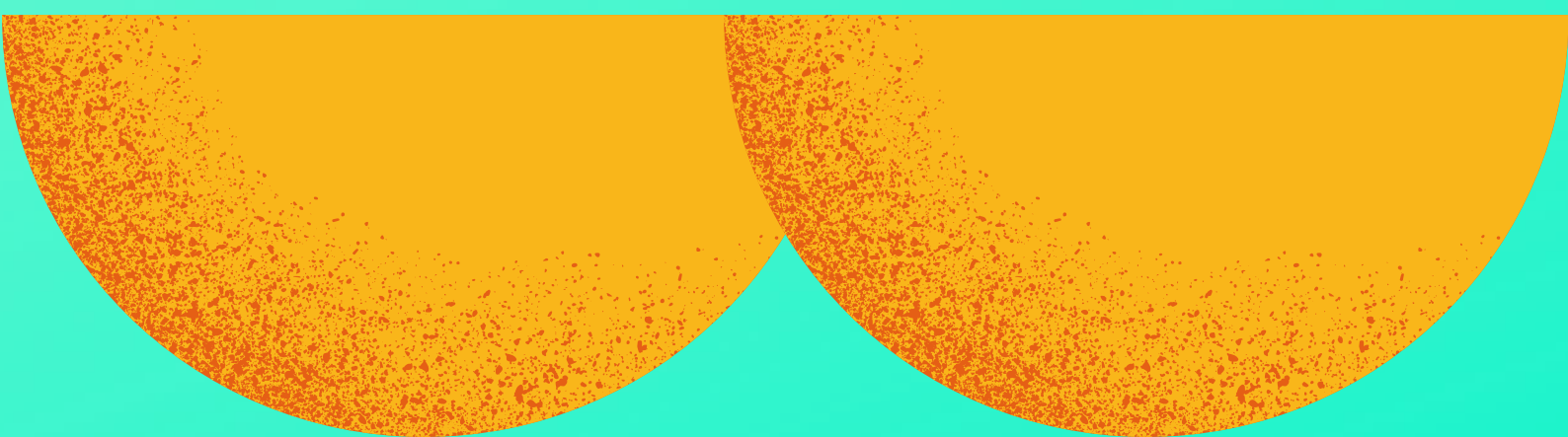
AWS Cloud Development Kit CookBook

RAHUL LOKURTE



Introduction to Cloud Development Kit
- What is AWS CDK
- Working of AWS CDK
- AWS CDK Installation

AWS Lambda using CDK
- Introduction
- Create Lambda Function
- Deployment

AWS API-Gateway Using CDK
- Introduction
- Lambda Integration With Proxy
- Lambda Integration Without Proxy

AWS S3 Using CDK
- Introduction
- Create an S3 bucket
- Deployment

AWS Step Function Using CDK
- Introduction
- Create a Step Function
- Deployment

AWS Code Artifact Using CDK
- Introduction
- Create an CodeArtifact
- Deployment

Introduction to Cloud Development Kit

Infrastructure-as-Code is fast emerging as a de-facto standard for development organizations. The principle driving force of Devops is to treat infrastructure same as the application code. We write application code in some defined format which is according to the rules of programming language in which you are writing the code. Similarly, to manage and provision the infrastructure, we write configuration files. We write all the configuration files in a declarative way. we also want to store the configuration files in source control system similar to application code.

Infrastructure was traditionally provisioned using scripts and manual processes. These scripts were stored in version control systems or documented step by step in text files or play-books. Often the person writing the play books is not the same person executing these scripts or following through the play-books. If these scripts or runbooks are not updated frequently, they can potentially become a show-stopper in deployments. This results in the creation of new environments not always being repeatable, reliable, or consistent.

IaC allows you to build, change, and manage your infrastructure in a safe, consistent, and repeatable way by defining resource configurations that you can version, reuse, and share across different environment. Some of the IAC Tools are : Ansible, Puppet, Terraform, Cloud Formation, Cloud Development Kit.

What is AWS CDK

The AWS Cloud Development Kit (AWS CDK) is an open source software development framework to define your cloud application resources using familiar programming languages. We can use all the features that the given language provides. With CDK developers can use existing IDE, testing tools, and workflow patterns. By leveraging tools like autocomplete and in-line documentation, AWS CDK enables you to spend less time switching between service documentation and your code.

AWS CDK enables you to reference your runtime code assets in the same project with the same programming language.
For example, you can include your AWS Lambda runtime code in your CDK project, and when you deploy your application, the CDK framework automatically uploads and configures the AWS service with your runtime assets. When the CDK deployment is complete, you will have a fully functional application.

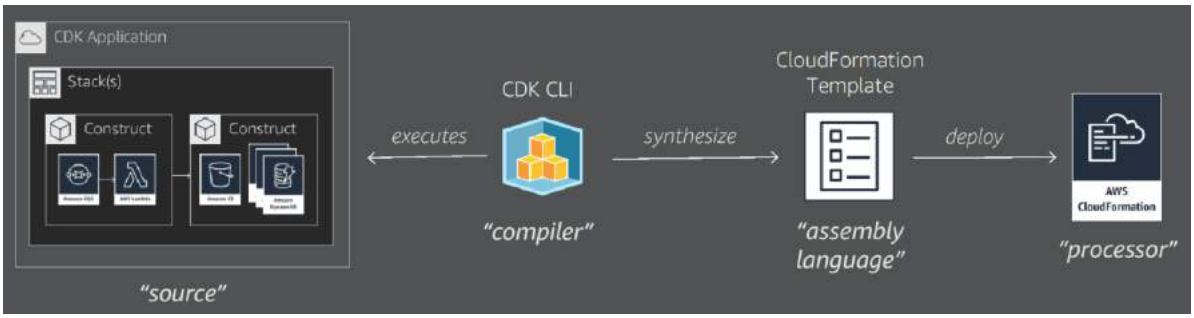
The AWS CDK provides an CLI which enables you to interact with your CDK applications and enables functionality such as showing the differences between the running stack and proposed changes, confirming security related changes prior to deployment, and deploying multiple stacks across multiple environments.

Working of AWS CDK

As you can see from the below diagram, There are two core concepts in AWS CDK. Those are **stacks** and **constructs**.

Stack is an unit of deployment in AWS CDK. All the AWS resources defined within the scope will be provisioned as single stack. Constructs defines one or more aws resources. It is defined in such a way that, we can reuse the constructs as a single component consisting of mutiple AWS resources.

Once we define the Stacks and Constructs, we synthesize the source using CDK CLI compiler to generate a cloudformation template and then deploy the infrastructure.



AWS CDK Installation

Before CDK can be installed, we need to make sure to install the AWS CLI by using below commands

```
curl "https://awscli.amazonaws.com/AWSCLIV2.pkg" -o
"AWSCLIV2.pkg"
sudo installer -pkg AWSCLIV2.pkg -target /
```

Once AWS CLI is installed, we need to configure using the command

```
aws configure
```

Now, you have to manually create or edit the `~/.aws/config` and `~/.aws/credentials` (macOS/Linux) or `%USERPROFILE%\.aws\config` and `%USERPROFILE%\.aws\credentials` (Windows) files to contain credentials and a default region, in the following format.

- In `~/.aws/config` or `%USERPROFILE%\.aws\config`

```
[default]
region=us-west-2
```

- In `~/.aws/credentials` or `%USERPROFILE%\.aws\credentials`

```
[default]
aws_access_key_id=<Your AWS Access Key>
aws_secret_access_key=<Your AWS Secret Access>
```

Once AWS CLI is configured, we now Install the AWS CDK Toolkit globally using the following npm command.

```
npm install -g aws-cdk
```

Run the following command to verify correct installation and print the version number of the AWS CDK.

```
cdk --version
```

Many AWS CDK stacks that you write will include external files that are deployed with the stack. So, we need to bootstrap the aws account by issuing the below command.

```
cdk bootstrap aws://ACCOUNT-NUMBER/REGION
```

We have setup the AWS CLI and the CDK in this lesson. In the future lessons, we will see provisioning of some commonly used AWS services using CDK.

AWS Lambda using CDK

Introduction

Infrastructure as code has become a go-to process to automatically provision and manage cloud resources. AWS provides two options for infrastructure as code.

1. AWS Cloud Formation
2. AWS Cloud Development Kit

With Cloud Formation, we have to write a lot of YAML templates or JSON files. As AWS adds more services, we have to add more files to Cloud Formation. It becomes difficult to work with lots of files. YAML/JSON is based on data serialization and not an actual programming language. The AWS CDK will overcome the limitations of cloud formation by enabling the reuse of code and proper testing.

AWS CDK is a framework that allows developers to use familiar programming languages to define AWS cloud infrastructure and provision it. CDK provides the **Constructs** cloud component that cover many of the AWS services and features. It helps us to define our application infrastructure at high level.

Create Lambda Function

we will create a Lambda Function and the infrastructure around lambda function using AWS CDK.

Create a new directory on your system.

```
mkdir cdk-greetapp && cd cdk-greetapp
```

We will use **cdk init** to create a new JavaScript CDK project:

```
cdk init --language javascript
```

The cdk init command creates a number of files and folders inside the **cdk-greetapp** directory to help us organize the source code for your AWS CDK app.

We can list the stacks in our app by running the below command. It will show CdkGreetappStack.

```
cdk ls
CdkGreetappStack
```

Let us install AWS lambda construct library.

```
npm install @aws-cdk/aws-lambda
```

Edit the file **lib/cdk-greetapp-stack.js** to create an AWS lambda resource as shown below.

```
const cdk = require("@aws-cdk/core");
const lambda = require("@aws-cdk/aws-lambda");

class CdkGreetappStack extends cdk.Stack {
  /**
   *
   * @param {cdk.Construct} scope
   * @param {string} id
   * @param {cdk.StackProps=} props
   */
  constructor(scope, id, props) {
    super(scope, id, props);
    // defines an AWS Lambda resource
    const greet = new lambda.Function(this, "GreetHandler", {
      runtime: lambda.Runtime.NODEJS_14_X,
      code: lambda.Code.fromAsset("lambda"),
      handler: "greet.handler",
    });
  }
}

module.exports = { CdkGreetappStack };
```

- Lambda Function uses NodeJS 14.x runtime
- The handler code is loaded from the directory named **lambda** where we will add the lambda code.
- The name of the handler function is greet.handler where **greet** is the name of file and **handler** is exported function name.

Lets create a directory name **lambda** in root folder and add a file **greet.js**.

```
mkdir lambda
cd lambda
touch greet.js
```

Add the lambda code to **greet.js**

```
exports.handler = async function (event) {
  console.log("request:", JSON.stringify(event, undefined, 2));
  let response = {
    statusCode: 200,
    body: `Hello ${event.path}. Welcome to CDK!`,
  };
  return response;
};
```

Deployment

Before deploying the AWS resource, we can take a look on what resources will be getting created by using below command.

```
cdk diff
```

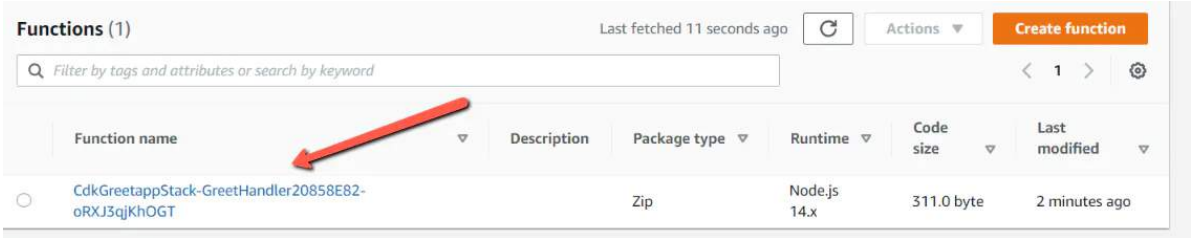
NOTE: If we have multiple profiles set in our system, we need to tell cdk to look into particular profile. This can be done, by adding below key-value in **cdk.json** which was generated when we created a CDK project.

```
"profile": "<YOUR_PROFILE_NAME>"
```

Now, once we are ok with the resources which will be created, we can deploy it using below command:

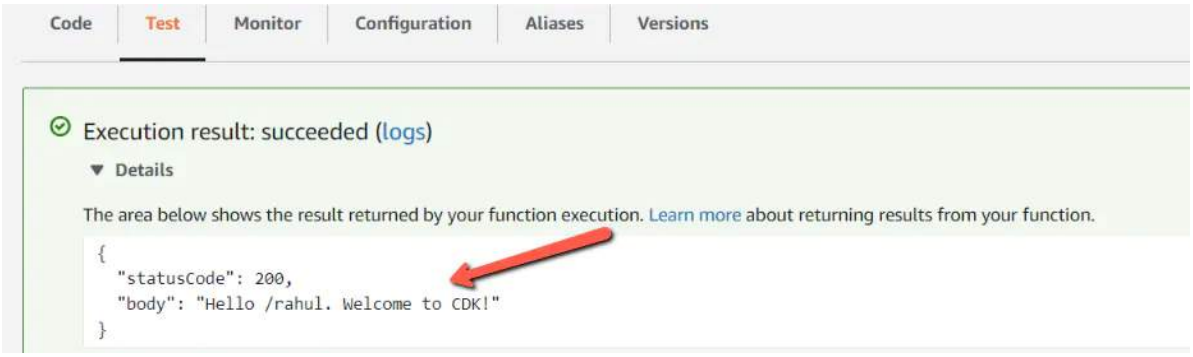
```
cdk deploy
```

Let us open the AWS Lambda console



Select Amazon API Gateway AWS Proxy from the Event template list.

Click on Test, we can see that, we get the proper response as shown below.



we saw how to create a lambda function and also lambda resource by using AWS Cloud Development Kit. We also saw various commands related to CDK for initiating projects, deploying the resources to AWS. The code repository link is here

AWS API-Gateway Using CDK

Introduction

Amazon API Gateway is a fully managed service that makes it easy for developers to publish, maintain, monitor, and secure APIs at any scale. Create an API to access data, business logic, or functionality from your back-end services, such as applications running on Amazon Elastic Compute Cloud (Amazon EC2), code running on AWS Lambda, or any web application.

AWS CDK (Cloud Development Kit) is a framework that allows developers to use familiar programming languages to define AWS cloud infrastructure and provision it. CDK provides the **Constructs** cloud component that covers many of the AWS services and features. It helps us to define our application infrastructure at a high level.

We will create an API Gateway using AWS CDK and connect it to lambda which we were developed in chapter **AWS Lambda using CDK**

There are two types of integration of API Gateway with Lambda that can be done. We can have the Lambda proxy integration or the Lambda custom integration.

Lambda Integration With Proxy

A very common practice is to use Amazon API Gateway with AWS Lambda as the backend integration. The **LambdaRestApi** construct makes it easy.

Edit the file **lib/cdk-greetapp-stack.js** to create an AWS Gateway resource as shown below.

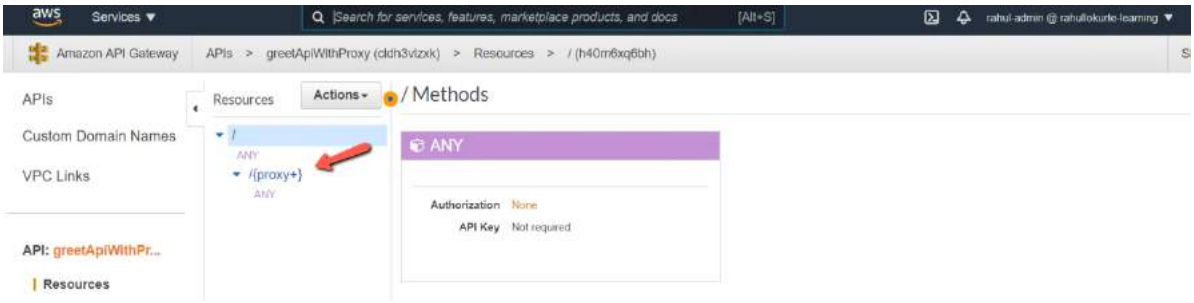
```
// defines an AWS API Gateway resource With Proxy
const apiGatewayWithProxy = new gateway.LambdaRestApi(
  this,
  "greetApiWithProxy",
  {
    handler: greet,
  }
);
```

The above code defines a Gateway REST API that routes all requests to the specified AWS Lambda function **greet**.

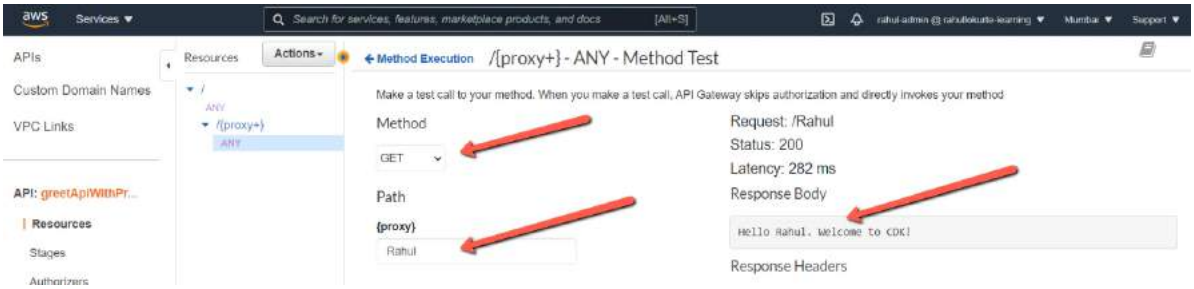
After adding the above code, make sure the resources are as expected by using the command `cdk diff` and once, verified, deploy the stack using `cdk deploy`.

Once deployment is done, go to the AWS console and you see **greetApiWithProxy** API

Now, if you check the Resources, you can see the proxy resource and **ANY** method, which means all requests are routed to AWS Lambda function **greet**.



Click on **ANY** and click on Test and add the below details and you can see the result as shown below.



Lambda Integration Without Proxy

Let us create one more API without proxy integration to lambda. In this, implementation, we need to explicitly define the API model and make **proxy** as **false**.

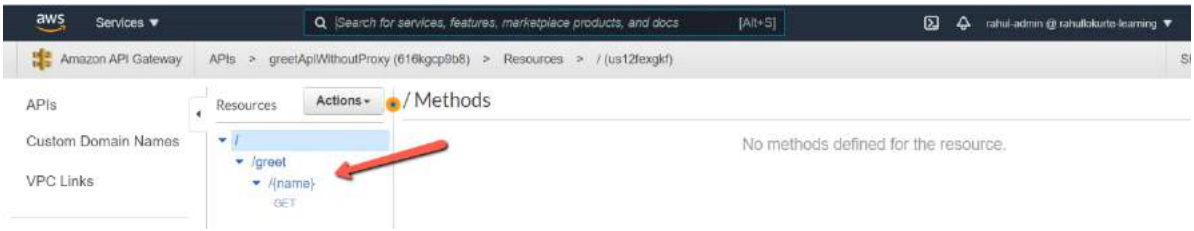
```
// defines an AWS API Gateway resource Without Proxy
const apiGatewayWithoutProxy = new gateway.LambdaRestApi(
  this,
  "greetApiWithoutProxy",
  {
    handler: greet,
    proxy: false,
  }
);

const greetResources =
apiGatewayWithoutProxy.root.addResource("greet");
const greetResource = greetResources.addResource("{name}");
greetResource.addMethod("GET");
```

The above code defines a Gateway REST API that routes the **GET** requests to the resource **/greet/{name}** of AWS Lambda function **greet**.

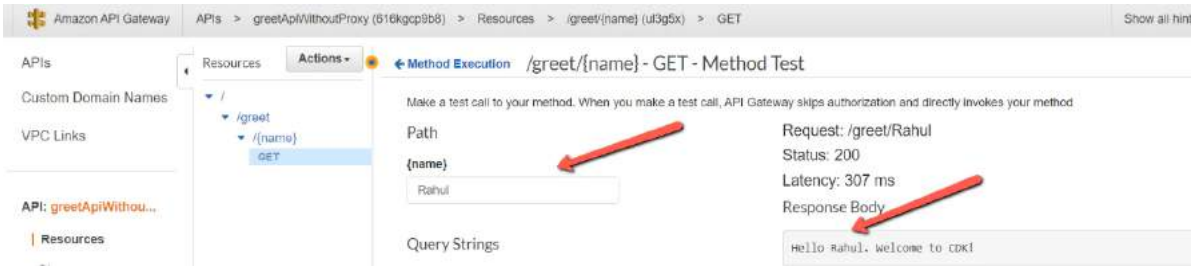
After adding the above code, make sure the resources are as expected by using the command `cdk diff` and once, verified, deploy the stack using `cdk deploy`.

Once deployment is done, go to the AWS console and you see **greetApiWithoutProxy** API

Now, if you check the Resources, you can see the proxy resource **/greet/{name}** and **GET** method.



Click on **GET** and click on Test and add the below details and you can see the result as shown below.



We saw how to create an AWS API Gateway by using AWS Cloud Development Kit. We also saw various ways in which we can integrate the API Gateway with Lambda. We saw how to create a proxy resource and custom resource. The code repository link is **here**

AWS S3 Using CDK

Introduction

Amazon simple storage service (S3) is object storage built to store and retrieve unlimited amounts of data. We can use it to store various kinds of data. We can have data lakes to push the data into S3, we can collect the click events of a web application, we can also use it for data analytics.

Files on S3 are stored in **buckets**, which are unlimited in size and each bucket has a globally unique name.

In this lesson, we're going to create and deploy a new S3 bucket to AWS Using Cloud Development Kit.

Create an S3 bucket

Create a new directory on your system.

```
mkdir cdk-s3 && cd cdk-s3
```

We will use cdk init to create a new JavaScript CDK project:

```
cdk init --language javascript
```

The cdk init command creates a number of files and folders inside the **cdk-s3** directory to help us organize the source code for your AWS CDK app.

We can list the stacks in our app by running the below command. It will show CdkS3Stack.

```
$ cdk ls
CdkS3Stack
```

Let us install AWS S3 construct library and AWS Key Management Service (KMS) to protect the keys.

```
npm install @aws-cdk/aws-s3 @aws-cdk/aws-kms
```

Edit the file lib/cdk-s3-stack.js to create an AWS lambda resource as shown below.

```
const cdk = require("@aws-cdk/core");
const kms = require("@aws-cdk/aws-kms");
const s3 = require("@aws-cdk/aws-s3");

class CdkS3Stack extends cdk.Stack {
  /**
   *
   * @param {cdk.Construct} scope
   * @param {string} id
   * @param {cdk.StackProps=} props
   */
  constructor(scope, id, props) {
    super(scope, id, props);

    // The code that defines your stack goes here
    const myKmsKey = new kms.Key(this, "MyKey", {
      enableKeyRotation: true,
    });

    const myBucket = new s3.Bucket(this, "MyBucket", {
      bucketName: cdk.PhysicalName.GENERATE_IF_NEEDED,
      encryption: s3.BucketEncryption.KMS,
      encryptionKey: myKmsKey,
      versioned: true,
    });
  }
}

module.exports = { CdkS3Stack };
```

- The Bucket name will be Generated one.
- We are using KMS encryption and the Key which we have defined **myKmsKey** .
- Versioning is set to true.

Deployment

Before deploying the AWS resource, we can take a look at what resources will be getting created by using the below command.

```
cdk diff
```

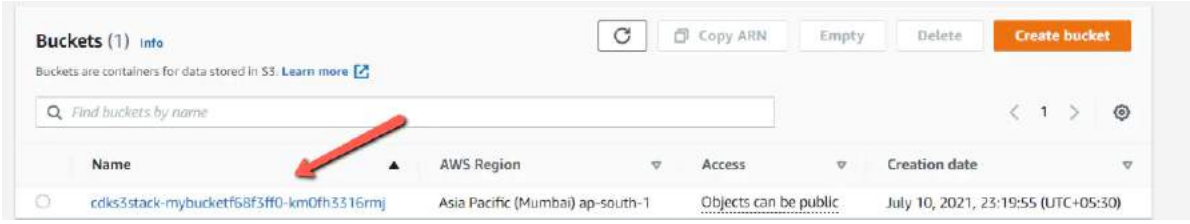
NOTE: If we have multiple profiles set in our system, we need to tell cdk to look into the particular profile. This can be done, by adding the below key-value in ***cdk.json*** which was generated when we created a CDK project.

```
"profile": "<YOUR_PROFILE_NAME>"
```

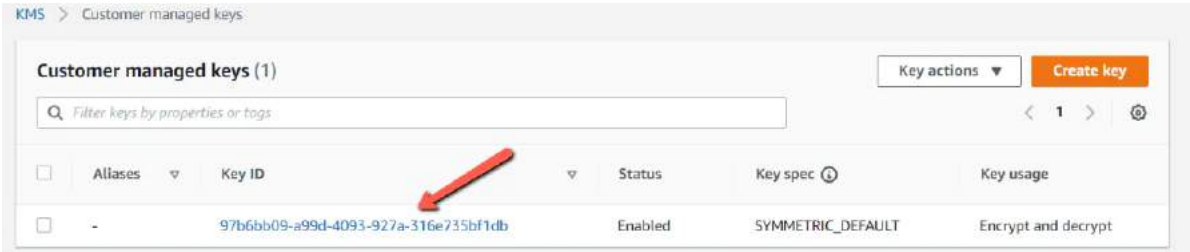
Now, once we are ok with the resources which will be created, we can deploy them using the below command

```
cdk deploy
```

Let us open the AWS Lambda console. We can see that the S3 bucket has been created.



We can also see the KMS which we have created.



We saw how to create an S3 Bucket and also KMS by using AWS Cloud Development Kit. We also saw various commands related to CDK for initiating projects, deploying the resources to AWS. The associated Git Repository is **here**

AWS Step Function Using CDK

Introduction

AWS Step Function service is used to orchestrate a workflow consisting of serverless Functions and integrating with other AWS services to bundle applications.

Create a Step Function

In this lesson, we will look at how to orchestrate lambda functions to State Machine using AWS Step Functions.

Create a new directory on your system.

```
mkdir cdk-step && cd cdk-step
```

We will use CDK init to create a new JavaScript CDK project:

```
cdk init --language javascript
```

The CDK init command creates a number of files and folders inside the **cdk-step** directory to help us organize the source code for your AWS CDK app.

We can list the stacks in our app by running the below command. It will show CdkStepStack.

```
$ cdk ls
CdkStepStack
```

Let us install AWS lambda, AWS step functions and AWS step-functions task dependencies

```
npm install @aws-cdk/aws-lambda @aws-cdk/aws-stepfunctions
@aws-cdk/aws-stepfunctions-tasks
```

Create an AWS Lambda Function **Hello-Function** with inline code:

```
const helloFunction = new lambda.Function(this, "MyLambdaFunction", {
  functionName: "Hello-Function",
  code: lambda.Code.fromInline(`
          exports.handler = (event, context, callback) => {
              callback(null, "Hello World!");
          };
      `),
  runtime: lambda.Runtime.NODEJS_12_X,
  handler: "index.handler",
  timeout: cdk.Duration.seconds(25),
});
```

- The functionName is the name of our lambda function.
- The code is where we put the inline code for our lambda function.
- Nodejs 12.x is used as a runtime.
- The handler will create a file **index.js** and exports the **handler** method.
- Timeout duration will be 25 seconds.

Create the Step definitions:

```
const definition = new tasks.LambdaInvoke(this, "MyLambdaTask", {
  lambdaFunction: helloFunction,
}).next(new sfn.Succeed(this, "GreetedWorld"));
```

- The task is to invoke **Hello-Function** lambda
- The lambda just passes into the Succeed Step which is an inbuilt step
- It returns output as **GreetedWorld**

Let us create a State Machine with AWS Step Function.

```
const stateMachine = new sfn.StateMachine(this, "MyStateMachine", {
  stateMachineName: "Hello-World-Step",
  definition,
});
```

Deployment

Before deploying the AWS resource, we can take a look at what resources will be getting created by using the below command.

```
cdk diff
```

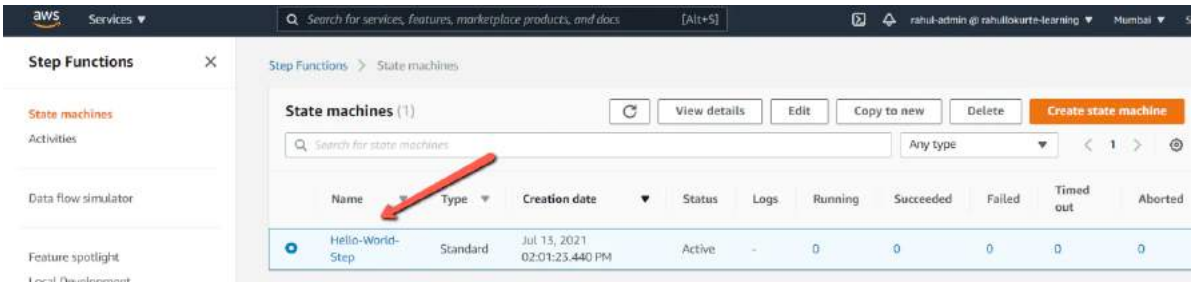
NOTE: If we have multiple profiles set in our system, we need to tell CDK to look into the particular profile. This can be done, by adding the below key-value in **cdk.json** which was generated when we created a CDK project.

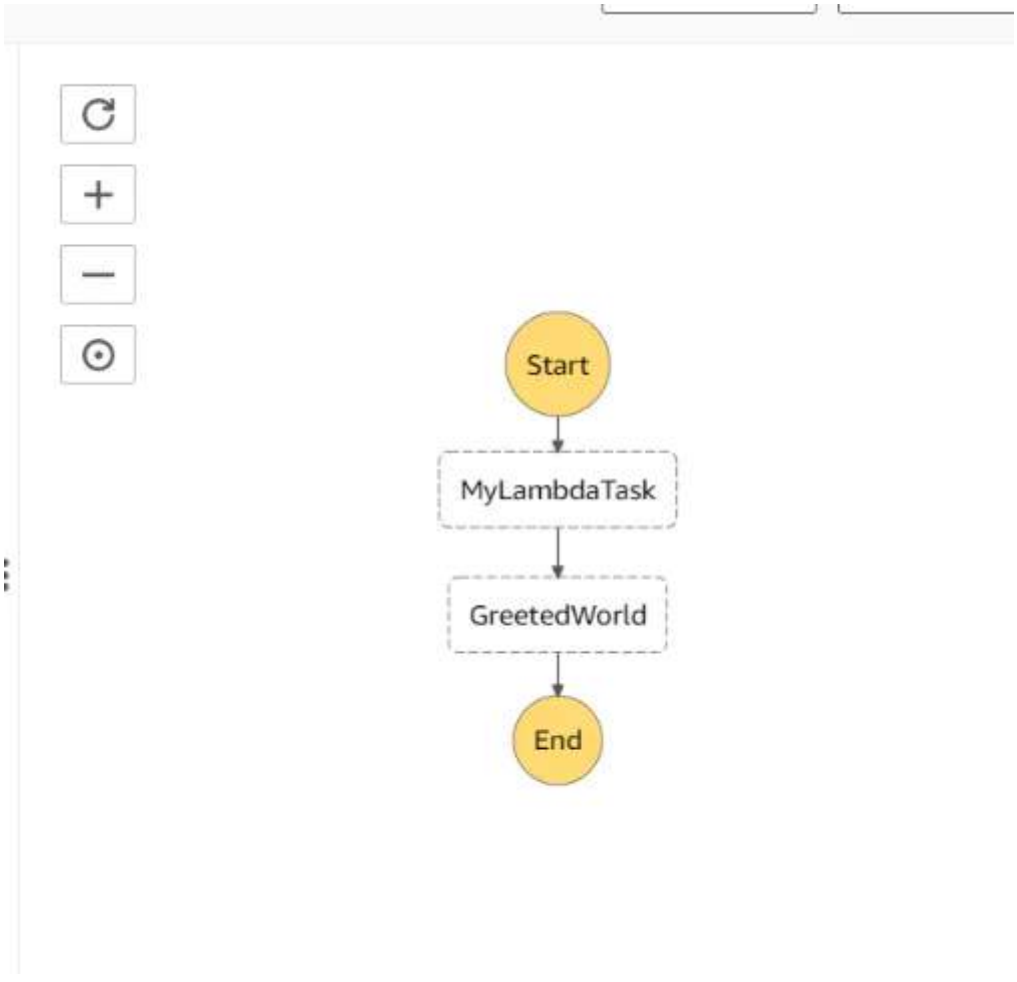
```
"profile": "<YOUR_PROFILE_NAME>"
```

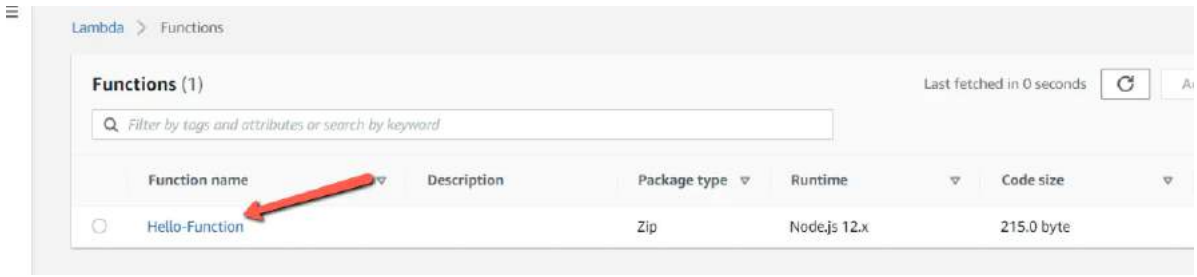
Now, once we are ok with the resources which will be created, we can deploy them using the below command

```
cdk deploy
```

Let us open the AWS Lambda console. We can see that State Machine, Step Function, and Lambda has been created.







Now Go to the Step Function **Hello-World-Step** and click on **Start execution** button. We can see the execution details of the Step Function.

We saw how to create a State Machine and Step Functions using AWS Cloud Development Kit. We also saw various commands related to CDK for initiating projects, deploying the resources to AWS. The associated Git Repository is **here**

AWS Code Artifact Using CDK

Introduction

The software packages need to be stored in a central repository as artifacts. The artifacts need to be stored securely. We should be able to publish new artifacts and also needs to share the artifacts across the organization. Amazon provides a managed repository service called **AWS CodeArtifact**. It works with commonly used package managers and with build tools like Maven, Gradle, npm, pip.

With CodeArtifact there is no software to update or servers to manage. It is completely a part of a serverless stack. We can set up central repositories that make it easy for development teams to find and use the software packages they need. We can control access to the artifacts and validate the software packages. We can do versioning and teams can fetch the latest artifacts from the CodeArtifact.

In this lesson, we will see, how to create different components of CodeArtifact using Cloud Development Kit (CDK).

CodeArtifact stores software packages in repositories. Every CodeArtifact repository is a member of a single CodeArtifact domain. We can use one domain for our organization with one or more repositories. Each repository might be used for a different development team. Packages in your repositories can then be discovered and shared across your development teams.

Create an CodeArtifact

Create a new directory on your system.

```
mkdir cdk-codeartifact && cd cdk-codeartifact
```

We will use cdk init to create a new JavaScript CDK project:

```
cdk init --language javascript
```

The cdk init command creates a number of files and folders inside the cdk-codeartifact directory to help us organize the source code for your AWS CDK app.

We can list the stacks in our app by running the below command. It will show CdkCodeartifactStack.

```
$ cdk ls
CdkCodeartifactStack
```

Let us install the AWS Codeartifact construct library.

```
npm install @aws-cdk/aws-codeartifact
```

Edit the file **lib\cdk-codeartifact-stack.js** to create an AWS Code artifact resource as shown below.

```
const cdk = require("@aws-cdk/core");
const codeartifact = require("@aws-cdk/aws-codeartifact");

class CdkCodeartifactStack extends cdk.Stack {
  /**
   *
   * @param {cdk.Construct} scope
   * @param {string} id
   * @param {cdk.StackProps=} props
   */
  constructor(scope, id) {
    super(scope, id);

    // Create a Domain
    const domain = new codeartifact.CfnDomain(this, "CodeArtifactDomain", {
      domainName: "sample-domain",
    });

    // Create a public repository
    const publicSampleRepo = new codeartifact.CfnRepository(
      this,
      "PublicSampleRepository",
      {
        repositoryName: "public-npm-store",
        externalConnections: ["public:npmjs"],
        domainName: domain.domainName,
      }
```

```
    );

    publicSampleRepo.addDependsOn(domain);

    // Create a custom repository
    const customSampleRepo = new codeartifact.CfnRepository(
      this,
      "CustomSampleRepository",
      {
        repositoryName: "custom-repository-store",
        upstreams: [publicSampleRepo.repositoryName],
        domainName: domain.domainName,
      }
    );
    customSampleRepo.addDependsOn(publicSampleRepo);
  }
}

module.exports = { CdkCodeartifactStack };
```

- Create a Domain **sample-domain**.
- Create a public repository **public-npm-store** and connect it with an external public npm registry.
- The public repository is dependent on the domain which is achieved by using methods **addDependsOn**.
- Create one more custom repository **custom-repository-store** and it can have the same dependency as a public repository by using the upstream property which points to the public repository.

Deployment

Before deploying the AWS resource, we can take a look at what resources will be getting created by using the below command.

```
cdk diff
```

```
Resources
[+] AWS::CodeArtifact::Domain CodeArtifactDomain CodeArtifactDomain
[+] AWS::CodeArtifact::Repository PublicSampleRepository PublicSampleRepository
[+] AWS::CodeArtifact::Repository CustomSampleRepository CustomSampleRepository
```

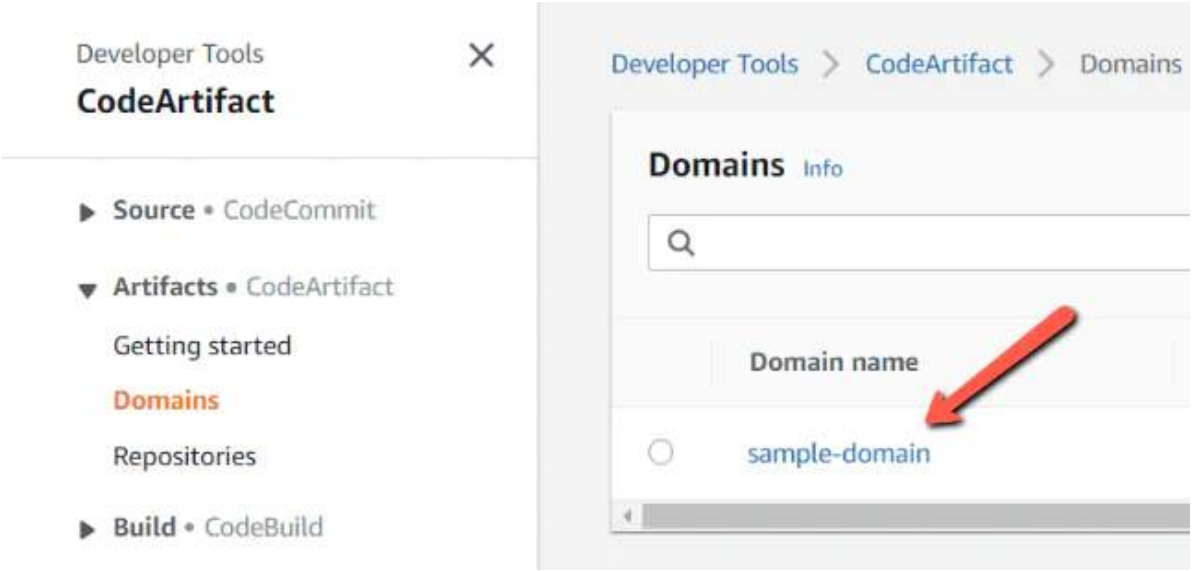
NOTE: If we have multiple profiles set in our system, we need to tell CDK to look into the particular profile. This can be done, by adding the below key-value in cdk.json which was generated when we created a CDK project.

```
"profile": "<YOUR_PROFILE_NAME>"
```

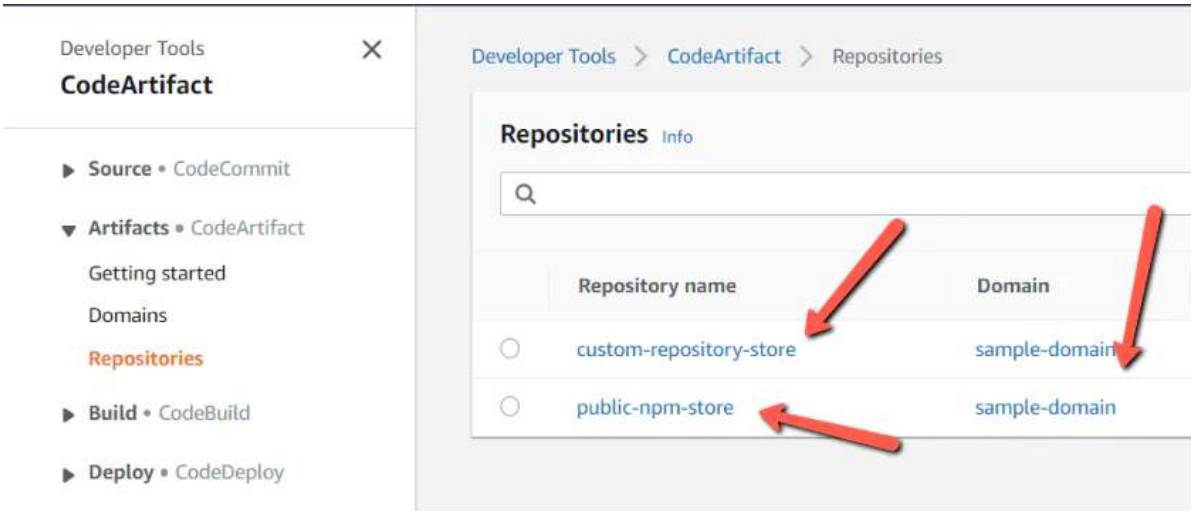
Now, once we are ok with the resources which will be created, we can deploy them using the below command

```
cdk deploy
```

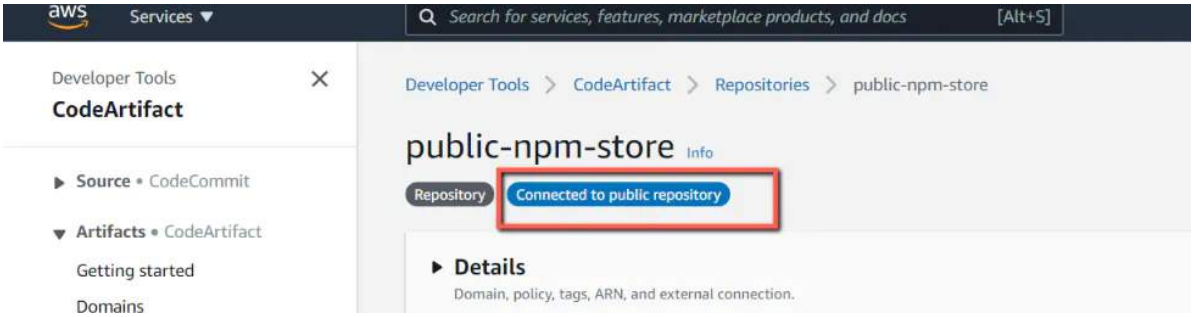
Let us open the AWS Lambda console. We can see that the **domain** has been created.



Also, we can see two repositories are created and are connected to one domain.



We can also see that our **public-npm-store** repository is connected to the public npm repository.



We used the Aws managed CodeArtifact service. We saw how to create a domain and the repositories inside the domain. The associated Git Repository [here]