Meilleures pratiques d'utilisation de l' AWS CDK in pour TypeScript créer des projets IaC

# AWS Conseils prescriptifs

# AWS Conseils prescriptifs: Meilleures pratiques d'utilisation de l' AWS CDK in pour TypeScript créer des projets IaC

# Table of Contents

# Meilleures pratiques d'utilisation de l' AWS CDK in pour TypeScript créer des projets IaC

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

octobre 2025 (historique du document)

Ce guide fournit des recommandations et des bonnes pratiques pour utiliser l'Cloud Development Kit AWS (AWS CDK)in TypeScript pour créer et déployer des projets d'infrastructure en tant que code (IaC) à grande échelle. AWS CDK Il s'agit d'un framework permettant de définir l'infrastructure cloud dans le code et de provisionner cette infrastructure par le biais AWS CloudFormation de celui-ci. Si vous ne disposez pas d'une structure de projet bien définie, la création et la gestion d'une AWS CDK base de code pour des projets de grande envergure peuvent s'avérer difficiles. Pour relever ces défis, certaines organisations utilisent des anti-modèles pour les projets à grande échelle, mais ces modèles peuvent ralentir votre projet et engendrer d'autres problèmes qui ont un impact négatif sur votre organisation. Par exemple, les anti-modèles peuvent compliquer et ralentir l'intégration de développeurs, les corrections de bogues et l'adoption de nouvelles fonctionnalités.

Ce guide propose une alternative à l'utilisation d'anti-modèles et vous explique comment organiser votre code à des fins de capacité de mise à l'échelle, de test et d'alignement sur les bonnes pratiques de sécurité. Vous pouvez utiliser ce guide pour améliorer la qualité du code de vos projets IaC et optimiser l'agilité de votre entreprise. Ce guide est destiné aux architectes, aux responsables techniques, aux ingénieurs d'infrastructure et à toute autre personne cherchant à créer un AWS CDK projet bien conçu pour des projets de grande envergure.

## Objectifs

- Coûts réduits — Vous pouvez utiliser le AWS CDK pour concevoir vos propres composants réutilisables qui répondent aux exigences de sécurité, de conformité et de gouvernance de votre entreprise. Vous pouvez également partager facilement des composants au sein de votre organisation, afin d'amorcer rapidement de nouveaux projets conformes aux bonnes pratiques par défaut.
- Réduction des délais de mise sur le marché : profitez des fonctionnalités habituelles du AWS CDK pour accélérer votre processus de développement. Cela augmente la réutilisabilité pour le déploiement et réduit les efforts de développement.

- Productivité accrue des développeurs : les développeurs peuvent utiliser des langages de programmation familiers pour définir l'infrastructure. Cela aide les développeurs à exprimer et à gérer AWS les ressources. Cela peut améliorer l'efficacité et la collaboration des développeurs.

# Bonnes pratiques

Cette section fournit une vue d'ensemble des bonnes pratiques suivantes :

- Organiser le code pour les projets à grande échelle
- Développer des modèles réutilisables
- Créer ou étendre des constructions
- Suivez les TypeScript meilleures pratiques
- Rechercher les vulnérabilités de sécurité et les erreurs de formatage
- Développer et affiner la documentation
- Adopter une approche de développement piloté par les tests
- Utiliser le contrôle des éditions et des versions pour les constructions
- Appliquer la gestion des versions de la bibliothèque

## Organiser le code pour les projets à grande échelle

### Pourquoi l'organisation du code est-elle importante ?

Il est essentiel que les AWS CDK projets de grande envergure aient une structure bien définie et de haute qualité. À mesure qu'un projet prend de l'ampleur et que le nombre de fonctionnalités et de constructions prises en charge augmente, la navigation du code devient plus difficile. Cette difficulté peut avoir un impact sur la productivité et ralentir l'intégration des développeurs.

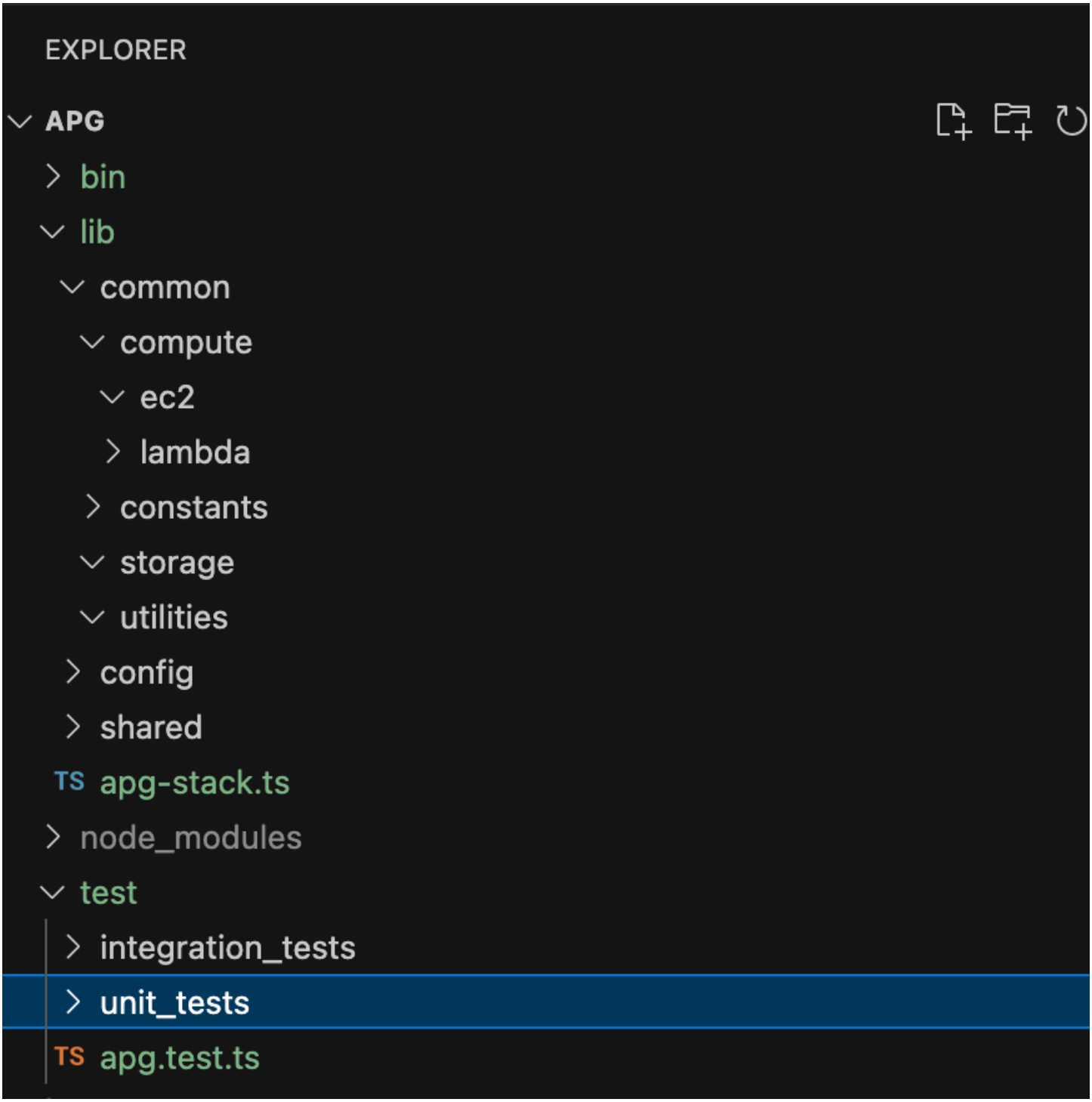
### Comment organiser votre code pour le mettre à l'échelle

Pour atteindre un niveau élevé de flexibilité et de lisibilité du code, nous vous recommandons de le diviser en éléments logiques en fonction des fonctionnalités. Cette division reflète le fait que la plupart de vos constructions sont utilisées dans différents domaines métier. Par exemple, vos applications frontales et dorsales peuvent nécessiter une AWS Lambda fonction et consommer le même code source. Les usines peuvent créer des objets sans exposer la logique de création au client et utiliser une interface commune pour faire référence aux objets récemment créés. Vous pouvez utiliser une usine comme modèle efficace pour créer un comportement cohérent dans votre base de code. En outre, une usine peut servir de source de vérité fiable unique pour vous aider à éviter le code répétitif et à simplifier le dépannage.

Pour mieux comprendre le fonctionnement des usines, prenons l'exemple d'un constructeur automobile. Un constructeur automobile n'a pas besoin des connaissances et de l'infrastructure nécessaires pour fabriquer des pneus. Le constructeur automobile sous-traite plutôt cette expertise à un fabricant spécialisé dans les pneus, puis lui commande simplement les pneus en fonction des besoins. Le même principe s'applique au code. Par exemple, vous pouvez créer une usine Lambda capable de créer des fonctions Lambda de haute qualité, puis appeler l'usine Lambda dans votre code chaque fois que vous devez créer une fonction Lambda. De même, vous pouvez utiliser ce même processus d'externalisation pour découpler votre application et créer des composants modulaires.

## Exemple d'organisation de code

L' TypeScript exemple de projet suivant, comme le montre l'image suivante, inclut un dossier commun dans lequel vous pouvez conserver toutes vos constructions ou fonctionnalités communes.




Par exemple, le dossier calculer (résidant dans le dossier commun) contient toute la logique des différentes constructions de calcul. Les nouveaux développeurs peuvent facilement ajouter de nouvelles constructions de calcul sans affecter les autres ressources. Toutes les autres constructions n'auront pas besoin de créer de nouvelles ressources en interne. Au lieu de cela, ces constructions

appellent simplement l'usine de constructions communes. Vous pouvez organiser d'autres constructions, telles que le stockage, de la même manière.

Les configurations contiennent des données basées sur l'environnement que vous devez dissocier du dossier commun dans lequel vous conservez la logique. Nous vous recommandons de placer vos données de configuration dans un dossier partagé. Nous vous recommandons également d'utiliser le dossier utilitaires pour servir toutes les fonctions d'assistance et les scripts de nettoyage.

# Développer des modèles réutilisables

Les modèles de conception logicielle sont des solutions réutilisables aux problèmes courants du développement logiciel. Ils servent de guide ou de paradigme pour aider les ingénieurs logiciels à créer des produits conformes aux bonnes pratiques. Cette section fournit un aperçu de deux modèles réutilisables que vous pouvez utiliser dans votre AWS CDK base de code : le modèle Abstract Factory et le modèle Chain of Responsibility. Vous pouvez utiliser chaque modèle comme plan et le personnaliser en fonction du problème de conception particulier de votre code. Pour plus d'informations sur les modèles de conception, consultez la section Modèles de conception dans la Refactoring.Guru documentation.

## Abstract Factory

Le modèle Abstract Factory propose des interfaces permettant de créer des familles d'objets associés ou dépendants sans spécifier leurs classes concrètes. Ce modèle s'applique aux cas d'utilisation suivants :

- Lorsque le client est indépendant de la façon dont vous créez et composez les objets dans le système.
- Lorsque le système est composé de plusieurs familles d'objets et que ces familles sont conçues pour être utilisées ensemble.
- Lorsque vous devez disposer d'une valeur d'exécution pour construire une dépendance particulière.

Pour plus d'informations sur le modèle Abstract Factory, voir Abstract Factory TypeScript dans la Refactoring.Guru documentation.

L'exemple de code suivant montre comment le modèle Abstract Factory peut être utilisé pour créer une usine de stockage Amazon Elastic Block Store (Amazon EBS).

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

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

abstract class AbstractFactory {
    abstract createEbs(): EBSStorage
}

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

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

## Chain of Responsibility

Chain of Responsibility est un modèle de conception comportementale qui vous permet de transmettre une demande à la chaîne des gestionnaires potentiels jusqu'à ce que l'un d'eux la traite. Le modèle Chain of Responsibility s'applique aux cas d'utilisation suivants :

- Lorsque plusieurs objets, déterminés lors de l'exécution, sont candidats pour traiter une demande.
- Lorsque vous ne souhaitez pas spécifier de gestionnaires explicitement dans votre code
- Lorsque vous souhaitez envoyer une demande à l'un des nombreux objets sans spécifier explicitement le destinataire.

Pour plus d'informations sur le modèle de chaîne de responsabilité, voir Chaîne de responsabilité TypeScript dans la Refactoring.Guru documentation.

Le code suivant montre un exemple de la manière dont le modèle Chain of Responsibility est utilisé pour générer une série d'actions nécessaires à l'exécution de la tâche.

```
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);
    }
}
```

# Créer ou étendre des constructions

## Qu'est-ce qu'une construction ?

Une construction est l'élément de base d'une AWS CDK application. Une construction peut représenter une AWS ressource unique, telle qu'un bucket Amazon Simple Storage Service (Amazon S3), ou il peut s'agir d'une abstraction de niveau supérieur composée de plusieurs ressources connexes. AWS Les composants d'une construction peuvent inclure une file d'attente de travail avec sa capacité de calcul associée, ou une tâche planifiée avec des ressources de surveillance et un tableau de bord. AWS CDK Il inclut une collection de constructions appelée AWS Construct Library. La bibliothèque contient des constructions pour chaque Service AWS. Vous pouvez utiliser

le Construct Hub pour découvrir des constructions supplémentaires provenant de AWS tiers et de la communauté open source AWS CDK .

## Quels sont les différents types de construction ?

Il existe trois types de constructions différents pour : AWS CDK

- Constructions L1 — Les constructions de couche 1, ou L1, sont exactement les ressources définies par CloudFormation —ni plus, ni moins. Vous devez fournir vous-même les ressources nécessaires à la configuration. Ces constructions L1 sont très basiques et doivent être configurées manuellement. Les constructions L1 ont un `Cfn` préfixe et correspondent directement aux spécifications. CloudFormation Services AWS Les nouveaux sont pris en charge AWS CDK dès que ces services sont CloudFormation pris en charge. CfnBucketest un bon exemple de construction L1. Cette classe représente un compartiment S3 dans lequel vous devez configurer explicitement toutes les propriétés. Nous vous recommandons de n'utiliser une construction L1 que si vous ne trouvez pas la construction L2 ou L3 correspondante.
- Constructions L2 : les constructions Layer 2, ou L2, ont un code réutilisable et une logique de collage communs. Ces constructions comportent des valeurs par défaut pratiques et réduisent la quantité de connaissances que vous devez connaître à leur sujet. Les constructions L2 utilisent des API basées sur l'intention pour créer vos ressources et encapsulent généralement les modules L1 correspondants. Un bon exemple de construction L2 est Compartiment. Cette classe crée un compartiment S3 avec des propriétés et des méthodes par défaut telles que bucket.add LifeCycleRule (), qui ajoute une règle de cycle de vie au compartiment.
- Constructions L3 : une construction Layer 3, ou L3, est appelée un modèle. Les constructions L3 sont conçues pour vous aider à effectuer des tâches courantes AWS, impliquant souvent plusieurs types de ressources. Elles sont encore plus spécifiques et mieux définies que les constructions L2 et répondent à un cas d'utilisation spécifique. Par exemple, les aws-ecs-patterns. ApplicationLoadBalancedFargateServiceconstruct représente une architecture qui inclut un cluster de AWS Fargate conteneurs utilisant un Application Load Balancer. Un autre exemple est le aws-apigateway. LambdaRestApiconstruire. Cette construction représente une API Amazon API Gateway qui est soutenue par une fonction Lambda.

À mesure que les niveaux de construction augmentent, de plus en plus d'hypothèses sont formulées quant à la manière dont ces constructions seront utilisées. Cela vous permet de fournir des interfaces avec des valeurs par défaut plus efficaces pour des cas d'utilisation très spécifiques.

## Comment créer votre propre construction

Pour définir votre propre construction, vous devez suivre une approche spécifique. Cela est dû au fait que toutes les constructions étendent la classe `Construct`. La classe `Construct` est la composante de base de l'arbre de construction. Les constructions sont implémentées dans des classes qui étendent la classe de base `Construct`. Toutes les constructions prennent trois paramètres lorsqu'elles sont initialisées :

- Portée — Le parent ou le propriétaire d'une construction, qu'il s'agisse d'une pile ou d'une autre construction, détermine sa place dans l'arbre des constructions. Vous devez généralement transmettre `this` (ou `self` en Python), qui représente l'objet actuel, pour la portée.
- ID : identifiant qui doit être unique dans cette portée. L'identifiant sert d'espace de noms pour tout ce qui est défini dans la construction actuelle et est utilisé pour attribuer des identités uniques, telles que les noms de ressources et les identifiants CloudFormation logiques.
- Accessoires — Ensemble de propriétés qui définissent la configuration initiale de la construction.

L'exemple suivant montre comment définir une construction.

```
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
  }
}
```

## Création ou extension d'une construction L2

Une construction L2 représente un « composant cloud » et encapsule tout ce qui CloudFormation doit être nécessaire pour créer le composant. Une construction L2 peut contenir une ou plusieurs AWS ressources, et vous êtes libre de la personnaliser vous-même. L'avantage de créer ou d'étendre une

construction L2 est que vous pouvez réutiliser les composants dans des CloudFormation piles sans redéfinir le code. Vous pouvez simplement importer la construction en tant que classe.

Lorsqu'il existe une relation « existe » avec une construction existante, vous pouvez étendre une construction existante pour ajouter des fonctionnalités par défaut supplémentaires. Il est recommandé de réutiliser les propriétés de la construction L2 existante. Vous pouvez remplacer les propriétés en les modifiant directement dans le constructeur.

L'exemple suivant montre comment s'aligner sur les bonnes pratiques et étendre une construction L2 existante appelée `s3.Bucket`. L'extension définit les propriétés par défaut, telles que `versioned`, `publicReadAccess`, `blockPublicAccess`, pour s'assurer que tous les objets (dans cet exemple, les compartiments S3) créés à partir de cette nouvelle construction auront toujours ces valeurs par défaut définies.

```
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
    });
  }
}
```

## Création d'une construction L3

La composition est le meilleur choix lorsqu'il existe une relation « a » avec une composition de construction existante. La composition signifie que vous générez votre propre construction par-dessus d'autres constructions existantes. Vous pouvez créer votre propre modèle pour encapsuler toutes les ressources et leurs valeurs par défaut dans une seule construction L3 de niveau supérieur pouvant être partagée. L'avantage de créer vos propres constructions L3 (modèles) est que vous pouvez réutiliser les composants dans des piles sans redéfinir le code. Vous pouvez simplement importer la construction en tant que classe. Ces modèles sont conçus pour aider les consommateurs à fournir de manière concise plusieurs ressources basées sur des modèles communs avec une quantité limitée de connaissances.

L'exemple de code suivant crée une AWS CDK construction appeléeExampleConstruct. Vous pouvez utiliser cette construction comme modèle pour définir vos composants cloud.

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

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

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

L'exemple suivant montre comment importer la nouvelle construction dans votre AWS CDK application ou votre pile.

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

L'exemple suivant montre comment instancier une instance de la construction que vous avez étendue à partir de la classe de base.

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

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

Pour plus d'informations, consultez AWS CDK Workshop dans la documentation de l' AWS CDK atelier.

## Trappe de secours

Vous pouvez utiliser une trappe d'échappement dans le AWS CDK pour monter d'un niveau d'abstraction afin d'accéder au niveau inférieur des constructions. Les trappes d'évacuation sont utilisées pour étendre la construction aux fonctionnalités qui ne sont pas exposées dans la version actuelle de AWS mais disponibles dans CloudFormation.

Nous vous recommandons d'utiliser une trappe de secours dans les situations suivantes :

- Une Service AWS fonctionnalité est disponible via CloudFormation, mais il n'existe aucune `Construct` structure pour celle-ci.
- Une Service AWS fonctionnalité est disponible CloudFormation et il existe `Construct` des constructions pour le service, mais celles-ci n'exposent pas encore la fonctionnalité. Comme `Construct` les concepts sont développés « à la main », ils peuvent parfois être en retard par rapport aux concepts liés aux CloudFormation ressources.

L'exemple de code suivant montre un cas d'utilisation courant d'une trappe de secours. Dans cet exemple, la fonctionnalité qui n'est pas encore implémentée dans la construction de niveau supérieur permet d'ajouter `httpPutResponseHopLimit` pour la mise à l'échelle automatique de `LaunchConfiguration`.

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

L'exemple de code précédent montre le flux de travail suivant :

1. Vous définissez votre `AutoScalingGroup` à l'aide d'une construction L2. La construction L2 ne prend pas en charge la mise à jour du`httpPutResponseHopLimit`, vous devez donc utiliser une trappe d'évacuation.
2. Vous utilisez `node.findChild()` cette méthode pour localiser l'`LaunchConfig`enfant spécifique de la `AutoScalingGroup` construction L2 et le convertir en `CfnLaunchConfiguration` ressource.
3. Vous pouvez à présent définir la propriété `launchConfig.metadataOptions` sur la `CfnLaunchConfiguration` L1.

## Ressources personnalisées

Vous pouvez utiliser des ressources personnalisées pour écrire une logique de provisionnement personnalisée dans des modèles qui CloudFormation s'exécutent chaque fois que vous créez, mettez

à jour (si vous avez modifié la ressource personnalisée) ou supprimez des piles. Par exemple, vous pouvez utiliser une ressource personnalisée si vous souhaitez inclure des ressources qui ne sont pas disponibles dans le AWS CDK. De cette façon, vous pouvez continuer à gérer toutes les ressources connexes dans une seule pile.

La génération d'une ressource personnalisée implique l'écriture d'une fonction Lambda qui répond aux événements du cycle de vie CREATE, UPDATE et DELETE d'une ressource. Si votre ressource personnalisée ne doit effectuer qu'un seul appel d'API, envisagez d'utiliser la AwsCustomResourceconstruction. Cela permet d'effectuer des appels au SDK arbitraires lors d'un CloudFormation déploiement. Sinon, nous vous suggérons d'écrire votre propre fonction Lambda pour effectuer le travail que vous devez exécuter.

Pour plus d'informations sur les ressources personnalisées, consultez la section Ressources personnalisées dans la CloudFormation documentation. Pour un exemple d'utilisation d'une ressource personnalisée, consultez le référentiel de ressources personnalisées sur GitHub.

L'exemple suivant montre comment créer une classe de ressources personnalisée pour lancer une fonction Lambda et envoyer CloudFormation un signal de réussite ou d'échec.

```
import cdk = require('aws-cdk-lib');
import customResources = require('aws-cdk-lib/custom-resources');
import lambda = require('aws-cdk-lib/aws-lambda');
import { Construct } from 'constructs';

import fs = require('fs');

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

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

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

    const fn = new lambda.SingletonFunction(this, 'Singleton', {
      uuid: 'f7d4f730-4ee1-11e8-9c2d-fa7ae01bbebc',
```

```
      code: new lambda.InlineCode(fs.readFileSync('custom-resource-handler.py',
 { encoding: 'utf-8' })),
      handler: 'index.main',
      timeout: cdk.Duration.seconds(300),
      runtime: lambda.Runtime.PYTHON_3_6,
    });

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

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

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

L'exemple suivant montre la logique principale de la ressource personnalisée.

```
def main(event, context):
    import logging as log
    import cfnresponse
    log.getLogger().setLevel(log.INFO)

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

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

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

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

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

L'exemple suivant montre comment la AWS CDK pile appelle la ressource personnalisée.

```
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();
```

# Suivez les TypeScript meilleures pratiques

TypeScript est un langage qui étend les fonctionnalités de JavaScript. C'est un langage fortement typé et orienté objet. Vous pouvez l'utiliser TypeScript pour spécifier les types de données transmis dans votre code et vous pouvez signaler des erreurs lorsque les types ne correspondent pas. Cette section fournit une vue d'ensemble des TypeScript meilleures pratiques.

## Description de vos données

Vous pouvez l'utiliser TypeScript pour décrire la forme des objets et des fonctions de votre code. L'utilisation du type `any` équivaut à se désinscrire de la vérification du type pour une variable. Nous vous recommandons d'éviter d'utiliser `any` dans votre code. Voici un exemple.

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

## Utilisation d'énumérations

Vous pouvez utiliser des énumérations pour définir un ensemble de constantes nommées et définir des normes qui peuvent être réutilisées dans votre base de code. Nous vous recommandons d'exporter vos énumérations une seule fois au niveau global, puis de laisser les autres classes importer et utiliser les énumérations. Supposons que vous souhaitiez créer un ensemble d'actions possibles pour capturer les événements dans votre base de code. TypeScript fournit des énumérations numériques et basées sur des chaînes. L'exemple suivant utilise une énumération.

```
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)
```

## Utilisation d'interfaces

Une interface est un contrat pour la classe. Si vous créez un contrat, vos utilisateurs doivent le respecter. Dans l'exemple suivant, une interface est utilisée pour normaliser les `props` et veiller à ce que les appelants fournissent le paramètre attendu lors de l'utilisation de cette 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
})
```

Certaines propriétés ne peuvent être modifiées que lors de la création initiale d'un objet. Vous pouvez le spécifier en plaçant `readonly` avant le nom de la propriété, comme le montre l'exemple suivant.

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

## Extension d'interfaces

L'extension des interfaces réduit les doublons, car il n'est pas nécessaire de copier les propriétés entre les interfaces. En outre, le lecteur de votre code peut facilement comprendre les relations au sein de votre application.

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

## Évitement des interfaces vides

Nous vous recommandons d'éviter les interfaces vides en raison des risques potentiels qu'elles présentent. Dans l'exemple suivant, il existe une interface vide appeléeBucketProps. Les objets mys3Bucket1 et mys3Bucket2 sont tous deux valides, mais ils suivent des normes différentes, car l'interface n'applique aucun contrat. Le code suivant compilera et imprimera les propriétés, mais cela introduit des incohérences dans votre application.

```
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",
```

```
});
```

## Utilisation d'usines

Dans un modèle Abstract Factory, une interface est chargée de créer une usine d'objets connexes sans spécifier explicitement leurs classes. Par exemple, vous pouvez créer une usine Lambda pour créer des fonctions Lambda. Au lieu de créer une nouvelle fonction Lambda dans votre construction, vous déléguez le processus de création à l'usine. Pour plus d'informations sur ce modèle de conception, voir Abstract Factory TypeScript dans la Refactoring.Guru documentation.

## Utilisation de la déstructuration sur des propriétés

La déstructuration, introduite dans ECMAScript 6 (ES6), est une JavaScript fonctionnalité qui vous permet d'extraire plusieurs données d'un tableau ou d'un objet et de les attribuer à leurs propres variables.

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

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

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

## Définition des conventions de dénomination standard

L'application d'une convention de dénomination permet de maintenir la cohérence de la base de code et de réduire la surcharge lorsque vous réfléchissez à la manière de nommer une variable. Nous vous recommandons la procédure suivante :

- Utilisez CamelCase pour les noms de variables et de fonctions.
- Utilisez UPPER_CASE pour les constantes globales afin d'indiquer clairement les valeurs immuables au moment de la compilation.
- PascalCase À utiliser pour les noms de classe et les noms d'interface.
- Utilisez CamelCase pour les membres de l'interface.
- PascalCase À utiliser pour les noms de type et les noms d'énumération.

- Nommez les fichiers avec CamelCase (par exemple, `ebsVolumes.tsx` ou `storage.ts`).

Vous trouverez ci-dessous des exemples de ces conventions de dénomination recommandées :

```
// 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 { }
```

## N'utilisez pas le mot clé var

L'`let`instruction est utilisée pour déclarer une variable locale dans TypeScript. Il est similaire au `var` mot clé, mais sa portée est limitée par rapport au `var` mot clé. Une variable déclarée dans un bloc avec `let` est uniquement disponible en vue d'utilisation dans ce bloc. Le `var` mot clé ne peut pas être limité à un bloc, ce qui signifie qu'il est accessible en dehors d'un bloc particulier (représenté par`{}`) mais pas en dehors de la fonction dans laquelle il est défini. Vous pouvez redéclarer et mettre à jour les `var` variables. Il est recommandé d'éviter d'utiliser le `var` mot clé.

## Envisager d'utiliser ESLint et Prettier

ESLint analyse statiquement votre code pour détecter rapidement les problèmes. Vous pouvez utiliser ESLint pour créer une série d'affirmations (appelées règles de validation) qui définissent l'apparence ou le comportement de votre code. ESLint propose également des suggestions de correction automatique pour vous aider à améliorer votre code. Enfin, vous pouvez utiliser ESLint pour charger des règles de validation à partir de plug-ins partagés.

Prettier est un formateur de code reconnu qui prend en charge une variété de langages de programmation différents. Vous pouvez utiliser Prettier pour définir votre style de code afin d'éviter de

le formater manuellement. Après l'installation, vous pouvez mettre à jour votre fichier `package.json` et exécuter les commandes `npm run format` et `npm run lint`.

L'exemple suivant vous montre comment activer ESLint et le formateur Prettier pour votre projet. 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)'"
}
```

## Utilisation des modificateurs d'accès

Le modificateur privé dans TypeScript limite la visibilité à la même classe uniquement. Lorsque vous ajoutez le modificateur privé à une propriété ou à une méthode, vous pouvez accéder à cette propriété ou méthode au sein de la même classe.

Le modificateur public permet aux propriétés et aux méthodes des classes d'être accessibles depuis n'importe quel emplacement. Si vous ne spécifiez aucun modificateur d'accès pour les propriétés et les méthodes, ils utiliseront le modificateur public par défaut.

Le modificateur protégé permet aux propriétés et aux méthodes d'une classe d'être accessibles au sein de la même classe et des mêmes sous-classes. Utilisez le modificateur protégé lorsque vous prévoyez de créer des sous-classes dans votre AWS CDK application.

## Utiliser des types d'utilitaires

Les types d'utilitaires TypeScript sont des fonctions de type prédéfinies qui effectuent des transformations et des opérations sur des types existants. Cela vous permet de créer de nouveaux types basés sur des types existants. Par exemple, vous pouvez modifier ou extraire des propriétés, les rendre facultatives ou obligatoires, ou créer des versions immuables de types. En utilisant des types d'utilitaires, vous pouvez définir des types plus précis et détecter les erreurs potentielles au moment de la compilation.

### <Type partiel >

`Partial`marque tous les membres d'un type d'entrée `Type` comme facultatifs. Cet utilitaire renvoie un type qui représente tous les sous-ensembles d'un type donné. Voici un exemple de `Partial`.

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

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

### <Type requis >

`Required`fait le contraire de`Partial`. Cela rend tous les membres d'un type d'entrée `Type` non facultatifs (en d'autres termes, obligatoires). Voici un exemple 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
};
```

## Rechercher les vulnérabilités de sécurité et les erreurs de formatage

L'infrastructure en tant que code (IaC) et l'automatisation sont devenues essentielles pour les entreprises. Compte tenu de la solidité d'IaC, vous avez une grande responsabilité dans la gestion des risques de sécurité. Les risques de sécurité courants d'IaC peuvent inclure les suivants :

- Over-permissive Gestion des identités et des accès AWS Privilèges (IAM)
- Groupes de sécurité ouverts
- Ressources non chiffrées
- Journaux d'accès non activés

## Approches et outils de sécurité

Nous vous recommandons d'implémenter les approches de sécurité suivantes :

- Détection des vulnérabilités dans le développement : la correction des vulnérabilités en production est coûteuse et prend du temps en raison de la complexité du développement et de la distribution de correctifs logiciels. En outre, les vulnérabilités de la production comportent un risque d'exploitation. Nous vous recommandons d'analyser le code de vos ressources IaC afin de détecter et de corriger les vulnérabilités avant leur mise en production.
- Conformité et correction automatique : AWS Config propose des règles AWS gérées. Ces règles vous aident à renforcer la conformité et vous permettent de tenter une correction automatique en utilisant l'AWS Systems Manager automatisation. Vous pouvez également créer et associer des documents d'automatisation personnalisés à l'aide de AWS Config règles.

## Outils de développement courants

Les outils présentés dans cette section vous aident à étendre leurs fonctionnalités intégrées avec vos propres règles personnalisées. Nous vous recommandons d'aligner vos règles personnalisées sur les normes de votre organisation. Voici quelques outils de développement courants à prendre en compte :

- Utilisez cdk-nag pour vérifier que les constructions d'une portée donnée sont conformes à un ensemble de règles défini. Vous pouvez également utiliser cdk-nag pour la suppression de règles et les rapports de conformité. L'outil cdk-nag valide les constructions en étendant les aspects du. AWS CDK Pour plus d'informations, consultez la section Gérer la sécurité et la conformité des applications avec le Cloud Development Kit AWS (AWS CDK) et cdk-nag dans le blog. AWS DevOps
- Utilisez l'outil open source Checkov pour effectuer une analyse statique sur votre environnement IaC. Checkov aide à identifier les erreurs de configuration du cloud en scannant le code de votre infrastructure dans Kubernetes, Terraform ou. CloudFormation Vous pouvez utiliser Checkov pour

obtenir des sorties dans différents formats, notamment JSON, JUnit XML ou CLI. Checkov peut gérer efficacement les variables en créant un graphique qui montre la dépendance dynamique du code. Pour plus d'informations, consultez le référentiel GitHub Checkov de Bridgecrew.

- Utilisez TFLint pour vérifier les erreurs et les syntaxes obsolètes et pour vous aider à appliquer les bonnes pratiques. Notez qu'il se peut que TFLint ne valide pas les problèmes propres au fournisseur. Pour plus d'informations sur TFlint, consultez le référentiel GitHub TFlint de Terraform Linters.
- Utilisez Amazon Q Developer pour effectuer des analyses de sécurité. Lorsqu'il est utilisé dans un environnement de développement intégré (IDE), Amazon Q Developer fournit une assistance au développement AI-powered logiciel. Il peut discuter du code, fournir des compléments de code en ligne, générer du nouveau code, scanner votre code pour détecter les failles de sécurité et apporter des mises à niveau et des améliorations au code.

## Développer et affiner la documentation

La documentation est essentielle à la réussite de votre projet. Non seulement la documentation explique le fonctionnement de votre code, mais elle aide également les développeurs à mieux comprendre les caractéristiques et les fonctionnalités de vos applications. Le développement et le perfectionnement d'une documentation de haute qualité peuvent renforcer le processus de développement logiciel, maintenir des logiciels de haute qualité et faciliter le transfert de connaissances entre les développeurs.

Il existe deux catégories de documentation : la documentation contenue dans le code et la documentation associée relative au code. La documentation contenue dans le code se présente sous forme de commentaires. La documentation associée relative au code peut se composer des fichiers README et des documents externes. Il n'est pas rare que les développeurs considèrent la documentation comme une surcharge, car le code lui-même est facile à comprendre. Cela peut être vrai pour les petits projets, mais la documentation est cruciale pour les projets de grande envergure impliquant plusieurs équipes.

Il est recommandé que l'auteur du code rédige la documentation, car il en comprend bien les fonctionnalités. Les développeurs peuvent avoir du mal à supporter la surcharge liée à la maintenance de la documentation associée distincte. Pour relever ce défi, les développeurs peuvent ajouter les commentaires dans le code et ces commentaires peuvent être extraits automatiquement afin que chaque version du code et de la documentation soit synchronisée.

Il existe différents outils pour aider les développeurs à extraire les commentaires du code et à générer la documentation correspondante. Ce guide se concentre sur le fait TypeDoc qu'il s'agit de l'outil préféré pour AWS CDK les constructions.

## Pourquoi la documentation du code est-elle requise pour AWS CDK construit

AWS CDK les concepts communs sont créés par plusieurs équipes au sein d'une organisation et partagés entre différentes équipes à des fins d'utilisation. Une bonne documentation aide les utilisateurs de la bibliothèque de constructions à intégrer facilement les constructions et à créer leur infrastructure avec un minimum d'effort. La synchronisation de tous les documents est une tâche conséquente. Nous vous recommandons de conserver le document dans le code, qui sera extrait à l'aide de la TypeDoc bibliothèque.

## En utilisant TypeDoc avec AWS Construire une bibliothèque

TypeDoc est un générateur de documents pour TypeScript. Vous pouvez l'utiliser TypeDoc pour lire vos fichiers TypeScript source, analyser les commentaires contenus dans ces fichiers, puis générer un site statique contenant la documentation de votre code.

Le code suivant vous montre comment intégrer TypeDoc la bibliothèque AWS Construct, puis ajouter les packages suivants dans votre `package.json` fichier`devDependencies`.

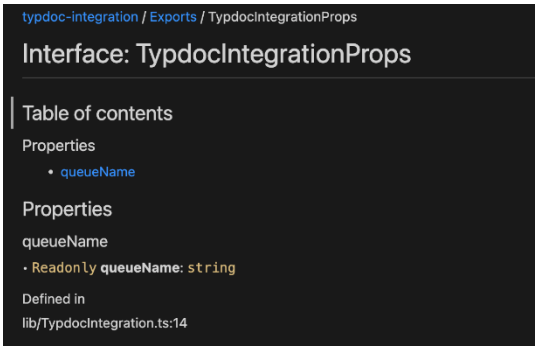
```
{

  "devDependencies": {

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

}
```

Pour ajouter `typedoc.json` dans le dossier de la bibliothèque CDK, utilisez le code suivant.

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

Pour générer les fichiers README, exécutez la `npx typedoc` commande dans le répertoire racine du projet de bibliothèque de AWS CDK construction.

L'exemple de document suivant est généré par TypeDoc.




Pour plus d'informations sur les options TypeDoc d'intégration, consultez les commentaires relatifs aux documents dans la TypeDoc documentation.

## Adopter une approche de développement piloté par les tests

Nous vous recommandons de suivre une approche de développement piloté par les tests (TDD) avec le. AWS CDK Le TDD est une approche du développement logiciel dans laquelle vous développez des cas de test pour spécifier et valider votre code. En résumé, vous créez d'abord des cas de test pour chaque fonctionnalité et si le test échoue, vous écrivez le nouveau code pour transmettre le test, le simplifier et le rendre exempt de bogues.

Vous pouvez d'abord utiliser l'approche de TDD pour écrire le scénario de test. Cela vous permet de valider l'infrastructure avec différentes contraintes de conception en termes d'application de la politique de sécurité pour les ressources et de respect d'une convention de dénomination unique pour le projet. L'approche standard pour tester AWS CDK les applications consiste à utiliser le module AWS CDK assertions et les frameworks de test populaires, tels que Jest pour JavaScript et/ou TypeScript pytest pour Python.

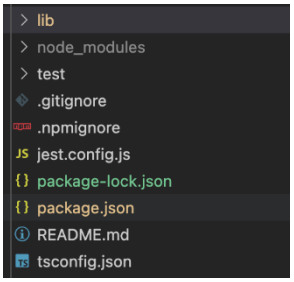
Il existe deux catégories de tests que vous pouvez rédiger pour vos AWS CDK applications :

- Utilisez des assertions détaillées pour tester un aspect spécifique du CloudFormation modèle généré, par exemple « cette ressource possède cette propriété avec cette valeur ». Ces tests peuvent détecter des régressions et sont également utiles lorsque vous développez de nouvelles fonctionnalités à l'aide de TDD (écrivez d'abord un test, puis faites-le réussir en écrivant une implémentation correcte). Fine-grained les assertions sont les tests que vous allez le plus écrire.

- Utilisez des tests instantanés pour tester le modèle synthétisé par rapport à un CloudFormation modèle de référence précédemment stocké. Les tests d'instantanés permettent de refactoriser librement, car vous pouvez être sûr que le code refactorisé fonctionne exactement de la même manière que le code d'origine. Si les modifications étaient intentionnelles, vous pouvez accepter une nouvelle référence pour les futurs tests. Toutefois, les AWS CDK mises à niveau peuvent également entraîner la modification des modèles synthétisés. Vous ne pouvez donc pas vous fier uniquement aux instantanés pour vous assurer que votre implémentation est correcte.

## Test unitaire

Ce guide se concentre sur l'intégration des tests unitaires en TypeScript particulier. Pour activer les tests, assurez-vous que votre `package.json` fichier possède les bibliothèques suivantes :`@types/jest`,`jest`, et `ts-jest` in`devDependencies`. Pour ajouter ces packages, exécutez la commande `cdk init lib --language=typescript`. Une fois que vous avez exécuté la commande précédente, la structure suivante s'affiche.




Le code suivant est un exemple de `package.json` fichier activé avec la bibliothèque 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",
    ...
  }
```

```
}
```

Sous le dossier Test, vous pouvez écrire le cas de test. L'exemple suivant montre un cas de test pour une AWS CodePipeline construction.

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

Pour exécuter un test, exécutez la commande `npm run test` dans votre projet. Le test renvoie les résultats suivants.

```
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
```

Pour plus d'informations sur les scénarios de test, consultez la section Tester des constructions dans le Guide du Cloud Development Kit AWS (AWS CDK) développeur.

## Test d'intégration

Les tests d'intégration pour AWS CDK les constructions peuvent également être inclus à l'aide d'un `integ-tests` module. Un test d'intégration doit être défini comme une AWS CDK application. Il doit y avoir une relation biunivoque entre un test d'intégration et une AWS CDK application. Pour plus d'informations, consultez le module integ-tests-alpha dans la référence de l'API.AWS CDK

# Utiliser le contrôle des éditions et des versions pour les constructions

## Contrôle de version pour AWS CDK

AWS CDK des concepts communs peuvent être créés par plusieurs équipes et partagés au sein d'une organisation à des fins de consommation. Généralement, les développeurs publient de nouvelles fonctionnalités ou corrigent des bogues dans leurs AWS CDK constructions communes. Ces constructions sont utilisées par les AWS CDK applications ou par toute autre AWS CDK construction existante dans le cadre d'une dépendance. Pour cette raison, il est crucial que les développeurs mettent à jour et publient indépendamment leur construction avec des versions sémantiques appropriées. AWS CDK Les applications en aval ou d'autres AWS CDK constructions peuvent mettre à jour leur dépendance pour utiliser la version de AWS CDK construction récemment publiée.

La gestion des versions sémantique (Semver) est un ensemble de règles, ou de méthodes, permettant de fournir des numéros de logiciel uniques au logiciel informatique. Les versions sont définies comme suit :

- Une version MAJEURE consiste en des modifications d'API incompatibles ou en une modification radicale.
- Une version MINEURE comprend des fonctionnalités ajoutées de manière rétrocompatible.
- Une version de CORRECTIF consiste en des corrections de bogues rétrocompatibles.

Pour plus d'informations sur le versionnement sémantique, consultez la section Spécification de version sémantique (SemVer) dans la documentation sur le versionnage sémantique.

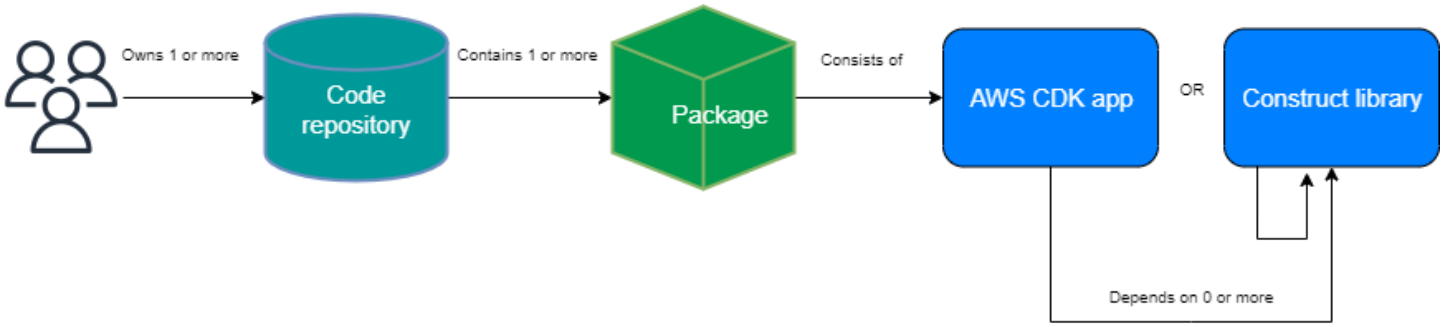
# Référentiel et empaquetage pour AWS CDK construit

Comme AWS CDK les constructions sont développées par différentes équipes et utilisées par plusieurs AWS CDK applications, vous pouvez utiliser un référentiel distinct pour chaque AWS CDK construction. Cela peut également vous aider à appliquer le contrôle d'accès. Chaque dépôt peut contenir tout le code source lié à la même AWS CDK construction ainsi que toutes ses dépendances. En conservant une seule application (c'est-à-dire une AWS CDK construction) dans un référentiel unique, vous pouvez réduire l'impact des modifications lors du déploiement.

AWS CDK Non seulement il génère des CloudFormation modèles pour le déploiement de l'infrastructure, mais il regroupe également les actifs d'exécution tels que les fonctions Lambda et les images Docker et les déploie parallèlement à votre infrastructure. Il est non seulement possible de combiner le code qui définit votre infrastructure et le code qui implémente votre logique d'exécution en une seule construction, mais c'est également une bonne pratique. Ces deux types de code n'ont pas besoin de se trouver dans des référentiels séparés ni même dans des packages distincts.

Pour consommer des packages au-delà des limites du référentiel, vous devez disposer d'un référentiel de packages privé, similaire à npm ou à Maven Central PyPi, mais interne à votre organisation. Vous devez également disposer d'un processus de publication qui génère, teste et publie le package dans le référentiel de packages privé. Vous pouvez créer des référentiels privés, tels que des PyPi serveurs, à l'aide d'une machine virtuelle (VM) locale ou d'Amazon S3. Lorsque vous concevez ou créez un registre de packages privé, il est essentiel de prendre en compte le risque d'interruption de service en raison de la haute disponibilité et de la capacité de mise à l'échelle. Un service géré sans serveur hébergé dans le cloud pour stocker les packages peut considérablement réduire les frais de maintenance. Par exemple, vous pouvez l'utiliser AWS CodeArtifactpour héberger des packages pour les langages de programmation les plus courants. Vous pouvez également l'utiliser CodeArtifact pour définir des connexions au référentiel externe et les répliquer au sein CodeArtifact de celui-ci.

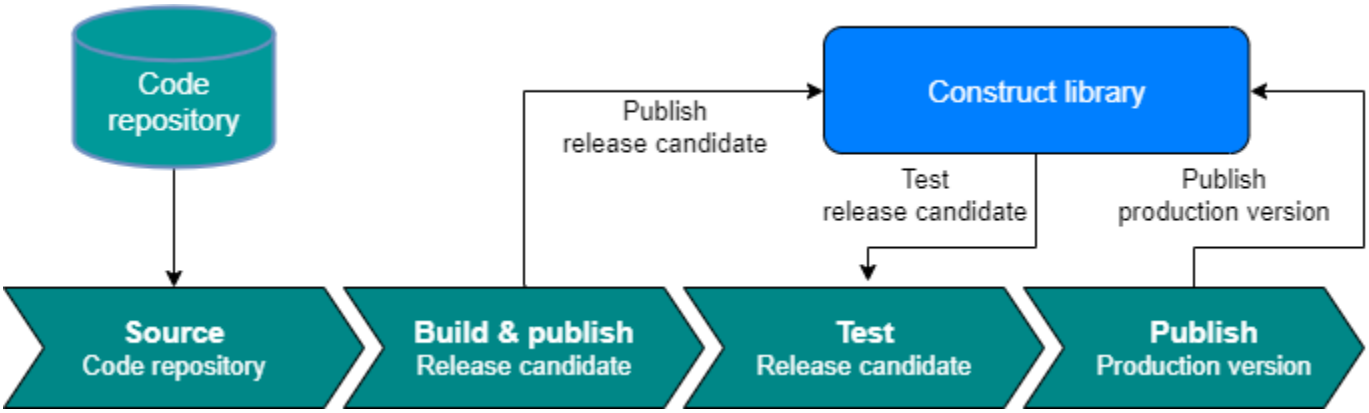
Les dépendances des packages du référentiel de packages sont gérées par le gestionnaire de packages de votre langue (par exemple, npm for TypeScript ou JavaScript applications). Votre gestionnaire de packages s'assure que les générations sont reproductibles en enregistrant les versions spécifiques de chaque package dont dépend votre application, puis vous permet de mettre à niveau ces dépendances de manière contrôlée, comme le montre le schéma suivant.




## Constructez la publication pour le AWS CDK

Nous vous recommandons de créer votre propre pipeline automatisé pour créer et publier de nouvelles versions de AWS CDK construction. Si vous mettez en place un processus d'approbation des demandes d'extraction approprié, une fois que vous avez validé et transféré votre code source dans la branche principale du référentiel, le pipeline peut générer et créer une version possible. Cette version peut être transférée CodeArtifact et testée avant de publier la version prête pour la production. Vous pouvez éventuellement tester votre nouvelle version de AWS CDK construction localement avant de fusionner le code avec la branche principale. Ainsi, le pipeline va publier la version prête pour la production. Tenez compte du fait que les constructions et les packages partagés doivent être testés indépendamment de l'application utilisatrice, comme s'ils étaient mis à la disposition du public.

Le schéma suivant montre un exemple de pipeline de publication de AWS CDK versions.




Vous pouvez utiliser les exemples de commandes suivants pour générer, tester et publier des packages npm. Tout d'abord, connectez-vous au référentiel d'artefacts en exécutant la commande suivante.

```
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>
```

Ensuite, procédez comme suit :

1. Installez les packages requis sur la base du fichier `package.json` : `npm install`
2. Créez la version possible pour la publication : `npm version prerelease --preid rc`
3. Générez le package npm : `npm run build`
4. Testez le package npm : `npm run test`
5. Publiez le package npm : `npm publish`

## Appliquer la gestion des versions de la bibliothèque

La gestion du cycle de vie représente un défi majeur lorsque vous gérez des bases de AWS CDK code. Supposons, par exemple, que vous démarriez un AWS CDK projet avec la version 1.97, puis que la version 1.169 soit disponible ultérieurement. La version 1.169 propose de nouvelles fonctionnalités et des corrections de bogues, mais vous avez déployé votre infrastructure à l'aide de l'ancienne version. Aujourd'hui, la mise à jour des constructions devient complexe, car cet écart augmente en raison des modifications majeures qui pourraient être introduites dans les nouvelles versions. Cela peut s'avérer compliqué si vous disposez de nombreuses ressources dans votre environnement. Le modèle présenté dans cette section peut vous aider à gérer la version de votre AWS CDK bibliothèque à l'aide de l'automatisation. Voici le flux de travail de ce modèle :

1. Lorsque vous lancez un nouveau produit CodeArtifact Service Catalog, les versions de la AWS CDK bibliothèque et ses dépendances sont stockées dans le `package.json` fichier.
2. Vous déployez un pipeline commun qui assure le suivi de tous les référentiels afin de pouvoir leur appliquer des mises à niveau automatiques en l'absence de modifications importantes.
3. Une AWS CodeBuild étape vérifie la présence de l'arbre de dépendances et recherche les modifications les plus importantes.
4. Le pipeline crée une branche de fonctionnalités, puis exécute `cdk synth` avec la nouvelle version pour confirmer qu'il n'y a aucune erreur.
5. La nouvelle version est déployée dans l'environnement de test et exécute enfin un test d'intégration pour s'assurer que le déploiement est sain.
6. Vous pouvez utiliser deux files Amazon Simple Queue Service (Amazon SQS) pour suivre les piles. Les utilisateurs peuvent examiner les piles manuellement dans la file des exceptions et

corriger les modifications importantes. Les éléments qui réussissent le test d'intégration peuvent être fusionnés et publiés.

# FAQ

## Quels problèmes peuvent être TypeScript résolus ?

Généralement, vous pouvez éliminer les bogues dans votre code en écrivant des tests automatisés, en vérifiant manuellement que le code fonctionne comme prévu, puis en demandant à une autre personne de valider votre code. La validation des connexions entre chaque partie de votre projet est une tâche chronophage. Pour accélérer le processus de validation, vous pouvez utiliser un langage de type vérifié, par exemple TypeScript pour automatiser la validation du code et fournir un feedback instantané pendant le développement.

## Pourquoi devrais-je l'utiliser TypeScript ?

TypeScript est un langage open source qui simplifie le JavaScript code, le rendant plus facile à lire et à déboguer. TypeScript fournit également des outils JavaScript IDEs et des pratiques de développement hautement productifs, tels que la vérification statique. De plus, TypeScript offre les avantages du ECMAScript 6 (ES6) et peut augmenter votre productivité. Enfin, cela TypeScript peut vous aider à éviter les bogues pénibles que les développeurs rencontrent fréquemment lors de l'écriture JavaScript en vérifiant le type du code.

## Dois-je utiliser le AWS CDK ou CloudFormation ?

Nous vous recommandons d'utiliser le Cloud Development Kit AWS (AWS CDK) plutôt que AWS CloudFormation, si votre organisation possède l'expertise en développement nécessaire pour tirer parti du AWS CDK. C'est parce que AWS CDK c'est plus flexible que CloudFormation, puisque vous pouvez utiliser un langage de programmation et des concepts OOP. N'oubliez pas que vous pouvez utiliser CloudFormation pour créer AWS des ressources de manière ordonnée et prévisible. Dans CloudFormation, les ressources sont écrites dans des fichiers texte au format JSON ou YAML.

## Et si le AWS CDK ne prend pas en charge un nouveau lancement Service AWS ?

Vous pouvez utiliser une dérogation brute ou une ressource CloudFormation personnalisée.

# Quels sont les différents langages de programmation pris en charge par le AWS CDK ?

AWS CDK est généralement disponible en Python JavaScript TypeScript, Java, C# et Go (dans Developer Preview).

# Combien cela AWS CDK coûte-t-il ?

Il n'y a aucun frais supplémentaire pour le AWS CDK. Vous payez pour les AWS ressources (telles que les instances Amazon EC2 ou les équilibreurs de charge Elastic Load Balancing) créées lorsque vous les utilisez AWS CDK de la même manière que si vous les créiez manuellement. Vous ne payez qu'à la demande et à l'utilisation. Aucun frais minimum ni aucun engagement initial requis ne s'appliquent.

# Étapes suivantes

Nous vous recommandons de commencer à construire avec Cloud Development Kit AWS (AWS CDK) TypeScript. Pour plus d'informations, consultez l'atelier AWS CDK d'une journée d'immersion.

# Ressources

## Références

- AWS Constructions de AWS solutions (solutions)
- Cloud Development Kit AWS (AWS CDK) (GitHub)
- AWS Référence de l'API Construct Library (documentation de AWS CDK référence)
- AWS CDK Documentation de référence (documentation AWS CDK de référence)
- AWS CDK Atelier d'une journée d'immersion (AWS Workshop Studio)

## Outils

- sac cdk () GitHub
- TypeScript ESLint(TypeScript ESLint documentation)

## Guides et modèles

- AWS Solutions Construit des modèles (AWS documentation)

# Historique du document

Le tableau suivant décrit les modifications importantes apportées à ce guide. Pour être averti des mises à jour à venir, abonnez-vous à un fil RSS.

| Modification | Description | Date |
| --- | --- | --- |
| Mise à jour des bonnes pratiques | Nous avons supprimé la recommandation d'utiliser cfn-nag dans la section Outils de développement courants. Dans la section Définir les conventions de dénomination standard, nous avons ajouté des conventions de dénomination standard pour les constantes globales et ajouté des exemples de dénomination. | 23 octobre 2025 |
| Code de mise à jour | Nous avons mis à jour les exemples de code dans la section Suivre les TypeScript meilleures pratiques. | 16 février 2024 |
| Ajouter des sections | Nous avons ajouté les sections Utiliser les types d'utilitaires et Test d'intégration. | 10 janvier 2024 |
| Mise à jour mineure | Exemple de code mis à jour pour créer une construction L3. | 16 juin 2023 |
| Publication initiale | — | 8 décembre 2022 |

Les traductions sont fournies par des outils de traduction automatique. En cas de conflit entre le contenu d'une traduction et celui de la version originale en anglais, la version anglaise prévaudra.