Lambada Documentation

Release 0.2.1

Superpedestrian, Inc.

October 24, 2016

Contents

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

CHAPTER 1

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 get cool is when there are many *dancers* with different requirements, VPCs, timeouts, security configuration, 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 .

CHAPTER 2

Bouncers

AWS Lambda doesn't yet feature a way to add secure configuration items through environment variables (if it ever will), but there is often a need to have secrets that you don't want checked into source control such as API keys, passwords, certificates, etc. Generally it is nice to specify these with an out of source tree configuration file or environment variables. To achieve that here, we have the concept of `Bouncer` objects. This configuration object is created by default when you instantiate the `Lambada` class with a default configuration that you can use out of the box. The default *`lambada.Bouncer`* object looks for YAML configuration files in the following paths:

- Path specified by the environment variable `BOUNCER_CONFIG`
- The current working directory for `lambada.yml`
- Your `HOME` directory for `.lambada.yml`
- `/etc/lambada.yml`

and it does so in that order, terminating as soon as it successfully finds one.

In addition to those configuration files, it also will automatically add any variable prefixed with `BOUNCER_` (again default, and can be changed to an arbitrary prefix) to the bouncer configuration. This means that without any code you can add configuration to your Lambada project by just adding say `BOUNCER_API_KEY` to your local configuration and referencing it in your code as `tune.bouncer.api_key` (assuming `tune` is the variable you chose for your lambada class.

Similarly, if you define a `lambada.yml` configuration file that looks like:

```
api_key: 1234abcd
```

it will be accessible in the same way as `tune.bouncer.api_key`.

It is worth noting that the environment variable will override the same named variable in your yaml file.

How this works in Lamda is that the Bouncer configuration on the Lambada is read when packaged for AWS and written to a _lambada.yml configuration and is looked for first when running in Lambda.

2.1 Customizing Bouncers

If those defaults don't work for you, you can also pass in your own `Bouncer` to the `Lambada` object on creation. It allows you to directly pass in the path to the configuration and/or change the environment variable prefix like so:

```
from lambada import Bouncer, Lambada

bouncer = Bouncer(config_file='foobar.yml', env_prefix='COOL_')
tune = Lambada(bouncer=bouncer, role=bouncer.role)
```

```
@tune.dancer
def test_lambada(event, context):
    print(bouncer.role)
```

as an example, which lets you use bouncer to help configure the `Lambada` object

2.1.1 Releases

0.2.1

- Fixed relative import bug when searching for Lambada class

0.2.0

- Added bouncer configuration injection feature

0.1.0

First release

2.1.2 Lambada Reference API Documentation

Lambada package entry point

class `lambada.`**`Bouncer`**(*config_file=None*, *env_prefix=u'BOUNCER_'*)

Bases: `object`

Configuration class for lambda type deployments where you don't want to keep security information in source control, but don't have environment variable configuration as a feature of Lambda. It pairs with the command line interface to package and serializes the current configuration into the zip file when packaging. This allows changing with local environment variable changes or different configuration file paths at package time).

Configuration files, if not specified, are loaded in the following order, with the first one found being the only configuration (they do not layer):

- Path specified by `BOUNCER_CONFIG` environment variable.
- `lambada.yml` in the current working directory.
- `.lambada.yml` in your `HOME` directory
- `/etc/lambada.yml`

That said no configuration file is required and any environment variable with a certain prefix will be added to the attributes of this class when instantiated. The default is `BOUNCER_` so the environment variable `BOUNCER_THING` will be set in the object as `bouncer.thing` after construction.

These prefixed environment variables will also override whatever value is in the config, so if your config file has a `thing` variable in it, and `BOUNCER_THING` is set, the value of the environment variable will override the configuration file.

Finds and sets up configuration by yaml file

Parameters

- **`config_file`** (*str*) – Path to configuration file

- **env_prefix** (*str*) – Prefix of environment variables to use as configuration

export(*stream*)
Write out the current configuration to the given stream as YAML.

Parameters **destination** (*str*) – Path to output file.

Raises

- IOError
- yaml.YAMLError

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'*, *bouncer=<lambada.Bouncer object>*, ***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*

lambada.**get_config_from_env**(*env_prefix=u'BOUNCER_'*)
Get any and all environment variables with given prefix, remove prefix, lower case the name, and add as a dictionary item that is returned.

Parameters **env_prefix** (*str*) – environemnt variable prefix.

Returns

empty if no environment variables were found, or the dictionary of found variables that match the prefix.

Return type dict

lambada.**get_config_from_file**(*config_file=None*)
Finds configuration file and returns the python object in it or raises on parsing failures.

Parameters **config_file** (*str*) – Optional path to configuration file, runs through CONFIG_PATHS if none is specified.

Raises yaml.YAMLError

Returns

empty if no configuration file was found, or the contents of that file.

Return type dict

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.

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

B

C

D

E

G

L