lamapi Documentation

wuwentao

Aug 10, 2019

Contents:

A simple framework for building serverless api web applications based on AWS lambda and API Gateway.

CHAPTER 1

User's Guide

1.1 Installation

1.1.1 Python Version

We recommend using the latest version of Python 3. Lamapi supports Python 3.5 and newer.

1.1.2 Dependencies

Lamapi is a serverless framework and depends on AWS Lambda and API Gateway, so we try to restrict dependent packages. This is the only package you need to try Lamapi except built-in packages.

Lamapi depends on AWS Lambda and API Gateway, you should deploy the code to AWS after development. We recommend you to use something like Serverless Framework, check here to see how to use serverless framework on AWS.

1.1.3 Installation

You should install lamapi in the current directory since you can upload it to Lambda.

```
pip install -t lib lamapi
```

Lamapi is now installed. Check out the *Quick Start* or go to the *Documentation Overview*.

Check out our source from Github.

1.1.4 Start Serverless Project using Serverless Framework and Lamapi

We assume you have **Serverless Framework** installed. Installation Guide

Create A Project

```
serverless create --template aws-python3 --path hello
```

Update Configuration File

Update *serverless.yml* to something like this:

```
service: hello

provider:
  name: aws
  runtime: python3.7

functions:
  hello:
    handler: handler.handler
    events:
      - http:
          path: hello
          method: get
```

We use the python3.7 interpreter and add a function named *hello* listening at **/hello** path.

Then we edit *handler.py* file as below:

```
from lib.lamapi import Application


def handler(event, context):
    app = Application()

    @app.route(path='/hello', method='GET')
    def hello(request):
        return ['hello world']

    return app.run(event)
```

Deploy & Test Your Project

Deploy project

```
serverless deploy
```

Invoke function

```
serverless invoke -f hello -l
```

Next start to use **Lamapi** to build a web API *Quick Start*.

1.2 Quick Start

1.2.1 A Minimal Application

We assume you install Lamapi under **lib** directory and start a serverless project using Serverless Framework. *Installation*.

A minimal Lamapi application looks something like this:

```
from lib.lamapi.app import Application

def lambda_handler(event, context):
    app = Application()

    @app.route(path='/', method='GET')
    def hello(request):
        return 'Hello'

    return app.run(event)
```

So what did that code do?

1. First we import **Application** class, an instance of this class will help us to handle http event and return response.
2. Next in the **lambda_handler()** function (which is the entry point for the Lambda) we create an instance of this class.
3. Then we use the **route()** decorator to tell Lamapi what URL should trigger our function.
4. The function receives a request object and returns the message we want to display in the user's browser.

Just save it as hello.py or something similar. Then deploy it to the AWS Lambda and API Gateway or deploy using **Serverless Framework** as below.

```
serverless deploy
```

Then test it as:

```
serverless invoke -f hello
```

Then you will get output as below:

```
blalbal
```

Is it as simple as Flask?

1.2.2 Routing

Routing is as simple as **Flask**. Use *app.route* descriptor to functions with path and/or methods.

Listening on path / with all methods,

```
def lambda_handler(event, context):
    app = Application()

    @app.route(path='/')
    def hello(request):
```

(continues on next page)

(continued from previous page)

```
        return 'Hello'

    return app.run(event)
```

Listening on path / with GET method,

```
def lambda_handler(event, context):
    app = Application()

    @app.route(path='/', method='GET')
    def hello(request):
        return 'Hello'

    return app.run(event)
```

Use list to listen on more methods,

```
def lambda_handler(event, context):
    app = Application()

    @app.route(path='/', method=['GET', 'POST'])
    def hello(request):
        return 'Hello'

    return app.run(event)
```

1.2.3 Request

Each function will receive a request instance as parameter. You can use this object to get anything you want.

A request body from API Gateway will look like this:

```
{
  "body": "",
  "resource": "/hello",
  "path": "/hello",
  "httpMethod": "GET",
  "isBase64Encoded": false,
  "queryStringParameters": null,
  "pathParameters": null,
  "stageVariables": null,
  "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": "max-age=0",
    "User-Agent": "Custom User Agent String"
  }
  "requestContext": {
    //......
  }
}
```

Lamapi framework will store this object in *request* object. And you can get all you want by accessing request object's attributes.

Request Path

```
def hello(request):
    path = request.path
```

Request Method

```
def hello(request):
    method = request.method
```

Query Parameters

```
def hello(request):
    id = request.query.get('id')
    # if request url is /hello?id=1
    # then id = 1
    name = request.query.get('name') or 'default'
    # give some default value
```

Form Data

To access form data (data transmitted in a *POST* or *PUT* request) you can use the data attributes.

```
def hello(request):
    name = request.data.get('name') or 'default'
```

Path Parameters

If you define some path parameters in API Gateway, you can get them by *path_param*. If you define your path as */hello/{name}*, then requested as */hello/world*, you will get path parameters as *name=world*.

```
def hello(request):
    name = request.path_param.get('name') or 'default'
```

Request Header

Headers will be stored in *header* attribute as a dict.

```
def hello(request):
    accept = request.header.get('Accept')
```

1.2.4 Response

Anything you return to handler function will be translated to a JSON object which will be returned to client. You can return a string, dict, list or any object can be encoded to json.

```
def hello(request):
    return 'hello world'
```

If you want to return a custom http code such as *400* or custom headers, you can build response object by yourself.

```
from lib.lamapi.wrappers import Response


def hello(request):
    return Response([], status=400, headers={'X-CUSTOM': 'xxx'})
```

Where to go next? Learn deep about *Configuration*.

1.3 Configuration

1.3.1 Use your own configuration object

Lamapi use a simple config object to store variables. You can get config object by *config* attribute of *request* object.

```
def hello(request):
    // get config object
    config = request.config
    // get config variables
    log_level = request.config.LOG_LEVEL
```

You can define your own variables by extending the base config class.

Add a **config.py** file,

```
from lib.lamapi.config import BaseConfig


class Config(BaseConfig):

    VAR1 = 'value1'
```

Then you can start application by this config class,

```
from lib.lamapi import Application
from config import Config


def handler(event, context):
    config = Config()
    app = Application(config)

    @app.route(path='/', method='GET')
    def hello(request):
        // get config variable
        var1 = request.config.VAR1
```

1.3.2 Load configuration from environments

Mostly you want to config your application by environments.

Change your **config.py** file,

```
import os
from lib.lamapi.config import BaseConfig


class Config(BaseConfig):

    VAR1 = os.environ.get('VALUE1') or 'default'
```

1.4 Middlewares

CHAPTER 2

API Reference

CHAPTER 3

Indices and tables

- genindex
- modindex
- search