EBOOK DE FÁCIL APRENDIZADO

CRIANDO AWS LAMBDA COM JAVA E TERRAFORM λ

Coffee and Tips

Tech Tutorials

COFFEEANDTIPS.COM

O que é um AWS Lambda?

AWS Lambda é um serviço de computação sem servidor, também conhecido como Serverless. A execução do Lambda permite que você crie aplicativos de back-end usando diferentes linguagens de programação como Java, Python, Node.js, .Net, Ruby, Go e muito mais.

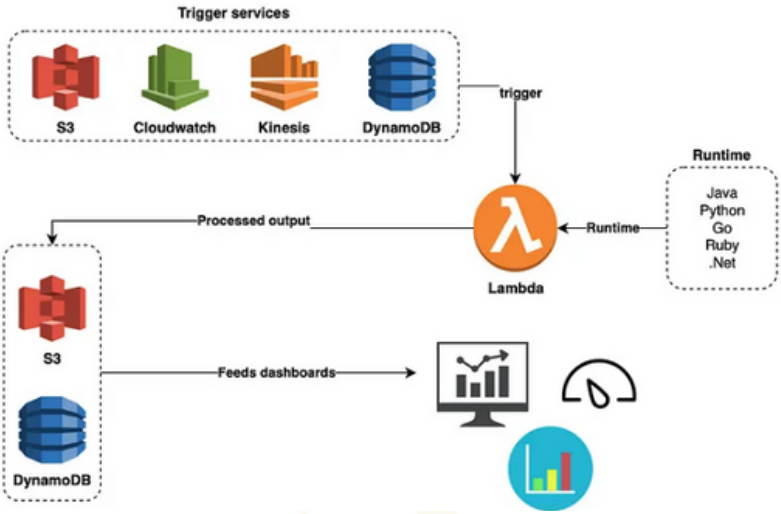
A melhor parte do Lambda é que você não precisa se preocupar com servidores de aplicação para implantá-lo e executá-lo.

Sem preocupações com as responsabilidades de capacidade de provisionamento que geralmente são utilizados em aplicações tradicionais como aplicações Web, onde requer o redimensionamento dependendo do tráfego, tornando-se uma alternativa mais barata para compor arquiteturas.

Como funciona

Lambdas também são usados para compor arquiteturas sendo responsáveis por cargas de trabalho específicas. Por exemplo, usando o Lambda, você pode começar a capturar/escutar arquivos de um S3 Bucket para efetuar algum tipo de normalização ou também pode usar o EventBridge (Cloudwatch events) criando agendamentos por meio de uma expressão cron para acionar o Lambda executando cargas de trabalho e depois encerrar o mesmo.

Conforme mostrado na imagem abaixo, temos alguns exemplos de serviços que integram-se com o Lambda, dessa forma você pode usá-los para invocar os Lambdas para uma variedade de cenários.



Limitações

Lambdas podem ser executados por até 15 minutos, ou seja, seu timeout é de no máximo 15 minutos ou 900 segundos. Portanto, se for usá-lo, atente-se ao lidar com cargas de trabalho que levam mais de 15 minutos.

Integrações

Conforme mencionado anteriormente, o AWS Lambda permite que várias integrações de serviços sejam usadas como um gatilho/trigger. Se você deseja escutar objetos criados no S3 Bucket, pode usar o S3 como trigger. Se você precisar processar notificações do SNS, também poderá definir o Amazon Simple Notification Service (SNS) como o trigger e o Lambda receberá todas as notificações para serem processadas.

Observe que temos diferentes cenários em que o Lambda pode resolver soluções com eficiência. Aqui você pode ver uma lista completa sobre os serviços integrados.

Preços

A AWS possui certas políticas sobre o uso de cada serviço. Lambdas são basicamente cobrados pelo número de requisições e pelo tempo de execução do código.

Benefícios

- **Preço**: Pague apenas por requisições e tempo de execução do código
- **Serverless**: Não há necessidade de um servidor de aplicação
- **Integrado**: o Lambda fornece integração com boa parte dos serviços da AWS
- **Linguagem de Programação**: É possível utilizar as principais linguagens de programação
- Escala e concorrência: Permite controlar a concorrência e dimensionar o número de execuções até o limite da conta

Mãos na massa

Lembrando que para executar todo o conteúdo deste ebook, é necessário ter uma conta na AWS.

Para este ebook, vamos criar um AWS Lambda que será trigado através do CloudWatch Events por meio de um agendamento automatizado usando uma expression cron.

Normalmente, podemos criar qualquer recurso da AWS usando o console, mas aqui usaremos o Terraform como uma ferramenta IaC (Infraestrutura como código) que criará qualquer recurso necessário para executar nosso AWS Lambda. Como runtime ou linguagem de programação, escolhemos Java.

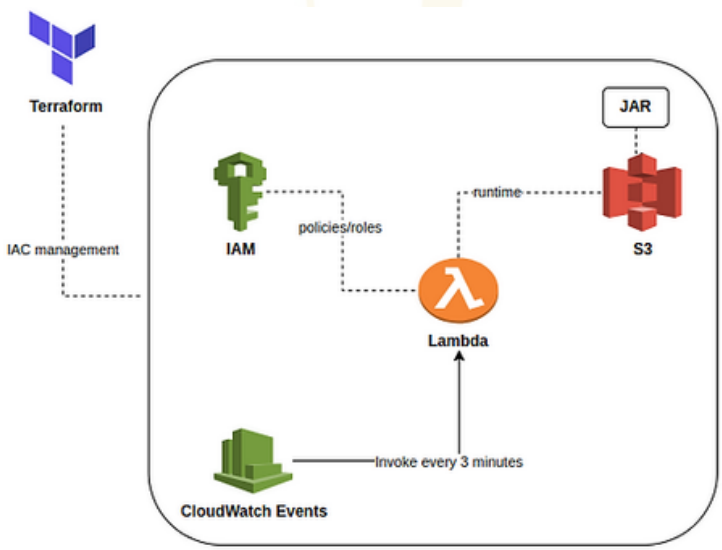
Lembre-se de que você pode executar o Lambda usando diferentes linguagens de programação, como Java, Python, .NET, Node.js e muito mais.

Mesmo que seja um projeto Java, a parte mais importante deste ebook é tentar entender sobre o Lambda e como você pode cria-lo através do Terraform.

Introdução

O Terraform será responsável por criar todos os recursos para este post, como o prórprio Lambda, roles, policies, CloudWatch Events e S3 Bucket, onde manteremos o arquivo JAR de nosso aplicativo.

Nosso Lambda será invocado pelo CloudWatch Events a cada 3 minutos executando um método simples em Java no qual imprimirá uma mensagem conforme imagem abaixo.



Você pode observar na imagem acima que estamos usando S3 para armazenar nosso pacote de implantação, arquivo JAR neste caso. É uma recomendação da AWS fazer upload de pacotes de implantação maiores diretamente para o S3, em vez de manter no próprio Lambda. Isso acontece pois o S3 oferece suporte para armazenar arquivos maiores. Não se preocupe em carregar arquivos manualmente, o Terraform também será responsável por fazer isso durante a fase de build.

Criando o projeto

Vamos gerar o projeto utilizando o Maven onde o mesmo vai criar toda a nossa estrutura de pastas e arquivos necessários para rodar o projeto.

Instalando o Maven

Baixe o arquivo de instalação de acordo com o seu sistema operacional acessando este link https://maven.apache.org/download.cgi e instale conforme os passos deste link https://maven.apache.org/install.html.

Após a instalação, abra o terminal e execute o comando abaixo:

mvn archetype:generate -DgroupId=coffee.tips.lambda -DartifactId=aws-lambda-java-terraform -DarchetypeArtifactId=maven-archetype-quickstart -DarchetypeVersion=1.0 -DinteractiveMode=false

Estrutura do projeto

Após gerar o projeto Maven, vamos criar os mesmos arquivos e pacotes conforme imagem abaixo com o progresso do nosso estudo, exceto o pom.xml que foi criado pelo gerador maven mas que será alterado em breve.

```
aws-lambda-java-terraform/
├── pom.xml
├── src
│   └── main
│       └── java
│           └── coffee
│               └── tips
│                   └── lambda
│                       └── Handler.java
└── terraform
    ├── development
    │   └── vars.tfvars
    ├── lambda.tf
    ├── main.tf
    ├── s3.tf
    └── vars.tf
```

É uma característica dos projetos Maven gerar essas estruturas de pastas como mostrado src/main/java/. Dentro da pasta java/, crie um pacote chamado coffee.tips.lambda e crie uma classe Java chamada Handler.java dentro deste mesmo pacote.

Estrutura do projeto

Para este tutorial, adicione as seguintes dependências e também copie a fase de build abaixo:

Pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.example</groupId>
  <artifactId>aws-lambda-java-terraform</artifactId>
  <version>1.0</version>
  <name>Archetype - aws-coffee.tips.lambda-java-terraform</name>
  <url>http://maven.apache.org</url>

  <dependencies>
    <dependency>
      <groupId>com.amazonaws</groupId>
      <artifactId>aws-java-sdk-lambda</artifactId>
      <version>1.12.395</version>
    </dependency>

    <dependency>
      <groupId>com.amazonaws</groupId>
      <artifactId>aws-lambda-java-core</artifactId>
      <version>1.2.0</version>
    </dependency>

    <dependency>
      <groupId>com.amazonaws</groupId>
      <artifactId>aws-java-sdk-core</artifactId>
      <version>1.12.395</version>
    </dependency>

    <dependency>
      <groupId>com.amazonaws</groupId>
      <artifactId>aws-lambda-java-events</artifactId>
      <version>3.9.0</version>
    </dependency>
  </dependencies>

  <build>
    <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <configuration>
        <source>11</source>
        <target>11</target>
      </configuration>
    </plugin>
  </plugins>
</build>

</project>
```

Criando o Handler

Uma classe handler é basicamente o controlador Lambda. O Lambda sempre procura um handler para iniciar seu processo, resumindo, é o primeiro código a ser invocado.

Criamos um handler básico apenas para "printar" mensagens quando invocadas pelo CloudWatch Events. Obeserve que implementamos a interface RequestHandler permitindo receber como parâmetro um objeto Map<String, Object>. Mas para este exemplo não vamos explorar os dados deste parâmetro.

Handler.java

```
package coffee.tips.lambda;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.LambdaLogger;
import com.amazonaws.services.lambda.runtime.RequestHandler;


import java.util.Map;

public class Handler implements RequestHandler<Map<String, Object>, Void> {

    @Override
    public Void handleRequest(Map<String, Object> input, Context context) {
        LambdaLogger logger = context.getLogger();
        logger.log("Hello Lambda triggered by EventBridge (Cloudwatch Events)");
        return null;
    }
}
```

Criando os arquivos Terraform

Após criado o Handler é hora de criarmos os arquivos Terraform. Caso você não tenha instalado o Terraform, sugiro acessar este link para a instalação https://developer.hashicorp.com/terraform/downloads

Após instalado, vamos entender como os recursos serão criados usando o Terraform.

Arquivo vars.tf

No arquivo vars.tf é onde declaramos as variáveis. As variáveis fornecem flexibilidade quando precisamos trabalhar com diferentes recursos e ambientes de execução. Segue o conteúdo abaixo:

vars.tf

```
variable "region" {
default = "us-east-1"
}

variable "bucket" {}
variable "lambda_function" {}
variable "lambda_filename" {}
variable "file_location" {}
variable "lambda_handler" {}
variable "runtime" {}
variable "cron" {}
variable "timeout" {}
```

Arquivo vars.tfvars

Agora precisamos definir os valores dessas variáveis. Então, vamos criar uma pasta chamada /development dentro da pasta terraform.

Após a criação da pasta. Crie um arquivo chamado vars.tfvars como na imagem abaxio e cole o conteúdo seguinte.

```
terraform
├── development
│   └── vars.tfvars
```

vars.tfvars

```
bucket = "your.bucket.name.must.be.unique"
lambda_function = "coffee_tips_aws_lambda"
lambda_filename = "aws-lambda-terraform-java-1.0.jar"
file_location = "../target/aws-lambda-terraform-java-1.0.jar"
lambda_handler = "coffee.tips.lambda.Handler"
runtime = "java11"
cron = "rate(2 minutes)"
timeout = 2
```

Observe que no campo **bucket**, você deve especificar o nome do seu próprio bucket. O nome do bucket deve ser único.

Arquivo main.tf

Para o arquivo main.tf, apenas declaramos o provedor conforme o conteúdo abaixo.

main.tf

```
provider "aws" {
  region = var.region
}
```

Provider é o serviço de nuvem que usaremos para criar nossos recursos. Neste caso, estamos usando a AWS como provider e o Terraform fará o download dos pacotes necessários para criar os recursos.
vars.tf.

Observe que, para o campo **region**, estamos usando a palavra-chave **var** para obter o valor da região já declarado no arquivo vars.tfvars.

Arquivo s3.tf

É no s3.tf que vamos declarar os recursos relacionados ao S3 em geral. Nesse caso, criamos apenas o bucket S3. Mas se você quiser criar mais recursos relacionados ao S3, como policies, roles ou qualquer recurso relacionado, pode declarar aqui. É uma forma de separar os arquivos Terraform por recurso.

s3.tf

```
resource "aws_s3_bucket" "bucket" {
    bucket = var.bucket
}
```

Observe novamente que estamos usando a palavra-chave var para a variável bucket declarada no arquivo var.tf.

Arquivo lambda.tf

Finalmente nosso último arquivo terraform, neste arquivo estamos declarando recursos relacionados ao Lambda e ao próprio Lambda.

lambda.tf

```
data "aws_iam_policy_document" "coffee_tips_aws_lambda_iam_policy_document" {
  statement {
    effect = "Allow"

    principals {
      type        = "Service"
      identifiers = ["lambda.amazonaws.com"]
    }

    actions = ["sts:AssumeRole"]
  }
}

data "aws_iam_policy_document" "aws_iam_policy_coffee_tips_aws_lambda_iam_policy_document" {
  statement {
    effect = "Allow"
    resources = ["*"]
    actions = [
      "logs:CreateLogGroup",
      "logs:CreateLogStream",
      "logs:PutLogEvents"
    ]
  }
}

resource "aws_iam_role" "iam_for_lambda" {
  name = "iam_for_lambda"
  assume_role_policy = data.aws_iam_policy_document.coffee_tips_aws_lambda_iam_policy_document.json
}

resource "aws_iam_role_policy" "aws_lambda_iam_policy" {
  policy = data.aws_iam_policy_document.aws_iam_policy_coffee_tips_aws_lambda_iam_policy_document.json
  role = aws_iam_role.iam_for_lambda.id
}

resource "aws_s3_object" "s3_object_upload" {
  depends_on = [aws_s3_bucket.bucket]
  bucket = var.bucket
  key    = var.lambda_filename
  source = var.file_location
  etag = filemd5(var.file_location)
}
```

```
resource "aws_lambda_function" "coffee_tips_aws_lambda" {
  function_name = var.lambda_function
  role          = aws_iam_role.iam_for_lambda.arn
  handler       = var.lambda_handler
  source_code_hash = aws_s3_object.s3_object_upload.key
  s3_bucket     = var.bucket
  s3_key        = var.lambda_filename
  runtime       = var.runtime
  timeout       = var.timeout
}

resource "aws_cloudwatch_event_rule" "event_rule" {
  name = "event_rule"
  schedule_expression = var.cron
}

resource "aws_cloudwatch_event_target" "event_target" {
  arn  = aws_lambda_function.coffee_tips_aws_lambda.arn
  rule = aws_cloudwatch_event_rule.event_rule.name
  target_id = aws_lambda_function.coffee_tips_aws_lambda.function_name
}

resource "aws_lambda_permission" "lambda_permission" {
  statement_id = "AllowExecutionFromCloudWatch"
  action        = "lambda:InvokeFunction"
  function_name = aws_lambda_function.coffee_tips_aws_lambda.function_name
  principal     = "events.amazonaws.com"
  source_arn = aws_cloudwatch_event_rule.event_rule.arn
}
```

Agora acho que vale a pena explicar alguns detalhes sobre o arquivo acima.

1. Declaramos 2 objetos do tipo data aws_iam_policy_document, que definem quais ações os recursos que serão atribuídos a essas policies podem executar.

2. aws_iam_role: recurso que fornece a função IAM e que também controlará algumas ações do Lambda.

3. aws_iam_role_policy: fornece IAM role inline policy e registrará a role e as policies anteriores relacionadas ao aws_iam_policy_document.aws_iam_policy_coffee_tips_aws_lambda_iam_policy_document.

4. Declaramos o recurso aws_s3_object porque queremos armazenar nosso arquivo jar no S3. Assim, durante a fase de deploy, o Terraform obterá o arquivo jar que será criado na pasta target/ do projeto e fará o upload para o S3.

- depends_on: Terraform deve criar este recurso antes do atual.
- bucket: É o nome do bucket onde irá armazenar o arquivo jar.
- key: nome da jar.
- source: localização do arquivo de origem.
- etag: aciona atualizações quando acontecem alterações.

5. aws_lambda_function is the resource responsible to create Lambda and we need to fill some fields such as:

- function_name: Nome do Lambda
- role: role declarada nas etapas anteriores que fornece acesso aos serviços e recursos da AWS.
- handler: Neste campo você precisa especificar o diretório da classe principal.
- source_code_hash: Este campo é responsável por acionar as atualizações do lambda.
- s3_bucket: É o nome do bucket onde irá armazenar o arquivo jar gerado durante o deploy.
- s3_key: Nome do jar
- runtime: Aqui você pode especificar a linguagem de programação suportada pelo Lambda. Para este exemplo, java 11.
- timeout: Tempo limite de execução do Lambda.

6. aws_cloudwatch_event_rule é a regra relacionada à execução do evento do CloudWatch. Neste caso, podemos definir o cron através do campo schedule_expression para definir quando o lambda será executado. Neste caso, definimos que o Lambda será executado a cada 2 minutos - rate(2 minutes), especificado no arquivo vars.tfvars.

7. aws_cloudwatch_event_target é o recurso responsável por acionar o Lambda usando eventos do CloudWatch.

8. aws_lambda_permission permite execuções do CloudWatch.

Packaging

Agora que você está familiarizado com o Lambda e o Terraform, vamos empacotar nosso projeto via Maven antes da criação do Lambda. A ideia é criar um arquivo jar que será usado para execuções do Lambda e armazenado no S3.

Para este exemplo, vamos empacotar localmente. Vale lembrar que para um ambiente de produção, poderíamos utilizar ferramentas de integrações contínuas como Jenkins, Drone ou até Github actions para automatizar esse processo.

Primeiro, abra o terminal e verifique se você está no diretório raiz do projeto e execute o seguinte comando maven:

mvn clean install -U

Este comando além de empacotar o projeto, irá baixar e instalar as dependências declaradas no arquivo pom.xml.

Depois de executar o comando acima, um arquivo jar será gerado dentro da pasta target/ também criada.

```
target/
├── aws-lambda-terraform-java-1.0.jar
├── classes
│   └── coffee
│       └── tips
│           └── lambda
│               └── Handler.class
├── generated-sources
│   └── annotations
├── maven-archiver
│   └── pom.properties
└── maven-status
    └── maven-compiler-plugin
        └── compile
            └── default-compile
                ├── createdFiles.lst
                └── inputFiles.lst
```


Executando Terraform

Bem, estamos quase lá. Agora, vamos provisionar nosso Lambda via Terraform. Então, vamos executar alguns comandos do Terraform. Dentro da pasta terraform, execute os seguintes comandos no terminal:

terraform init

O comando acima iniciará o terraform, baixando as bibliotecas do terraform e também validará os arquivos do terraform.

Para o próximo comando, vamos executar o comando plan para checar quais recursos serão criados.

terraform plan -var-file=development/vars.tfvars

Por fim, podemos solicitar a criação definitiva dos recursos através do seguinte comando:

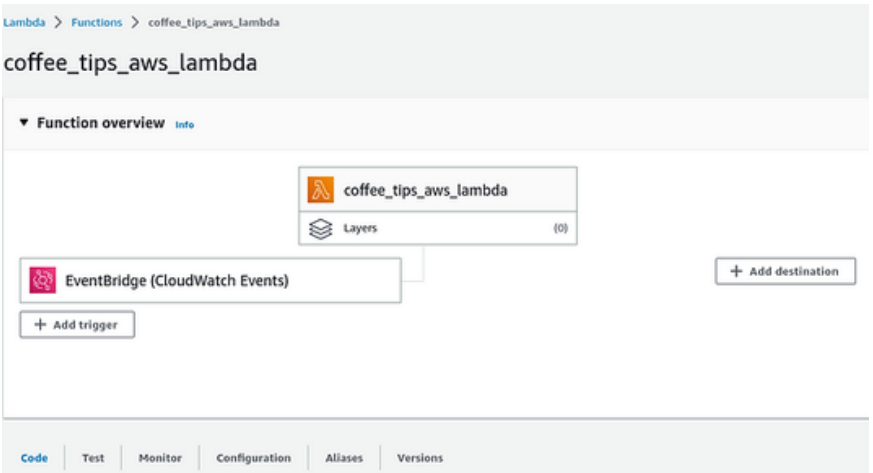
terraform apply -var-file=development/vars.tfvars

Após a execução, você deve confirmar para executar as ações, digite "yes".

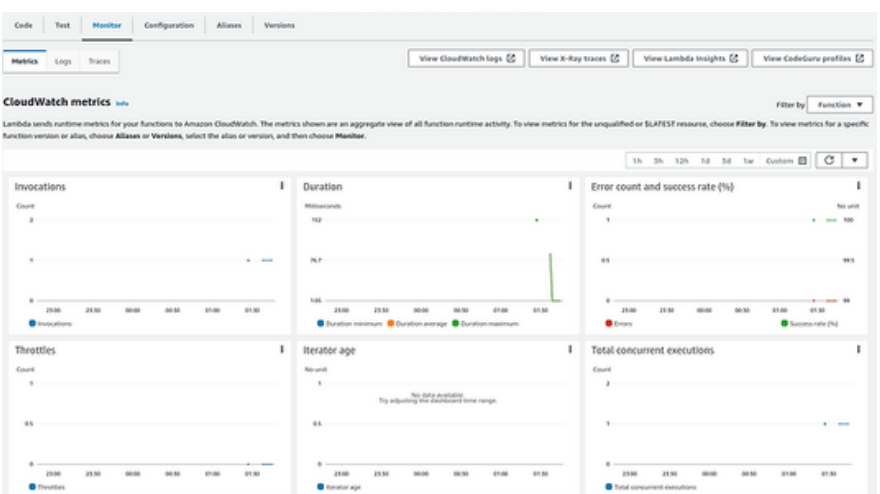
Agora a provisão foi concluída!

Execução do Lambda

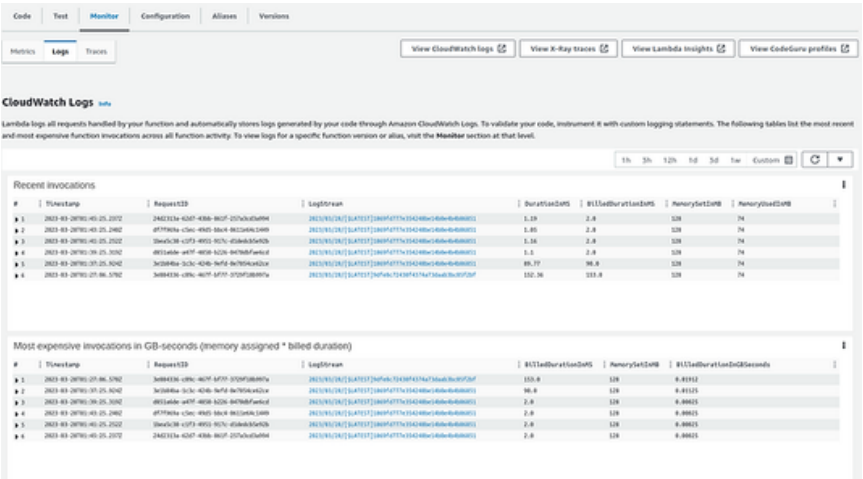
Vá e acesse o console da AWS para ver a execução do Lambda que acabamos de criar.



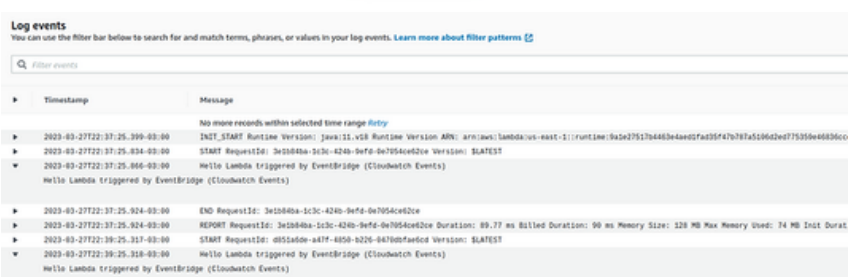
Acesse a aba **monitor**



Acesse a aba **Logs** dentro da seção **Monitor**



Veja as mensagens abaixo que serão impressas a cada 2 minutos.



Destroy

As cobranças de faturamento da AWS ocorrerão se você não destruir esses recursos. Portanto, recomendo destruí-los evitando algumas cobranças desnecessárias. Para evitar, execute o comando abaixo.

terraform destroy -var-file=development/vars.tfvars

Lembre-se que você precisa confirmar essa operação, ok?

Conclusão

Neste tutorial, criamos um AWS Lambda provisionado através do Terraform. Lambda é um serviço da AWS que podemos usar para diferentes casos de uso trazendo facilidade para compor uma arquitetura de software. Pudemos notar que o Terraform traz flexibilidade criando recursos para diferentes serviços em nuvem e fácil de implementar em projetos de softwares.

Repositório

Encontre mais detalhes neste repositório no Github

https://github.com/coffeeandtips-tech/aws-lambda-terraform-java

CURTIU? ACESSE MAIS TUTORIAIS COMO ESSE EM COFFEEANDTIPS.COM

E NOS SIGA NO INSTAGRAM

Coffee and Tips

Tech Tutorials