THE CDK BOOK

BHAT, BONIG, COULTER & HOEGER

The CDK Book

Sathyajith Bhat, Matthew Bonig, Matt Coulter, Thorsten Hoeger

Version v1.2.1, 2022-01-24

Table of Contents

Foreword

A while back, a notification popped up in the community-led cdk.dev Slack. This is where thousands of teams from across the globe are helping each other with anything related to the CDK, the *Cloud Development Kit*. The private message said, "A group of us are teaming up to write a book about the AWS CDK, and we would love for you to write the foreword."

It's been over four years since we started the CDK journey. We set out to improve the experience of developing cloud applications, but the thing that I'm truly humbled by is the community that formed organically around this mission. A community that is not only about the AWS CDK as a specific product, but around the CDK as a programming model for creating abstractions by defining desired state through software. This programming model, which we sometimes refer to as the Construct Programming Model (CPM), is used today to define applications on Kubernetes with CDK8s, Terraform with CDKTF, Azure with Armkit, and even things like complete development experiences with Projen.

Our community is ready for a book, and I'm honored to write this foreword and share my perspective. To start with, I'd like to take you back in time and tell you the origin story of the AWS CDK.

In early 2016, I joined a group of Amazon engineers who were exploring hardware utilization in one of the most important and large-scale services behind Amazon.com—Amazon Product Search. One piece of this solution required processing, in real time, all activity from Amazon.com to determine which products were being purchased. Amazon was in the midst of a company-wide program to enable internal teams to use AWS to its full extent. Back then, many of our services were deployed in a centrally managed AWS account within a single network. This new project gave us an opportunity to use some of the latest AWS building blocks. The service had to serve Amazon-scale production traffic across multiple marketplaces from day one, so we devised an architecture that heavily relied on AWS Lambda,

Amazon Kinesis, and Amazon DynamoDB to be able to process millions of events per day with minimal operational costs.

We kicked off a two-pizza team (an Amazon term for a small, focused, highly cooperative team) with the responsibility to deliver this service. Given its reliance on many AWS resources, and our need to deploy the service across development, integration, and production environments, we decided to use AWS CloudFormation to provision our resources. It wasn't reasonable to manually configure all these resources through the AWS Management Console, and we wanted to capture our resource configuration in source control so we could incrementally iterate on it through our normal development process.

We started simple with a bunch of AWS Lambda functions, but as the project progressed, our architecture evolved, and we used more and more AWS resources and capabilities. It became increasingly harder to maintain our CloudFormation templates. Although the CloudFormation YAML files represented the configuration of our resources, they weren't designed to capture the *ideas* behind our architecture, such as logical units, relationships, best practices, and repeating patterns. We found ourselves copying and pasting too much. We constantly forgot to update certain values across multiple resources and templates, and we struggled to find good ways to validate our code with tests before it was deployed.

As software engineers, we've been solving this exact set of problems using object-oriented programming languages since the late 1960s. In computer programs, logic can be expressed with imperative code, conditions, and loops. Abstract ideas can be modeled through object-oriented primitives, and best practices and patterns can be shared and reused through libraries and package managers. We wanted the best of both worlds—we wanted to be able to define our infrastructure using modern programming languages, but still provision our cloud resources through a declarative desired-state engine that took care of updating our infrastructure in a safe and deterministic way.

We experimented with some existing tools for generating CloudFormation

from code. There were a few open-source projects like Troposphere and GoFormation that generated CloudFormation, but they lacked one basic ingredient—composability. Composability is the key to software abstraction because it allows solving problems by breaking them down into smaller problems. If we wanted to simplify the cloud, we needed to be able to create composable, reusable abstractions.

Furthermore, we wanted a solution in a programming language that our engineers would be comfortable using, and that would allow us to take advantage of the investment we already had in our development environment and processes. We didn't want to introduce a new toolchain into our build process, figure out the best IDE setup, and learn new programming patterns and idioms. We also wanted to use the same programming language to natively connect between our infrastructure code and our runtime code. We realized this was particularly important for serverless applications like ours because our AWS Lambda handlers directly interacted with many infrastructure resources. More and more we realized that infrastructure and runtime code are two sides of the same coin, and we wanted to develop, test, and release them together.

We created a library called "Cloudstruct." At the heart of this library were composable primitives we named constructs. Constructs were simple object-oriented classes that represented cloud building blocks and could be combined to form higher-level constructs. At the base level, each AWS resource was a construct, so we had a construct for "Amazon S3 Bucket," "AWS Lambda Function," "Amazon DynamoDB Table," and so forth. Then, we could combine these together to represent logical units in our system. So, we had an "ingestion" construct that represented our incoming data pipeline, a "publisher" construct that took care of publishing results to the downstream search index, and a nice little "dynamo scanner" construct that performed a daily full, concurrent scan of our Amazon DynamoDB table. When a Cloudstruct program was executed, it would synthesize a set of CloudFormation templates that we could examine and deploy to our accounts.

As soon as we started to migrate our application to Cloudstruct, everything clicked into place. This paradigm felt right, and solutions emerged and popped up naturally. A few months later, after we shipped our service to production, we demonstrated Cloudstruct to AWS leadership along with an internal "press release" document that described a vision for releasing it as a public AWS product in multiple programming languages. It was an easy pitch. We just needed to show examples from constructs we created for our service alongside the CloudFormation templates that they generated. Comparing a few easy-to-understand lines of code against hundreds of lines of YAML was all it took to get initial funding to build the AWS Cloud Development Kit.

And here we are, four years later. You are holding a book that is a practitioner's guide to using the AWS Cloud Development Kit. It offers guidance, best practices, and advice written by prominent members of the CDK community who have been involved with the project from its pre-1.0 days. This book builds on experience from real production use cases across dozens of different types of customers. It's based on learnings, failures, and respectful discussions across the growing CDK community.

I see the CDK community and ecosystem as our real success. However, even with millions of CDK stacks deployed, adoption from across the industry, three major CDK products (AWS CDK, CDK for Terraform, and CDK for Kubernetes), and hundreds of construct libraries listed in Construct Hub, I truly think we are just getting started. Building and operating cloud applications is still too complex: Developers and operators are still required to have a deep understanding of the low-level pieces. Managing the end-to-end development experience still involves stitching together dozens of tools and services every time, and it is still almost impossible to reuse construct-based abstractions across different provisioning domains.

As I look forward in the waning days of 2021, I'd like to share my perspective on how the CDK could continue to improve the experience of developing and operating cloud applications. I believe that we will see evolution in four different directions: more L3 constructs (**up**), cross-domain interoperability

(**down**), using constructs as "meta-IDEs" (**left**), and runtime representation (**right**).

Up: I believe we are going to see more and more truly high-level abstractions, or as we sometimes call them in CDK parlance, *L3 constructs*. The CDK offers a powerful programming model for simplifying the cloud, but so far most of the simplification has been up-leveling the API experience for individual AWS resources (what we call L2s). The "S3 Bucket" construct offers a rich, intent-based API for buckets, but it still requires users to understand what a bucket is. On the other hand, a "Static Website" construct offers a higher-level mental model: As long as users can wrap their heads around the concept of a static website, they don't need to care about the underlying implementation. It shouldn't matter if behind the scenes there is an Amazon S3 bucket and Amazon CloudFront distributions or Amazon API Gateway and AWS Lambda functions. Opportunities for new high-level mental models like this are abundant— microservice frameworks, big-data analytics, compliance patterns, regulatory constraints, line-of-business applications, and machine learning pipelines. We can already find many useful examples in the Construct Hub today, but I believe this is just the tip of the iceberg—there are still many high-level ideas across our industry waiting for you to codify them through constructs.

Down: There is still strong coupling between constructs and the underlying provisioning mechanism. To maximize the investment in high-level abstractions we will need cross-domain interoperability. For example, users will be able to define infrastructure using AWS L2 constructs within their CDK for Terraform or CDK for Kubernetes applications. This means that construct-based abstractions will no longer be confined to a specific provisioning domain, and they will be more broadly usable and valuable for more users. There is already initial work in this space, such as AWS CDK L2 support in CDKTF and CDK8s support in AWS CDK, but I believe we will see more of this decoupling and increased choice happening in the coming years.

Left: The constructs programming model can be used to codify full

development experiences. Constructs are not limited to synthesizing infrastructure declarations. They can also generate artifacts such as compiler and test setups, release pipelines, dependency upgrade policies, issue workflows and editor settings. This means that we can use constructs to simplify development experiences. We've been exploring this direction with some very interesting results through the Projen incubation project. One can think of this approach as a "*meta-IDE*"—a programming model for producing integrated development experiences. As these ideas mature, we will be able to address more and more development experience challenges through a common modeling space. Being able to create constructs that encompass cloud infrastructure, application logic, and developer ergonomics can open up whole new possibilities in software development.

Right: Construct-based abstractions are leaky at runtime. Today, constructs are primarily a design-time abstraction—they hide complexity when writing code. But what happens after a construct is deployed? The abstraction is lost, and operators are left to deal with a bunch of resources. If a developer uses the "Static Website" construct in their app they are not required to know how static websites are implemented. However, after their website is deployed, it stops being a website and becomes an Amazon S3 bucket, Amazon CloudFront distribution, and Amazon Route 53 hosted zone—the abstraction does not carry over. Solving this challenge is critical. If we want to raise the abstraction level, we must find ways to interact with these abstractions after they are deployed. We are exploring a few ideas on how to retain the fidelity of the construct tree after an app is deployed, and I'm excited by the experiences we will be able to offer with this information.

I would like to take this opportunity to thank our users, open-source contributors, and the amazing CDK team at AWS. I am proud, inspired, and humbled by this inclusive and welcoming community, which shares a relentless passion for moving the cloud industry forward through delightful development experiences. I also want to thank the authors for publishing this book. I couldn't imagine a more suitable group to collaborate on such an endeavor.

At Amazon we like to say, “It’s always day one!” It definitely feels like that with the CDK today, even though, as this book makes clear, we’ve come a long way. There are substantial challenges ahead, but there are also exciting opportunities to change how software is created through simplification and abstraction. I am excited that you decided to join the ride, and I hope this book will become a valuable tool in your journey to build better software.

Elad Ben-Israel
Principal Software Engineer
AWS Developer Tools
November 2021, Tel-Aviv

Special Thanks

A special thanks to some people who helped in the production of this book:

- Edwin Radtke (https://edwinradtke.com)
- Chase Douglas (https://twitter.com/txase)
- Sebastian Korfmann (https://twitter.com/skorfmann)
- Alex DeBrie (https://twitter.com/alexbdebrie)

This Book's Examples

When we got together to write this book, we decided we wanted to support languages other than just Typescript, but knew that writing all the examples in multiple languages manually would be a lengthy process.

Thankfully, jsii-rosetta made this easy, not only for Python, the second most popular CDK language, but C# and Java too. Go support will hopefully come in the near future and when it does, we'll provide the Go version of the book to everyone that has purchased it.

If you haven't heard of jsii, I recommend you go check it out real quick and then come back. It is a crucial element to the success of the CDK. It underpins the entire CDK, and we wanted to use it for handling our examples in this book.

The jsii, being open source and well-documented, a script was written that took examples written in Typescript throughout the book and converted them to the other supported languages of the CDK.

There are some places, like explaining Python-specific libraries, where the example is only available in one language. Also, examples of Lambda function handler won't be converted.

Some conversions to other language may not be standard or conform to your coding practices, but should always be compilable. If we're wrong, please let us know on Twitter at @thecdkbook.

It's not a perfect system, but between this, not supporting other languages, or spending many months replicating examples across other languages, this seemed like the best option.

We apologize for anything that is wrong and has led you astray. We'll update the book as possible and post updates on Issues on the GitHub page.

In addition to the examples appearing in the book, there is a repository of examples on Github. This will expand over time. Translations of those examples to languages other than Typescript wasn't possible in time for publishing, but something we plan on adding later, if workable, with jsii.

About The Authors

Sathyajith Bhat
Sathyajith Bhat is an AWS Community Hero, the author of Practical Docker with Python (first edition, second edition - coming soon) and is currently working as a Site Reliability Engineer. He started his career as a database programmer and pivoted to DevOps and Site Reliability Engineering. While not running communities and organizing AWS user group meets, Sathyajith spends his free time playing games and streaming on Twitch.

Matthew Bonig
Matthew Bonig is an AWS Dev Tools Hero, certified DevOps Engineer and AWS enthusiast. He has a background in application development. He co-hosts the CDK Day virtual conference and contributes in the community with the cdk.dev website, Slack server, and the CDK Weekly newsletter.

When not writing CDK code he's hiking the Rockies or playing with dogs.

Matt Coulter
Matt is an AWS DevTools Hero, Serverless Architect and conference speaker. You can usually find him sharing reusable, well architected, serverless patterns over at cdkpatterns.com or behind the scenes bringing CDK Day virtual conference to life. Outside the cloud, Matt loves low & slow BBQ, travelling to new countries and just making time for his friends.

Thorsten Höger
Thorsten Höger is an AWS DevTools Hero, AWS consultant and automation evangelist. He started his career as a software engineer. He is also the maintainer of several open-source projects and one of the top three non-AWS contributors to the AWS CDK. In his spare time, he enjoys indoor climbing and cooking.

1. What is the Cloud Development Kit?

The AWS Cloud Development Kit (CDK) is an open-source software development framework that lets us define our cloud resources using programming languages we are familiar with. With the CDK, we can build reusable, composable patterns of infrastructure. The currently supported languages are Python, TypeScript, JavaScript, Java, C#. Go support is currently in developer preview. Writing code for the infrastructure yields benefits from a traditional software development life cycle:

- There is history and evolution of code, letting us examine and look back at why/when we did a change.
- We can develop and ship infrastructure just like software. Using code reviews and testing, we can improve the reliability and robustness of the code that is deploying the infrastructure.
- Infrastructure and applications are deployed using CI/CD pipelines, allowing for a rapid feedback loop.

1.1. Concepts

The CDK is built around some key concepts. We will do a deep dive into these concepts in subsequent chapters, but for now, let's understand these concepts at a high level.

A CDK application (*app*) is written in one of the many languages supported by the CDK. The application describes the desired AWS infrastructure to be built. An app comprises one or many stacks. A *stack* is equivalent to a CloudFormation stack - a collection of related resources. Stacks can comprise one or more constructs. A *construct* is a CDK representation of one or more cloud resources. A construct defines the most basic unit of our CDK application. A construct is as narrowly scoped to a specific cloud resource (for example, an S3 bucket) or can describe broader collections of resources

and can encapsulate architecture patterns. A construct is categorized into different levels, depending on the construct:

- L1 Construct - L1 constructs represent a CloudFormation resource.
- L2 (aka Curated) Construct - L2 constructs are developed by the CDK project to create certain resources and perform common operations.
- L3 (aka Pattern) Construct - L3 constructs are a collection of constructs that define entire applications or organization patterns.

1.2. Behind the scenes

The CDK doesn't reinvent the wheel and define a new programming language or a DSL [1] to perform operations. Instead, the CDK uses CloudFormation as its foundation. The applications, stacks, and constructs are written using common programming languages. The CDK synthesizes these into CloudFormation templates to bring up the required resources. CDK Toolkit is the CLI tool that is used to perform these operations. CDK Toolkit can perform actions like synth, deploy and show differences. Writing a software development framework that supports multiple languages is no small feat and is almost impossible to be done manually. For this reason, the CDK team devised a strategy to write the CDK libraries in a single language and generate bindings for other languages automatically. jsii is the technology that enabled the CDK team to deliver support for many languages from a single codebase.

jsii allows for code in any language to interact with JavaScript classes. The jsii compiler extracts the class library APIs from the source modules and generates library bindings in the supported languages. These libraries include the original JavaScript library. When we write a CDK application using the generated packages/libraries and invoke a method of the generated library, the method calls are routed to the embedded JavaScript library code to perform the required operation. This allows for consistent behavior across

the supported languages. For this reason, CDK requires the Node.js JavaScript runtime to be available, even if you are using a language like Python to write your CDK code.

1.3. Comparing the CDK with the other IaC tools

While trying to adopt a new language or framework, comparisons are natural. One of the frequent questions that pop up about CDK is what is the difference between CDK and the rest. Let's find out more in this section.

1.3.1. CDK vs CloudFormation

AWS CloudFormation is AWS' native Infrastructure as Code (IaC) service. CloudFormation lets us model cloud resources declaratively, using either JSON or YAML templates. We can provide a list of template parameters to a CloudFormation template that act as input parameters to allow for customization of the deployed stack. When we submit the CloudFormation template to AWS CloudFormation, the CloudFormation service reads the template, parses it, and will start provisioning the required resources. CloudFormation manages the state of the deployment internally. Any changes to the template, depending on the type of resource, are applied as an inline update or a complete teardown and creation as a new resource. In case of any errors, CloudFormation does a rollback to the last working state.

Getting started with CloudFormation has a learning curve - JSON/YAML templates can get painful, even for simple operations. With CDK, we can represent the infrastructure using native programming constructs, and not templatized JSON/YAML representation of the required infrastructure. Since CDK uses CloudFormation, we get the benefits of CloudFormation's robust state management without having to worry about messy templates. With Constructs being a core part of our application, CDK opens avenues for testing our infrastructure easily (this is covered in the chapter on testing).

With CDK Pipelines, setting up CI/CD pipelines for the application as well as the infrastructure can be done via code.

1.3.2. CDK vs Terraform

Terraform is an open-source infrastructure as code tool by HashiCorp. With Terraform, we can describe the required infrastructure using HashiCorp Configuration Language(HCL), a declarative configuration file format. Terraform supports several cloud providers, AWS included, using an extensible plug-in format. The infrastructure defined in the HCL file is deployed using the `terraform` CLI tool which lets us preview a plan of changes and apply and provision the changes.

Getting started with Terraform requires a sound knowledge of HCL, especially once we model complex infrastructure. Being a DSL[1] [2], HCL initially lacked support for programming constructs such as loops, conditionals. Recent updates to Terraform have added support for looping constructs, but it can still be slow to adopt other features. Terraform's biggest drawback remains state management - we must track any infrastructure deployment in a state file. The state file maps the configuration to the cloud resources and also stores the metadata. Terraform supports remote state with locking so that the state is safely maintained without conflicts (when a Terraform configuration is deployed by multiple people separately), but the state management is still on us to manage. State management makes refactoring difficult, especially when migrating to a module or between modules.

1.4. Other CDKs

The CDK has become synonymous with AWS CDK, but that's not the only CDK available - there are other projects that use jsii, aspects of constructs, and provide patterns similar to the AWS CDK.

With the cdktf project, we can use a programming language that we are familiar with to define cloud resources. cdktf generates the corresponding HashiCorp Config Language(HCL) configuration files. Terraform can provision the infrastructure without us having to understand the intricacies of HCL.

cdktf can generate the classes required to define the resources for any of the supported Terraform providers. This allows us to target any provider that Terraform supports and leverage our programming knowledge to build infrastructure using programming constructs to give us the best of both worlds.

Similarly, cdk8s aims to help developers improve their Kubernetes developer experience by providing the familiar programmatic way to generate Kubernetes manifests. cdk8s also support Helm so we can reference a Helm chart, and, using the Helm construct, render the Kubernetes manifest from the Helm chart.

1.5. Review

The CDK lets us build infrastructure using the programming languages we're familiar with. In the next chapter, we'll take a deeper look at constructs - the building block of the CDK.

[1] A Domain-Specific Language is a custom language designed to deal with a specific domain.

[2] Here, HCL was written specifically with HashiCorp tools in mind, Terraform, but now serves as the foundation for most, if not all, of the HashiCorp tools.

2. What Are Constructs?

While the name is officially the AWS Cloud Development Kit, you wouldn't be blamed if you assumed it was **Construct** Development Kit. Constructs are the central object of the CDK and understanding what a construct is, what it represents, and how to use them together is necessary to using the CDK successfully.

Constructs form the building block of your code. Let's look at using one:

Some slight differences between the languages (`this` vs `self`, `new` vs nothing preceding the construct) exist, but the same ideas apply for all languages. This code creates a new `Bucket` construct which represents an AWS S3 Bucket.

Constructs may represent a few other AWS CloudFormation resources as well, but at minimum it always represents at least one.

Typically, you will create a few resources that are connected, in this example an S3 Bucket, and a CloudFront Distribution:

```
my_bucket = Bucket(self, "MyBucket")
Distribution(self, "MyDistribution",
    default_behavior={"origin": S3Origin(my_bucket)}
)
```

Notice on the third line where the Bucket variable is passed to a new `S3Origin` object. You don't need to know the details of how to set up the origin for the Distribution, the `S3Origin` object knows how to do that. All you have to give it is the Bucket to use, and it will do the rest.

Constructs are abstractions of CloudFormation resources.

What are abstractions?

> The essence of abstraction is preserving information that is relevant in a given context, and forgetting information that is irrelevant in that context.
>
> — John V. Guttag, Introduction to Computation and Programming Using Python (January 18 2013)

Here the CDK abstracts away the details of how to create an S3 origin for a CloudFront Distribution and lets you only provide the one thing it can't remove from the equation, the Bucket to point to. You have the option of providing additional information, but you often don't have to. In doing so, the CDK creates the simplest possible way of interacting with a Distribution while leaving you option to customize it as much as needed for your situation.

Continuing on, because the Bucket and Distribution resources are often created together, it's useful to group them together into one easier-to-use thing, a new abstraction.

Not only will you use constructs, you can create new ones. Here, the construct is named `MySimpleWebsite`:

```
class MySimpleWebsite(Construct):
    def __init__(self, scope, id):
        super().__init__(scope, id)
        my_bucket = Bucket(self, "MyBucket")
        Distribution(self, "MyDistribution",
            default_behavior={"origin": S3Origin(my_bucket)}
        )
```

Once defined, it can be used like this:

```
from aws_cdk_lib import App
from ..MySimpleWebsite import MySimpleWebsite

app = App()
stack = Stack(app, "SomeWebsite.com")
MySimpleWebsite(stack, "MySimpleWebsite")

app.synth()
```

It doesn't matter to the CDK if a construct was written by AWS or by you or by someone else.

Maybe you've been using the AWS CDK v1, where the modules were all named `@aws-cdk/aws-*`. With version 2 all the AWS CDK provided modules for S3 Bucket, API Gateway, etc., are all under in the `aws-cdk-lib` module now and are imported a little differently. This book will always show examples with v2 of the library, but would work just the same with v1 with a change to the import statements.

As you write your CDK code, you will work with a variety of constructs. Some from the AWS CDK, some you write yourself, and some others will write and make available for you to use.

2.1. The Tree

Constructs can only do so much by themselves. It's when constructs get together they produce something amazing. Like a lot of data, we organize constructs into trees. Think of the organizational structure of your company:

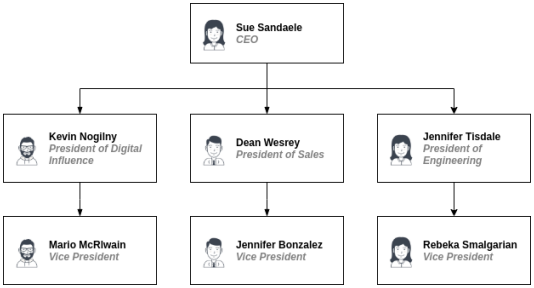

Figure 1. Example org chart

At the top, you have the leader, in this case the CEO. This is the root of the tree. Trees are often drawn 'top down', meaning the root is at the top of the image and the branches and leaves flow down. Just look at it like an upside-down spruce tree.

The CEO has people that report to them. Those people have other people that report to them. And, so on, until you get to the last employee.

Constructs, like the employees in an organization, are also organized this way. At the top is the App construct:

Figure 2. Simple App tree

This is like the CEO. Many presidents report to the CEO. In CDK terms, many Stacks are in an App

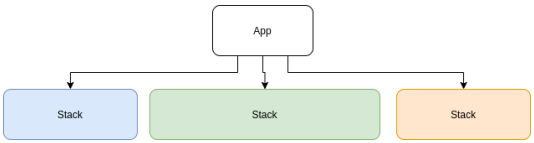

Figure 3. App with multiple Stacks

A CDK Stack and a CloudFormation Stack are analogous.

Presidents all have Vice Presidents and so on down the tree. Constructs are all logically joined this way.

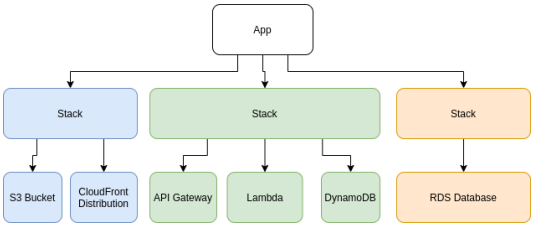

Figure 4. App with multiple Stacks and constructs

When any construct (other than an App) is created, you must tell it who its parent is; or who it reports to. This is also called the 'scope' of the construct.

Let's look at a simple CDK app:

```
app = App()
stack = Stack(app, "SomeWebsite.com")
MySimpleWebsite(stack, "MySimpleWebsite")
```

Line 1: The `App` is created and stored in the `app` variable. This becomes the top of the tree.

Line 2: A new Stack is created and stored in the 'stack' variable. It is given the `app` as the scope. This is always the first parameter of a construct.

Line 3: A new construct (MySimpleWebsite) is created and told the stack is its

scope.

This simple example is more a line than a tree:

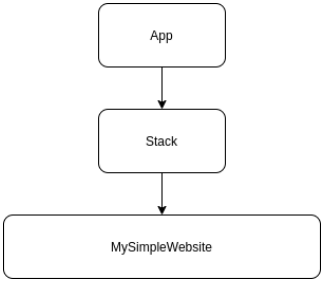

Figure 5. Simple tree

But, when we look at the items that MySimpleWebsite represents:

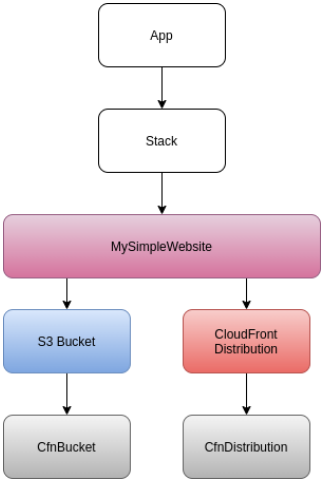

Figure 6. Simple tree with more constructs

We now see how this looks more like a thin tree. How thick or thin a tree is doesn't really matter.

With of our MySimpleWebsite construct, we've made an easy-to-reuse group of AWS resources that all work together to form something useful in fewer

lines of code. We've made our own abstraction.

2.2. Levels of Constructs

A construct can represent a wide range of things. It can be just a single resource, an S3 Bucket, or it can be an entire API including dozens of resources. As we talk more about constructs, it's useful to understand a common way of describing the complexity of a construct. In the previous example, `MySimpleWebsite` took two constructs and grouped them together. One construct was an S3 Bucket and one was a CloudFront Distribution. That S3 Bucket construct will use at least a CfnBucket resource (notice the 'Cfn'). The Distribution construct will use at least a CfnDistribution resource.

Each of these constructs, CfnBucket, Bucket, and MySimpleWebsite, are different 'levels' of constructs. Called Level 1, Level 2, and Level 3 constructs, respectively.

2.2.1. Level 0 Constructs

It doesn't come up often, but there is a Level 0 construct. This represents a basic CloudFormation resource without a type like 'AWS::S3::Bucket'. It forms the basis for all other constructs that are built. However, you'll probably never use this construct, so it's not something that comes up a lot.

2.2.2. Level 1 Constructs

Level 1 (L1) constructs represent a CloudFormation Resource, exactly as CloudFormation defines it. They are an exact representation with no sensible defaults or helper functions or anything useful. They are generated from the CloudFormation API specifications (the docs) and will always start with 'Cfn'.

For example, the 'CfnBucket' construct is an L1 construct.

You won't have to use L1 constructs very often, as there is usually a Level 2 construct already available that is easier to use. L1 constructs are there in case there isn't anything better to use.

2.2.3. Level 2 Constructs

Level 2 (L2) constructs are what the AWS CDK team has written to make it easier to create certain resources and perform common operations. For example, if you have created a DynamoDB Table and would like a Lambda Function to read and write to that table, we must provide it permissions to make those API calls using the IAM role attached to the Function. If you've written an IAM policy before, you know they can be incredibly powerful, but sometimes difficult to get right on the first try.

The L2 constructs often provide helper methods that handle writing those policies for you. These are the Grant functions:

```
my_handler = Function(self, "MyHandler")
my_table = Table(self, "MyTable")
my_table.grant_read_write(my_handler)
```

A lot of constructs have them, and they are in a category of 'helper' functions L2 and L3 constructs have made it easier to wire together AWS resources. Use them when you can.

The Grant Read Write function will create the necessary IAM permissions without fiddling around with writing least-permissive IAM policies yourself. This, and other helper functions, are what L2 constructs are focused on doing, they represent an L1 construct plus some additional things like sensible defaults and helper functions to make the simplest possible abstraction of the L1 resource.

The CDK will provide most of the L2 constructs, but there is nothing stopping us from writing our own if we chose to.

The S3 Bucket and CloudFront Distribution constructs in the MySimpleWebsite example are L2 constructs. While they use L1 constructs like CfnBucket and CfnDistribution, respectively.

The AWS CDK is an open source project and always welcomes Pull Requests (submissions) of L2 constructs. The CDK is not written only by AWS employees. Anybody can be a contributor.

2.2.4. Level 3 Constructs

Level 3 (L3) constructs are groups of L2 constructs (and L1 constructs, if needed). The MySimpleWebsite is an L3 construct, as it combines two L2 constructs together. Which, in turn, uses L1 constructs which are built off of the L0 construct.

The AWS Solution Constructs and CDK Patterns are examples of L3 constructs. The CDK also ships with some L3 constructs in libraries named 'patterns', like AWS ECS Patterns.

You are most likely to write L3 constructs, to represent common functionality in your organization.

For example, maybe you've adopted AWS Serverless and want to build all your APIs using API Gateway, Lambda, and DynamoDB. You could create a construct called MySimpleApi that creates a new API Gateway, Lambda Function, and DynamoDB Table L2 constructs according to your own specific architectural needs.

Smart defaults, observability, logging, and all the details you need can be put directly into the construct. All teams that need to create APIs can use this one

construct and don't need to know the details of what's being created behind the scenes, just what is unique to them.

Let's look at our previous diagram with each construct marked with what level it is.

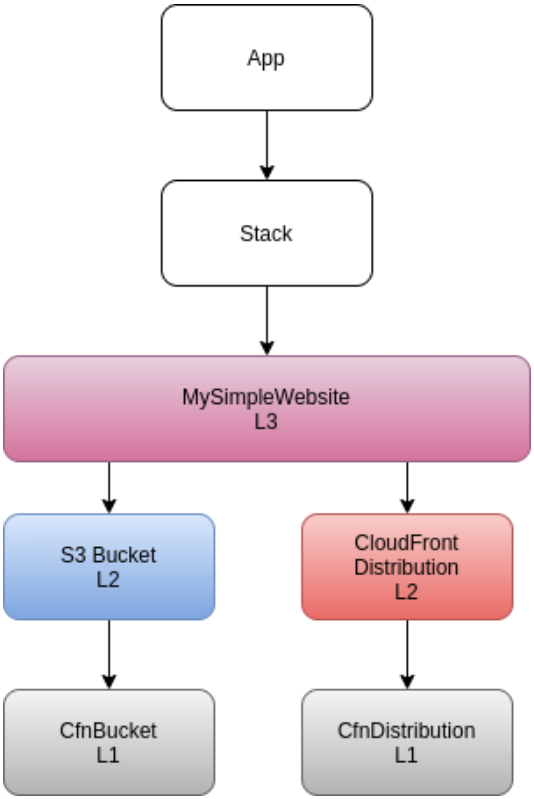

Figure 7. Constructs with levels

Constructs are built on top of other constructs.

2.3. The App Construct

The App construct is special. The App is always the root of the tree. Because it is the root of the tree, it is not given a scope, or parent, when it's created. There isn't much you need to know about the App construct. While it can take some optional parameters, it's rare you will need to provide any.

The App construct is unique because it doesn't represent a CloudFormation resource directly. It is only used to group together Stacks, and CloudFormation has nothing analogous to a group of Stacks in an environment. They're only useful to the CDK.

2.3.1. The Stack Construct

The Stack construct represents one whole CloudFormation Stack template and any associated assets (more on assets in the assets chapter). Stacks can be given either an App or another Stack as the scope during creation.

```
app = App()
parent_stack = Stack(app, "MyParentStack")
child_stack = Stack(parent_stack, "MyChildStack")
```

However, this doesn't create a CloudFormation Nested Stack as you may expect.

If you want to use Nested Stacks, there is a construct specifically for that:

```
app = App()
parent_stack = Stack(app, "MyParentStack")
nested_stack = NestedStack(parent_stack, "MyNestedStack")
```

This would create two stacks in CloudFormation, one nested under the other:

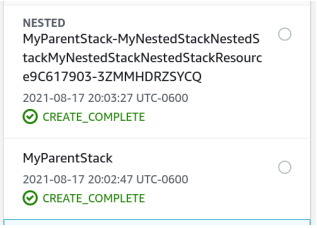

Figure 8. Nested Stacks

You can also have multiple Stacks in one App:

```
app = App()
frontend_stack = Stack(app, "MyFrontend")
backend_stack = Stack(app, "MyBackend")
```

While these stacks won't be nested, they will be grouped together in the same App which means they can share values between them. For example, resources created in the MyFrontend stack could be used in the MyBackend stack.

More on this in the project layout chapter.

2.4. Review

Constructs form the building blocks of all AWS CDK applications. The AWS CDK ships all level of constructs, although you will mostly use L2 constructs either directly or in your own L3 constructs.

The CDK also ships with wonderful L3 constructs throughout all the AWS modules and more are available from the community at the Construct Hub. The 'pipelines' module and the EKS module have some of the more useful L3 constructs.

If you find L2 constructs missing, we hope you contribute to the CDK

through Pull Requests.

3. Publishing Constructs

After we have learned what constructs are and how they are used, we might want to create our own constructs that solve a particular case or prevent others from reinventing the wheel. Most of the constructs we build will be L3 constructs that combine multiple building blocks from the AWS ecosystem and create something reusable.

This chapter will look into building our own construct library and publishing it for others to consume. We will set up the TypeScript project, build for multiple target languages and publish it to the construct hub.

3.1. How to write and publish constructs

While we can write our CDK app in any of the supported languages of the AWS CDK, we should stick to TypeScript when creating our libraries. Exactly as with the CDK itself, we can use a tool called jsii to create code in all CDK languages out of our source code if we use TypeScript. The tool will then build and package distributable archives for each language we can upload to the respective package managers.

If you do not want to support multiple languages, you can write your library in any CDK language, but we do not recommend this as it limits reuse.

3.1.1. What is jsii?

jsii is the JavaScript Interop Interface and is a tool created by the CDK team to support writing code in TypeScript and using it in many target languages. Or as the documentation states, it allows code in any language to interact with JavaScript classes.

So developers can use our code in their favorite programming language without manually writing and maintaining several versions of our library.

3.1.2. How does it work under the hood?

To make this happen, the jsii tool suite creates interface classes that represent your construct's API in all supported languages and packages them with the original JavaScript source code. In any target app the jsii runtime will spawn a child process that runs a NodeJS process with your JavaScript code that interacts with the generated wrapper classes in the app language. Each method call is then serialized and sent to the jsii runtime, handled by the original JavaScript code, and then sent back to the calling class. This class deserializes the response and returns a Java, Python, etc., object to the calling code.

It behaves as if it were a native implementation for the user, but in reality, everything is handled by an embedded JavaScript runtime. That is why you still need NodeJS installed even if using the AWS CDK in another language like Python.

An in-depth description of jsii would fill its own book. However, for more details visit the project page and learn about all the use cases that can be implemented using jsii.

The generation of the different versions of this book is also enabled by jsii to convert the sample code from TypeScript to all other languages.

3.2. Setting up a construct project (using projen)

Now that we know how this works, we want to get started. Basically, we create a default TypeScript library project with some custom settings and a special

build process to trigger jsii.

To skip configuring all these unique settings and the processes needed to package and distribute bundles to all package managers, we strongly recommend using projen for construct libraries.

We initialize a new project using projen and configure all necessary fields in the `.projenrc.js` file:

```
npx projen new awscdk-construct
```

We need to tell projen what minimal version of CDK we want to support with our construct and how to name it in the different package managers. For this, we enable all `publishXY` fields we want to support.

Running projen will also scaffold a folder structure, and our construct lives under `src/`.

As projen already configures our project as jsii compatible all compiler options and build scripts are setup automatically. Suppose we host our project's source code on GitHub. In that case, projen will also generate GitHub Action workflows that manage the auto-release to all package managers on every commit to the repository, including semantic versioning based on conventional commits.

You need to configure the required GitHub secrets with tokens for NPM, PyPI, Maven, etc for auto-release to work. Refer to the projen documentation for detailed help.

3.2.1. Roll-Your-Own (not recommended)

You can also skip projen and set up your project manually. You need a

default TypeScript project and change some settings:

- Make sure your `package.json` has `author`, `repository`, and `description` filled out correctly
- Initialize jsii by running `npx jsii-config`
- Add `jsii` and `jsii-pacmak` as dependencies
- Create a `build` and a `package` script that call `jsii` and `jsii-pacmak` respectively
- Use `jsii-config` to add all language targets

While this is perfectly doable, we still strongly recommend using projen as it handles all the setup, configuration, and release scripting for you.

3.3. Limitations and Best practices

When writing construct libraries, there are several things to keep in mind. Some of them are because of limitations of the underlying jsii serialization others are best practices that emerged in the evolution of the AWS CDK.

3.3.1. Limitations of jsii

There are several limitations to the TypeScript capabilities because the API needs to work in several languages.

All your project's dependencies must either be jsii projects themselves or must be marked as bundled dependencies, as we cannot use the package manager of the target language to resolve them later on.

Some naming restrictions to class members and methods apply because they would be invalid in one or more of the languages the API is translated into.

One example is that member fields cannot have the same name as the class because this breaks in C#.

All behavioral interfaces must start with the letter `I` so jsii knows how to interpret them.

All data entities must be represented as `structs` and can only contain read-only properties that are structs or primitives.

Parameterized types, known as `generics`, are not supported by jsii as Go does not support generics, and the semantics of generics on the other languages vary. This limitation especially applies to the TypeScript classes `Partial` and `Omit` that are not supported in construct libraries.

To prevent errors in any target language, you cannot use property names that are reserved words in any of the support languages of jsii. (e.g. synchronized, elif, range)

3.3.2. Build reusable components

When building constructs, one of the most important things is to ensure that the consumer can use them in many different ways they imagine. While a construct should only serve one purpose and do this well, users should be able to configure the resulting infrastructure to their liking and requirements.

To achieve reusability, make sure to add properties that allow the configuration of all kinds of behaviors.

For example, when writing a construct that provides a simple web hosting functionality, ensure allowing the user to configure certificates as needed, provide options to configure internal or external DNS, maybe configure authentication, and let the user decide how to supply the content files of the

website.

If a construct only serves one specific use case with one set of configuration, many people cannot use it in their projects.

3.3.3. Do not leak abstractions

While providing options to configure different behaviors is essential, it is also critical not to leak too much of the underlying technology to the user. It prevents you from iterating and changing the structure of your construct.

So with the website construct, while being able to configure all the certificate and DNS settings, you still should not allow the user to modify the used CloudFront distribution as you might want to swap this out to something else with the same features at a later version.

Another example is the creation of a CI/CD pipeline construct that allows setting up pipelines without providing access to the underlying AWS CodePipeline implementation. If you do not leak this implementation detail, you can support GitHub Actions, or Gitlab jobs in future versions and improve the user experience. That would not be possible if your users could modify the CodePipeline implementation, which behaves differently than GitHub.

If users of your construct need to access the underlying constructs and they really know what they are doing, escape hatches are a perfect way to modify properties that are not exposed by a construct.

3.4. Building & Publishing

When you are done implementing and testing your construct library, it is

time to package it up and release it into the world for others to consume.

3.4.1. Naming things

The first thing you need to decide is how to name your library. You also need to pick names in all supported languages as they all handle namespaces, dashes, and other characters differently. Also, make sure that no packages with similar names already exist to avoid confusing your users.

If you build a construct for website hosting you might name your packages like this:

- TypeScript: @company/cdk-website-hosting
- Python: company.cdk-website-hosting (module: cdk_website_hosting)
- Java: com.company;cdk-website-hosting

3.4.2. Releasing with projen

After configuring the package's name, you can build your project, create all the artifacts for distribution, and publish them to the package managers of your choice. As mentioned earlier, we strongly recommend using projen to manage your library as this configures everything for you and also creates release pipelines when using GitHub as your source code repository.

With projen setting up and releasing a new construct is a matter of minutes.

3.5. Construct Hub

All public construct libraries are listed on the Construct Hub. This collection

of libraries allows users to search for constructs that solve their current use case and see what others are building. Before building your own construct, make sure to check for similar libraries and try improving them first. The Construct Hub regularly scans all common package managers to detect new libraries and adds them to the index.

For this to work, your package needs to specify one of the keywords (`aws-cdk`, `cdk8s`, or `cdktf`) and be a valid jsii project. Also, it needs to have an open-source license attached to it that allows the use in other projects.

The Construct Hub will show the supported languages of the library and instructions on how to install the library in a user's project.

3.6. Review

Building and releasing your own constructs is easy if you use projen to manage your project and jsii to package your code in different languages. Keep best practices and jsii limitations in mind when writing your code.

4. Leveraging your Integrated Development Environment

A builder is at their best when they make the optimal use of the tools available to them. This is no different for software engineers - there are a lot of tools, utilities, and Integrated Development Environments (IDEs) available. It's not unusual for people to make use of a fraction of the features available. While diving deep into every single feature of every IDE available isn't feasible, we'll look at some features of Visual Studio Code which can improve our CDK experience.

4.1. AWS CDK Toolkit

AWS CDK Toolkit is the CLI tool that is used to build and deploy the CDK code. It is built with Typescript and published in the npm Registry. We can install it using the npm CLI tool, as shown below

```
npm install cdk
npm install cdk@1.132  # to install version 1.132 of the cdk toolkit
```

The examples in the book use `npx` as the prefix to `cdk` for running the `cdk` commands. `npx` is a tool that executes npm binaries. When working on multiple projects, not all projects would likely use the same version of CDK. By installing CDK libraries in the same directory where the application is being developed, we can use `npx` to run the CDK toolkit present in the local directory. This helps us in avoiding conflicts that may arise from using conflicting versions of CDK.

Once we have installed the CDK toolkit (`cdk`), we can run `npx cdk` to list the options. The toolkit has built-in help for nearly all the commands. To invoke help, pass `--help` to the command. For example, to know more about the diff command, we can type the `diff` command as shown below:

```
npx cdk diff --help
```

4.1.1. Creating a new app

To create a new CDK application, we use the `init` command as shown below:

```
npx cdk init
```

Just the `init` command is not enough - we'll get a prompt showing the templates that `cdk init` can scaffold. The choice of language for the cdk application can be specified by the `--language=language name` argument, where the `language name` is one of the supported CDK languages. For example, to initialize a CDK application based on Python, the command will be as shown below:

```
npx cdk init app --language=python
```

The CDK toolkit will scaffold a sample project with a name based on the current directory it is run in. An in-depth look at the directory structure of a CDK app is covered in the Project Layout chapter.

4.1.2. Synthesizing and Deploying apps

Once we have a working prototype, we can use the `synth` command to synthesize and preview the CloudFormation template that the CDK will generate. To deploy the application, the `deploy` command is used.

```
npx cdk synth
npx cdk deploy
```

If a CDK application includes multiple stacks and we want to explicitly mention the stack, we can pass the stack ids as arguments as shown below:

```
npx cdk synth stackName1 stackName2
npx cdk deploy stackName1 stackName2
```

Where `stackName1`, `stackName2` are the stack ids of the stack. The arguments also accept wildcards, so passing `*` will run the command for all stacks. We can do pattern-matching using wildcards by providing the wildcard characters:

- `?`: wildcard to match a single character
- `*`: wildcard to match many characters

For example, if we want to publish all stacks starting with `lambda`, the command will be as shown below

```
npx cdk synth lambda*
npx cdk deploy lambda*
```

4.1.3. Comparing stacks

While refactoring an app, or while adding new features, there can be some changes between the current state and the last deployed state. Before proceeding with deploying, it would be prudent to check for changes. The `diff` command compares the current version of the stack against the deployed version and shows a list of changes. Below is an example of the diff command in action, when a S3 bucket name was changed

```
npx cdk diff images-cdn
Stack images-cdn
Resources
[~] AWS::S3::Bucket images-cdn imagescdn40609981 replace
 └─ [~] BucketName (requires replacement)
     ├─ [-] images.examples.com
     └─ [+] i.example.com
[~] AWS::S3::BucketPolicy images-cdn/Policy imagescdnPolicy95334BDE replace
 └─ [~] Bucket (requires replacement)
     └─ [~] .Ref:
         ├─ [-] imagescdn40609981
         └─ [+] imagescdn40609981 (replaced)
[~] AWS::CertificateManager::Certificate Cert Cert5C9FAEC1 replace
 └─ [~] DomainName (requires replacement)
     ├─ [-] images.examples.com
     └─ [+] i.example.com
[~] AWS::CloudFront::CloudFrontOriginAccessIdentity OriginAccess
OriginAccess8057296B
 └─ [~] CloudFrontOriginAccessIdentityConfig
     └─ [~] .Comment:
         ├─ [-] Origin Access Identity for the S3 bucket images.examples.com
         └─ [+] Origin Access Identity for the S3 bucket i.example.com
[~] AWS::CloudFront::Distribution cf_cdn/CFDistribution
cfcdnCFDistribution8B2039C2
 └─ [~] DistributionConfig
     └─ [~] .Aliases:
         └─ @@ -1,3 +1,3 @@
            [ ] [
            [-]   "images.examples.com"
            [+]   "i.example.com"
            [ ] ]
```

Diff is quite useful to identify any breaking changes and gives us a chance to fix them.

4.2. Visual Studio Code Tips

In this section, we'll look at some tips that can improve developer experience when using Visual Studio Code ("VS Code") with the CDK.

4.2.1. AWS Toolkit for Visual Studio Code

Not to be confused with the CDK toolkit, AWS Toolkit for Visual Studio Code is an open-source plugin VS Code that makes it easier to work with AWS services. While primarily built for developing serverless applications, the plugin features a CDK Explorer feature that lets us visualize the stacks a CDK app includes.

The CDK Explorer is in early preview at the moment and it shows the constructs of an app in a tree view. Before we can make use of the CDK Explorer, we'll have to set it up. To do this, bring up the Command Palette by pressing Ctrl+Shift+P (Cmd+Shift+P, if on an Apple device), and type `AWS: Connect to AWS`. Once selected, VS Code should bring up a list of profiles present in your system to connect to AWS. Click on the AWS icon to activate the Explorer.

We can see a list of the resources that our CDK application has as shown below:

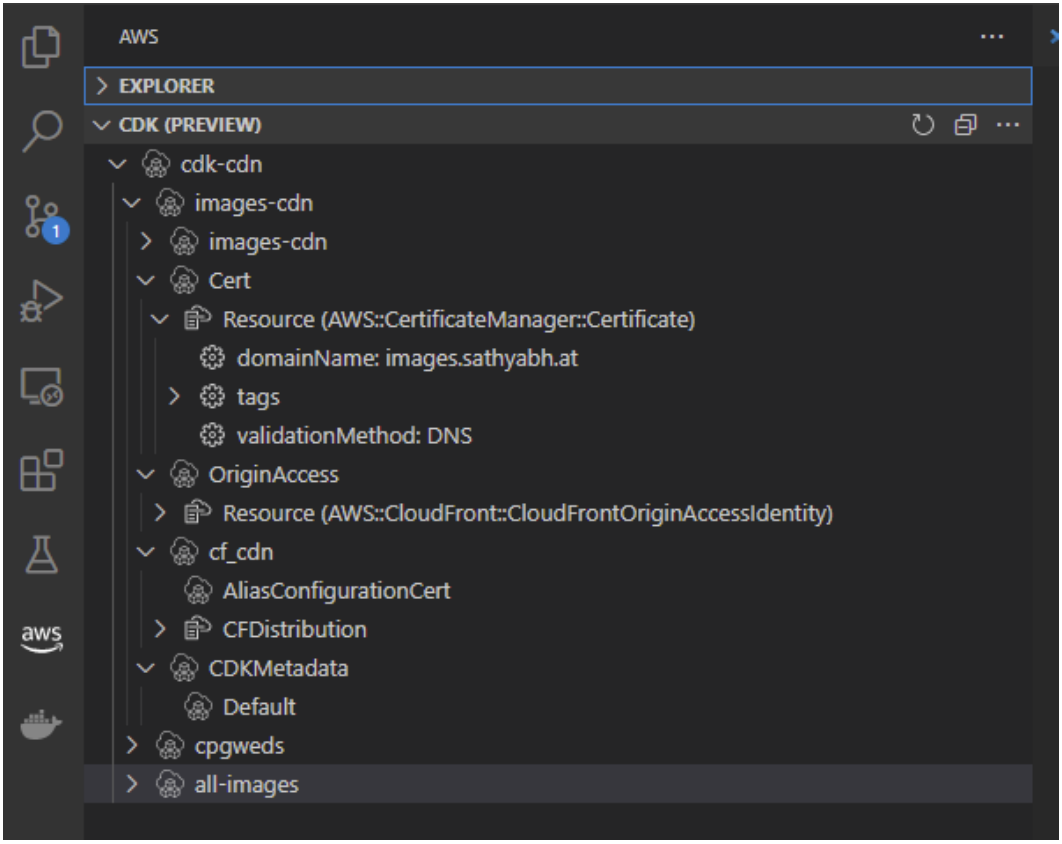

Figure 9. CDK Explorer resource view showing the resources in a CDN stack

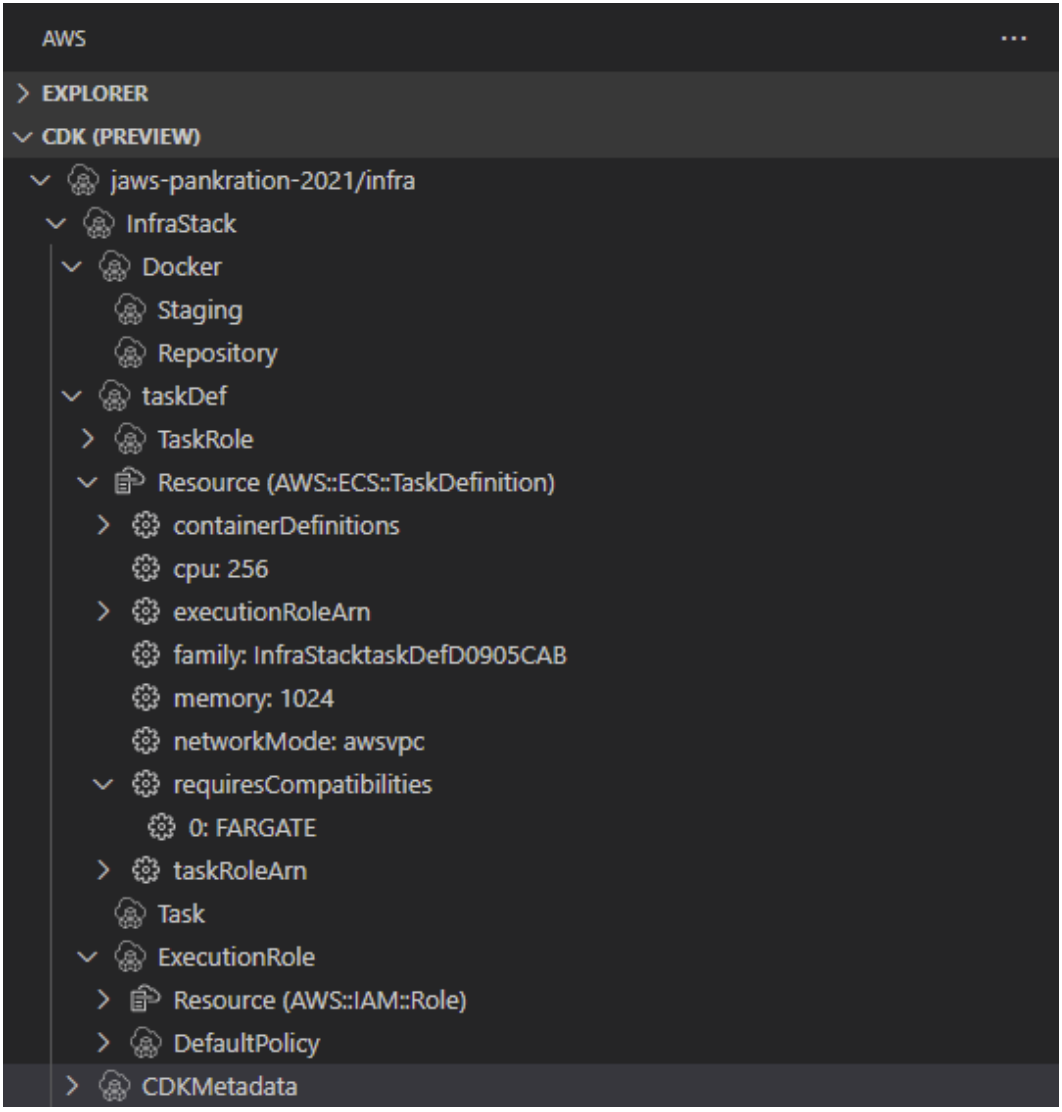

Figure 10. CDK Explorer resource view showing the resources in a Fargate stack

4.2.2. Tasks

When writing code, there's bound to be some commands that we would run

repeated. With CDK, the diff and synth commands are likely to be frequently used during local development. Tasks provide a way to run scripts with a few keyboard shortcuts or using the VS Code Command Palette.

VS Code can also auto-detect tasks from various build systems and populate them in the tasks menu. However, if no tasks have been detected, we can create tasks by creating a file called `tasks.json` in the `.vscode` directory of the repo we're working on.

Let's create a default build task that will do a `cdk diff`. To do this, click on the Terminal menu and click on Configure Tasks.

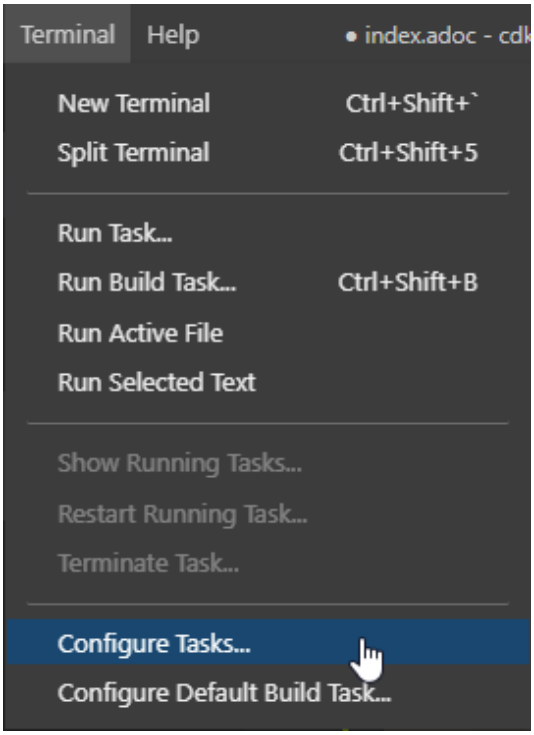

Figure 11. Configure tasks present in the Terminal menu

If we have previously created the `tasks.json` file, we can skip this step and open the file directly to edit it. If not, select the option "Create tasks.json

from template". Select "Others" as the type, and VS Code will open a `tasks.json`. The file structure will be as shown below:

```
{
    "version": "2.0.0",
    "tasks": [
        {
            "label": "echo",
            "type": "shell",
            "command": "echo Hello"
        }
    ]
}
```

The spec of the JSON file is available on VS Code's documentation page, but a brief mention of the various attributes are below:

- `label`: Label is the name of the task. This is shown in the Tasks menu.
- `type`: Type defines how the task is run. VS Code can run a task either as a shell command or as a process.
- `command`: The actual command to be invoked when the task is selected.

Some other keys that are available but not shown above include

- `options`: The `options` key lets us override
  - [] the current working directory by setting the `cwd` key
  - [] any environment variables by setting them in the `env` key
  - [] the default shell by setting the `shell` key

The `tasks.json` supports VS Code's IntelliSense. By pressing Ctrl+Space (or Cmd+Space, if you're on Apple devices), we can see the available attributes that can be defined.

For the CDK diff to work, we'll have to run `cdk diff` as the command. So

let's modify the `tasks.json` to as shown below

```
{
    "version": "2.0.0",
    "tasks": [
        {
            "label": "CDK diff",
            "type": "shell",
            "command": "npx cdk diff"
        }
    ]
}
```

With this task configured, now we can invoke the task by opening the Command Palette and typing "Tasks: Run Task" and then selecting "CDK Diff". To make this available at a button press, open the Command Palette and choose "Tasks: Configure the Default Build Task" and select the "CDK Diff" task. Now, you can invoke this task by pressing Ctrl+Shift+B.

Similarly, you can add default tests tasks, or any other task and invoke them on demand, as compared to opening a terminal and fumbling through the commands and flags. Committing the `tasks.json` makes this config available to anyone who has cloned the repo, providing a consistent experience for people using VS Code.

4.2.3. Debugging

When things aren't working as expected per our code, the best way to figure out what's going wrong is to use a debugger. With traditional infrastructure as code tools, debugging is typically restricted to having to write out the values to the standard output and try to make sense of what's going wrong. Since CDK apps are built with programming languages, we can make the best use of tools, including, to figure out what's going wrong and where.

VS Code comes with a built-in debugger that lets us set breakpoints. With the

breakpoints set, the program execution stops at the marked lines.

We can control the flow and analyze the variables, system state, and can step into or over each line, and try to figure out why the program isn't working as expected. The below sections look at how we can set up VS Code for debugging a CDK application.

To debug, we'll have to specify a create a `launch.json` file in the `.vscode` directory present in the root of the repository. The `launch.json` file tells VS Code how to launch the debugger. Like `tasks.json`, it supports IntelliSense, so pressing Ctrl+Space keys will show a prompt to enable auto-complete for the available attributes. For a TypeScript file, the `launch.json` will look like the below:

```
{
    "version": "0.2.0",
    "configurations": [

        {
            "type": "node",
            "request": "launch",
            "name": "Debug CDK App",
            "skipFiles": [
                "<node_internals>/**"
            ],
            // use ts-node to transpile TypeScript to JavaScript
            "runtimeArgs": [
                "-r", "./node_modules/ts-node/register/transpile-only"
            ],
            // the file to launch. Typically, the entry point of the program.
            "args": [
                "${workspaceFolder}/bin/s3assets.ts"
            ]
        }
    ]
}
```

For a Python CDK project, we would have to change the `launch.json` to suit the Python program like below

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "Debug CDK Py",
            "type": "python",
            "request": "launch",
            // we can use ${file}, but better to set this to the entry point
of the app
            "program": "${file}"
        }
    ]
}
```

An in-depth description of the attributes are available on VS Code's documentation page, but a brief mention of the various attributes are below:

- `type`: The type of debugger to use for this configuration.
- `request`: How VS Code should run the debugger. Currently supported types are launch and attach. A `launch` will start a new process and attach a debugger to that process, while an attach instructs VS Code how to attach an existing running process to debug.
- `name`: the name of the debug configuration. This is shown in the Command Palette.
- `skipFiles`: The files that VS Code should ignore when debugging. Setting this lets us step in/out of our functions and not do a line-by-line debug of libraries and internal modules.
- `runtimeArgs`: The parameters that are passed to the debugger runtime. In our case, setting `transpile-only` instructs ts-node to skip type checking to speed up execution time.

With this file saved, we can now see the Debug configuration in the Debug Mode.

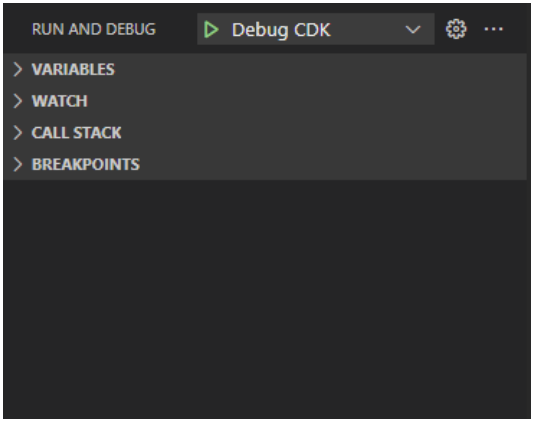

Figure 12. Debug section of VS Code showing our debug configuration

Before we can start debugging, let's put a breakpoint on a specific line so that the execution stops there. This is done by clicking to the left side of the line we want the program execution to stop. When a breakpoint is set, the corresponding line will have a red dot.

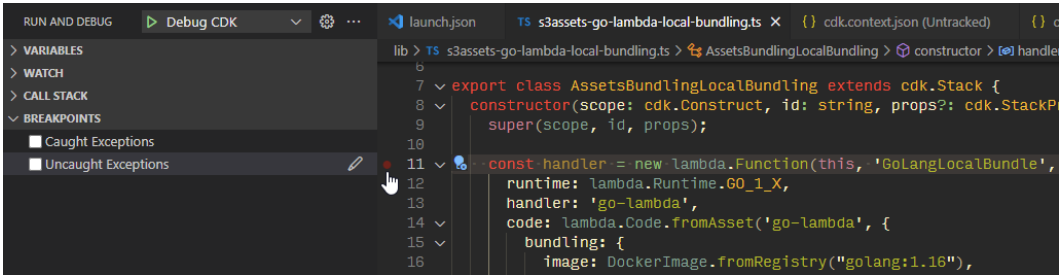

Figure 13. Setting a breakpoint

Once a breakpoint has been set, click on the Debug button (or press F5) to start the debugger. VS Code will stop the execution at the breakpoint. We can see it in action in the below snapshot:

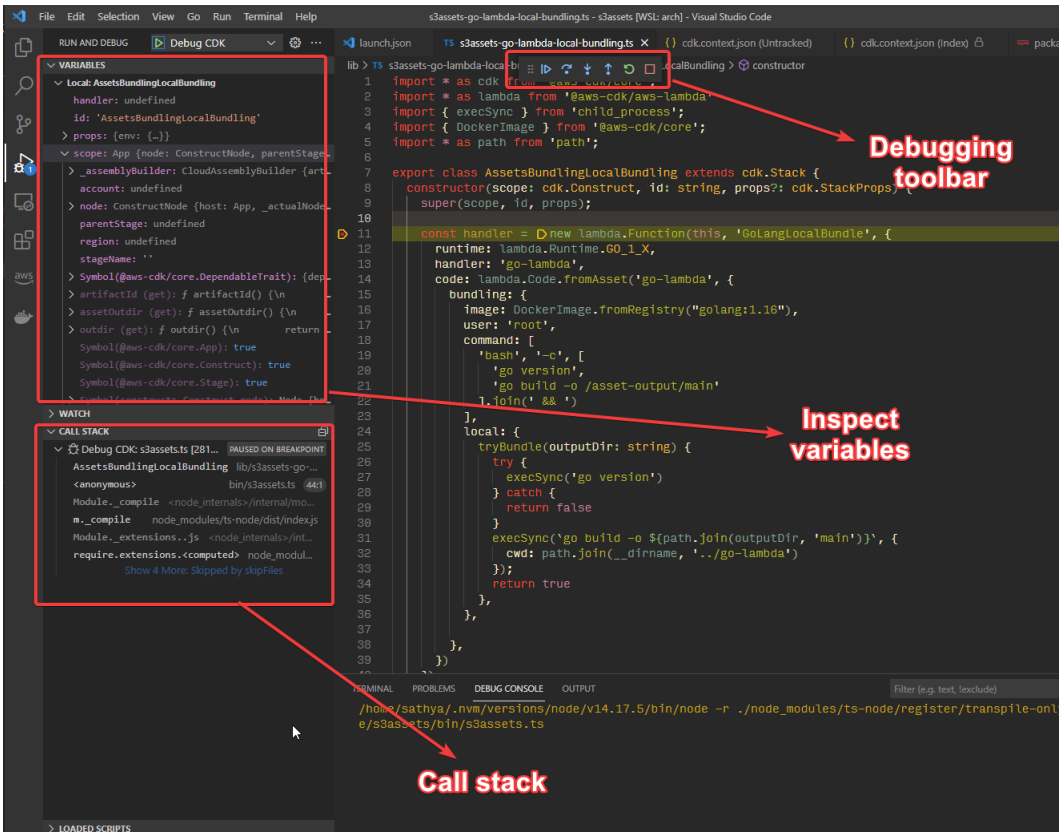

Figure 14. Debugger in action

The debugging toolbar has some buttons that we can use to control the flow of the program - whether we want to step into a function to debug further or to step over and skip debugging the function completely. The variables window shows the values of all the variables of the application. To watch the value of a specific variable, we can add them to the Watch window. The call stack window shows the order in which the functions and invoked.

With rich debugging support, VS Code makes debugging CDK apps easy and not have to rely on cumbersome `print()` statements entered all over the app.

4.2.4. ESLint Cleanup

ESLint is a powerful tool for ensuring that you have clean and easy-to-read Typescript/Javascript code. Visual Studio Code has an extension you can install for running commands directly in the IDE.

First, we begin by making sure you have the ESLint package installed, either globally:

```
npm i -g eslint
# or
yarn global add eslint
```

Next, ensure we have the extension installed. This can be done from the Extensions view (Ctl/Cmd-Shift-X) and searching for ESLint, or can be done from the command line:

```
code --install-extension dbaeumer.vscode-eslint
```

There are plenty of extensions in the VS Code marketplace for ESLint. These instructions are specific for this extension. If you'd like to use another extension, you can follow the instructions on the extension's page.

If we're using projen for our CDK projects, then we're set up with a good configuration file already. If not, we can use eslint to create a new file:

```
eslint --init
```

Follow the prompts. If you would like more details on configuring ESLint, refer to ESLint documentation.

Finally, we're going to set up ESLint to run on every save, so that we don't

have to do it manually. This helps ensure our code is always formatted nicely. Open either the project or your user's `.vscode/settings.json' file. Add the following JSON to the existing object:

```
{
  "editor.codeActionsOnSave": {
    "source.fixAll.eslint": true
  },
  "eslint.validate": ["javascript"]
}
```

Now when you save any file ESLint runs and fixes any issues it can.

This extension doesn't seem to traverse any directories looking for the proper configuration file, like .eslintrc.json. If you have put your CDK code in a subdirectory of your project and the configuration file is not in the root, it won't run with your CDK configuration. Just re-open VS Code in the subdirectory where the CDK code is and it should work as expected.

4.3. Review

The CDK cli is an essential tool that we would using when working with CDK apps. Using the debugger can save a lot of time when trying to figure out tricky issues, without having to resort to print statements all over.

5. Project Layout

A CDK project is the main codebase that represents all the CDK code you will write. When creating a new CDK project, you have different options on how to start. The CDK cli has an `init` command to create all the files needed to start a new project. Another popular way is to use projen to manage the project and its files. You could also create all the files from scratch.

Let's review the files you'll need to make a CDK project work. Then we'll look at where `cdk init` and projen puts these files.

5.1. Typescript Projects

5.1.1. Application Files

CDK projects must have at least one application file, but multiple CDK applications can exist in one project.

This is one simple CDK application

```
#!/usr/bin/env node

// An example CDK Application file written in Typescript

import 'source-map-support/register';
import * as cdk from '@aws-cdk/core';
import { MyTsCdkAppStack } from '../lib/my-ts-cdk-app-stack';

const app = new cdk.App();
new MyTsCdkAppStack(app, 'MyTsCdkAppStack');
```

The creation of the `cdk.App()` object identifies this as an Application file.

Keep applications in their own files and separated from each other to increase readability.

5.1.2. Stack Files

A CDK Stack is an abstraction of a CloudFormation Stack and contains one or more constructs. You will typically create new sub-classes of the CDK Stack class from the `core` module.

```
import * as cdk from '@aws-cdk/core';

export class MyTsCdkAppStack extends cdk.Stack {
  constructor(scope: cdk.Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // The code that defines your stack goes here
  }
}
```

The `MyTsCdkAppStack` class inherits from `cdk.Stack`, making this a Stack file.

5.1.3. Construct Files

Constructs represent resources in AWS. Constructs can be created as shown in the below snippet:

```
import * as cdk from '@aws-cdk/core';
import * as s3 from '@aws-cdk/aws-s3';

export class MyConstruct extends cdk.Construct {
  constructor(scope: Construct, id: string) {
    super(scope, id);
    new s3.Bucket(this, 'ABucket');
  }
}
```

The `MyConstruct` class extends the CDK `Construct` class, making this a construct file. Like Stacks and Applications, constructs should be in their own files and not combined into one extensive file, to help with readability.

5.1.4. cdk.json

The CDK configuration file that defines the entrypoint of the application and context. Notice how the "app" field is the full command to execute the Typescript code, including the call to ts-node, running the file created in the `bin` folder:

```
{
  "app": "npx ts-node --prefer-ts-exts bin/my-ts-cdk-app.ts"
}
```

Here, the `bin/my-ts-cdk-app.ts` is a CDK app. The application file and purpose will be covered in more detail below.

The CDK is very flexible, and you have complete control over how your CDK code is executed. You can choose to override ts-node parameters or execute any application file you have in your project.

You may additionally see context variables. Context variables are covered more in the Configuration Management chapter. Several context variables

may be added automatically. These serve as feature flags and change how the CDK code will run. This allows for backwards compatibility as the CDK matures. Refer to the source code @aws-cdk/cx-api/lib/features.ts to understand what each context variable does.

```
{
  "app": "npx ts-node --prefer-ts-exts bin/my-ts-cdk-app.ts",
  "context": {
    "@aws-cdk/aws-apigateway:usagePlanKeyOrderInsensitiveId": true,
    "@aws-cdk/core:enableStackNameDuplicates": "true",
    "aws-cdk:enableDiffNoFail": "true",
    "@aws-cdk/core:stackRelativeExports": "true",
    "@aws-cdk/aws-ecr-assets:dockerIgnoreSupport": true,
    "@aws-cdk/aws-secretsmanager:parseOwnedSecretName": true,
    "@aws-cdk/aws-kms:defaultKeyPolicies": true,
    "@aws-cdk/aws-s3:grantWriteWithoutAcl": true,
    "@aws-cdk/aws-ecs-patterns:removeDefaultDesiredCount": true,
    "@aws-cdk/aws-rds:lowercaseDbIdentifier": true,
    "@aws-cdk/aws-efs:defaultEncryptionAtRest": true,
    "@aws-cdk/aws-lambda:recognizeVersionProps": true
  }
}
```

For example, if we search the source code file for `@aws-cdk/core:enableStackNameDuplicates` we see directly above the line documentation on what it does.

```
/**
 * If this is set, multiple stacks can use the same stack name (e.g. deployed
to
 * different environments). This means that the name of the synthesized
template
 * file will be based on the construct path and not on the defined
`stackName`
 * of the stack.
 *
 * This is a "future flag": the feature is disabled by default for backwards
 * compatibility, but new projects created using `cdk init` will have this
 * enabled through the generated `cdk.json`.
 */
```

The CDK team has put the documentation for these directly in the source

code.

5.1.5. cdk.context.json

Context values are key-value pairs that constructs can use to cache information about your environment. Or, as we saw above, define feature flags that change the way constructs work in small ways.

Constructs like those in the ec2 module will need to read your AWS account data for information like a region availability zones or Amazon Managed Instance (AMI) identifiers. If you use the VPC construct's `Vpc.fromLookup` static function, the VpcId will be written to the `cdk.context.json` file. This allows your CDK application to be deterministic and always produce the same output, given the same inputs. This is an important concept and the runtime context and `cdk.context.json` file makes that possible.

For more information on Context variables, see Chapter 7.

5.1.6. README.md

All good projects have a README file and CDK projects are no different. There are plenty of good resources on how to maintain a good README file and a full discussion is out of the scope of this book. Use your favorite search engine and find "how to write a good README file".

However, there are a few things that I recommend you include in your CDK projects specifically:

- A diagram of what your CDK application will produce. cdk-dia can be used to render this from a sample application synthesis.
- How to execute the application if it is environment specific (needs AWS

access for `Vpc.fromLookup()` or similar).

5.1.7. tsconfig.json

Typescript requires some compiler configuration which is stored in the tsconfig.json file. You won't likely need to change much here unless you plan on rearranging directories or changing the way Typescript is going to manipulate your files.

Full Typescript configuration is outside the scope of this book, but you can read more on typescriptlang.org.

5.1.8. package.json

This is a standard Nodejs file. The package.json will contain information about your CDK app include a name, dependencies, and other important project-level configuration.

The package-lock.json (or yarn.lock) ensures, that when using npm/yarn to install dependencies you will get the same versions each time. If you'd rather use another package manager like Yarn to maintain your package, you can delete this file.

5.1.9. Tests

All good code is tested and CDK code is no different. Jest is a common Typescript testing framework and a single empty test will be created for you to get started.

```
import { expect as expectCDK, matchTemplate, MatchStyle } from '@aws-
cdk/assert';
import * as cdk from '@aws-cdk/core';
import * as MyTsCdkApp from '../lib/my-ts-cdk-app-stack';

test('Empty Stack', () => {
    const app = new cdk.App();
    // WHEN
    const stack = new MyTsCdkApp.MyTsCdkAppStack(app, 'MyTestStack');
    // THEN
    expectCDK(stack).to(matchTemplate({
      "Resources": {}
    }, MatchStyle.EXACT))
});
```

If you have never used Jest before, the testing chapter will cover what you need to know.

5.1.10. .gitignore and .npmignore

These ignore files are used by git and npm to determine which files can be ignored by their respective systems. For example, the .gitignore files will have `node_modules`, since you shouldn't be committing your NodeJS dependencies to the repository. The npm ignore file will, likewise, ignore the directory for a similar reason; you don't ship your dependencies.

5.1.11. Initializing with the CLI

To initialize a new CDK application using Typescript, run the following command:

```
$ npx cdk init app --language=typescript
```

The CDK cli will not initialize new code in a non-empty directory.

The project will be initialized and you'll see the files mentioned previously. You'll get one new application file in the `bin/` directory, one new stack file in the `lib/` directory, and one new test in the `test/` directory. These files are ready for you to edit. Rename them as the CLI will name them after the directory you were in when you created the project and that rarely represents a good stack name. For example, if your project directory was named `my-ts-cdk-app`, your stack name will be MyTsCdkAppStack which is both redundant and probably doesn't represent what you'll eventually put in that stack.

```
├── bin
│   └── my-ts-cdk-app.ts
├── cdk.json
├── .gitignore
├── jest.config.js
├── lib
│   └── my-ts-cdk-app-stack.ts
├── .npmignore
├── package.json
├── package-lock.json
├── README.md
├── test
│   └── my-ts-cdk-app.test.ts
└── tsconfig.json
```

bin/*

The application files are stored in the `bin` directory. Edit the generated file and add more application files as needed. You can add more application files to create different combinations of stacks. For example, you could also have an application specifically for a developer account that has most of the same stacks, but uses a shared RDS instance to reduce costs.

When you generate a new CDK project, a cdk.json file is generated with a

field called "app" set to `ts-node bin/my-ts-cdk-app.ts`, or similar. This is the default application, but you can always create more. You may want to create an application for integration testing or infrastructure testing.

Add additional files to the `bin` directory, like a `bin/my-integration-test.ts`, to create more applications. You can copy and paste the contents from the generated file to get started. To use those applications, you can either change the `cdk.json` file or you can override the application to use with the CLI command:

```
npm run synth --app 'npx ts-node bin/my-integration-test.ts'
```

I recommend this approach. If this is something you'll be doing a lot, you can also add this in as an additional NPM script by updating the package.json:

```
{
  "scripts": {
     "synth:integ": "cdk synth --app 'npx ts-node --prefer-ts-exts bin/my-
integration-test.ts'"
  }
}
```

lib/**

Stack and construct files are stored in the `lib` directory. Edit the generated files and add more stack and construct files here as needed.

5.1.12. Initializing with Projen

You can also create new projects using projen. Like the CDK, projen uses constructs. Instead of the constructs representing AWS resource, projen

constructs represent project files like .gitignore, .npmignore, package.json and other files.

To create a new CDK application using projen:

```
npx projen new awscdk-app-ts
```

Projen **will** initialize new code in a non-empty directory.

This will produce files similar to:

```
├── cdk.json
├── .eslintrc.json
├── .gitattributes
├── .github
│   ├── pull_request_template.md
│   └── workflows
├── .gitignore
├── LICENSE
├── .mergify.yml
├── .npmignore
├── package.json
├── .projen
│   ├── deps.json
│   └── tasks.json
├── .projenrc.js
├── README.md
├── src
│   └── main.ts
├── test
│   └── main.test.ts
├── tsconfig.eslint.json
├── tsconfig.jest.json
├── tsconfig.json
├── .versionrc.json
└── yarn.lock
```

Most of the standard files are there. In addition, projen adds several additional files that will help you manage your code long term.

.projenrc.js

The projen configuration file. In this file a projen construct is created, in this case a `AwsCdkTypeScriptApp` construct which encapsulates all of the files and functionality needed for a CDK application. Changing the properties of the construct will change how it generates other files. For example, if you set `buildWorkflow` to false, the Github Action for building PRs will not be created. There are many options available and are changing enough that they can't all be covered here. A few options to be aware of:

- cdkVersion - Use this to set the version of the CDK modules you'd like to use.
- cdkVersionPinning - A boolean, true or false, if you'd like to use a specific version of the CDK modules. True will use specifically the version defined with the `cdkVersion` option and false will use the latest version of the modules.
- cdkDependencies - An array of @aws-cdk modules you are going to use. For example: `['@aws-cdk/aws-route53', '@aws-cdk/aws-ec2']`

Many of the files that projen generates will be marked as read-only and shouldn't be edited directly. If you need to change these files, like the .gitignore, you should use methods and properties on the projen construct. Projen will add comments to files that it will maintain for you. Generated files will be read-only.

.npmignore

```
# ~~ Generated by projen. To modify, edit .projenrc.js and run "npx projen".
.cdk.staging/
/.eslintrc.json
```

If you see comments like this, don't edit the file and go find the properties

you need to change in `.projenrc.js`.

```
const project = new AwsCdkTypeScriptApp({
  cdkVersion: '1.95.2',
  defaultReleaseBranch: 'main',
  name: 'my-projen-ts-cdk-app',
  npmignore: ['someotherfile']
});
project.synth();
```

Refer to the projen documentation on various options and how to use them.

.eslintrc.json

An ESLint configuration file that will examine your CDK code and make sure it meets certain coding standards. This can catch many coding issues you wouldn't find until the code actually runs.

Like most projen generated files, don't edit this file directly and instead use options on the projen construct to control its contents.

.github/**

Several useful Github Action workflows will be created by default. You can disable these with options if you don't want the workflows.

LICENSE

A license file is generated for you. Review the license defined and decide if you'd like to change it through the `license` option.

.mergify.yml

A Mergify definition file that will help you manage the merges of your code. Refer to the Mergify (https://mergify.io/) website for more details.

.projen/**

Projen configuration files that define tasks and dependencies. You won't edit these files directly and they will be modified as you change the projen construct options.

src/**

Application and stack files are stored in the `src` directory. When code is built, it will be placed in the `lib` directory. Do not edit files in the `lib` directory, but edit the files in the `src` directory all you want.

When you generate a new CDK project, a cdk.json file is generated with a field called "app" set to `npx ts-node --prefer-ts-exts src/main.ts`, or similar. This is the default application, but you can always create more. Maybe you want to create an application for integration testing or infrastructure testing.

Add additional files to the `src` directory, like a `src/my-integration-test.ts`, to create more applications. You can copy and paste the contents from the generated file to get started. To use those applications, you can change the `cdk.json` file. However, if you are adding another application and want to retain the original, you can override the application to use with the CLI command:

```
npm run synth --app 'npx ts-node src/my-integration-test.ts'
```

The `--app` parameter is an entire command to synthesize the application, including the interpreter to use, in this case 'ts-node'. If this is something you'll be doing a lot, you can also add this in as an additional NPM script by updating the `.projenrc.js` file:

```
const project = new AwsCdkTypeScriptApp({
  cdkVersion: '1.95.2',
  defaultReleaseBranch: 'main',
  name: 'my-projen-ts-cdk-app',
  scripts: {
    'synth:integ': "cdk synth --app 'npx ts-node --prefer-ts-exts src/my-
integration-test.ts'",
  },
});
project.synth();
```

Later chapters will talk more about the value in adding more applications to a CDK project.

tsconfig.json

Projen will generate a few different TS configuration files, for jest, eslint, and ts[]js compilation.

5.2. Python CDK Projects

5.2.1. Initializing with the CLI

```
$ npx cdk init app --language=python
```

Irrespective of the language we choose, we still need npx (and therefore NodeJS) to use the CLI to create a project and run commands later.

```
├── app.py
├── cdk.json
├── my_python_cdk_app
│   ├── __init__.py
│   └── my_python_cdk_app_stack.py
├── README.md
├── requirements.txt
├── setup.py
└── source.bat
```

Let's review each listing:

app.py

This is the main entrypoint to the application. If you add Stacks or other code, you'll probably start here. You'll reference stack classes you create in the `my_python_cdk_app` directory. This filename can be anything as long as you also update the reference to it in `cdk.json`.

cdk.json

The CDK configuration file that defines the entrypoint of the application. Notice how the `app` field is the full command to execute the Python code, `python3 app.py`. If you change the `app.py` filename or add a new one, this file needs updating. This value can be overridden using the cli's `--app` parameter.

```
{
  "app": "python3 app.py"
}
```

You may additionally see context variables. Context variables are covered more in the Configuration Management chapter. Several context variables will be added automatically. They are feature flags in the CDK.

Feature flags are used to allow you to choose when to use new functionality that may not be backwards compatible with your code built with previous versions of a particular module, like @aws-cdk/aws-apigateway. Refer to the source code @aws-cdk/cx-api/lib/features.ts to understand what each context variable will do. Specific modules will call out feature flags they use, and you can enable or disable them using context variables.

Below are examples of the features flags enabled by context in version 1 of the CDK. When version 2 of the CDK was released, all the feature flags we're set to true by default, making them no longer necessary for new projects. Over time, feature flags will be added in version 2 and this list won't stay empty for long.

```
{
  "app": "python3 app.py",
  "context": {
    "@aws-cdk/aws-apigateway:usagePlanKeyOrderInsensitiveId": true,
    "@aws-cdk/core:enableStackNameDuplicates": "true",
    "aws-cdk:enableDiffNoFail": "true",
    "@aws-cdk/core:stackRelativeExports": "true",
    "@aws-cdk/aws-ecr-assets:dockerIgnoreSupport": true,
    "@aws-cdk/aws-secretsmanager:parseOwnedSecretName": true,
    "@aws-cdk/aws-kms:defaultKeyPolicies": true,
    "@aws-cdk/aws-s3:grantWriteWithoutAcl": true,
    "@aws-cdk/aws-ecs-patterns:removeDefaultDesiredCount": true,
    "@aws-cdk/aws-rds:lowercaseDbIdentifier": true,
    "@aws-cdk/aws-efs:defaultEncryptionAtRest": true,
    "@aws-cdk/aws-lambda:recognizeVersionProps": true
  }
}
```

We can learn more about each feature flag by searching the documentation. We can also search the code repository for the key. If we search that source code file for `@aws-cdk/core:enableStackNameDuplicates` we see directly above the constant's declaration the documentation on what it does.

```
/**
 * If this is set, multiple stacks can use the same stack name (e.g. deployed
to
 * different environments). This means that the name of the synthesized
template
 * file will be based on the construct path and not on the defined
`stackName`
 * of the stack.
 *
 * This is a "future flag": the feature is disabled by default for backwards
 * compatibility, but new projects created using `cdk init` will have this
 * enabled through the generated `cdk.json`.
 */
export const ENABLE_STACK_NAME_DUPLICATES_CONTEXT = '@aws-
cdk/core:enableStackNameDuplicates';
```

The CDK team has put the documentation for these directly in the source code.

my_python_cdk_app/__init__.py

This is an empty Python package definition file.

my_python_cdk_app/my_python_cdk_app_stack.py

This is the primary place you're likely to start with changing code. This is a new Python class that inherits from a CDK Stack. Add in additional constructs to this stack. As you need to add more Stacks to your application, you can add new files to this directory.

```
from aws_cdk import core as cdk
from aws_cdk import core

class MyPythonCdkAppStack(cdk.Stack):

    def __init__(self, scope: cdk.Construct, construct_id: str, **kwargs) ->
None:
        super().__init__(scope, construct_id, **kwargs)

        # The code that defines your stack goes here
```

Existing constructs can be added to `_init_` constructor. As you create constructs yourself, you'll add them to stacks, either to this one or to others you create as your project grows.

5.2.2. README.md

All good projects have a README file and CDK projects are no different. There are plenty of good resources on how to maintain a good README file and a full discussion is out of the scope of this book. Use your favorite search engine and find "how to write a good README file".

However, there are a few things that I recommend you include in your CDK projects specifically:

- A diagram of what your CDK application will produce. cdk-dia can be used to render this from a sample application synthesis.
- How to execute the application if it is environment specific (needs AWS access for `Vpc.fromLookup()` or similar).

requirements.txt

A Python Requirements file that contains all dependencies that the project

needs, referenced during a `pip install` to download and use external dependencies, like the CDK libraries and other Python libraries.

setup.py

A standard Python file for defining the CDK app. CDK versions used, the package directory, package description and other parameters can be changed, many of which you'll want to adjust.

```
import setuptools


with open("README.md") as fp:
    long_description = fp.read()


setuptools.setup(
    name="my_python_cdk_app",
    version="0.0.1",

    description="An empty CDK Python app",
    long_description=long_description,
    long_description_content_type="text/markdown",

    author="author",

    package_dir={"": "my_python_cdk_app"},
    packages=setuptools.find_packages(where="my_python_cdk_app"),

    install_requires=[
        "aws-cdk.core==1.108.1",
    ],

    python_requires=">=3.6",

    classifiers=[
        "Development Status :: 4 - Beta",

        "Intended Audience :: Developers",

        "Programming Language :: JavaScript",
        "Programming Language :: Python :: 3 :: Only",
        "Programming Language :: Python :: 3.6",
        "Programming Language :: Python :: 3.7",
        "Programming Language :: Python :: 3.8",

        "Topic :: Software Development :: Code Generators",
        "Topic :: Utilities",

        "Typing :: Typed",
    ],
)
```

source.bat

A batch file for Windows that will help you set up your Python virtual environment (venv)

Changing the Generated Code

Once the code is generated, you'll start editing it. The first place you'll start is by adding constructs to the stack class generated for you.

```
class MyPythonCdkAppStack(cdk.Stack):

    def __init__(self, scope: cdk.Construct, construct_id: str, **kwargs) ->
None:
        super().__init__(scope, construct_id, **kwargs)

        # The code that defines your stack goes here
        # add more constructs here
```

You can also add additional stacks as you need them by creating new files in the `my-python-cdk-app`, alongside the file generated for you. Once added, you create new instances in the `app.py`.

```
app = core.App()
MyPythonCdkAppStack(app, "MyPythonCdkAppStack")
MyPythonCdkDatabaseStack(app, "MyPythonCdkDatabaseStack")

app.synth()
```

At the time of this writing, projen does not support creating a Python-based CDK application project. However, projen is an open source project and is always accepting Pull Requests. If you can't contribute to open source projects because of legal/contractual requirements, then projen still works with your own private code. Projen also supports private 3rd-party libraries.

5.3. Organizing Your Project

The CDK doesn't much care how you organize your files.

When you first bootstrap a new project, using either the cdk cli or projen, some files will be generated for you. Overtime you will create more files yourself. After a while, you can have a lot of files for even moderately sized projects.

Most files will remain where they were generated, but the stack and construct files that are part of the lib/src directories will probably need some additional structure.

Starting with a baseline TS project:

```
├── lib
│   └── my-ts-cdk-app-stack.ts
```

After a few iterations, we could end up with a few more files:

```
├── lib
│   └── db-stack.ts
│   └── vpc-stack.ts
│   └── app-stack.ts
│   └── cronjobs-stack.ts
```

and with continued refactoring and logically grouping constructs, have a large 'lib' directory of files:

```
├── lib
│   └── db-stack.ts
│   └── vpc-stack.ts
│   └── app-stack.ts
│   └── cronjobs-stack.ts
│   └── db.ts
│   └── frontend.ts
│   └── api.ts
│   └── fixtures.ts
│   └── auth.ts
│   └── ingress.ts
```

After some growth, it can be a little tedious finding what you're looking for. A simple organization technique is to create new directories for each layer of the application:

```
lib
├── app
│   ├── api.ts
│   ├── app-stack.ts
│   ├── cronjob-stack.ts
│   ├── cronjob.ts
│   └── frontend.ts
├── db
│   ├── db-stack.ts
│   ├── db.ts
│   └── fixtures.ts
└── infra
    ├── auth.ts
    ├── ingress.ts
    └── vpc-stack.ts
```

Now the core application files (which are likely changing in step) are grouped together in the app directory. The db items (which are less likely to change) are grouped together in the db directory. Finally, the infrastructure components (which rarely change) are in the infra directory. Like things are grouped together.

This can include directories for the respective Lambda function handlers, as well:

```
lib/
├── app
│   ├── api.ts
│   ├── app-stack.ts
│   ├── cronjob-stack.ts
│   ├── cronjob.ts
│   ├── frontend.ts
│   └── handlers
│       ├── cronjob.ts
│       └── proxy.ts
```

This structure isn't very different from standard file structures in other objected oriented languages. We just take all the files in a current directory and find methods of grouping them.

In addition, you can go a step further and put constructs in their own directories too. This can be a useful setup because it means you can easily take any of your constructs out of the existing project into a new one and publish it.

Author's Note

One reason I have thoroughly enjoyed working with the CDK is that it's the first Infrastructure as Code software I used where I felt I could easily refactor my code. Refactoring is taking code and moving it around without changing the eventual results. Same inputs, same outputs. Application development in Java, C#, Python, Typescript and other general purpose languages can be refactored easily thanks to the robust tooling that exists for those languages. Developers of existing IaC systems, like CloudFormation, Terraform, Serverless, and others have not had similar tooling available and refactoring code by hand can be tedious and often break things. With the CDK you can, and should, move code around as you find better ways of organizing and leveraging it.

With a construct-focused organization, you'd have a directory structure like this:

```
lib
├── app
│   ├── api
│   │   ├── api.ts
│   │   └── handlers
│   │       └── proxy.ts
│   ├── app-stack.ts
│   ├── cronjob
│   │   ├── cronjob.ts
│   │   └── handlers
│   │       └── cronjob.ts
│   ├── cronjob-stack.ts
│   └── frontend
│       └── frontend.ts
```

Where each construct is in its own directory and all files needed for that construct, like the definition file and handlers, are all in one place. An enormous benefit to this is that if you want to move that construct into its own library and publish it, you just copy the entire directory over to a new repository, and you're 95% done.

Ultimately, these are just a few suggestions, but you can organize your project however works for you and your team.

The only place that you have to care at all is in the cdk.json file, or when you override the application to synth using the `--app | -a` cli parameter.

5.4. CDK Code and Your Application Code

When you create a new CDK project, you may be tempted to create an entirely new source code repository (git) for it. However, there are a couple of cases where adding a CDK project to an existing repository could be useful.

- If you have a React, Vue, or other static website that you want to deploy to S3 Buckets, by including the CDK project in the same repository you

could create a CDK Pipeline and CICD system that will build and deploy that code easily to an S3 bucket using the 's3-deployment' module.

- If you're building an AWS Serverless application that makes use of Lambda functions, you can include the Lambda function handler code in the same project. You could then use CDK Pipelines to build and deploy those Lambda functions without having to package any ZIP files manually.

In both cases, there can be only one pipeline as the application and infrastructure are so intertwined, or 'tightly coupled', that changing one inherently changes the other.

However, when that's not the case, and your infrastructure (CDK code) and application are 'loosely coupled', those pipelines and repositories benefit from being separated.

For example, if you had your application deployed to an Elastic Kubernetes Service (EKS) cluster. Your application code would need a pipeline that builds a container image, publishes it, then updates the EKS cluster. You probably also want the CDK code to have a pipeline that would synthesize and deploy it. If you had both codebases in the same repository, then each pipeline would need to know how to respond only when changes to either the application code, or CDK code, happened. Pipelines don't enjoy having that responsibility, and this isn't trivial to implement.

So, if you have two separate pipelines for your application and CDK code, then it should be separate repositories. If you can get away with one pipeline, have one repository.

5.4.1. Further Examples

We've put together several examples of different projects arranged differently. Please check them out in The CDK Book Examples repository at

github.com/cdkbook/examples.

Examples from this chapter can be found in the 'project-layout' directory in the repository.

5.5. Review

While there are some files that you'll likely never have to move around, all the constructs and stacks you create should go into directories that allow for smart logical groupings of common functionality. What that looks like specifically will depend on what you're building and how your team best communicates the architecture.

6. Custom Resources and CFN Providers

What if the existing constructs and patterns are not enough? What if you want to provision resources that do not have CloudFormation support or are provided by other parties? If there is no underlying L1 construct, you have several possibilities to extend CloudFormation and AWS CDK functionality. In this chapter, we will look at three ways to provision custom objects.

- CloudFormation custom resources
- CloudFormation resource providers
- Interacting with the AWS API from CDK

6.1. Custom Resources

Custom resources allow you to add functionality to your CloudFormation templates with a custom logic that runs on every stack creation, update, or delete. They are implemented by an AWS Lambda function or any other software listening to an SNS topic. But in most cases, you will want to implement it as a Lambda function.

When implementing a custom resource, you provide logic to create a resource if it is added to the CFN stack template and logic to remove the resource if we delete the stack or resource snippet. You can also offer the possibility to update resources. After creating the implementing code, you can use custom resources using the `Custom::MyResource` resource type in CloudFormation to use your custom resource.

You can use the AWS CDK by creating a new `cdk.Resource` instance and configuring the backing Lambda function manually. The AWS CDK contains a provider framework to simplify the creation of custom resources and a particular implementation if you only want to call an AWS API in your custom logic.

6.1.1. Implementing custom resources using AWS CDK

The most basic way to create a custom resource is by defining a Lambda function and using the function's ARN as the so-called service token for your custom resource.

```
# Create a Lambda function that handles the lifecycle
my_function = lambda_.Function(self, "MyResourceFunction", ...
)

# invoke an AWS Lambda function when a lifecycle event occurs
CustomResource(self, "MyResource",
    service_token=my_function.function_arn
)
```

The lambda function then needs to handle the creation, update, and delete logic.

```
import * as response from 'cfn-response';

export async function handler(event:
AWSLambda.CloudFormationCustomResourceEvent, context: any) {
  try {
    switch(event.RequestType) {
      case 'Create':
        // implement creation of custom resource
        response.send(event, context, response.SUCCESS, responseData);
        break;
      case 'Update':
        // implement update of custom resource
        response.send(event, context, response.SUCCESS, responseData);
        break;
      case 'Delete':
        // implement deletion of custom resource
        response.send(event, context, response.SUCCESS, responseData);
        break;
      default:
        response.send(event, context, response.FAILED, responseData);
    }
  } catch (err) {
    response.send(event, context, response.FAILED, responseData);
  }
}
```

It is essential to ensure that all requests to this Lambda function are handled, and we respond to CloudFormation. Otherwise, the stack operation will wait several hours until the CloudFormation timeout kicks in.

The incoming event contains all necessary information to create the resource, and the response sent to CloudFormation then contains the physical resource identifier you assign to this resource. Any subsequent update or delete call provides this id so you can detect the correct resource to modify or remove. If you want to send properties to the custom resource, you can specify a key-value map in your construct. CloudFormation will then provide these values in the Lambda event for your handler.

```
# create custom resource with properties
CustomResource(self, "MyResource",
    service_token=my_function.function_arn,
    properties={
        "Name": "MyName",
        "Bucket": my_bucket.bucket_name
    }
)
```

These custom resources using the provider framework are utilized inside AWS CDK to provide functionality that is not (yet) available in CloudFormation. One example is the DnsValidatedCertificate construct that creates a potentially cross-region certificate inside the certificate manager. As CloudFormation is a regional service, a custom resource is used to provision a certificate in another region like `us-east-1` for CloudFront.

6.1.2. Using the CDK Provider framework

The approach mentioned above has several drawbacks. Most importantly, you need to ensure that all calls to the backing Lambda function send a response to CloudFormation to prevent hour-long timeouts. Additionally, you need to handle the parsing of the lifecycle events on your own. The AWS CDK contains a mini framework for custom resources called Custom Resource provider framework to avoid this undifferentiated heavy-lifting. With this toolset, you can implement your backing Lambda function in a request-response styled structure. In addition, the framework handles sending CloudFormation events and making sure all runtime errors are mitigated.

The CDK docs list the following benefits:

- Handles responses to AWS CloudFormation and protects against blocked deployments
- Validates handler return values to help with correct handler

implementation

- Supports asynchronous handlers to enable operations that require a long waiting period for a resource, which can exceed the AWS Lambda timeout
- Implements default behavior for physical resource IDs.

The AWS CDK code of the previous section will then look like this:

```
on_event = lambda_.Function(self, "MyHandler")

my_provider = cr.Provider(self, "MyProvider",
    on_event_handler=on_event
)

CustomResource(self, "MyResource", service_token=my_provider.service_token)
```

The backing Lambda function now implements the same lifecycle events, but you do not need to handle sending responses to CloudFormation manually, but return the response, and the Custom Resource framework will turn it into a correct answer. If anything goes wrong, the framework will automatically send an error response to prevent long timeouts and blocking the deployment.

```
export async function handler(event:
AWSLambda.CloudFormationCustomResourceEvent, context: any) {
  switch(event.RequestType) {
    case 'Create':
      // implement creation of custom resource
      return { PhysicalResourceId: physicalId };
    case 'Update':
      // implement update of custom resource
      return;
    case 'Delete':
      // implement deletion of custom resource
      return;
    default:
      throw new Error('Invalid event');
  }
}
```

We can find a real example of this framework in the IAM library of the AWS CDK. Creating OpenIDConnect provider in AWS CDK uses a custom resource as this feature is not yet available in CloudFormation. The implementation of the Lambda function is:

```
export async function handler(event:
AWSLambda.CloudFormationCustomResourceEvent) {
  if (event.RequestType === 'Create') { return onCreate(event); }
  if (event.RequestType === 'Update') { return onUpdate(event); }
  if (event.RequestType === 'Delete') { return onDelete(event); }
  throw new Error('invalid request type');
}

async function onCreate(event:
AWSLambda.CloudFormationCustomResourceCreateEvent) {
  const issuerUrl = event.ResourceProperties.Url;
  // ... more properties

  const resp = await external.createOpenIDConnectProvider({...});

  return {
    PhysicalResourceId: resp.OpenIDConnectProviderArn,
  };
}

async function onUpdate(event:
AWSLambda.CloudFormationCustomResourceUpdateEvent) {
  const issuerUrl = event.ResourceProperties.Url;

  // determine which update we are talking about.
  const oldIssuerUrl = event.OldResourceProperties.Url;

  // if this is a URL update, then we basically create a new resource and cfn
will delete the old one
  // since the physical resource ID will change.
  if (oldIssuerUrl !== issuerUrl) {
    return onCreate({ ...event, RequestType: 'Create' });
  }

  const providerArn = event.PhysicalResourceId;

  // do updates of other properties

  return;
}

async function onDelete(deleteEvent:
AWSLambda.CloudFormationCustomResourceDeleteEvent) {
  await external.deleteOpenIDConnectProvider({
    OpenIDConnectProviderArn: deleteEvent.PhysicalResourceId,
  });
}
```

If the update handler needs to create a new resource instead of modifying an existing one, the handler should only create the new resource and return the new physical id. CloudFormation will then call the delete method of the old one during the cleanup phase of the stack. This is important to maintain the rollback capabilities of the stack update.

6.1.3. Simple custom resources for AWS API calls

The AWS CDK has a special provider implementation for the particular case, where your only action inside the lifecycle event handlers is a single call to the AWS API.

Using the `AwsCustomResource` construct, you specify the API call and the arguments, and CDK will create the correct Lambda function for you. So for verifying an SES identity, you can write the following CDK code:

```
verify_domain_identity = AwsCustomResource(self, "VerifyDomainIdentity",
    on_create={
        "service": "SES",
        "action": "verifyDomainIdentity",
        "parameters": {
            "Domain": "example.com"
        },
        "physical_resource_id":
PhysicalResourceId.from_response("VerificationToken")
    },
    policy=AwsCustomResourcePolicy.from_sdk_calls(resources
=AwsCustomResourcePolicy.ANY_RESOURCE)
)

route53.TxtRecord(self, "SESVerificationRecord",
    zone=zone,
    record_name="_amazonses.example.com",
    values=[verify_domain_identity.get_response_field("VerificationToken")]
)
```

CDK will create a Lambda function that issues the `verifyDomainIdentity` API call and uses the returned token as a physical resource id. As no handlers

for onUpdate and onDelete are specified, nothing will happen on these lifecycle events. The CDK will also derive the necessary IAM permissions for the Lambda function from the API call to make sure the Lambda function may call the AWS API.

6.2. CloudFormation Resource Types

Another way of implementing new features to CloudFormation is by using the CloudFormation resource types. These are official or third-party extensions to the types available in CloudFormation and can be implemented by any person or company. Defining the resource's schema and some handler methods for CRUDL operations makes it possible to build entirely new CloudFormation types that behave like the official ones.

To write, package, and distribute your own resource types, you need to install the CloudFormation CLI and start a new project with it. Then, after implementing everything, you can deploy your new types to your own AWS account or make them available to other users using the public registry.

6.2.1. What is the CloudFormation registry

The public CloudFormation registry allows individuals and vendors to distribute resource types to a broad audience. You can register as an official publisher using your AWS Marketplace seller account or using GitHub. After successful registration, you can publish resource types into the public listing and let all users of AWS CloudFormation consume them. Resource types need to be published in all regions where customers want to use them. Users can then search these types, activate them into their AWS accounts and use them in their templates.

6.2.2. Brief How-to for CloudFormation CLI

To create your own resource type for private use or to be published into the registry, you start by creating a new project using the CloudFormation CLI. You can write your type handlers in TypeScript, Java, Python, and Golang.

The first step is to install the CLI and the language plugin of your choice and to initialize a new project.

```
pip install cloudformation-cli cloudformation-cli-typescript-plugin
cfn init
```

You need to define your schema that defines the properties the new resource type will offer users to customize the resource. That also declares the identifiers and attributes you can access after creation. The required permissions are also declared in this configuration file if the resource handlers need to access the AWS API.

Now you can implement the handler methods for creation, updating, deletion, and reading of resources. If you also implement a list handler, CloudFormation will support importing existing resources into a stack for your resource type.

For details on implementing custom resource types, refer to the documentation of the CloudFormation CLI and the language plugin of your choice.

- CLI: https://github.com/aws-cloudformation/cloudformation-cli
- TypeScript: https://github.com/aws-cloudformation/cloudformation-cli-typescript-plugin/
- Python: https://github.com/aws-cloudformation/cloudformation-cli-python-plugin/

- Java: https://github.com/aws-cloudformation/cloudformation-cli-java-plugin/
- Golang: https://github.com/aws-cloudformation/cloudformation-cli-go-plugin/

6.2.3. Creating an L1 construct for a custom type

To use a third-party resource type in AWS CDK there are several options.

- Define the resource using the basic `CfnResource` class in your stack
- The vendor of the type provides a CDK construct library to consume
- Generate an L1 from the resource type definition using `cdk-import`

Using CfnResource

The most straightforward way to use a custom resource type is to leverage the `cdk.CfnResource` class inside the core library. You can specify the name of the type and fill out all properties that are sent to the type's method handlers. The downside of this approach is that you do not get type-completion as the schema of the properties is not known to AWS CDK.

```
CfnResource(self, "MyResource",
    type="Vendor::My::Resource",
    properties={
        "Name": "MyResource",
        "Enabled": True
    }
)
```

This code snippet will render to the following resource definition in the synthesized CloudFormation template:

```
"MyResource": {
  "Type": "Vendor::My::Resource",
  "Properties": {
    "Name": "MyResource",
    "Enabled": true
  },
  "Metadata": {
    "aws:cdk:path": "my-stack/MyResource"
  }
},
```

Using vendor libraries

Suppose you are the developer or vendor of the CloudFormation resource type. In that case, publish a CDK construct library that users can import that wraps this code into a typed L1 construct that provides type completion and documentation. Users can then consume the type above using code that might look like this:

```
CfnVendorMyResource(self, "MyResource",
    name="MyResource",
    enabled=True
)
```

Generate code from the schema definition

If the vendor of the resource type does not provide a construct library or you want to write one, you can generate the level 1 construct for any resource type using the `cdk-import` tooling. Make sure to be logged in to your AWS account on the command line and run the import command to generate the L1 code.

```
npx cdk-import -o src/generated 'Datadog::Integrations::AWS'
```

This will search for an available resource type in the public CloudFormation

registry, download the schema definition, and generate a file called `vendor-my-resource.ts` in the given output folder. As mentioned in the last section, this file provides all necessary classes and interfaces to use the resource type. You can then put this file into a JSII project and publish this library to all package managers for consumption by your users.

For more details about packaging construct libraries, see chapter publishing constructs

The CDK team also publishes all publicly available resource types under the cdk-cloudformation name-space. You can install them using the package manager of your choice. (e.g. `@cdk-cloudformation/datadog-integrations-aws`)

6.3. Interacting with the APIs during CDK deployments

There are several reasons you might want to interact with the AWS API during an infrastructure deployment that have nothing to do with creating resources but are meant to validate the state of resources or to notify third-party systems about updates.

Custom resources in CloudFormation are a great way to implement these tasks, and the following section will guide you through the usage.

6.3.1. Using a custom resource as a hook during the deployment lifecycle

Using custom resources as a post-deploy hook is done by:

- implementing the hook as a custom resource handler
- using `DependsOn` to configure the point at which the hook is fired
- Making sure the hook is fired during each update

An easy way to implement the hook is to put your code into the update handler and call it during create and update. You can leave the delete handler empty if you do not want to do something special on tear-down. A sample code if this using the CDK provider framework could look like this:

```
export async function handler(event:
AWSLambda.CloudFormationCustomResourceEvent) {
  if (event.RequestType === 'Delete') { return; }

  // implement your hook code here

  return {
    PhysicalResourceId: 'SomeUniqueIdForThisHook',
    ...event,
  };
}
```

To add this hook into your CDK application, you can then instantiate a custom resource.

```
on_event = lambda_.Function(self, "MyHandler")

my_provider = cr.Provider(self, "MyProvider",
    on_event_handler=on_event
)

hook = CustomResource(self, "MyResource",
    service_token=my_provider.service_token,
    timestamp=Date().get_time()
)
hook.node.add_dependency(other_cdk_construct)
```

To manage the point in time the hook is fired, add dependencies on all resources that need to be there before the hook is executed, and to make sure

the hook fires on every update, add a property that contains the current timestamp. This lets CloudFormation believe that an update is needed, and it will call your handler method.

Additionally to the timestamp, you can add any other properties you want to use. For example, if your hook tests an API you can reference the endpoint and use it in your hook to call the service and validate the response. There is an example of this in the testing chapter.

```
# ...

hook = CustomResource(self, "MyResource",
    service_token=my_provider.service_token,
    timestamp=Date().get_time(),
    endpoint=http_api.api_endpoint
)
```

And in your handler code, use it like this and fail if necessary to trigger a CloudFormation rollback:

```
export async function handler(event:
AWSLambda.CloudFormationCustomResourceEvent) {
  if (event.RequestType === 'Delete') { return; }

  const endpoint = event.ResourceProperties.endpoint;
  // call the endpoint using any HTTP library and validate correctness
  // throw an Error if this fails to trigger a rollback

  return {
    PhysicalResourceId: 'SomeUniqueIdForThisHook',
    ...event,
  };
}
```

6.4. Review

Custom resources and custom providers are powerful ways to extend the functionality of the CDK. If possible you should prefer resource types over custom resources as they have a clear schema, lifecycle, and ownership.

7. Configuration Management

Building AWS infrastructure requires setting a lot of properties on constructs. Let's take a look at a simple example:

```
import { App } from 'aws-cdk-lib';
import { SimpleApiWithTestsStack } from './SimpleApiWithTestsStack';

// for development, use account/region from cdk cli
const devEnv = {
  account: process.env.CDK_DEFAULT_ACCOUNT,
  region: process.env.CDK_DEFAULT_REGION,
};

const app = new App();

new SimpleApiWithTestsStack(app, 'SimpleApiWithTests', { env: devEnv });

app.synth();
```

In this case, nothing is really passed to the `SimpleApiWithTestStack` construct, other than the account and region to deploy to. Everything that stack creates will be the same no matter how many accounts and regions this stack is deployed to. Everytime the stack is synthesized, the results will be the same. You can get away with this in some cases but more often then not you'll need to pass some properties to your stacks. For example:

```
import { App } from 'aws-cdk-lib';
import { DbStack } from './DbStack';

// for development, use account/region from cdk cli
const devEnv = {
  account: process.env.CDK_DEFAULT_ACCOUNT,
  region: process.env.CDK_DEFAULT_REGION,
};

const app = new App();

// Create our RDS instance
new DbStack(app, 'Db', {
    env: devEnv,
    vpcId: 'vpc-2f09a348'
});

app.synth();
```

Our DbStack now gets a vpcId to deploy the RDS instance to.

```
import { App } from 'aws-cdk-lib';
import { InstanceClass, InstanceSize, InstanceType } from 'aws-cdk-lib/aws-
ec2';
import { DbStack } from './DbStack';

// for development, use account/region from cdk cli
const devEnv = {
  account: process.env.CDK_DEFAULT_ACCOUNT,
  region: process.env.CDK_DEFAULT_REGION,
};

const app = new App();

// Create our RDS instance
new DbStack(app, 'Db', {
    env: devEnv,
    vpcId: 'vpc-2f09a348',
    instanceType: 't3.micro',
});

app.synth();
```

Now there is an instanceType that defines how large the database instance

should be. Over time you'll likely have to pass a lot of properties to stacks.

These types of properties come in useful when reusing a stack across multiple environments:

```
import { App } from 'aws-cdk-lib';
import { InstanceClass, InstanceSize, InstanceType } from 'aws-cdk-lib/aws-
ec2';
import { DbStack } from './DbStack';

const devEnv = {
  account: process.env.CDK_DEFAULT_ACCOUNT,
  region: process.env.CDK_DEFAULT_REGION,
};

const prodEnv = {
  account: '123456789012',
  region: 'us-east-1',
};

const app = new App();

// Create our DEV RDS instance
new DbStack(app, 'DevDb', {
    env: devEnv,
    vpcId: 'vpc-2f09a348',
    instanceType: 't3.micro',
});

// Create our prod RDS instance
new DbStack(app, 'ProdDb', {
    env: prodEnv,
    vpcId: 'vpc-abcd0123',
    instanceType: 'r5.xlarge',
});

app.synth();
```

Now the first stack is synthesized with values for the first VPC and a 't3.micro' instance size, suitable for development environments. The second stack, in the production environment, will go in a different VPC and with a much larger database, so it can handle the large amount of traffic production environments often get.

The properties you pass to your stacks (and on to your underlying constructs) are considered configuration. How you define those values and maintain them over time is configuration management.

7.1. Static Management

The most common method is through a static configuration management. The values don't change frequently. The previous examples all use static values, where the properties we want to pass to the stack are written explicitly in the code:

```
new DbStack(app, 'Db', {
    env: prodEnv,
    vpcId: 'vpc-abcd0123',
    instanceType: 'r5.xlarge',
});
```

InstanceType and vpcId (and env) are all static values that don't change (until you change the code).

Keeping your configuration static has two primary characteristics:

- It doesn't change unexpectedly.
- It doesn't change easily.

The not changing unexpectedly is good. We're dealing with infrastructure and sometimes changes can cause infrastructure to be recreated, and you never want that to happen unexpectedly.

For example, if you were to accidentally change the AMI used by an EC2 instance then CloudFormation would recreate that instance. If that happened in a production system you might have an outage and then people would be upset.

Having code that doesn't change unexpectedly is good, great really. Programmers have had a term for this for years called deterministic. Deterministic software always produces the same output given the same input.

You may be asking "but isn't it easy to change my code"? Sure, it is! But, the people who often decide those values (like instanceTypes) aren't always the same people who are responsible for writing the code. Also, some of those values may need to change together, like the environment and the instance type. If you change one, you likely have to change the other.

So there is always some handoff, where someone says "we need the instance to be bigger" so a ticket is added to a system somewhere and later the DevOps engineer goes into the CDK code and changes the value. This isn't the type of exciting work you probably want to do.

Traditionally this is the only way to manage configuration. In CloudFormation there are Stack Parameters that define these types of values. In other systems, like Terraform, configuration is managed inline like the examples above.

However, this is the CDK! Because it is using general purpose programming languages we can leverage a lot of different ways to manage this configuration. By picking the right system for your situation you can make long term maintenance of your CDK code easier and keep things deterministic.

Let's look again at the previous example:

```
// Create our DEV RDS instance
new DbStack(app, 'DevDb', {
    env: devEnv,
    vpcId: 'vpc-2f09a348',
    instanceType: 't3.micro',
});

// Create our prod RDS instance
new DbStack(app, 'ProdDb', {
    env: prodEnv,
    vpcId: 'vpc-abcd0123',
    instanceType: 'r5.xlarge',
});
```

We've got 4 values we're tracking here, a vpcId and instanceType across two environments (two properties times two environments, 2 x 2 = 4).

But what if there were a lot more properties? What if there were 10? And what if we had 4 environments, like dev, qa, staging, and production? Now we have 40 properties we need to manage. We can start to reorganize our code to make those 40 values easier to read and manage.

Side Note

When I was first introduced to the CDK I was working on a project that had 30 databases per environment. Each database had an engine type (mysql or postgres), an instanceType, initial storage size, and few other properties. There were also 7 environments in use. In total there were a few hundred configuration points to keep track of.

The power of using general purpose languages really starts to shine here, as we're able to take advantage of all the tools and patterns available to us to decide the best way to maintain this configuration

7.1.1. Context Variables

Context variables are another way you can manage configuration data. For

example, let's look at our previous example:

```
// Create our DEV RDS instance
new DbStack(app, 'DevDb', {
    env: devEnv,
    vpcId: 'vpc-2f09a348',
    instanceType: 't3.micro',
});

// Create our prod RDS instance
new DbStack(app, 'ProdDb', {
    env: prodEnv,
    vpcId: 'vpc-abcd0123',
    instanceType: 'r5.xlarge',
});
```

If we wanted to, we could use context variables to set the vpcId of the stacks:

```
const app = new App();
const devVpcId = app.node.tryGetContext('dev-vpc-id') ?? 'vpc-2f09a348';
const prodVpcId = app.node.tryGetContext('prod-vpc-id') ?? 'vpc-abcd0123';

// Create our DEV RDS instance
new DbStack(app, 'DevDb', {
    env: devEnv,
    vpcId: devVpcId,
    instanceType: 't3.micro',
});

// Create our prod RDS instance
new DbStack(app, 'ProdDb', {
    env: prodEnv,
    vpcId: prodVpcId,
    instanceType: 'r5.xlarge',
});
```

Now that our code reads context variables we need a way to set them when we synthesize the code. There are a few different ways to set context variables. We can do it from the command line when synthesizing the code:

```
cdk synth -c dev-vpc-id=vpc-4321abcd -c prod-vpc-id=vpc-9876edcb
```

We can also set those context variables in the cdk.json file:

```
{
  "app": "npx ts-node --prefer-ts-exts src/main.ts",
  "context": {
    "dev-vpc-id": "vpc-4321abcd",
    "prod-vpc-id": "vpc-9876edcb"
  }
}
```

If you're using projen, then you wouldn't edit this file directly, you would set the `context` property on your projen class:

```
const project = new AwsCdkTypeScriptApp({
  cdkVersion: '2.0.0-rc.10',
  defaultReleaseBranch: 'main',
  name: 'infra-apig',

  cdkDependencies: [],
  deps: ['axios', '@aws-sdk/client-dynamodb'],
  context: {
    'dev-vpc-id': 'vpc-4321abcd',
    'prod-vpc-id': 'vpc-9876edcb',
  },
});
```

However, context variables don't scale particularly well. For example, if you want to represent deep structures in your configuration you'd need to use a naming convention on your variables, something like:

```
{
  "app": "npx ts-node --prefer-ts-exts src/main.ts",
  "context": {
    "dev_vpc-id": "vpc-4321abcd",
    "dev_app_security-group": "sg-012345678'",
    "prod_vpc-id": "vpc-9876edcb",
    "prod_app_security-group": "sg-abcd01234'"
  }
}
```

While certainly possible, it's not always the easiest to read and write, and small typos and mistakes can lead to consistency problems later. Context variables work well for small amounts of configuration but if you are dealing with a lot, it's probably best to move on.

In fact, the various service modules, like the EC2 module, sometimes needs to store this kind of configuration information, like a resolved VPC attributes. Just as you can provide context variables at synthesize time either through CLI parameters or the cdk.json file, some constructs will write their own.

Anytime you use a `.from*` function, like `Vpc.fromLookup()` the construct needs to read the AWS API to get information it needs to do its job. It then writes that information into the `cdk.context.json` file, storing it for future use as context variables. Future synthesizes of the construct will result in that cache from the cdk.context.json file being read and used instead of making an API call. This is done specifically to ensure that any of the `.from*` functions you use stay deterministic. It inherently becomes part of the input to your application.

Remember to commit your cdk.context.json file with your codebase so all users and automation leverages that same data, ensuring determinism.

7.1.2. Static Files

Because we're dealing with general purpose languages we can move this data around and store it anywhere we want that the language supports.

The first direction you may go with this is to refactor all of these properties into a separate file. That file is then loaded and the properties passed to our stacks. Let's look at the previous example refactored in this way:

```
const devProperties = require('./env/dev.json');
const prodProperties = require('./env/prod.json');

// Create our DEV RDS instance
new DbStack(app, 'DevDb', {
    env: devEnv,
    ...devProperties,
});

// Create our prod RDS instance
new DbStack(app, 'ProdDb', {
    env: prodEnv,
    ...prodProperties
});
```

In this case, the configuration is stored in files specific to the environment they're for. Those files are read and the values are passes to the stacks.

The dev.json file looks like:

```
{
  "vpcId": "vpc-2f09a348",
  "instanceType": "t3.micro"
}
```

And the prod.json file:

```
{
  "vpcId": "vpc-abcd0123",
  "instanceType": "r5.xlarge"
}
```

Now all the configuration for a specific environment is in a single file per environment. By placing it all in one location knowing where to go to change something in an environment is clear and easy.

However, there is one place this falls apart a little bit. In this case the instanceType is just a string, 't3.micro' or 'r5.xlarge'. String values like this are easy to type in wrong, putting in 't3,micro' by accident and leading to errors. IDE's and linters are notoriously bad at catching this type of error.

Static files are great when you've got a lot of strings you're passing around that you're just copying/pasting or that are hard to get wrong.

7.1.3. Less Static-y Files

Static .json files don't let us specify more complex types, like an InstanceType class. InstanceType allows us to strongly define our database instance in a way that doesn't allow easily for typos. It also allows us to define things with code that IDEs and linters are able to tell us quickly are wrong.

We can gain some type safety if we instead change to using .ts files:

```
import { InstanceClass, InstanceSize, InstanceType } from 'aws-cdk-lib/aws-
ec2';
export const config = {
    vpcId: 'vpc-abcd0123',
    instanceType: InstanceType.of(InstanceClass.R5, InstanceSize.XLARGE),
};
```

Now we have some type safety and take advantage of all the power of

Typescript and strongly-typed languages to guarantee we get good values. The import of this resource only changes slightly:

```
import { config as devProperties } from './env/dev';
import { config as prodProperties } from './env/prod';

// Create our DEV RDS instance
new DbStack(app, 'DevDb', {
  env: devEnv,
  ...devProperties,
});

// Create our prod RDS instance
new DbStack(app, 'ProdDb', {
  env: prodEnv,
  ...prodProperties,
});
```

The advantage here is now our dev.ts and prod.ts files can do more complex logic and represent more complex types than we get with json. For example, maybe we want some of these values to be overridden from environment variables:

```
import { InstanceClass, InstanceSize, InstanceType } from 'aws-cdk-lib/aws-
ec2';
export const config = {
    vpcId: process.env.DEV_VPC ?? 'vpc-abcd0123',
    instanceType: InstanceType.of(InstanceClass.R5, InstanceSize.XLARGE),
};
```

Now this configuration will look for the `DEV_VPC` environment variable and use it if set. If not set, the `??`, the value `vpc-abcd0123` is used. When synthesizing this CDK code, you would override the environment variable like so:

```
$ DEV_VPC=vpc-dcba9876 cdk synth
```

This can come in useful when you're leveraging environment variables in your CICD environment, like CodePipeline, Github Actions, or CircleCI pipelines.

For example, CodeBuild manages a lot of useful environment variables already, like the CODEBUILD_RESOLVED_SOURCE_VERSION variable. You could pass this value in to your CDK code to use in Tags on all your resources.

Always remember to sanitize and check your inputs:

```
const vpcId = process.env.VPC_ID;
if (!vpcId || !isValidVpcId(vpcId)) {
    throw new Error("Please provide a valid VPC_ID environment variable");
}
```

As soon as you start creating these additional input values, either through context variables or environment variables, you are affecting the way the CDK code synthesizes. It's important to keep this in mind as you should aim for your CDK code to be deterministic.

Deterministic software will always produce the same output with the same input.

This is a fundamental principle of the CDK.

If you start introducing new inputs to your code it's important to remember that they affect the way the code is synthesized into CloudFormation resources.

Let's go back to the idea of using an environment variable to provide a vpcId as input. If the same environment variable is provided every time, then you will always get the same output. However, it's very easy for environment variables to get set incorrectly (or not at all), in which case your output will be

different. This could cause some nasty side effects in the system. If the vpcId accidentally changes because of modifications to your pipeline then the stack might fail. Failure would actually be the best result, the worst probably being that it would succeed and just recreate your resources in another VPC which is probably going to cause a lot of problems.

The code is properly deterministic, a change of input created a change of output (resources in another VPC). But it doesn't excuse you from being mindful of how easily, and destructively, your inputs can change.

Let's look at another example, because we really want to drill this point home. Imagine you have introduced an input that determines if an RDS instance is created. Maybe some deployments of your CDK code would require a database to exist (production) but some environments wouldn't require one, like a development environment where the database may be either shared or mocked. If this input is from an environment variable and due to a bug in your CICD process it gets changed unexpectedly, you could wipe out a production database.

To prevent this, make sure that you are always providing your inputs in a way that is very hard for them to be wrong. Storing configuration in static files that are included in your codebase, like the .json examples above, is a great way to do this. For those values to change they likely need to go through a code review process where someone or something has the ability to go "nuh uh uh".

Other input methods, like context variables or environment variables should have adequate validation.

7.2. Dynamic Management

There is a more dynamic way you can also get the configuration for your code. Because the CDK is written for your programming language of choice,

there is almost no limit to where you store your configuration.

Let's go back to the example where we have our RDS instance configuration stored in JSON files:

```
{
  "vpcId": "vpc-2f09a348",
  "instanceType": "t3.micro"
}
```

JSON is a highly portable and frequently used format for data transfer. If we wanted, this data could be retrieved from a service instead of a local file:

```
axios.get('https://someconfig.mycompany.com/dev')
    .then(devConfig => {
        const app = new App();
        const stack = new DbStack(app, 'DevDb', results);
        app.synth();
    });
```

Here the 3rd-party `axios` library, a simple HTTP client, makes a call to some config server at our company to retrieve the config information and then creates the dev DB Stack.

As you can see, this is very powerful!

However, it's also very dangerous.

Remember what we said before about determinism? This kills determinism.

Each time this code is run, there is no guarantee that the results from the request to 'someconfig.mycompany.com/dev' will return the same results. If the same results don't come back each time, then you have different input each time. If you have different input, your stack will be changing when you didn't expect it to.

While determinism is a great goal, you may find occasions where it might be worth breaking it.

Risk from losing determinism can be reduced by putting a manual approval step into your deployment pipeline so that unexpected changes can get caught before they affect an environment. In static configuration management this is done through the code review process (a Pull Request or otherwise).

You can also reduce your risk by making this type of retrieval less dynamic. You can take a page from the cdk.context.json file and the `.from*()` lookup functions and store the data locally.

This gives you the power of driving your CDK code from API calls while keeping your determinism. Essentially, we just automate the maintenance of static configuration management.

Let's take our previous example of retrieving data from an API:

```
axios.get('https://someconfig.mycompany.com/dev')
    .then(devConfig => {
        const app = new App();
        const stack = new DbStack(app, 'DevDb', results);
        app.synth();
    });
```

To make this more deterministic this code would be rewritten slightly to save the data to a file:

```
axios.get('https://someconfig.mycompany.com/dev')
    .then(devConfig => {
        fs.readFileSync('./env/dev.json', devConfig);
    });
```

Then we would go back to using a static file read like we had before:

```
const devProperties = require('./env/dev.json');

// Create our DEV RDS instance
new DbStack(app, 'DevDb', {
    env: devEnv,
    ...devProperties,
});
```

The retrieval of data is now two separate steps with the first writing results to a cache for the second to use. By making this two separate steps we've reduced the chance of an unexpected change, since synthesizing the code will always result in the same output as it's driven from static configuration again. If we want different output, we need different input, and the code to pull the configuration from the 'someconfig.mycompany.com/dev' endpoint needs to be run.

This can be accomplished pretty easy using npm scripts (or similar in Python and other languages). If we assume the code is moved into a file called 'refreshConfig.js', then:

```
{
  "scripts": {
    "refresh:config": "node refreshConfig.js"
  }
}
```

If you're using projen, you'd set the 'scripts' property.

```
const project = new AwsCdkTypeScriptApp({
  ...
  scripts: {
    'refresh:config': 'node refreshConfig.js',
  },
});
```

When the underlying config is changed, a simple `npm run refresh:config`

(or `yarn refresh:config`) will retrieve the data from the API and store it in the JSON file to be used by the CDK code during the next `.synth()`.

At first, I was against this extra step, but after a number of times when I got nervous about deploying CDK code to an environment I appreciated the value of having all input in static files. Yes, I may have to run a script to update that data, but it's a small price to pay to know I'll never have an outage from an unexpected change.

Being able to leverage data from external systems is one of my favorite parts of the CDK, and most other Infrastructure as Code has really done well.

7.3. Application Configuration

So far I've covered all of this configuration around your CDK code, but we haven't talked about application configuration code. Application configuration are those things like database connection strings, external service URLs, and other things your application code needs to be told to function within an environment.

While nothing the CDK does is specifically aimed at solving this problem, I figure it's good to still cover the basics.

First, let's step back and talk about the AWS services that allow you to store and retrieve these types of values. There is the Systems Manager Parameter Store and the Secrets Manager. Both serve similar purposes but are integrated differently into different systems.

7.3.1. Systems Manager Parameter Store

Older than the Secrets Manager service, the Parameter Store is tightly integrated into CloudFormation.

You may have used them before with CloudFormation. You can specify a template parameter value coming from Parameter Store:

```
# Reference/use existing Systems Manager Parameter in CloudFormation
Parameters:
  InstanceType :
    Type : 'AWS::SSM::Parameter::Value<String>'
    Default: myEC2TypeDev
  KeyName :
    Type : 'AWS::SSM::Parameter::Value<AWS::EC2::KeyPair::KeyName>'
    Default: myEC2Key
```

In this case, the InstanceType and KeyName value will be retrieved from the Parameter Store using the keys 'myEC2TypeDev' and 'myEC2Key', respectively.

While the CDK generates CloudFormation, it's actually rare that you ever set up Parameters for your Stacks and even more rare that you'd use the Parameter Store as a source for those values. As we've already discussed, handing values to constructs is typically done through properties and CloudFormation Parameters becomes superfluous.

Parameter Store values can be used in many other ways, like in your applications themselves, if your application code is making the necessary API calls to get them.

In terms of the CDK code, you can create these Parameter Store values very easily:

```
new StringParameter(stack, 'Parameter', {
  allowedPattern: '.*',
  description: 'The value Foo',
  parameterName: 'FooParameter',
  stringValue: 'Foo',
  tier: ParameterTier.ADVANCED,
});
```

Notice here that you're providing the value for the parameter directly in the code: 'Foo'. This means you shouldn't be using this for any secret values, like passwords, api keys, or other sensitive information you don't want committed to your repository or showing up in templates. However, this does work great for other values that aren't so secret, like URL endpoints and feature flags.

7.3.2. Secrets Manager

Secrets Manager is a purpose-built storage service for values that would be considered 'secret' or sensitive. While very similar in features and functionality to Parameter Store, there are some key differences.

First, Secrets Manager is the preferred method for handling what Parameter Store values did for a long time. The security, encryption, and safety of Secrets in Secrets Manager is better that Parameter Store. When possible, learn towards using them over Parameter Store values.

Second, it's a bit trickier to set the value and that's good. You don't want sensitive values hardcoded into your templates, and allowing you to specify a sensitive value in your CDK code results in it showing up in your template.

While a `StringParameter` gives you the `stringValue` property to set the value, Secrets does not.

Technically you can do it, but shouldn't:

```
// Templated secret
const templatedSecret = new secretsmanager.Secret(this, 'TemplatedSecret', {
  generateSecretString: {
    secretStringTemplate: JSON.stringify({ password:
'myactualpasswordnowexposedtotheworld' }),
    generateStringKey: 'ignorethis',
  },
});
```

This will create the secret with a value of:

```
{
  "password": "myactualpasswordnowexposedtotheworld",
  "ignorethis": "somerandomstringitgenerated"
}
```

But it will also create a template resource with your password in it:

```
{
  ...
  "SecretA720EF05": {
    "Type": "AWS::SecretsManager::Secret",
      "Properties": {
        "GenerateSecretString": {
          "GenerateStringKey": "ignorethis",
          "SecretStringTemplate":
"{\"password\":\"myactualpasswordnowexposedtotheworld\"}"
        }
      },
    "Metadata": {
      "aws:cdk:path": "my-stack-dev/Secret/Resource"
    }
  }
}
```

We could just ignore the 'ignorethis' field as the 'password' is really what we cared about. It's doable when the value you're caring about is in a json format and is not sensitive. It's a hack, and I don't recommend you do it, but it's possible.

Ideally, your secrets will be integrated through your environment well enough that the value can be wrong temporarily, and you edit it later.

For example, if you're setting up a CodePipeline that reads from Github, you'll likely need to have a Secret that represent your Github OAuth Token:

```
const sourceAction = new codepipeline_actions.GitHubSourceAction({
  oauthToken: cdk.SecretValue.secretsManager('my-github-token'),
  ...
});
```

In this case you'll need to have a Secret in Secrets Manager:

```
const secret = new secretsmanager.Secret(this, 'Secret', {
  secretName: 'my-github-token'
});
```

When this code first gets deployed the value of the `my-github-token` secret will be generated and incorrect. If the pipeline tries running, it will error.

At this point you'd need to go take your generated key from Github and manually update the Secret value in AWS, by using the UI console or the API:

```
aws secretsmanager update-secret --secret-id my-github-token --secret-string
myactualgithubtoken
```

Once the value is updated, the pipeline could be restarted and should run. A minor inconvenience, but right now there isn't a much better way to get it done.

Technically, there is a way you could set a secret's value in Secrets Manager without having the value exposed in the template. It involves a Custom Resource, Lambda functions, and a central place to store the value you want (in which case, why are you using Secrets Manager?). I challenge you to think about how that would work.

7.4. Review

Managing your configuration can start small. Hard code values in the beginning. As they grow and get more complex, pull them into a common file like those `./env/*.json` files mentioned in the Static Management section.

Your IDE likely has tools that makes it easy to refactor these types of values around, so you're not manually trying to copy, paste, rename, and restructure code. Doing it manually can introduce errors easily and make you not want to continue trying. Refer to the Integrated Development Environment chapter on tips on how to use these tools.

If those inputs get too large or need to come from other places, convert them to something more dynamic like discussed in the Dynamic Management section.

But, try to maintain a determinism by using stored context and caches of data so that you're never caught off guard by unexpected changes that can really ruin a week.

8. Assets

The CDK lets us build infrastructure using programming languages that we are familiar with. But what about the actual application itself? Consider when we are working on a containerized application to be deployed to AWS Fargate. CDK lets us describe the infrastructure required to deploy the application in the same language as that is used to build the application.

However, when it comes to building the Docker image that is used to deploy the application to ECS, this still needs to be built in a separate step. Wouldn't it be great to have the Docker image, or any other local file required by the app to be managed and built by CDK itself? Assets solve this problem.

8.1. What Are Assets

Assets are files that may be required to support or run the application. Assets can be files or a directory that contains the code for the AWS Lambda handler function, or even Docker Images. These assets can be referenced using constructs that are exposed by the CDK.

Currently, the CDK supports two types of assets:

- Docker Image Assets
- Amazon S3 Assets

Before we dive into both, let's talk about bootstrapping for a bit, as to use assets in the stack, an environment needs to be bootstrapped.

8.2. Bootstrapping

For the CDK to deploy CDK apps, it needs to store the resources somewhere. This is particularly the case when:

- We have a CDK stack that makes use of assets
- We have a CDK app that generates a CloudFormation template that is greater than 50 kB
- We make use of CDK Pipelines or another feature that makes use of the `DefaultStackSynthesizer` [3].

The step in which the initial resources, the S3 bucket and ECR repository, are provisioned is called Bootstrapping. Bootstrapping creates the resources using a CloudFormation stack, with the name CDKToolkit.

If we attempt to use one of the features that require an environment to be bootstrapped without having the bootstrap completed, CDK will complain about this and will not let us proceed further:

```
npx cdk deploy
DockerAssetStack failed: Error: This stack uses assets, so the toolkit stack
must be deployed to the environment (Run "cdk bootstrap")
```

We can bootstrap using the `cdk bootstrap` command:

```
npx cdk bootstrap
```

CDK will use the region defined in our profile. If we have an AWS_REGION environment variable set, that value will be used instead. We can choose to be more explicit and mention the account and region to bootstrap:

```
npx cdk bootstrap aws://<account-number>/<region-name>
```

If you prefer to bootstrap multiple accounts or multiple regions at a time, then you can bootstrap them all by providing the account number and the region as parameters:

```
npx cdk bootstrap aws://<account-number-1>/<region-name-1> aws://<account-
number-1>/<region-name-2> aws://<account-number-2>/<region-name-1>
```

The bootstrap command creates a CloudFormation template and submits it to the AWS account to create the bootstrap stack. If you're curious which resources will be deployed, you can inspect the bootstrap CloudFormation template using the `--show-template` option:

```
npx cdk bootstrap --show-template
```

8.2.1. Legacy and Modern Bootstrap Templates

Depending on what version of CDK you're using, and/or what features of CDK you're using, CDK might create the bootstrap stack using the legacy or modern bootstrap template. Currently, the CDK uses legacy template by default, with the modern template being applied if you're using CDK Pipelines (we cover CDK Pipelines in the Deployment Strategies chapter).

The modern bootstrapping template adds support for cross-account deployments, a versioned bootstrap stack. In addition, the Modern template gives us the option to assume a custom role for deploys to improve security and compliance posture.

The modern bootstrapping template will be the default bootstrapping

template starting with CDK Version 2.0. Currently, setting the `CDK_NEW_BOOTSTRAP` will trigger CDK bootstrap to use the modern template.

```
CDK_NEW_BOOTSTRAP=1 npx cdk bootstrap --show-template
CDK_NEW_BOOTSTRAP set, using new-style bootstrapping
```

8.2.2. Customizing Bootstrap Templates

If your organization has naming policies and conventions in place, and the names generated by the bootstrap stack are not in line with them, then bootstrap resources can be customized. CDK lets us customize resources created by the bootstrap process using

- Command-line arguments to the `cdk bootstrap` command
- Directly modifying the Bootstrap CloudFormation template

Customizing using cdk bootstrap

The `cdk bootstrap` command can accept some command-line options for overriding the bootstrap template. These include

- `--bootstrap-bucket-name`: Override the name of the S3 bucket where the assets are stored
- `--bootstrap-kms-key-id`: Override the AWS KMS key used to encrypt the bootstrap bucket.
- `--tags`: Add tags to the bootstrap template.
- `--termination-protection`: Provides an option to enable/disable termination protection, which determines whether or not a CloudFormation stack can be deleted.

For example, if we need add two tags, a "used_for" and a "created_by" tag to the bootstrap resources, use the `--tags` override as shown below:

```
npx cdk bootstrap --tags used_for=bootstrap --tags created_by=sathya
aws://<account-number>/<region>
```

After pressing enter, CDK will create a CloudFormation changeset with the changes. The response from the command should be similar to as shown below:

```
Bootstrapping environment aws://<account-number>/<region-name>...
CDKToolkit: creating CloudFormation changeset...

Environment aws://<account-number>/<region-name> bootstrapped.
```

If Termination Protection needs to be enabled on the stack, the bootstrap command will look like below:

```
npx cdk bootstrap --termination-protection=true aws://<account-
number>/<region-name>
```

CDK will create a CloudFormation changeset to update the stack:

```
Bootstrapping environment aws://<account-number>/<region-name>
CDKToolkit: creating CloudFormation changeset...

Environment aws://<account-number>/<region-name> bootstrapped (no changes).
```

If you choose to bootstrap the modern template, some additional override options are available. These include:

- `--cloudformation-execution-policies`: which lets us override what IAM policies should be assumed when bootstrap stacks are deployed.

- `--qualifier`: Qualifiers are unique strings appended to bootstrap resources. Qualifiers are used to uniquely identify bootstrap stacks, when there are multiple bootstrap stacks in the same environment (ie, in the same AWS account and region combination).

8.2.3. Customizing the CloudFormation template

Customizing the bootstrap stack using the `cdk bootstrap` multiple times can get tedious. The CDK lets us generate the bootstrap template, customize it to our needs, and use this template for bootstrapping. To do this, pass the path of the template using the `--template` command-line option:

```
npx cdk bootstrap --template cfn-template.yaml
```

Rather than creating the bootstrap template from scratch, we can make use of the default template by the `--show-template` command and use that as the basis for the custom bootstrap template.

Grab the template first and save it `cfn-template.yaml`:

```
npx cdk bootstrap --show-template >  cfn-template.yaml
```

Once you have customized the template, you can bootstrap using the `--template` command-line option: as shown above.

8.3. Docker Image Assets

Docker Image assets allow for bundling of Docker images as assets. With this construct, the CDK can build Docker images by reading a Dockerfile present

in the local directory. The Docker image is then published to Amazon Elastic Container Registry (Amazon ECR).

To define the asset, we will use the `DockerImageAsset` construct from `aws-cdk-lib/lib/aws-ecr-assets` module as shown here:

```
from aws_cdk_lib.lib.aws_ecr_assets import DockerImageAsset

images = DockerImageAsset(self, "DockerImage",
    directory=path.join(__dirname, ".")
)
```

With this code, the CDK will look for a Dockerfile named `Dockerfile` in the directory specified by the `directory` keyword. CDK sets the Docker context to the directory mentioned in the directory keyword, and proceed to invoke the Docker build command.

Docker needs to be installed and available to be called from the directory. If Docker is not installed, or if the Docker daemon is not running, then CDK deploy will fail with an error message:

```
npx cdk deploy
CdkDockerAssetsStack: deploying...
[0%] start: Publishing
5586ea5a5b131c7b6015ca0fff4f6fce656b8772befe20d26e24bc114028d78d:current
[100%] fail: docker login --username AWS --password-stdin
https://137234234.dkr.ecr.eu-central-1.amazonaws.com exited with error code
1:

CdkDockerAssetsStack failed: Error: Failed to publish one or more assets. See
the error messages above for more information.
```

Since CDK invokes the Docker build command, the directory specified by the `directory` will be set as the *context* for the Docker build and the contents of the entire directory are uploaded to the Docker daemon. This can result in increased build times, especially if the Docker daemon is configured to run on a remote system, and is bad security hygiene in general.

Docker supports setting up an ignore list of files to be excluded using a file called `.dockerignore`. When Docker finds a `.dockerignore` file, it reads the contents of this file and modifies the Docker build context to exclude the files, directories, and patterns mentioned in the `.dockerignore` file. The CDK respects the `.dockerignore` approach of excluding unwanted files from the Docker context, and to this extent, most new projects are configured to use the `.dockerignore` approach. We can verify this by checking the value of the flag `@aws-cdk/aws-ecr-assets:dockerIgnoreSupport` of the CDK context, found in `cdk.json` of the root directory of the CDK project.

If the Dockerfile needs some build-time arguments for the build, we can pass them using the `buildArgs` property:

```
from aws_cdk.aws_ecr_assets import DockerImageAsset

images = DockerImageAsset(self, "DockerImage",
    directory=".",
    build_args={
        "BASE_IAMGE": "alpine",
        "BASE_IMAGE_TAG": "3.9"
    }
)
```

For the CDK to build to a specific target of a multi-stage Dockerfile, we can do so by providing the name of the target to the `target` property:

```
from aws_cdk_lib.lib.aws_ecr_assets import DockerImageAsset

images = DockerImageAsset(self, "DockerImage",
    directory=path.join(__dirname, "."),
    target="alpine"
)
```

Combining them all, if we want to deploy a containerized Golang app whose Dockerfile is in the `src/` directory, has a build argument of `BASE_TAG`, and build target being `prod`, the code for the construct would look like below:

```
from aws_cdk_lib.lib.aws_ecr_assets import DockerImageAsset

images = DockerImageAsset(self, "DockerImage",
    directory=path.join(__dirname, "src"),
    target="prod",
    build_args={
        "BASE_TAG": "latest"
    }
)
```

Synthesizing this using the `cdk synth` command

```
npx cdk synth
```

CDK will generate a cloud assembly by default in the `cdk.out/` directory, which will contain a copy of the assets.

Docker Image Assets are meant to be referenced within the CDK app, and as such, we cannot modify the Docker repository or tag that was attached to the built Docker image.

8.3.1. Docker Image Asset in Action: ECS Fargate

Let us look at how we can use the Docker Image Asset in CDK along with an ECS Task Definition. To start with, we define an asset

```
from aws_cdk_lib.lib.aws_ecr_assets import DockerImageAsset

docker_image = DockerImageAsset(self, "DockerImage",
    directory="src/docker",
    target="alpine",
    build_args={
        "BASE_TAG": "latest"
    }
)
```

The ECS module has a `fromDockerImageAsset` property for the `ContainerImage` method to which we pass a reference to:

```
from aws_cdk_lib.lib.aws_ecs import ContainerImage, FargateTaskDefinition

task_definition = FargateTaskDefinition(self, "FargateTaskDef",
    cpu=256,
    memory_limit_mi_b=1024
)

task_definition.add_container("HelloGo",
    image=ContainerImage.from_docker_image_asset(docker_image)
)
```

With this code, CDK will build a Docker image based on the Dockerfile present in `src/docker`, push the Docker image to Amazon ECR, create a Fargate task definition, and deploy a new Fargate task Definition with this Docker image as the container. To see this action, let's try to deploy it with `cdk deploy` command:

```
npx cdk deploy
```

Once we issue the command, CDK will synthesize the template and will ask us for confirmation of deployment. Once the deployment is complete, we can see the new task definition with the referenced Docker Image Asset.

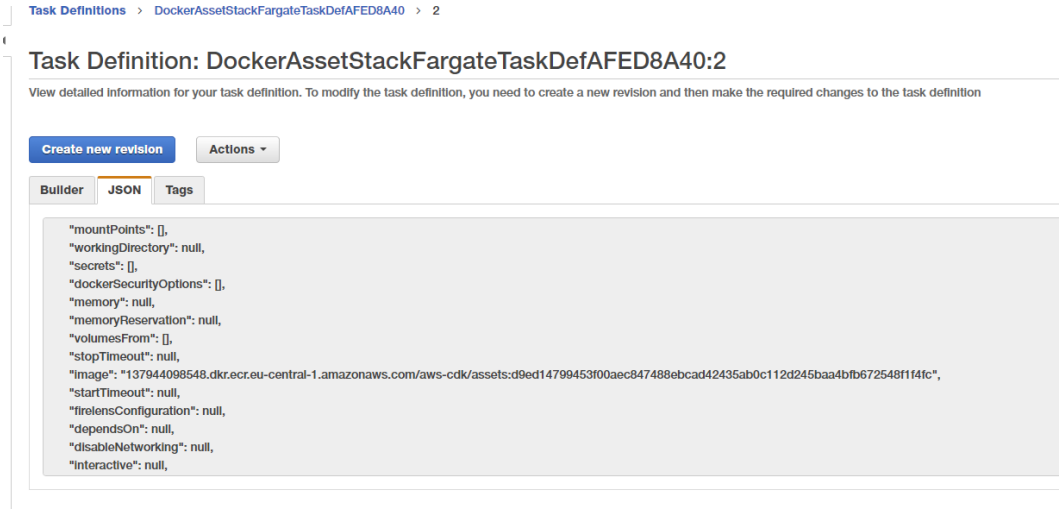

Figure 15. A look at the deployed Task Definition with the Docker Image Asset

8.4. S3 assets

Much like how we could point to a Dockerfile and let CDK build an image into an asset, we can do the same for local files/directories. This time, the difference is that the CDK will upload the files/directories to an S3 bucket. We can then refer to this as an asset.

To use S3 assets, we start with using the `aws-cdk-lib/lib/aws-s3-assets` module. With with module, we can use a single file as an asset as shown below:

```
from aws_cdk_lib.lib.aws_s3_assets import Asset
import path as path

file_asset = Asset(self, "File",
    path=path.join(__dirname, "helloworld.txt")
)
```

We can deploy this using the command `npx cdk deploy`. When we deploy

this, CDK creates a hash of this file and uploads it to the S3 assets bucket that was bootstrapped. What happens if we point the `path` keyword to a directory?

```
from aws_cdk_lib.lib.aws_s3_assets import Asset

directory_asset = Asset(self, "Directory",
    path=path.join(__dirname, "hello-world/")
)
```

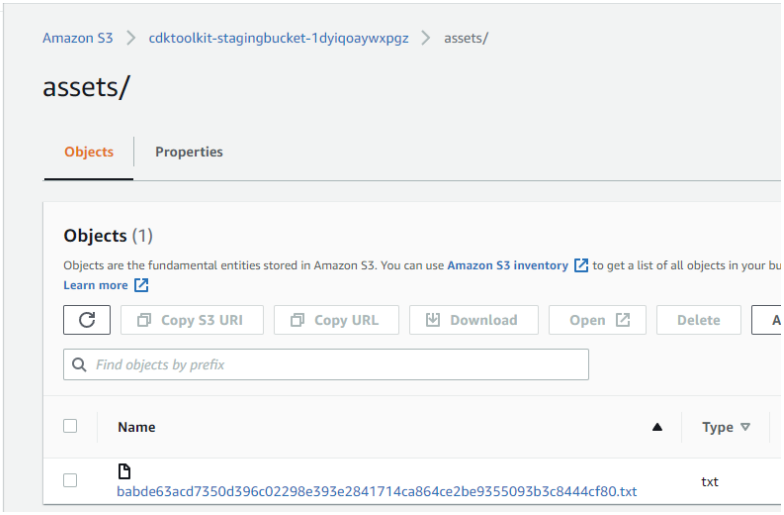

Figure 16. The file asset that was created

Deploying with `npx cdk deploy`, we can see that CDK zips the directory, creates a hash of the zipped file and uploads it to the bootstrapped staging bucket.

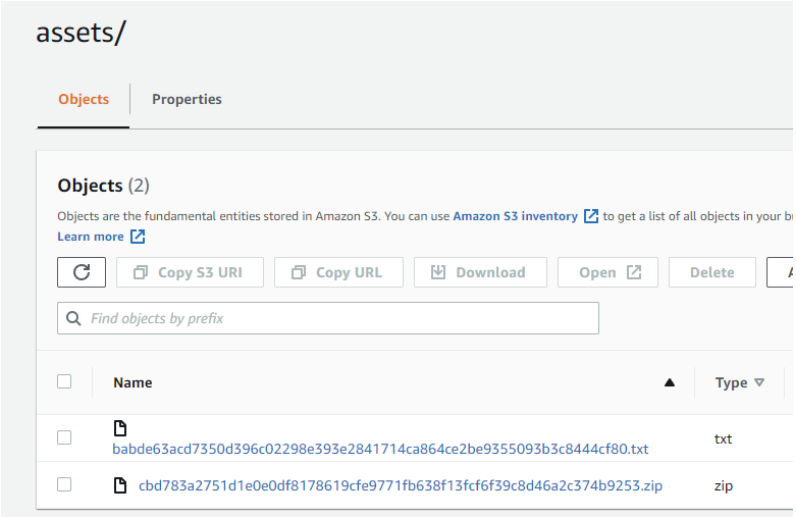

Figure 17. The zipped directory asset

Similar to the Docker Image Asset, file or directory assets created by CDK will not have user-readable file names and are not expected to be directly used/modified by the end-user. Instead, the assets are intended to be referenced from within CDK.

8.4.1. S3 Assets in Action: Deploying Lambda Functions using S3 Assets

With CDK, creating and deploying Lambda functions gets a lot simpler. S3 Assets handles deploying of the handler for the Lambda functions using the CDK as well. Let us see how we can deploy a Lambda function and the code for the Lambda function handler using CDK.

For this case, the code for the Lambda function handler will be in a file called `app.py` under the `lambda` directory. The code for the CDK constructs to build the Lambda will be in a directory called `lib`. the overall directory structure should look like below:

```
.
├── bin
│   └── s3assets.ts
├── jest.config.js
├── lambda
│   └── app.py
├── lib
│   ├── s3assets-lambda.ts
├── package.json
├── package-lock.json
├── README.md
├── test
│   └── s3assets.test.ts
└── tsconfig.json
```

Our Lambda function is simple - it will return a "Hello from Lambda!" back when we invoke it. Let us save the Lambda function handler to the `lambda\app.py` file with the contents as below:

```
def lambda_handler(event, context):
  message = 'Hello from Lambda!'
  return {
    'message': message
  }
```

For the CDK code, we use the `@aws-cdk/aws-lambda` module's `lambda.Function` method to create a Lambda function with Python 3.9 runtime. The construct will look like below:

```
from aws_cdk.aws_lambda import Handler, Runtime
import path as path

Function(self, "HelloLambda", {
    "code": Code.from_asset(path.join(__dirname, "..", "lambda")),
    "runtime": Runtime.PYTHON_3_9,
    "handler": "app.lambda_handler"
})
```

Note the `code` keyword which uses the `fromAsset()` method to point to the

directory where the Lambda handler code is present. In this case, since the Lambda function handler is in the `lambda` directory, this is set as the parameter to the `lambda.Code.fromAsset()` method. That's all that it takes for CDK to manage.

If a Docker image is used to package the code and dependencies for Lambda, that is also covered using assets. We'll have the option of using the `lambda.Function` method with the `code` parameter modified to use `lambda.Code.fromAssetImage`. The CDK also exposes a method `lambda.DockerImageFunction` which extends the `lambda.Function` method and adds some defaults needed for `lambda.Function` to handle Docker Image Asset.

To make use of `lambda.Function`, the construct code will look like below:

```
from aws_cdk_lib.lib.aws_lambda import Runtime, Handler, Function, Code
import path as path

Function(self, "HelloLambdaDockerImageFn",
    code=Code.from_asset_image(path.join(__dirname, "..", "docker")),
    runtime=Runtime.FROM_IMAGE,
    handler=Handler.FROM_IMAGE
)
```

In this case, we create a new Lambda function with the handler code being referenced from the Docker Image Asset. The parameter to the `fromAssetImage` points to the directory where the Dockerfile is present and is used to build the Docker image.

As an alternative, the `lambda.DockerImageFunction` method can be used, in which case the construct code will look like below:

```
from aws_cdk_lib.lib.aws_lambda import DockerImageFunction, DockerImageCode
import path as path

DockerImageFunction(self, "HelloLambdaDockerImageFn",
    code=DockerImageCode.from_image_asset(path.join(__dirname, "..",
"docker"))
)
```

Both are functionally identical, are currently marked as stable, and can be used.

8.4.2. S3 Assets in Action: EC2 User Data

User data is a series of commands/scripts that can be provided to EC2 instances. When an EC2 instance is provisioned and launched, these scripts are run, installing dependencies or in more complicated cases, bootstrapping a configuration management solution like Ansible or Chef.

We can configure the user data to run a few commands using the `UserData.addCommands()` method from the `@aws-cdk/aws-ec2` module. If the user data exceeds a few commands, it can be downloaded from an S3 bucket.

This is where S3 Assets shine - by having the user data scripts as part of the code, we can define an S3 Asset, let the EC2 construct reference the S3 asset, download and run the user data.

Consider the case where the user data is present in a file called `userdata.sh` in the `userdata` directory. For CDK to manage the userdata as an asset, the code for this will look as shown below:

```
from aws_cdk_lib.lib.aws_s3_assets import Asset
from aws_cdk_lib import aws_ec2 as ec2

# Assuming we've created the construct for building an EC2 instance
ec2_instance = ec2.Instance(self, "EC2", ...
)
# Create the userdata asset, pointing to `userdata/userdata.sh'
user_data_asset = Asset(self, "UserData",
    path="userdata/userdata.sh"
)

# IAM role to allow download from the asset.
user_data_asset.grant_read(ec2_instance)

# download from S3 & execute it
user_data_path = ec2_instance.user_data.add_s3_download_command(
    bucket=user_data_asset.bucket,
    bucket_key=user_data_asset.s3_object_key
)

ec2_instance.user_data.add_execute_file_command(
    file_path=user_data_path
)
```

Here, we define an `ec2Instance` construct, create a `userDataAsset` pointing to the user data in `userdata/userdata.sh` file, grant the IAM instance profile the right to download the user data from the CDK managed assets bucket, and run the user data.

When deployed, the user data will look like the below in the EC2 instance settings

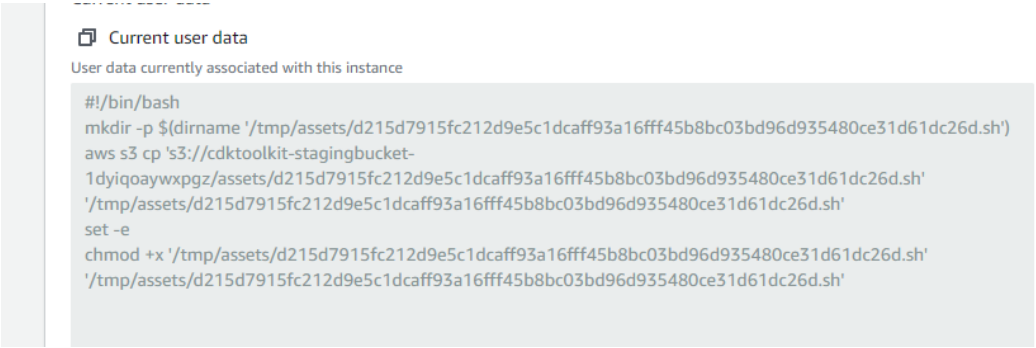

Figure 18. User data asset in EC2 instance settings

8.5. Asset Bundling

When an asset is defined, the CDK will perform operations and add a metadata entry in the CloudFormation template. This metadata record will have instructions on where to find the asset and what operations are to be done on the asset. While defining assets, some additional operations might be needed before an asset is ready for use. For example, when defining an S3 asset for a Lambda function handler, the Lambda function handler might need some dependencies to be installed and packaged.

The CDK lets us do these operations via the `bundling` option. Bundling can be done either locally, or via a Docker Image. With local bundling, the CDK will try to run the commands in the environment in which it's running. If local bundling is not an option (for example, if the environment doesn't have the necessary tools/files/dependencies), CDK can do the bundling using a Docker image.

Consider the case where a Lambda function needs to be bundled. The Lambda function handler is written in Go and needs some dependencies to be installed and the executable to be built. The CDK can handle these using local bundling as well as using Docker.

8.5.1. Docker Bundling

With Docker bundling, the CDK will invoke the bundling command in a Docker container, with an image that is provided as part of the bundling options. Consider a Lambda function handler based on Go. Without bundling, the construct for defining an asset would look like the below:

```
from aws_cdk_lib.lib.aws_lambda import Function, Runtime, Code

Function(self, "GoLambda",
    code=Code.from_asset("go-lambda"),
    runtime=Runtime.GO_1_X,
    handler="main"
)
```

The CDK will fetch the Go executable from the `go-lambda` directory, zip it, and publish it to the pre-defined CDK Assets bucket. In the absence of this executable, the deploy step would have failed. We can mention the bundling parameter which will tell the CDK how to build this executable as shown below:

```
from aws_cdk_lib.lib.aws_lambda import Function, Runtime, Code

handler = Function(self, "GoLangDockerECRImageBundle",
    handler="main",
    runtime=Runtime.GO_1_X,
    code=Code.from_asset("go-lambda",
        bundling={
            "image": Runtime.GO_1_X.bundling_image,
            "user": "root",
            "command": ["bash", "-c", ["go version", "go build -o /asset-
output/main"
            ].join(" && ")
            ]
        }
    )
)
```

With this code, CDK will create a Docker container that will build the Go

executable using the `lambda.Runtime.GO_1_X.bundlingImage`, which points to the `public.ecr.aws/sam/build-go1.x` Docker image. We can provide our Docker images, from any Docker Registry or by providing our own Dockerfile. To use a Docker image from a registry, we provide the Docker image using the `DockerImage.Registry` method:

```
from aws_cdk_lib.lib.aws_lambda import Function, Runtime, Code

handler_with_docker_registry = Function(self,
"GoLangDockerRegistryImageBundle",
    handler="main",
    runtime=Runtime.GO_1_X,
    code=Code.from_asset("go-lambda",
        bundling={
            "image": DockerImage.from_registry("golang:1.16"),
            "user": "root",
            "command": ["bash", "-c", ["go version", "go build -o /asset-
output/main"
            ].join(" && ")
            ]
        }
    )
)
```

With this code, CDK will fetch the Go Docker image from Docker Hub and use it to build the Lambda function. With a custom Dockerfile, the image can be specified by `DockerImage.fromBuild as shown below:

```
from aws_cdk_lib.lib.aws_lambda import Function, Runtime, Code

handler_with_docker_file = Function(self, "GoLangDockerFileBundle",
    handler="main",
    runtime=Runtime.GO_1_X,
    code=Code.from_asset("go-lambda",
        bundling={
            "image": DockerImage.from_build("go-lambda"),
            "user": "root",
            "command": ["./build.sh"
            ]
        }
    )
)
```

The `build.sh` file can contain the steps needed for building the executable. Regardless of which way you build it, CDK expects the files upon which it would take as input in the `/asset-input`. Similarly, CDK expects the assets to be managed by it to be available in `/asset-output`. The assets in this directory will be zipped and uploaded to S3.

8.5.2. Local Bundling

If we have the required build environment and tools already installed, we could make use of local bundling. With local bundling, the CDK will attempt to run the bundling on the local environment. Local bundling can make things faster as there's no need for the CDK to spin up a container and run the bundling steps in the container. For local bundling, we'll have to pass a `local` parameter that tells CDK what commands it must attempt to run. The local bundling needs to be implemented by a method `tryBundle` that returns `true` if local bundling was successful and `false` if not. If `tryBundle` returns a `false` then CDK will bundle using Docker. The code for local bundling with Docker bundling fallback will look like the below:

```
from child_process import exec_sync
from aws_cdk_lib import DockerImage
from aws_cdk_lib.lib.aws_lambda import Function, Runtime, Code
import path as path

handler = Function(self, "GoLangLocalBundle",
    runtime=Runtime.GO_1_X,
    handler="go-lambda",
    code=Code.from_asset("go-lambda",
        bundling={
            "image": DockerImage.from_registry("golang:1.16"),
            "user": "root",
            "command": ["bash", "-c", ["go version", "go build -o /asset-
output/main"
            ].join(" && ")
            ],
            "local": {
                def try_bundle(output_dir): try {
                                execSync('go version')
                            } catch {
                              return false
                            }
                    exec_sync(f"go build -o {path.join(outputDir, 'main')}",
                        cwd=path.join(__dirname, "../go-lambda")
                    )
                    return True
            }
        }
    )
)
```

The main difference with the local bundler is the presence of the `local` parameter, which will try to do the bundling locally.

Bundling requires a Docker image to be provided, even if local bundling is to be done.

It might be worthwhile to check if the tooling required to do local bundling is available - if this is not present, the `tryBundle()` returns false and CDK can perform the Docker bundling instead. This approach improves the performance as CDK will not have to resort to Docker bundling.

8.6. Review

Assets simplifies managing files or Docker images that may be required by our CDK application. By adding creating an asset object and pointing it to the required files, the CDK handles all aspects of building and managing it, allowing us to manage the infrastructure and the application from within a single repo.

[3] The stack synthesizer is how CDK controls how a stack's CloudFormation template gets generated

9. Testing

You've gotten a request to build some infrastructure in AWS. You know you want to use the CDK to do it. But how do you know when you've succeeded?

The question of "is my code right?" is possibly the hardest to answer in programming. Testing any system completely is often costly. Every application development team I've worked with over the years struggles to balance the costs of writing tests with the benefits of having those tests. Not all tests written are valuable and if you spend 2 days writing them and bugs are still slipping through, it won't be long before managers are asking you to skip the tests to hit deadlines.

That's not to say developers don't still test their code, but often it is 'touch testing' where an API call is executed using a testing tool (like Postman) and a response is visually inspected to see if it's "right". Or, the website is opened and buttons are clicked to see if forms can be saved and pages refreshed, showing the right data. The developer knows what 'correct' is, and they usually drive from some documentation, ticket, or story.

QA departments follow a similar process when testing the code the developers have written. They follow a script that tests the software end-to-end. This kind of testing can be swiftly executed once, but it's slow to repeat.

Automated tests that are easily repeatable are required to build complex systems. We want to drive writing good, clean, easy-to-read tests from the start. If you get to the end of the project and say "ok, now we'll go back and test", you will always see those efforts cut short instead of feature development and bug fixes.

Thankfully, the CDK makes it relatively easy to write repeatable tests for your code to ensure that it is "correct". Different tests will have different levels of complexity in writing them.

There are three types of tests you are likely to write for your CDK code:

- Unit Tests - Does it work in pieces?
- Deployment Tests - Does it deploy?
- Infrastructure Tests - Does it all work together?

Each type of test builds on the one before it, covering aspects of the code that the previous type doesn't cover well. Also, each type of test usually requires more effort to write than the type before it and has a longer time until you know if the tests passed.

Unit tests are very cheap to write and quick to run. If something is wrong, you get feedback within a few seconds.

Deployment tests will require a little more time to write, but not much. However, the feedback is not nearly as fast as you have to wait for the entire stack to deploy, which might take anywhere from 5 to 30 minutes (or more) depending on what you're building.

Infrastructure tests will probably take more time to write than unit tests and the feedback time is often as much as deployment tests (since you still have to deploy the changes).

First you will write unit tests, then advance to deployment tests and then, once you're comfortable with both, you can do infrastructure testing.

9.1. Unit Tests

Unit tests are used to test pieces, or small units, of code. A single test will only verify a small part of your entire system. You will write a lot of unit tests to cover all the individual parts of the system, verifying the entire system.

All the testing shown here is in Typescript using the Jest testing framework. https://jestjs.io/. They chose this for you when you use the CDK CLI or Projen to create a new CDK codebase. You can, of course, use any testing framework you like. You will just need to set that up yourself using the guides provided by the framework. Other languages also have their own testing frameworks. However, at the time this book was published, testing in other languages is not set up automatically, and they require you to do that yourself.

For example, a unit test would validate that an S3 bucket is created properly. All these tests are written in Typescript because they use the Jest testing framework to illustrate the point. Later sections will show similar tests with Python.

```
import { Template } from '@aws-cdk/assertions-alpha';
import { App } from 'aws-cdk-lib';
import { MyTestStack } from '../src/my-test-stack.ts';

describe('S3 Bucket', () => {
  const app = new App();
  const stack = new MyTestStack(app, 'MyTestStack');
  const assert = Template.fromStack(stack);

  test('Has correct properties', () => {
    assert.hasResourceProperties('AWS::S3::Bucket', {
      BucketEncryption: {
        ServerSideEncryptionConfiguration: [
          {
            ServerSideEncryptionByDefault: {
              SSEAlgorithm: 'AES256',
            },
          },
        ],
      },
      PublicAccessBlockConfiguration: {
        BlockPublicAcls: true,
        BlockPublicPolicy: true,
        IgnorePublicAcls: true,
        RestrictPublicBuckets: true,
      },
      VersioningConfiguration: {
        Status: 'Enabled',
      },
    });
  });
});
```

But that's it, nothing more.

Let's inspect what's happening. First, we have all of our imports:

```
import { Template } from '@aws-cdk/assertions-alpha';
import { App } from 'aws-cdk-lib';
import { MyTestStack } from '../src/my-test-stack.ts';
```

Imported into this file is the `Template` class from the @aws-cdk/assertions library [4], the App class from `@aws-cdk/core` and our `MyTestStack` stack

class being tested.

Next, we begin creating a group of tests, using the `describe` function:

```
describe('S3 Bucket', () => {
});
```

Group tests that verify similar things, like all your S3 Bucket tests. Tests that verify other constructs would go together in another group, like 'API Gateway'. How they're alike is up to you. How you name them is also up to you, as long as it makes it clear what is being tested.

If there were multiple S3 buckets being created by the stack, this 'S3 Bucket' name would probably have to change. Naming things is one of the three hard things in computer programming. I'm not here to tell you how to do it, just that you should think carefully about it since this name is going to show up in reports and error logs, so it should point any engineer to exactly what they need to know to fix the problem. Talk to your team. Bring it up in code reviews. Your team should agree on the conventions and patterns of naming things.

Next, a new App is created, a new instance of the MyTestStack, and a new Template class from the stack. These three objects are saved as variables (app, stack, assert), so they can be used later:

```
const app = new App();
const stack = new MyTestStack(app, 'MyTestStack');
const assert = Template.fromStack(stack);
```

The `assert` variable will be especially helpful later.

A test is created next, using the 'test' function. It's given the name of the test, and a function to execute that has all the logic of the test.

```
  test('has correct properties', () => {
    assert.hasResourceProperties('AWS::S3::Bucket', {
      BucketEncryption: {
        ServerSideEncryptionConfiguration: [
          {
            ServerSideEncryptionByDefault: {
              SSEAlgorithm: 'AES256',
            },
          },
        ],
      },
      PublicAccessBlockConfiguration: {
        BlockPublicAcls: true,
        BlockPublicPolicy: true,
        IgnorePublicAcls: true,
        RestrictPublicBuckets: true,
      },
      VersioningConfiguration: {
        Status: 'Enabled',
      },
    });
  })
```

The test is named 'has correct parameters'. The tests are then grouped together nicely when running your tests:

```
$ yarn test
PASS  test/bucket.test.ts (6.695 s)
  S3 Bucket
    ✓ has correct properties (115 ms)
```

In this case, the `yarn test` command is used to run all tests. This is what Projen will setup in the project. If you use `cdk init` to create your project, then is `npm run test`. If you're using another language and set up the testing yourself, you'll know the command to run the tests.

In this test, the `.hasResourceProperties` function is used to see if an S3 bucket exists and has the properties set. All the Properties will be compared to what it generated and, if they match, the test passes.

These are AWS CloudFormation resource properties, the same thing you'd

see written in a CloudFormation file. Even the casing of "BucketEncryption" vs the typical JS/TS "bucketEncryption".

This is not an 'exact' match. Whatever you supply as the expected result is checked to existing on the generated resource. However, if there are more properties on the resource than what you supplied, the test will still pass. This means you can be selective about how much you pass in to the test. If you don't care about certain properties, just ignore them.

However, this only checks the properties of a resource and not the whole resource. If you want to test the entire resource definition, then a `hasResourceDefinition` function can be used. This is great when you want to verify other things than the resource-specific properties, like `UpdateReplacePolicy` and `DeletionPolicy`; very useful things to test with S3 buckets (or anything that contains state).

```
test('is retained after delete', () => {
  assert.hasResourceDefinition('AWS::S3::Bucket', {
    Properties: {
      BucketEncryption: {
        ServerSideEncryptionConfiguration: [
          {
            ServerSideEncryptionByDefault: {
              SSEAlgorithm: 'AES256',
            },
          },
        ],
      },
      PublicAccessBlockConfiguration: {
        BlockPublicAcls: true,
        BlockPublicPolicy: true,
        IgnorePublicAcls: true,
        RestrictPublicBuckets: true,
      },
      VersioningConfiguration: {
        Status: 'Enabled',
      },
    },
    UpdateReplacePolicy: 'Retain',
    DeletionPolicy: 'Retain',
  });
});
```

While you may write your tests before you write your code (called Test Driven Development) you don't have to. You can write your CDK code first. The fact is most people probably never write tests because the chances of getting this code 'wrong' are pretty low:

```
  new Bucket(this, 'Bucket', {
    blockPublicAccess: {
      blockPublicAcls: true,
      ignorePublicAcls: true,
      restrictPublicBuckets: true,
      blockPublicPolicy: true,
    },
    encryption: BucketEncryption.S3_MANAGED,
    versioned: true,
  });
```

However, if a test is needed after the code's been written, it's pretty easy to add a test. Create a new test like before but replace all the properties with something that can't be right, causing a failure in all situations:

```
  test('is retained after delete', () => {
    assert.hasResourceDefinition('AWS::S3::Bucket', {
      Not: "Valid",
    });
  });
```

This test will always fail, because AWS::S3::Bucket never has a 'Not' property. The test will fail with a message like:

```
Error: None of 1 resources matches resource 'AWS::S3::Bucket' with {
  "$objectLike": {
    "Not": "Valid"
  }
}.
- Field Not missing in:
    {
      "Type": "AWS::S3::Bucket",
      "Properties": {
        "PublicAccessBlockConfiguration": {
          "BlockPublicAcls": true,
          "BlockPublicPolicy": true,
          "IgnorePublicAcls": true,
          "RestrictPublicBuckets": true
        }
      },
      "UpdateReplacePolicy": "Retain",
      "DeletionPolicy": "Retain"
    }
```

Now you can just copy and paste the block of code from the error into your test!

```
  test('is retained after delete', () => {
    assert.hasResourceDefinition('AWS::S3::Bucket', {
      "Properties": {
        "PublicAccessBlockConfiguration": {
          "BlockPublicAcls": true,
          "BlockPublicPolicy": true,
          "IgnorePublicAcls": true,
          "RestrictPublicBuckets": true
        },
      },
      "UpdateReplacePolicy": "Retain",
      "DeletionPolicy": "Retain"
    });
  })
```

Now you will have JSON in your code and your IDE (and linters) may throw errors. To remove any errors you will need to convert this to standard Typescript/Javascript code.

However, tools like 'eslint' [5] can automatically fix that for you.

```
eslint --fix test/**/*.ts
```

This test now checks to make sure the produced CloudFormation resources match a known definition. It's like taking a picture of your house yesterday, taking one today, and comparing the two images to make sure your roof didn't blow away. They're referred to as snapshot tests. Retrofitting these simple snapshot tests onto existing code should be pretty easy in most cases.

In fact, if you're using Jest there is a very easy 'catch-all' snapshot test you can write:

```
import { Template } from '@aws-cdk/assertions';
import { App } from '@aws-cdk/core';
import { MyStack } from '../src/main';

test('Snapshot', () => {
  const app = new App();
  const stack = new MyStack(app, 'TestStack');
  expect(Template.fromStack(stack)).toMatchSnapshot();
});
```

And in Python:

```
def test_matches_snapshot(snapshot):
    # Set up the app and resources in the other stack.
    app = cdk.App()
    stack = MyStack(app, "TestStack")

    # Prepare the stack for assertions.
    template = Template.from_stack(stack)

    assert template.to_json() == snapshot
```

Using the the template's `toJson()` means the contents of the entire CloudFormations 'Resources' will be checked. This kind of test is very thorough and easy to write. The first time the test is run, the results are taken as the snapshot and subsequent tests will compare the values against the snapshot, failing the test if they have changed in any way. If you change your code later and need to update the snapshot, a special 'update' command can be run:

```
# jest cli directlry
jest --updateSnapshot

# projen style
yarn test:update
```

Let's take a quick look at a slightly more complex example. This builds on top of the previous example.

Now the stack has an optional property called 'createBucket'. If a 'true' value is given, the bucket should exist. If a 'false' value is given, the bucket should not exist.

The previous tests will be changed a little:

```
describe('S3 Bucket', () => {
  function getTestAssets(props: MyTestStackProps) {
    const app = new App();
    const stack = new MyTestStack(app, 'MyTestStack', props);
    const assert = Template.fromStack(stack);
    return { assert, stack, app };
  }
);
```

At the top of our group of tests, a new `getTestAssets` function, that given a set of properties, will create a new App, MyTestStack, and assertion Template, then return them to the caller. This will allow us to create these objects easily in our tests:

```
  test('has correct properties', () => {
    const { assert } = getTestAssets({ createBucket: true });
    assert.hasResourceProperties('AWS::S3::Bucket', {
      PublicAccessBlockConfiguration: {
        BlockPublicAcls: true,
        BlockPublicPolicy: true,
        IgnorePublicAcls: true,
        RestrictPublicBuckets: true,
      },
      VersioningConfiguration: {
        Status: 'Enabled',
      },
    });
  });
```

And now we can write the new test for the `createBucket: false` scenario:

```
test('no bucket is created', () => {
  const { assert } = getTestAssets({ createBucket: false });
  assert.resourceCountIs('AWS::S3::Bucket', 0);
});
```

Sometimes you want to make sure a resource isn't included because it will be provided by the client of the construct instead of generated. For example, the AWS Solutions Constructs `LambdaToDynamoDB` construct will create a Lambda function unless you provide your own.

By writing a lot of discrete unit tests, you can verify all the parts of your entire system. You'd write tests for each of the resources you want to create and for any scenarios where their creation varies, like with the `createBucket` property. Besides verifying the bucket, unit tests would be written to verify that all the resources in the stack are created as you intended.

Jest will make writing unit tests easy and comprehensive with the simple `expect(Template.fromStack(stack)).toMatchSnapshot();`

If you write tests that check specific resources using the `.hasResourceProperties` and `.hasResourceDefinition` you are testing smaller portions of your code at a time, making expectations clearer. These are referred to in the CDK docs as 'fine-grained assertions', as they define detailed expectations of the code.

If you write snapshot tests for the entire stack you may save a lot of time and make sure you cover the entire stack, but will also lose the intention of your code and would have to review snapshots to understand what you expect to have created. They can be good to retrofitting tests into an existing infrastructure.

9.1.1. Matchers

Besides taking an object, the Has Resource Properties function can take a Matcher which provides other methods to compare your desired definition to what was synthesized.

Object Like

The Object Like matcher is similar to just providing an object, checking that the object provided is a subset of the object generated.

```
template.has_resource_properties("AWS::S3::Bucket",
    Match.object_like(
        BucketEncryption={
            "ServerSideEncryptionConfiguration": [{
                "ServerSideEncryptionByDefault": {
                    "SSEAlgorithm": "AES256"
                }
            }
            ]
        }
    ))
```

In this case, only the `BucketEncryption` property is compared. The other properties generated are not checked. In other words, having more defined on your resource than just the properties provided doesn't fail the test.

Object Equals

The Object Equals matcher is stricter than the Object Like.

```
template.has_resource_properties("AWS::S3::Bucket",
    Match.object_equals(
        BucketEncryption={
            "ServerSideEncryptionConfiguration": [{
                "ServerSideEncryptionByDefault": {
                    "SSEAlgorithm": "AES256"
                }
            }
            ]
        }
    ))
```

This test will fail if there are more properties on the AWS::S3::Bucket than just the BucketEncryption property.

Any Value

The Any Value can be used when you don't care about the specific value but want to ensure that it's still provided.

```
assert.has_resource_properties("AWS::Lambda::Function",
    Match.object_like(
        Code=Match.any_value(),
        Role={
            "Fn::GetAtt": ["FixedResponseHandlerServiceRole92BFE295", "Arn"
            ]
        },
        Environment={
            "Variables": {
                "AWS_NODEJS_CONNECTION_REUSE_ENABLED": "1"
            }
        },
        Handler="index.handler",
        Runtime="nodejs14.x"
    ))
```

In this test the values for the Lambda function are all checked and the Code property is matched against any value. If it hadn't shown up, the test would fail.

Absent

If you want to ensure a property is not set, you can use the Absent matcher.

```
assert.has_resource_properties("AWS::Lambda::Function",
    Match.object_like(
        Code=Match.any_value(),
        Role={
            "Fn::GetAtt": ["FixedResponseHandlerServiceRole92BFE295", "Arn"
            ]
        },
        Environment={
            "Variables": {
                "AWS_NODEJS_CONNECTION_REUSE_ENABLED": "1",
                "PGPASSWORD": Match.absent()
            }
        },
        Handler="index.handler",
        Runtime="nodejs14.x"
    ))
```

In this test we're ensuring that the PGPASSWORD environment variable is not set.

Serialized Json

Finally, there is the Serialized Json matcher. This allows you to check properties on resources that have a JSON format.

The AWS::StepFunctions::StateMachine has the 'DefinitionString' property that needs to be checked against a JSON string value.

```
assert.has_resource_properties("AWS::StepFunctions::StateMachine",
    DefinitionString=Match.serialized_json(
        StartAt="The Beginning",
        States={
            "The Beginning": {
                "Type": "Pass",
                "End": True,
                "Next": Match.absent()
            }
        }
    )
)
```

9.1.2. Captures

Captures allow you to write your own assertions against specific values, in case the matchers that exist aren't enough.

In this example, we'll capture the Role that is assigned to the function and then do our own comparison to see if it's what we expect.

```
role_capture = Capture()
assert.has_resource_properties("AWS::Lambda::Function",
    Match.object_like(
        Code=Match.any_value(),
        Role=role_capture,
        Environment={
            "Variables": {
                "AWS_NODEJS_CONNECTION_REUSE_ENABLED": "1"
            }
        },
        Handler="index.handler",
        Runtime="nodejs14.x"
    ))
```

And then you can run any comparison you want against the results of the '.as' methods.

```
role_capture.as_string()
role_capture.as_object()
```

9.1.3. Refactoring

After tests are run, you may find you'd like to change your code. Perhaps you'd like to take some common functionality and create a new construct out of it, so you can reuse it later.

While you can move your code around all you want, doing so will change the LogicalId of the CloudFormation resource. If the LogicalId changes, then CloudFormation will see this as a new resource, create it, and delete the old one on any stacks you may have already deployed.

Let's look at an example:

```
from aws_cdk.aws_s3 import Bucket
from aws_cdk.core import Construct, Stack, StackProps

class OriginalStack(Stack):
    def __init__(self, scope, id, props):
        super().__init__(scope, id, props)

        # The first time you run the unit test, have this bucket created
here:
        Bucket(self, "Bucket")
```

Just a very simple stack with a single bucket. If we decide to move that bucket into a new construct:

```
from aws_cdk.aws_s3 import Bucket
from aws_cdk.core import Construct

class SomeConstruct(Construct):
    def __init__(self, scope, id):
        super().__init__(scope, id)
        # this is the code, copy and pasted from its original usage directly
in the stack above
        Bucket(self, "Bucket")
```

Now if we synthesize the original stack, the LogicalId of the bucket will change from `Bucket83908E77` to `ThingyBucket292460C0`. An S3 Bucket isn't something you often want to just recreate, as it contains state and throwing state away can be costly.

In many cases, that might be totally fine, especially when you're working with stateless resources. But, if you've got live resources in an environment and don't want to replace them, this could detour you from wanting to make those changes.

Thankfully, the CDK allows us to override the LogicalId of any construct. We can use Aspects to apply these overrides easily.

```
from aws_cdk.core import CfnElement, CfnResource, IAspect, IConstruct, Stack

class LogicalIdMapper(IAspect):
    def __init__(self, *):
        pass

    def visit(self, node):
        current_logical_id = Stack.of(node).get_logical_id(node)
        # is there a map for this logicalId?
        if not not self.id_map[current_logical_id]:
            # if we're on a L1 resource, try to do the override directly
            if (node).override_logical_id: return (node).
override_logical_id(self.id_map[current_logical_id])
```

Then it's just a matter of applying the Aspect to the stack and handing it the map of new Ids to old Ids:

```
from aws_cdk.core import Aspects, Construct, Stack, StackProps
from ..LogicalIdMapper import IdMap, LogicalIdMapper
from ..SomeConstruct import SomeConstruct

class RefactoredStack(Stack):
    def __init__(self, scope, id, *, idMap=None):
        super().__init__(scope, id, idMap=idMap)

        SomeConstruct(self, "Thingy")

        # if an idMap was provided, add the aspect.
        if id_map:
            Aspects.of(self).add(LogicalIdMapper(id_map))
```

The stack implementation changed from OriginalStack to RefactoredStack, but that's only so you can follow along with the examples.

Finally, a test will verify everything stayed the same.

```
from aws_cdk.assertions import Template
from aws_cdk.core import App
from ...src.OriginalStack import OriginalStack

test("Snapshot", () => {
      /**
       * This is a simple snapshot test using Jest. If you run this test
before your refactoring, it'll take a snapshot
       * of the LogicalId.
       * Refactor your code and then re-run this test. It will fail as the
LogicalIds have changed.
       *
       * Then, uncomment out the idMap property and verify the correct values
are provided. The failed test will show
       * you the new LogicalId (what it received) and the original LogicalId
(the snapshot).
       * In the example code, the new LogicalId of 'ThingyBucket7D8CBF87'
will be rewritten to the value it
       * was before the refactor, 'Bucket83908E77', satisfying the test and
ensuring your resource won't be recreated.
       */

      const app = new App();
      const stack = new OriginalStack(app, 'test', {});
      // const stack = new RefactoredStack(app, 'test', { idMap: {
ThingyBucket7D8CBF87: 'Bucket83908E77' }  });
      expect(Template.fromStack(stack)).toMatchSnapshot();
    })
```

This is one of the primary reasons unit testing is so important to the development process. While other checks could be put into place, few will be as easy as writing a three-line snapshot test for your existing stacks.

There is another trick, although it's not documented and rather limited. There is a magical value for a constructs Id, 'Default'. If you use this as your Id for the new construct you have created, any constructs it creates will not have the parent scope affect the Id, meaning the values won't change. However, you can only do this once as Ids still need to be unique within a scope.

You can also move constructs from one stack to another, although the process will involve more than just shifting some code around. You'll also

need to work with CloudFormation directly to import resources into the new stack.

CloudFormation has some limitations on what can be imported. Review the current abilities before trying to change your code.

Let's look how we'd move a Bucket from a stack called AStack to a new stack called BStack.

1. Update AStack setting the Bucket's deletion policy to Retain. This can be done directly in the CDK.
2. Update AStack, removing the Bucket. If you have other resources that reference this Bucket already, replace those with a `.from*` lookup functions.
3. Create BStack with **only** the Bucket definition in it. You will need to create the BStack in CloudFormation directly and skip using the CDK cli to create the stack. The CDK cli doesn't currently support handling imported resources so fall back to Console, API, or AWS CLI to do this. During the import you'll be asked to provide the ID of the existing resource, enter the Bucket name in this case.

You may edit the synthesized template to remove any resources you can't import, like the CDK Metadata resource (although options are available on the App constructor to remove these). CloudFormation won't let you create a stack with a mix of imported and non-imported resources.

After the BStack is created with just the imported Bucket resource, you can apply future updates using the CDK cli (`cdk deploy`), or any other mechanism.

9.1.4. Mocking Assets

CDK Assets are one of the best features of the CDK. They are bundles of files that are used with various AWS resources, like Lambda functions or ECR Docker images or many other constructs. If you write a Lambda function, the Code object will take your source code files and package them in a zip file. Then, the cli will upload those zip files to an S3 bucket when you run `cdk deploy`. The CloudFormation templates that are synthesized point to those objects in the bucket.

Similar processes exist for other constructs like the `DockerImageAsset` construct and the `BucketDeployment` construct.

When you write unit tests for these constructs, you could just let the system generate those assets. However, this means the tests will take longer to run, which can be annoying. You could also run into some technical issues when running in CICD pipelines as some constructs use Docker and running Docker in some automated build pipelines is non-trivial.

You can mock your asset creation. Mocking replaces the normal functionality with special testing functionality. If you want to mock asset creation during your unit tests:

```
describe('mocking Code', () => {
  let fromAssetMock: jest.SpyInstance;

  beforeAll(() => {
    // mock the Code calls so tests run quicker
    fromAssetMock = jest.spyOn(Code, 'fromAsset').mockReturnValue({
      isInline: false,
      bind: (): CodeConfig => {
        return {
          s3Location: {
            bucketName: 'my-bucket',
            objectKey: 'my-key',
          },
        };
      },
      bindToResource: () => {
        return;
      },
    } as any);
  });
  afterAll(() => {
    // restore the Code from mock
    fromAssetMock?.mockRestore();
  });

  test('Check snapshot', () => {
    const app = new App();
    const stack = new PubSubStack(app, 'test');

    expect(app.synth()
              .getStackArtifact(stack.artifactId)
              .template)
          .toMatchSnapshot();
  });
});
```

Special Thanks

Special thanks to Ryan Dsouza for sharing this on the cdk.dev Slack channel.

Now any tests in this group will not make actual calls to the `fromAsset` function in the Code class, so no files will be generated and tests will run quicker. If you are using other functions on the Code class, like `fromInline`, you'd change the code slightly:

```
  beforeAll(() => {
    // mock the Code calls so tests run quicker
    fromAssetMock = jest.spyOn(Code, 'fromInline')
                        .mockReturnValue({} as any);
  });
```

9.2. Deployment Tests

However, unit testing alone doesn't guarantee that your code is 'correct', even if you wrote a lot of tests. You could write the test wrong, or could miss a vital component of how a construct (and underling CloudFormation resource) must be set up. There are many reasons your unit tests might all pass, but the creation of the CloudFormation Stack in an AWS environment would still fail.

For example, let's say you create an IAM Role that uses an AWS-managed Policy `arn:aws:iam::aws:policy/AWSControlTowerRolePolicy`. This is likely a hardcoded string somewhere in your CDK code:

```
new Group(this, 'ReadOnlyGroup', {
    managedPolicies: [
        ManagedPolicy.fromManagedPolicyArn(
            this,
            'ReadOnlyManagedPolicy',
            'arn:aws:iam::aws:policy/AWSControlTowerRolePolicy'
        ),
    ],
  });
```

A snapshot test isn't great in this case because if you wrote the ARN wrong in one place, you'll probably write it wrong in the other. This type of problem is also hard to catch in code reviews, so pull requests won't save you. In fact, `arn:aws:iam::aws:policy/AWSControlTowerRolePolicy` is wrong, and you may not have noticed (it's 'AWSControlTowerServiceRolePolicy'). Your test would pass, but the deployment would fail because the ARN is invalid.

Deployment tests aim to catch these types of errors by synthesizing your CDK application and deploying the resulting CloudFormation template to an AWS environment. Just having a new stack created can expose many types of errors that unit testing isn't likely to catch.

While this helps you exercise your code more and find errors, it comes with some costs:

- You need access to deploy all resources in an AWS environment. Managing the keys and getting permissions right to do this isn't always easy.
- It takes time. Many resources in AWS can take a long time to create, like an RDS instance, which often takes almost 30 minutes. While this may seem like a minor inconvenience, it can have knock-on effects on CICD systems that could be undesirable (like delaying the deploy to a production environment in a hot-fix scenario).

Deployment testing itself is pretty easy. Just synthesize your CDK application and then deploy it to a sandbox AWS environment. Something that is not shared with any active environment like dev, qa, or production. If the stack completes without error, your deployment test has passed! Sounds simple, and it is. However, you may find that you're unable or unwilling to deploy your entire application to an environment as a deployment test. In that case, you'll want to create a new CDK application and use the `app` option:

```
npx cdk deploy --app 'ts-node test/integ.test.ts'
```

This allows you to create a deployment test tailored to your needs. You can remove components you may deem unworthy of testing, because they're unnecessary to the part of your system you're trying to verify or because they're slow to create.

This also gives you an avenue to test upgrades to your CloudFormation

stacks. Creating a stack is a solid test, but upgrading the last version is an even better one.

Modify your pipeline so that the stack you create as a deployment test has a consistent name, like 'deployment-test-apil'. Do not delete the stack after it's The first time the pipeline runs, it will create the stack. The second time, and until you delete the stack manually, it will update the existing stack.

You could even do both in your pipeline: * Create a new stack with a random name that is deleted. * Create or update an existing stack with a consistent name that is not deleted.

Doing both would provide a large amount of coverage for any potential issues in your templates. But this could be cost prohibitive depending on the resources, since you'd have one additional set running at all times.

9.3. Infrastructure Tests

Finally, there are infrastructure tests. Infrastructure tests aim to verify parts of your system that unit tests and infrastructure tests are unlikely to verify. These are often called "end-to-end" tests, or e2e tests. End-to-end tests are common in application development. In fact, you may already have some of these as your build and deployment process.

As Serverless systems are gaining adoption, the code typically written by developers is now handled by infrastructure components and their configuration. While the developers may not have to write tests for code they no longer own, someone else (you) will need to.

You may wonder: "If I wrote all my unit tests and wrote deployment tests, what left is there to test?"

Infrastructure tests aim to verify that all your various resources are wired

together correctly, and the system is working as a whole because some aspects are hard to verify through unit tests or deployment tests alone. For example, an infrastructure tests would verify that a Lambda function that reads and writes to a DynamoDB table has the correct permissions to do that. It's easy to forget to write the unit test that checks for this permission and a deployment test would not catch the missing IAM permission as it wouldn't execute the Lambda function.

Infrastructure tests quickly encroach on general application testing. Whole books have been written on application testing and this book can't possibly do it full justice. We want to focus on how to validate your CDK code, not your application code.

The following use-cases are meant just as examples of how you could test a system of varying complexities.

Don't take these as "the only way", just "one way". Hopefully, these inspire you on how you can verify your own infrastructure.

In each of these cases, a Custom Resource will be added to a stack. This Custom Resource will execute the test. If the test fails, then the Custom Resource fails and the stack changes will roll back [6].

By incorporating the test directly into the stack, a stack creation or update will fail if the test fails, ensuring your infrastructure is in a valid state without manual intervention (or writing more automation to redeploy a previous version).

There is an entire chapter dedicated to Custom Resources if you'd like to read more.

9.3.1. Use Case 1 - Static Website

We'll begin with one of the simplest tests that could be run, a static website. The website is hosted in S3 and a successful test is a request of the root of the site and getting a successful 200 response. The CDK code looks like this:

```
class WebsiteStack(Stack):
    def __init__(self, scope, id, props=None):
        super().__init__(scope, id, props)

        bucket = Bucket(self, "WebsiteBucket",
            versioned=True,
            public_read_access=True,
            encryption=BucketEncryption.S3_MANAGED,
            website_error_document="index.html",
            website_index_document="index.html",
            removal_policy=RemovalPolicy.DESTROY
        )

        deployer = BucketDeployment(self, "BucketDeployment",
            destination_bucket=bucket,
            sources=[Source.asset(path.join(__dirname, "..", "..", "src"))]
        )

        website_tests_handler = NodejsFunction(self, "WebsiteTestsHandler",
            entry=path.join(__dirname, "handlers", "website-test-
handler.ts"),
            bundling={}
        )

        website_tests = CustomResource(self, "WebsiteTests",
            resource_type="Custom::WebsiteTests",
            properties={
                "url": bucket.bucket_website_url,
                "timestamp": Date()
            },
            service_token=website_tests_handler.function_arn
        )

        website_tests.node.add_dependency(deployer)
```

This creates a bucket, uses the `BucketDeployment` construct from the aws-s3-deployment module and a Lambda function backing a Custom Resource to test the deployment worked properly.

Here we are using a direct Lambda integration with the CustomResource construct (`serviceToken: websiteTestsHandler.functionArn`). In this situation we could also use the Provider construct and saved ourselves some trouble. However, with this example, we don't want to complicate something that is covered in great detail over in the Custom Resources chapter.

A timestamp with the current time is added as a property on the Custom Resource. This will force an update of the Custom Resource on every deploy, causing the tests to run on every deploy. Without this, the test would only run when the Custom Resource was created or when the bucketWebsiteUrl was changed, and that's not enough. We probably want this test to run on every deployment, so a timestamp will force that. The value itself is ignored, but it's enough to cause a resource change in CloudFormation and an update of the Custom Resource.

The Custom Resource's Lambda function handler, `website-test-handler.ts`, looks like:

```
const response = require('@matthewbonig/cfn-response');
import * as axios from 'axios';

export const handler = async (event: any, context: any) => {
  console.log('event:', JSON.stringify(event, null, 2));
  const { url } = event.ResourceProperties;
  if (event.RequestType === "Delete") {
    // let's do nothing.
    return await response.send(event, context, response.SUCCESS, {});
  }

  const responseData = {};
  try {
    await runTests(url);
    console.log("Tests passed!");
    await response.send(event, context, response.SUCCESS, responseData);
  } catch (err) {
    console.error('Tests failed!', err);
    await response.send(event, context, response.FAILED, { err });
  }
}

async function runTests(url: string) {
  return await axios.get(url)
}
```

The specifics of the test are executed in the `runTests` function. It uses the `axios` 3rd-party module to run a GET request against the url for the site. Any failure, a 4xx/5xx status code, will cause the function to fail. That will return a FAILED status back to the Custom Resource and cause the stack changes to rollback. If the site responds properly with a 2xx/3xx response, the test has passed and a SUCCESS status is sent back to the Custom Resource, allowing the stack to continue updating.

This is a simple test, but illustrates the basic mechanic we'll be using in the next two use cases: a Custom Resource (and Lambda function) the executes tests during deployment.

9.3.2. Use Case 2 - Lambda-backed API Gateway

Now we're going to step it up a notch. We're going to look at an API that reads and writes records to a database.

Remember, the goal of infrastructure tests is to make sure everything is wired together correctly. Let's test a simple API created with API Gateway, a Lambda function, and DynamoDB:

```
const table = new Table(this, 'SomeTable', {
  partitionKey: {
    name: 'PK',
    type: AttributeType.STRING
  }
});
const backend = new lambda.Function({
  environment: {
    TABLE_NAME: table.tableName
  }
});
new apigateway.LambdaRestApi(this, 'SomeApi', {
  handler: backend,
});
```

If this was the entire CDK code, then when the Lambda function handler tried to access the DynamoDB table, it would fail, because we haven't granted access to the Lambda function. We're missing this line, or one like it:

```
table.grantReadWrite(backend);
```

This is a common mistake and something that you could easily miss during unit testing and deployment testing wouldn't produce an error. This is the type of problem we hope to catch with infrastructure testing, something that wouldn't occur until runtime. Waiting until an error occurs at runtime is usually too slow, because then it's affecting your users and we want to catch problems before the users do.

In the previous example, a simple GET request against the website was enough to test the system. This time, the test is a bit more complex. We need to make a call to the API that tries to write a record in the database. We'd want to make a call that reads that same record from the database. This would test that the permissions were applied correctly to the Lambda function. Finally, we'd want to clean up that record from the database, as we don't want our database polluted with test records. They could cause issues later.

There are two approaches we could take: 1. Make 3 API calls, one to write, one to read, one to delete the test record. 2. Make one call that does all three.

There is a lot to consider when picking one over the other, so let's review:

If we use approach #1 then we don't have to change existing API code. The test will make three calls to the API and do all the work.

But, the test then needs to be very knowledgeable about the API. It needs to know the schema, or shape, of the data to send. It needs to know what is valid data, as sending invalid data would cause an error and the test to fail. Having to know all of this can make writing the test more difficult and more error-prone. It's also something that probably requires the help of another team.

The API must support the read, write, and delete operations, and maybe your API does not. For example, if you have an API for managing users in your system, you may not allow deletion, just deactivation. That means you'd end up with a lot of test users in your database, polluting it with data not actually important to the system.

This approach also creates a less generalized solution, which may be ok depending on the size of your project.

If we use approach #2 then someone has to change the existing code so the API has a special 'test' function.

Yet, by doing this, we make the test we managed considerably easier to write and more generalized, and therefore more reusable. We don't have to care much about the schema or what is valid data, we just ask the API "please test yourself". APIs often have a 'health' check endpoint that components like load balancers, reverse proxies, and infrastructure monitoring tools can use to know that the API is up and running.

A similar 'test' check endpoint is added that would run the testing code. The primary advantage to this approach is that the infrastructure test is very simple, just hitting an endpoint while the responsibility of knowing exactly how to test the system is the responsibility of the API.

The major drawback to this approach is you have to change your API. However, certain techniques can reduce risk.

Let's look at how each approach would be implemented.

The first approach would be implemented very similar to the static website test, just with some additional calls. The previous 'runTests' function now looks something like this:

```
async function runTests(url: string) {
  let userWasWritten = false;
  try {
    await axios.post(url, A_TEST_USER);
    console.log('A test user was written: ', A_TEST_USER);
    userWasWritten = true;
    const testUser = axios.get(url);
    if (!usersMatch(testUser, A_TEST_USER)) {
        throw new Error('The TEST user failed');
    }
    console.log('The written user and read user match, test passed!');
  } finally {
    if (userWasWritten) {
      // let's clean up the test record
      await axios.delete(url);
    }
  }
}
```

First, the test has to know what a valid user is, and store that in the `A_TEST_USER` variable. In the above code, I have not shown how this A_TEST_USER would be created, for brevity.

Next, it needs to know how to compare the data that was sent to the database and the data that was retrieved. This may not just be simply comparing the fields, there might be more complex logic as APIs don't always return the exact record you sent. Sometimes they add additional fields like a `createdBy` or `modifiedOn` fields that your logic would have to account for (and likely ignore) in the test. Finally, we have to make sure that we delete the test data from the system if it had been written successfully. Good API documentation will make this fairly easy.

This code is going to exist in our CDK code, whereas the API code may be pretty far away, even in another repository. When related code is physically far apart, the chances of it being out of sync, and wrong, go way up. Developers have to know changes in one place usually require changes in another place and that's a hard thing to enforce.

Think about documentation. If you put the documentation for your code in a wiki and not in your code, your developers need to remember to go update that wiki anytime code changes are made. Developers starting putting API documentation in their code because it resulted in more consistent and correct documentation. The two were close together, so remembering to change both places is easier to remember. It was easy to verify in pull requests and code reviews that code changes in one place should be accompanied by a doc change right next to it.

The same goes for these tests. If the test code is far away from the api code it's testing, it's more likely to be wrong.

Let's compare that to approach number #2, where we'll change the test function so that it runs a specific 'test' routine on the API. Let's look first at the test in our Custom Resource handler:

```
async function runTests(url: string) {
  return await axios.post(url + '/test', {});
}
```

Like the website test, we go back to a simple request, now a POST, to the API on an alternative path.

Since the API will need to respond to this request and run the test procedure, the API code needs to change. However, we don't actually want to change the API code directly as we don't want to accidentally change the regular functionality.

Instead, we're going to use a "decorator" to add in the testing functionality without modifying the existing API code (much). A decorator is functional composition, sometimes called higher-order functions.

Let's look at the decorator:

```
1. function addTester(handler: Function) {
2.   return async (event: any, context: any) => {
3.     if (event.httpMethod === 'POST' && event.path === '/test') {
4.       try {
5.         await runTest();
6.       } catch (err) {
7.         return { statusCode: 500, body: err.toString() };
8.       }
9.       return { statusCode: 200, body: 'Test Ran!' };
10.    }
11.    return handler(event, context);
12.  };
13. }
```

This new function, `addTester`, takes the existing Lambda function handler as an argument called `handler` (line 1). It then returns a new function handler that will either run the original handler, or the test code (line 2).

The new function handler will evaluate the `event` and if httpMethod is `POST`

and the path is '/test' (line 3), then the special test code function is called (line 5). Either a success (200) or error (500) response is given back to the client (lines 9 and 7, respectively). If the method **isn't** a POST or the X-Test header doesn't exist (line 3), the original handler is called (line 11).

Decorators are a handy way of extending functionality of a system without having to change existing code extensively. Without a decorator, we'd have to change the original handler function, and that could cause errors both big and small. Here we've added code that said 'if we're trying to test, run that test. If not, do your normal job'.

The `runTest()` function does a write, a read, and a delete on a test record to the database. Using the decorator is simple, we just change the existing Lambda handler code:

```
const originalApplicationHandler = async (event) => { };
export const handler = originalApplicationHandler;
```

to:

```
import { addTester } from './testing-decorator'; // new
const originalApplicationHandler = async (event) => { };

export const handler = addTester(originalApplicationHandler); // changed
```

This allows us to add the new functionality to the Lambda function without editing (and possibly breaking) the existing code. It also keeps the testing code near the API's regular code, which means it's more likely to be right. These two functions exist in the same directory and could exist in the same file.

The `runTest` function itself looks similar to the test in approach #1:

```
import {
    DeleteItemCommand,
    DynamoDBClient,
    GetItemCommand,
    PutItemCommand
} from '@aws-sdk/client-dynamodb';

export async function runTest() {
  console.log('Running test!');
  const tableName = process.env.TABLE_NAME!;
  const client = new DynamoDBClient({
    region: process.env.AWS_REGION || 'us-east-1'
  });
  let itemWritten: boolean = false;
  const testItem = createNewTestItem();
  let key = { PK: testItem.PK };
  try {
    // let's write a record to dynamodb
    const putItemCommand = new PutItemCommand({
      Item: testItem,
      TableName: tableName,
    });

    const response = await client.send(putItemCommand);
    itemWritten = true;
    console.log('Response from write: ', response);

    // then read that record and compare
    const readResponse = await client.send(new GetItemCommand({
      Key: key,
      TableName: tableName,
    }));
    console.log('Response from read: ', readResponse);

    let areEqual = compare(readResponse.Item, testItem);
    console.log('Comparing...', areEqual);
    if (!areEqual) {
      let s = JSON.stringify(readResponse.Item, null, 2);
      let s1 = JSON.stringify(testItem, null, 2);
      throw new Error(`Tests failed, items not equal: \n${s}\n${s1}`);
    }
  } finally {
    // then delete that record
    if (itemWritten) {
      await client.send(new DeleteItemCommand({
        Key: key, TableName: tableName,
      }));
    }
  }
}
```

```
// stupid simple compare function
function compare(one: any, two: any) {
  return JSON.stringify(one) === JSON.stringify(two);
}

// stupid simple test data factory
function createNewTestItem() {
  return {
    PK: { S: 'test#' + new Date().toISOString() },
  };
}
```

With approach #2 we have an API that knows how to verify itself with the added benefit that it can be called from any system, not just our Custom Resource. Here, it reads and writes records to the database to make sure permissions and access are set up correctly. Other APIs could be simpler or more complex in how they validate themselves.

But, the goal with these infrastructure tests is not to validate the API code written by the developers with full coverage, but to test the CDK code written by you. The primary benefit to approach #2 is that it defers the definition of 'working' to the API code, meaning the CDK is more generalized and can be reused, so it scales better.

Regardless of which approach you take, you'll be able to test your API and ensure that any configuration issues that are present are caught as soon as possible and not when your users complain.

9.3.3. Use Case 3 - A Simple Pub Sub Bus

Both of the previous use cases used a synchronous test. When the Custom Resource calls the Lambda function, the handler starts the test, evaluates the results, and registers the Custom Resource as successfully created (or not if the test fails). The test starts and finishes entirely within the one Lambda execution.

Example Code!

You can download an example of this from the Examples repo: https://github.com/cdkbook/examples, look in the 'testing/infra-async' folder.

However, you may have an asynchronous test process that can't be started and completed within the same Lambda execution. Here, the Lambda function for the Custom Resource starts the testing process but won't know when the test is complete. Another part of the testing system will run the tests and report status back to CloudFormation.

Building AWS systems is not in the scope of this book, but if you're interested in learning more, a great place to start is understanding the AWS Well-Architected Framework and the Serverless Lens. It's a guide on how to build your systems to be resilient and reliable. Look specifically for building event-based architectures.

Let's look at testing a simple publish and subscribe bus, or pub-sub. In this process, a message is published to a Simple Notification Service (SNS) Topic. A Simple Queue Service (SQS) queue is a subscriber to that topic. Finally, a Lambda function is a consumer of that queue and will process the messages.

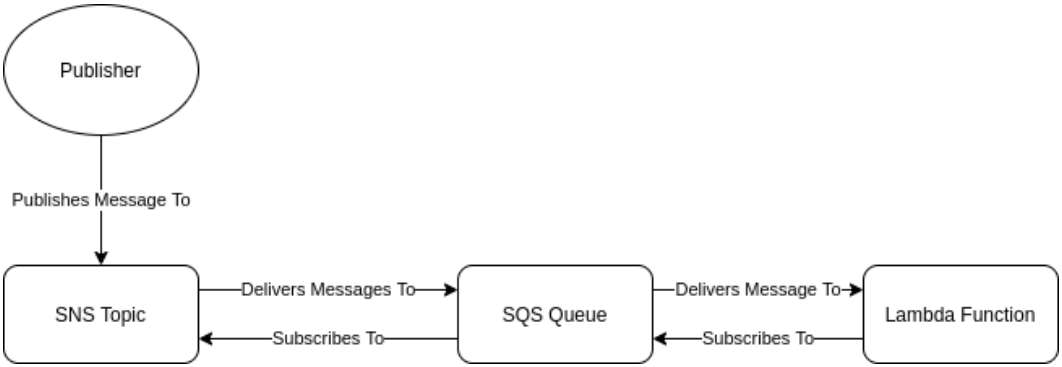

Figure 19. A Simple Pub Sub Pattern

We can't use a single Lambda function for the entire test because the Lambda function that publishes the message wouldn't also subscribe to the

queue. It could start the test, but it wouldn't know when the message was properly received by the SQS message consumer.

We will test this process by passing the Custom Resource information (all the things needed to tell CloudFormation the Custom Resource status) on the test message and let the queue's consumer notify the Custom Resource that it's complete, but that's not very flexible. Once the first test is created against a system, the hard work is done and adding more tests should be easy. Having a better method to orchestrate these tests than burying it in our application code will pay off in the long run. Instead, we're going to use a Step Function State Machine to orchestrate the test and notify the Custom Resource that the test passed or failed.

Step Functions state machines are powerful for coordinating multiple tasks. Our test case will be simple, but this could easily be expanded to encompass multiple different tests and ensure all pass. Adding more tests won't change the overall functionality of the Custom Resource and system. Here's what it looks like:

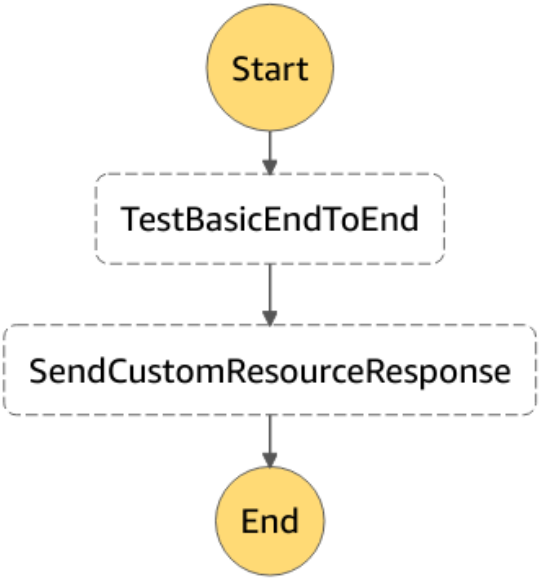

Figure 20. A Basic Test State Machine

The workflow is started by the Custom Resource when it starts a new instance of the state machine. The input event for the state machine and the first state is information from the Custom Resource. This will be used later to notify CloudFormation about the status of the Custom Resource.

```
{
  "customResource": {
    "PhysicalResourceId":
"2021/07/24/[$LATEST]d43720545ff847f393ed3333959d490c",
    "StackId": "arn:aws:cloudformation:us-east-
1:0123456789012:stack/pubsub/ab766750-e747-11eb-b5c3-12235dab63f5",
    "RequestId": "195687b2-7d7d-4f42-969f-6c0afcb14a20",
    "LogicalResourceId": "TestsStartTestsCustomResourceD227E88A",
    "ResponseURL": "https://cloudformation-custom-resource-response-
useast1.s3.amazonaws.com/removedforberivty"
  }
}
```

The first state causes a simple test message to be published to the SNS Topic. That message is passed to the subscriber, the SQS queue. The Lambda function consumes the message from the queue. Like with the API test, the consumer needs to know what to watch for, and respond to this test message. A decorator can be used again, so that the functionality can be added to the Lambda function without changing the normal operation.

```
function addTester(handler: Function) {
  return async (event: SQSEvent, context: any) => {
    console.log('Event: ', JSON.stringify(event, null, 2));

    // look for a test message:
    // this would change depending on how the test message was published
    // in this case, just look to see if any records have a taskToken on them
    // again, demo code. You'd likely want to test differently here.
    const testMessages = event.Records.filter((message: SQSRecord) => {
      try {
        return !!(JSON.parse(JSON.parse(message.body).Message).taskToken);
      } catch (err) {
        // ok, we couldn't parse which means we didn't have the right
structure.
        // I'd love a tryParse or something on JSON but we don't have it,
        // so this makes the most sense
        return false;
      }
    });

    if (!!testMessages && testMessages.length > 0) {
      for (const testMessage of testMessages) {
        try {
          const { taskToken } =
```

```
JSON.parse(JSON.parse(testMessage.body).Message);

          // we got the test message, great!
          // let's let the task know that
          await sfn.send(new SendTaskSuccessCommand({
            taskToken,
            output: '{"success":"The tests passed"}',
          }));
          console.log('Activity sent success');

        } catch (err) {
          console.error('Error when trying to send task success: ', err);
          throw err;
        }

      }
    }

    // let's go ahead and remove the test messages from what the normal
handler now needs to process
    const testRemove = new Set(testMessages);
    event.Records = event.Records.filter(x => !testRemove.has(x));

    // time to do the normal thing
    return handler(event, context);
  };
}

export const originalHandler = async (event: any) => {
  console.log('Event:', JSON.stringify(event, null, 2));
};

export const handler = addTester(originalHandler);
```

This input will be passed through from the first state to the second state and used to respond to the Custom Resource with a status, either success or failure. The first state, TestBasicEndToEnd, publishes a message to the SNS topic which should pass all the way through to the Lambda function. It has a timeout of 30 seconds. If the message isn't received in 30 seconds, the task fails and continues to the next step passing that failure message.

The second state then looks for an error. If one is found, then it's reported back to CloudFormation, causing the Custom Resource to fail and the rollback to begin. If no error is found, a success is reported back to CloudFormation.

Fundamentally, this works similar to the first two use-cases, except now the tests are represented in a Step Functions state machine. Additional states could be added, either in parallel or in sequence, that test other aspects of the system. By using a state machine, very complex workflows can test the system extensively.

One thing to keep in mind: these tests may have side effects, or otherwise affect the runtime of a production system. For example, let's say we wanted to test the Dead-Letter Queue (DLQ) setup. The test would need to send a message that the consumer would be wired to fail, forcing the message back to the queue. After this occurs several times, the message would be pushed to the DLQ. However, if the message is mixed in with non-test messages, then the entire batch would be failed back to the queue, which is bad.

Alternatively, the consumer could manually put the message back on to the queue. But this would require the Lambda function to have permissions it wouldn't normally need. This could open up some dangerous scenarios, like a message that is put back onto the queue infinitely, causing unlimited Lambda executions, which is terrible.

These tests shouldn't be run against systems producing actual data, as it could break normal processing. That only has to occur once for those tests to be ripped out of the system by angry managers. When your tests can't be comfortably executed without side effects, only run them against sandbox environments where any negative side effects could likely be ignored.

And this would likely be part of a CICD pipeline. You could pass a variable to your CDK code that would control adding these tests or not. An automation pipeline would use this flag while production deployments would not. The pipeline would deploy the application to a test environment, and then you'd need to delete it upon completion.

9.3.4. Hotswapping Lambdas

Let's face it, all this testing is great, but sometimes your development cycle is going to be changing some Lambda function handler code and redeploying. Once deployed, you'll run some manual test to see if things are working now. Maybe that's hitting an API endpoint with Postman. Maybe that's reprocessing a message and watching logs.

That's fine too, and the CDK still has you in mind.

In version 1.122.0 of the CDK they released the '--hotswap' option to the CLI. For example:

```
cdk synth --hotswap
```

If the only thing that changed in your CDK code is your Lambda function, the CDK now bypasses the entire CloudFormation system and deploys the Lambda's new code through API calls instead.

Sortly after, Step Function state machine definitions were made hotswappable. Future versions of constructs are likely to support hotswapping.

This deliberately creates CloudFormation drift. This should only be used in development environments.

9.4. Application Code Tests

Your CDK app will often include Lambda function handler code. It's important to test this as well for all the same reasons as testing your CDK code.

If you write your Lambda functions in the same language as the CDK, then you can reuse the testing framework for the CDK! It costs you almost nothing to add application tests to what's already set up.

This book was aimed at helping CDK developers. We're crossing into general application development and there are tons of resources and books available dedicated to these subjects. Testing your Lambda functions isn't inherently dependent on the CDK at all, so all of those existing resources will guide you well. While I see the need to cover some basics here, it felt inappropriate to spend another 40 pages going in-depth. There's no way I could do it justice.

Only some basic examples of application testing will be covered here. Should be enough to show the basic concepts involved, but you will need to explore these details more for your own projects.

9.4.1. A Simple Typescript Example

Let's look at a trivial example where a CDK stack has one Lambda function and the handler just returns a fixed response.

```
from aws_cdk_lib import Stack, StackProps
from aws_cdk_lib.lib.aws_lambda_nodejs import NodejsFunction
from constructs import Construct

class FixedResponseStack(Stack):
    def __init__(self, scope, id, props=None):
        super().__init__(scope, id, props)

        NodejsFunction(self, "FixedResponseHandler")
```

Just a simple Lambda function and the related test:

```
from aws_cdk_lib import App, CfnElement
from aws_cdk_lib.assertions import Capture, Match, Template
from aws_cdk_lib.aws_iam import CfnRole
from aws_cdk_lib.aws_stepfunctions import Pass, StateMachine
from ...src.FixedResponseStack import FixedResponseStack

test("Stack Snapshot", () => {
      const app = new App();
      const stack = new FixedResponseStack(app, 'test');
      const assert = Template.fromStack(stack);

      expect(assert.toJSON()).toMatchSnapshot();
    })

describe("Fine Grained", () => {
      function getTestStack() {
        const app = new App();)
```

With `npm run test`, that code will be verified.

Now, just add another test file:

```
from ...src.FixedResponseStack.FixedResponseHandler import handler

test("Handler", async () => {
      const response = await handler();
      expect(response).toEqual({
        statusCode: 200,
        body: 'Hello World',
      });
    })
```

It'll already be included with your other tests, so it's very simple to cover your application code too.

This example is in Typescript and has a fixed response. Your Lambda functions are probably more complex and have dependencies like databases or external services like email and notifications.

9.4.2. Testing More Complex Lambdas

It was discussed in the 'Unit Test' section where you could mock the Asset class to speed up your unit tests by skipping the steps to actually compile your various Assets. We can also mock other things in a system. Mocking allows a unit test to replace normal functionality of a system with special testing functionality.

Let's say our Lambda handler did something more complex than just return a fixed response. Let's say it made calls to DynamoDB. Now, DynamoDB is a lovely system to work with, and it'd be pretty trivial to create a table specifically for testing and just let the Lambda function read and write to it, but that would require a few things:

- Authentication with the DynamoDB
- An available sandbox table to test with
- An eventual cleanup of any test records created

However, by adding these three things we now need to pass keys around for authentication, ensure there is a table in a clean state for each test and then work in a cleanup process that probably has a 'skip cleanup' option so we can review data if something didn't pass and need to debug.

What if I told you we can skip all of that, too?

Mocking allows you to replace these external systems with an in-memory process you more tightly control. Before we used mocking to skip asset creation, now we can use it to skip making calls to AWS or other systems.

Some additional knock-on effects include: * Faster automated testing * No additional billing costs for infrastructure used only in testing * More resilient tests

Let's review how to use a few different mocking libraries in Typescript and Python.

If you're not using Typescript or Python, here are some places to get started on specific mocking frameworks:

- Go - gomock
- Java - Mockito
- C# - Moq

9.4.3. Typescript/Javascript

Mocking in Jest

Since the CDK uses Jest as its native testing framework, we'll cover Jest's system for mocking. If you have changed your projects to use a different testing system, you may need to find a different mocking library, although most should work. Some will work better with other testing frameworks.

We won't cover a full discussion of Jest's Mocks here, just a simple example that should get you started.

Let's start with the code we want to test:

```
// @ts-ignore
import axios from 'axios';

export const handler = async () => {
  const { data: ipAddress } = await
axios.get('https://api.ipify.org?format=json');
  return ipAddress;
};
```

It simply makes a call to get the IP of the requester and then returns the value back.

Now let's look at the test:

```
import axios from 'axios';
import { handler } from '../src/MockedStack.IPFinder';

jest.mock('axios');
test('Full Snapshot', async () => {
  // given
  let fakeData = { ip: 'fakeaddress' };
  // @ts-ignore
  axios.get.mockResolvedValueOnce({ data: fakeData });

  // when
  const results = await handler();

  // then
  expect(axios.get).toHaveBeenCalled();
  expect(results).toEqual(fakeData);
});
```

Now we're using `jest.mock` to create a fake version of the axios framework. Then it's a matter of using a helper method to specify what we actually want returned when the 'get' call is made:

```
  axios.get.mockResolvedValueOnce({ data: fakeData });
```

When the test runs, the normal axios functionality is replaced with our special testing code. Here, that just returns a value we already defined. No calls are made to the external site.

This provides a huge time and complexity benefit to your unit tests by removing any need for a network connection and all that comes with that, like firewalls, authentication and authorization. By doing this, we ensure our tests have less of a reason to fail. They're less brittle, and less subject to the whims of network connections and external dependencies.

This is just a trivial test, and I recommend you review Jest's entire documentation on Mocks at jestjs.io/docs/mock-functions.

One word of warning, it's very easy to write your tests, and therefore your code, with the wrong mocks. When we first wrote the tests above we incorrectly mocked the return value:

```
axios.get.mockResolvedValue(fakeResults);
```

The mock then returned the 'fakeResults' object ({ ip: 'notarealipaddress' }), but this isn't actually what the actual call would return. The actual call would return an object where the 'fakeResults' were on the data field, like so:

```
{
  "data": {
    "ip": "192.168.1.1"
  }
}
```

If we write my tests wrong, our code will be wrong. While type checking and strongly typing all your variables will help you not make mistakes, it's not perfect. It's good to have a backup.

One trick I learned to do in the past when mocking calls in Javascript is to let the first test run without the mock, make the actual call to the external dependency once, and then output the results to the console as JSON. I can then copy and paste that JSON result into my test directly, ensuring that the mock is returning a valid response. I change that slightly to put in my fake data, but I know the schema starts correct.

Mocking the AWS SDK

If your Lambda function uses the AWS SDK, then there are purpose-built mocking libraries you can use. You can use Jest mocks, and other mocks, but you may find it much easier to use the purpose-built ones.

aws-sdk-client-mock (SDK v3)

The newer AWS SDK v3 for Javascript/Typescript is growing in adoption, and I'll cover it first.

Let's look first at the code being tested:

```
import { DynamoDBClient } from '@aws-sdk/client-dynamodb';
import { DynamoDBDocumentClient, GetCommand } from '@aws-sdk/lib-dynamodb';

const ddbClient = new DynamoDBClient({});

export const handler = async (event: { key: { [key: string]: string } }) => {
  const TABLE_NAME = process.env.TABLE_NAME;
  console.log('event:', JSON.stringify(event, null, 2));
  const client = DynamoDBDocumentClient.from(ddbClient);
  const { Item: item } = await client.send(new GetCommand({ TableName:
TABLE_NAME, Key: event.key }));
  return item;
};
```

Very simple, takes a key from the event, tries to retrieve the record from DynamoDB and passes the Item back to the caller.

And the test, using the 'aws-sdk-client-mock' library:

```
import { DynamoDBDocumentClient, GetCommand } from '@aws-sdk/lib-dynamodb';
import { mockClient } from 'aws-sdk-client-mock';
import { handler } from '../src/MockedStack.AWSv3';

const ddbMock = mockClient(DynamoDBDocumentClient);

test('Mocked Call', async () => {

  // given
  process.env.TABLE_NAME = 'sometable';
  const fakeKey = { pk: 'whatever' };

  const fakeItem = {
    ...fakeKey,
    something: 'else',
  };
  ddbMock.on(GetCommand).resolves({
    Item: fakeItem,
  });

  // when
  const results = await handler({ key: fakeKey });

  // then
  expect(ddbMock.calls().length).toBeTruthy();
  expect(results).toEqual(fakeItem);
});
```

Lines 5, 17, and 18 are where the action happens. First, we explicitly mock the DynamoDBClient from the AWS client. Next, we tell that mock what we expect to be called with. Finally, what we want as a return result is given.

When the test executes, it will bypass making the matching network call to DynamoDB and just return the indicated response. We validate the return result from the handler aligns with data mocked from DynamoDB Here, we can use type checking to make sure we're returning correct results.

Further reading is available on the NPM package page.

aws-sdk-mock (SDK v2)

The v2 mocking library works similar to the v3. Let's look at the code being tested first:

```
import * as AWS from 'aws-sdk';

export const handler = async (event: { key: { [key: string]: string } }) => {
  const ddbClient = new AWS.DynamoDB.DocumentClient();
  const TABLE_NAME = process.env.TABLE_NAME!;
  console.log('event: ', JSON.stringify(event, null, 2));
  const { Item: data } = await ddbClient.get({ TableName: TABLE_NAME, Key:
event.key }).promise();
  return data;
};
```

Here, using the v2 DocumentClient from DynamoDB to make a similar call and response.

And the test:

```
import * as AWS from 'aws-sdk';
import * as AWSMock from 'aws-sdk-mock';
import { GetItemInput } from 'aws-sdk/clients/dynamodb';
import { handler } from '../src/MockedStack.AWSv2';

test('Mocked Call', async () => {
  // given
  AWSMock.setSDKInstance(AWS);
  process.env.TABLE_NAME = 'sometable';
  const fakeKey = { pk: 'whatever' };
  const fakeItem = {
    ...fakeKey,
    something: 'else',
  };
  AWSMock.mock('DynamoDB.DocumentClient', 'get', (_params: GetItemInput,
callback: Function) => {
    console.log('DynamoDB.DocumentClient', 'get', 'mock called');
    callback(null, { Item: fakeItem });
  });

  // when
  const results = await handler({ key: fakeKey });

  // then
  expect(results).toEqual(fakeItem);
  AWSMock.restore('DynamoDB.DocumentClient');
});
```

The v2 mocking language is a little less friendly, but basically the same set up. You'll mock a specific call on a client with a provided response.

Further reading is available at the NPM package page.

9.4.4. Python

Mocking with unittest.mock

Python has the 'unittest.mock' library to provide mocking functionality in your tests. This section won't cover all the ways to mock your calls, but just

one of the more common ways.

I recommend you read more on the developers guides.

Let's first look at the handler that is being tested, a simple function:

```
import json
import requests


def handler(event, context):
    response = requests.get('https://api.ipify.org?format=json')
    return response.json()
```

This just makes a GET request to an endpoint and returns the body of the results. We call this a proxy.

Let's look at the test that will mock that call to the external endpoint:

```
    @patch('requests.get')
    def test_service_gets_mocked(self, mock_get):
        # given
        mock_get.return_value = Mock()
        fake_value = {'ip':'notarealip'}
        mock_get.return_value.json = Mock(return_value=fake_value)

        # when
        response = handler({}, {})

        #then
        self.assertEqual(fake_value, response)
        mock_get.assert_called_with('https://api.ipify.org?format=json')
```

First there is a 'patch' decorator applied to the test. We discussed decorators in the section about Infrastructure Tests. Here, this decorator will create a new mock on the 'requests' module 'get' function and then passes it as the second parameter, 'mock_get'.

The next few lines set up the return value of the mock and then the handler is called. The results from the handler are compared to the results we faked. They should be the same. Finally, to double-check things, it's asserted that the mock was called with the expected value.

Calls to other external services and libraries can be done similarly. You can also force exceptions to occur when the mock is called, verifying your error handling code.

Mocking with the AWS SDK

There are a number purpose-built AWS SDK mocking libraries, like moto and ddbmock. We won't be covering those here, as they're more specific than I felt warranted covering in this book.

While you could mock the boto3 library with unittest.mocks, there is built-in functionality in boto3 for handling it.

Stubber is a function of the Boto3 library. There are a lot of good examples and documentation out there, but you can start here.

First, the code we want to test:

```
def handler(event, context):

    response = AWSClient.dynamodb.get_item(
        TableName=os.getenv('TABLE_NAME'),
        Key=event['key']
    )
    return response['Item']
```

Just a simple call to DynamoDB, getting the Table Name from the environment and the Key from the event. It returns the one item retrieved.

Now the test:

```
class DynamoDBHandlerTestCases(unittest.TestCase):
    def test_service_gets_mocked(self):
        # given
        # set up the client Stubber
        client = AWSClient.dynamodb
        stubber = Stubber(client)

        # the fake result
        get_item_response = {
            "Item": {
                'PK':{'S':'Something'}
            }
        }

        # setup the fake result in the stubber
        stubber.add_response('get_item', get_item_response)

        # when
        with stubber:
            # make our handler call
            response = handler(event={'key': {}}, context=None)

        #then
        self.assertEqual(get_item_response['Item'], response)
```

There are a few things going on, so let's review them one by one:

Line 5: Get our common singleton reference to the AWS DynamoDB client. Line 6: Create a new Stubber for that client. This makes the client use any replacement functionality we define. Line 9: Create a fake response object for the DynamoDB client's Stubber to use. Line 16: Tell the stubber to replace the 'get_item' call with a mock one that just returns the provided object and skips making any API calls to AWS. Line 18: Activate the Stubber. Now any 'get_item' calls will be replaced. Line 24: Assert we got the fake results we expected.

This is just a simple example, and you may write more complicated scenarios, but notice how we never properly supplied a Table name in the environment or a proper Key on the handler's event because they don't

matter.

Now we have a test that can run in a network-less environment and still validate our code. We've mocked the dependency, making less for us to set up and maintain to run these tests.

9.5. Review

Testing your CDK code will evolve over time. You can start small and work your way into mature and robust testing systems. Take baby steps, and before long you will be sprinting.

[4] At the time of this writing the assertions library was in alpha status. By the time you read this it will hopefully be included in the main aws-cdk-lib module. Additionally, so that all these tests will compile they'll be written using Jest. However, the assertions library is built with the jsii and is used the same in other languages to test your constructs.

[5] Refer to the chapter on Integrated Development Environments for more on how to use eslint directly in your editor.

[6] There is an option in CloudFormation to not roll back a stack on failure, so this assumes you wouldn't choose that option.

10. Enterprise Construct Libraries

The CDK really starts to shine within an Enterprise environment, this is where you can take advantage of inner/open source reuse through constructs to realise your cloud adoption goals in a manner that meets all your enterprise compliance and guardrail needs.

The ultimate goal of constructs within the Enterprise environment is not to "abstract the cloud" but rather to create the environment where your engineers feel empowered to rapidly deliver business value.

10.1. How To Explain AWS CDK to Central Infra Support Teams

This may not apply to your organization but it is a real scenario where some companies have invested heavily in CloudFormation based automation, this is typically supported by a central team. When you suggest using the CDK, the question gets raised how they will support both the CDK and the existing CloudFormation processes.

It also raises a skillset concern because the central teams supporting the CloudFormation Templates produced today may now need to be experts in CloudFormation template YAML but also TS/Java/Python/.Net etc to match all of the jsii languages.

The reality here is that you need to view the CDK as a developer acceleration tool on top of CloudFormation. Yes, it may come in many languages but the end result is at least one CloudFormation stack.

This creates a phenomenal opportunity for partnership between developers wanting to use the latest cloud features and the central teams supporting the shared infra. They can work together to identify the guardrails needed. Now

the motivated and invested developer can produce one of the types of enterprise constructs mentioned later in the chapter, accelerating usage of the new cloud feature in the company. The central team supports the CloudFormation guardrails and the developer uses an inner source model for support of the acceleration construct.

10.2. How To Drive Adoption From Engineers

One of the most controversial aspects of any reuse effort is how to gain adoption by the wider engineers you work with. For any technology to be successful long term it has to be adopted by choice rather than mandate.

The very first thing you need to do is educate everyone on how to use the CDK. For that we would recommend you fork the open source code behind cdkworkshop.com and tailor it to work on your AWS accounts. Now you have the beginning of a learning pathway.

Next we are talking real implementations. For some of the enterprise construct types below like the base class implementation, you need developers to be using it on day one when they start a new project. They shouldn't even need to care that it is specific to your company if you do it right.

It is our opinion that a set of working patterns that developers can clone to be up and running using the latest standards is the only natural way to drive this adoption. The creation of a starter / skeleton pattern based on the least amount of code needed to have a working, compliant pipeline to production is especially key.

On day one you don't need a big internal platform, just a common repo like cdkpatterns.com is enough to start. Then as your usage increases you can follow user needs.

The trick is that if you can pair your tailored cdkworkshop implementation with the compliant, starter pattern you have now created a funnel where developers go from 0 to working pipeline and just need to experiment after they leave.

For construct creators, the above needs to be paired early with your internal jsii build process to release constructs in all desired languages. Don't leave that for everyone to figure out independently or you will end up heavily TypeScript biased. TypeScript is a great language choice but so are all of the other jsii supported languages, if you make everything other than TS feel less than then you will alienate a large section of your developer community. For more details about packaging construct libraries, please read the publishing constructs chapter.

If you can drive an inner source model on construct creation combined with working, compliant examples of construct implementation for consumption in multiple jsii languages then you can create a self sustaining flywheel.

10.3. Types of Enterprise Construct

For the purposes of this book, let us consider there to be three types of Enterprise Construct:

1. Internal, customized abstractions
2. Core reusable utilities
3. Base stack implementations

10.3.1. Internal, Customized Abstractions

We would define these as the solution to a situation where your current required standards differ significantly from the default provided by AWS. One note of caution however, if the implementation is not specific to your company and would be useful to others then we would suggest releasing an open source construct in the first instance.

Example Situation

Don't picture a simple construct like one to ensure encryption is enabled on an S3 bucket (you can warn developers about that using CDK Aspects), think about the situation where your company requires all API Gateways to be REST APIs, with each route secured by a custom Authorizer Lambda Function and protected by AWS WAF. The Auth logic will be specific to your organization so you don't want to go open source; Inner source is a better fit at least to start.

The reason for creating this construct is that you do not want every developer in the company to take time working out how to configure all of those pieces. Realistically, you should be using a standardised Auth Lambda Function implementation that gets updated regularly and the WAF rules

managed externally from the development teams. In this situation, an internal construct for a SecureRESTAPI makes sense.

An implementation from the consumer viewpoint could look something like:

```
from company_name.secure_rest_api import SecureRestApi
import aws_cdk.aws_lambda as lambda_

example_fn = lambda_.Function(self, "exampleFn",
    runtime=lambda_.Runtime.NODEJS_12_X,
    code=lambda_.Code.from_asset("lambdaFns/exampleFn"),
    handler="handler.example"
)

apigw = SecureRestApi(self, "exampleGateway", {
    "stage_name": "dev"
}, StackConfiguration)

apigw.add_lambda_integration_route("/test", "POST", example_fn)
```

Underneath, this construct can be associating an Authorizer Lambda implementation with the API Gateway route but 99% of developers do not need to reimplement that solution.

Naming

It is very important with these constructs to never create a suite of {CompanyName}{AWS Service Name} constructs e.g. CDKBookAPIGateway, CDKBookS3Bucket or CDKBookDynamoDBTable. You need to follow standard naming conventions and name the construct after a single purpose.

If you use generic names, developer adoption will be hard due to the level of investigation needed to uncover the implementation details. This is true in open source but especially true in inner source models where small inconveniences can have huge impact on developer engagement.

On the example above if you have multiple different Authorization choices in your company, you can take advantage of Object Oriented programming techniques so that the class names are specific to the Authorization method but they share common implementation logic underneath. e.g. Auth0SecureRestApi, PingSecureRestApi, CognitoSecureRestApi can all be an implementation of SecureRestApi

Tracking Usage

Since you know these constructs are being deployed into your company's AWS accounts, you can track usage using tags. All you need to do is add a tag to the stack with your construct name as the key and the version number as the value. You can then pull a report of the breakdown of versions deployed in every AWS account where you have permissions. Alternatively you can have a common identifier as the key and the construct name combined with the version number as the value then you can easily pull usage for all of your team's constructs.

Documentation

Internally, you should aim for the same documentation standards as if you published externally to The CDK Construct Hub. If you are not deploying an internal instance of the Construct Hub then you need to decide how you will surface these capabilities.

You can start out using a readme.md in the project repository but remember developers need to manually find it and it will not auto generate the code bindings so can easily fall out of date.

We would recommend the following sections:

High Level Overview	You only need a paragraph at most to spell out the unique value of this construct.
Architecture Diagram	This should ideally be generated through something like the cdk-dia construct. It should always be honest and digestible at a glance.
Detailed Feature List	You have built a developer product in this construct, tell your consumers what features they can use. Do you have features on the roadmap? Drop them in here clearly marked as coming soon
Sample Implementation Code (all supported languages)	Nothing speeds up developers more than working code that they can copy paste in their chosen language. You don't need to cover every single feature but all the big hitters are a must.

Post Deployment User Guide	Put yourself in your consumer's shoes. I just deployed your construct, now what can I do with it? Tell people how to find endpoint urls, how to gain auth tokens and anything else needed. Do not assume your consumers are all experts at every deployed piece of your construct (otherwise they wouldn't need it)
Debugging Process	There is nothing worse than using someone else's abstraction and then something stops working but you don't know what is built in to help you diagnose the issue without reading source code, ruining the abstraction. If you setup cloudwatch alarms, enable xray etc tell people how to achieve operational excellence with your construct.
Contribution process	This can be in a separate contributing.md file but without stating the logical process for inner source contributions my bet is you will see more duplicated constructs than collaboration.
Versioned table of changes	This one is optional because if you are using a readme.md you have the full source control system that can be combined with effective commit messages but sometimes it is just nice to highlight some of the evolution of the construct so that potential consumers can check back and see the feature they needed was added.

Exposing Internals

Unlike in an open source construct where it would not be considered best practice, you need to make sure that the AWS L1/L2 constructs are accessible to any consumers of the construct so they can experiment with changes before feeding back to the main implementation. This is due to the most common use case for inner source constructs being pre-configured infrastructure that meets your compliance standards vs external where a construct is typically a complete abstraction to solve a problem. There will always be edge cases where developers need to use a feature that you haven't accounted for yet so exposing those internals keeps everyone moving at speed for a small tradeoff of some optional abstraction leakage.

Yes, developers could always fork and experiment but that is a massive extra congnitive load on top of the simple change they want to make, plus there is no guarantee you merge the pull request in a time frame that meets their business goals.

We can use the code above to demonstrate a situation where the construct has only implemented LambdaIntegration. What if the consumer wants to have a direct SNS integration on the same gateway through VTL? They need access to the CDK LambdaRestAPI object directly after adding the LambdaIntegration routes.

Implementing this access is fairly straightforward, all we need to do is have a public LambdaRestAPI object in our construct.

```
from aws_cdk.aws_apigateway import LambdaRestApi
import aws_cdk.aws_lambda as lambda_

class SecureRestApi(cdk.Construct):

    def __init__(self, scope, id, props):
        super().__init__(scope, id)
        # ...
        self.lambda_rest_aPI = self.create_gateway()

    def create_gateway(self):
        pass

    def add_lambda_integration_route(self, route, http_method, lambda_fn):
        pass
```

then consumers can access and modify the API Gateway directly

```
apigw = SecureRestApi(self, "exampleGateway", {
    "stage_name": "dev"
}, StackConfiguration)

apigw.add_lambda_integration_route("/test", "POST", example_fn)

# direct access to the LambdaRestAPI
lambda_rest_aPI = apigw.lambda_rest_aPI
```

After they have implemented this change for themselves, if you have an inner source model they can open a Pull Request and start the process of enabling others based on already working code. The overall goal should never be to lock people into your abstraction, just to accelerate them to working software.

10.3.2. Core, reusable utilities

Your deployed AWS applications across the company will share at least some infrastructure, this could be in the form of a custom resource you call to integrate with your non AWS resources or it could be using shared VPCs.

In these instances you can significantly improve developer experience by providing constructs that are pre configured. Typically you will bundle all of these utilities together in one monorepo rather than separate build/publish pipelines per utility. Then you can include the utilities in the starter pattern mentioned above and every developer has out of the box access.

10.3.3. Base Stack Implementations

The base stack is your opportunity to craft a developer experience when deploying CDK stacks in your organization. At the lowest level it allows you to do things like enforce tagging standards but at a more advanced level you can use Aspects to smooth over rough edges for developers or add in some default features like VPC auto lookup:

```
from aws_cdk.core import Stack, Construct, Aspects
from company_name.vpc import VpcUtilities

class CdkBookBaseStack(Stack):
    def __init__(self, scope, id, props=None):
        super().__init__(scope, id,
            (SpreadAssignment ...props
              props)
        )
        # You can add whatever logic you want to be in every stack
        self.default_vPC = VpcUtilities.lookup_default_vpc()
```

The above means any CDK stack that extends CdkBookBaseStack can just use the defaultVPC variable without needing to perform the lookup logic themselves.

10.4. Governance and compliance

The most important thing we must stress is that you should never base your entire governance strategy at the AWS CDK level. This would be akin to only

having JavaScript validation on a website with a wide open API, your rules need enforced at the AWS account level through something like Control Tower. Anything you do in AWS CDK is about developer experience, acceleration and education.

This may lead you to ask "why bother with governance and compliance at this level?", technically you could just leave the governance at the AWS account level. The problem this causes, is that you end up in a situation where your developers embrace a "deploy and hope" strategy. They will never have confidence that any change they make will successfully deploy until they see the post deploy tests show up green and real traffic flows through the changes.

If you are undergoing a cloud transformation, only having AWS account side validation will be akin to driving with the parking brake on. Driving at slow speeds you may think everything is normal then you try to accelerate faster and notice a burning smell in the car until finally you lose the ability to brake.

If your engineers do not have full confidence in their ability to deploy into your AWS accounts then every time a change fails, they will naturally assume it is related to governance. When moving slow is acceptable, everyone will joke about it over tea/coffee but as expectations grow for you to move faster, the trust will be completely eroded to the point developers will fall back onto their trusted implementations. If you can shift that compliance left into the developers IDE you build trust rather than lose it.

If you want to meet in the middle you can pair AWS CDK with tools like CloudFormation Guard to run rulesets against the generated CloudFormation. The advantage being these rules will work across any dev tool that produces CloudFormation rather than specific implementations per tool. This is still an evolving space with some limitations today but it is something to keep an eye out for in the future.

We have seen three strategies for compliance implementation in AWS CDK:

1. Proactively modify non-compliant resources
2. Throw errors about non compliance
3. Internal, customized abstractions

A word of warning before explaining the technique for proactively modifying non-compliant resources, there is a dangerous lack of context around the changes you are applying. If you write a rule to set all S3 Buckets to automatically have encryption enabled and include it in your base stack, there will probably be an edge case where that breaks certain applications. This will cause a really bad experience for your developers trying to debug that the change happened through your Aspect and it is not easy to ignore that one resource. This technique is still very useful however if you bring in open source constructs which don't meet your compliance rules, in that instance the developer can choose to add their own aspect on that stack to bring it into a compliant standard without having to fork and maintain an internal version of the open source construct.

Now that we have warned you, let's look at how you would find all S3 Buckets and enable encryption. This approach works best for simple compliance rules that do not need any extra input to remediate (like a customer managed KMS key) or if the developer is custom writing their own specific aspects like discussed in the last paragraph. The exact syntax for implementing this kind of rule would look like:

```
import aws_cdk.core as cdk
import aws_cdk.aws_s3 as s3

class S3BucketEncryptionEnablement(cdk.IAspect):
    def visit(self, scope):
        if scope instanceof s3.CfnBucket:
            if typeof scope.bucketEncryption == "undefined":
                scope.bucket_encryption = {
                    "server_side_encryption_configuration": [{
                        "server_side_encryption_by_default": {
                            "sse_algorithm": "AES256"
                        }
                    }]
                }
```

The next level is when you throw a warning or an error and stop the AWS CDK stack from deploying. You will still be using CDK Aspects to find the out of compliance resource but in this situation you need the developer to change something themselves before a deploy will work successfully. In your error message it is important that you link to how developers can find out detailed documentation around this error somewhere on an internal information store.

```
import aws_cdk.core as cdk
import aws_cdk.aws_s3 as s3

class S3BucketCMKEncryptionCheck(cdk.IAspect):
    def visit(self, scope):
        if scope instanceof s3.CfnBucket:
            if typeof scope.bucketEncryption == "undefined":
                # Throw an error
                cdk.Annotations.of(scope).add_error("Compliance Error 23456:
All S3 Buckets must use a customer managed KMS Key.")

                # Warn the developer they may need to do some work
                cdk.Annotations.of(scope).add_warning("Possible compliance
error 23456, if this is desirable you will need to request an exception
before deploy will succeed.")
```

The final level which is where you should land if the previous two approaches aren't enough, this is to build your own compliant construct that

developers can use. The details of this were discussed under the previous Internal, customized abstractions section. The only additional information I will add here is just to always keep your consumer in mind, someone needs to support the infrastructure deployed through your construct so you need to make it as transparent and easy as possible. Make sure all compliance rules implemented are listed in the documentation with links to more detail.

10.5. Review

Just to wrap up, the CDK and the Enterprise work fantastically well together. As you can hopefully see from the chapter you just read, you can use it as a bridge between dev and infra teams who maybe traditionally only spoke when something went wrong. It enables a strong inner source culture where teams can layer in their unique learnings on top of one another so everyone can benefit. It even brings the cognitive load down of learning aws (the unique way that is tailored to your Enterprise) to first deploying an L3 construct, then you can dive deeper as you learn.

11. Deployment Strategies

In this chapter, we want to look into several ways to deploy your CDK apps into AWS accounts and to create CI/CD pipelines to automate this. We will also talk a bit about version control and branching, as this impacts how to structure your pipelines.

Continuous integration describes the goal of integrating code changes, done by every team member, in a short time frame to prevent drift in the code base of several branches. The common target is to merge changes at least once a day. All changes are the automatically built and tested to provide fast feedback to the developer.

Continuous deployment describes the process to take every change in a certain branch and deploy it to your test and production environments in an automated way. This includes unit testing, end-to-end testing and often load testing of your software in multiple setups.

While running `cdk deploy` locally on your development machine is the easiest way to materialize your CDK app, it has several drawbacks regarding determinism, automation, and dependencies on people and their machines. Therefore, we want to create pipelines that are deployed and run in some managed environment, and we only commit our changes to a version control system and let the service do the rest.

We can host these pipelines on AWS CodePipeline or any other CI/CD system. I will assume you are using git as your version control system for this chapter.

11.1. Branching strategies

Several strategies are used throughout the industry to structure your

branches and decide where to base your deployments on.

The most popular ones are trunk-based development and variants of GitFlow. With trunk-based development, you have one main branch and potentially several short-lived branches for changes. In GitFlow, your development happens on a branch called `develop`. Then, if you are satisfied with your code, you merge to a `release` or `main` branch, depending on your interpretation. GitFlow is designed to support release management of software that needs to be maintained in several versions. For applications you deploy on your own using a continuous deployment model, this workflow is not well suited, and is also stated by the author of GitFlow in the documentation. Sometimes there is a middle-ground between the two and you often need to find the right balance that works for you and your team.

Another common branching strategy is to have one branch per stage you want to support, so you might have a `dev` branch, a `qa` branch, and a `prod` branch. Promotion of code and configuration through the stages is done by merging the `dev` to the `qa` branch and so on.

In the following sections, I want to elaborate on why I recommend trunk-based development for CDK apps and clarify why I do not recommend the branch per stage approach.

11.1.1. Trunk-based development

If you follow the trunk-based development strategy, you make several assumptions that I want to highlight:

- You only roll forward by adding commits to the `main` branch
- The `main` branch always needs to work to be able to deploy a change
- Every change goes through all stages. There is no shortcut!
- Integration of changes by other developers happens all the time

One of the most important goals when using trunk-based development is to ensure that you are always able and comfortable pushing the current `main` branch into production, given that all tests pass. You cannot merge features that are not ready to be released as this blocks the whole pipeline. But if you overcome this hurdle, you have a quick deployment cycle and reduced cycle time for your changes.

The terms `trunk`, `master`, and `main` all mean the same in this context and originate from different points in the history of version control systems

11.1.2. Why not one branch per stage?

As mentioned, another popular way of structuring your repository is to have branches per stage or environment. This allows developers to change the code, which means application and infrastructure, of these stages separately. While this seems to give freedom and flexibility, it brings a lot of danger to your release process.

One issue might and will be drift in your infrastructure, and your test and QA environments no longer can tell you if production will work. By being able to shortcut changes into production by committing to the `prod` branch, you also risk crashing your production environment because you skipped all your safeguards and test for the sake of deployment speed.

And even if you enforce and adhere to strict merge processes and make sure that all changes need to go through the stages, you still have the problem of multiple pipelines building the same artifacts repeatedly. This could cause slightly different artifacts because of dependencies that change upstream or non-deterministic build results. You then have untested artifacts running in production for the first time.

What does this mean?

Immutability of assets and artifacts

A critical aspect of CI/CD pipelines is that you need to ensure that the artifacts you deploy into production are the ones you tested in your QA stage. With the most setups I have seen of stage branches, every stage has its own pipeline that builds all needed artifacts and then deploys them to the respective stage. In the case of CDK, that means synthesizing your CloudFormation template once for every stage in the stage's pipeline. The dangerous thing is that the template and any assets created by CDK, like Lambda function code and container images, are created again for every stage and are not the same. This means that we never tested the container running in production in any non-prod environment before going live.

So it is crucial to make sure that you build all your assets exactly once per commit and then deploy this set of artifacts to all of your stages, one after the other.

11.2. Environment / Stages / Waves

In this chapter, we will talk a lot about environments, stages, and waves. So let me clarify what this means, as many companies have different names for these concepts. In this book, we will use the wording the CDK team and AWS use to align with the documentation.

The underlying idea is to have multiple deployments of your infrastructure and application to test before going live, to show to stakeholders, and to simulate certain situations in a deployment that is not your production infrastructure.

Additionally, you can also have multiple production setups for a globally distributed workload or as a fail-over.

As an example for this chapter, let's assume we have a workload that will be

deployed in two US regions and two EU regions for production, and we will also have a dev setup and a QA setup. Our application will consist of one CloudFormation stack for simplicity, but you can also do this with multiple stacks that build one application.

- Dev Setup in eu-central-1 in account 1
- QA setup in eu-west-1 in account 2
- Production setups in account 3
- eu-central-1
- eu-west-1
- us-east-1
- us-west-1

11.2.1. Definitions

Environment

We define an environment as the pair of an AWS account and an AWS region. This is also the level of granularity of one CDK bootstrap, as explained in chapter assets. This environment definition does not correlate with the separation of dev, QA, and prod or any other reduction of the blast radius. It is just the definition of which target environment to use.

In AWS CDK, you can, and should, specify the environment for every stack in the `StackProps` you pass to the constructor.

```
cdk.Stack(self, "Stack1",
    env={
        "account": "123456789012",
        "region": "eu-central-1"
    }
)
```

I highly recommend providing this information explicitly and not using whatever credentials are currently active in your CLI to determine the target environment of your CDK stack instance. For example, suppose you have multiple CLI profiles on your machine and access to several accounts. In that case, you might deploy CDK apps into accounts of other customers, projects, or stages. This is something you definitely want to prevent.

Stage

In a more extensive CDK application, you will have multiple CloudFormation stacks that build one installation of your application - for example, a database stack, a backend stack, and a frontend stack. To group these stacks into one unit, we introduce the concept of stages. A stage is a collection of stacks that are connected and build one application.

An instance of a stage should never reference the resources of another stage. Stages need to be independent to prevent the risk of one stage producing errors in another one.

In AWS CDK there is a class representing this concept, and it is an essential part of CDK pipelines later on.

```
class MyStage(cdk.Stage):
    def __init__(self, scope, id, env):
        super().__init__(scope, id, env=env)
        db_stack = MyDatabaseStack(self, "database")
        MyBackendStack(self, "backend", db_stack)

MyStage(self, "Development", {
    "env": {
        "account": "123456789012",
        "region": "eu-central-1"
    }
})
```

We then deployed these stages one after another in your CI/CD pipeline to propagate your changes from dev, to QA and finally to all your production setups.

For a full example see the section about CDK Pipelines.

Wave

If you have a lot of stages, your time from commit to deployment of the last stage might be too long. To reduce the deployment time of new features, you can group stages into deployment 'waves'. All stages in one wave are deployed in parallel and speed up the total pipeline execution time.

So in our example, we might want to deploy the European and the American stages as waves, which lead to the following grouping.

- Deploy development
- Deploy QA
- Deploy Europe (both stages)
- Deploy America (both stages)

We achieve this in your AWS CDK pipeline by creating a wave and then adding stages to it.

```
prod_wave_eU = pipeline.add_wave("ProductionWaveEurope")
prod_wave_eU.add_stage(MyStage(self, "ProdFra", prod_env_fra))
prod_wave_eU.add_stage(MyStage(self, "ProdDub", prod_env_dub))
```

You should still make sure that you only group stages at the same level in your pipeline. For example, grouping QA and prod would not be a good idea, as failures you find in QA would already be live because of the parallel deployment.

But if you have multiple production setups, it is good to deploy one stage first and then increase the parallelism of the deployments further down your pipeline.

See Clare Liguori's post in the Amazon Builder's Library about safe deployments for a more in-depth discussion on deployment strategies.

11.3. Basic deployment flow

So, with the definitions being clear, let's look at the basic structure of a CI/CD pipeline for CDK apps. We want a deterministic behavior of the deployment. To make sure we really get what we want, we will synthesize all templates in one step at the beginning of the pipeline. Then we deploy these artifacts to our stages.

The first step is to install all the dependencies and build all needed artifacts referenced in your CDK app, like any code folders for Lambda or Docker, etc. To avoid surprises, installing dependencies should use a lock file with pinned versions to get reproducible builds.

In the next step, we can run `cdk synth` to create our cloud assembly in the `cdk.out` folder. This produces all CloudFormation templates and assets, and we can ignore our source code from this point on. The assembly folder is self-contained and needs to be propagated to all further steps in your pipeline.

Our first operation on this assembly is to upload all assets, if we have any, to their respective target environments. This will push docker images to ECR repositories and upload zip files to S3. The ECR repository and the S3 bucket used are from the bootstrapping process described earlier in chapter assets.

With all assets in place, we can then deploy the first stage. Instead of running `cdk deploy <stackname>` directly, which would recreate the assembly from our source code, we tell the CDK CLI to use our cloud assembly by calling `cdk --app cdk.out deploy <stackname>` and pointing CDK to our assembly folder. We can also have nested assemblies when using CDK pipelines and stages so that we would call `cdk --app cdk.out/assembly-<name> deploy '*'` which deploys all stacks in the given sub-assembly.

If the first stage succeeds and all our automated and manual tests are fine, we can proceed to the next stage until all stages are deployed. By splitting the deployment into multiple jobs, we can already start new deployments in earlier stages while deployments of a previous version are still running in later target stages.

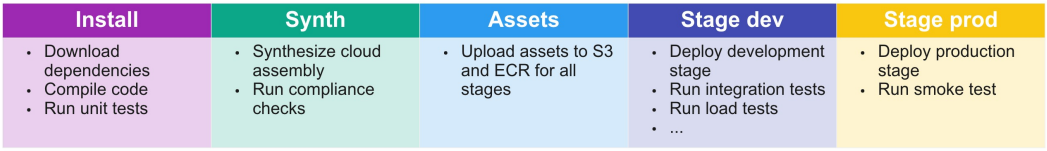

Figure 21. Pipeline stages

11.4. CDK Pipelines

As mentioned earlier, the AWS CDK has an integrated solution for deployment processes called CDK Pipelines. This project provides a construct to define all relevant parts of a deployment pipeline like information about the source code repository, the targets stages and environments, and deployment groups like waves.

The default implementation of this pipeline API is using the AWS CodeSuite tools like CodeCommit for source code, CodeBuild to execute jobs, and CodePipeline to orchestrate the steps. In addition, all needed IAM permissions and cross-account policies are set up automatically to create a great developer experience.

Other implementations, like Github Actions, are on the roadmap or even in a preview phase.

11.4.1. General implementation

To get started, we import the pipelines module and create a new stack for the pipeline. We will deploy this stack into our CI/CD account and orchestrate the deployment and initialize a new CodePipeline and provide the basic information.

```
import aws_cdk_lib as cdk
import aws_cdk_lib.pipelines as pipelines
from constructs import Construct

class Pipeline(cdk.Stack):
    def __init__(self, scope):
        # provide your CI/CD account info here
        super().__init__(scope, "Pipeline", env=ci_env)

        pipeline = pipelines.CodePipeline(self, "Pipeline",
            # we need to activate this for cross account deployments
            cross_account_keys=True, ...
        )
```

This pipeline stack is the only stack we add to our CDK app in the main file.

```
app = cdk.App()
Pipeline(app)
app.synth()
```

The only other required information for the pipeline is the configuration of the synthesize step. Next, we need to tell CDK Pipelines how to fetch and build our source code.

```
pipeline = pipelines.CodePipeline(self, "Pipeline",
    cross_account_keys=True,
    synth=pipelines.ShellStep("Synth",
        # configure source repo here
        input=pipelines.CodePipelineSource, ...,
        # configure installation of dependencies here
        install_commands=["yarn install --frozen-lockfile"],
        # configure build steps here
        commands=["npx cdk synth"]
    )
)
```

11.4.2. Source providers

CDK pipelines support multiple source providers out of the box. For example, you can select GitHub, BitBucket, S3, or CodeCommit as your source repository.

On every change, the pipeline will start a new execution and deploy your application.

The configuration of your source happens as factory methods on the CodePipelineSource class:

GitHub, GitHub Enterprise, BitBucket using a CodeStar connection

To get started with CodeStar connections, you need to log in to the AWS console and add this connection manually. It needs permissions to the third-party repo, which are granted interactively. Our pipeline definition will then use the ARN of the created connection. You can reuse one CodeStar connection for many pipelines based on different repositories at the same source provider. So one connection for, e.g. GitHub, is enough.

```
pipelines.CodePipelineSource.connection("org/repo", "branch",
    connection_arn="arn:aws:codestar-connections:eu-central-
1:123456789012:connection/11111111-2222-3333-4444-555555555555"
)
```

CodeCommit

If you want to use the AWS native repository service CodeCommit, you can either create or import that the CodeCommit repository and then use CodePipelineSource.codeCommit to reference it:

```
# Pick one
repository = codecommit.from_repository_name(self, "Repository", "my-
codecommit-repository")
repository = codecommit.Repository(self, "Repository", ...)

pipelines.CodePipelineSource.code_commit(repository)
```

S3

You can also use a ZIP file in an S3 bucket as your source, and every time the file changes, the pipeline is triggered. This is useful if you have a source repository that is on-premises or not using git.

```
bucket = s3.Bucket.from_bucket_name(self, "Bucket", "my-source-bucket")
pipelines.CodePipelineSource.s3(bucket, "my/source.zip")
```

11.4.3. Synthesizing the app

In your CodePipeline, you have to add a build step that synthesizes your CDK app. Then, you can instantiate a new object of the `pipelines.ShellStep` class and provide as necessary properties.

The minimal information you need to provide is the source and the build command as shown. But you can also configure a lot more to tweak the configuration of your CDK build.

Additional sources

You can add a secondary repo or another source to your pipeline that will also trigger a deployment. For example, you can use this to deploy when your repo changes or the upstream Docker base image gets updated.

By providing the source under the `additionalInputs` property you specify the folder to check out the source code into.

```
additionalInputs: {
  '../siblingdir': pipelines.CodePipelineSource.gitHub('org/source2'),
}
```

Environment variables

The `env` property allows setting variables that will be available in during sythesis like proxy settings, authentication to repositories or registries, etc.

```
env: {
  'HTTP_PROXY': 'http://myproxy.company.com',
}
```

11.4.4. Self-mutation

After the CDK app synthesizes, it will check if the pipeline is still current or if there are structural changes like new assets, new waves, or new deployments. If this is the case, the CDK pipeline will automatically mutate the pipeline using `cdk deploy` and restart the pipeline once this update is done.

The benefit of this behavior is that you do not need to care about adding new stages or waves upfront, as the pipeline will just add them on demand.

If you modify existing stages, self-mutation will cancel running tasks that are currently deploying older invocations of the pipeline to said stages. So be careful when changing existing stuff.

If you need to test some changes to your pipeline, you can disable self-mutation and deploy changes manually using `cdk deploy` from your local

machine. Set `selfMutation: false` on the `CodePipeline`.

11.4.5. Assets

After successfully creating the Cloud assembly and ensuring the pipeline is up-to-date, all created assets are uploaded to their respective destinations. S3 assets are transferred to the asset buckets, and Docker images are built and pushed to the ECR repositories in the target environment.

By default, all assets are processed in parallel using separate CodeBuild jobs. However, you can configure CDK pipelines to limit assets processing to one job per asset type by setting the `publishAssetsInParallel` property to `false`. The advantage of this setting is that adding or removing assets does not mutate the pipeline anymore ant for many small assets, it is also cheaper as the minimum billing amount of every CodeBuild job is one minute which could be used to upload several Lambda function code assets at once.

11.4.6. Deployment of stages and waves

Now that we have made sure that the pipeline is up-to-date and all assets are in place, we can deploy our application to the different stages. Each wave we deployed after the previous one. If a wave or stage fails, the pipeline will stop promoting this commit, and the deployment will stop. We can then provide a fix in another commit, let it flow through our pipeline, and make everything right. Each wave consists of one or more stages, and each stage represents one set of stacks that make up our application in one defined AWS account and AWS region.

By default, all stages and waves are deployed until one fails or we reach the end of the pipeline. However, if we begin using pipeline, we might not be comfortable deploying every change to production automatically and might want to approve changes manually. To do this, we add a

`ManualApprovalStep` in our stages and waves that wait for human interaction to continue. We can add the manual approval in front of the stage if we want to wait for human interaction before deploying a stage. The second option is to add it after a stage to have the pipeline wait for a signal before completing the stage.

While the result is very similar and it seems more like a technicality, there is a significant semantic difference here.

If we are planning to approve the deployment of a stage, we should use the `pre` step as it will cumulate all changes, and when we finally approve, it will deploy the latest version to our target stage.

Using the `post` step, we indicate we want to confirm the correct deployment of the stage before the pipeline can proceed or deploy anything else to this stage. So in confirmation after the stage, we prevent updates to this stage until we approve the action. This is meant for manual validation of a deployment to make sure everything works as desired.

```
production_stage = MyStage(self, "ProdStage", env=prod_env)
test_stage = MyStage(self, "TestStage", env=test_env)

pipeline.add_stage(test_stage,
    post=[
        pipelines.ManualApprovalStep("ConfirmTestSetup")
    ]
)

pipeline.add_stage(prod_stage,
    pre=[
        pipelines.ManualApprovalStep("PromoteToProd")
    ]
)
```

We can also add additional `ShellSteps` to stages to automate these validations we would add using the `post` action. If we can run automated acceptance tests for our deployments, this is the place to activate them.

We can specify commands that are then executed in a CodeBuild project and also inject values of the deployed application that the tests just use.

Reusing the stage definition from the example before this would be implemented as follows:

```
pipeline.add_stage(test_stage,
    post=[
        pipelines.ShellStep("SmokeTest",
            env_from_cfn_outputs={
                # Make the load balancer address available as $URL inside the
commands
                # the field needs to be a CfnOutput created in a stack (ARN,
URL, etc)
                "URL": lb_app.load_balancer_address
            },
            commands=["curl -Ssf $URL"]
        )
    ]
)
```

If you want to provide the tests as code in your repo instead of adding commands here, you can reference the source code artifact of the synth step.

```
pipeline.add_stage(test_stage,
    post=[
        pipelines.ShellStep("SmokeTestFromSource",
            input=synth.add_output_directory("testFolder"),
            commands=["cd testFolder && ./test.sh"]
        )
    ]
)
```

11.5. Roll your own pipeline

If you do not want or cannot use AWS CodePipeline, you can also use any other CI/CD solution to mimic the same principles. Currently, CDK

pipelines only support AWS CodePipeline as an engine, but you can achieve the same using the CDK CLI.

Given that tools like GitHub Action, BitBucket pipelines, or Gitlab CI jobs describe their configuration in a file in your repository, we do not need the self-mutation step. So our pipeline comes down to a build and synth step, an asset upload, and numerous deployments based on the generated cloud assembly.

By default calling the deploy or diff command will synthesize your app by running the command configured in your `cdk.json` file as `app` like running NodeJS or Python. The CDK CLI allows overriding your CDK application's app configuration per CLI execution by specifying the `--app` option. If you provide a folder instead of a runnable command, CDK assumes that this folder is a previously generated cloud assembly and uses it to execute the desired command.

So running deploy or diff will not execute your code again but use templates and assets that were created once by running `cdk synth`.

So disregarding all needed steps to install dependencies, run local tests, and so on, our pipeline will look similar to this:

```
# Synthesize all templates to the cdk.out
cdk synth

# Deploy our testing stage and reference the assembly folder
cdk deploy --app 'cdk.out/' MyWebserver-dev

# Do some tests here and approve test stage

# Deploy our qa stage
cdk deploy --app 'cdk.out/' MyWebserver-qa

# Do some tests here and approve QA stage

# Deploy our production stage
cdk deploy --app 'cdk.out/' MyWebserver-prod
```

Each of these commands should then be executed as a separate job or step depending on your CI/CD system. Finally, you need to make sure that you pass the generated assembly artifact around as it is not part of the repository checkout.

11.5.1. Outlook

The separation of CDK pipeline blueprint classes like `ShellStep` and the CodePipeline implementation like the concrete `CodeBuildStep` makes it possible to implement other variants in the future. So it will be possible to describe pipelines as shown before and synthesize GitHub action definitions or GitLab job configurations using the same feature-rich set of blueprints as for CodePipeline.

11.6. Review

In this chapter we learned the pros and cons of different branching models and pipeline implementations. No matter what decision you make, the most important part is to version your code and to automate the deployment instead of doing manual runs of `cdk deploy` on your local machine.

12. Importing CloudFormation Templates

While you might look at adopting the AWS CDK for projects, you probably also have a lot of CloudFormation files already being used throughout your environments. You may choose to leave the existing CloudFormation templates as they are. There are many reasons you might choose to, because the templates aren't being changed anyway, are not in your direct control, or priorities don't allow you to replace that code.

Thankfully, the CDK team gave you the option to import existing CloudFormation templates into your CDK code. Perhaps you just want to put a CDK Pipelines CI/CD pipeline around the deployment of that template. Maybe you'd like to use something akin to a Strangler pattern to replace your existing template with CDK code.

Let's review what how the importing functionality works.

Earlier versions of the CDK had a CfnInclude construct in the 'core' library. That has been deprecated in favor of the `@aws-cdk/cloudformation-include` module.

The `CfnInclude` construct can be used with existing templates. Its props has 4 fields:

- templateFile - the path to the template you want to import
- loadNestedStacks (optional) - If using nested stacks, a map of child stacks and their associated properties
- parameters (optional) - A map of parameter values to provide to the template
- preserveLogicalIds (optional, default to true) - Whether to keep the template's logical IDs for resources or replace them.

As always, the module docs provide good examples of how to use the

construct.

Rather than just repeat the docs, I'll cover two specific uses cases.

First is when you just want to incorporate an existing template into a stack and use its values with other constructs to extend the functionality of the template without having to rewrite it all in the CDK.

The second is using the existing template as a blueprint for replacing the entire template with CDK constructs.

12.1. Extending Existing Templates

You can import existing CloudFormation templates into your stacks and then make changes or additions.

Start by importing the template with the construct.

If you can provide parameters, doing so will prevent them from synthesizing. The CDK advises against using Parameters as they move information that should be part of your CDK apps outside to the environment, making the template less deterministic.

Once imported, you can add to its definition or add other constructs to your stack that reference it.

```
import path as path
from aws_cdk.aws_iam import AccountPrincipal, PolicyStatement, Role
from aws_cdk.aws_s3 import CfnBucket
from aws_cdk.cloudformation_include import CfnInclude
from aws_cdk.core import Construct, Stack, StackProps

class ExtendedStack(Stack):
    def __init__(self, scope, id, props=None):
        super().__init__(scope, id, props)

        template_with_bucket = CfnInclude(self, "ExistingThing",
            template_file=path.join(__dirname, "bucket.yaml"),
            parameters={
                "BucketAcl": "PublicRead"
            }
        )

        cfn_bucket = (template_with_bucket.get_resource("S3Bucket"))
        cfn_bucket.versioning_configuration = {
            "status": "Enabled"
        }

        bucket_maintainer_role = Role(self, "BucketMaintainer",
            assumed_by=AccountPrincipal(Stack.of(self).account)
        )
        bucket_maintainer_role.add_to_policy(PolicyStatement(actions=
["s3:*Get*"], resources=[cfn_bucket.attr_arn]))
```

This allows you to do some level of maintenance of existing templates with CDK code.

The import process does not process includes and transformations. Also, some yaml might have trouble parsing, so passing it through a converter to get JSON might get you a better template to import.

12.2. Replacing Existing Templates

Perhaps you're looking to replace your existing CloudFormation templates with CDK code. Great! The importing can help you with that.

Start by importing the existing template and creating a snapshot unit test. Refer to the Testing chapter's Unit Tests section for more information about writing snapshot tests.

```
from aws_cdk.assertions import Template
from aws_cdk.core import App
from ...src.ReplacementStack import ReplacementStack

test("Snapshot", () => {
      const app = new App();
      const stack = new ReplacementStack(app, 'test');
      const assert = Template.fromStack(stack);
      expect(assert.toJSON()).toMatchSnapshot();
    })
```

Once the snapshot is taken, you can remove the CfnInclude and start using L2 and L3 constructs (and L1 if still needed) in its place. You'll probably have to re-run the tests many times until you get everything exactly right.

But, you don't have to wait until everything matches 100%. If something changes that you're comfortable with (like S3 auto-delete functionality) then you'd just work until only those differences remain and then update the snapshot.

```
from aws_cdk.aws_s3 import Bucket, BucketAccessControl, CfnBucket
from aws_cdk.core import Construct, Stack, StackProps

class ReplacementStack(Stack):
    def __init__(self, scope, id, props=None):
        super().__init__(scope, id, props)

        bucket = Bucket(self, "S3Bucket",
            access_control=BucketAccessControl.PUBLIC_READ
        )

        (bucket.node.default_child).override_logical_id("S3Bucket")
```

You are certain to get new Logical IDs along the way unless you override it in your code. The Testing chapter has some examples in the refactoring section

about how to change a resource's Logical ID.

12.3. Review

You will hopefully be writing CDK code from here on out, but if you still need to leverage existing templates, the CfnInclude construct should help. It is also useful for when you are migrating to the CDK.

13. Combining AWS CDK With Other Dev Tools

The AWS CDK is an incredible tool that can be used completely stand alone but sometimes you want to use the strengths that other tools bring to the table. This is not a complete list of tools, just a small sample.

13.1. AWS CDK and AWS SAM

AWS SAM is the Serverless Application Model, an open-source framework for building serverless applications.

In April 2021 a public preview was announced that allows you to natively use the local features of SAM on a CDK application.

Eric Johnson created a great blog post detailing how this all works called "Better together: AWS SAM and AWS CDK"

13.1.1. sam build

If you are using the standard Lambda Function construct then you need to add steps to manually install the dependencies and compile the Lambda Fn source code. You can use other constructs like aws-lambda-nodejs to make this process automatic or you can run the sam-beta-cdk build command which will magically install and package your lambda functions into a deployable artifact

13.1.2. sam local invoke

This allows you to invoke a specific named Lambda function with a defined mock event and can pass in set environment variables.

```
sam-beta-cdk local invoke CDKStackName/FunctionID -e events/mock.json -n env-
local.json
```

13.1.3. sam local start-api

You can mock some of the AWS API Gateway functionality locally using SAM if you have one in your stack. Note, this does not support 100% of the features of API Gateway (like custom auth lambda fns) at time of publish so a deployed test is still more accurate. This is still useful though.

```
sam-beta-cdk local start-api -n env-local --warm-containers EAGER
```

Now you should have a server on port 3000 that you can test through your tool of choice

13.1.4. sam local start-lambda

This starts a local emulator of the Lambda service that you can then use to invoke your Lambda Function via the SDK or CLI.

So you have two commands:

One to start the mock Lambda service

```
sam-beta-cdk local start-lambda -n env-local.json
```

Then one to invoke the function

```
aws lambda invoke --function-name [function logical-name] --endpoint-url
http://127.0.0.1:3001/ response.json
```

13.2. AWS CDK and AWS Amplify

AWS Amplify is a set of purpose-built tools and services that makes it quick and easy for front-end web and mobile developers to build full-stack applications on AWS.

In December 2021 it was announced that you can now import Amplify projects into AWS CDK as a stack. This new capability allows frontend developers to build their app backend quickly and, each time it is ready to ship, hand it over to DevOps teams to deploy to production.

This should be as simple as:

```
amplify export --out ../cdkproject/lib
```

then inside your CDK project you can use the AmplifyExportedBackend construct

```
from aws_amplify.cdk_exported_backend import AmplifyExportedBackend

class AmplifyStage(Stage):

    def __init__(self, scope, id, props=None):
        super().__init__(scope, id, props)

        # appname is the name of your amplify project
        amplify_stack = AmplifyExportedBackend(self,
"amplifyexportedbackend",
            path=path.resolve(__dirname, "..", "amplify-export-appname"),
            amplify_environment="dev"
        )
```

13.3. AWS CDK and CDK for Terraform

Cloud Development Kit for Terraform (CDKTF) allows you to use familiar programming languages to define cloud infrastructure and provision it through HashiCorp Terraform.

In December 2021 a technical preview was announced that allows you to import AWS CDK constructs into CDKTF using the AwsTerraformAdapter. The AwsTerraformAdapter uses the aws_cloudcontrolapi_resource Terraform resource to communicate with the AWS Cloud Control API.

The list of supported resources for the Cloud Control API is available at https://docs.aws.amazon.com/cloudcontrolapi/latest/userguide/supported-resources.html.

The offical code for this currently looks like:

```
import {
  Duration,
  aws_lambda
} from "aws-cdk-lib";
import { AwsTerraformAdapter, AwsProvider } from "@cdktf/aws-cdk";

export class MyStack extends TerraformStack {
  constructor(scope: Construct, name: string) {
    super(scope, name);

    new AwsProvider(this, "aws", { region: "us-east-1" });

    const awsAdapter = new AwsTerraformAdapter(this, "adapter");

    new aws_lambda.Function(awsAdapter, "lambda", {
        //...
        handler: "index.main",
        runtime: aws_lambda.Runtime.PYTHON_3_8,
    });
  }
}
```

14. Next Steps

Thank you for reading our book!

We have included a lot of information on these pages but this is not where your AWS CDK journey has to end. We hope that you take the next steps and build real applications, this means that you will have further specific questions or concerns that will make you want to interact with experienced builders.

Well we have good news for you. We have an incredibly supportive community for all the CDK products over at cdk.dev. There, you can join our Slack community where there is a channel called the-cdk-book for questions about this book and a generic aws-cdk channel for you to join and ask anything.

15. Appendix 1: Migrating from CDK 1 to 2

The AWS Cloud Development Kit released v1 GA in July 2019. At time of publishing (Dec 2021), v2 has just been released at re:invent 2021.

How fun, that we get a long history of the CDK just from looking at the git repository.

```
commit 2db0565cd23a4d828dbc415b6b7ecd7cd4382057
Author: Elad Ben-Israel <benisrae@amazon.com>
Date:   Wed May 30 12:18:52 2018 +0300

    Initial external release: repository skelaton
```

From these first humble commits became a very powerful system for defining the resources we manage in AWS.

Now, about 40 months later, v2 has been released.

15.1. Major Changes

The v2 release of the CDK was focused primarily around easing a specific pain point with the users of the CDK: making it easier to manage your CDK dependencies.

V1 of the CDK published separate packages to the various registries (npm, pypi, maven, nuget). If you wanted to use a construct in a module, explicit dependencies were needed, separate from the cdk `core` dependency. Over time, you may see a dozen or more '@aws-cdk/*' dependencies.

The resulted several issues, but most prominent was that it was easy with node/npm to end up with mismatching package versions.

For example, the `core` package would be on an older 1.124.0 when you install `npm i -s @aws-cdk/aws-s3` and the latest version is now 1.130.0, so that package is now 1.130 while core is still on 1.124.0. This mismatch in versions causes compilation errors.

V2's biggest change is that all the first party constructs, like those in `@aws-cdk/aws-s3`, are included in one monolithic package instead of individual packages. Now, developers must only add a single dependency to get access to all the first party constructs and libraries.

Other changes in v2 are mostly around future stability in the API:

- Experimental modules will be kept out of this monolithic package until their API stabilizes. If you'd like to use an experimental module, you will need to add it directly by using the v1 style package name with an '-alpha' suffix, e.g. @aws-cdk/assertions-alpha. Once the API is stable, the 'experimental' note is removed, and it's merged into the main monolithic package. You can remove the specific dependency.
- The core `Construct` class has been removed from the CDK and it's now a peer dependency from the 'constructs' library. This allows the CDK to be more consistent with other similar projects like cdk8s, CDK for Terraform, and projen. There are some changes to this core functionality, so if you're creating advanced constructs for the CDK, you will want to review and see if you can upgrade.
- Deprecated properties are removed. If you're still using deprecated properties, you will need to update your code to the new API. Refer to the docs on each property to see recommended replacement APIs.
- Feature flags previously off by default are now on by default.
- All bootstrapped environments need updating to the modern bootstrap stack.

When upgrading your CDK code to v2, you may get to the end and find that a `cdk diff` is producing some unexpected changes. This is typically caused

by there being a large jump between the v1 code you used (<1.85.0) and v2. Make sure you do the upgrade in a branch of your git repository so that you can easily shelve them.

Resolving unexpected changes usually means you have to upgrade your v1 code to a more recent version of v1, like v1.100.0+, and then work through minor changes in your constructs. Once upgraded to the latest v1 code, then try the v2 upgrade steps again.

There are a couple of pre-requisites before upgrading to v2:

- For TypeScript developers, TypeScript 3.8 or later is required.
- The latest v2 version of the CDK Toolkit is recommended. To install it, run `npm i -g aws-cdk`.

I've been using the v1 cli with v2 libraries during pre-release and I have not run across problems. If you do the same but get odd errors, I recommend upgrading to v2 of the cli.

15.2. Upgrading Existing Projects

Upgrading an existing project involves a few steps and updates to your code. Most of these steps are the same for all projects but any steps that are language or project type (projen or other) will be detailed separately.

- Start with a clean git repo and clean build. You should have no changes, either staged or unstaged, in your git repository. You should be able to run a successful build (either `yarn build` or `npm run build`). Stash any changes, or resolve any issues and commit changes before proceeding.
- Update your external dependencies. If you initialized your project with `cdk init`:

  [] Typescript (package.json):

```
{
  "dependencies": {
    "aws-cdk-lib": "^2.0.0",
    "@aws-cdk/assert": "^2.0.0",
    "constructs": "^10.0.0"
  }
}
```

If you're writing a construct library, you'll want your dependencies to be listed in both devDependencies and peerDependencies:

```
{
  "peerDependencies": {
    "aws-cdk-lib": "^2.0.0-rc.1",
    "constructs": "^10.0.0"
  },
  "devDependencies": {
    "aws-cdk-lib": "2.0.0-rc.1",
    "@aws-cdk/assert": "^2.0.0-rc.1",
    "constructs": "^10.0.0",
    "typescript": "~3.9.0"
  }
}
```

- Python (setup.py):

```
install_requires=[
      "aws-cdk-lib>=2.0.0",
      "constructs>=10.0.0",
      # ...
],
```

- Java (pom.xml):

```
<dependency>
    <groupId>software.amazon.awscdk</groupId>
    <artifactId>aws-cdk-lib</artifactId>
    <version>2.0.0-rc.1</version>
</dependency>
<dependency>
    <groupId>software.constructs</groupId>
    <artifactId>constructs</artifactId>
    <version>10.0.0</version>
</dependency>
```

- C# (.csproj):

```
<PackageReference Include="Amazon.CDK.Lib" Version="2.0.0-rc.1" />
<PackageReference Include="Constructs" Version="10.0.0" />
```

- If you initialized your project with projen:
  - [] Update the .projenrc.js file, find the `cdkVersion` value in your project, it will be something like '1.XX.Y'. Change this version to `2.XX.Y`.
  - [] Run `npx projen` to update your dependencies.
- Update any third party dependencies. If a third party dependency hasn't upgraded to v2 of the CDK, you will need to replace its functionality or delay your upgrade until all your constructs will work with v2.
- You need to update all references in all your files to 'aws-cdk-lib' instead of '@aws-core/*'.
  - [] Typescript:

```
import { Construct } from 'constructs';
import { App, Stack } from 'aws-cdk-lib';
import { aws_s3 as s3 } from 'aws-cdk-lib';
import { Bucket } from 'aws-cdk-lib/lib/aws-s3';
```

- Python:

```
from constructs import Construct
from aws_cdk import App, Stack
import aws_cdk as cdk
from aws_cdk import aws_s3 as s3
```

- Java:

```
import software.constructs.Construct;
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;
import software.amazon.awscdk.App;
import software.amazon.awscdk.services.s3.Bucket;
```

- C#:

```
using Constructs;         // for Construct class
using Amazon.CDK;         // for core classes like App and Stack
using Amazon.CDK.AWS.S3;  // for service constructs like Bucket
```

- Remove any imports of 'Construct' from core and replace with import from 'constructs'
- Remove any imports of 'Tags' from core and replace with import from 'aws-cdk-lib'
- Remove any imports of 'Environment' from core and replace with import from 'aws-cdk-lib'
- Remove all feature flags from your context. All previous feature flags in v1 are true by default in v2. If you need, there are three feature flags being carried forward from v1 if you still need that functionality:
  - [] `@aws-cdk/aws-apigateway:usagePlanKeyOrderInsensitiveId` - If your application uses multiple Amazon API Gateway API keys and associates them to usage plans.
  - [] `@aws-cdk/aws-rds:lowercaseDbIdentifier` - If your application uses Amazon RDS database instance or database clusters, and explicitly

specifies the identifier for these.

- [] `@aws-cdk/core:stackRelativeExports` - If your application uses multiple stacks and you refer to resources from one stack in another, this determines whether absolute or relative path is used to construct AWS CloudFormation exports.

An example cdk.json file:

```
{
  "context": {
    "@aws-cdk/aws-rds:lowercaseDbIdentifier": false,
    "@aws-cdk/aws-apigateway:usagePlanKeyOrderInsensitiveId": false,
    "@aws-cdk/core:stackRelativeExports": false
  }
}
```

15.3. Find and Replace with your IDE

VS Code and IntelliJ products 'find and replace in files' function supports regex. This makes it easy to go through your entire project and make the 'import' changes needed. While this is useful in Typescript, Python only needs a change of the Construct import mentioned above, and you don't need to make any other changes to the way you import modules.

Webstorm:

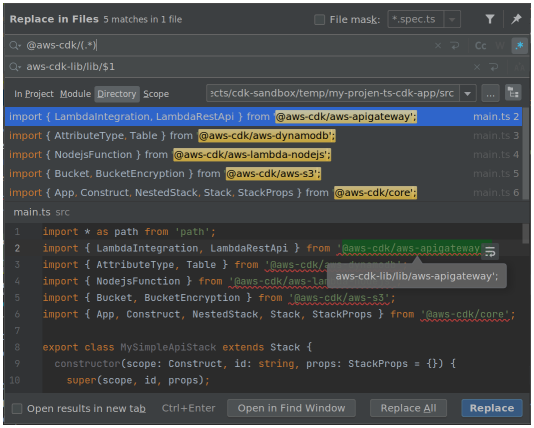

Figure 22. Webstorm's find and replace in files window

VS Code:

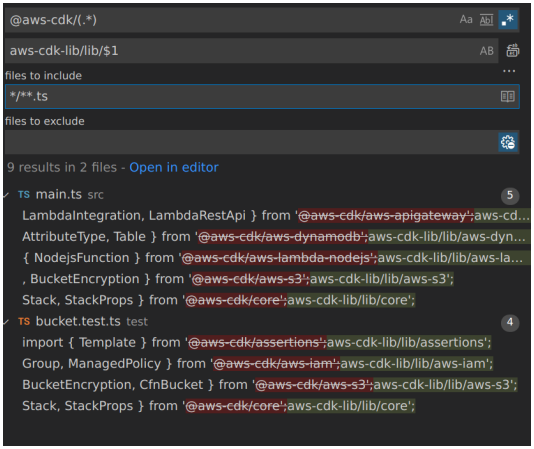

Figure 23. VS Code's find and replace in files window

For Typescript:

Search: `@aws-cdk/(.*)`

Replace: `aws-cdk-lib/lib/$1`