesario para crear el componente. Una construcción L2 puede contener uno o más AWS recursos, y puedes personalizar la construcción tú mismo. La ventaja de crear o ampliar una construcción de L2 es que se pueden reutilizar los componentes de las CloudFormation pilas sin tener que redefinir el código. Simplemente puede importar el constructo como una clase.

Cuando existe una relación «es una» con una construcción existente, puede ampliarla para añadir características predeterminadas adicionales. Se recomienda reutilizar las propiedades de la construcción L2 existente. Puede sobrescribir las propiedades al modificarlas directamente en el constructo.

En el siguiente ejemplo, se muestra cómo alinearse con las prácticas recomendadas y ampliar un constructo de C2 existente denominado `s3.Bucket`. La extensión establece las propiedades predeterminadas, como `versioned`, `publicReadAccess`, `blockPublicAccess`, para garantizar que todos los objetos (en este ejemplo, los buckets de S3) creados a partir de este constructo nuevo tengan siempre establecidos estos valores predeterminados.

```
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
    });
  }
}
```

Crear un constructo de C3

La composición es la mejor opción cuando existe una relación «tiene una» con una composición de construcción existente. La composición significa que crea su propio constructo sobre otros constructos existentes. Puede crear su propio patrón para incorporar todos los recursos y sus valores predeterminados dentro de un único constructo de C3 de nivel superior que se puede compartir. El beneficio de crear sus propios constructos de C3 (patrones) es que puede reutilizar los componentes en pilas sin tener que redefinir el código. Simplemente puede importar el constructo como una clase. Estos patrones se han diseñado para ayudar a los consumidores a ofrecer diferentes recursos basados en patrones comunes con un conocimiento limitado y conciso.

En el siguiente ejemplo de código se crea una AWS CDK construcción llamada`ExampleConstruct`. Puede utilizar este constructo como plantilla para definir los componentes en la nube.

```
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
  }
}
```

En el siguiente ejemplo, se muestra cómo importar la construcción recién creada en AWS CDK la aplicación o pila.

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

En el siguiente ejemplo, se muestra cómo puede instanciar una instancia del constructo que ha ampliado desde la clase base.

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

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

Para obtener más información, consulte AWS CDK Workshop en la documentación del AWS CDK Workshop.

Escotilla de escape

Puede utilizar una trampilla de escape AWS CDK para subir un nivel de abstracción y acceder al nivel inferior de los constructos. Las trampillas de escape se utilizan para extender el componente fijo a elementos que no están expuestos en la versión actual, AWS pero que están disponibles en. CloudFormation

Le recomendamos utilizar una escotilla de escape en los siguientes escenarios:

- Hay una Servicio de AWS función disponible a través de CloudFormation ella, pero no hay `Construct` componentes fijos para ella.

- Una Servicio de AWS función está disponible a través del servicio CloudFormation y hay `Construct` componentes fijos para él, pero estos componentes aún no exponen la función. Como `Construct` las construcciones se desarrollan «a mano», a veces pueden estar a la zaga de las construcciones de CloudFormation recursos.

En el siguiente código de ejemplo, se muestra un caso de uso común para el uso de una escotilla de escape. En este ejemplo, la funcionalidad que aún no se ha implementado en el constructo de nivel superior se establece a fin de agregar `httpPutResponseHopLimit` para el escalado automático de `LaunchConfiguration`.

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

En el ejemplo de código anterior, se muestra el siguiente flujo de trabajo:

1. Define su `AutoScalingGroup` mediante el uso de un constructo de C2. La construcción L2 no permite actualizarla`httpPutResponseHopLimit`, por lo que debes usar una trampilla de escape.
2. Usas el `node.findChild()` método para localizar al elemento `LaunchConfig` secundario específico de la `AutoScalingGroup` construcción L2 y convertirlo en recurso. `CfnLaunchConfiguration`
3. Ahora puede establecer la propiedad `launchConfig.metadataOptions` en la C1 `CfnLaunchConfiguration`.

Recursos personalizados

Puedes usar recursos personalizados para escribir una lógica de aprovisionamiento personalizada en las plantillas que CloudFormation se ejecute siempre que crees, actualices (si cambiaste el recurso personalizado) o elimines pilas. Por ejemplo, puedes usar un recurso personalizado si quieres incluir recursos que no estén disponibles en. AWS CDK De esta forma, puede seguir administrando todos los recursos relacionados en una sola pila.

La creación de un recurso personalizado implica escribir una función de Lambda que responda a los eventos del ciclo de vida CREATE, UPDATE y DELETE de un recurso. Si tu recurso personalizado debe realizar solo una llamada a la API, considera usar la AwsCustomResourceconstrucción. Esto permite realizar llamadas arbitrarias al SDK durante una CloudFormation implementación. De lo contrario, le sugerimos que escriba su propia función de Lambda para llevar a cabo el trabajo que debe realizar.

Para obtener más información sobre los recursos personalizados, consulta los recursos personalizados en la CloudFormation documentación. Para ver un ejemplo de cómo utilizar un recurso personalizado, consulte el repositorio de recursos personalizados en GitHub.

El siguiente ejemplo muestra cómo crear una clase de recurso personalizada para iniciar una función Lambda y enviar CloudFormation una señal de éxito o error.

```
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();
    }
}
```

En el siguiente ejemplo, se muestra la lógica principal del 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)
```

El siguiente ejemplo muestra cómo la AWS CDK pila llama al 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 las TypeScript prácticas recomendadas

TypeScript es un lenguaje que amplía las capacidades de JavaScript. Es un lenguaje fuertemente tipificado y orientado a objetos. Se puede utilizar TypeScript para especificar los tipos de datos que se transmiten en el código y tiene la capacidad de informar de errores cuando los tipos no coinciden. En esta sección se proporciona una descripción general de las prácticas TypeScript recomendadas.

Describir los datos

Puede utilizarla TypeScript para describir la forma de los objetos y funciones del código. El uso del tipo any equivale a dejar de comprobar el tipo de una variable. Recomendamos que evite el uso de any en su código. A continuación se muestra un ejemplo.

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

Utilizar enumeraciones

Puede utilizar enumeraciones para definir un conjunto de constantes con nombre y definir estándares que se puedan reutilizar en su base de código. Le recomendamos que exporte las enumeraciones una vez a nivel global y, a continuación, deje que otras clases importen y utilicen las enumeraciones. Suponga que desea crear un conjunto de posibles acciones para capturar los eventos en su base de código. TypeScript proporciona enumeraciones numéricas y basadas en cadenas. En el siguiente ejemplo, se utiliza una enumeración.

```
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)
```

Utilizar interfaces

Una interfaz es un contrato para la clase. Si crea un contrato, sus usuarios deberán cumplirlo. En el siguiente ejemplo, se utiliza una interfaz para estandarizar las `props` y garantizar que las personas que llaman ingresen el parámetro esperado al utilizar esta clase.

```
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
})
```

Algunas propiedades solo se pueden modificar cuando se crea un objeto por primera vez. Puede especificar esto al poner `readonly` antes del nombre de la propiedad, como se muestra en el siguiente ejemplo.

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

Ampliar interfaces

La amplificación de las interfaces reduce la duplicación, ya que no es necesario copiar las propiedades entre las interfaces. Además, el lector del código puede entender con facilidad las relaciones de la aplicación.

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

Evitar interfaces vacías

Le recomendamos que evite las interfaces vacías debido a los posibles riesgos que conllevan. En el siguiente ejemplo, hay una interfaz vacía llamada. `BucketProps` Los objetos `myS3Bucket1` y `myS3Bucket2` son válidos, pero siguen estándares diferentes porque la interfaz no exige ningún contrato. El siguiente código compilará e imprimirá las propiedades, pero esto generará incoherencias en la aplicación.

```
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",
```

```
});
```

Utilizar fábricas

En un patrón Fábrica abstracta, una interfaz es responsable de crear una fábrica de objetos relacionados sin especificar sus clases de forma explícita. Por ejemplo, puede crear una fábrica de Lambda para crear funciones de Lambda. En lugar de crear una nueva función Lambda dentro de su construcción, delega el proceso de creación en la fábrica. Para obtener más información sobre este patrón de diseño, consulte Abstract Factory TypeScript en la Refactoring.Guru documentación.

Utilizar la desestructuración en las propiedades

La desestructuración, introducida en ECMAScript 6 (ES6), es una JavaScript función que permite extraer varios datos de una matriz u objeto y asignarlos a sus propias variables.

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

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

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

Definir las convenciones de nomenclatura estándar

La aplicación de una convención de nomenclatura mantiene la coherencia de la base de código y reduce la sobrecarga a la hora de pensar en cómo nombrar una variable. Le recomendamos lo siguiente:

- Utilice camelCase para los nombres de variables y funciones.
- Utilice UPPER_CASE para que las constantes globales indiquen claramente los valores inmutables en tiempo de compilación.
- Se utiliza para los nombres de las clases PascalCase y los nombres de las interfaces.
- Utilice camelCase para los miembros de la interfaz.
- Se utiliza PascalCase para nombres de tipos y nombres de enumeración.
- Nombre los archivos con camelCase (por ejemplo, `ebsVolumes.tsx` o `storage.ts`)

A continuación se muestran ejemplos de estas convenciones 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 { }
```

No utilices la palabra clave var

La `let` sentencia se utiliza para declarar una variable local en TypeScript. Es similar a la `var` palabra clave, pero tiene algunas restricciones de alcance en comparación con la `var` palabra clave. Una variable declarada en un bloque con `let` solo se puede utilizar en ese bloque. La `var` palabra clave no puede tener un ámbito de bloques, lo que significa que se puede acceder a ella desde fuera de un bloque concreto (representado por`{}`), pero no fuera de la función en la que está definida. Puede volver a declarar y actualizar las variables. `var` Se recomienda evitar el uso de la palabra clave. `var`

Considerar el uso de ESLint y Prettier

ESLint analiza de forma estática su código para encontrar problemas con rapidez. Puede utilizar ESLint para crear una serie de afirmaciones (denominadas reglas lint) que definen cómo debe verse o comportarse su código. ESLint también tiene sugerencias de reparación automática para ayudarlo a mejorar su código. Por último, puede utilizar ESLint para cargar reglas lint desde complementos compartidos.

Prettier es un formateador de código conocido que admite una variedad de lenguajes de programación diferentes. Puede utilizar Prettier para establecer el estilo de su código y evitar formatear el código de forma manual. Tras la instalación, puede actualizar su archivo `package.json` y ejecutar los comandos `npm run format` y `npm run lint`.

El siguiente ejemplo le muestra cómo habilitar ESLint y el formateador Prettier para su proyecto. 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)'"
}
```

Utilizar modificadores de acceso

El modificador privado TypeScript limita la visibilidad solo a la misma clase. Cuando agrega el modificador privado a una propiedad o método, puede acceder a esa propiedad o método dentro de la misma clase.

El modificador público permite acceder a las propiedades y los métodos de las clases desde todas las ubicaciones. Si no especificas ningún modificador de acceso para las propiedades y los métodos, usarán el modificador público de forma predeterminada.

El modificador protegido permite acceder a las propiedades y los métodos de una clase dentro de la misma clase y dentro de las subclases. Usa el modificador protegido cuando pienses crear subclases en tu aplicación. AWS CDK

Usa tipos de utilidades

Los tipos de utilidades TypeScript son funciones de tipos predefinidas que realizan transformaciones y operaciones en los tipos existentes. Esto le ayuda a crear tipos nuevos basados en los tipos existentes. Por ejemplo, puede cambiar o extraer propiedades, hacer que las propiedades sean opcionales u obligatorias, o crear versiones inmutables de los tipos. Al usar tipos de utilidades, puede definir tipos más precisos y detectar posibles errores en tiempo de compilación.

<Tipo parcial >

`Partial`marca todos los miembros de un tipo de entrada `Type` como opcionales. Esta utilidad devuelve un tipo que representa todos los subconjuntos de un tipo determinado. A continuación se muestra un ejemplo de `Partial`.

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

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

Tipo obligatorio < >

`Required`hace lo contrario de`Partial`. Hace que todos los miembros de un tipo de entrada `Type` no sean opcionales (en otras palabras, obligatorios). A continuación se muestra un ejemplo 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
};
```

Buscar vulnerabilidades de seguridad y errores de formato

La infraestructura como código (IaC) y la automatización se han vuelto fundamentales para las empresas. Debido a que la IaC es sumamente sólida, tiene la gran responsabilidad de administrar los riesgos de seguridad. Los riesgos de seguridad más comunes de la IaC pueden incluir los siguientes:

- Over-permissive AWS Identity and Access Management (IAM) privilegios
- Grupos de seguridad abiertos
- Recursos no cifrados
- Registros de acceso no activados

Enfoques y herramientas de seguridad

Recomendamos que implemente los siguientes enfoques de seguridad:

- Detección de vulnerabilidades en el desarrollo: corregir las vulnerabilidades en la producción es caro y conlleva mucho tiempo debido a la complejidad del desarrollo y la distribución de las revisiones de software. Además, las vulnerabilidades en la producción implican el riesgo de ser explotadas. Le recomendamos que emplee el escaneo de código en sus recursos de IaC para poder detectar y corregir las vulnerabilidades antes de pasar a la producción.
- Cumplimiento y corrección automática: AWS Config ofrece reglas AWS administradas. Estas reglas le ayudan a garantizar el cumplimiento y le permiten intentar la corrección automática mediante AWS Systems Manager la automatización. También puede crear y asociar documentos de automatización personalizados mediante AWS Config reglas.

Herramientas de desarrollo comunes

Las herramientas que se describen en esta sección lo ayudarán a ampliar su funcionalidad integrada con sus propias reglas personalizadas. Le recomendamos que alinee las reglas personalizadas con los estándares de su organización. Estas son algunas herramientas de desarrollo comunes que se deben tener en cuenta:

- Use cdk-nag para validar que las construcciones dentro de un ámbito determinado cumplan con un conjunto definido de reglas. También puede utilizar cdk-nag para la supresión de reglas y los informes de conformidad. La herramienta cdk-nag valida las construcciones ampliando aspectos de. AWS CDK Para obtener más información, consulte Administrar la seguridad de las aplicaciones y el cumplimiento de las normas y cdk-nag en el AWS Cloud Development Kit (AWS CDK) blog. AWS DevOps
- Utilice la herramienta de código abierto Checkov para realizar análisis estáticos en su entorno de IaC. Checkov ayuda a identificar los errores de configuración en la nube escaneando el código de su infraestructura en Kubernetes, Terraform o. CloudFormation Puede utilizar Checkov para obtener resultados en diferentes formatos, incluidos JSON, JUnit XML o CLI. Checkov puede gestionar las variables de forma eficaz mediante la creación de un gráfico que muestre la dependencia dinámica del código. Para obtener más información, consulte el repositorio de Checkov de Bridgecrew. GitHub
- Utilice TFLint para comprobar si hay errores y sintaxis obsoleta y para obtener ayuda en la aplicación de las prácticas recomendadas. Tenga en cuenta que es posible que TFLint no valide

los problemas específicos del proveedor. Para obtener más información sobre TFlint, consulte el repositorio TFlint de Terraform Linters. GitHub

- Utilice Amazon Q Developer para realizar escaneos de seguridad. Cuando se utiliza en un entorno de desarrollo integrado (IDE), Amazon Q Developer proporciona asistencia para el desarrollo de AI-powered software. Puede hablar sobre el código, completar el código en línea, generar código nuevo, escanear el código en busca de vulnerabilidades de seguridad y realizar actualizaciones y mejoras en el código.

Desarrollar y perfeccionar la documentación

La documentación es fundamental para el éxito de su proyecto. La documentación no solo explica cómo funciona su código, sino que también ayuda a los desarrolladores a comprender mejor las características y la funcionalidad de sus aplicaciones. Desarrollar y perfeccionar la documentación de alta calidad puede fortalecer el proceso de desarrollo de software, mantener un software de alta calidad y ayudar a la transferencia de conocimientos entre los desarrolladores.

Existen dos categorías de documentación: la documentación incluida en el código y la documentación de respaldo sobre el código. La documentación incluida en el código se presenta en forma de comentarios. La documentación de respaldo sobre el código puede consistir en archivos README y documentos externos. No es raro que los desarrolladores piensen en la documentación como una sobrecarga, ya que el código en sí es fácil de entender. Esto podría ser cierto para proyectos pequeños, pero la documentación es fundamental para proyectos a gran escala en los que participan varios equipos.

Se recomienda que el autor del código escriba la documentación, ya que conoce bien sus funcionalidades. Los desarrolladores pueden tener dificultades con la sobrecarga adicional que supone mantener una documentación de respaldo independiente. Para superar este desafío, los desarrolladores pueden agregar los comentarios al código y extraerlos de forma automática a fin de que todas las versiones del código y la documentación se encuentren sincronizadas.

Existe una variedad de herramientas diferentes que ayudan a los desarrolladores a extraer los comentarios del código y generar la documentación correspondiente. Esta guía se centra en TypeDoc la herramienta preferida para AWS CDK las construcciones.

Por qué se requiere documentación de código para AWS CDK constructos

AWS CDK Varios equipos de una organización crean estructuras comunes y las comparten entre diferentes equipos para su consumo. Una buena documentación ayuda a los usuarios de

la biblioteca de constructos a integrar con facilidad los constructos y a crear su infraestructura con el mínimo esfuerzo. Mantener todos los documentos sincronizados es una tarea ardua. Le recomendamos que mantenga el documento dentro del código, que se extraerá mediante la TypeDoc biblioteca.

TypeDoc Utilizándolo con el AWS Construir biblioteca

TypeDoc es un generador de documentos para TypeScript. Puede utilizarlos TypeDoc para leer los archivos TypeScript fuente, analizar los comentarios de esos archivos y, a continuación, generar un sitio estático que contenga la documentación del código.

El siguiente código muestra cómo realizar la integración TypeDoc con la biblioteca AWS Construct y, a continuación, añadir los siguientes paquetes a su `package.json` archivo`devDependencies`.

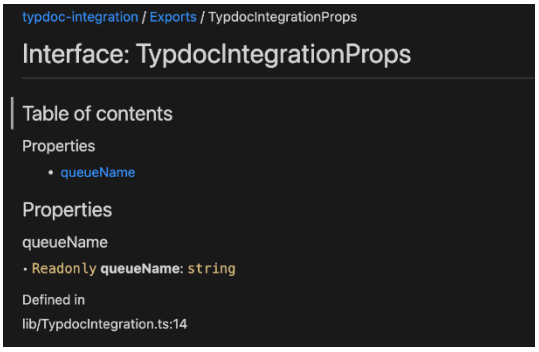
```
{

  "devDependencies": {

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

}
```

Para agregar `typedoc.json` en la carpeta de la biblioteca de CDK, utilice el siguiente código.

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

Para generar los archivos README, ejecute el `npx typedoc` comando en el directorio raíz del proyecto de la biblioteca de AWS CDK construcciones.

El siguiente documento de ejemplo ha sido generado por TypeDoc.



Para obtener más información sobre las opciones de TypeDoc integración, consulte los comentarios del documento en la TypeDoc documentación.

Adoptar un enfoque de desarrollo basado en pruebas

Le recomendamos que siga un enfoque de desarrollo basado en pruebas (TDD) con el. AWS CDK El TDD es un enfoque de desarrollo de software en el que se desarrollan casos de prueba para especificar y validar el código. En pocas palabras, primero se crean casos de prueba para cada funcionalidad y, si la prueba falla, se escribe el código nuevo a fin de pasar la prueba y hacer que el código sea simple y libre de errores.

Puede utilizar el TDD para escribir primero el caso de prueba. Esto lo ayuda a validar la infraestructura con diferentes restricciones de diseño en términos de aplicar la política de seguridad para los recursos y seguir una convención de nomenclatura única para el proyecto. El enfoque estándar para probar AWS CDK aplicaciones es utilizar el módulo de AWS CDK aserciones y los marcos de prueba más populares, como Jest para JavaScript y/o TypeScript pytest para Python.

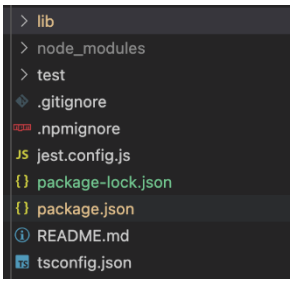
Hay dos categorías de pruebas que puedes escribir para tus aplicaciones: AWS CDK

- Usa afirmaciones detalladas para probar un aspecto específico de la CloudFormation plantilla generada, como «este recurso tiene esta propiedad con este valor». Estas pruebas pueden detectar regresiones y también son útiles cuando se desarrollan nuevas funciones con TDD (escribe primero una prueba y luego haz que pase escribiendo una implementación correcta). Fine-grained las aserciones son las pruebas que escribirás con más frecuencia.
- Utilice pruebas instantáneas para comparar la CloudFormation plantilla sintetizada con una plantilla de referencia previamente almacenada. Las pruebas de instantáneas permiten refactorizar con libertad, ya que puede estar seguro de que el código refactorizado funciona exactamente de la misma manera que el original. Si los cambios eran intencionales, puede aceptar un punto de referencia nuevo para pruebas futuras. Sin embargo, AWS CDK las actualizaciones también

pueden provocar cambios en las plantillas sintetizadas, por lo que no puede confiar únicamente en las instantáneas para asegurarse de que la implementación es correcta.

Prueba unitaria

Esta guía se centra TypeScript específicamente en la integración de las pruebas unitarias. Para habilitar las pruebas, asegúrate de que el `package.json` archivo tenga las siguientes bibliotecas: `@types/jestjest`, y `ts-jest` en`devDependencies`. Para agregar estos paquetes, ejecute el comando `cdk init lib --language=typescript`. Tras ejecutar el comando anterior, verá la siguiente estructura.



El siguiente código es un ejemplo de un `package.json` archivo que está habilitado con la 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",
    ...
  }
}
```

En la carpeta Prueba, puede escribir el caso de prueba. El siguiente ejemplo muestra un caso de prueba para una AWS CodePipeline construcción.

```
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 realizar una prueba, ejecute el comando `npm run test` en su proyecto. La prueba devuelve los siguientes resultados.

```
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 obtener más información sobre los casos de prueba, consulte Probar construcciones en la Guía para AWS Cloud Development Kit (AWS CDK) desarrolladores.

Prueba de integración

Las pruebas de integración para AWS CDK construcciones también se pueden incluir mediante un `integ-tests` módulo. Una prueba de integración debe definirse como una AWS CDK aplicación.

Debe haber una relación de uno a uno entre una prueba de integración y una AWS CDK aplicación. Para obtener más información, visita el módulo integ-tests-alpha en la referencia de la API.AWS CDK

Utilizar el control de versiones y lanzamiento para los constructos

Control de versiones para el AWS CDK

AWS CDK Varios equipos pueden crear estructuras comunes y compartirlas en una organización para su consumo. Por lo general, los desarrolladores lanzan nuevas funciones o corrigen errores en sus AWS CDK construcciones comunes. Estas construcciones las utilizan las AWS CDK aplicaciones o cualquier otra AWS CDK construcción existente como parte de una dependencia. Por este motivo, es fundamental que los desarrolladores actualicen y publiquen su constructo con las versiones semánticas adecuadas de forma independiente. AWS CDK Las aplicaciones posteriores u otras AWS CDK construcciones pueden actualizar su dependencia para usar la versión de construcción publicada recientemente. AWS CDK

El control de versiones semántico (Semver) es un conjunto de reglas, o un método, para proporcionar números de software únicos al software del equipo. Las versiones se definen de la siguiente manera:

- Una versión IMPORTANTE consiste en cambios de API incompatibles o cambios importantes.
- Una versión MENOR consiste en una funcionalidad que se agrega de forma compatible con versiones anteriores.
- Una versión REVISIÓN consiste en correcciones de errores compatibles con versiones anteriores.

Para obtener más información sobre el control de versiones semántico, consulte la Especificación del control de versiones semántico (SemVer) en la documentación sobre el control de versiones semántico.

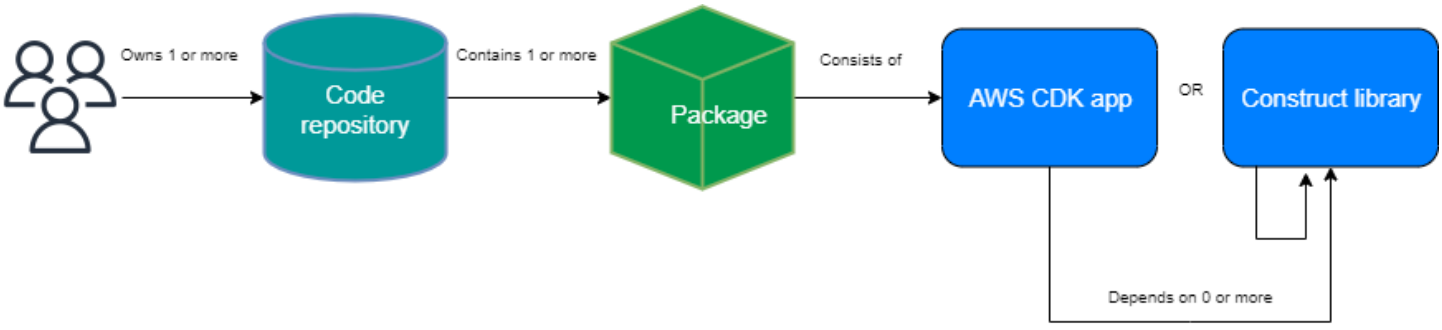
Repositorio y empaquetado para AWS CDK constructos

Como AWS CDK las construcciones las desarrollan diferentes equipos y son utilizadas por varias AWS CDK aplicaciones, puede utilizar un repositorio independiente para cada AWS CDK construcción. Esto también puede ayudarlo a aplicar el control de acceso. Cada repositorio puede contener todo el código fuente relacionado con la misma AWS CDK construcción junto con todas sus dependencias. Al mantener una sola aplicación (es decir, una AWS CDK construcción) en un único repositorio, se puede reducir el alcance del impacto de los cambios durante la implementación.

AWS CDK No solo genera CloudFormation plantillas para implementar la infraestructura, sino que también agrupa activos en tiempo de ejecución, como funciones Lambda e imágenes de Docker, y los implementa junto con su infraestructura. No solo es posible combinar el código que define la infraestructura y el código que implementa la lógica de tiempo de ejecución en una sola construcción, sino que es una práctica recomendada. No es necesario que estos dos tipos de código se encuentren en repositorios separados o incluso en paquetes separados.

Para consumir paquetes más allá de los límites del repositorio, debes tener un repositorio de paquetes privado, similar a npm o Maven Central PyPi, pero interno en tu organización. También debe tener un proceso de publicación que compile, pruebe y publique el paquete en el repositorio de paquetes privado. Puede crear repositorios privados, como un PyPi servidor, mediante una máquina virtual (VM) local o Amazon S3. Al diseñar o crear un registro de paquetes privado, es fundamental tener en cuenta el riesgo de interrupción del servicio debido a la alta disponibilidad y escalabilidad. Un servicio gestionado sin servidor alojado en la nube para almacenar paquetes puede reducir considerablemente la sobrecarga de mantenimiento. Por ejemplo, se puede utilizar AWS CodeArtifactpara alojar paquetes para los lenguajes de programación más populares. También se puede utilizar CodeArtifact para establecer conexiones de repositorios externos y replicarlas en ellas CodeArtifact.

El administrador de paquetes de tu idioma (por ejemplo, npm for TypeScript o JavaScript applications) gestiona las dependencias de los paquetes del repositorio de paquetes. El administrador de paquetes se asegura de que las compilaciones sean repetibles al registrar las versiones específicas de cada paquete del que depende su aplicación y, a continuación, le permite actualizar esas dependencias de forma controlada, como se muestra en el siguiente diagrama.

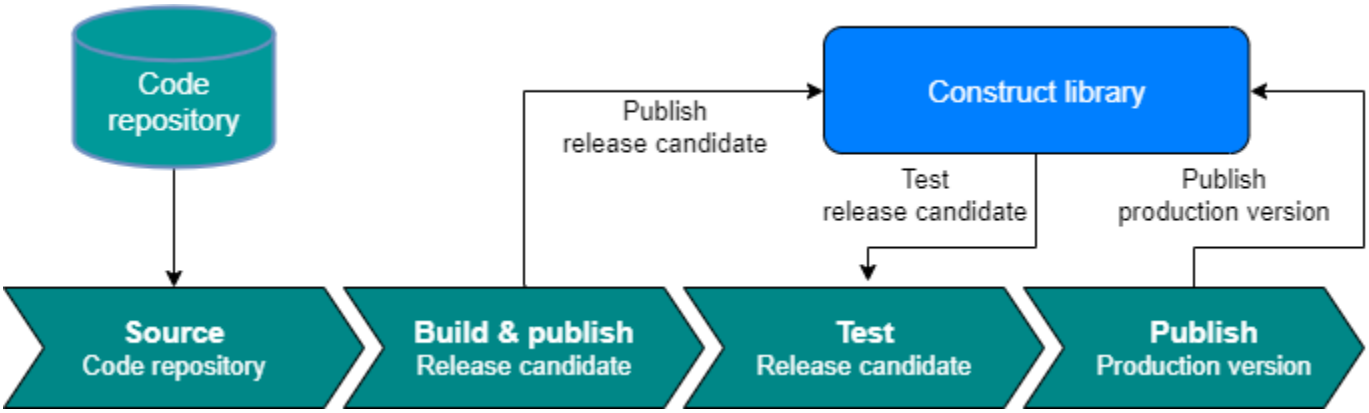


Construya una versión para AWS CDK

Le recomendamos que cree su propia canalización automatizada para crear y lanzar nuevas versiones de AWS CDK construcción. Si implementa un proceso de aprobación de solicitudes de extracción adecuado, una vez que confirme e inserte su código fuente en la rama principal del

repositorio, la canalización podrá compilar y crear una versión candidata a ser lanzada. Puedes actualizar esa versión CodeArtifact y probarla antes de lanzar la versión lista para producción. Si lo desea, puede probar la nueva versión de AWS CDK construcción localmente antes de fusionar el código con la rama principal. Esto hace que la canalización publique la versión lista para la producción. Tenga en cuenta que los constructos y los paquetes compartidos deben probarse por separado de la aplicación que los utilice, como si se estuvieran lanzando al público.

En el siguiente diagrama se muestra un ejemplo del proceso de publicación de una AWS CDK versión.



Puede utilizar los siguientes comandos de ejemplo para crear, probar y publicar paquetes npm. En primer lugar, inicie sesión en el repositorio de artefactos al ejecutar el siguiente comando.

```
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>
```

A continuación, complete los pasos siguientes:

1. Instale los paquetes necesarios en función del archivo `package.json`: `npm install`
2. Cree la versión candidata a ser lanzada: `npm version prerelease --preid rc`
3. Cree el paquete npm: `npm run build`
4. Pruebe el paquete npm: `npm run test`
5. Publique el paquete npm: `npm publish`

Aplicar la administración de versiones de bibliotecas

La gestión del ciclo de vida es un desafío importante cuando se mantienen bases AWS CDK de código. Por ejemplo, supongamos que comienza un AWS CDK proyecto con la versión 1.97 y, después, la versión 1.169 estará disponible más adelante. La versión 1.169 ofrece características nuevas y correcciones de errores, pero ha implementado su infraestructura con la versión anterior. Ahora, actualizar los constructos se convierte en un desafío, ya que esta brecha aumenta debido a los cambios importantes que podrían introducirse en las versiones nuevas. Esto puede ser un desafío si tiene muchos recursos en su entorno. El patrón presentado en esta sección puede ayudarle a administrar la versión de la AWS CDK biblioteca mediante la automatización. Este es el flujo de trabajo de este patrón:

1. Al lanzar un nuevo producto de CodeArtifact Service Catalog, las versiones de la AWS CDK biblioteca y sus dependencias se almacenan en el `package.json` archivo.
2. Implementa una canalización común que realiza un seguimiento de todos los repositorios para que pueda aplicarles actualizaciones automáticas si no se producen cambios importantes.
3. Una AWS CodeBuild etapa comprueba el árbol de dependencias y busca los cambios más importantes.
4. La canalización crea una rama de característica y, a continuación, ejecuta `cdk synth` con la versión nueva para confirmar que no hay errores.
5. La versión nueva se implementa en el entorno de prueba y, finalmente, ejecuta una prueba de integración para asegurarse de que la implementación es correcta.
6. Puede utilizar dos colas de Amazon Simple Queue Service (Amazon SQS) para realizar un seguimiento de las pilas. Los usuarios pueden revisar las pilas de forma manual en la cola de excepciones y corregir los cambios importantes. Los elementos que superen la prueba de integración pueden fusionarse y publicarse.

Preguntas frecuentes

¿Qué problemas se pueden TypeScript resolver?

Por lo general, puede eliminar los errores del código al escribir pruebas automatizadas, verificar de forma manual que el código funciona según lo esperado y, por último, al hacer que otra persona lo valide. Validar las conexiones entre todas las partes del proyecto lleva mucho tiempo. Para acelerar el proceso de validación, puedes usar un lenguaje de tipografía comprobada, por ejemplo, TypeScript para automatizar la validación del código y proporcionar comentarios instantáneos durante el desarrollo.

¿Por qué debo usarlo? TypeScript

TypeScript es un lenguaje de código abierto que simplifica el JavaScript código, lo que facilita su lectura y depuración. TypeScript también proporciona herramientas JavaScript IDEs y prácticas de desarrollo altamente productivas, como la comprobación estática. Además, TypeScript ofrece los beneficios de ECMAScript 6 (ES6) y puede aumentar su productividad. Por último, TypeScript puede ayudarte a evitar los molestos errores que suelen encontrar los desarrolladores al escribir JavaScript comprobando el código.

¿Debo usar el AWS CDK o CloudFormation?

Le recomendamos que utilice el AWS Cloud Development Kit (AWS CDK) en lugar de AWS CloudFormation, si su organización tiene la experiencia en desarrollo para aprovechar el AWS CDK. Esto se debe a que AWS CDK es más flexible que CloudFormation, ya que puede utilizar un lenguaje de programación y conceptos de programación orientada a objetos. Tenga en cuenta que puede utilizarlos CloudFormation para crear AWS recursos de forma ordenada y predecible. En CloudFormation, los recursos se escriben en archivos de texto con el formato JSON o YAML.

¿Qué pasa si AWS CDK no es compatible con un lanzamiento reciente? Servicio de AWS

Puedes usar una anulación sin procesar o un recurso CloudFormation personalizado.

¿Cuáles son los diferentes lenguajes de programación compatibles con? AWS CDK

AWS CDK está disponible generalmente en Python JavaScript TypeScript, Java, C# y Go (en la versión preliminar para desarrolladores).

¿Cuánto AWS CDK cuesta?

No hay ningún cargo adicional por el AWS CDK. Usted paga por los AWS recursos (como las instancias de Amazon EC2 o los balanceadores de carga de Elastic Load Balancing) que se crean cuando los usa AWS CDK de la misma manera que si los creara manualmente. Solo paga por lo que utiliza, cuando lo utiliza. No se requieren pagos mínimos ni compromisos iniciales.

Pasos a seguir a continuación

Le recomendamos que comience a construir desde AWS Cloud Development Kit (AWS CDK) dentro TypeScript. Para obtener más información, consulte el taller del día de AWS CDK inmersión.

Recursos

Referencias

- AWS Construcciones de AWS soluciones (soluciones)
- AWS Cloud Development Kit (AWS CDK) (GitHub)
- AWS Referencia de la API de Construct Library (documentación AWS CDK de referencia)
- AWS CDK Documentación de referencia (documentación AWS CDK de referencia)
- AWS CDK Taller de un día de inmersión (AWS taller de estudio)

Herramientas

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

Guías y patrones

- AWS Soluciones: construye patrones (AWS documentación)

Historial de documentos

En la siguiente tabla, se describen cambios significativos de esta guía. Si quiere recibir notificaciones de futuras actualizaciones, puede suscribirse a las notificaciones RSS.

Cambio	Descripción	Fecha
Prácticas recomendadas actualizadas	Eliminamos la recomendación de usar cfn-nag en la sección de herramientas de desarrollo comunes. En la sección Definir convenciones de nomenclatura estándar, agregamos convenciones de nomenclatura estándar para las constantes globales y agregamos ejemplos de nomenclatura.	23 de octubre de 2025
Código de actualización	Hemos actualizado los ejemplos de código en la sección Siga las TypeScript mejores prácticas.	16 de febrero de 2024
Agrega secciones	Hemos añadido las secciones Uso de tipos de utilidades y Pruebas de integración.	10 de enero de 2024
Actualización menor	Ejemplo de código actualizado para crear un constructo de C3.	16 de junio de 2023
Publicación inicial	—	8 de diciembre de 2022

Las traducciones son generadas a través de traducción automática. En caso de conflicto entre la traducción y la version original de inglés, prevalecerá la version en inglés.