Step Function con AWS CDK in azione: il nostro punto di vista utilizzando Typescript

15 Aprile 2022 - 5 min. read

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

Introduzione

L'espressione "infrastructure as Code" (IaC) negli ultimi tempi è stata una delle keyword più popolari in ambito cloud.

Questo ha reso molte implementazioni infrastrutturali molto più riutilizzabili, versionabili e mantenibili.

Negli ultimi tempi sono nati parecchi standard e implementazioni che usano questo concetto.

Alcuni di questi sono più orientati ai sistemisti, e solitamente sono basati su file di configurazione, alcuni esempi sono Terraform o AWS SAM.

Altri strumenti sono più orientati ai developer e si basano su linguaggi di programmazione come Typescript, Java o Python.

Lo strumento ufficiale orientato agli sviluppatori, per implementare un'infrastruttura usando il concetto di IaC, si chiama Cloud Development Kit (CDK) ed è sviluppato da AWS.

Noi lo abbiamo usato parecchio, per diversi casi d'uso, vorremmo quindi condividere una delle nostre esperienze in questo articolo.

Il linguaggio di programmazione: Typescript

AWS CDK è disponibile per diversi linguaggi di programmazione, noi abbiamo scelto di usare Typescript, perchè consente allo sviluppatore di usare l'elasticità portata da Javascript, unita ad una type safety simile a Java.

Queste caratteristiche sono molto utili per mantenere brevi i tempi di sviluppo, senza sacrificare i vantaggi portati da un linguaggio type-oriented, che facilita l'implementazione di design pattern orientati agli oggetti.

Struttura di progetto

Una cosa che ci ha colpito fin da subito di CDK è la sua struttura modulare.

Ogni servizio AWS è rappresentato da un modulo Node.JS, quindi identificabile come una dipendenza all'interno del progetto.

Questo ci consente di mantenere il progetto leggero e senza import inutili.

Di solito, noi strutturiamo il nostro progetto CDK dividendo lo stack primario in vari nested stack, usando un criterio Domain driven.

Livelli di astrazione

AWS CDK fornisce diversi livelli di astrazione che consentono di creare le risorse AWS.

Noi abbiamo identificato 3 livelli di astrazione:

1 - CloudFormation Plain resources:

Tipicamente possiamo identificarle dalla radice del nome: "Cfn". Questi elementi danno la possibilità di personalizzare le tue risorse nello stesso modo in cui lo fai in un template Cloudformation.

(esempio: CfnStateMachineProps)

2 - Attributes abstraction resources:

Questi costrutti sono i nostri preferiti, sono in grado di astrarre e semplificare la configurazione delle risorse, senza perderne il controllo.

(esempio: StateMachine)

3 - Group of resource abstraction:

Questi costrutti possono essere molto utili, se il tuo requisito non prevede una personalizzazione avanzata delle tue risorse, poichè viene fornita un unica configurazione per creare gruppi di elementi, già integrati fra loro.

Noi raccomandiamo di leggere molto attentamente la documentazione prima di procedere con l'implementazione di questi costrutti, per essere sicuri che le risorse create vi servano realmente.

(esempio: FargateService)

Un caso di successo con StepFunction

AWS StepFunction è un servizio che abilita alla creazione di macchine a stati in Cloud.

Noi lo abbiamo trovato molto comodo e ben implementato all'interno di AWS CDK.

Ogni step della macchina a stati ha un input e un output e può essere di diverso tipo:

- **Pass:** Il valore di input viene forwardato in output, può essere utile a scopo di debug
- **Task:** Rappresenta un operazione da eseguire, questa tipologia di stato è integrabile direttamente con una LambdaInvoke, oppure si può fare forward dei parametri specificati verso un altro servizio AWS
- **Choice:** È possibile configurare una condizione che permette di cambiare il flusso della macchina a stati in base all'output dello stato precedente
- **Wait:** È possibile sospendere la macchina per un tempo specificato
- **Succeed:** Determina la fine positiva del nostro flusso
- **Fail:** Determina la fine negativa del nostro flusso
- **Parallel:** Permette la creazione di un set di stati in parallelo utilizzando un singolo input.
- **Map:** Permette l'esecuzione di un set di stati per ogni elemento di una lista di input

Per creare e far funzionare una macchina a stati, Amazon ha creato un linguaggio basato su JSON, Amazon States Languages.

Questo permette agli sviluppatori di creare e personalizzare gli stati, in modo da connetterli a vicenda.

Questo linguaggio è composto da un macro oggetto JSON che contiene questi attributi:

Comment

Questo campo è usato come i commenti nei linguaggi di programmazione tradizionali.

È utilizzato per riassumere il comportamento della macchina a stati, non è un campo obbligatorio.

TimeoutSeconds

Questo campo non è obbligatorio e definisce il numero massimo di secondi che un esecuzione di una macchina a stati può impiegare a finire.

Se questo limite viene violato, lo stato della macchina diventa Fail.

Version

Questo campo non è obbligatorio, definisce la versione del Amazon State Language utilizzato (il default è "1.0").

States

Questo campo è obbligatorio, è un oggetto JSON, e viene usato per descrivere gli stati che compongono la macchina a stati.

Ogni chiave di quell'oggetto rappresenta il nome dello stato.

Ecco un esempio:

```
{
    "State1" : {
    },

    "State2" : {
    },
```

```
    ...
}
```

StartAt

Questo attributo è obbligatorio e definisce il primo step della macchina a stati da invocare.

Per creare una StepFunction usando le tecnologie IaC tramite strumenti AWS ufficiali, puoi usare il servizio Cloudformation.

Qui sotto troviamo un esempio di implementazione di una StepFunction in CloudFormation.

```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 ]
```

Come puoi notare, un semplice HelloWorld può essere molto verboso e non semplice da configurare con un template Cloudformation, questo succede perchè il JSON state language non è ben integrato con il nostro YAML Template.

In uno dei nostri casi d'uso, avevamo bisogno di sviluppare una macchina che da una serie di operazioni multiple, eseguite in parallelo, riuscisse ad elaborare un report al

termine di tutte le computazioni.

Con un template standard, questo può diventare difficile da implementare, così abbiamo provato a farlo usando CDK.

Abbiamo scoperto che il costrutto relativo a StepFunction, oltre a fornire un modo semplice per integrare le risorse esterne con la tua macchina a stati, è in grado di fornire un ulteriore astrazione per lo State Language visto in precedenza.

Grazie a questo, sviluppare un flusso diventa molto più intuitivo e mantenibile.

```
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
    });
```

AWS CDK è indubbiamente uno strumento molto potente. Ad ogni modo, è importante tenere a mente che, prima di avvicinarsi ai costruttori di CDK, è necessario padroneggiare alcune conoscenze di programmazione fondamentali come l'OOP (Programmazione orientata agli oggetti).

Per concludere

In questo articolo abbiamo condiviso un caso d'uso reale, dove AWS CDK può essere molto utile, insieme all'applicazione di qualche nostra best practice interna.

Spero che questa panoramica sul modo CDK e Step Function vi sia stata utile. Diteci cosa ne pensate!

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

Paolo Di Ciaula

DevOps Engineer, Frontend Developer e Mobile App Developer @ beSharp. Il mio tempo libero è diviso tra musica, sviluppo... e birra (spesso insieme, per migliori risultati ;D)