

Lambda を使って開発してみよう

IDE の基本操作編

開発環境をセットアップしよう（事前準備）

今回ローカル環境で開発するので各自の端末に以下をセットアップ

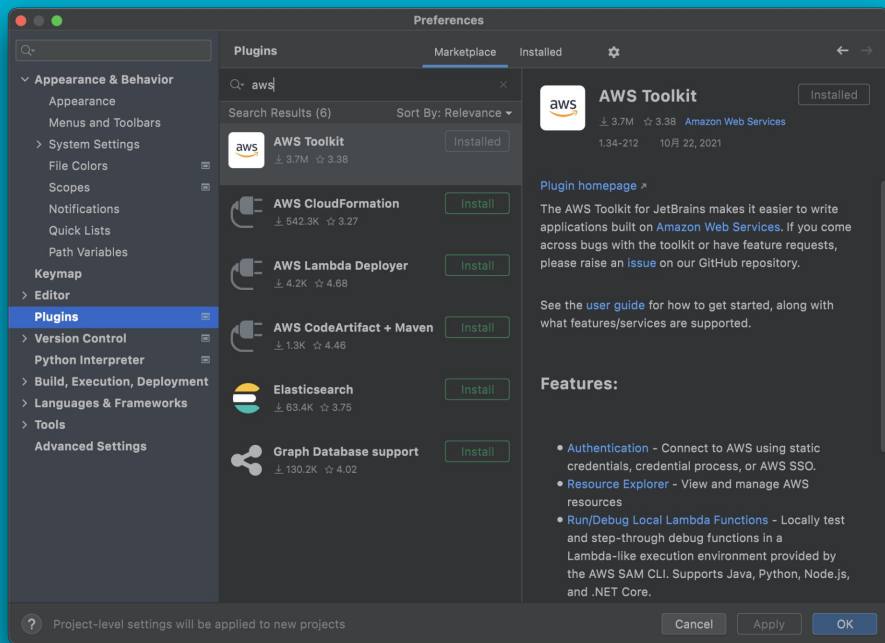
- pyenv \geq 2.2.0
- Python \geq 3.9.6
- ~/.aws/config（AWS CLI で aws configure コマンドを実施）
- PyCharm Community Edition
 - [ダウンロード PyCharm](#)
- SAM CLI \geq 1.28.0
 - [AWS SAM CLI のインストール](#)
- Docker Desktop
 - [Docker Desktop for Mac and Windows](#)

```
$ pyenv --version  
pyenv 2.2.0  
$ python -V  
Python 3.9.6
```



AWS Toolkit for PyCharm のインストール

PyCharm メニューの [Preferences] > [Plugins] > 「aws」で検索 > [Install]



参考: [AWS Toolkit for PyCharm](#)

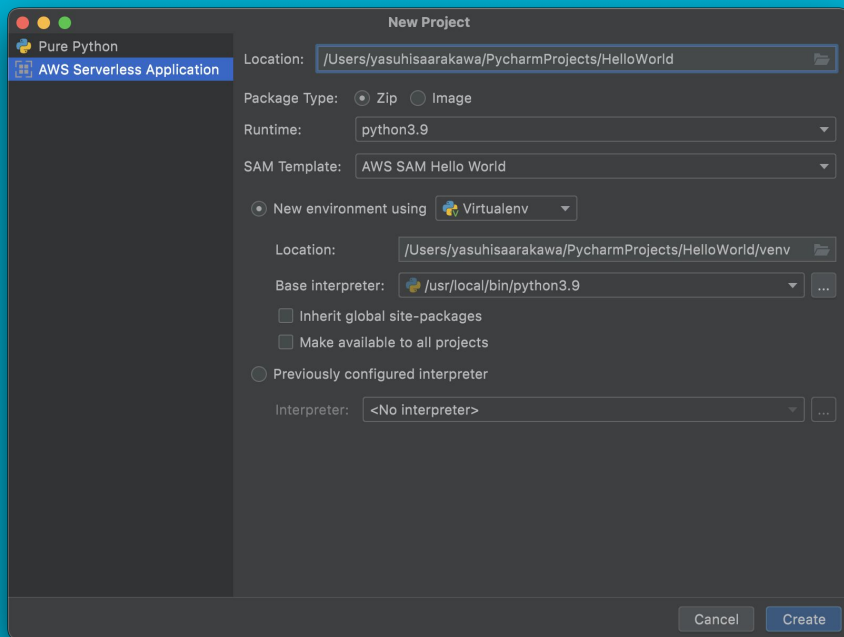
AWS SAM Application プロジェクトを作成

の前にまずは用語をおさえよう

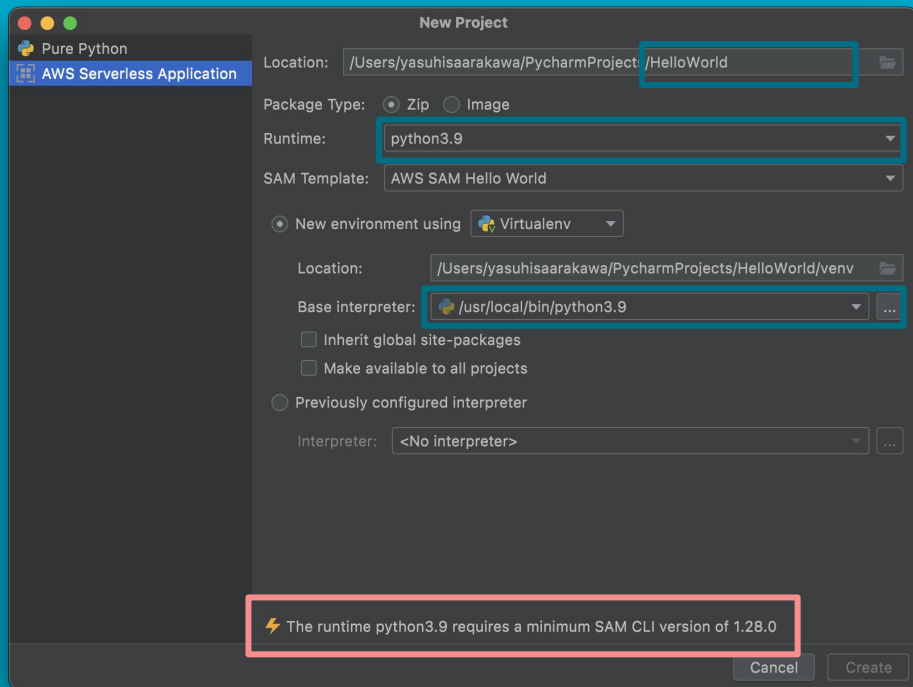
- **SAM**: Serverless Application Model
 - Lambda を動かす環境をまるっと作ってくれる AWS 提供のテンプレート
 - 中身は CloudFormation の拡張
- **IDE**: Integrated Development Environment, 統合開発環境
 - 開発規模に関わらず基本は IDE を使って開発
 - 最近はわりと VSCode でもいけちゃう
- **プロジェクト**: IDE で管理するアプリケーション（プログラムを束ねて動作するもの）の単位
 - プロジェクトマネジメントのプロジェクトとは違う

AWS SAM Application プロジェクトを作成

[New Project (Projects)] > [AWS Serverless Application] > 次頁へ続く



AWS SAM Application プロジェクトを作成



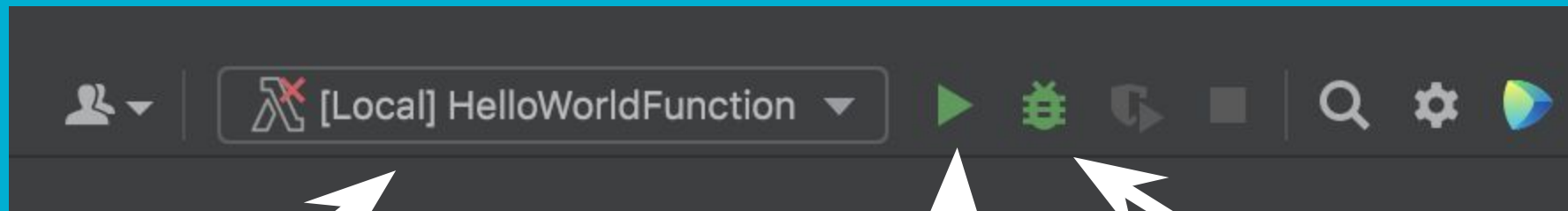
1. Location: **untitled** → **HelloWorld**
2. Runtime: **python3.9**
3. Base interpreter: **python3.9**
(**python3** でも OK)
4. [Create] をクリック

SAM CLI をこのタイミングで
インストールしたら PyCharm を再起動

参考: [AWS SAM CLI のインストール](#)

AWS SAM Application プロジェクトを実行

実行の UI に関してはどの IDE も似てる。



実行のターゲットやデバッグの設定

実行 (Run)
大体再生ボタン

デバッグ (Debug)
大体虫ボタン

VSCode



Xcode



実行とデバッグの違い - 実行 (Run)



—
実行は書いたプログラムを動かす。Lambda でいう Invocation (呼び出し)。プログラムの途中の状態で止めることはできず、ひたすらコードの終わりまで処理が走り続ける。なので Run と呼ぶのかな。

実際に実行してみるとコンソールにこんな感じの出力メッセージが出て終わる

```
Mounting /Users/yasuhisaarakawa/PycharmProjects/HelloWorld/.aws-sam/build/HelloWorldFunc
START RequestId: 18c98339-05dd-4e5f-abe5-2e530465708b Version: $LATEST
END RequestId: 18c98339-05dd-4e5f-abe5-2e530465708b
{"statusCode": 200, "body": "{\"message\": \"hello world\"}"}REPORT RequestId: 18c98339-
```


実行とデバッグの違い - デバッグ (Debug)



デバッグもプログラムを実行するが、**ブレークポイント**でプログラムの処理を止めることができる。

実行中の変数の値の変化を監視 (**ウォッチ**) したり、関数内の処理を追跡 (**ステップイン**) したり、関数外だけ処理を追跡 (**ステップオーバー**) したりといった様々な操作がある。

バグ = bug: 昆虫、バグ取りでデバッグと呼ぶのかな。

一言で表すのが難しい。が操作に慣れると開発がかなりスムーズになる。

オペのイメージだと CloudWatch コンソール操作的な感じ。慣れたら早いでしょ。

ソースコードを見よう

HelloWorld > hello_world > app.py

```
36     return {  
37         "statusCode": 200,  
38         "body": json.dumps({  
39             "message": "hello world",  
40             # "location": ip.text.replace("\n", "")  
41         }),  
42     }
```

実行してみよう - Docker Desktop の起動

実行には事前に Docker Desktop の起動が必要



なぜ？

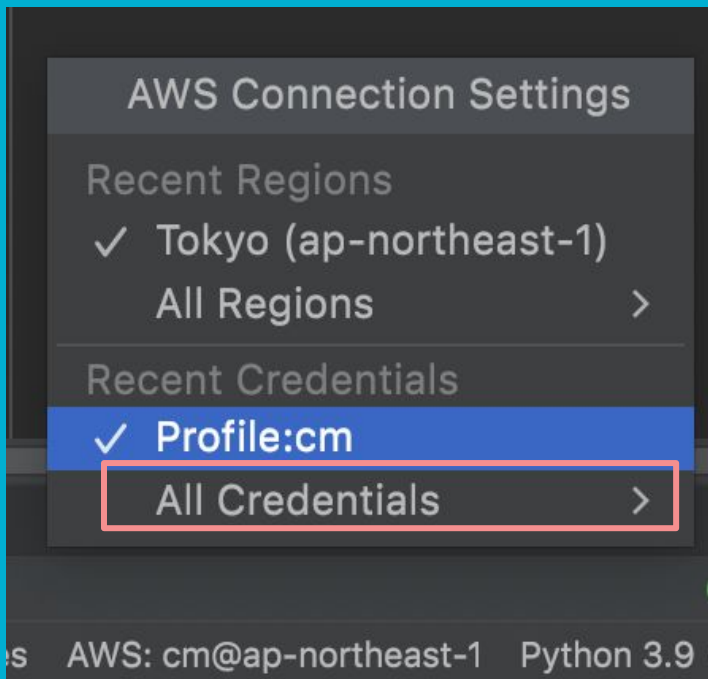
本来なら インターネット -> API Gateway -> Lambda とリクエストが実行される。Lambda の動作確認をするには実行環境を構築して、ブラウザなどからアクセスする必要あり。軽微な動作確認でも手間がかかっていた（昔話）

その問題を SAM Local が解決する。SAM Local がプログラムの実行時に仮想の Lambda 環境を Docker コンテナとして起動して仮想実行環境を用意してくれる。

SAM Local が Docker を使うから Docker の起動が必要。

参考: [sam local start-api](#)

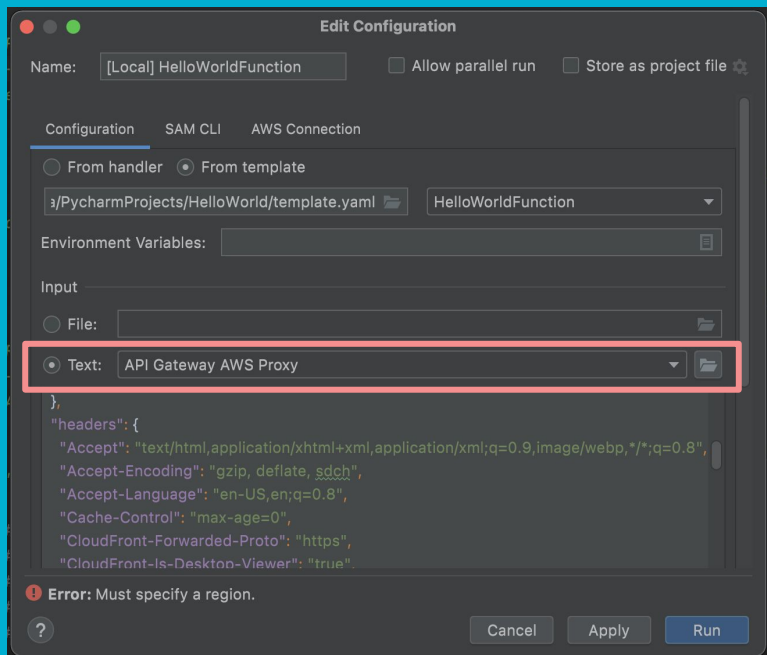
実行してみよう - クレデンシャルの初回設定



1. 画面右下 [AWS: No credentials selected]
2. Profile を適当に選ぶ
普段使っている環境のプロファイルで OK、IAM ユーザーのプロファイルのみ選択可能

実行してみよう - コンフィグの初期設定

[Local] HelloWorldFunction >



1. Input で Text: [API Gateway AWS Proxy] を選択
2. [Run] をクリック

実行してみよう - Run

```
Mounting /Users/yasuhisaarakawa/PycharmProjects/HelloWorld/.aws-sam/build/HelloWorldFunc
START RequestId: 18c98339-05dd-4e5f-abe5-2e530465708b Version: $LATEST
END RequestId: 18c98339-05dd-4e5f-abe5-2e530465708b
{"statusCode": 200, "body": "{\"message\": \"hello world\"}"}REPORT RequestId: 18c98339-
```

```
{
  "statusCode": 200,
  "body": {
    "message": "hello world"
  }
}
```

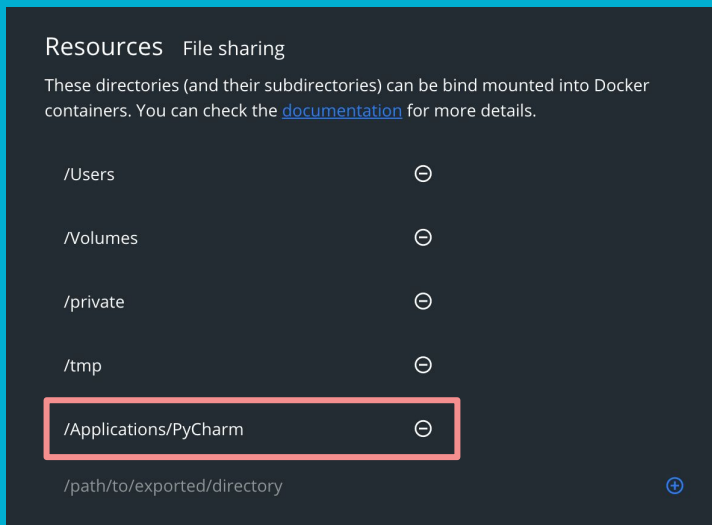
結果は JSON で出力される

- ステータスコード: 200 (OK)
- message: hello world

デバッグしてみよう - Docker File Sharing

デバッグ前に Docker へ PyCharm のファイルを共有する必要あり

Docker メニュー > [Preferences..] > [Resources] > [FILE SHARING]
> 末尾に /Applications/PyCharm を入力 (Mac) > [Apply & Restart]



デバッグしてみよう - ブレークポイント

ブレークポイントをはる（「張る」「貼る」どちらか不明だけど、はると言う）

app.py 36, 37 行目の行番号 (36, 37) の右側をクリックするとブレークポイント (赤丸) が出てくる

赤丸をクリックするとブレークポイントが解除される、解除したらまたはってね

```
35  
36 ● ○ return {  
37 ● ○     "statusCode": 200,  
38 ○     "body": json.dumps({  
39     "message": "hello world",  
40     # "location": ip.text.replace("\n", "")  
41     }},  
42     }  
43
```


デバッグしてみよう - ステップ

[Local] HelloWorldFunction >



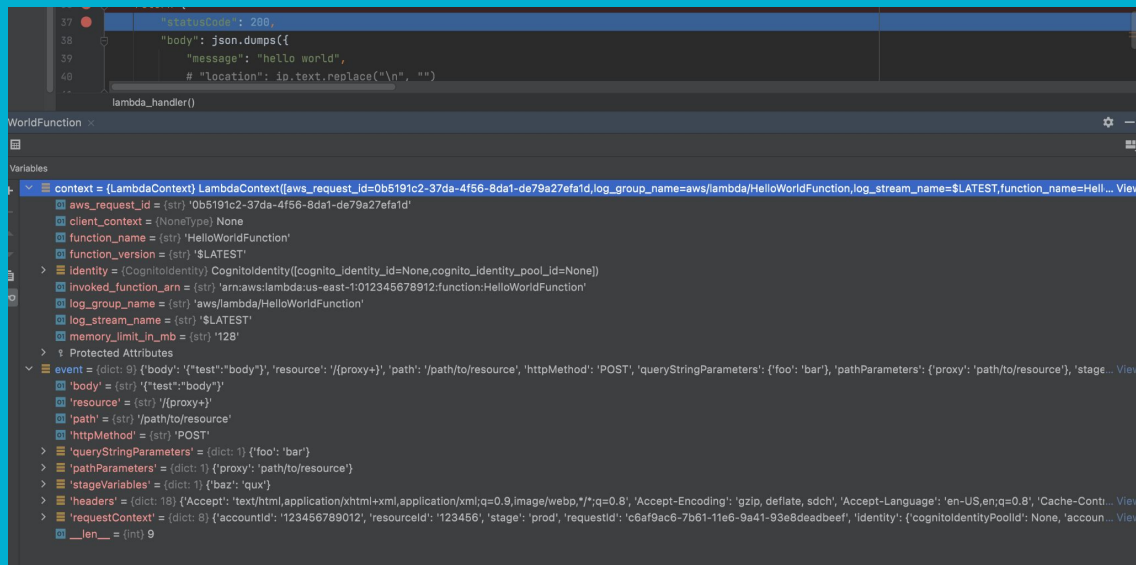
37 行目で止まるはず。止まった行から 1 行ずつ（ステップ）実行が可能。
36 行目は return なので、処理順は {} 内の処理が終わってから。
処理がある行のみブレークポイントは有効 → 空行やコメント行は無視

```
35  
36 ● return {  
37 ●   "statusCode": 200,  
38   "body": json.dumps({  
39     "message": "hello world",  
40     # "location": ip.text.replace("\n", "")  
41   }),  
42 }  
43
```

デバッグしてみよう - 式のウォッチ

デフォルトでデバッグコンソール下部に表示される context, event を開くと現在の変数の値が見れる



→ コードを変えてバグを直す、コードを変更したら 16p~ からやり直し



```
37     "statusCode": 200,
38     "body": json.dumps({
39         "message": "hello world",
40         # "location": ip.text.replace("\n", "")
    })
    lambda_handler()

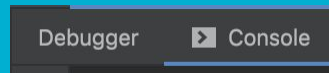
WorldFunction
Variables
context = (LambdaContext) LambdaContext([aws_request_id=0b5191c2-37da-4f56-8da1-de79a27efa1d,log_group_name=aws/lambda>HelloWorldFunction,log_stream_name=$LATEST,function_name=HelloWorldFunction])
  aws_request_id = (str) '0b5191c2-37da-4f56-8da1-de79a27efa1d'
  client_context = (NoneType) None
  function_name = (str) 'HelloWorldFunction'
  function_version = (str) '$LATEST'
  identity = (CognitoIdentity) CognitoIdentity([cognito_identity_id=None,cognito_identity_pool_id=None])
  invoked_function_arn = (str) 'arn:aws:lambda:us-east-1:012345678912:function:HelloWorldFunction'
  log_group_name = (str) 'aws/lambda>HelloWorldFunction'
  log_stream_name = (str) '$LATEST'
  memory_limit_in_mb = (str) '128'
  Protected Attributes
  event = (dict: 9) {'body': '{"test":"body"}', 'resource': '/{proxy+}', 'path': '/path/to/resource', 'httpMethod': 'POST', 'queryStringParameters': {'foo': 'bar'}, 'pathParameters': {'proxy': 'path/to/resource'}, 'stageVariables': {'baz': 'qux'}, 'headers': {'Accept': 'text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8'}, 'Accept-Encoding': 'gzip, deflate, sdch', 'Accept-Language': 'en-US,en;q=0.8', 'Cache-Control': 'no-cache', 'Request-Id': '123456789012', 'requestContext': {'accountId': '123456', 'resourceId': '123456', 'stage': 'prod', 'requestId': 'c6af9ac6-7b61-11e6-9a41-93e8d4debeef', 'identity': {'cognitoIdentityPoolId': None, 'accountId': None, 'principalId': None, 'provider': 'aws:iam'}, 'type': 'User', 'auth': 'Basic realm=example.com'}}
    body = (str) '{"test":"body"}'
    resource = (str) '/{proxy+}'
    path = (str) '/path/to/resource'
    httpMethod = (str) 'POST'
    queryStringParameters = (dict: 1) {'foo': 'bar'}
    pathParameters = (dict: 1) {'proxy': 'path/to/resource'}
    stageVariables = (dict: 1) {'baz': 'qux'}
    headers = (dict: 18) {'Accept': 'text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8'}, 'Accept-Encoding': 'gzip, deflate, sdch', 'Accept-Language': 'en-US,en;q=0.8', 'Cache-Control': 'no-cache', 'Request-Id': '123456789012', 'requestContext': {'accountId': '123456', 'resourceId': '123456', 'stage': 'prod', 'requestId': 'c6af9ac6-7b61-11e6-9a41-93e8d4debeef', 'identity': {'cognitoIdentityPoolId': None, 'accountId': None, 'principalId': None, 'provider': 'aws:iam'}, 'type': 'User', 'auth': 'Basic realm=example.com'}}
    requestContext = (dict: 8) {'accountId': '123456', 'resourceId': '123456', 'stage': 'prod', 'requestId': 'c6af9ac6-7b61-11e6-9a41-93e8d4debeef', 'identity': {'cognitoIdentityPoolId': None, 'accountId': None, 'principalId': None, 'provider': 'aws:iam'}, 'type': 'User', 'auth': 'Basic realm=example.com'}}
    __len__ = (int) 9
```

デバッグしてみよう - ステップを進める

- 画面左下の Resume ボタン  を押すと次のブレークポイントまで進む
- 画面左下の Stop ボタン  を押すとプログラムが中断される
 - 実際に DB にデータをインサートしたくない場合などに中断できる
- 画面左下の矢印アイコンのボタンを押すとブレークポイントからのステップ操作（1行ずつ実行）ができる



今回は Resume  を 2 回押した後 Console タブの結果を確認する



勘の良い人は気づいたかもしれませんが、プログラムの実装に絶対の自信がなければ基本的には Run ではなく Debug で実装していきます。（Debug が Run を兼ねるため）

やってみよう - レスポンスボディを変更する

Q. レスポンスボディ 38 行目の “body” の値を Lambda 実行環境の ARN (arn:aws:lambda...) へ変えてください

期待する結果

```
{
  "statusCode": 200,
  "body": "arn:aws:lambda:us-east-1:012345678912:function>HelloWorldFunction"
}
```

ヒント: API Gateway から渡される情報のどこかに ARN の情報が入っています。
デバッグで変数 context の中身を見ると見つかるかも。body は文字列でも JSON 文字列でも良いよ。

```
def lambda_handler(event, context):
```

やってみよう - レスポンスボディを変更する

A.

```
return {  
    "statusCode": 200,  
    "body": context.invoked_function_arn,  
}
```

ポイント

- Lambda 実行環境の情報は `lambda_handler` 関数の引数 `context` に入っている
- `context` のデータにアクセスするには `context.キー名` (`context.invoked_function_arn`) と繋げて書く

まとめ

—

- SAM + Docker でローカルに Lambda のデバッグ環境が用意できる
- IDE とデバッガーを使いこなして快適に開発しよう
- プログラムはデバッグしながら書こう



次回

—

定時イベント処理を実装してみよう



資料作りで参考にさせていただいたブログ

Pycharmの力を使って爆速でAWS Lambdaを開発・テスト・デプロイする方法 - Qiita

<https://qiita.com/tez/items/36b5d1ace6519c1631cc>