lambda-tools Documentation

Release 0.1

James McKay

Apr 30, 2018

Contents:

Lambda Tools is a utility to help you to build, test, deploy, and set up Continuous Delivery pipelines for Python-based serverless code in AWS Lambda.

CHAPTER 1

Quick start

To install Lambda Tools, type:

```
pip install lambda-tools
```

Note: you need to be using Python 3.5 or later (3.6 preferred).

Now create a file in the root directory of your project called `aws-lambda.yml`. A minimal lambda definition file will look something like this:

```
version: 1

functions:
  hello_world:
    build:
      source: hello_world

    deploy:
      handler: hello.handler
      role: lambda-role
      region: eu-west-2
```

Create a folder next to your lambda function called `hello_world` and create a Python script within it called `hello.py`. Copy and paste the following contents into it:

```
def handler(event, context):
    return 'Hello world'
```

If you don't already have an IAM role set up to run AWS Lambda functions, create one in the AWS console:

- Select the IAM service under "Services"
- Under "Roles", select "Create Role"
- Under "AWS Service" choose "Lambda" then click "Next: Permissions"
- Select any policies that you want to apply to the role, then choose "Next: Review"

- Give the role a name — in this case, "lambda-role"
- Click "Create role"

Now run:

```
ltools build
```

You will see a file created next to your source directory called `hello_world.zip`. This is the package that will be uploaded to AWS Lambda.

Now run:

```
ltools deploy
```

All being well, this will deploy your code to AWS, in the `eu-west-2` (London) region.

CHAPTER 2

Configuration

Lambda Tools is configured by a file called `aws-lambda.yml` placed in your project's root directory. This can contain definitions for more than one Lambda function. A sample Lambda file might look like this:

```
# Configuration schema version 1.

version: 1

functions:
  hello_world:
    runtime: python3.6
    build:
      source: src/hello_world
      requirements:
        - file: requirements.txt
      use_docker: false
      compile_dependencies: false
      package: build/hello_world.zip
      ignore:
        - __pycache__
        - "*.py[cdo]"

    deploy:
      description: A basic Hello World handler
      region: eu-west-1
      handler: hello.handler
      memory_size: 128
      timeout: 60

      # Role, VPC, subnets, security groups and KMS key are all specified by name.
      role: service-role/NONTF-lambda

      vpc_config:
        name: My VPC
        subnets:
          - Public subnet
```

```
        - name: Private subnet
      security_groups:
        - name: allow_database

    kms_key:
      name: aws/lambda

    tags:
      wibble: wobble

    environment:
      variables:
        foo: baz
        bar:

    # tracing: PassThrough | Active
    tracing_config:
      mode: PassThrough

    # dead_letter: [ARN of SQS queue or SNS topic]
    dead_letter_config:
      target_arn: some-dead-letter-arn
```

It is a little known fact that YAML is actually a superset of JSON. This means that you can also provide your configuration in JSON format if preferred. As of version 0.1.2, lambda-tools will look for filenames `aws-lambda.yml`, `aws-lambda.yaml` or `aws-lambda.json` by default.

For example, a minimal JSON configuration file might look like this:

The configuration sections are as follows:

2.1 version

This is required; it should be set to 1.

2.2 functions

The `functions` section is required. It contains a list of function definitions; the name of each definition will be the name of the function as uploaded to AWS Lambda.

Each function has a number of different options:

2.2.1 runtime

The `runtime` parameter is optional and defaults to `python3.6`. It indicates which language runtime is used by the function.

Note that while you may specify any language supported by AWS, only `python3.6` (the default) is currently fully supported by Lambda Tools. Support for other AWS-supported runtimes is planned.

2.2.2 build

The `build` section is required. It tells Lambda Tools what resources are to be bundled into the zip file that is uploaded to AWS Lambda, how they are to be collated, and where the package is to be saved to disk.

Its parameters are as follows:

source

The folder containing your function's source code. This is specified relative to the `aws-lambda.yml` file. **Required**

requirements

A list of `requirements.txt` files specifying the Python packages to be downloaded from PyPI for inclusion with your function.

compile_dependencies

Compile the Python files in dependent packages into `.pyc` files. **Default: false**

By default, `.py` files in your dependencies are not compiled into `.pyc` files. This may increase the startup time of your lambda function, especially if the number of dependencies that you have specified is large but it does mean that the same build will produce exactly the same binary. This is important, for example, if you are using ltools in conjunction with Terraform, which looks for changes in your build output.

package

The filename where your function's bundled package should be saved, ready to upload to AWS. This is relative to the aws-lambda.yml file.

If not specified, it will be saved into a zip file next to the folder containing your source code.

use_docker

Build the lambda in a Docker container. **Default: false**

You will normally not need to use Docker, unless you are building your lambda function on OSX or Windows and some of your dependencies are written partly in C. If you get "Invalid ELF header" errors in AWS after uploading your lambda to AWS, change this setting to true. For more information see this article.

ignore

Specifies a list of file patterns to ignore when bundling the source code for your lambda function.

This allows you to specify, for example, compiled Python scripts (`*.pyc` files or `__pycache__` folders) or your `requirements.txt` file if it is located in the same folder as your source code.

2.2.3 deploy

The `deploy` section tells Lambda Tools how to deploy your code to AWS Lambda. It is optional; you only need it if you are using `ltools deploy` itself to deploy your function to AWS Lambda. If you are using a different mechanism, such as Terraform, you can omit it.

The parameters are as follows:

handler

The function's entry point into your code. For Python, this is specified in the format `module.handler`. **Required.**

role

The name of the IAM role attached to the lambda function. This determines who or what can run your function, as well as what resources it can access. **Required.**

source

The folder containing your function's source code. This is specified relative to the *aws-lambda.yml* file. **Required.**

description

A short description of what your function does.

memory_size

The amount of memory that your function can use at runtime, in gigabytes. Must be a multiple of 64 gigabytes. **Default: 128**.

region

The AWS region into which your function is to be deployed.

If not specified, it will be taken from either the environment variables or the configuration information that you have set using `aws configure`.

timeout

The maximum time, in seconds, that your function is allowed to run before being terminated. **Default: 3 seconds**.

dead_letter_config

Configures your lambda function's dead letter queue, to which notifications of failed invocations are sent. This can be either an SNS topic or an SQS queue, and it can be specified either by name or by ARN.

It can be configured in one of the following ways:

```
dead_letter_config:
  target_arn: (the ARN of your queue or topic)

dead_letter_config:
  target:
    sns: (the name of your SNS topic)

dead_letter_config:
  target:
    sqs: (the name of your SQS queue)
```

environment

The environment variables to be passed to your function. It is configured as follows:

```
environment:
  variables:
    VARIABLE: some value
    PASSTHROUGH_VARIABLE:
```

Variables whose value is left blank will be passed through to the function configuration from the environment which invokes `ltools`.

kms_key

The KMS key used to encrypt the environment variables. This can be specified either by name or by ARN:

```
kms_key:
  name: aws/lambda

kms_key:
  arn: "arn:aws:kms:eu-west-1:123456789012:key:01234567-89ab-cdef-0123-456789abcdef"
```

If no key is specified, the default key, `aws/lambda`, will be used.

tags

The tags to be assigned to your lambda function. For example:

```
tags:
  Account: marketing
  Application: newsletters
```

tracing_config

The tracing settings for your application. This contains a single argument, `mode`:

```
tracing_config:
  mode: PassThrough
```

`mode` can be set to either `PassThrough` or `Active`. If `PassThrough`, Lambda will only trace the request from an upstream service if it contains a tracing header with `sampled=1`. If `Active`, Lambda will respect any tracing

header it receives from an upstream service. If no tracing header is received, Lambda will call X-Ray for a tracing decision.

vpc_config

Add this section if you want your lambda function to access your VPC. You will need to specify subnets and security groups:

```
vpc_config:
  subnets:
    - id: subnet-12345678
    - name: public-subnet
    - another-subnet
  security_groups:
    - id: sg-12345678
    - name: some-group
    - another-group
```

Security groups and subnets can be specified either by ID or by name, as shown above. As a shortcut, you can omit `name:` when specifying it by name.

If you have two or more security groups or subnets with the same name in different VPCs, you will also need to specify the ID or name of the VPC in order to disambiguate them:

```
vpc_config:
  name: My VPC
  subnets:
    - id: subnet-12345678
    - name: public-subnet
    - another-subnet
  security_groups:
    - id: sg-12345678
    - name: some-group
    - another-group
```

CHAPTER 3

Command line instructions

Lambda Tools is run from the command line by using the `ltools` command. If you type `ltools --help`, you will be shown a list of the available commands. These are as follows.

3.1 ltools build

Usage: `ltools build [OPTIONS] [FUNCTIONS]...`

Build the specified lambda functions into packages ready for manual upload to AWS.

Options:

-s, --source TEXT	Specifies the source file containing the lambda definitions. Default: `aws-lambda.yml`.
--terraform	Renders output suitable for Terraform's external data source.
--help	Show this message and exit.

3.2 ltools deploy

Usage: `ltools deploy [OPTIONS] [FUNCTIONS]...`

Deploy the specified lambda functions to AWS.

Options:

-s, --source TEXT	Specifies the source file containing the lambda definitions. Default `aws-lambda.yml`.
--help	Show this message and exit.

Note: The lambda functions being deployed must already have been built using `ltools build`.

3.3 ltools list

Usage: `ltools list [OPTIONS] [FUNCTIONS]...`

Lists the lambda functions in the definition file.

Options:

-s, --source TEXT	Specifies the source file containing the lambda definitions. Default aws-lambda.yml.
--help	Show this message and exit.

3.4 ltools version

Usage: `ltools version [OPTIONS]`

Print the version number and exit.

Options:

--help	Show this message and exit.

CHAPTER 4

Using Lambda Tools with Terraform

If you are using Terraform to build up your infrastructure, you may want to use Lambda Tools to create the zip files to be uploaded to AWS.

Lambda Tools includes a "Terraform mode" switch in `ltools build` which allows you to use it in conjunction with Terraform's external data source.

The setup for the external data source will look something like this:

```
data "external" "lambda" {
  program = [
    "ltools",
    "build",
    "--terraform",
    "-s",
    "${path.module}/aws-lambda.yml",
    "${var.lambda_name}",
  ]
}

resource "aws_lambda_function" "my_lambda" {
  function_name = "${var.lambda_name}"

  filename         = "${lookup(data.external.lambda.result, "${var.lambda_name}")}"
  source_code_hash = "${base64sha256(file(lookup(data.external.lambda.result, "${var.
↪lambda_name}")))}"
  role             = "${aws_iam_role.github-user-management.arn}"

  handler = "main.handler"
  runtime = "python3.6"
  timeout = 30
}
```

What this is doing is passing an extra parameter, `--terraform`, to `ltools build`, which instructs it to render its output in the format required by Terraform's external data source — specifically, a JSON dictionary of strings. The dictionary so returned will list the names of the functions which have been built, together with the paths to

their respective build artefacts. You can then use Terraform's lookup function to get the filename to be passed to the aws_lambda_function resource.

Note: The `--terraform` option also redirects any output from `ltools build` from `stdout` to `stderr`. This output will only be rendered by Terraform if the build fails for any reason.
