



# Curso Programação Backend com Python

DynamoDB e Lambda com python

© Cleuton Sampaio 2021

## Sumário

DynamoDB	<b>3</b>
Modelagem no DynamoDB	6
Padrões de consulta	6
AWS Lambda	<b>8</b>
Criando um script lambda em python	9
Invocando externamente	12

## DynamoDB

**Atenção:** O uso de serviços em nuvem, especialmente da AWS (Amazon Web Services) representa custo e você poderá receber cobranças pelo uso deles, mesmo que esteja em modalidade gratuita! Se não se sentir à vontade com isso, não faça os exercícios, apenas acompanhe.

Por favor, leia o parágrafo anterior, em vermelho mais umas duas vezes, ok? A responsabilidade pelos custos gerados ao tentar fazer os exercícios é sua e somente sua e acredite: Negligenciar isso pode gerar altíssimas cobranças em seu cartão de crédito!

É possível instalar uma versão local do DynamoDB para testes, mas eu não farei isso aqui, neste curso. Se quiser instalar uma versão local, estará livre de cobranças:

<https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/DynamoDBLocal.html>

Esta explicação sobre o DynamoDB é apenas um resumo, para que você entenda o que estou fazendo neste exemplo. Não é intenção deste curso esgotar o assunto e há outros cursos, inclusive meus sobre isto.

### Tabela

O DynamoDB é um serviço serverless, o que significa que você não cria uma instância dele em uma VPC, como faria com o EC2. Você precisa selecionar uma região onde vai criar sua tabela, que é a unidade dele. Você cria tabelas e elas podem conter dados heterogêneos.

#### Tabela “carros”:

```
{  
    "placa" : "xpto215",  
    "marca": "vw",
```

```
"modelo": "up",
"cor": "preto",
"ano": "2019"
}
{
  "placa" : "abcd415",
  "marca": "gm",
  "modelo": "spin",
  "cor": "branco",
  "ano": "2020",
  "obs": "Batido na trazeira"
}
```

Notou algo interessante? Na tabela “carros”, todos os itens (registros) devem ter o mesmo atributo de chave, que é a “placa”, mas cada item pode ter um esquema diferente. Por exemplo, o segundo item tem um atributo “obs”, ausente no primeiro.

Toda tabela tem que ter uma **partition key** (chave primária, utilizada para hash ou cálculo da posição do item na tabela física), que pode ser simples, como na tabela “carros” ou composta por uma **partition key** e uma **sort key**. A chave primária não pode ser duplicada! Se a chave primária for simples, não pode haver duplicata, por exemplo, não pode haver dois carros com mesma placa. Mas chaves compostas podem ter **partition key** duplicadas, desde que a **sort key** seja diferente. Vejamos um exemplo:

#### Tabela proprietários:

```
{
  "cpf" : "11122233344",
  "placa": "xpto201",
  "aquisicao": 2019
}
{
  "cpf" : "11122233344",
  "placa": "abcd5555",

```

```
"aquisicao": 2019  
}
```

Nesta tabela “proprietários” vemos dois itens. A chave primária é composta por uma partition key (“cpf”) e uma sort key (“placa”). Note que um proprietário tem 2 carros diferentes.

## Índice secundário

Além do índice da partition key (simples ou composta), você pode criar índices secundários:

- **Global secondary index:** Um índice com partition key e sort key diferentes das da tabela;
- **Local secondary index:** Um índice com a mesma partition key da tabela, mas com outra sort key;

Índices têm custo. Cuidado.

## Capacity mode / Modo de capacidade

O desempenho da sua tabela e a maneira como você é cobrado dependem do modo de capacidade que escolher:

- **On demand:** Você pode servir milhares de requisições simultâneas sem necessidade de planejamento prévio, mas será cobrado proporcionalmente;
- **Provisioned:** Você determina a quantidade de reads e writes por segundo e paga apenas por isso, mas pode ter um desempenho ruim, em caso de picos de acesso;

Não vou entrar em detalhes aqui, mas, como vamos usar o nível gratuito (Free tier), vamos usar o default, que é provisionado. O nível gratuito permite:

- 25 unidades de capacidade de gravação (WCU) provisionada
- 25 unidades de capacidade de leitura (RCU) provisionada

Suficiente para processar até 200 milhões de solicitações por mês por tabela.

## Modelagem no DynamoDB

Como muitos bancos NoSQL, o DynamoDB foi criado pensando em desempenho específico. Se modelado e utilizado desta maneira, o desempenho será excelente a um baixo custo. Fora desta maneira, o desempenho e o custo podem ser ruins.

Entenda que o DynamoDB foi criado para ser um banco eficiente para backend de aplicações, enquanto que um banco Relacional é voltado para flexibilidade de queries e para ser um repositório estruturado dos dados.

Tendo isso em mente, modelar um banco relacional é relativamente conhecido e simples, bastando, na maioria dos casos, normalizá-lo. Mas esta abordagem não é a melhor para um banco NoSQL, em especial o DynamoDB.

### Padrões de consulta

No DynamoDB você é cobrado por acesso, tráfego e volume armazenado, para cada tabela. E, devido à sua estrutura sem esquema, não dá para fazer consultas flexíveis sem comprometer o desempenho e o custo. É um banco de dados voltado para operação eficiente do seu negócio e não flexibilidade de armazenamento.

Antes de modelar seu banco DynamoDB, você deve conhecer bem os casos de uso que pretende desenvolver. Quais são os padrões de acesso e os dados que são consumidos juntos. Normalizar tabelas não é uma boa opção.

Alguns conselhos úteis são:

- Crie o mínimo de tabelas possível, organizando os dados de acordo com o caso de uso;
- Saiba o tamanho dos dados para particioná-los apropriadamente em tabelas;
- Armazene como serão consultados. Ao contrário de um banco relacional, não é prático mudar o formato da resposta a uma consulta, então, é melhor desenhar suas tabelas da maneira como serão consultadas;
- Dados relacionados devem estar juntos na mesma tabela;
- Use a ordem de classificação corretamente para os dados;
- Use índices globais.

Eu tenho grande experiência com bancos NoSQL, como: MongoDB, Redis e DynamoDB e posso dizer com segurança que você errará nas suas primeiras modelagens. No caso do MongoDB e do Redis, como eram locais, o custo se refletiu em queda de performance e necessidade de criar mais instâncias e particionamentos. No caso do DynamoDB, além da queda de performance, vai refletir em seu cartão de crédito também.

Vamos tentar modelar o problema dos carros e proprietários, mostrado anteriormente. Como faremos isso? Qual é a melhor maneira de modelar isso? Depende do nosso caso de uso... Vamos ver:

- Um estacionamento:
  - Registrar entradas e saídas de carros: 80% do tempo;
  - Contabilizar tempo estacionado: 15%;
  - Gerar cobrança ao proprietário: 5%;

É uma estimativa. 80% das queries são para registrar entrada ou saída de carros, 15% para contabilizar o tempo estacionado (uma vez ao dia) e 5% para gerar cobranças (uma vez ao mês).

A maioria das vezes queremos os dados dos carros independentemente do proprietário, portanto, os dados do “proprietário” são utilizados apenas em 5% das consultas. É claro que podemos modelar como uma só tabela: cpf + placa, mas note que, na maioria das vezes, queremos apenas saber a placa e não o cpf. Melhor seria modelar tendo a placa como partition key e criar um global index por cpf.

Outro detalhe é que juntar os dados na mesma tabela nos fará mudar o formato da resposta, já que nem sempre os dados do proprietário são necessários, o que também aumentaria o custo do tráfego de dados.

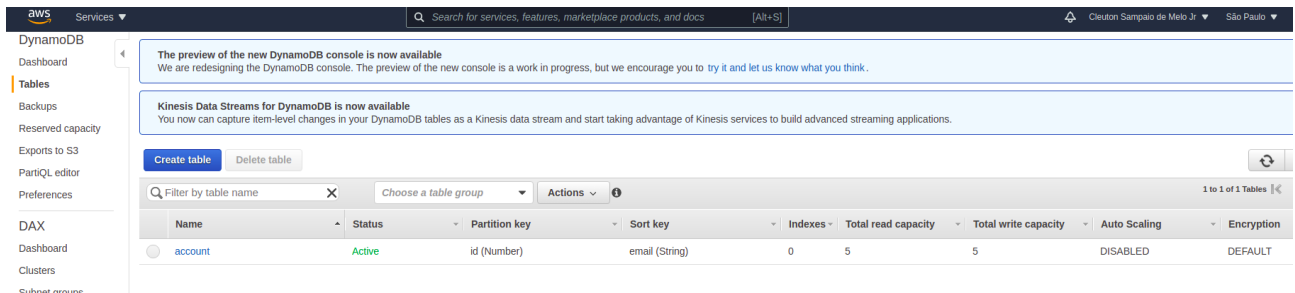
Agora vejamos outro caso de uso: Um sistema de controle de veículos:

- Quais são os veículos de um proprietário;
- Dado um veículo, saber quem é o proprietário;

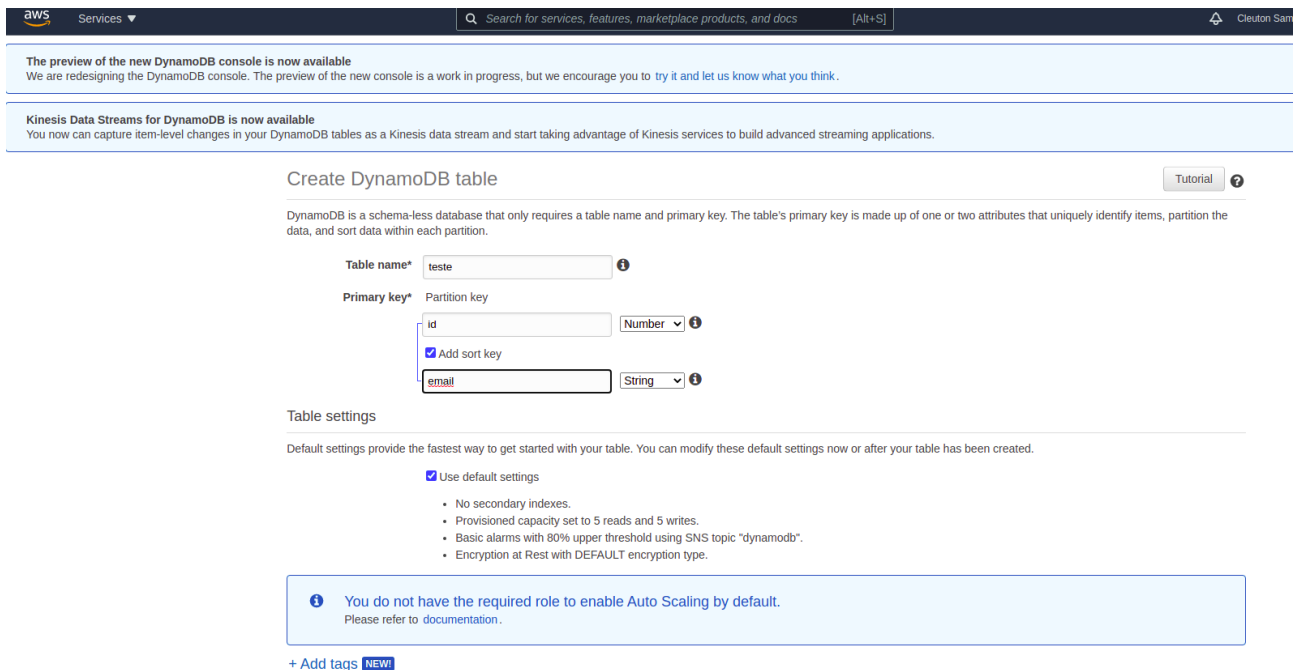
Neste caso, faz sentido modelarmos tudo junto, tendo cpf + placa como chave primária.

## Criando uma tabela de teste

Acesse a console AWS (<https://console.aws.amazon.com>) e selecione o serviço DynamoDB.

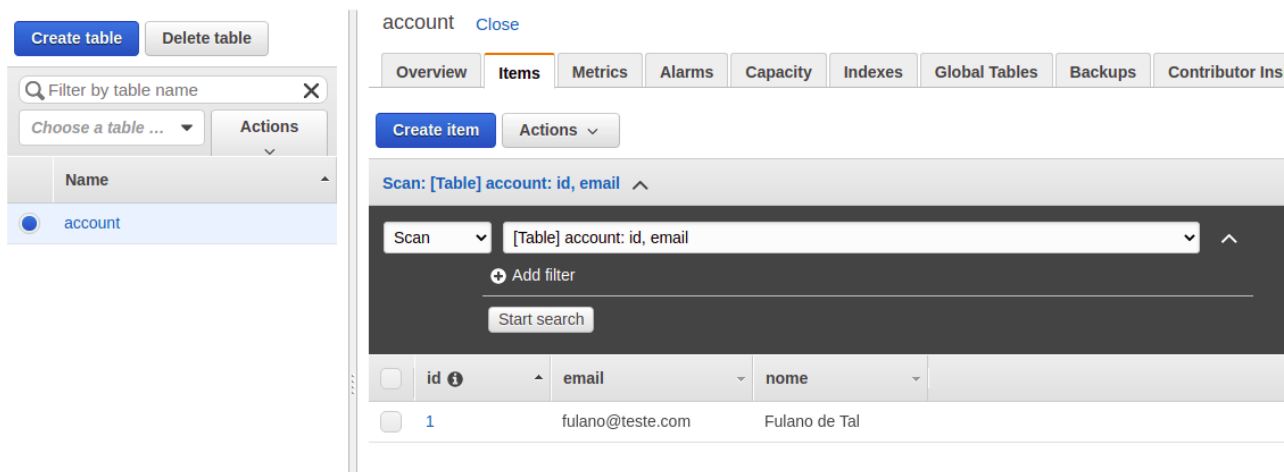


Crie uma nova tabela (a minha já foi criada) usando ID (numérico) e Email (String) como chave primária:

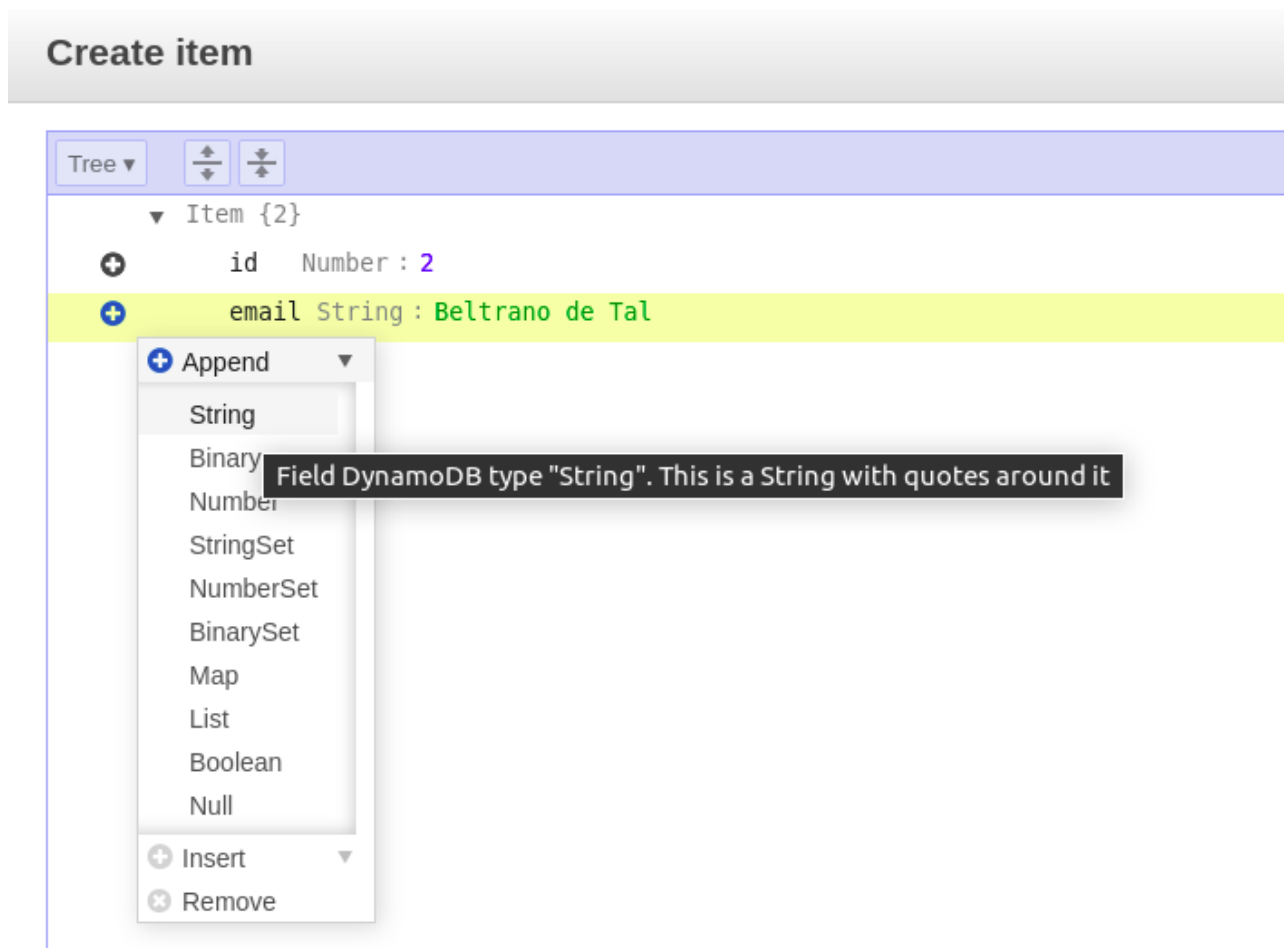


Depois, insira um ou mais itens na tabela:





Para inserir mais campos, clique no símbolo “+”:



## AWS Lambda

***Atenção: O uso de serviços em nuvem, especialmente da AWS (Amazon Web Services) representa custo e você poderá receber cobranças pelo uso deles, mesmo que esteja em modalidade gratuita! Se não se sentir à vontade com isso, não faça os exercícios, apenas acompanhe.***

**Por favor, leia o parágrafo anterior, em vermelho mais umas duas vezes, ok? A responsabilidade pelos custos gerados ao tentar fazer os exercícios é sua e somente sua e acredite: Negligenciar isso pode gerar altíssimas cobranças em seu cartão de crédito!**

Já ouviu falar em **serverless**, certo? E em **FaaS** (Function as a Service)? É uma maneira de implantar seu código fonte como um serviço, sem necessidade de provisionar infraestrutura física ou virtual nem se preocupar com reserva de recursos.

O Lambda oferece a mesma praticidade e simplicidade que o DynamoDB, já que é um serviço serverless, e permite disponibilizarmos soluções rapidamente, com boa performance. Assim, podemos nos concentrar na funcionalidade que desejamos servir e não na infraestrutura necessária para isso.

Lambda é uma ótima alternativa para serviços **on demand**, para os quais você não tem um tráfego constante. Se você tem uma média relativamente constante de tráfego em vários momentos do dia, talvez seja melhor utilizar um VPS, como o AWS EC2 - Elastic Compute Cloud. Assim, você reserva um servidor virtual pagando por período de tempo, independentemente da utilização.

Lambda tem se tornado excelente alternativa como backend mobile ou web, sendo utilizado em conjunto com o DynamoDB e o AWS API Gateway, para disponibilizar e controlar acesso à sua função Lambda.

Neste curso veremos apenas o AWS Lambda e o DynamoDB.

## Criando um script lambda em python

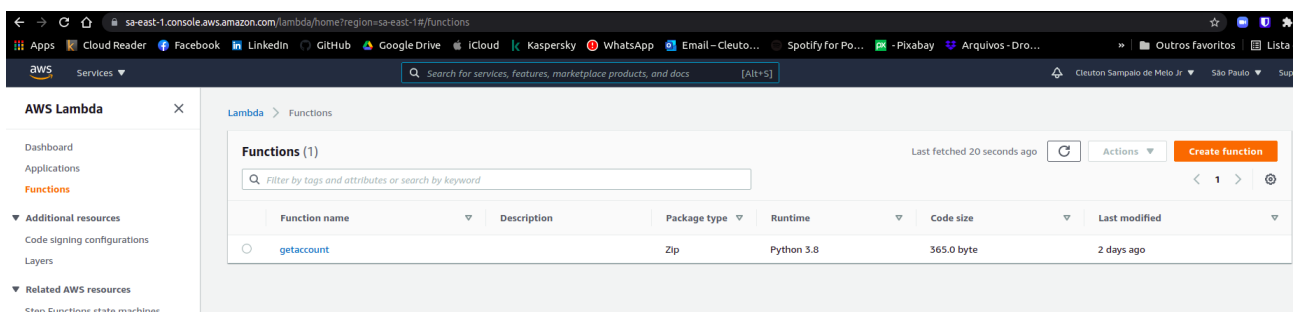
Usando a console do AWS Lambda, você pode criar uma função python, o que veremos mais adiante. Uma função lambda é apenas isso:

```
def lambda_handler(event, context):  
    message = 'Oi {} {}'.format(event['nome'], event['sobrenome'])  
    return {  
        'statusCode': 200,  
        'body': json.dumps(message)  
    }
```

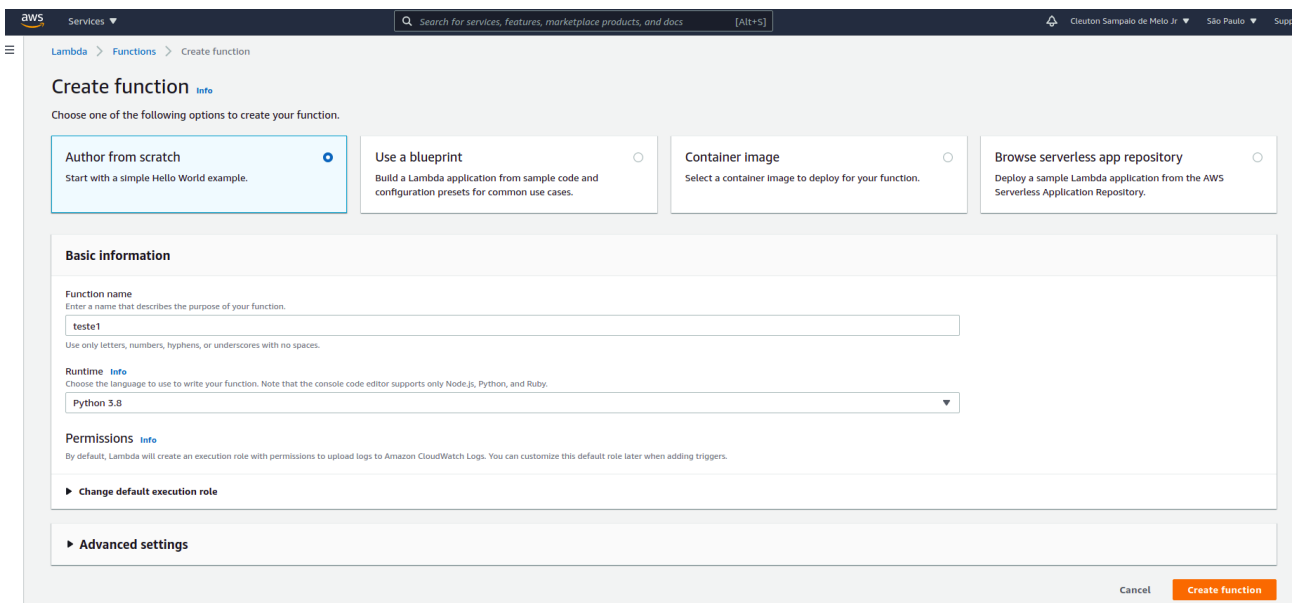
Precisamos de uma função handler, que recebe a chamada do Lambda. Essa é a função principal do seu programa. Ela recebe uma instância de evento (**event**) que contém todas os atributos JSON passados a ela, e um contexto, que tem várias informações.

Precisamos retornar um objeto JSON, então utilizamos o objeto **json** e o método **dumps()** para transformar a mensagem em um string Json. Pronto! Só isso! Sem anotações, sem preocupação com threads... Nada disso.

Para criar esta função usaremos a console AWS (<https://console.aws.amazon.com>). Basta usar sua conta, abrir a console e buscar a console do serviço AWS Lambda:



Estas são as funções que já foram criadas. Podemos selecionar uma delas, editar ou apagar. Ou podemos criar nova função clicando no botão “Create function”.



**Create function** [Info](#)

Choose one of the following options to create your function.

- Author from scratch** ☒ Start with a simple Hello World example.
- Use a blueprint** ☐ Build a Lambda application from sample code and configuration presets for common use cases.
- Container image** ☐ Select a container image to deploy for your function.
- Browse serverless app repository** ☐ Deploy a sample Lambda application from the AWS Serverless Application Repository.

**Basic information**

**Function name**  
Enter a name that describes the purpose of your function.  
  
Use only letters, numbers, hyphens, or underscores with no spaces.

**Runtime** [Info](#)  
Choose the language to use to write your function. Note that the console code editor supports only Node.js, Python, and Ruby.

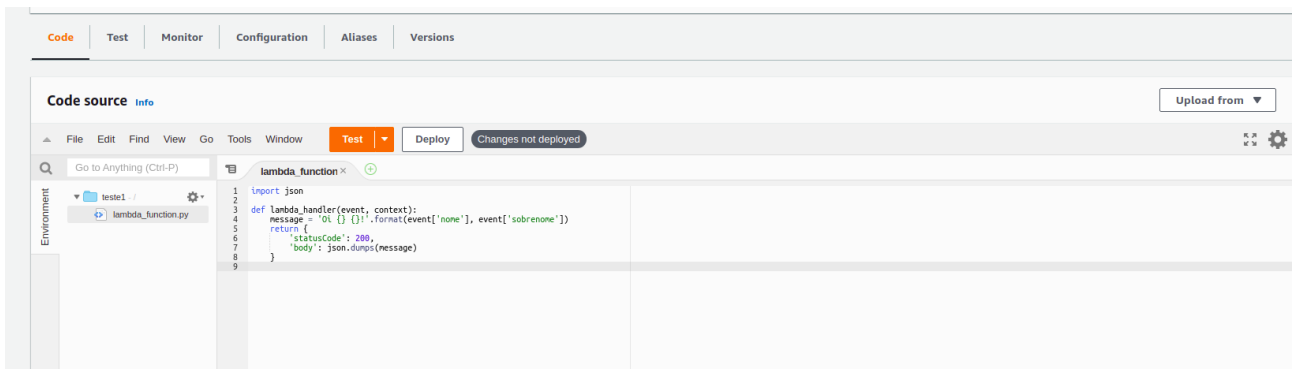
**Permissions** [Info](#)  
By default, Lambda will create an execution role with permissions to upload logs to Amazon CloudWatch Logs. You can customize this default role later when adding triggers.

► **Change default execution role**

► **Advanced settings**

[Cancel](#) [Create function](#)

Precisamos informar o nome da função, selecionar o runtime (linguagem), as permissões (neste caso deixaremos as defaults) e mandamos criar a função. Depois, teremos que informar o código-fonte:



**Code source** [Info](#) [Upload from](#)

File Edit Find View Go Tools Window **Test** Deploy Changes not deployed

Go to Anything (Ctrl-P)

Environment

- teste1
- lambda\_function.py

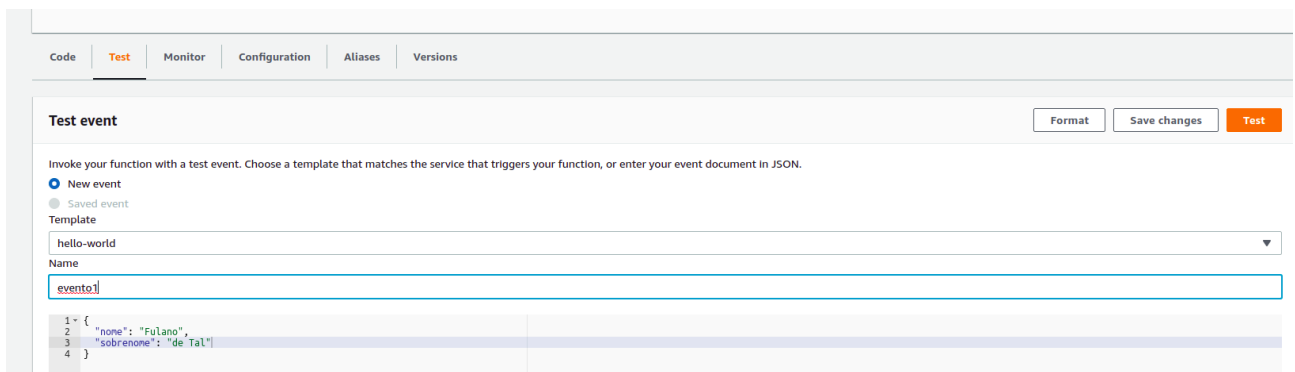
```
1 import json
2
3 def lambda_handler(event, context):
4     message = 'Oi {} {}'.format(event['nome'], event['sobrenome'])
5     return {
6         'statusCode': 200,
7         'body': json.dumps(message)
8     }
9
```

Temos várias abas aqui. As mais importantes são: “Code”, para editar o código e instalar (botão “Deploy”) e “Test”, para criarmos eventos de teste e invocarmos nossa função. Eu fiz uma pequena alteração no código gerado para mim, de modo a demonstrar como obter propriedades do objeto passado para a função:

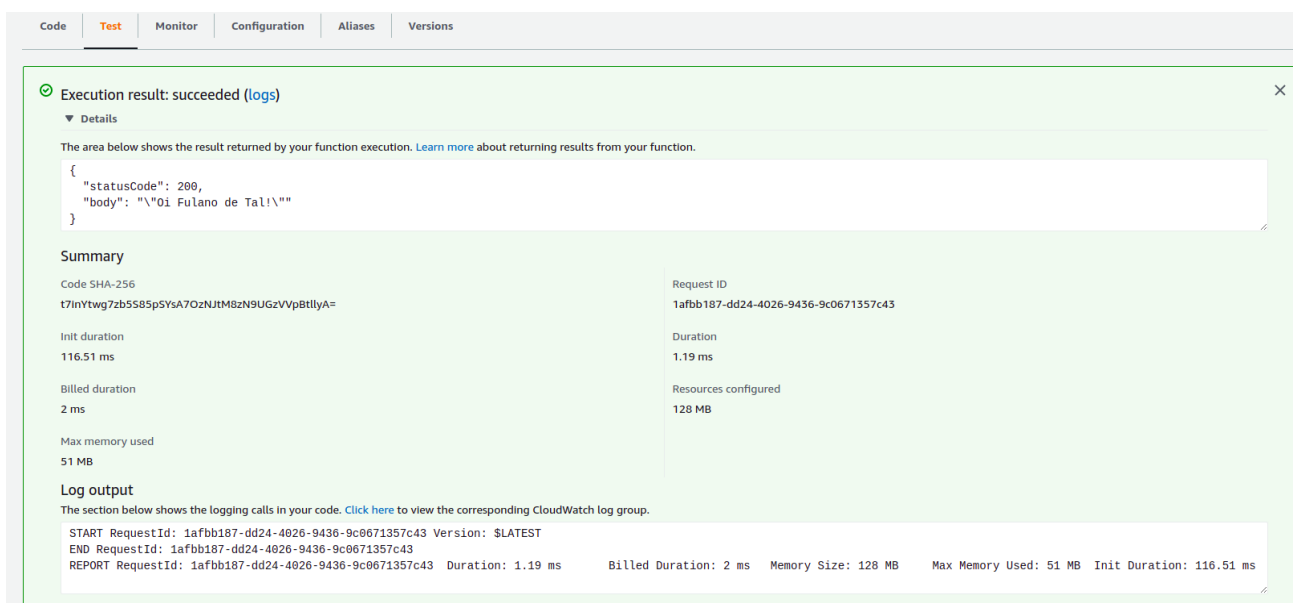
```
message = 'Oi {} {}'.format(event['nome'], event['sobrenome'])
```

Não se esqueça de clicar no botão “Deploy” após cada alteração no código-fonte.

Ao clicar na aba “Test” podemos criar um evento de teste com um objeto JSON que será enviado:



Ao clicarmos no botão “Test” uma invocação é disparada e podemos ver o resultado:



Pronto! Uma função Lambda em python funcionando!

## Invocando externamente

Você pode invocar essa função externamente, a partir de um código rodando em um servidor on premises ou dentro de um EC2/ECS. Não é objetivo desse curso explicar isso em detalhes, mas vamos ver as maneiras de fazer. O recomendado é usar o serviço API Gateway da AWS para publicar, controlar e acessar sua função Lambda, mas isso está fora do escopo deste curso.

### Invocando com a AWS CLI (Command Line Interface):

Se você instalar o AWS CLI

(<https://docs.aws.amazon.com/cli/latest/userguide/cli-chap-install.html>) basta utilizar o comando:

```
aws lambda invoke --function-name my-function --payload '{ "key":
"value" }' response.json
```

### Invocando com o boto3:

Se você instalou a biblioteca **boto3** para python (pip install boto3) pode invocar funções lambda de dentro de seu código-fonte:

```
response = client.invoke(
    FunctionName='<nome_da_função>',
    InvocationType='RequestResponse',
    LogType='None',
    ClientContext='',
    Payload=b'<JSON>',
    Qualifier='versão'
)
```

O usuário que configurar no cliente boto3 terá que possuir permissão para invocar a função. E também terá que informar access key do AWS IAM:

[https://docs.aws.amazon.com/IAM/latest/UserGuide/id\\_credentials\\_access-keys.html](https://docs.aws.amazon.com/IAM/latest/UserGuide/id_credentials_access-keys.html)

## Lambda e DynamoDB

Podemos criar funções lambda que interajam com tabelas DynamoDB. E é o que veremos agora. Primeiramente, vejamos o acesso mais básico a uma tabela DynamoDB que seria listar a tabela inteira (table scan). Veja este código:

```
import json
import boto3

client = boto3.client('dynamodb')

def lambda_handler(event, context):
    data = client.scan(
        TableName='account'
    )

    response = {
        'statusCode': 200,
        'body': json.dumps(data),
        'headers': {
            'Content-Type': 'application/json',
            'Access-Control-Allow-Origin': '*'
        },
    }

    return response
```

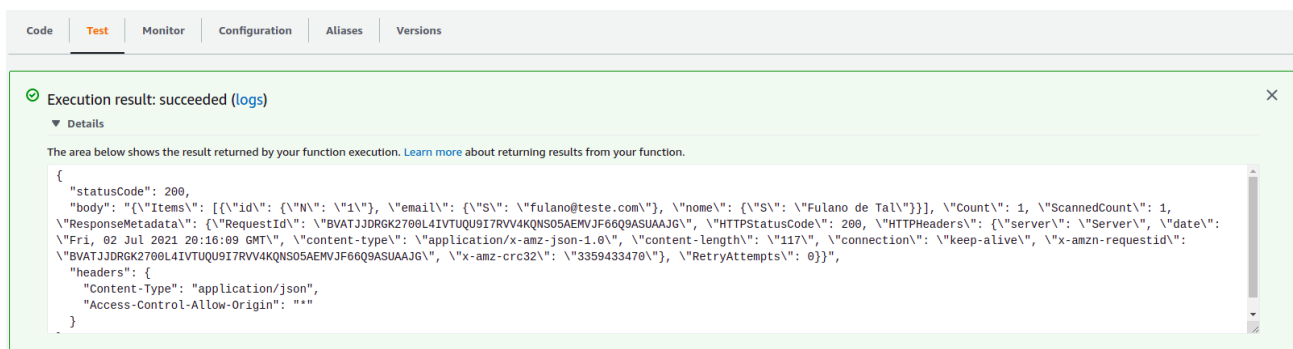
Ele foi feito para ser executado dentro do AWS Lambda e utiliza a biblioteca boto3 (pré-instalada pelo AWS Lambda) para acessar o DynamoDB. Primeiro, criamos uma instância de **cliente** lambda:

```
client = boto3.client('dynamodb')
```

Depois executamos uma operação de table scan:

```
data = client.scan(  
    TableName='account'  
)
```

E montamos a resposta. Pronto! Simples assim. Vejamos o teste dessa função na AWS Console:



Se você criou aquela tabela de teste (account) então é fácil acessá-la.

## Policies e Roles

Para que uma função Lambda possa fazer qualquer coisa com o DynamoDB, é preciso criar uma Role (papel) contendo uma Policy (permissão) e anexá-la à função Lambda.

Não vou entrar em detalhes sobre policies aqui. Existe uma policy genérica que é AmazonDynamoDBFullAccess, dando direito total de acesso ao DynamoDB. É claro que você não utilizará isso em produção, criando sua própria policy, mas, para simplificar as coisas, é assim que faremos aqui.

Para criar uma Role que permita à nossa função Lambda acessar o DynamoDB, precisamos abrir o serviço **IAM** - Identity and Access Management, na console aws.

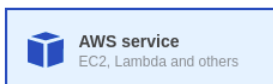


Criamos uma Role com o nome “lambda\_dynamodb” selecionando “AWS Service” para indicar que é uma Role de serviço para serviço. Selecione o caso de uso Lambda, para informar que aplicará a ele:

### Create role

1 2 3 4

#### Select type of trusted entity



**AWS service**  
EC2, Lambda and others

**Another AWS account**  
Belonging to you or 3rd party

**Web identity**  
Cognito or any OpenID provider

**SAML 2.0 federation**  
Your corporate directory

Allows AWS services to perform actions on your behalf. [Learn more](#)

#### Choose a use case

##### Common use cases

###### EC2

Allows EC2 instances to call AWS services on your behalf.

###### Lambda

Allows Lambda functions to call AWS services on your behalf.

Or select a service to view its use cases

No momento de selecionar as permissões, indique a policy:  
AmazonDynamoDBFullAccess:

The screenshot shows the AWS IAM console interface. On the left is a navigation menu with 'Identity and Access Management (IAM)' selected. The main content area shows the configuration for the 'lambda\_dynamodb' role. A summary table lists the role's details, and the 'Permissions' tab is active, showing the 'AmazonDynamoDBFullAccess' policy attached.

Property	Value
Role ARN	arn:aws:iam::002744765652:role/lambda_dynamodb
Role description	Allows Lambda functions to call AWS services on your behalf.   <a href="#">Edit</a>
Instance Profile ARNs	/
Path	/
Creation time	2021-06-30 17:06 UTC-0300
Last activity	2021-06-30 17:33 UTC-0300 (Yesterday)
Maximum session duration	1 hour <a href="#">Edit</a>

**Permissions** | Trust relationships | Tags | Access Advisor | Revoke sessions

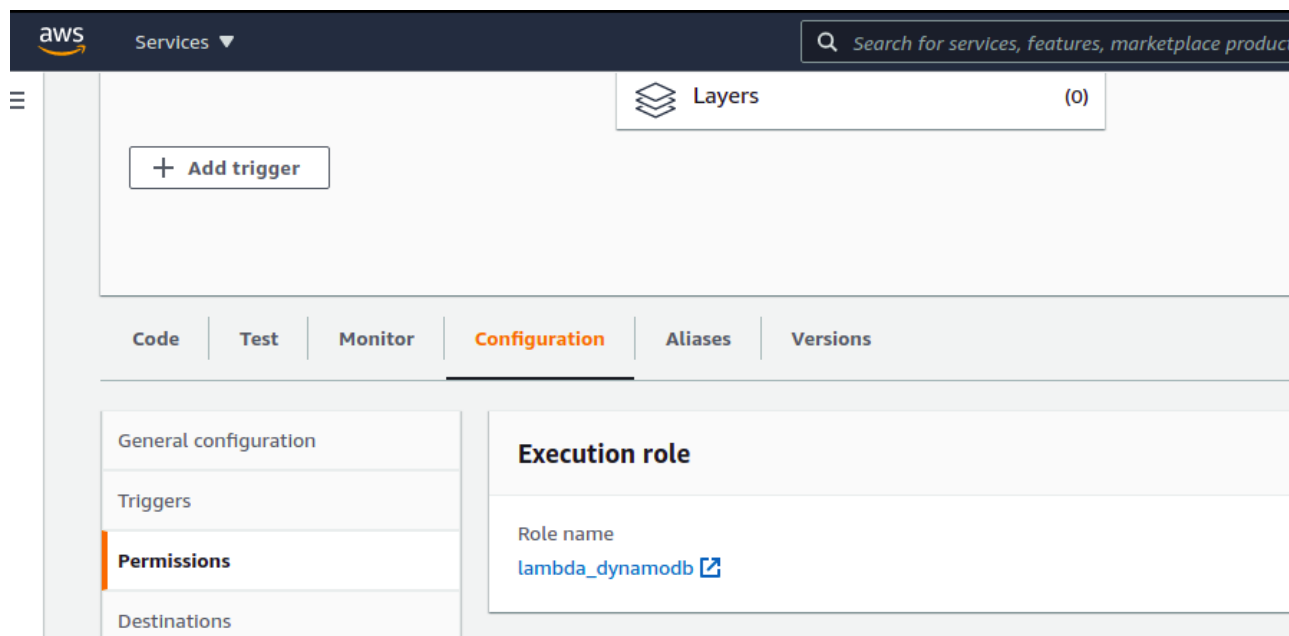
▼ Permissions policies (1 policy applied)

[Attach policies](#)

Policy name ▼

- AmazonDynamoDBFullAccess

Agora, ao criar sua função no serviço Lambda, indique qual é a permissão associando a Role criada:



Ao atribuir a Role de execução “lambda\_dynamodb” você está dando direito à sua função de acessar qualquer tabela sua. Mais uma vez, para produção é melhor criar uma policy mais restritiva (não é objetivo deste curso).

## Operações CRUD

Podemos realizar diversas operações CRUD em uma função lambda. Vejamos uma por uma.

### Criar registros

Podemos criar uma instância representando a tabela com o método “Table()” e usar seus métodos para operações CRUD, como o “put\_item” que cria um novo item:

```
import boto3
from botocore.exceptions import ClientError

dynamodb = boto3.resource('dynamodb')

def lambda_handler(event, context):
    id = event['id']
    email = event['email']
    nome = event['nome']
    table = dynamodb.Table('account')
    response = table.put_item(
        Item={
            'id': id,
            'email': email,
            'nome': nome
        }
    )
    return response
```

### Ler um registro pela chave

Podemos utilizar o método **get\_item()** passando um JSON com a chave:

```
import boto3
from botocore.exceptions import ClientError

dynamodb = boto3.resource('dynamodb')

def lambda_handler(event, context):
    table = dynamodb.Table('account')
    try:
        response = table.get_item(Key={'id': event['id'],
            'email': event['email']})
```

```
except ClientError as e:
    print(e.response['Error']['Message'])
else:
    return response['Item']
```

Você pode retornar algum objeto de erro com o código apropriado, por exemplo:

```
return {
    "statusCode": 404,
    "headers": {
        "Content-Type": "application/json"
    },
    "body": json.dumps({
        "status": "Nao encontrado "
    })
}
```

### Atualizando um item

Podemos passar um objeto json com a chave e os campos que queremos atualizar dentro do evento:

```
import boto3
from botocore.exceptions import ClientError
```

```
dynamodb = boto3.resource('dynamodb')
```

```
def lambda_handler(event, context):
    table = dynamodb.Table('account')
    nome = event['nome']
    id = event['id']
    email = event['email']
    response = table.update_item(
        Key={
            'id': id,
```

```
        'email': email
    },
    UpdateExpression="set nome=:n",
    ExpressionAttributeValues={
        ':n': nome
    },
    ReturnValues="UPDATED_NEW"
)
return response
```

Os argumentos **UpdateExpression** e **ExpressionAttributeValues**, passados para o método **update\_item**, são relacionados e indicam os valores a serem substituídos no item.

### Apagando um item

É fácil apagar um item:

```
import boto3
from botocore.exceptions import ClientError

dynamodb = boto3.resource('dynamodb')

def lambda_handler(event, context):
    table = dynamodb.Table('account')
    nome = event['nome']
    id = event['id']
    email = event['email']
    try:
        response = table.delete_item(
            Key={
                'id': id,
                'email': email
            }
        )
```

```
)  
except ClientError as e:  
    if e.response['Error']['Code'] ==  
"ConditionalCheckFailedException":  
        print(e.response['Error']['Message'])  
    else:  
        raise  
else:  
    return response
```