WA3045 Using TypeScript to Manipulate AWS CDK

Web Age Solutions Inc.
USA: 1-877-517-6540
Canada: 1-877-812-8887
Web: http://www.webagesolutions.com

For customizations of this book or other sales inquiries, please contact us at:

USA: 1-877-517-6540, email: getinfousa@webagesolutions.com
Canada: 1-877-812-8887 toll free, email: getinfo@webagesolutions.com

Web Age Solutions Inc.
821A Bloor Street West
Toronto
Ontario, M6G 1M1

Table of Contents

Chapter 1 - Introduction to the AWS CDK

Objectives

Key objectives of this chapter

- Cloud Resources
- Provisioning Cloud Applications
- CDK Workflow
- AWS CloudFormation
- Stacks
- AWS Constructs
- Construct Library
- CDK CLI commands
- cdk bootstrap command
- TypeScript Compilation
- cdk synth command
- cdk deploy command

1.1 AWS Cloud Development Kit

- Also known as: AWS CDK
- Open source software development framework
- Used to define/provision AWS cloud application resources
- Provides high-level components with pre-configured defaults
- Use programming language to define infrastructure
- Supports repeatability when setting up resources

Notes

AWS - Amazon Web Services

https://aws.amazon.com/cdk/

https://docs.aws.amazon.com/cdk/latest/guide/getting_started.html

1.2 AWS Cloud Resources

- AWS includes over 175 cloud based services/resources
- Some of the more common ones include:
  - S3 - Storage
  - EC2 - Cloud Servers
  - Athena - Serverless SQL Execution
  - Lambda - Serverless Code Execution
  - DynamoDB - Database
  - etc.

1.3 Using AWS Cloud Resources

- AWS cloud resources can be configured and provisioned using:
  - Manually via:
    - AWS management console (web application)
    - AWS command line interface (AWS CLI)
  - Programatically via:
    - AWS cloud development kit (CDK)
    - AWS CLI scripting

- This course discusses the AWS CDK

1.4 Provisioning Cloud Applications

- Provisioning involves configuration and deployment:
- Configuring resources
  - Setting up essential properties
  - Examples
    - Setting an S3 bucket name
    - Defining security parameters for the bucket
- Deploying resources
  - Instantiating resources in a given AWS account
  - Examples:
    - Creating an S3 bucket
      - Uploading files to the bucket

Notes

1.5 Automation

- There are many situations where an automated solution is preferred to setting up resources manually in an AWS console:
  - When onboarding multiple users to AWS
  - When reproducing cloud environments
    - Creating testing or development environments that matches the production environment
    - Ensures uniform configuration when reproducing environment stacks
  - When operations need to be executed repeatedly:
    - Running weekly/monthly analytics or reports
    - Updating applications on a regular basis

- Moving logs from production servers to aggregation environments

1.6 CDK Workflow

- Using the CDK to deploy resources involves these steps:
  - Create a CDK 'app'
  - Use programming language to model your infrastructure using 'Constructs' that map to AWS resources like S3 Buckets, Lambda Functions, etc.
  - Compile to an AWS CloudFormation Stack
  - Deploy the Stack to an AWS account

Notes

Supported programming languages include:

TypeScript, JavaScript, Python, Java and C#

1.7 AWS Cloud Formation Stacks

- Stacks created by the CDK are compatible with those created in AWS Cloud Formation
- AWS Cloud Formation has its own console where Stacks can be created and deployed manually
- Stacks deployed manually show up in the Cloud Formation console.
- Stacks deployed using the CDK show up in the CloudFormation console as well.
- Updating a deployed stack results in changes being applied to existing deployed resources.
- Destroying a stack deletes its deployed resources(*).

Notes

https://aws.amazon.com/cloudformation/

(*) If needed, some resources such as S3 buckets and the data they hold may be set up in such a way that they are retained and not deleted when the stack is deleted.

1.8 Constructs

- Stacks consist of various Constructs each representing different cloud components(*)
- Constructs:
  - Include all configuration needed to create the component
  - Are available for every AWS resource
  - Can be added to stacks manually (in console)
  - Can be added to stacks programmatically (via CDK)

- Constructs are documented in the
  - AWS Construct Library API Reference
  - AWS resource and property types reference

Notes

(*) Some constructs represent a group of related resources that together support a specific funciton (patterns).

AWS Construct Library API Reference

https://docs.aws.amazon.com/cdk/api/latest/docs/aws-construct-library.html

AWS resource and property types reference:

https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-template-resource-type-ref.html

1.9 AWS Construct Library

- Is a library of types and functions for creating and deploying stacks
- Is implemented in various programming languages
  - Python, TypeScript, Java, .Net
- Includes various modules:
  - `aws-{service-name}` Supporting specific AWS services
  - `aws-{service-name}-targets` Used to connect AWS services
  - {other-packages} Supporting CDK operations
- Examples include (in TypeScript):
  - @aws-cdk/**aws-s3**
  - @aws-cdk/**aws-glue**
  - @aws-cdk/**aws-athena**
  - @aws-cdk/**aws-lambda**
  - @aws-cdk/**core**
  - @aws-cdk/**aws-events-targets**

Notes

https://docs.aws.amazon.com/cdk/api/latest/docs/aws-construct-library.html

1.10 Construct Types

- Level 1 (L1) AWS CloudFormation resource types
  - AKA CFN Resources
  - Name Prefix: `Cfn`
  - Example: `CfnBucket`
  - All properties must be specified by developer
- Level 2 (L2) Curated resource types

- Example: `s3.Bucket`
- Practical defaults, boilerplate and glue logic are supplied

- Level 3 (L3) Patterns
  - Multi-resource Constructs for specific use-cases
  - Example: `aws-apigateway.LambdaRestApi`

Notes

The example pattern construct "`aws-apigateway.LambdaRestApi`" represents a lamba function that can be invoked via a REST endpoint.

1.11 The AWS Cloud Development Kit (CDK)

- An installation of Node is required by the CDK
- The CDK is installed via node package manager (npm)

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

- This command installs the CDK globally so that it can be used from any project directory
- The CDK is used by calling its command line interface CLI functions at a command or terminal prompt

1.12 CDK CLI Commands

CDK CLI commands include:

`cdk --help` Displays command line help text

`cdk bootstrap` Deploys the CDK toolkit stack into an AWS account

`cdk init` `{template}` Creates a new CDK app project

`cdk synthesize` Creates a template from the current project

`cdk deploy` Deploys stack template(s) into an AWS account

`cdk destroy` Removes stacks from an AWS account

1.13 CDK bootstrap

- This command prepares an AWS account to be used with the CDK

```
cdk bootstrap
```

- This command is run once to provision resources used by the CDK to perform deployments
- Bootstrapping creates a stack named CDKToolkit in CloudFormation
- The stack includes a bucket, for example:

```
cdktoolkit-stagingbucket-7klfajiweruj
```

- The stack also includes a buket policy, for example:

```
CDKToolkit-StagingBucketPolicy-16ALSJFD
```

- You will not work with these resources directly, they are only to be used by the CDK toolkit itself

Notes

https://docs.aws.amazon.com/cdk/latest/guide/bootstrapping.html

1.14 CDK init Command

- The CDK init command creates a new CDK application project

```
mkdir my-cdk-app
cd my-cdk-app
cdk init app --language typescript
```

- The second parameter 'app' is the name of the template to use when creating the project. A list of available templates can be displayed:

- **`cdk init --list`** Shows available templates

- The resulting project has this basic structure:

```
\my-cdk-app
    \bin
    \lib
    \node_modules
    \test
    cdk.json
    package.json
```

- The code you develop goes in the "\lib" directory
- When developing with TypeScript the following command is used to invoke the TypeScript compiler:

```
npm run build (or npm run watch)
```

Notes

1.15 Compatible Programming Languages

- As can be see from the 'cdk init' command '--language' property, the CDK can initialize projects in a variety of programming languages.
- Language choices include:
  - TypeScript
  - JavaScript
  - Python
  - Java
  - C#
- CDK application projects can be created in any of these languages
- The AWS Construct Library is available in these languages as well

1.16 CDK Code

- Example TypeScript code for creating an S3 Bucket

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

export class HelloCdkStack extends cdk.Stack {
  constructor(scope: cdk.App,
              id: string, props?: cdk.StackProps) {

    super(scope, id, props);

    new s3.Bucket(this, 'MyFirstBucket', {
      versioned: true
      ,removalPolicy: cdk.RemovalPolicy.DESTROY
    });
  }
}
```

- Filename: `\lib\hello-cdk-stack.ts`
- Note the module imports on the first two lines
- This code should be compiled to JavaScript before running the "cdk synthesize" or "cdk deploy" commands

Notes

1.17 TypeScript Compilation

- TypeScript needs to be compiled (*) before running the 'cdk synth' or 'cdk deploy' commands.
- Compilation is done using the TypeScript compiler 'tsc' which is invoked in a CDK project with the one of the following commands:

```
npm run build
```

```
npm run watch
```

- The second form of the command is run in its own terminal where it stays active and re-runs the compilation whenever one of the TypeScript source code files is saved.

Notes

(*) Although the process applied to typescript code to convert it into JavaScript code is called 'Transpilation' it is often referred to as 'Compilation' instead.

'Transpilation' converts source code in one language to source code in another language.

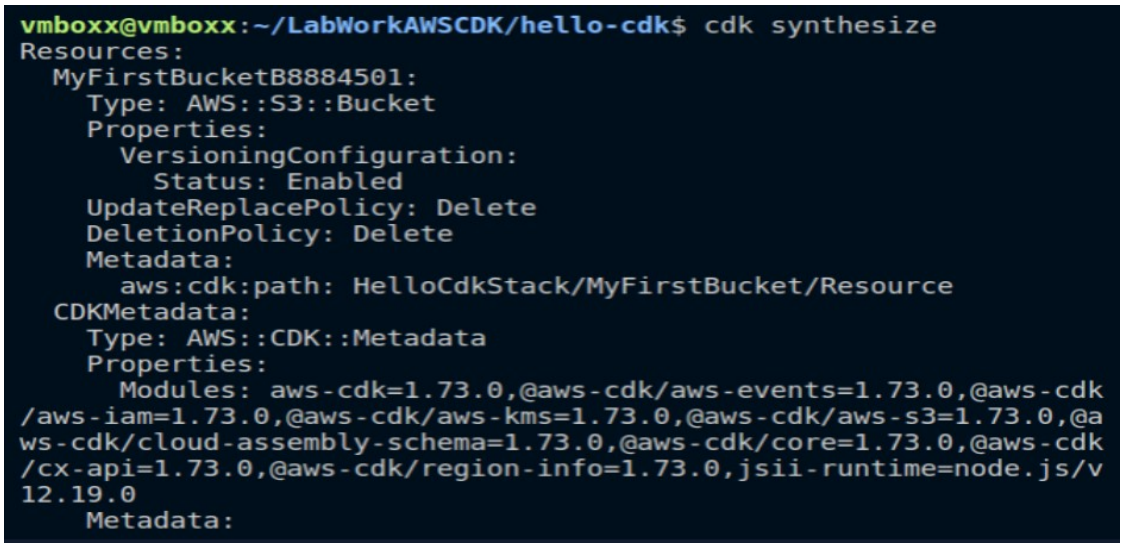
'Compilation' converts source code into another form such as bytecode or machine code.

1.18 CDK Synth Command

- The synthesize (synth for short) command is run after the TypeScript compiler in order to convert the code you've written into a CloudFormation template that can be used to create a Stack when applied to your AWS account.

```
cdk synthesize (or cdk synth)
```

- The command is run from within your CDK project directory
- Output like this shows in the console after running the command:



Notes

1.19 CDK Deploy Command

- The CDK deploy command:
  - Checks for changes since the last 'cdk synth' command
  - If changes are detected it runs 'cdk synth' first
  - Then it applies the created CloudFormation template to your AWS account
    - A new stack is created in CloudFormation
    - The resources defined by the stack are created in the account
- A 'cdk destroy' command is available to remove the stack and its resources
  - non-empty S3 buckets will not be deleted unless they were created with the removal policy = cdk.RemovalPolicy.DESTROY

Notes

1.20 Summary

In this chapter we covered:

- Cloud Resources
- Provisioning Cloud Applications
- CDK Workflow
- AWS CloudFormation

- Stacks
- AWS Constructs
- Construct Library
- CDK CLI commands
- cdk bootstrap command
- TypeScript Compilation
- cdk synth command
- cdk deploy command

Chapter 2 - AWS CloudFormation and Stacks

Objectives
Key objectives of this chapter ■ What is CloudFormation? ■ CloudFormation Web Console ■ Templates ■ Template Designer ■ Sample Templates ■ Resource Stacks ■ Resource Types ■ Deploying Stacks ■ Updating Stacks ■ Deleting Stacks

2.1 What is CloudFormation?

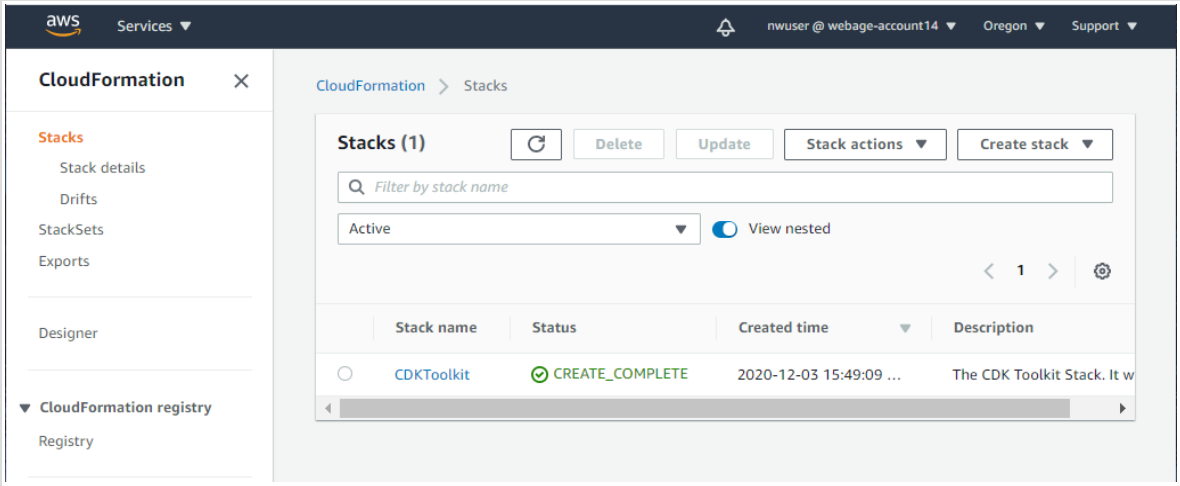
- CloudFormation:
  - Is an Amazon AWS service
  - Is free to use
  - Manages individual resources or sets of resources as Stacks
  - Resource stacks are based on JSON/YAML templates
  - Stacks of resources can be created, updated, or destroyed using simple commands.

Notes

aws - amazon web services

2.2 CloudFormation WebConsole

- Templates and Stacks can be created in the Web Console:



2.3 Templates

- Templates provide a blueprint for AWS resource deployment
- Two formats are supported for templates:
  - JSON
  - YAML
- Templates are required for creating stacks
- Templates can be created using:
  - Text editor
  - CloudFormation Designer

2.4 CloudFormation Designer

- CloudFormation Designer is a cloud based graphic and text editing environment for creation/modification of resource templates
- Resources can be added to templates using a drag and drop interface.

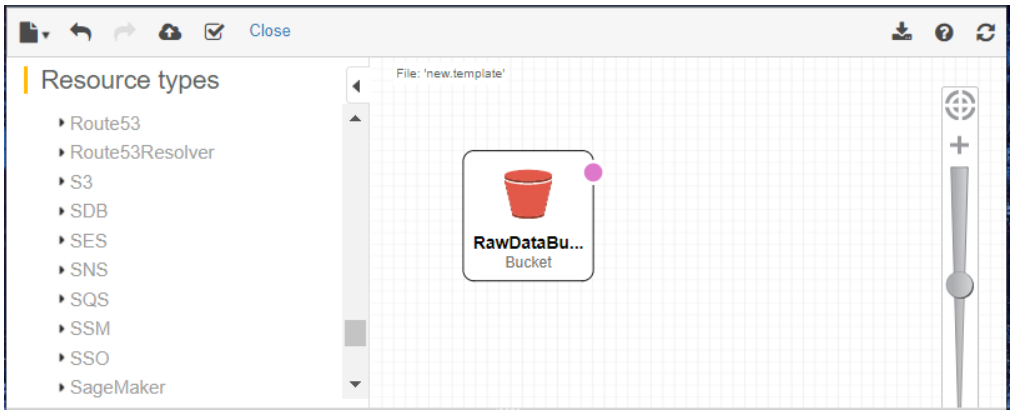
- Relationships between resources can be seen in the graphic editor
- Resource properties are configured in the JSON/YAML editor area of designer
- Designer can be found in the CloudFormation web console
- Errors when defining resources or their connections are reduced by Designer's built-in validation
- Templates can be opened and saved in S3 or in your local file system.
- Templates can also be converted between formats (JSON/YAML) within Designer.

Notes

https://console.aws.amazon.com/cloudformation/designer

2.5 The Designer Resource GUI

- Designer's Graphic Interface for Managing Resources



2.6 The Designer Template Editor

- Designer's Template Editor

2.7 Sample Templates

- Sample templates are available from the AWS CloudFormation sample template library

  https://aws.amazon.com/cloudformation/resources/templates/

- Library templates are organized by:
  - AWS Service (EC2, S3, RedShift, etc.)
  - Application Framework (LAMP, Ruby on Rails, etc.)
  - Reference Implementations (Win Server, SAP, etc.)
  - Sample Solutions (3rd party end-to-end solution templates)

Notes

2.8 Resource Stacks

- Stacks are collections of AWS Resources that are created by deploying templates to an AWS account.
- Templates can be deployed:
  - Manually in the CloudFormation console

- Programmatically using the AWS CDK

- Once a stack is created it can be:
  - Updated to modify or add/remove resources
  - Deleted to remove all its resources

- The user initiating a deployment needs to have sufficient permissions to deploy and manage the templated resources

Notes

2.9 Resource Types

- The "Resources" section of a template includes sub-sections for each added resource:

```
"Resources": {
  "RawDataBucket": {
     "Type": "AWS::S3::Bucket",
     "Properties": {
        "AccessControl": "Private"
      },
     "Metadata": {
       "AWS::CloudFormation::Designer": {
         "id": "fa85faa8-d989-48a1-a568-3428d870d82c"
       }
     }
  },
  ...
}
```

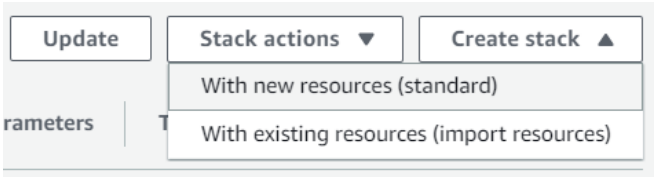
- These sections are added automatically by the template designer each time you drag a resource to the canvas area
- The sections are customized by the template developer who will rename the resource and add any required properties:

Notes

2.10 Deploying Stacks

Stacks are deployed in two ways:

- **With new resources (standard)**
  - Specify an S3 URL for the template
  - Upload a template file
- **With existing resources (import resources)**



Notes

2.11 Updating Stacks

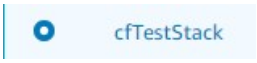
- Updates to a stack's configuration or resources can be made after its been created.
- There are two ways to update a stack
  - Direct update - requested a change for immediate deployment
  - Using 'change sets' - changes can be 'previewed' before being applied
- Before deploying changes CloudFormation determines the differences between the new configuration and the existing one and chooses an update method that will minimize disruptions in service:
  - Update with no Interruption
  - Update with Some Interruption

- Replacement
-

Notes

https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/using-cfn-updating-stacks.html

2.12 Updating a Stack Directly

- To update after creating and deploying a stack
  - Make changes to the stack's template
  - Go to the CloudFormation console
  - Select the stack you to update 

  - Click the "Update" button 

  - Supply the changed template
  - OK the changes
  - Execute the update

Notes

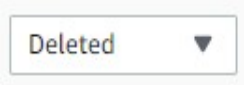
https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/updating.stacks.walkthrough.html

2.13 Updating Using Change Sets

- Procedure for updating using change sets
  - Edit and save the Stack's template
  - Generate a change set from the updated template
  - View the change set to see how the changes will be deployed
  - Execute the change set to update the stack

2.14 Deleting Stacks

- Stacks with the following statuses can be deleted from the CloudFormation console
  - CREATE_COMPLETE
  - ROLLBACK_COMPLETE
- The delete process cannot be aborted once it has started
- Deletion removes all resources defined in the stack from the AWS account(*) and sets the stack's status to DELETE_COMPLETE
- CloudFormation does not show Deleted stacks by default. To access them once they've been deleted you need to change the filter drop down to "Deleted".



Notes

(*) Rules exist that limit the removal of certain types of resources based on their current state. For example: removing an S3 Bucket when it is not empty.

Deleting a Stack:

https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/cfn-console-delete-stack.html

2.15 DELETE_FAILED when Deleting a Stack

- Certain resources, for example an S3 bucket, must be empty before they can be deleted. If your stack includes a Bucket then deleting the stack will normally result in DELETE_FAILED

- This behavior allows you to delete the rest of a stack's resources without disturbing files that have been uploaded
- For this use-case, when you want to delete a stack but keep the bucket and its contents, you should choose to **"Retain Resources"** when deleting the stack. This will allow you to delete the stack cleanly and results in the status DELETE_COMPLETE.
- Alternately you can delete the stack cleanly by emptying the bucket first before running the delete

Notes

A bucket's contents can be deleted in the S3 console. It can also be deleted using an AWS CLI command:

```
aws s3 rm s3://bucket-name --recursive
```

Troubleshooting DELETE_FAILED

https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/troubleshooting.html#troubleshooting-errors-delete-stack-fails

2.16 Summary

In this chapter we covered:

- What is CloudFormation?
- CloudFormation Web Console
- Templates
- Template Designer
- Sample Templates
- Resource Stacks
- Resource Types
- Deploying Stacks
- Updating Stacks

- Deleting Stacks

Chapter 3 - TypeScript Basics

Objectives
Key objectives of this chapter ◇ What is TypeScript? ◇ Setup for TypeScript ◇ TypeScript Features ◇ Defining Code with Types ◇ Type Inference ◇ Classes ◇ Arrow Functions ◇ let and const variables ◇ Template Strings ◇ Code Modules ◇ Export and Import Statements

3.1 What is TypeScript

- Open Source Scripting language that extends JavaScript
- Supported by Microsoft
- Supports syntax from latest versions of JavaScript
- Adds Types
- Helps you catch type related bugs during development
- Transpiles to standard JavaScript that can be run on browsers and with node

Notes

Transpilation refers to translation from one source code version to another form of source code. The output is still source code.

Compilation refers to conversion of a program so something other than source code such as bytecode or

machine code.

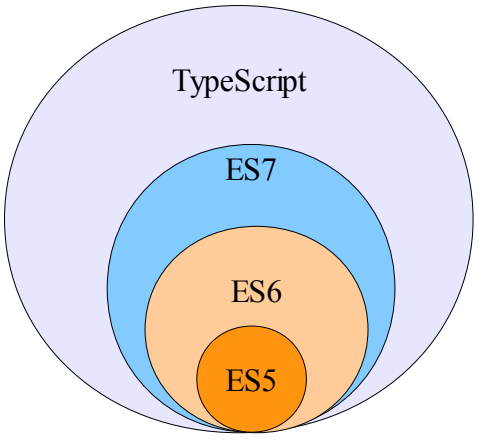
3.2 TypeScript vs. JavaScript

JavaScript - ES5

- **Widely supported by browsers**
- No type support

JavaScript - ES6

- Adds full object orientation
- Still no type support
- Not yet supported by all browsers



TypeScript

- Super-set of standard JavaScript
- Is "transpiled" into JavaScript(ES5)
- Includes full type support
- Includes full support for Object Orientation

Notes

Although JavaScript has been evolving even the latest version, ES6, does not include some features you might expect such as strongly typed variables and full object orientation.

Typescript is a strongly typed language that, due to its inclusion of types, catches many coding errors at compile time thus reducing defects. TypeScript code is compiled or converted into JavaScript code in a process called 'transpilation'.

3.3 Benefits of TypeScript

TypeScript:

- Leverages existing JavaScript programming skills
- All JavaScript code is valid TypeScript code
- Allows (but does not require) use of types
- Can generate type-specific error messages when types are used

- Supports standard 'Class' based syntax for object orientation
- Supports advanced JavaScript features such as:
  - arrow functions,
  - template strings,
  - let and const,
  - module import and export

Notes

3.4 TypeScript Support

- Support for TypeScript is often included during project creation for technologies such as: Angular, React, AWS CDK, etc...
- package.json
  - devDependencies

```
"typescript": "^4.1.2"
```

  - Scripts

```
"scripts": {
  "build": "tsc",
  "watch": "tsc --watch"
}
```

- Scripts are run using the node package manager (npm):

```
npm run build
npm run watch
```

- In these cases projects can be set to automatically transpiled TypeScript code whenever source code changes are saved (tsc --watch).

- Projects often include a TypeScript configuration file "tsconfig.json" specifying various compiler options.

3.5 Setting up a Standalone TypeScript Development Environment

- A standalone TypeScript development environment can be setup like this:
  - Install Node
  - Create a project: "npm init"
  - Install TypeScript: "npm install typescript --save-dev"
  - Install Typings: npm install @types/node --save-dev
  - Create some code: myapp.ts
  - Transpile the code: npx tsc myapp.ts
  - Run the code: node ./myapp.ts

3.6 TypeScript Features

- We will be taking a look at the following TypeScript Features:
  - Adding Types to JavaScript Code
  - Working with Classes
  - Arrow Functions
  - Let & Const variables
  - Template strings
  - Code Modules
  - Imports and Exports

3.7 The Type System – Defining Variables

- Standard JavaScript variables are untyped:

```
let x = 'stuff';     // string data
let y = true         // boolean data
var z = 33           // numeric data

x = y  // OK. Type switching allowed.
```

- TypeScript variables are strongly typed:

```
let x: string = 'stuff';   // string data
var y: boolean = true      // boolean data
let z: number = 33         // numeric data
let a : any = 'hello' //any type

x = y  // Compile error
let s : string = 25 //Compile error

a = y //OK
```

Notes

Variables in JavaScript can hold any type of data. This allows you to get going quickly but becomes a problem when you assign values of the wrong type to a variable, when you pass a variable of the wrong type to a function that requires a specific type or when you try to process an array with elements of various types.

TypeScript requires you to specify the variable type in the variable declaration:

```
var x: string = 'stuff';     //a string type
```

This line, which is fine in JavaScript, will produce an error at compile time in TypeScript:

```
var x: string = 25;          // assigns a number to a string var
```

If you do need to create a variable that holds various types of data you can do that using the 'any' type:

```
var a: any = 'stuff';        // any data
a = 35;                      // this works
```

3.8 The Type System – Defining Arrays

- Arrays in standard JavaScript

```
let colors = ['red', 'white', 'blue'];
let my_nums = [10, 20, 30];
let people = [{name:'John'},{name:'Lisa'}];
```

- Arrays in TypeScript

```
let colors: string[] = ['red', 'white', 'blue'];
let my_nums:number[] = [10, 20, 30];
let names: Object[] = [{name:'John'},{name:'Lisa'}];
```

Notes:

Arrays can have types too. Setting the type of an array restricts the kind of data that can be added to it. For example the following array can only hold numbers:

```
var my_nums: number[] = [10,20,30];
```

This array can hold various types of objects:

```
var names: Object[];
```

This array can only hold Person type objects:

```
var people: Person[];
```

We will take a closer look at Classes and Objects shortly.

3.9 Type in Functions

- Parameter and return types need to be specified.

```
function sayHello(name: string) : string {
    return `Hello ${name}!`
}

var h1 : string = sayHello("Daffy Duck") //OK
var h2 : string = sayHello(10) //Compile error
var h3 : number = sayHello("Daffy Duck") //Compile error
```

3.10 Type Inference

- The compiler can infer types of variables from initial value assignment.

```
var x = "hello" //x is a string

x = 10 //Compile error.

var a //Uninitialized variable inferred as any type

a = "hello" //OK
a = 10 //Also OK
```

- Function return type can be inferred.

```
function sayHello(name: string) {
    return `Hello ${name}!`
}

var h3 : number = sayHello("Daffy Duck") //Compile error
```

- Explicitly specify types for better readability of your code.

3.11 Defining Classes

- Classes define a custom datatype:

```
class Cat {
    name: string = ""
    breed: string = ""
}

function meow(cat: Cat) {
    console.log(`${cat.name} says meow!`)
}

let c = new Cat

c.name = "Fluffy"
c.breed = "Persian"

meow(c)
```

3.12 Class Methods

- Does not use the **function** keyword.
- Must access properties using the **this** keyword.

```
class Cat {
  name: string = ""
  breed: string = ""

  meow() {
    console.log(`${this.name} says meow!`)
  }
}

let c = new Cat

c.name = "Fluffy"
c.breed = "Persian"

c.meow()
```

3.13 Visibility Control

- Class methods and variables can be marked as **public**, **private** and **protected**.

```
class Employee {
    name:string //Public by default
    private salary:number
    private giveRaise() {this.salary += 100.00}
}

let e = new Employee
e.giveRaise() //Compile error!
```

- **public** - By default items are public. They are accessible from outside the class.
- **private** - Items are accessible only from within the class.

- **protected** - Items are accessible from within the class and any other class that extends from it.

3.14 Class Constructors

- TypeScript allows only one constructor implementation.

```
class Building{
    address:string;
    units:number;

    constructor(address: string, units: number){
      this.address = address;
      this.units = units;
    }
}

var bld1 = new Building("1 main street", 4);
var bld2 = new Building("13 park ave.", 5);

var properties: Building[] = [bld1, bld2];
```

3.15 Class Constructors – Alternate Form

- This Class works just like the one on the previous slide:

```
class Building{
    constructor(public address: string,
                public units: number){}
}
var bld1 = new Building("1 main street", 4);
console.log(bld1.address  +", " + bld1.units);
```

- Note that it does not explicitly define properties.
- Adding visibility qualifiers to the inputs of the constructor also defines them as class properties.

```
public address: string,
public units: number
```

Notes

The syntax shown here is commonly used when defining classes.

3.16 Arrow Functions

- Arrow function is a way to define higher order functions that can be passed to other functions as argument and invoked at a later time:

```
let list = [1, 12, 5, 7, 20]
let evenNumbers = list.filter((n: number) : boolean => {
  if (n % 2 == 0) {
    return true
  } else {
    return false
  }
})

console.log(evenNumbers) //Prints [ 12, 20 ]
```

Notes

In the example above we are passing an arrow function to the filter() method. It takes as input a number and returns a boolean value. The filter() methods includes the number if the arrow function returns true.

3.17 Importing and Exporting Code

- We will be covering:
  - Code Modules
  - Importing from library modules
  - Basic Export/Import Syntax
  - Export Syntax Variations (ways to export, things to export)
  - Import Syntax Variations

3.18 Arrow Function Compact Syntax

- Typescript can infer the parameter and return types of arrow functions. This results in less typing.

```
let evenNumbers = list.filter((n) => {
  if (n % 2 == 0) {
    return true
  } else {
    return false
  }
})
```

- If the arrow function has only one statement then the "return" keyword and { } can be omitted.

```
let evenNumbers = list.filter((n) => n % 2 == 0)
```

- If the arrow function takes only one parameter then parenthesis can be omitted.

```
let evenNumbers = list.filter(n => n % 2 == 0)
```

3.19 let and const

- '**let**' and '**const**' provide alternatives to using '**var**' to define variables
- Variables declared with '**let**' differ from ones declared using '**var**' in that:
  - Re-declaring the same variable is not allowed (produces an error)
  - The variable adheres to block scope

```
// redefining a variable
var a = 13;
var a = 24;  // legal, same as 'a = 24'

let b = 13;
```

```
let b = 24;  // TypeError
```

3.20 'var' Variable Scope

- One of the biggest differences between using '**var**' or '**let**/**const**' to declare variables is the scope of the variable
- With '**var**' variables defined in a function always have "function scope" even when defined inside a code block.

// **var** variable with function scope

```
function myFunction() {
  if ( ... ) {
     var a = "some text"; // a defined in code block
     console.log(a);      // a used in code block
  }
  console.log(a);  // a used outside code block
}
```

- Allowing 'a' to be used outside the block it was defined in is something that is not allowed in most programming languages and can cause bugs

3.21 'let' Variable Scope

- With 'let' variables can have either "function" or "block" scope depending on where they were defined:

// **let** variable with block scope

```
function myFunction() {
  if ( ... ) {
     let a = "some text"; // a defined in code block
     console.log(a);      // a used in code block
  }
  console.log(a);  // a will produce error when
```

```
                        // accessed outside the block
}
```

- Restricting the scope of 'a' to block scope matches how it works in most programming languages.

3.22 The 'const' keyword

- The '**const**' keyword declares a variable:
  - The variable must be immediately initialized
  - The variable's value cannot be changed later on

```
// const variables
const foo;        // error, needs a value
const bar = 123; // correct usage
bar = 456;  // error, can't assign new  value here
```

3.23 Template Strings

- Defined using a pair of backticks. Allows expressions to be embedded inside such a string. Example:

```
var name = "Harold" //Regular string
var x = `Hello There ${name}` //Template string
console.log(x) //Prints: Hello There Harold
```

- Template strings can be multi-line:

```
var html = `
April is the cruellest month, breeding
Lilacs out of the dead land, mixing
Memory and desire, stirring
`;
```

3.24 Code Modules

- Modules provide a way to package code that makes it reusable
- Using modules an application can take advantage of code libraries and frameworks written by 3rd parties
- Modules also provide a way to separate an application's own code into separate files for easier development and testing
- TypeScript supports ES6 modules syntax
  - Variables, Classes and Functions are made available via Export statements

```
// module.ts
export function doThis(){ ... }
```

  - Modules and their exported contents are accessed via Import statements

```
// app.ts
import { doThis } from 'some_path/module';
let value = doThis();
```

3.25 Basic Export/Import Syntax

- Module's consist of a file containing classes, functions and variables.
- Items exported from the file can be used in other code files

```
// person.ts
export class Person {
  constructor( public fname,  public lname){}
  display(){console.log(fname + ", " + lname);
}
export var settings = {...}
```

- You need to import items from another module to use them.

```
// app.ts
import {Person} from './person';
var p1 = new Person("Joe", "Smith");
p1.display();
```

Notes

Notice how the term "export" is added before the class definition.

"export" is used to make a class, variable or function visible outside the file where it is defined.

"import" is used to access the exported item from inside a separate file.

Notice how we reference the name of the class we are importing. Also note that the extension is left off of the name of the module file when defining the 'from' clause.

```
'./person'
```

3.26 Export Statements

Valid export statements:

- Export as part of the declaration:

```
export myNum: number = 25;
```

- Export after declaration

```
class Shape { name: string; volume: number }
export { Shape }
```

- Default Export

```
var myUtil = {
  multiply(a: number, b: number) { return a * b; },
  log(msg: string) { console.log('msg: ' + msg) }
}
export default myUtil;
```

- a

Notes

```
Default exports are imported differently than other imports, see the
next slide for details.

// mymodule.ts
var myNum: number = 25;
function makeGreet(name: string) { return 'Hello ' + name }
class Person { name: string; phone: string; }
export var count = 0;
export class Shape { name: string; volume: number }
export function getPi(): number { return 3.1415; }
var myUtil = {
  multiply(a: number, b: number) { return a * b; },
  utillog(msg: string) { console.log('msg: ' + msg) }
}

export default myUtil;
export { myNum, makeGreet, Person };
```

3.27 Import Statements

Valid import statements:

- Import individual exported items:

```
import { myNum, Shape } from './mymodule';
var x = myNum
var box = new Shape();
```

- Import **all** exports at once:

```
import * as mod from './mymodule';
var pi = mod.getPi();
var cnt = mod.count;
```

- Import the **default** export:

```
import utils  from './mymodule';
var z = utils.multiply(3,5);
```

Notes

```
// myapp.ts
import * as mod from './mymodule';
import { Person, getPi as Pi } from './mymodule';
import utils  from './mymodule';

utils.multiply(2,3);
let pi = Pi();
let greet = mod.makeGreet('jack');
let person: Person = {name: 'joe', phone: '212-555-1212'};


// mymodule.ts
var myNum: number = 25;
function makeGreet(name: string) { return 'Hello ' + name }
class Person { name: string; phone: string; }
export var count = 0;
export class Shape { name: string; volume: number }
export function getPi(): number { return 3.1415; }
var myUtil = {
  multiply(a: number, b: number) { return a * b; },
  utillog(msg: string) { console.log('msg: ' + msg) }
}

export default myUtil;
export { myNum, makeGreet, Person };
```

3.28 Programming Editors

- While any text editor can be used to create TypeScript code the following editors include features that can make working with TypeScript easier:

WebStorm	$	Web development IDE from JetBrains. Well regarded. Includes TS support.
Visual Studio Code	Free	Lightweight cross-platform editor from Microsoft that includes TS support.
Sublime Text	$	Programming editor. Supports TS via plugin.
Atom	Free	Open source text editor. TS support via plugin package.

Brackets	Free	Open source code editor. TS support via plugin extension.

Notes

3.29 Summary

In this chapter we covered:

- What is TypeScript?
- Setup for TypeScript
- TypeScript Features
- Defining Code with Types
- Type Inference
- Classes
- Arrow Functions
- let and const variables
- Template Strings
- Code Modules
- Export and Import Statements

Chapter 4 - AWS CDK Setup

Objectives

Key objectives of this chapter

- Setup Overview
- HW/SW Environment
- Account, Permissions, Credentials
- AWS-CLI installation
- aws configure
- AWS-CDK installation
- cdk commands
- Initializing a CDK Project
- cdk bootstrap and CDKToolkit
- Removing CDKToolkit
- AWS Toolkit for Visual Studio Code

4.1 Setup Overview

- Setting up a development environment for the AWS Cloud Development Kit (CDK) involves the installation of:
  - NodeJS, NPM and GIT
  - AWS-CDK

- Using the CDK requires:
  - AWS account login
  - AWS account keyfile

- Not required but recommended:
  - AWS-CLI

- AWS Toolkit for Visual Studio Code

Notes

4.2 HW/SW Environment

- Hardware/Software Environment
- The CDK can be run on all major operating system platforms:
  - Windows
  - Mac
  - Linux
- For installation and application support the CDK:
  - Requires node/node package manager
  - Makes use of GIT
- CDK apps can be programmed in these languages:
  - TypeScript
  - JavaScript
  - Python
  - Java
  - C#

Notes

4.3 Accounts, Credentials & Permissions

- To use the AWS CDK you need either of the following:

- AWS **root user account** email-address/password
- AWS **IAMS user account** username/password/account-id (recommended)

- IAM(Identity and Access Management) user accounts are created under a root user account using the AWS IAMS service.
- While a root account has all permissions the IAMS user only has the permissions it is given and is a better choice for security reasons.

Notes

4.4 AWS-CLI installation

- AWS-CLI is the Amazon Web Services Command Line Interface
- It allows you to perform at a command prompt many of the resource operations available in the AWS Web Consoles such as:
  - Creating/Listing/Updating/Deleting Resources
  - Deploying/Destroying Stacks
  - etc.
- AWS-CLI commands come in handy when working with the CDK where they allow you to easily check the results of your CDK deployments.
- AWS-CLI can be installed on:
  - **Windows:** Download and run the AWSCLIV2.msi installer
  - **Mac:** Download and run the AWSCLIV2.pkg installer
  - **Linux:** Download unzip and run awscliv2.zip
  - **Docker:** Run the "amazon/aws-cli" image

Notes

For more details on aws-cli installation see:

https://docs.aws.amazon.com/cli/latest/userguide/cli-chap-install.html

4.5 aws configure

- Many aws-cli commands require account access to work
- This is a two step process:
  - Create an account keyfile in AWS IAMS
  - Run the "aws configure" command
- The "aws configure" command creates two files - '*credentials*' and 'config' on the development machine to hold the account key information. The file's location is OS specific.
- Both AWS-CLI and AWS-CDK use the account keys in this file to access AWS accounts.

Notes

For more details on configuring aws-cli see:

https://docs.aws.amazon.com/cli/latest/userguide/cli-chap-configure.html

4.6 Some aws-cli Commands

- Some useful aws-cli commands

Command	Description
`aws iam list-users --output table`	list iams users
`aws s3 ls`	list buckets
`aws s3 ls s3://{bucket-name}`	list bucket contents
`aws s3 rb s3://{bucket-name} --force`	remove a bucket including contents
`aws cloudformation delete-stack --stack-name {stack-name}`	delete a stack
`aws s3 sync {local-dir} s3://{bucket-name}`	upload dir of files to a bucket
`aws s3 cp {filename} s3://{bucket-name}`	upload a single file to a bucket

Notes

aws-cli command reference:

https://docs.aws.amazon.com/cli/latest/index.html

4.7 AWS-CDK installation

- AWS-CDK can be installed and its version checked using the following npm commands:

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

- The npm command above installs the aws-cdk to a global location on the development machine so that the 'cdk' command can be accessed from any directory.

4.8 Some cdk Commands

- Some useful cdk commands

Command	Description
`cdk init`	create a cdk project
`npm run watch`	compile CDK TypeScript code
`cdk synth`	synthezise a CF template from CDK project
`cdk deploy`	deploy a CDK project stack
`cdk bootstrap`	prepare an account for CDK deployments
`cdk destroy`	remove a CDK project stack
`cdk diff`	compare CDK project stack with deployed stack

Notes

CF: CloudFormation

aws-cdk toolkit reference:

https://docs.aws.amazon.com/cdk/latest/guide/cli.html

4.9 Initializing a CDK Project

- A CDK application project is initialized using the following command:
  - `syntax: cdk init {template} --language {language-name}`
  - `example: cdk init app --language typescript`

- Values for {template} include:
  - app - creates an blank app
  - lib - creates a basic construct library app
  - sample-app - creates sample app with some resources

Notes

4.10 cdk bootstrap and CDKToolkit

- The CDK requires a staging area within the AWS account in order to do the work of deploying the resources specified in your project.
- This staging area consists of a dedicated S3 bucket which can be created using the CDK bootstrap command:

```
cdk bootstrap
```

- The command is run from within a CDK project directory. It gets your account and region from the local aws '*config*' and '*credentials*' files that were created by 'aws configure'
- If an existing bootstrap environment exists the bootstrap command will detect and use it.
- The bootstrap command creates a stack named CDKToolkit that deploys an s3 bucket named like this: cdktoolkit-stagingbucket-8liiaeqwom2a

Notes

There are a few limited situations where a bootstrap area is not needed. You can find out more that and about bootsrapping in general here:

https://docs.aws.amazon.com/cdk/latest/guide/bootstrapping.html

4.11 Removing CDKToolkit

- The same CDK bootstrap environment can be kept and used by multiple CDK application projects.
- That being said there are some situations where you might need to clean up an account and remove the stack and bucket created for the bootstrap environment.
- The following aws-cli commands can be used if needed:
- Empty the bucket:

```
aws s3 rm --recursive s3://$(aws s3 ls | grep cdktoolkit | cut -d' ' -f3)
```

- Delete the Stack:

```
aws cloudformation delete-stack --stack-name CDKToolkit
```

- Because the bucket is emptied first the stack delete will also remove the bucket.

Notes

This part of the bucket empty command uses standard terminal operations 'grep' and 'cut' to output the name of the bucket:

```
aws s3 ls | grep cdktoolkit | cut -d' ' -f3
```

4.12 AWS Toolkit for Visual Studio Code

- Visual Studio Code is a popular programming editor with good support for TypeScript and other types of development.
- The "AWS Toolkit for VSC" is a plug-in that assists with development of AWS CDK projects.
- The toolkit will use existing 'credentials' and 'config' files setup by 'aws configure' to connect to an AWS account
- Among other features the toolkit includes:
  - "AWS:Explorer" that displays resources from the connected account.
  - Pop-up help for AWS constructs and properties
  - AWS commands can be run from the VSC command pallet

Notes

See here for more information on the AWS Toolkit for Visual Studio Code:

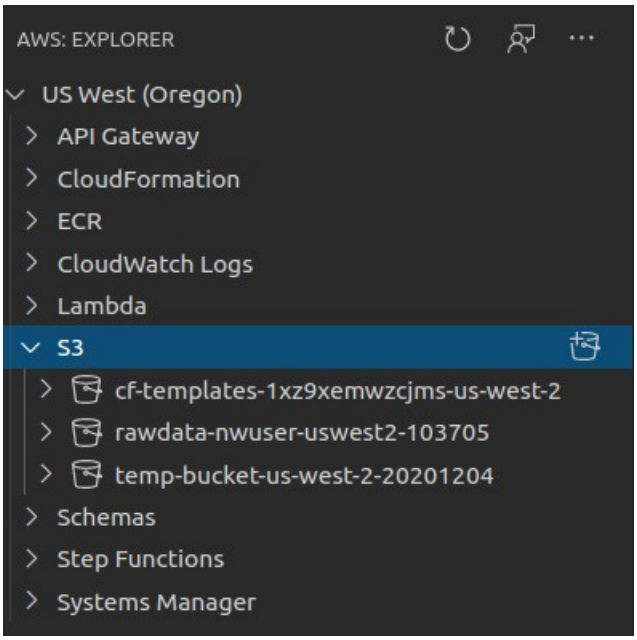
https://aws.amazon.com/visualstudiocode/

Installation:

https://docs.aws.amazon.com/toolkit-for-vscode/latest/userguide/setup-toolkit.html

4.13 AWS Toolkit for VSC - Explorer

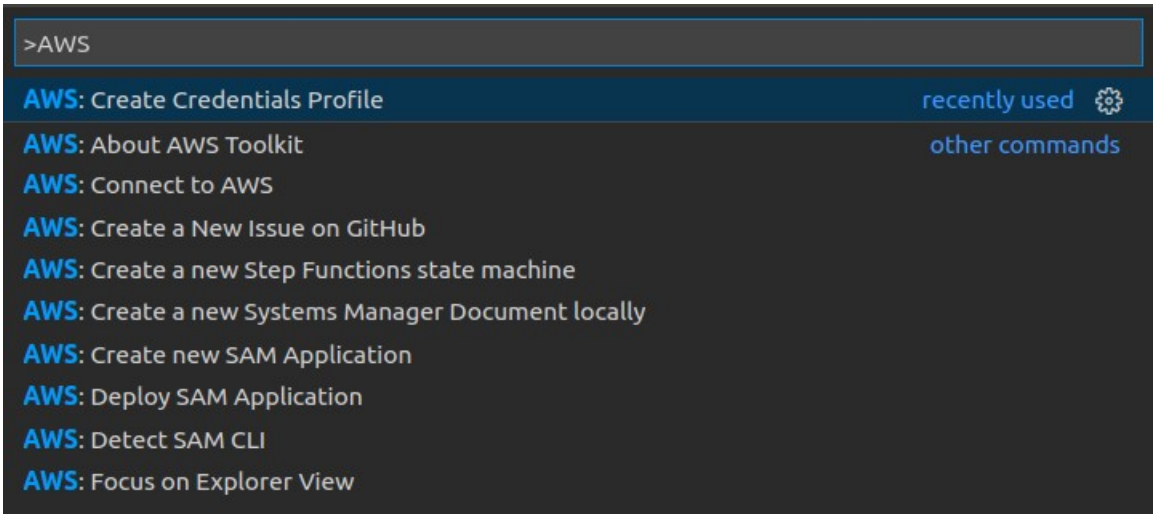
- AWS: Explorer in Visual Studio Code



Notes

4.14 AWS Toolkit for VSC - Commands

- AWS: Commands in Visual Studio Code



Notes

4.15 Summary

In this chapter we covered:

- Setup Overview
- HW/SW Environment
- Account, Permissions, Credentials
- AWS-CLI installation
- aws configure
- AWS-CDK installation
- cdk commands
- Initializing a CDK Project
- cdk bootstrap and CDKToolkit
- Removing CDKToolkit
- AWS Toolkit for Visual Studio Code

Chapter 5 - Working with S3 in the AWS CDK

Objectives

Key objectives of this chapter

- S3 Overview
- The AWS-CDK
- Create a Basic Bucket
- Setting Construct Properties
- Construct Properties
- publicReadAccess Property
- removalPolicy
- versioned Property
- bucketName
- websiteindexDocument Property
- S3 Bucket Website URL
- Deploying the Stack/Bucket
- The BucketDeployment Object
- File Prefixes

5.1 S3 Overview

- **S3** refers to Amazon's Simple Storage Service which supports cloud storage of files
- S3 **Buckets** are containers that hold files
- Files uploaded to Buckets can include prefixes (similar to directories)
- Files in S3 buckets are accessed via URLs like this:

```
S3://{bucket-name}/{prefix}/{filename}
```

- Example:

```
s3://rawdata-uswest2-103705/sales_data/2020-sept.csv
```

Notes

5.2 Managing Buckets and Files

- Buckets and Files can be managed via:
  - AWS S3 Web Console
  - AWS CloudFormation Web Console
  - AWS-CLI command line interface
  - AWS-CDK applications
- In this chapter we'll look at managing S3 Buckets and Files via AWS-CDK applications

5.3 The AWS-CDK

- AWS-CDK (cloud developer kit) Applications work like this:
  - A CDK application project is created
  - Code to manage assets is added to the application
  - The application is compiled to a CloudFormation template
  - The template is deployed to an AWS account as a CloudFormation stack

5.4 The CDK Application Project

- The application project is created like this:

```
mkdir my-cdk-app
cd my-cdk-app
cdk init app --language typescript
```

- Two important code files:

  `\bin\my-cdk-app.ts` - The app's entry point

  `\lib\my-cdk-app-stack.ts` - Code that defines the app's stack

- S3 assets are defined in the second of these two files.

5.5 The app-stack Code File

- The app-stack code extends cdk.Stack

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

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

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

- Changes to this file typically include:
  - aws-cdk module imports added at the top of the file
  - CDK Constructs added after the line "`// The code that defines...`"

5.6 Create a Basic S3 Bucket

- The aws-s3 module includes the Bucket class which is required to create a Bucket. The module needs to be installed into a CDK project before it can be used.

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

- After installing the module it is imported into your code file like this:

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

- A basic S3 bucket is created with the s3.Bucket construct:

```
// The code that defines your stack goes here
new s3.Bucket(this, 'MyFirstBucket');
```

- The above bucket will be created using default values Construct and Bucket properties

Notes

5.7 Setting Properties on a Construct

- The third constructor parameter is an object holding properties.

```
// The code that defines your stack goes here
new s3.Bucket(this, 'MyFirstBucket',{
  bucketName: 'thename',
  publicReadAccess: true
});
```

- Properties define various aspects of the Bucket including how it is created and how it can be used.

5.8 Construct Properties

- Some useful Construct properties include:
  - publicReadAccess
  - removalPolicy
  - versioned

- websiteIndexDocument
- ...

- A list of bucket properties that can be set via the CDK can be found here:

  https://docs.aws.amazon.com/cdk/api/latest/docs/@aws-cdk_aws-s3.Bucket.html#construct-props

Notes

5.9 publicReadAccess Property

- **`publicReadAccess`**
  - Definition: Grants public read access to all objects in the bucket
  - Type: Boolean
    - true: access is granted
    - false: (default) access is denied
- Example:

```
new s3.Bucket(this, 'MyFirstBucket',{
  publicReadAccess: true
});
```

Notes

5.10 removalPolicy

- **`removalPolicy`**
  - Definition: Policy to apply when the bucket is removed from this stack.
  - Type: RemovalPolicy
    - DESTROY: (default) resource is destroyed along with stack

        - RETAIN: resource retained when stack deleted
        - SNAPSHOT: snapshot copy of resource retained and resource itself destroyed along with stack
- Example:

```
new s3.Bucket(this, 'MyFirstBucket',{
  removalPolicy: RemovalPolicy.RETAIN
});
```

- Regardless of the removalPolicy s3.Buckets are not removed unless they are empty. This can cause stack removal to fail if one of its buckets removalPolicy is set to DESTROY.

Notes

5.11 versioned Property

- **`versioned`**
    - Definition: determines if file versioning is turned on or off for the bucket
    - Type: Boolean
        - true: versioning is turned on
        - false: (default) versioning is turned off
- Example:

```
new s3.Bucket(this, 'MyFirstBucket',{
  versioned: false
});
```

- When 'versioned' is true all file versions must be removed before a bucket can be deleted (not just the latest file version).

Notes

5.12 bucketName

- **`bucketName`**
  - Definition: sets the physical name for the bucket
  - Type: string
  - Default: name assigned by CloudFormation (recommended)
- Example:

```
new s3.Bucket(this, 'MyFirstBucket',{
  bucketName: 'big-bucket-987ad9f7ds7'
});
```

- Bucket (and stack) creation will fail if you try to set a bucketName that has already been used. For most purposes it is better to leave this blank and let AWS set a name. The name it creates will be globally unique and usually starts with your stackname and the construct id string like this:

```
mycdkappstack-myfirstbucketa9798686-lkajk879
```

Notes

5.13 websiteIndexDocument Property

- **`websiteIndexDocument`**
  - Definition: sets static website hosting on for the bucket's contents and sets the name of the default document to return
  - Type: string
  - Default: empty, static web hosting is turned off
- Example:

```
new s3.Bucket(this, 'MyFirstBucket',{
  websiteIndexDocument: 'index.html'
});
```

- When 'versioned' is true all file versions must be removed before a bucket can be deleted (not just the latest file version).

Notes

5.14 S3 Bucket Website URL

- Website configured Buckets are accessed via a url formatted like this:

```
http://{bucket-name}.s3-website.{region}.amazonaws.com
```

- Example URL

```
http://my-bucket-98s7dfa9.s3-website.us-west-2.amazonaws.com

Where:
{bucket-name} = my-bucket-98s7dfa9
{region} = us-west-2
```

- The above URL will return the default "index.html" file.
- To retrieve other files you would append their name: `"\other-file.html"`
- Files in subdirectories are also accessible: `"\sub-dir\other-file.html"`

5.15 Deploying the Stack/Bucket

- Once the required Bucket properties have been added the bucket is ready to be deployed.
- In most cases though you will define other resources as well inside your app-stack code file.
- Assuming you have defined all the resources you need you can go ahead and deploy the stack using the command:
  - **`CDK deploy`** - Deploys the project's default stack

- ◇ **`CDK deploy StackName`** - Deploys the named stack

Notes

5.16 Deploy files to a Bucket

- The ***aws-s3-deployment*** module can deploy files to a newly created bucket. To install the module into a CDK project use this command:

```
npm install @aws-cdk/aws-s3-deployment --save
```

- Import the module into your code file like this:

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

- To prepare for the upload:
  - ◇ Create a directory in your project for the files you want to upload
  - ◇ Copy the files-to-upload into the directory
- Create a BucketDeployment object (see next slide)

Notes

5.17 The BucketDeployment Object

- Add the following to upload files from the local project subdirectory "/_filesource" to the bucket

```
let newBucket = new s3.Bucket(this, 'MyFirstBucket', {...});

new s3deploy.BucketDeployment(this, 'DeployDataFile', {
  sources: [s3deploy.Source.asset('./_filesource')],
  destinationBucket: newBucket,
  retainOnDelete: false
```

```
});
```

- Note how *destinationBucket* takes the Bucket's reference variable "newBucket" and not the new bucket's physical name (which does not exist yet).

5.18 File prefixes

- File prefixes can be used with BucketDeployment to upload files into virtual sub-directories in a Bucket.
- For example if you want to load files to the "html" subdirectory inside of "newBucket"

```
new s3deploy.BucketDeployment(this, 'DeployDataFile', {
  sources: [s3deploy.Source.asset('./_html')],
  destinationBucket: newBucket,
  destinationKeyPrefix: '/html',
  retainOnDelete: false
});
```

- Note that the dir '/html' will be created by the above command. It does not need to be created beforehand.

Notes

5.19 Summary

In this chapter we covered:

- S3 Overview
- The AWS-CDK
- Create a Basic Bucket
- Setting Construct Properties

- Construct Properties
- publicReadAccess Property
- removalPolicy
- versioned Property
- bucketName
- websiteindexDocument Property
- Deploying the Stack/Bucket
- The BucketDeployment Object
- File prefixes

Chapter 6 - Programming Lambdas in the AWS CDK

Objectives

Key objectives of this chapter

- AWS Lambda Overview
- Managing AWS Lambdas
- The AWS-CDK
- The app-stack Code File
- Create a Basic Lambda
- The lambda.Function Construct
- The lambda function
- Deploying a Lambda
- Invoking a Lambda

6.1 AWS Lambda Overview

- AWS Lambda is a service that runs code without having to provision server ahead of time
- The term Lambda is also used to refer to the code being run.
- Lambdas:
  - Automatically scale
  - Can be run from other AWS apps, web or mobile applications
  - Can be programmed in Node, Python, Go and Java

Notes

For more information on AWS lambdas see:
https://aws.amazon.com/lambda/

6.2 Managing AWS Lambdas

- AWS Lambdas can be managed via:
  - AWS S3 Web Console
  - AWS CloudFormation Web Console
  - AWS-CLI command line interface
  - AWS-CDK applications
- In this chapter we'll look at managing AWS Lambdas via AWS-CDK applications

Notes

6.3 The AWS-CDK

- AWS-CDK (cloud developer kit) Applications work like this:
  - A CDK application project is created
  - Code to manage assets such as Lambdas is added to the application
  - The application is compiled to a CloudFormation template
  - The template is deployed to an AWS account as a CloudFormation stack
- Lambdas can be used once they are deployed

Notes

6.4 The CDK Application Project

- The application project is created like this:

```
mkdir my-lambda-app
```

```
cd my-lambda-app
cdk init app --language typescript
```

- Two important code files:

  `\bin\my-cdk-app.ts` - The app's entry point

  `\lib\my-lambda-app-stack.ts` - Code that defines the app's stack

- Lambda constructs are added in the second of these two files
- A directory is created in the project root to hold the Lambda's Code

  `{project-root}\lambda`

6.5 The app-stack Code File

- The app-stack code extends cdk.Stack

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

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

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

- Changes to this file typically include:
  - aws-cdk module imports added at the top of the file
  - CDK Constructs added after the line "`// The code that defines...`"

6.6 Create a Basic Lambda

- Two pieces of code are required to implement the Lambda:

- A lambda.Function construct is added to *my-lambda-app-stack.ts*
- A lambda function is added to *./lambda/hello.js*

Notes

6.7 The lambda.Function Construct

- The **aws-lambda** module includes the lambda.Function construct which is required to create a Lambda. The module needs to be installed into the CDK project before it can be used.

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

- After installing the module it is imported into your code file like this:

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

- A basic Lambda is created with the lambda.Function construct:

```
const hello = new lambda.Function(this,
  'HelloHandler', {
    runtime: lambda.Runtime.NODEJS_10_X,
    code: lambda.Code.fromAsset('lambda'),
    handler: 'hello.handler'
  });
```

- The syntax of the above command:

```
new lambda.Function(scope: Construct, id: string, props: FunctionProps)
```

Notes

6.8 Function Construct Properties

- The lambda.Function constructor is passed an object with three properties as its third parameter:

```
{ runtime: lambda.Runtime.NODEJS_10_X,
  code: lambda.Code.fromAsset('lambda'),
  handler: 'hello.handler'}
```

- `runtime` – specifies how the lambda will be executed
- `code` - specified the local directory location of the lambda function code
- `handler` - specifies the {file-name}.{function-name} of the code. In this case the code is in the 'hello.js' file and the function is named 'handler'.

Notes

More information on the lambda.Function construct and its properties can be found here:

https://docs.aws.amazon.com/cdk/api/latest/docs/@aws-cdk_aws-lambda.Function.html

6.9 The lambda function

- Lambda functions can include any code you wish to execute but their structure need to follow a certain pattern.
- The structure of a lambda function:

```
exports.handler = async function(event) {
  // run some code and create a result
  let result = `Lambda says Hello!\n`;
  return {
    statusCode: 200,
    headers: { "Content-Type": "text/plain" },
    body: result
  };
};
```

- The function works like a REST service:
  - It receives a request object (event)
  - It returns a response object with http status, headers and body.

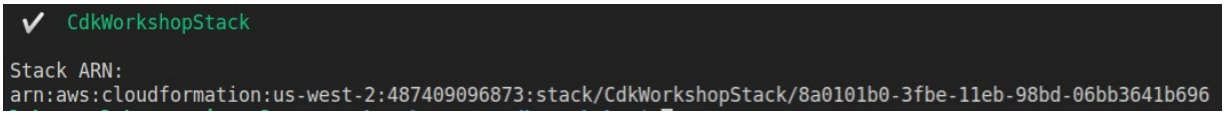
6.10 Deploying a Lambda

- Lambdas are deployed along with any other resources that are specified in the cdk-stack TypeScript file.
- To check which resources will be created use this command:

```
cdk diff
```

- To deploy the CDK project stack use this command:

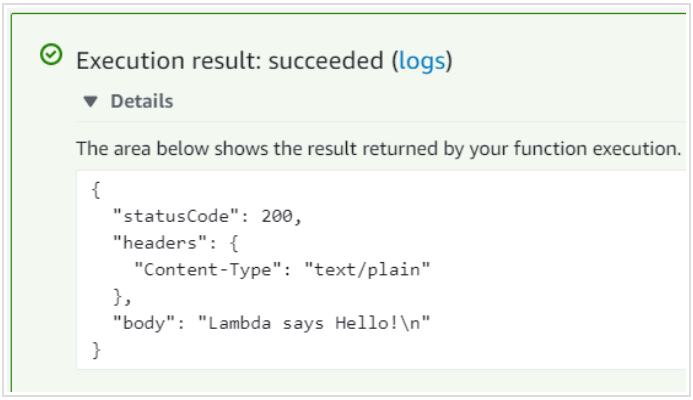
```
cdk deploy
```



Notes

6.11 Testing a Lambda in the Lambda Web Console

- Lambda functions that have been deployed can be found on the Functions page of the AWS Lambda Web Console
- Clicking on the lambda's name brings you to a page where you can inspect the lambda and test it.
- Testing involves:
  - Creating a sample event object
  - Calling the lambda with the event object
- Results of the lambda call are displayed



Notes

AWS Lambda Web Console Functions page:

https://us-west-2.console.aws.amazon.com/lambda/home?region=us-west-2#/functions

6.12 Invoking a Lambda using AWS-CLI

- The AWS-CLI includes the following command for invoking lambdas

```
aws lambda invoke --function-name my-lambda-function output.json
```

- Example:

```
aws lambda invoke \
--function-name CdkWorkshopStack-HelloHandler2E4FBA4D-YAMQAN80D7SA \
output.json
```

- The lambdas output will be saved in the '*output.json*' file.
- A payload object can also be sent by adding the --payload parameter:

```
--payload {"path": "/"}
```

Notes

In some cases the payload needs to be placed in a file and referenced from the command line (instead of being passed directly) for example:

Full invoke syntax:

```
aws lambda invoke --function-name CdkWorkshopStack-HelloHandler2E4FBA4D-
YAMQAN80D7SA --payload fileb://payload.json  output.json
```

To do this, first create a file named payload.json, in the dir where you are running the command, with the required payload contents:

```
{"path": "/"}
```

Then add the payload parameter like this:

```
--payload fileb://payload.json
```

6.13 Invoke Lambda via REST API Endpoint 1/2

- Adding an **apigw.LambdaRestApi** construct to the CDK project lets you call your lambda function via an HTTP REST call.
- The apigw module needs to be installed into the project:

```
npm install @aws-cdk/aws-apigateway --save
```

- Then it needs to be imported:

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

- The construct is then added:

```
new apigw.LambdaRestApi(this, 'Endpoint', {
  handler: hello
});
```

- The hander 'hello' refers to the variable holding the value returned from the lambda.Function() construct that created the lamba resource.

Notes

6.14 Invoke Lambda via REST API Endpoint 2/2

- After adding the endpoint construct you need to deploy again:

```
cdk deploy
```

- The output from the deploy command will include the following:

- You can call this endpoint using curl like this:

```
curl https://u5a658otu7.execute-api.us-west-2.amazonaws.com/prod/
Lambda says Hello!
```

Notes

6.15 Summary

In this chapter we covered:

- AWS Lambda Overview
- Managing AWS Lambdas
- The AWS-CDK
- The app-stack Code File
- Create a Basic Lambda
- The lambda.Function Construct
- The lambda function
- Deploying a Lambda
- Invoking a Lambda