Building a Deno Serverless Application using Lambda Custom Runtime & CDK

2 September 2022 - 8 min. read

AWS CDK *AWS Lambda* *CI/CD* *Continuous Delivery* *Deno* *Lambda Custom Runtime* *Serverless*

Introduction

Welcome back to Proud2beCloud, the blog by beSharp! Today we'll be exploring some capabilities offered by the Cloud with a focus on **customization** while always relying on **Infrastructure-as-Code (IaaC)**, **Serverless**, and **managed services**. We'll talk about Lambda, CDK, pipelines, Serverless Repository, and more.

AWS Lambda Functions (FaaS) are great and versatile tools of the AWS Cloud that can be useful in many different situations and applications.

But how can we leverage them if we want to leverage a programming language that is not already supported? The answer is easy: **Custom Runtime**.

We've already seen in one of our previous articles how to create an application C++ on AWS Lambda. Now, two years later, we want to present a new version of the article choosing another vastly talked about a programming language: **Deno**.

We'll also focus on how to **automate the delivery of our application on Lambda thanks to pipelines CI/CD and CDK**.

Custom runtime on Lambda

As previously stated, if you are an assiduous reader of our blog, you probably already know about the possibility to use custom runtime on Lambda.

To quickly summarize: Lambda with a custom runtime differs from a ‘standard’ lambda function because it must contain, other than the source code of the application, all the libraries needed for the execution and to resolve dependencies. Furthermore, if the language is interpreted, the interpreter is also needed in the package.

Using a custom runtime for AWS Lambda guarantees the **elasticity** of the proposed solutions for the resolution and optimization of the problem that might require a specific programming language.

The previous article described in detail how to create a custom runtime from scratch defining all the bootstrap scripts needed. In Today’s article, we’ll **simplify the process further through the use of ready-to-use and reusable elements** when possible.

In fact, for the most part of programming languages, it is likely that a custom runtime already exists and is ready to use or at least a good starting point for a custom solution. They are usually shared via Git or other sharing platforms. In our case, we’ll use an existing runtime of Deno.

What’s Deno

Deno is a runtime for **Javascript**, **Typescript**, and **WebAssembly** based on JS Engine and Rust that has the role of both runtime and package manager. With Deno it is not necessary to have a separate package manager.

Deno aims to be a **secure and reliable scripting environment, intuitive for the developer.** It is open-source under an MIT license.

Like Node.js, **Deno focuses on an event-driven architecture** and can be used to create web servers, execute scientific computation, and other solutions.

Deno differs from Node.js in various aspects:

- It supports only URLs to load local and remote dependencies while Node.js supports both the usage of modules and URLs.
- A package manager for resource fetching is not needed; a registry link npm for Node.js is not necessary.

- It supports a single API to use promise, ES6, and Typescript functionality while Node.js supports both promises and callback Apis.
- It minimizes the size of the core APIs while still giving access to a large standard library with lots of functionalities without any external dependencies.

It should be noted that this is not intended to be a Node.js-Deno comparative article. Node.js as a runtime continues to be an excellent choice ensuring a minimum application code time-to-market and a very fast development experience when compared to that of other programming languages.

Although still at the beginning of its journey, Deno offers some very interesting opportunities that we will investigate along these lines.

Let's start with **sandboxing**, one of its most important features. Deno is secure by default. There are no already given networks, files, or accesses: all types of access require explicit enabling.
In the image below, you can see the various types of sandboxing supported by Deno.

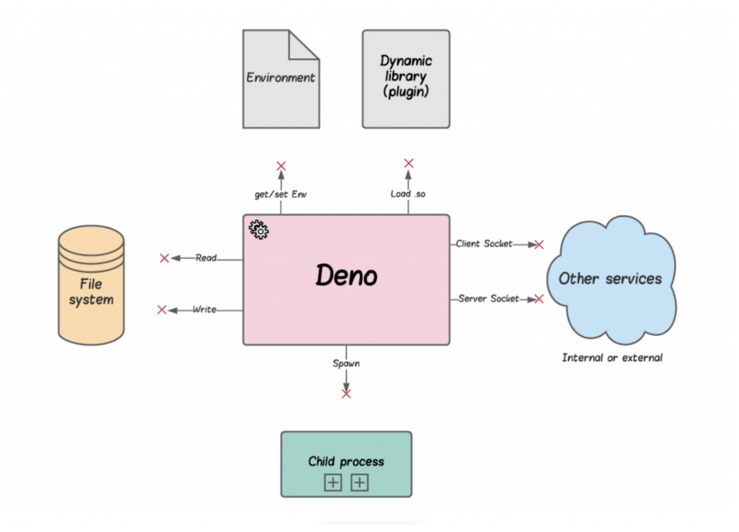


image source: https://medium.com/deno-the-complete-reference/sandboxing-in-deno-b3d514d88b63

AWS Cloud Development Kit (CDK)

The scope of the article is to explain and demonstrate how to create a simple project with lambda with custom runtime, all while keeping a **IaaC approach**. On AWS for

Infrastructure as a Code, we use AWS CDK.

AWS Cloud Development Kit (CDK) is a framework for software development that allows the usage of languages like Typescript and Python, with the advantages that come with them, to describe and define the resources needed for the cloud infrastructure. CDK translates the file written in the language of choice in Cloudformation autonomously at deployment time.

The language chosen for this PoC is DENO.

For the image of the Deno runtime, we're gonna use the AWS Serverless Application Repository.

AWS Serverless Application Repository, as the name might suggest, is a managed repository for Serverless applications that allows teams and developers to share reusable applications other than enabling them to assemble and deploy pieces of infrastructure already written in a simple and effective manner.

It is possible, in fact, to use pre-existent applications directly from the Serverless Application Repository in your own architecture, avoiding duplicating work and saving time.

Every application is packaged with an AWS Serverless Application Model (AWS SAM) template, which defines the resources used. The publicly shared applications include a link to the source code of the application.

For the solution described in this article, we're leveraging the AWS Serverless Application Repository for a direct pull of the necessary scripts to set up the Deno custom Runtime. In this way, there's no need to redefine all the bootstrap scripts:

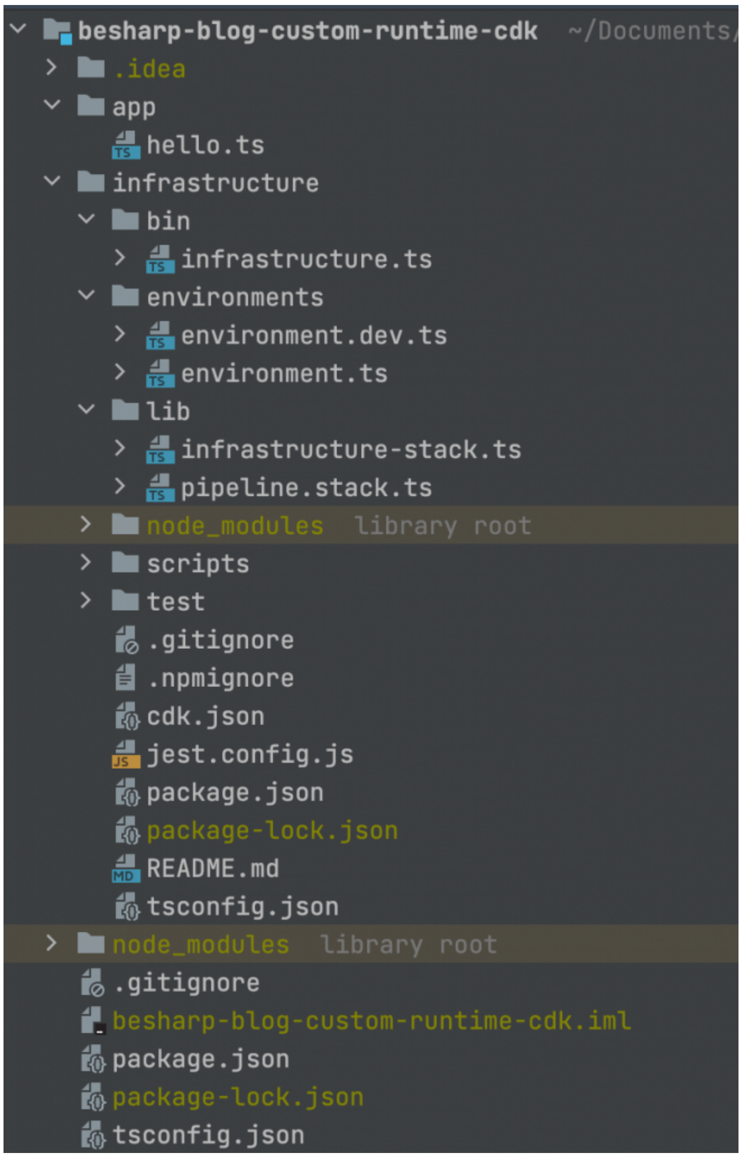
```
const denoRuntime = new CfnApplication(this, "DenoRuntime", {
 location: {
   applicationId:
       "arn:aws:serverlessrepo:us-east-1:390065572566:applications/de
no",
   semanticVersion: "1.24.3",
 },
});
```

Hands on

Let's get into it!

After having created our repository or cloned/pulled it from our Github repository, we can start the project.

We're gonna use a common scaffolding for Serverless applications: both applicative code and the one needed for the infrastructure are in the same codebase. This scaffolding enables us to correlate directly the AWS Lambda functions code (defined in CDK) with the applicative code that will be executed. This avoids the need for any external link.



```
besharp-blog-custom-runtime-cdk  ~/Documents/
  .idea
  app
    hello.ts
  infrastructure
    bin
      infrastructure.ts
    environments
      environment.dev.ts
      environment.ts
    lib
      infrastructure-stack.ts
      pipeline.stack.ts
    node_modules  library root
    scripts
    test
    .gitignore
    .npmignore
    cdk.json
    jest.config.js
    package.json
    package-lock.json
    README.md
    tsconfig.json
  node_modules  library root
  .gitignore
  besharp-blog-custom-runtime-cdk.iml
  package.json
  package-lock.json
  tsconfig.json
```

Starting from root directory we'll find two main folders.

1. App: applicative code, a file in Deno written in typescript *.ts*
2. Infrastructure:

- Stack dedicated to the pipeline
- Stack dedicated to the resources (API Gateway and Lambda) necessary for this demo

Pipeline CI/CD stack

Keeping a IaaC approach with a single repository, we obtain a great advantage over the previous solution described in the first article: we can **deploy directly every update and enhancement without using the AWS console**.

Thanks to this approach, **the risk of misalignment between the code and the underlying infrastructure is avoided**. Also, thanks to Git, both the infrastructure and applicative code are monitorable, versioned, and historicized.

Through the

```
cdk deploy
```

command we'll deploy on our AWS account the **CDK Pipeline**.

This pipeline, translating our CDK Typescript into Cloudformation, will be the one actually deploying our infrastructure as well as the source code.

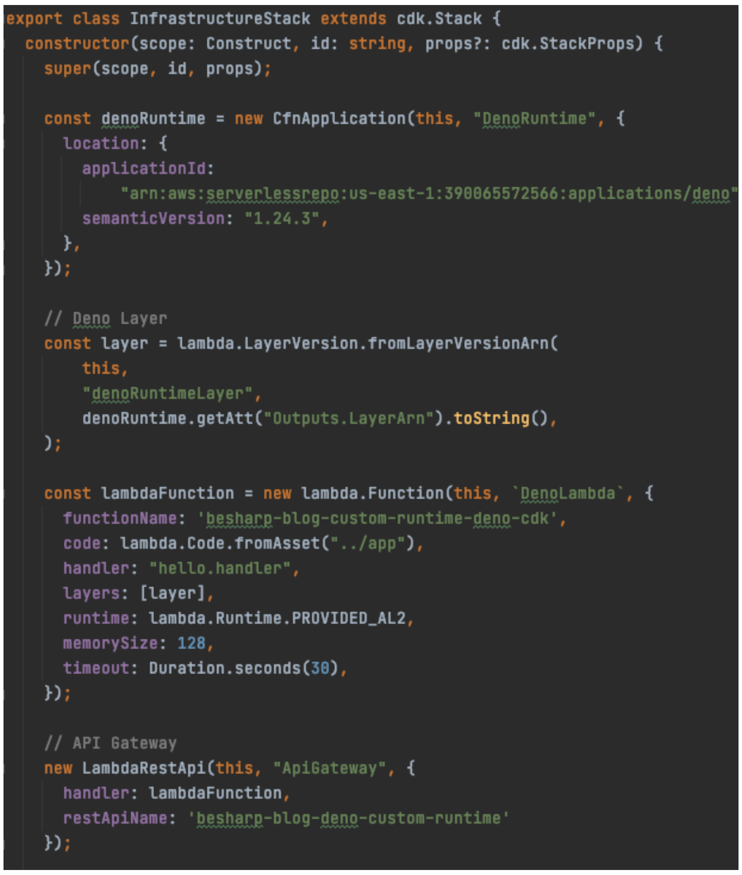
With the automatic Pipeline connected to AWS Codecommit (or Github in this case), the release of a new update is **triggered from a push on a repository's branch**. The deployment will update/create new resources as described in CDK and, subsequently, the applicative code. (eg.: lambda = resource, lambda handler = applicative).

Infrastructure Stack

In the infrastructure-stack.ts we'll be defining the configuration of our resources, in our case Lambda function and API Gateway.

We're defining the name of the function, memory allocated, Lambda layer, and the already cited custom runtime.

The API Gateway is very simple in this project and is used to expose the Lambda code through an endpoint.



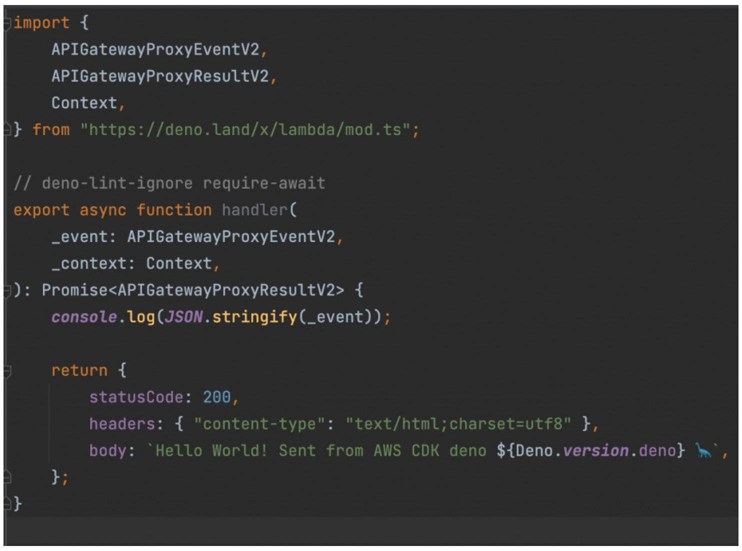
```
export class InfrastructureStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    const denoRuntime = new CfnApplication(this, "DenoRuntime", {
      location: {
        applicationId:
            "arn:aws:serverlessrepo:us-east-1:390065572566:applications/deno"
        semanticVersion: "1.24.3",
      },
    });

    // Deno Layer
    const layer = lambda.LayerVersion.fromLayerVersionArn(
        this,
        "denoRuntimeLayer",
        denoRuntime.getAtt("Outputs.LayerArn").toString(),
    );

    const lambdaFunction = new lambda.Function(this, `DenoLambda`, {
      functionName: 'besharp-blog-custom-runtime-deno-cdk',
      code: lambda.Code.fromAsset("../app"),
      handler: "hello.handler",
      layers: [layer],
      runtime: lambda.Runtime.PROVIDED_AL2,
      memorySize: 128,
      timeout: Duration.seconds(30),
    });

    // API Gateway
    new LambdaRestApi(this, "ApiGateway", {
      handler: lambdaFunction,
      restApiName: 'besharp-blog-deno-custom-runtime'
    });
```

This CDK Typescript code snippet will enable the provisioning of a Lambda function with a Deno custom runtime, pulled directly from the AWS Serverless Application Repository and a simple API Gateway.

The handler defined in the configuration is the *hello.handler* and refers to the *hello.ts* file:

```
code: lambda.Code.fromAsset("../app"),
handler: "hello.handler",
```

The applicative code is in the *hello.ts* file. For the purpose of this demo the function does nothing more than returning a message without discriminating the HTTP Method used during invocation.



```
import {
    APIGatewayProxyEventV2,
    APIGatewayProxyResultV2,
    Context,
} from "https://deno.land/x/lambda/mod.ts";

// deno-lint-ignore require-await
export async function handler(
    _event: APIGatewayProxyEventV2,
    _context: Context,
): Promise<APIGatewayProxyResultV2> {
    console.log(JSON.stringify(_event));

    return {
        statusCode: 200,
        headers: { "content-type": "text/html;charset=utf8" },
        body: `Hello World! Sent from AWS CDK deno ${Deno.version.deno} 🦕`,
    };
}
```

Conclusions

The usage of CDK is widespread and common when using a IaaC approach on AWS. In this article, we depicted how to use Custom Runtime in CDK without the need of writing firsthand all the bootstrapping scripts necessary.

Thanks to the help of the AWS Serverless Application Repository and a ready Deno runtime we succeeded in setting up a Serverless project based on API Gateway and Lambda with Deno custom runtime.

It was not necessary to rewrite the bootstrapping scripts. By keeping a single repository for code and infrastructure, it was possible to connect the provisioning of the resources directly to the applicative source code.

This solution entails many advantages:

- the IaaC approach was kept thanks to the usage of AWS CDK.
- the ability to use AWS Serverless Application Repository for different runtimes already publicly shared or the possibility to define some new ones and share them privately within your organizations or team.
- the Custom Runtime advantages, depending on what runtime is used and for what use case.

In this Deno-based scenario, the dependencies are not installed in the node_modules. By being imported directly from a CDN the deployment package is faster even starting the execution may take some time.

In conclusion, the usage of Custom Runtime depends on the use case. However, having the ability to use custom runtime in pair with CDK without losing the perks of a IaaC approach is surely a great opportunity allowing many other possibilities.

Have you ever used AWS Lambda Custom Rutime for your projects? Tell us your experience in the comment section below.

See you un 14 days with a new article on **Proud2beCloud**!

About Proud2beCloud

Proud2beCloud is a blog by beSharp, an Italian APN Premier Consulting Partner expert in designing, implementing, and managing complex Cloud infrastructures and advanced services on AWS. Before being writers, we are Cloud Experts working daily with AWS services since 2007. We are hungry readers, innovative builders, and gem-seekers. On Proud2beCloud, we regularly share our best AWS pro tips, configuration insights, in-depth news, tips&tricks, how-tos, and many other resources. Take part in the discussion!

Alessandro Bertini

DevOps Engineer @ beSharp. I deal with Cloud-Native software development, strongly oriented to the serverless paradigm!Passionate about board games and video games (as the best geeks do!)

Alberto Casadei

DevOps Engineer @ beSharp. I approach problems with a ‘lazy’ mindset only to find the best and more efficient solution!Cards collector and athlete, "a 'bit' of everything" is the way to go!
