Costruire applicazioni Serverless in Deno con Lambda Custom Runtime & CDK

2 Settembre 2022 - 8 min. read

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

Introduzione

Bentornato su Proud2beCloud, il blog di beSharp! Oggi andremo ad esplorare insieme alcune possibilità offerte dal mondo Cloud concentrandoci sulla customizzazione delle soluzioni mantenendo sempre un approccio ***Infrastructure-as-a-Code*** **(IaaC)**, **Serverless** e il più possibile **fully managed**. Parleremo quindi di Lambda, CDK, pipeline, Serverless Repository e molto altro.

Le **Funzioni AWS Lambda** (FaaS) sono uno strumento estremamente versatile del cloud AWS poiché è possibile utilizzarle per molteplici applicazioni.

Ma come possiamo fare se vogliamo utilizzare un linguaggio di programmazione non attualmente supportato? La risposta è semplice: **Custom Runtime**.

Di custom runtime abbiamo già parlato in un nostro precedente articolo in cui abbiamo visto come realizzare un'applicazione C++ su AWS Lambda. Ora, a 2 anni di distanza, vogliamo presentare una nuova versione dell'articolo scegliendo un altro linguaggio molto chiacchierato al momento: **Deno**.

Ci concentreremo inoltre su come automatizzare il delivery della nostra applicazione in Deno su Lambda grazie a pipeline CI/CD e CDK.

Custom runtime on Lambda

Come anticipato, se sei un lettore del nostro blog, probabilmente conoscerai la possibilità di utilizzare custom runtime all'interno di AWS Lambda.

Per riassumere in breve: Lambda con custom runtime differisce da una lambda 'standard' poiché deve contenere, oltre al codice sorgente, tutte le librerie compilate necessarie per l'esecuzione dell'applicazione e, se il linguaggio scelto è interpretato, anche l'interprete.

Utilizzare una runtime custom per AWS Lambda garantisce l'**elasticità** delle soluzioni proposte dallo sviluppatore per la risoluzione e l'ottimizzazione di problemi che richiedono un linguaggio di programmazione specifico.

Nell'articolo precedente è descritto nel dettaglio come creare una custom runtime partendo da zero. Oggi invece cerchiamo di **semplificare il processo e utilizzare dove possibile elementi pronti e/o riutilizzabili**.

Infatti, per la maggior parte dei linguaggi di programmazione è probabile che esista già una custom runtime pronta all'utilizzo, oppure che ci sia comunque una buona base di partenza per una soluzione custom su GitHub o su altri strumenti di condivisione.

La runtime che andremo ad utilizzare nei paragrafi successivi è quella di Deno.

Cos'è Deno

Deno è una runtime per **Javascript**, **Typescript** e **WebAssembly** basata su JS Engine e su Rust.

Ha il ruolo sia di runtime che di package manager all'interno dello stesso eseguibile, quindi non è necessario un gestore di pacchetti separato.

Deno punta a essere un **ambiente di scripting sicuro e intuitivo** per il programmatore ed è open-source sotto licenza MIT.

Come Node.js anche Deno si concentra su un'architettura ad eventi e può essere utilizzato per creare web server, eseguire computazioni scientifiche ed altre soluzioni.

Deno si differenzia da Node.js in diversi aspetti:

- Supporta solo URLs per caricare dipendenze locali e remote, a differenza di Node.js che supporta sia moduli che URLs.
- Non è necessario un package manager per il fetch delle risorse. Non serve quindi un registro come npm per Node.js.
- Supporta una singola API per l'utilizzo delle promise, ES6, e funzionalità Typescript mentre Node.js supporta sia promises che callback APIs.
- Minimizza la grandezza delle API core comunque dando accesso a una libreria standard con molte funzionalità senza dipendenze esterne.

È bene precisare che questo non vuole essere un articolo comparativo tra Node.js e Deno. Node.js come runtime continua ad essere un'ottima scelta assicurando un time-to-market del codice applicativo minimo e un'esperienza di sviluppo molto veloce se paragonata a quella di altri linguaggi di programmazione.

Se pur ancora acerbo, però, Deno offre alcune opportunità molto interessanti che andremo ad indagare proprio in queste righe.

Partiamo con il ***sandboxing***, una delle sue funzionalità più importanti. Deno è sicuro by default, non ci sono network, file o accessi agli ambienti già dati, tutti i tipi di accesso richiedono l'abilitazione esplicita.

Nell'immagine possiamo vedere i vari tipi di sandboxing supportati da Deno.

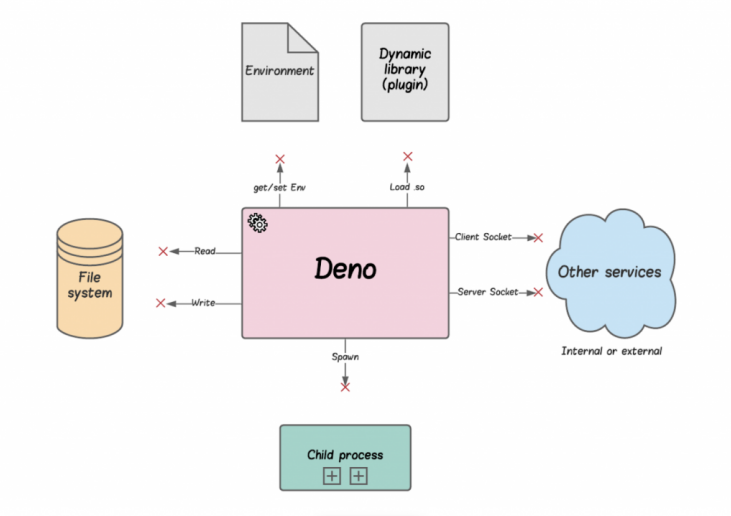


Image source: https://medium.com/deno-the-complete-reference/sandboxing-in-deno-b3d514d88b63

AWS Cloud Development Kit (CDK)

Entriamo nel vivo del nostro articolo: creare un semplice progetto che utilizzi Lambda con custom runtime mantenendo l'approccio **IaaC**.

Su AWS, per l'infrastructure as a code utilizziamo AWS CDK.

L'AWS Cloud Development Kit (CDK), è un framework di sviluppo software che permette l'utilizzo di linguaggi noti come Typescript o Python, con i vantaggi che comportano, per descrivere e definire l'infrastruttura cloud richiesta. Autonomamente CDK traduce poi il linguaggio scelto in Cloudformation al momento del deploy su cloud.

Per il pull dell'immagine runtime di Deno, utilizziamo AWS Serverless Application Repository.

L'AWS Serverless Application Repository, proprio come dice il nome, è una repository gestita per applicazioni serverless e rende possibile a team e sviluppatori di memorizzare e condividere applicazioni riutilizzabili, oltre a poter assemblare e deployare architetture in maniera semplice e veloce.

È possibile infatti utilizzare applicazioni preesistenti direttamente dal Serverless Application Repository nella propria architettura evitando così di duplicare il lavoro in progetti simili e di risparmiare tempo.

Ogni applicazione è pacchettizzata con un AWS Serverless Application Model template che definisce le risorse utilizzate. Le applicazioni condivise pubblicamente includono un link al codice sorgente dell'applicazione.

Per la soluzione descritta in questo articolo utilizziamo AWS Serverless Application Repository per il pull diretto degli script necessari per il setup di una custom runtime in modo da non dover fornire nuovamente i vari shell script di bootstrap.

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

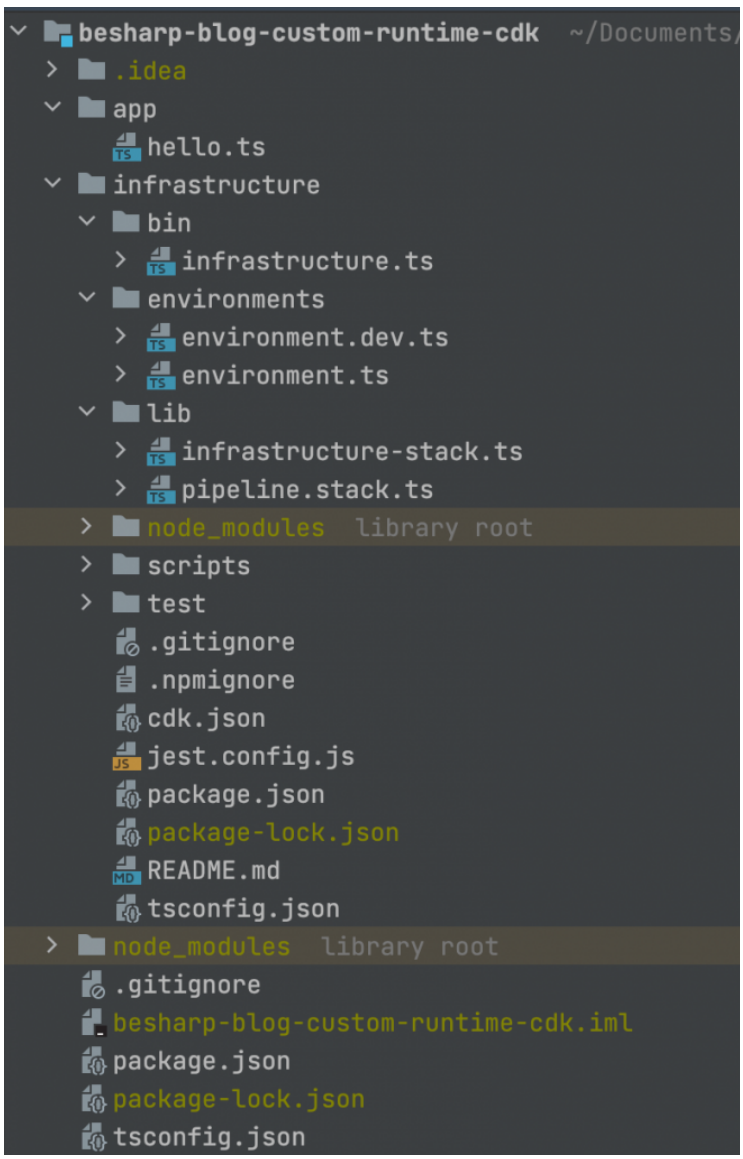
```
  },
});
```

Hands-on

Iniziamo!

Per semplicità, potete trovare il codice sorgente della demo sul nostro repository Github!

Utilizziamo uno scaffolding molto comune per le applicazioni Serverless, ovvero codice applicativo e infrastrutturale sotto la stessa codebase. Questo ci permette di correlare direttamente il codice delle funzioni Lambda, definite in CDK, al codice contenuto nello stesso repository Git, senza alcun link esterno necessario.



Partendo dalla root directory, è quindi possibile trovare due folder principali:

- App: Codice applicativo, un file in Deno, scritto in typescript

- Infrastructure:
  - Stack dedicato alla pipeline
  - Stack dedicato ai servizi (API Gw e Lambda) necessari nella demo

Pipeline CI/CD stack

Mantenendo un approccio IaaC con unica repository abbiamo un altro grande vantaggio rispetto alla soluzione del precedente articolo: poter **rilasciare direttamente modifiche e migliorie cambiando le configurazioni lato codice** e non dovendo per forza utilizzare la console.

Grazie a questo approccio si **evita il rischio di disallineamento** tra il codice e l'infrastruttura sottostante e la condivisione della codebase col team. Facendo leva su Git, è monitorabile, storicizzata e versionata.

Tramite il comando:

```
cdk deploy
```

andiamo a rilasciare sul nostro account AWS la **CDK Pipeline** e sarà quest'ultima, traducendo il codice typescript in Cloudformation, a fare il deploy effettivo dell'infrastruttura.

Con la pipeline automatica collegata ad AWS CodeCommit, o Github nel nostro caso, viene infatti **triggerato un rilascio al push su branch della repository**. Il deploy aggiornerà e/o creerà nuove risorse come descritte in CDK e successivamente l'applicativo che le utilizza (eg: lambda = risorsa , handler = applicativo)

Stack infrastrutturale

All'interno dell'*infrastructure-stack.ts* andremo a definire le configurazioni per la nostra lambda, complete di runtime (vedi sopra), lambda layer in cui verranno inserite le librerie per l'utilizzo di Deno, e configurazioni generiche come memoria, nome dell'handler, nome funzione, timeout etc..

Definiamo anche un API Gateway molto semplice per esporre l'utilizzo della lambda con 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'
    });
```

Questo snippet di codice Typescript in CDK permette quindi il provisioning di una lambda con custom runtime pullata direttamente da AWS Serverless Repository e di un API Gateway collegato.

L'handler definito nelle configurazioni è *hello.handler* e fà riferimento al file *hello.ts* definito al path definito nella configurazione

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

L'effettivo codice applicativo viene descritto sotto la folder *app* nel file *hello.ts*. Per lo scopo di questa demo la funzione non fà altro che ritornare un messaggio dopo essere stata invocata senza discriminare per tipo di richiesta HTTP.

```
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} 🦕`,
    };
}
```

Conclusione

L'utilizzo di CDK è ormai diffuso e molto comune nell'approccio *Infrastructure-as-a-Code* su AWS. In questo articolo abbiamo spiegato come è possibile utilizzare Custom Runtime in CDK senza dover scrivere in prima persona i vari script di bootstrapping della runtime per ogni lambda.

Grazie quindi all'ausilio di AWS Serverless Application Repository e a diverse Runtime già pubbliche e pronte all'utilizzo abbiamo setuppato un progetto serverless basato su Lambda ed API Gateway con la custom Runtime di Deno.

Non è stato necessario definire gli script di definizione della runtime e, mantenendo una repository unica per infrastruttura e codice, è stato possibile collegare il provisioning delle risorse direttamente al codice sorgente applicativo.

I vantaggi che questa possibilità comporta sono vari:

- Approccio IaaC mantenuto con le possibilità date dall'utilizzo di AWS CDK
- Possibilità di utilizzare AWS Serverless Application Repository per runtime già condivise e pubbliche, oppure di poterne definire di personalizzate una volta sola per poi sharare con il team o gli sviluppatori della propria organizzazione
- Custom Runtime: in questo caso i vantaggi dipendono dalla runtime utilizzata e dal caso d'uso di quest'ultima.

Nel caso di Deno rispetto a Node, ad esempio, non vengono installate tutte le dipendenze nei node_modules. Venendo importate direttamente da cdn, il pacchetto di rilascio è più snello (ci sono constraint in MB) anche se ciò comporta uno start leggermente più lungo rispetto a node.

Per concludere, anche se l'utilizzo di custom runtime è molto collegato alle necessità di business, poter utilizzare le runtime direttamente con CDK senza perdere i vantaggi dell'approccio IaaC è sicuramente un'ottima possibilità.

Avete già utilizzato il custom runtime di AWS Lambda per i vostri progetti? Raccontateci nei commenti come lo avete fatto.

Ci vediamo tra 14 giorni su **Proud2beCloud** con un nuovo articolo!

About Proud2beCloud

Proud2beCloud è il blog di beSharp, APN Premier Consulting Partner italiano esperto nella progettazione, implementazione e gestione di infrastrutture Cloud complesse e servizi AWS avanzati. Prima di essere scrittori, siamo Solutions Architect che, dal 2007, lavorano quotidianamente con i servizi AWS. Siamo innovatori alla costante ricerca della soluzione più all'avanguardia per noi e per i nostri clienti. Su Proud2beCloud condividiamo regolarmente i nostri migliori spunti con chi come noi, per lavoro o per passione, lavora con il Cloud di AWS. Partecipa alla discussione!

Alessandro Bertini

DevOps Engineer @ beSharp, mi occupo di sviluppo software Cloud-native, fortemente orientato al paradigma Serverless!Appassionato di giochi da tavolo e videogame (come ogni buon smanettone!)

Alberto Casadei

DevOps Engineer @ beSharp. Sono pigro, questa è la realtà: il che mi permette di trovare sempre la soluzione tecnica più efficiente! Collezionista di carte, atleta e un sacco di altre cose nel tempo libero: "a 'bit' of everything" è il mio motto!