Lambada Documentation

Release 0.0.0

Superpedestrian, Inc.

September 28, 2016

Contents

CHAPTER 1

Lambada

A flask like framework for building multiple lambdas in one library/package by utilizing lambda-uploader.

CHAPTER 2

Quickstart

All you'll need to do to create a minimal lambada application is to add the following to a file called `lambda.py`:

```
from lambada import Lambada

tune = Lambada(role='arn:aws:iam:xxxxxxx:role/lambda')


@tune.dancer
def test_lambada(event, context):
    print('Event: {}'.format(event))
```

and a `requirements.txt` file that includes the lambada package (either `lambada` or `https://github.com/Superpedestrian/lambada` for the latest release or developer version respectively.

Much like a flask app, we now have a python file that is configured to upload a lambda function with the name `test_lambada` in your AWS account in the `us-east-1` region (since that is the default), and the handler will be set to `lamda.tune`, again the default.

So what is this doing over just writing the same thing without this framework?

For one it gives you a command line toolset to test, list, and publish multiple functions to AWS as independant Lambda's with one code base.

Now that you have your code, you can run the `lambada` command line tool after running `pip install -r requirements.txt` to do something like `lambada list`

```
List of discovered lambda functions/dancers:

test_lambada:
    description:
```

You can also test that lambda with an event passed on the command line using `lambada run test_lambada --event 'Hello'` to get:

```
Event: Hello
```

which creates a faked AWS Context object before running the specified *dancer*.

From there we can also package the functions (the same package works for all defined *dancers*/Lambda functions). So without configuring any AWS credentials, we can run `lambada package` to create a zip file with all your requirements packaged up (from the earlier created `requirements.txt`) that you can manually upload to AWS Lambda through the Web interface or similar.

If you have your AWS API credentials setup, and the correct permissions, you can also run `lambada upload` to have the function created and/or versioned with the packaged code for each *dancer*.

Pretty neat so far, but where it starts to cool is when there are many *dancers* with different requirements, VPCs, timeouts, and memory requirements all in the same deployable package similar to the following. We're going to go ahead and call our file `fouronthefloor.py` just as a reference for the customization you can do, so the contents of `fouronthefloor.py` would look like:

```
from lambada import Lambada

chart = Lambada(
    handler='fouronthefloor.chart',
    role='arn:aws:iam:xxxxxxx:role/lambda',
    region='us-west-2',
    timeout=60,
    memory=128
)


@chart.dancer
def test_lambada(event, context):
    print('Event: {}'.format(event))


@chart.dancer(
    name='not_the_function_name',
    description='Cool description',
    memory=512,
    region='us-east-1',
    requirements=['requirements.txt', 'xtra_requirements.txt']
)
def cool_oneoff(event, context):
    print('Wow, so much memory! in a diff region and extra reqs!')


@chart.dancer(memory=1024, timeout=5)
def bob_loblaw(event, _):
    print('Such a great reference!')
```

Which gives a `lambada list` that looks like:

```
List of discovered lambda functions/dancers:

bob_loblaw:
    description:
    timeout: 5
    memory: 1024

test_lambada:
    description:

not_the_function_name:
    description: Cool description
    region: us-east-1
    requirements: ['requirements.txt', 'xtra_requirements.txt']
    memory: 512
```

And with a few lines we've created three lambdas with different execution requirements all with one `lambada upload` command. Such a simple seductive dance .

2.1 Lambada Reference API Documentation

Lambada package entry point

class `lambada.`**`Dancer`**(*function*, *name=None*, *description=u''*, ***kwargs*)
Bases: `object`

Simple function wrapping class to add context to the function (i.e. name, description, memory.)

Creates a dancer object to let us know something has been decorated and store the function as the callable.

Parameters

- **`function`** (*`callable`*) – Function to wrap.
- **`name`** (*`str`*) – Name of function.
- **`description`** (*`str`*) – Description of function.
- **`kwargs`** – See `OPTIONAL_CONFIG` for options, if not specified in dancer, the Lambada objects configuration is used, and if that is unspecified, the defaults listed there are used.

`config`
A dictionary of configuration variables that can be merged in with the Lambada object

class `lambada.`**`Lambada`**(*handler=u'lambda.tune'*, ***kwargs*)
Bases: `object`

Lambada class for managing, discovery and calling the correct lambda dancers.

Setup the data structure of dancers and do some auto configuration for us with deploying to AWS using `lambda_uploader`. See `OPTIONAL_CONFIG` for arguments and defaults.

`dancer`(*name=None*, *description=u''*, ***kwargs*)
Wrapper that adds a given function to the dancers dictionary to be called.

Parameters

- **`name`** (*`str`*) – Optional lambda function name (default uses the name of the function decorated.
- **`description`** (*`str`*) – Description field in AWS of the function.
- **`kwargs`** – Key/Value overrides of either defaults or Lambada class configuration values. See `OPTIONAL_CONFIG` for available options.

Returns

Object with configuration and callable that is the function being wrapped

Return type *Dancer*

2.1.1 Modules

2.1.2 lambada.cli module

Command line interface for running, packaging, and uploading commands to AWS.

`lambada.cli.`**`create_package`**(*path*, *tune*, *requirements*, *destination='lambada.zip'*)
Creates and returns the package using `lambda_uploader`.

2.1.3 lambada.common module

Common classes, functions, etc.

class lambada.common.**LambadaConfig**(*path*, *config*)
Bases: lambda_uploader.config.Config

Small override to load config from dictionary instead of from a configuration file.

Takes config dictionary directly instead of retrieving it from a configuration file.

class lambada.common.**LambdaContext**(*function_name*, *function_version=None*, *invoked_function_arn=None*, *memory_limit_in_mb=None*, *aws_request_id=None*, *log_group_name=None*, *log_stream_name=None*, *identity=None*, *client_context=None*, *timeout=None*)
Bases: object

Convenient class duplication of the one passed in by Amazon as defined at:

http://docs.aws.amazon.com/lambda/latest/dg/python-context-object.html

Setup all the attributes of the class.

get_remaining_time_in_millis()
If we have a timeout return the amount of time left.

lambada.common.**get_lambada_class**(*path*)
Given the path, find the lambada class label by dir() ing for that type.

Parameters **path** (*click.Path*) – Path to folder or file

lambada.common.**get_time_millis**()
Returns the current time in milliseconds since epoch.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

l

C

D

G

L