Step Function with AWS CDK in action: our points of view about it using Typescript

15 April 2022 - 5 min. read

AWS Step Functions *Infrastructure as Code (IaC)* *TypeScript*

Introduction

The infrastructure as Code (IaC) has been one of the most popular keywords in the cloud topic.

This made cloud infrastructure implementation much more reusable, versionable, and maintainable.

Several implementations and standards of this concept are recently born.

Some of those are *Ops*-oriented, which are usually based on declarative configuration files, like **Terraform** or **AWS SAM**.

Other tools are more *Dev*-oriented, using popular programming languages like **Typescript**, **Java**, or **Python**.

The official tool to develop your IaC in your AWS account is the Cloud Development Kit (CDK), made by AWS.

We have used it a lot, for different use cases, and today we are going to share our experiences to provide you with some useful tips. Let's dive deep!

Programming language: Typescript

AWS CDK is available for different programming languages, but we usually chose to use Typescript because **it allows developers to use Javascript elasticity and Java type safety**.

Those characteristics are very useful to speed up development, keeping type safety's advantages and enabling the usage of object-oriented design patterns.

Project Structure

A very cool thing about AWS CDK is the modularized structure.

Each AWS service it's represented as a node module, so it's a dependency of your node project.

This permits you to create a lightweight project without useless imports.

Usually, we design the CDK project structure by splitting the main stack into several nested stacks, using domain-driven criteria.

Abstractions layers

AWS CDK provides you with several abstraction layers to create AWS -resources.

We have identified 3 abstraction's macro-categories:

1 - CloudFormation Plain resources:

Typically can be identified from their name's root: "Cfn". Those elements give the possibility to customize your resources the same way you'd do on a CloudFormation Template.

(ie: CfnStateMachineProps)

2 - Attributes abstraction resources:

These constructors are our favorites, they abstract and simplify some tricky resource configurations, without losing control of the created resources.

(ie: StateMachine)

3 - Group of resource abstraction:

These constructors can be very useful if you don't want to customize every resource's details.

They provide a very abstract configuration that creates several cloud resources.

We recommend reading them very carefully the documentation before implementing them to be sure that you need all resources that will be created.

(ie: FargateService)

A success case with StepFunction

AWS StepFunction it's a service that enables the creation of State machines on the cloud.

We found it very handy and well implemented into AWS CDK.

Each step of a state machine has one input and one output and can be of several types:

- **Pass:** This state passes its input to its output, without performing work. Pass states are useful when constructing and debugging state machines.
- **Task:** This represents an operation to execute, it's integrable directly with a Lambda Invoke, or you can specify parameters that call a specific AWS service
- **Choice:** It is possible to configure a condition that permits the user to change execution flow based on the output of the previous state
- **Wait:** It's possible to suspend the machine execution for a specific time
- **Succeed:** When a machine execution finishes with success
- **Fail:** When a machine execution finishes with some errors
- **Parallel:** This permits the execution of a state set that will be executed in parallel, using a single input value.
- **Map:** This permits the execution of a state set, using an array of input for each state.

To create and make the state machine work there is a JSON-based language called Amazon States Language, it enables developers to create conditions and transitions that require manual interactions that connect steps to each other.

This language is a JSON object that contains these attributes:

Comment

This field is used as you use comments in a traditional programming language, it is used to describe State machine behaviors, is not required, so you can omit it if you want.

TimeoutSeconds

This field is not required and it is a number, it defines the maximum number of seconds an execution of the state machine can run. If it runs longer than the specified time, the execution fails with a state.

Version

This attribute is not required, it is the version of the Amazon States Language and is used in the state machine (default is "1.0").

States

This field is mandatory and is a JSON object, used to describe the states that compose machine state.

Each key of that object is the step name.

Here is an example of it:

```
{
    "State1" : {
    },

    "State2" : {
    },
    ...
}
```

StartAt

This attribute is required and defines the first state to invoke to start State Machine.

To create a Step Function using IaC technology with official AWS tools, you can use the CloudFormation service.

This is an example of a very simple flow implementation of a state machine.

```yaml
AWSTemplateFormatVersion: "2010-09-09"
Description: "An example template with an IAM role for a Lambda state machine."
Resources:
  LambdaExecutionRole:
    Type: "AWS::IAM::Role"
    Properties:
      AssumeRolePolicyDocument:
        Version: "2012-10-17"
        Statement:
          - Effect: Allow
            Principal:
              Service: lambda.amazonaws.com
            Action: "sts:AssumeRole"

  MyLambdaFunction:
    Type: "AWS::Lambda::Function"
    Properties:
      Handler: "index.handler"
      Role: !GetAtt [ LambdaExecutionRole, Arn ]
      Code:
        ZipFile: |
          exports.handler = (event, context, callback) => {
              callback(null, "Hello World!");
          };
      Runtime: "nodejs12.x"
      Timeout: "25"

  StatesExecutionRole:
    Type: "AWS::IAM::Role"
    Properties:
      AssumeRolePolicyDocument:
        Version: "2012-10-17"
        Statement:
          - Effect: "Allow"
            Principal:
              Service:
                - !Sub states.${AWS::Region}.amazonaws.com
            Action: "sts:AssumeRole"
      Path: "/"
      Policies:
        - PolicyName: StatesExecutionPolicy
          PolicyDocument:
            Version: "2012-10-17"
            Statement:
              - Effect: Allow
                Action:
                  - "lambda:InvokeFunction"
                Resource: "*"

  MyStateMachine:
    Type: "AWS::StepFunctions::StateMachine"
    Properties:
      DefinitionString:
        !Sub
          - |-
            {
              "Comment": "A Hello World example using an AWS Lambda function",
              "StartAt": "HelloWorld",
              "States": {
                "HelloWorld": {
                  "Type": "Task",
                  "Resource": "${lambdaArn}",
                  "End": true
                }
              }
            }
          - {lambdaArn: !GetAtt [ MyLambdaFunction, Arn ]}
      RoleArn: !GetAtt [ StatesExecutionRole, Arn ]
```

As you can notice, a simple HelloWorld can be very verbose and tricky to configure with a standard Cloudformation template, because the JSON State language is not well integrated with the YAML template.

In one of our use cases, we needed to develop a flow for parallel operations and elaborate a report when all are completed.

With a standard template, this can be a little painful, so we tried to do it with CDK.

We found out that CDK other than providing a way to create and integrate cloud resources with your StepFunction easily also provides State Language abstractions.

Thanks to that writing a flow is more intuitive and maintainable.

```
const definition = new LambdaInvoke(this, 'Retrieve reports', {
      lambdaFunction: this.retrievePeriods,
      outputPath: '$.Payload'
    }).next(new Choice(this, 'Any entity to close?')
      .when(Condition.stringEquals('$.status', 'true'),
        new Map(this, 'Close entity in parallel', {itemsPath: '$.periodToClose'})
          .iterator(performLastCalculation)
          .next(new Map(this, 'Create last report')
            .iterator(createLastCsv)
            .next(notifyErpIsClosing)
            .next(closePeriods)))
      .otherwise(new Pass(this, 'Passing to new states', {
        outputPath: '$.installationId'
      }))
      .afterwards())
      .next(syncWithErp)
      .next(new Map(this, 'Count current periods')
        .iterator(performCalculation)
        .next(new Map(this, 'Create csv')
          .iterator(createCsv)
          .next(notifyErp)
          .next(new Succeed(this, 'Calculation terminated'))));


    this.stateMachine = new StateMachine(this, `${this.envName}-StepFunction`, {
      definition
    });
```

Undoubtedly, AWS CDK is a very powerful tool. Anyway, it is important to keep in mind that it brings the need for a lot of programming knowledge. For this reason, it is necessary to master fundamental topics such as OOP (**Object-oriented programming**) before approaching CDK constructors.

This will allow you to create a scalable and maintainable code structure and to fully take advantage of IaC.

In conclusion

In this article, we have shared a real use case where AWS CDK can be very useful and some best practices that we usually implement in projects.

I hope you enjoyed this overview of AWS CDK and StepFunction. All questions are welcome!

See you again in 14 days for a new article on **Peoud2beCloud**!

Paolo Di Ciaula

DevOps Engineer, Frontend Developer, and Mobile App Developer @ beSharp.I divide my free time between good music, development, and... beer (sometimes mixed for better results ;D)
