# slam Documentation

*Release*

Miguel Grinberg

Jun 17, 2017

# Contents

Slam is a simple command-line utility that makes it very easy to do serverless deployments of Python web applications to AWS, using the Lambda and API Gateway services. In particular, Slam supports the transparent deployment of WSGI compliant applications.

CHAPTER 1

# About Slam

In this page you can find some background information on Slam and AWS.

## What is Serverless Computing?

Modern clouds, such as AWS, offer different ways to host applications. At the least involved level, you can create server instances, which are fully enabled virtual machines that run the operating system of your choice and are connected to the Internet. Once you have an instance up, you can login to it and install your software, exactly like you would on a local server. In AWS, this is the Elastic Compute Cloud (EC2) service.

Being able to work on virtual servers is nice, but the tendency is, however, to move towards a model in which developers only need to concentrate on their applications, leaving most or all of the installation and administration tasks to the cloud operator. This is what serverless computing is about.

AWS provides a number of services that make the life of the application developer easier. For example, it offers options for managed databases, message queues, notifications, emails and so on. You as a developer have the option to install your own stack on instances or containers, but if you want to spend all your energy on your application, using the managed services offered by AWS makes a lot of sense. And in addition to being convenient, these services have very attractive pricing based on a "you only pay for what you use" model, so in many cases you even end up saving money.

## What are Lambda and API Gateway?

In AWS, Lambda is the *function-as-a-service* (or *FaaS*) offering. With this service, you can upload your Python, Node.js, Java or C# code, and Lambda will deploy it and run it for you when you need to. To work with the Lambda service you upload your project's code packaged as a zip file, containing your application plus all its dependencies. You have to designate a function in your code as the entry point, and this function will be called by AWS when the Lambda function is invoked.

Because Lambda functions are supposed to be short lived, and are not running constantly like a web server, there are some types of applications that are not a good match for this service. In particular, any applications that rely on a server maintaining a long lived connection with the client will not work well. Examples of these applications are those that return live information as a stream, or those that use long-polling or WebSocket to provide constant updates to the client.

Deploying a web application to AWS Lambda also has its challenges. Web applications expose their functionality as one or more HTTP services, so they cannot directly work on the Lambda platform, since they rely servers that need to be running constantly to receive client requests. The Amazon API Gateway service bridges this gap, by allowing you to construct API endpoints, and configure what actions these endpoints trigger when the client sends a request to them. The service takes care of scaling, rate limiting, and even authentication if you want to offload that to the cloud too. Among the available actions you can associate with an API Gateway endpoint, there is invoking a Lambda function.

As you can probably guess, creating a web application on AWS Lambda and API Gateway is substantially different than what Python web developers are used to when creating their projects using Flask, Django or other web frameworks. The goal of Slam is to allow you to continue developing your web applications in the way you are used to, by automatically making your project compatible with the AWS serverless paradigm.

## How does Slam work?

Slam's command-line utility allows you to package and deploy your Python web application without having to make any changes to it. The idea is that you can continue to develop your application locally, and deploy it to AWS with a single command that takes care of the transformation required for the project to run in the serverless environment.

For regular Python functions, Slam does very little besides creating a self-contained package with your code and all of its dependencies. But for web applications, Slam converts HTTP requests and responses between the API Gateway and WSGI formats. When a request is received by API Gateway and passed on to the Lambda function, this request is converted to the WSGI format and given to your application, which is exactly what web servers such as gunicorn or uWSGI do. The WSGI response from the application is then converted back to the API Gateway format, before the Lambda function ends. Slam does all these conversions for you, so your application does not need to be changed at all.

One of the nicest features of Slam is how it creates neat and tidy deployments that are a pleasure to manage. For this, it relies on Cloudformation, the AWS orchestration service. Slam uses the project configuration to generate a Cloudformation template, and then runs this template to make changes on your AWS account. The end result is that every single resource that is allocated for your deployment is owned by the Cloudformation template, making it easy to keep track of what resources are in use.

If you ever find the need to create a custom deployment that differs from the standard structure used by Slam, it is possible to create plugins that extend Slam's Cloudformation template to suit different needs. In fact, a good part of the functionality offered by Slam natively is written as plugins.

## Alternatives to Slam

There are other serverless frameworks that create Lambda and API Gateway deployments similar to Slam. If for any reason Slam does not work for you, these may be good alternatives to research.

Chalice is an open-source framework from AWS that uses a decorator-based syntax similar to Flask and Bottle to create API Gateway and Lambda projects. The main disadvantage of Chalice compared to Slam is that it is not built on top of WSGI, so a project based on this framework cannot be run locally like you would with a standard WSGI application.

Zappa is another open-source framework that deploys Python functions and APIs to AWS Lambda and API Gateway. It is more mature than Slam, but overall similar. The main difference with Slam is that it invokes a variety of AWS APIs directly during a deployment, instead of using Cloudformation to orchestrate the deployment.

# CHAPTER 2

# Basic Tutorial

In this section you will learn how to deploy a Python function to AWS using Slam.

## Installing the Tutorial Project

To do this tutorial, you need to download a small Python project that consists of two files:

- fizzbuzz.py
- requirements.txt

Download these two files by right-clicking on the links above and selecting "Save link as..." to write them to your disk. Please put the files in a brand new directory.

This project is a version of the popular Fizz Buzz coding exercise. To become familiar with this application, you can run it as follows:

```
$ python3 fizzbuzz.py 2
2
$ python3 fizzbuzz.py 12
fizz
$ python3 fizzbuzz.py 15
fizz buzz
$ python3 fizzbuzz.py 5
buzz
```

If you prefer, you can also use Python 2.7 to run this function.

## Configuration

To prepare to deploy this application to Lambda, begin by installing the Slam utility with pip in a brand new virtual environment:

```
$ python3 -m venv venv
$ . venv/bin/activate
(venv) $ pip install slam
```

This will add a `slam` command to your virtual environment. You can use `slam --help` to see what are all the available options.

The `slam init` command can be used to create a starter configuration file:

```
(venv) $ slam init fizzbuzz:fizzbuzz
The configuration file for your project has been generated. Remember to add slam.yaml␣
↪to source control.
```

The above command generates a *slam.yaml* configuration file, with some initial settings. When you are working on a real project, you would want to add this file to source control, along with your own files. As your project evolves, you will hand edit this configuration file to make changes to your deployment.

The `fizzbuzz:fizzbuzz` argument tells Slam that the function is located in a module named `fizzbuzz` (the one on the left of the colon), and that the function that we want to deploy from that module is also named `fizzbuzz` (the one on the right of the colon).

Note that up to this point your AWS account has not been touched. All that has happened so far is configuration.

## AWS Credentials

Slam expects AWS credentials for your account to be installed in your system. As explained here, there are many possible sources of configuration, including environment variables or credential files.

If you are familiar with how AWS stores credentials, then feel free to use your preferred way. The following instructions use the AWS command-line utility to store credentials in configuration files in your home directory.

To be able to access AWS service from the command line, you first need to set up access keys on the AWS Console. If you are not familiar with AWS account security, it is highly recommended that you read the AWS Security Credentials section of the AWS documentation.

Once you have obtained your access and secret keys on the AWS Console, you can use the AWS command-line utility to store them in your system.

Install the AWS command-line utility with pip:

```
(venv) $ pip install awscli
```

Then use the `aws configure` command to enter your credentials. The command will prompt you to type them one by one:

```
(venv) $ aws configure
AWS Access Key ID [None]:
AWS Secret Access Key [None]:
Default region name [None]:
Default output format [None]:
```

The first two prompts are for your access keys. For the third prompt you have to pick one of the AWS regions. If you have no preference, use `us-east-1`, or pick the region closest to where you are located. In the screencast above, the `us-west-2` region is used.

## Deployment

With the AWS credentials installed, you can now proceed to deploy this project to AWS with the `slam deploy` command:

```
(venv) $ slam deploy
Building lambda package...
Deploying fizzbuzz:dev...
fizzbuzz is deployed!
  Function name: fizzbuzz-Function-1CUMOX2834PA0
  S3 bucket: fizzbuzz-J5FTHI40
  Stages:
    dev:$LATEST
```

The deployment process will take between about a minute. After the command finishes, you will have the function deployed and ready to be used!

The output from the `deploy` command indicates that the function was deployed to a `dev` stage, and that its version is `$LATEST`. Do not worry about this for this tutorial, stages and versioning will be covered in the second tutorial.

## Invoking your Lambda Function

The `slam invoke` command can be used to quickly test that the function hosted on AWS Lambda. If you look at the code of the function, you'll notice that the input is an argument named `number`. Below you can see how to invoke the function and pass a value for this argument using the `invoke` command:

```
(venv) $ slam invoke number:=2
2
(venv) $ slam invoke number:=12
fizz
(venv) $ slam invoke number:=15
fizz buzz
(venv) $ slam invoke number:=5
buzz
```

The `invoke` command needs to know the correct type of the arguments you are passing to your function. For each argument, you have to include the name of each argument and its value. For string arguments, you can use the `argument=value` syntax. If the argument is not a string, use `argument:=value` to have the argument intrepreted as JSON.

## Cloudformation Template

The deployment that you just finished was done through Cloudformation, the AWS orchestration service. If you are curious to see what resources were created, you can go to the Cloudformation section of the AWS console and view the stack that corresponds to this deployment.

You can also use the `slam template` command to view the Cloudformation template that was used for the deployment.

## Deleting the Project

A deployment orchestrated with Slam contains two high-level resources:

- A Cloudformation stack
- A S3 bucket with the Lambda zip file package inside

Every other resource allocated for the deployment is owned by the Cloudformation stack, which is very convenient, as this prevents resources to inadvertently be left behind or orphaned.

When you are done experimenting with this example project, you may want to remove it from your AWS account. If you want to perform a manual delete, you can just delete the Cloudformation stack and the S3 bucket, and that will leave your account clean of this deployment.

As a convenience to users, there is a `slam delete` command that performs the above two tasks for you:

```
(venv) $ slam delete
Deleting fizzbuzz...
Deleting logs...
Deleting files...
```

Congratulations! You have reached the end of this first tutorial. The second tutorial covers more advanced usages that include the deployment of a REST API project.

# CHAPTER 3

# Advanced Tutorial

In this second tutorial you will learn how to deploy a Python API project with Slam through a hands-on tutorial. In this tutorial you will use most features of Slam, and will have a small Python API deployed to AWS Lambda and API Gateway.

The screencast below is a recorded run through the entire tutorial. Feel free to use it as a reference when you go through the steps yourself, but note that it was created for an older release of Slam, so there are minor variations in the commands used.

## Installing the Tutorial Project

To do this tutorial, you need to download a small API project that consists of two files:

- tasks_api.py
- requirements.txt

Download these two files by right-clicking on the links above and selecting "Save link as..." to write them to your disk. Please put the files in a brand new directory.

The project uses a DynamoDB database, which is a AWS managed database service. To be able to run this project locally, you will need to download and run dynamodb-local.

Assuming you have dynamodb-local running, you set up and run this API with these commands:

```
$ virtualenv venv
$ source venv/bin/activate
(venv) $ pip install -r requirements.txt
(venv) $ python tasks_api.py
```

If you are following this tutorial on Windows, the virtual environment activation command (2nd line above) is `venv\Scripts\activate.bat`.

Once the API server is running, you can send requests to it at the address *http://localhost:5000*.

## Configuration

This is going to be similar to what you did in the first tutorial. To begin, install the Slam utility with pip:

```
(venv) $ pip install slam
```

Now use the `slam init` command to create a starter configuration file:

```
(venv) $ slam init tasks_api:app --wsgi --stages dev,prod --dynamodb-tables tasks
The configuration file for your project has been generated. Remember to add slam.yaml␣
↪to source control.
```

The above command generates a *slam.yaml* configuration file, with some initial settings. When you are working on a real project, you would want to add this file to source control, along with your own files. As your project evolves, you will hand edit this configuration file to make changes to your deployment.

Let's go over the options included in the `slam init` command above one by one:

- `tasks_api:app` is a standard notation used by Python web servers to designate the WSGI entry point of the application. What appears to the left of the colon, is the package or module where the WSGI callable instance is located. The name that appears to the right of the colon is the variable that holds the WSGI application instance. Slam uses this information to know where it needs to forward HTTP requests as they come into the Lambda function.
- `--wsgi` indicates that this project should be deployed as a WSGI complaint web application.
- `--stages dev,prod` creates two *stages*, named `dev` and `prod`. In slam, stages are independent versions of your deployed project. Having multiple stages allows you to deploy new features on one stage for development and testing purposes, while having a stable version of your project on another. You can create as many stages as you want. The first defined stage is configured as the default stage to receive new deployments. You will see later that Slam provides the tooling necessary to control what's deployed to the additional stages.
- `--dynamodb-tables tasks` creates a DynamoDB table called `tasks` for each defined stage. To make table names unique, Slam prefixes the requested name with the stage name. In this case, two DynamoDB tables will be created as part of this deployment, with names `dev.tasks` and `prod.tasks`.

Note that up to this point your AWS account has not been touched. All that has happened so far is configuration.

## AWS Credentials

If you haven't configured your AWS credentials yet, please go over the instructions to do so in the first tutorial.

## Deployment

With the AWS credentials installed, you can now proceed to deploy this API project to AWS with the `slam deploy` command:

```
(venv) $ slam deploy
Building lambda package...
Deploying tasks-api...
tasks-api is deployed!
  Function name: tasks-api-Function-1D55AHXPNB02D
  S3 bucket: tasks-api-MHPGRXBK
  Stages:
    dev:$LATEST: https://ukhhy78b6a.execute-api.us-west-2.amazonaws.com/dev
    prod:$LATEST: https://ukhhy78b6a.execute-api.us-west-2.amazonaws.com/prod
```

The deployment process can take between one and two minutes. After the command finishes, you will have the API deployed!

The command shows the URLs where the two stages are exposed. Since this is the first deployment, both `dev` and `prod` are unversioned stages that are reported as `$LATEST`, which means that the track the most current version of the code. We will see how to create versions in the next secion of this tutorial.

At this point, you can send requests to the `dev` request URL and it should behave exactly like the version you tested locally on your computer.

## Publishing a Version

Slam promotes a development cycle in which new versions of your project are deployed to your development stage, tested there, and then *published* to another stage, which could be a production stage, or maybe a staging stage.

When the project is published to a stage, it receives a permanent version number, which ensures the version running on that stage does not change regardless of what other code is deployed or published on other stages.

To publish the version of the API deployed in the previous section to the `prod` stage, the `slam publish` command is used:

```
(venv) $ slam publish prod
Publishing tasks-api:dev to prod...
tasks-api is deployed!
  Function name: tasks-api-Function-1D55AHXPNB02D
  S3 bucket: tasks-api-MHPGRXBK
  Stages:
    dev:$LATEST: https://ukhhy78b6a.execute-api.us-west-2.amazonaws.com/dev
    prod:1: https://ukhhy78b6a.execute-api.us-west-2.amazonaws.com/prod
```

Note that after the publish command completes, the `prod` stage is shown as `prod:1`, indicating that this stage is running version 1.

You can now continue working on the project, and run `slam deploy` to deploy the changes to the `dev` stage, and that is not going to affect the version of the project running on `prod`. If you want to upgrade the `prod` stage to a newer version of the project, just issue issue another `slam publish prod` command, and the current code in the `dev` stage will be used to upgrade `prod`, with a new version number.

## Project Status

The status report that is shown after the deploy or publish commands run can also be requested on its own using the `slam status` command:

```
(venv) $ slam status
tasks-api is deployed!
  Function name: tasks-api-Function-1D55AHXPNB02D
  S3 bucket: tasks-api-MHPGRXBK
  Stages:
    dev: https://ukhhy78b6a.execute-api.us-west-2.amazonaws.com/dev
    prod:1: https://ukhhy78b6a.execute-api.us-west-2.amazonaws.com/prod
```

## Deleting the Project

When you are done experimenting with this example project, you may want to remove it from your AWS account. If you want to perform a manual delete, you can just delete the Cloudformation stack and the S3 bucket used by this project, and that will leave your account clean of this deployment.

Alternatively, you can use the `slam delete` command, which performs the above two tasks for you:

```
(venv) $ slam delete
Deleting tasks-api...
Deleting logs...
Deleting files...
```

## The End

Congratulations! You have reached the end of the second and last tutorial.

Please review the reference sections in this documentation for complete information on all the commands and the options available through the configuration file.

CHAPTER 4

# Command Reference

## slam

The command `slam` provides access to all the features of this package thorugh subcommands. To find the list of available subcommands, use `slam --help`, and to find options available to a specific subcommand, use `slam <subcommand> --help`.

```
usage: slam [-h] [--config-file CONFIG_FILE]
            {init,build,deploy,publish,invoke,delete,status,logs,template} ...

positional arguments:
  {init,build,deploy,publish,invoke,delete,status,logs,template}
    init                Generate a configuration file.
    build               Build lambda package.
    deploy              Deploy the project to the development stage.
    publish             Publish a version of the project to a stage.
    invoke              Invoke the lambda function.
    delete              Delete the project.
    status              Show deployment status for the project.
    logs                Dump logs to the console.
    template            Print the default Cloudformation deployment template.

optional arguments:
  -h, --help            show this help message and exit
  --config-file CONFIG_FILE, -c CONFIG_FILE
                        The slam configuration file. Defaults to slam.yaml.
```

### Common arguments

The following command-line arguments are available to all subcommands, and when given, must appear before the subcommand name:

- `--config-file CONFIG_FILE` or `-c CONFIG_FILE`

Specify a custom configuration file. If this option is not given, the configuration is loaded from file *slam.yaml* in the current directory.

## slam init

The `slam init` command creates a brand new configuration file.

```
usage: slam init [-h] [--name NAME] [--description DESCRIPTION]
                 [--bucket BUCKET] [--timeout TIMEOUT] [--memory MEMORY]
                 [--stages STAGES] [--requirements REQUIREMENTS]
                 [--runtime RUNTIME] [--wsgi] [--no-api-gateway]
                 [--dynamodb-tables DYNAMODB_TABLES]
                 function

positional arguments:
  function              The function or callable to deploy, in the format
                        module:function.

optional arguments:
  -h, --help            show this help message and exit
  --name NAME           API name.
  --description DESCRIPTION
                        Description of the API.
  --bucket BUCKET       S3 bucket where lambda packages are stored.
  --timeout TIMEOUT     The timeout for the lambda function in seconds.
  --memory MEMORY       The memory allocation for the lambda function in
                        megabytes.
  --stages STAGES       Comma-separated list of stage environments to deploy.
  --requirements REQUIREMENTS
                        The location of the project's requirements file.
  --runtime RUNTIME     The Lambda runtime to use, such as python2.7 or
                        python3.6
  --wsgi                Treat the given function as a WSGI app.
  --no-api-gateway      Do not deploy API Gateway.
  --dynamodb-tables DYNAMODB_TABLES
                        Comma-separated list of table names to create for each
                        stage.
```

### Required arguments

- `wsgi_app`

  A reference to the project's WSGI application callable. This argument must be in the format `<module>:<app>`, where `module` is the module or package name where the WSGI application callable is located, and `app` is the name of the variable that holds it.

### Optional arguments

- `--name NAME`

  The name of the project. If this argument is not given, the WSGI module is used as the project name.

- `--description DESCRIPTION`

  A short project description.

- `--bucket BUCKET`

  The name of an S3 bucket to use as storage for Lambda packages. If this argument is not given, the project name is used as bucket name.

- `--timeout TIMEOUT`

  The timeout to configure on the Lambda function, in seconds. The default is 10 seconds.

- `--memory MEMORY`

  The amount of memory to provision for the Lambda function, in megabytes. The default is 128 MB.

- `--stages STAGES`

  A comma-separated list of stage names to create as part of the deployment. If this argument is not provided, a single stage named `dev` is created.

- `--requirements REQUIREMENTS`

  The name of the Python requirements file that contains the project dependencies. If this argument is not given, slam looks for a *requirements.txt* file in the project's root directory.

- `--runtime RUNTIME`

  The name of the Lambda runtime to use for the function. This can be either `"python2.7"` or `"python3.6"`. If this argument is not provided, the runtime is guessed from the version of python that is being used.

- `--dynamodb-tables DYNAMODB_TABLES`

  A comma-separated list of DynamoDB table names to create for each stage. Once these tables are created, they will be named using the format `<stage>.<table_name>`, so that each stage has a unique table name.

### Examples

```
$ slam init fizzbuzz:fizzbuzz --stages dev,prod
The configuration file for your project has been generated. Remember to add slam.yaml␣
↪to source control.

$ slam init tasks_api:app --wsgi --stages dev,staging,prod --dynamodb-tables users,
↪tasks
The configuration file for your project has been generated. Remember to add slam.yaml␣
↪to source control.
```

## slam build

The `slam build` command builds a Lambda package, without deploying it.

```
usage: slam build [-h] [--rebuild-deps]

optional arguments:
  -h, --help       show this help message and exit
  --rebuild-deps   Reinstall all dependencies.
```

### Required arguments

None.

### Optional arguments

- `--rebuild-deps`

  To speed up the build process, this command reuses dependencies from a previous build (installing any requirement changes on top). If this option is given, old requirements are deleted and everything is installed from scratch.

### Example

```
$ slam build
lambda_package.20170112_143002.zip has been built successfully.
```

## slam deploy

The `slam deploy` command deploys your project to a stage on AWS.

```
usage: slam deploy [-h] [--rebuild-deps] [--no-lambda]
                   [--lambda-package LAMBDA_PACKAGE] [--stage STAGE]

optional arguments:
  -h, --help            show this help message and exit
  --rebuild-deps        Reinstall all dependencies.
  --no-lambda           Do no deploy a new lambda.
  --lambda-package LAMBDA_PACKAGE
                        Custom lambda zip package to deploy.
  --stage STAGE         Stage to deploy to. Defaults to the stage designated
                        as the development stage
```

### Required arguments

None.

### Optional arguments

- `--rebuild-deps`

  To speed up the deployment process, this command reuses dependencies from a previous deploy (installing any requirement changes on top). If this option is given, old requirements are deleted and everything is installed from scratch.

- `--no-lambda`

  Skip a deployment of a new lambda package. This can be used when a deployment has been updated, but the code has not. A typical example of when this is convenient is when the configuration file is edited to add or remove stages or database tables.

- `--lambda-package LAMBDA_PACKAGE`

  Instead of building a new lambda package, use the one provided. The given package must be a zip file in the format required by AWS Lambda. The zip files produced by the `slam build` command can be used here.

- `--stage STAGE`

  The stage that receives the updated Lambda function. By default this is the stage that is marked as the development stage in the configuration. The stage that receives the deployment will be updated to the latest version of the Lambda function as part of the deployment.

### Example

```
$ slam deploy
Building lambda package...
Deploying simple-api...
simple-api is deployed!
  Function name: simple-api-Function-1XARPP7W4H3KR
  Stages:
    dev:$LATEST: https://ukhhy78b6a.execute-api.us-west-2.amazonaws.com/dev
    prod:31: https://ukhhy78b6a.execute-api.us-west-2.amazonaws.com/prod
    staging:30: https://ukhhy78b6a.execute-api.us-west-2.amazonaws.com/staging
```

## slam publish

The `slam publish` command makes a version of your project available on a stage with a persistent version number.

```
usage: slam publish [-h] [--version VERSION] stage

positional arguments:
  stage              Stage to publish to.

optional arguments:
  -h, --help         show this help message and exit
  --version VERSION  Stage name or numeric version to publish. Defaults to the
                     development stage.
```

### Required arguments

- `stage`

  The stage that receives the published version of the project.

### Optional arguments

- `--version VERSION`

  Publish a specific Lambda version. The given version can be a number, or a stage name. When a stage name is given, the version of the project stored in that stage is published.

### Examples

Assuming a project that has three stages named `dev`, `staging` and `prod`, new code versions in the `dev` stage can be published to `staging` with this command:

```
$ slam publish staging
Publishing simple-api:dev to staging...
simple-api is deployed!
  Function name: simple-api-Function-1XARPP7W4H3KR
  Stages:
    dev:$LATEST: https://ukhhy78b6a.execute-api.us-west-2.amazonaws.com/dev
    prod:1: https://ukhhy78b6a.execute-api.us-west-2.amazonaws.com/prod
    staging:2: https://ukhhy78b6a.execute-api.us-west-2.amazonaws.com/staging
```

Later a version running on staging can be published to `prod` with:

```
$ slam publish prod --version staging
Publishing simple-api:staging to prod...
simple-api is deployed!
  Function name: simple-api-Function-1XARPP7W4H3KR
  Stages:
    dev:$LATEST: https://ukhhy78b6a.execute-api.us-west-2.amazonaws.com/dev
    prod:2: https://ukhhy78b6a.execute-api.us-west-2.amazonaws.com/prod
    staging:2: https://ukhhy78b6a.execute-api.us-west-2.amazonaws.com/staging
```

## slam status

The `slam status` command shows the current deployment status of your project.

```
usage: slam status [-h]

optional arguments:
  -h, --help  show this help message and exit
```

### Required arguments

None.

### Optional arguments

None.

### Example

```
$ slam status
simple-api is deployed!
  Function name: simple-api-Function-1XARPP7W4H3KR
  Stages:
    dev:$LATEST: https://ukhhy78b6a.execute-api.us-west-2.amazonaws.com/dev
    prod:4: https://ukhhy78b6a.execute-api.us-west-2.amazonaws.com/prod
    staging:3: https://ukhhy78b6a.execute-api.us-west-2.amazonaws.com/staging
```

## slam invoke

The `slam invoke` command invokes the Lambda function.

```
usage: slam invoke [-h] [--stage STAGE] [--async] [--dry-run]
                   [args [args ...]]

positional arguments:
  args           Input arguments for the function. Use arg=value for strings,
                 or arg:=value for integer, booleans or JSON structures.

optional arguments:
  -h, --help     show this help message and exit
  --stage STAGE  Stage of the invoked function. Defaults to the development
                 stage
  --async        Invoke the function but don't wait for it to return.
  --dry-run      Just check that the function can be invoked.
```

### Required arguments

None.

### Optional arguments

- `--stage STAGE`

  The stage on which to run the function. Defaults to the development stage.

- `--async`

  Invoke the function, but don’t wait for it to run.

- `--dry-run`

  Do not invoke the function, just check that the current user is allowed to invoke it.

- `args [args ...]`

  Input arguments to pass to the function. To pass a string argument, use `argument=value`. To pass a non-string argument, use `argument:=value`, where `value` is a number, boolean (`true` or `false`) or raw JSON string.

### Example

```
$ slam invoke number:=15
fizzbuzz

$ slam invoke name=john age:=34
OK
```

## slam template

The `slam template` command dumps the slam Cloudformation template to the console.

```
usage: slam template [-h]

optional arguments:
  -h, --help  show this help message and exit
```

### Required arguments

None.

### Optional arguments

None.

### Example

```
$ slam template
<template output dumped to the console>
```

## slam logs

The `slam logs` command dumps logs to the console.

```
usage: slam logs [-h] [--stage STAGE] [--period PERIOD] [--tail]

optional arguments:
  -h, --help            show this help message and exit
  --stage STAGE         Stage to show logs for. Defaults to the stage
                        designated as the development stage
  --period PERIOD, -p PERIOD
                        How far back to start, in weeks (1w), days (2d), hours
                        (3h), minutes (4m) or seconds (5s). Default is 1m.
  --tail, -t            Tail the log stream
```

### Required arguments

None.

### Optional arguments

- `--stage STAGE`

  The stage to dump logs for.

- `--period PERIOD`

  How far back to start the log listing. The period can be given in weeks (1w), days (2d), hours (3h), minutes (4m) or seconds (5s). The default is 1 minute.

- `--tail`

  Dump new logs as they appear.

### Example

```
$ slam logs
<log output dumped to the console>
```

## slam delete

The `slam delete` command completely removes a deployment from AWS.

```
usage: slam delete [-h] [--no-logs]

optional arguments:
  -h, --help   show this help message and exit
  --no-logs    Do not delete logs.
```

### Required arguments

None.

### Optional arguments

- `--no-logs`

  Do not delete the project logs.

### Example

```
$ slam delete
Deleting api...
Deleting logs...
Deleting files...
```

# CHAPTER 5

# Configuration Reference

This section enumerates all the options that can be provided in the *slam.yaml* configuration file.

## Core Options

- `name`

  The name of the project.

- `description`

  A description for the project.

- `function`

  Options that describe the function that is being deployed.

  - `module`

    The Python module or package that contains the application callable.

  - `app`

    The name of the function or callable to invoke.

- `requirements`

  The project's requirements filename.

- `devstage`

  The name of the stage designated as the development stage.

- `environment`

  A collection of variables, specified as key-value pairs, that are made available to the Lambda function as environment variables.

  Example:

```
environment:
  IN_LAMBDA: "1"
  ADMIN_URL: "1.2.3.4"
```

- `stage_environments`

  A collection of stages. Each stage contains a collection of variables, given as key-value pairs. These variables are exposed as environment variables to the Lambda function when running on the stage.

  Example:

  ```
  stage_environments:
    dev:
      DEBUG: "1"
    prod:
      DEBUG: "0"
  ```

  Note: When using multiple stages, it is important to that any stage variables defined in this section are given values for all stages. This is necessary because sometimes AWS reuses Lambda containers, so environment variables from a previous invocation on a different stage may still exist.

- `aws`

  A collection of settings specific to AWS.

  – `s3_bucket`

    The bucket on S3 where Lambda packages are to be stored. If this bucket does not exist, it is created during the deployment.

  – `lambda_timeout`

    The timeout, in seconds, for the Lambda function.

  – `lambda_memory`

    The memory size, in megabytes, for the Lambda function.

  – `lambda_security_groups`

    If the Lambda function needs to access resources inside a VPC, this entry must contain the list of security groups for the function to use. When VPC access is not desired, this entry must be left blank.

  – `lambda_subnet_ids`

    If the Lambda function needs to access resources inside a VPC, this entry must contain the list of subnet IDs in that VPC that have to be connected to the function. When VPC access is not desired, this entry must be left blank.

  – `lambda_managed_policies`

    This entry can define additional managed policies to be assigned to the Lambda function execution role. These can be AWS managed policies (you can provide just the policy name, such as `AWSLambdaDynamoDBExecutionRole`), or custom managed policies, for which you must provide the fully qualified ARN.

  – `lambda_inline_policies`

    This entry can define additonal inline policies to be assigned to the Lambda function execution role.

  – `cfn_resources`

    A list of additional Cloudformation resources to add to the deployment.

  - `cfn_outputs`

    A list of additional Cloudformation outputs to add to the deployment.

## WSGI Plugin

- `wsgi`

  If this configuration option exists, the project is assumed to be a web application compliant with the WSGI protocol. The values under the `function` option (described above) are assumed to be of the WSGI callable.

  The following options provide more details on how the WSGI deployment should be configured:

  - `deploy_api_gateway`

    If set to `true` (the default), an API Gateway resource is created to map to the Lambda function, so that HTTP requests can be made transparently. If set to `false`, no API Gateway resources are deployed.

  - `log_stages`

    A list of stages that are configured to include API Gateway logging. For included stages, API Gateway will produce detailed logging. For stages not included, logging will only be produced for errors. This option is only meaningful when `deploy_api_gateway` is set to `true`.

## DynamoDB Plugin

- `dynamodb_tables`

  A collection of DynamoDB tables to create for each stage. Each table entry is defined by the table name, and contains a sub-collection of settings that define the table schema.

  Tables created by this plugin have a name with the format *stage.name*, so for example, for a project that defines `dev` and `prod` stages, a table named `mytable` in the configuration will result in DynamoDB tables `dev.mytable` and `prod.mytable` created.

  - `attributes`

    A collection of attributes, as key-value pairs where the key is the attribute name, and the value is the attribute type. Attribute types are defined by DynamoDB and can be `"S"` for string, `"N"` for number, `"B"` for binary, and `"BOOL"` for boolean.

  - `key`

    The name of the attribute that is the table's hash key, or a list of two elements with the attributes that are the table's hash and range keys.

  - `read_throughput`

    The read throughput units for the table.

  - `write_throughput`

    The write throughput units for the table.

  - `local_secondary_indexes`

    A collection of local secondary indexes to define for the table. The indexes are defined by their name, and contain a sub-collection that specifies their structure.

    * `key`

      Same as the table-level `key` attribute. For a local secondary index, the hash key must match the key selected for the table-level index.

    * `project`

      The attributes to project on this index. If set to `"all"` all table attributes are projected. Else it can be set to a list of attribute names to project, or to an empty list to only project the key attributes.

  - `global_secondary_indexes`

    A collection of global secondary indexes to define for the table. The indexes are defined by their name, and contain a sub-collection that specifies their structure.

    * `key`

      Same as the table-level `key` attribute.

    * `project`

      The attributes to project on this index. If set to `"all"` all table attributes are projected. Else it can be set to a list of attribute names to project, or to an empty list to only project the key attributes.

    * `read_throughput`

      The read throughput units for the index.

    * `write_throughput`

      The write throughput units for the index.

Example:

```
dynamodb_tables:
  # a simple table with "id" as hash key
  mytable:
    attributes:
      id: "S"
    key: "id"
    read_throughput: 1
    write_throughput: 1

  # a more complex table with hash/sort keys and secondary indexes
  mytable2:
    attributes:
      id: "S"
      name: "S"
      age: "N"
    key: ["id", "name"]
    read_throughput: 1
    write_throughput: 1
    local_secondary_indexes:
      myindex:
        key: ["id", "age"]
        project: ["name"]
    global_secondary_indexes:
      myindex2:
        key: ["age", "name"]
        project: "all"
        read_throughput: 1
        write_throughput: 1
```

# CHAPTER 6

# Plugin Development

Coming soon!