tight-cli Documentation

Release 1

Michael McManus

Oct 01, 2018

Getting Started Tutorial

1 Overview **3**

2 Motivation & Philosophy **5**

3 Getting Started **7**

4 Installation **9**
4.1 `tight-cli` . . . 9
4.2 `tight` . . . 9

5 Quickstart **11**
5.1 Install . . . 11
5.2 Generate a Tight App . . . 11
5.3 Install Environment Dependencies . . . 11
5.4 Install Application Dependencies . . . 11
5.5 Generate Environment File . . . 12
5.6 Generate a Function & Tests . . . 12

6 Tutorial **13**
6.1 Setup . . . 13
6.1.1 Virtual Environment . . . 13
6.1.2 Package Installation . . . 13
6.2 Generate An App . . . 14
6.2.1 Generate Project Directory . . . 14
6.2.2 The Anatomy of a Default App . . . 14
6.2.3 Install Dependencies . . . 17
6.2.4 Environment Dependencies . . . 17
6.2.5 App Dependencies . . . 18
6.2.6 Conclusion . . . 18
6.3 Generate Environment Files . . . 20
6.3.1 Working With dist.env.yml . . . 20
6.3.2 App Environment Variables . . . 20
6.4 Generate A Function . . . 20
6.4.1 Lambda Proxy Event Controllers . . . 20
6.4.2 Test Stubs . . . 20
6.5 Working with Tight Core . . . 20
6.5.1 Lambda App . . . 20

6.5.2 Lambda Proxy Event . . . 20
6.5.3 Clients . . . 20
6.5.4 Test Helpers . . . 20
6.6 Models and Data . . . 20
6.6.1 DynamoDb Support . . . 20
6.6.2 Generate a Model . . . 20
6.6.3 Install DynamoDb Locally . . . 20
6.6.4 Running DynamoDb Locally . . . 20
6.6.5 Testing DynamoDb Interactions . . . 20
6.7 Working with Responses . . . 20
6.7.1 Using Serializers . . . 20
6.8 Creating an Artifact . . . 20
6.9 Deploy to AWS . . . 20
6.9.1 Create Lambda Function . . . 20
6.9.2 Create API Gateway . . . 20
6.10 Deploy With Serverless . . . 20
6.10.1 Create Serverless Project . . . 20
6.10.2 Define Serverless Service . . . 20

7 tight-cli **21**
7.1 tight generate . . . 21
7.1.1 tight generate app . . . 21
7.1.2 tight generate function . . . 22
7.1.3 tight generate model . . . 24
7.1.4 tight generate env . . . 25
7.2 tight pip . . . 25
7.2.1 tight pip install . . . 25
7.3 tight dynamo . . . 25
7.3.1 tight dynamo installdb . . . 26
7.3.2 tight dynamo rundb . . . 26
7.3.3 tight dynamo generateschema . . . 26

8 tight **29**
8.1 tight.providers . . . 29
8.1.1 tight.providers.aws.lambda_app.app . . . 29
8.2 tight.core.logger . . . 31

Python Module Index **33**

Tight is a microframework created and optimized for serverless runtimes. With `tight-cli` and `tight` you can quickly scaffold serverless applications that are conventional, testable by default and free of boilerplate.

Tight currently supports AWS Lambda and the Python2.7 runtime.

Tight is a microframework created and optimized for serverless runtimes. With `tight-cli` and `tight` you can quickly scaffold serverless applications that are conventional, testable-by-default and free of boilerplate.

Tight currently supports AWS Lambda and the Python2.7 runtime.

CHAPTER 1

Overview

The modules provided by `tight` make it easy to write lambda functions that act as REST resource controllers. Simply author your function, following Tight's conventional naming and directory schemes (don't worry, `tight-cli` can generate these files and directories for you!):

my_app/app/functions/my_controller/handler.py:

```
import tight.providers.aws.controllers.lambda_proxy_event as lambda_proxy

@lambda_proxy.get
def get_handler(*args, **kwargs):
    return {
        'statusCode': 200,
        'body': {
            'hello': 'world'
        }
    }

@lambda_proxy.post
def post_handler(*args, **kwargs):
    event = kwargs.pop('event')
    # Process request event...
    # ...
    return {
        'statusCode': 201,
    }

@lambda_proxy.put
def put_handler(*args, **kwargs):
    event = kwargs.pop('event')
    return {
        'statusCode': 200,
    }

@lambda_proxy.patch
```

(continues on next page)

(continued from previous page)

```
def patch_handler(*args, **kwargs):
    pass

@lambda_proxy.delete
def delete_handler(*args, **kwargs):
    pass
```

Import and run a `tight` app (as generated by `tight generate app my_app`:

my_app/app_index.py:

```
# Project structure generated by `tight generate app my_app`
# Dependencies installed by running `tight pip install --requirements`
from app.vendored.tight.providers.aws.lambda_app import app as app
app.run()
```

And call the controller on the module defined by *app_index.py*:

```
app_index.my_controller
```

And as mentioned the `tight-cli` command line tool will scaffold services, functions with test stubs and much more!

Create an application:

Create functions with tests:

CHAPTER 2

Motivation & Philosophy

Tight is inspired by tools and frameworks such as Sinatra, Ember, Flask, Chalice, and Serverless – to name just a few. Tight aspires to help you and your team build servereless applications in a repeatable and conventional manner. To achieve this goal Tight provides two distinct packages: tight and tight-cli.

The core package, tight, provides modules that map request events to REST style resource handlers – this makes it easy to author declarative resource controllers that group method handlers logically and legibly. Additional modules help you interact with external services, like DynamoDb, in an intuitive and fluent manner. Finally, a suite of test helpers makes it easy to record and simulate HTTP requests and AWS SDK calls.

The Tight command line tool, tight_cli, helps you quickly scaffold and test Tight applications and functions. The tutorials walk through every `tight-cli` command and will demonstrate how to:

- Generate application directories and files
- Generate functions and tests
- Install and manage application dependencies
- Install, configure, and run a local instance of DynamoDb.

The `tight-cli` package does the dirty work of preparing your application for predictable and hassle-free deployments. With `tight-cli` you can easily generate CloudFormation compatible DynamoDb schemas from application model definitions as well as create deployable artifacts that can be used with a variety of deployment strategies. Tight does not make assumptions or prescriptions about which approach you and your team follow; Tight apps can be deployed directly through the AWS console or in conjunction with other tools and services.

Tight doesn't try to enforce a specific application deployment process, rather it allows you to get your application to a deployable state quickly and in a convention-over-configuration manner. This is where Tight differs with existing serverless-specific tools like Serverless, Chalice, Gordon and Zappa. Another significant divergence from other frameworks and tools (all of which have inspired Tight in some way!) is that Tight doesn't provide a mechanism to invoke or run applications locally via a development server or some other "simulated" service. This is because Tight makes writing tests *easy* which in turn makes building resilient apps and services approachable and frictionless.

CHAPTER 3

Getting Started

The best way to get started with Tight is to follow the tutorial, which will guide you through the process of building and deploying an app. There are also references for both tight and tight-cli.

CHAPTER 4

Installation

The `tight` and `tight-cli` packages are both available to install via Github while the packages are in preview.

4.1 `tight-cli`

```
$ pip install git+git://github.com/lululemon/tight-cli.git#egg=tight-cli
```

4.2 `tight`

```
$ pip install git+git://github.com/lululemon/tight.git#egg=tight
```

CHAPTER 5

Quickstart

For a through introduction to Tight and all that it offers read through and follow the tutorials. This quickstart guide is designed to show you the sequence of commands you have to run to get a new project up and running and ready for development.

5.1 Install

Install `tight-cli` via, Git.

```
$ pip install git+git://github.com/lululemon/tight-cli.git#egg=tight-cli
```

5.2 Generate a Tight App

```
$ tight generate app my_service
```

5.3 Install Environment Dependencies

```
$ cd my_service
$ pip install -r requirements.txt
```

5.4 Install Application Dependencies

```
$ cd my_service
$ tight pip install --requirements
```

5.5 Generate Environment File

```
$ cd my_service
$ tight generate env
{CI: false, NAME: my-service, STAGE: dev}
```

5.6 Generate a Function & Tests

```
$ cd my_service
$ tight generate function my_function

============================================= test session starts␣
↪=============================================
platform darwin -- Python 2.7.10, pytest-3.0.5, py-1.4.32, pluggy-0.4.0
rootdir: /Users/michael/Development/my_service, inifile:
collected 2 items

tests/functions/integration/my_function/test_integration_my_function.py .
tests/functions/unit/my_function/test_unit_my_function.py .

========================================= 2 passed in 0.12 seconds␣
↪=========================================
Successfully generated function and tests!
```

CHAPTER 6

Tutorial

6.1 Setup

To get up and running with `tight-cli` you'll have to install the package to a virtual environment. Fore more information about setting up and installing virtual environments, visit virtualenv.

6.1.1 Virtual Environment

Create a new virtual environment:

```
$ mkvirtualenv my_tight_app_env
```

Or activate an existing environment:

```
$ workon my_tight_app_env
```

The commands `mkvirtualenv` and `workon` are provided by virtualenvwrapper.

6.1.2 Package Installation

Currently, the `tight-cli` package is installed via git. Once you are in your virtual environment you can install the package by issuing the following command:

```
$ pip install git+git://github.com/lululemon/tight-cli.git#egg=tight-cli
```

You are now ready to start building a Tight app!

6.2 Generate An App

Tight has been designed to aid in fast and repeatable development of microservices. As such, Tight provides a suite of commands that generate files and directories that are common to all projects. The very first generate command that we are going to explore is `tight generate app`.

6.2.1 Generate Project Directory

Navigate to the location where you want to create your app. I like to keep my code projects in the `Development` directory in my home location, so I'll head that way.

```
$ cd ~/Development
```

Now I'm going to use `tight generate app` to generate a new project. Simply provide a name as the command's only argument:

```
$ tight generate app tight_app
```

Your app has been generated! To verify, you can list the contents of the `tight_app` directory.

```
drwxr-xr-x  12 user  group   408B Jan  2 14:56 .
drwxr-xr-x@ 24 user  group   816B Jan  2 14:56 ..
-rw-r--r--   1 user  group   143B Dec 25 17:07 .gitignore
drwxr-xr-x   8 user  group   272B Jan  2 14:56 app
-rw-r--r--   1 user  group    76B Dec 25 16:38 app_index.py
-rw-r--r--   1 user  group   476B Jan  2 14:56 conftest.py
-rw-r--r--   1 user  group    56B Dec 26 02:28 env.dist.yml
-rw-r--r--   1 user  group    60B Dec 17 18:03 requirements-vendor.txt
-rw-r--r--   1 user  group    40B Dec 17 18:05 requirements.txt
drwxr-xr-x   4 user  group   136B Dec 23 12:39 schemas
drwxr-xr-x   4 user  group   136B Dec 23 11:22 tests
-rw-r--r--   1 user  group    54B Jan  2 14:56 tight.yml
```

6.2.2 The Anatomy of a Default App

app/

The app directory, not surprisingly, contains your application's runtime logic. This directory organizes your application's function code along with dependencies and share libraries.

List the contents of the directory to see what was created:

```
$ ls -la app

drwxr-xr-x   8 user  group   272B Jan  2 14:56 .
drwxr-xr-x  13 user  group   442B Jan  2 14:57 ..
-rw-r--r--   1 user  group   339B Dec 23 11:22 __init__.py
drwxr-xr-x   3 user  group   102B Dec 17 17:25 functions
drwxr-xr-x   3 user  group   102B Dec 23 11:22 lib
drwxr-xr-x   3 user  group   102B Dec 23 10:46 models
drwxr-xr-x   3 user  group   102B Dec 23 10:46 serializers
drwxr-xr-x   3 user  group   102B Jan  2 14:56 vendored
```

app/__init__.py

The `app` directory's `__init__.py` file augments the current environment so that application dependencies are discoverable for import.

The generator currently produces the following file:

```
import sys, os
here = os.path.dirname(os.path.realpath(__file__))
sys.path = [os.path.join(here, "./vendored")] + sys.path
sys.path = [os.path.join(here, "./lib")] + sys.path
sys.path = [os.path.join(here, "./models")] + sys.path
sys.path = [os.path.join(here, "./serializers")] + sys.path
sys.path = [os.path.join(here, "../")] + sys.path
```

These statements are what allow function code to import packages in the `vendored`, `lib`, `models`, and `serializers` directories without having to use relative imports. If you do not wish to alter the environment's import path, the contents of this file can be removed. However, do not remove the file completely.

app/functions/

The functions directory is where your application's business logic lives. Later in the tutorial, when we start creating functions, we'll explain the naming and file conventions that should be followed within the `functions` directory.

For now all you need to know is that the directory is identified as a package, since it has a blank `__init__.py` file.

app/lib/

The `lib` directory is where you should keep modules and packages that are shared across functions but that aren't installable via pip.

app/models/

The `models` directory is where the domain objects that your application manipulates should be stored. Like the `function` directory, files placed in the `model` directory should conform to Tight's conventions.

Modules in this directory should define a single model and the name of the model and file should be the same.

Imagine creating an `Account` model:

```
$  ls -la app/models
drwxr-xr-x  4 user  group   136B Jan  2 15:27 .
drwxr-xr-x  8 user  group   272B Jan  2 14:56 ..
-rw-r--r--  1 user  group    80B Jan  2 15:27 Account.py
-rw-r--r--  1 user  group   460B Dec 23 10:46 __init__.py

$  less Account.py

def Account(id):
    """ Account factory """
    return {
        'id': id
    }
```

app/models/__init__.py

Unlike the other directories that get created inside of `app`, the `__init__.py` file inside of `models` is not empty. This file will loop through the files in the directory and automatically import the models that are defined. So long as the convention described above is followed, you will be able to succinctly import models into function modules.

The `Acccount` model defined above would be imported like so:

```
from Account import Account
```

app/serializers/

Tight encourages you to maintain serialization logic separately from model modules. As such, Tight provides a location where serializers can be kept.

app/vendored/

The `vendored` directory is where your _application's_ `pip` packages are installed.

app_index.py

This is the module that is used to route Lambda events to the correct function.

```
from app.vendored.tight.providers.aws.lambda_app import app as app
app.run()
```

The function `tight.providers.aws.lambda_app.run` collects functions from `app/functions` and sets attributes on the module for each function found. This means that when you go to configure your Lambda function within AWS, you can refer to module attributes that mirror functions:

```
app_index.a_function_in_your_app
```

This will call the `handler` function on the module located at `app/functions/a_function_in_your_app/handler.py`.

conftest.py

`conftest.py` provides the minimum confugration needed to run function tests. The module also imports the `tight.core.test_helpers` module, which exposes test fixtures and other goodies to help you start writing tests right away.

env.dist.yml

This file contains the default values for the environment variables that your application expects. You shouldn't store sensitive or environment specific values here. By default, this file specifies that the environment variables, `CI` and `STAGE` are expected:

```
# Define environment variables here
CI: False
STAGE: dev
```

As your application evolves remember to update this file with new names:

```
# Define environment variables here
CI: False
STAGE: dev
SOME_API_KEY: <optionally provide a default value>
```

requirements-vendor.txt

Specify pip package dependencies, which will be installed to `app/vendored` by default.

requirements.txt

Specify pip package dependencies that are to be installed to the virtual environment. Typically this is where you'll define dependencies that are required for developing and testing your app.

Dependencies specified here will not be packaged with your application artifact.

schemas/

This directory will contain CloudFormation compatible DynamoDb schemas, which can be auto-generated from model definitions.

tests/

Tight really wants to help you develop your application test-first. It would be a tragedy and an embarrasment if Tight didn't provide you a place to store your tests. Once we stater generating functions, we'll dive deeper into the structure of this directory.

tight.yml

`tight.yml` is this Tight app's configuration file. There's nothing too fancy about it and throughout the course of the tutorial, you'll rarely have to modify it. Just be aware that it exists and that it is the location from which the command line tool pulls the application name. Every time a `tight` command is run, this file is discovered and parsed and the values it defines are used throughout various commands.

6.2.3 Install Dependencies

Now that our application structure has been scaffolded, it's time to install our dependencies. First we'll install our virtual environment depedencies and then we'll install our application specific dependencies.

6.2.4 Environment Dependencies

Install environment dependencies just as you would for any other virtual environment.

```
$ pip install -r requirements.txt
```

6.2.5 App Dependencies

Application dependencies are also installed via `pip` but we need to be sure that they get installed to the correct location so that when our application artifact is deployed, any third-party libraries that your application relies on are available. To install your application dependencies, navigate to your project root and run `tight pip install --requirements`:

```
$ tight pip install --requirements
```

When you run this command, you'll notice that at the very end of the run you are notified that the `boto3` and `botocore` packages have been removed from the `app/vendored` directory. This is because both packages are supplied by the AWS Lamba execution environment. Generally, you shouldn't include these packages in your application artifact.

6.2.6 Conclusion

By now, you have scaffolded your first Tight app and should have a basic grasp of the purpose and reason for the auto-generated files and directories.

You also learned how to install your application and virtual environment dependencies.

Continue reading to learn about how Tight helps you initialize and manage application environment variables.

6.3 Generate Environment Files

6.3.1 Working With dist.env.yml

6.3.2 App Environment Variables

6.4 Generate A Function

6.4.1 Lambda Proxy Event Controllers

6.4.2 Test Stubs

6.5 Working with Tight Core

6.5.1 Lambda App

6.5.2 Lambda Proxy Event

6.5.3 Clients

Logger

Safeget

DynamoDb

6.5.4 Test Helpers

6.6 Models and Data

6.6.1 DynamoDb Support

6.6.2 Generate a Model

6.6.3 Install DynamoDb Locally

6.6.4 Running DynamoDb Locally

6.6.5 Testing DynamoDb Interactions

6.7 Working with Responses

6.7.1 Using Serializers

6.8 Creating an Artifact

6.9 Deploy to AWS

6.9.1 Create Lambda Function

CHAPTER 7

tight-cli

The tight-cli package is one of two components, which together form Tight: the toolset that helps you build event driven applications for serverless runtimes. tight-cli helps you scaffold and maintain Tight apps. This document describes the available commands exposed by tight-cli. For a more thorough discussion of how to use tight-cli to create and manage your application visit the tutorials.

Once installed, you can invoke tight-cli simply by calling tight from the command line:

```
$ tight

Usage: tight [OPTIONS] COMMAND [ARGS]...

Options:
  --help  Show this message and exit.

Commands:
  dynamo
  generate
  pip
```

7.1 tight generate

The generate group currently supports two sub-commands: app and function. Use these commands to quickly scaffold your application, functions, and tests.

7.1.1 tight generate app

```
Usage: tight generate app [OPTIONS] NAME

Options:
  --provider TEXT  Platform providers
```

(continues on next page)

(continued from previous page)

```
  --type TEXT      Provider app type
  --target TEXT    Location where app will be created.
  --help           Show this message and exit.
```

`tight generate app` only currently supports the default values for `provider` and `type`. Therefore, `NAME` is really all that is currently required. If you don't want to generate the app in the current directory, you can override the write target by specifying the `--target` option.

```
$ tight generate app my_service
$ cd my_service
$ ls -la

drwxr-xr-x  12 user  group   408B Dec 30 14:40 .
drwxr-xr-x@ 24 user  group   816B Dec 30 14:40 ..
-rw-r--r--   1 user  group   143B Dec 25 17:07 .gitignore
drwxr-xr-x   8 user  group   272B Dec 23 11:22 app
-rw-r--r--   1 user  group    76B Dec 25 16:38 app_index.py
-rw-r--r--   1 user  group   477B Dec 30 14:40 conftest.py
-rw-r--r--   1 user  group    56B Dec 26 02:28 env.dist.yml
-rw-r--r--   1 user  group    60B Dec 17 18:03 requirements-vendor.txt
-rw-r--r--   1 user  group    40B Dec 17 18:05 requirements.txt
drwxr-xr-x   4 user  group   136B Dec 23 12:39 schemas
drwxr-xr-x   4 user  group   136B Dec 23 11:22 tests
-rw-r--r--   1 user  group    55B Dec 30 14:40 tight.yml
```

7.1.2 `tight generate function`

Quickly generate a function and test stubs in a Tight app.

```
Usage: tight generate function [OPTIONS] NAME

Options:
  --provider [aws]       Platform providers
  --type [lambda_proxy]  Function type
  --help                 Show this message and exit.
```

This command will generate a function module and will also stub integration and unit tests for the generated module:

```
$ tight generate function my_controller
============================================= test session starts␣
↪=============================================
platform darwin -- Python 2.7.10, pytest-3.0.5, py-1.4.32, pluggy-0.4.0
rootdir: /Users/michael/Development/my_service, inifile:
collected 2 items

tests/functions/integration/my_controller/test_integration_my_controller.py .
tests/functions/unit/my_controller/test_unit_my_controller.py .

========================================== 2 passed in 0.10 seconds␣
↪==========================================
Successfully generated function and tests!
```

This command generates the following files and directories:

```
-app/
---functions/
-----my_controller/
-------handler.py
-tests/
---functions/
-----unit/
-------my_controller/
---------test_unit_my_controller.py
-----integration/
-------my_controller/
---------expectations/
-----------test_get_method.yml
---------placebos/
---------test_integration_my_controller.py
```

The contents of the generated files:

app/functions/my_controller/handler.py

```
from tight.providers.aws.clients import dynamo_db
import tight.providers.aws.controllers.lambda_proxy_event as lambda_proxy
db = dynamo_db.connect()

@lambda_proxy.get
def get_handler(*args, **kwargs):
    return {
        'statusCode': 200,
        'body': {
            'hello': 'world'
        }
    }

@lambda_proxy.post
def post_handler(*args, **kwargs):
    pass

@lambda_proxy.put
def put_handler(*args, **kwargs):
    pass

@lambda_proxy.patch
def patch_handler(*args, **kwargs):
    pass

@lambda_proxy.options
def options_handler(*args, **kwargs):
    pass

@lambda_proxy.delete
def delete_handler(*args, **kwargs):
    pass
```

tests/functions/integration/my_controller/test_integration_my_controller.py

```
import os, json
here = os.path.dirname(os.path.realpath(__file__))
from tight.core.test_helpers import playback, record, expected_response_body
```

(continues on next page)

(continued from previous page)

```
def test_get_method(app, dynamo_db_session):
    playback(__file__, dynamo_db_session, test_get_method.__name__)
    context = {}
    event = {
        'httpMethod': 'GET'
    }
    actual_response = app.my_controller(event, context)
    actual_response_body = json.loads(actual_response['body'])
    expected_response = expected_response_body(here, 'expectations/test_get_method.yml
↪', actual_response)
    assert actual_response['statusCode'] == 200, 'The response statusCode is 200'
    assert actual_response_body == expected_response, 'Expected response body matches␣
↪the actual response body.'
```

tests/functions/integration/my_controller/expectations/test_get_method.yml

```
body: '{"hello":"world"}'
headers: {Access-Control-Allow-Origin: '*'}
statusCode: 200
```

tests/functions/unit/my_controller/test_unit_my_controller.py

```
def test_no_boom():
    module = __import__('app.functions.my_controller.handler')
    assert module
```

7.1.3 `tight generate model`

Generate a Flywheel model and write to `app/models`.

```
Usage: tight generate model [OPTIONS] NAME

Options:
  --help  Show this message and exit.
```

Example:

```
$ tight generate model account
$ cd app/models
$ ls
-rw-r--r--  1 user  group   390B Dec 30 15:28 Account.py
-rw-r--r--  1 user  group   460B Dec 23 10:46 __init__.py
```

The generated model:

```
from flywheel import Model, Field, Engine
import os

# DynamoDB Model
class Account(Model):
    __metadata__ = {
        '_name': '%s-%s-accounts' % (os.environ['NAME'], os.environ['STAGE']),
        'throughput': {
            'read': 1,
```

(continues on next page)

(continued from previous page)

```
            'write': 1
        }
    }

    id = Field(type=unicode, hash_key=True)

    # Constructor
    def __init__(self, id):
        self.id = id
```

7.1.4 tight generate env

This command will generate a `env.yml` file, merging values defined in `env.dist.yml` and values in the current shell environment.

```
$ tight generate env
{CI: false, NAME: my-service, STAGE: dev}
```

7.2 tight pip

`tight pip` is a lightweight command that helps manage dependencies in the context of a Tight app.

7.2.1 tight pip install

```
Usage: tight pip install [OPTIONS] [PACKAGE_NAME]...

Options:
  --requirements / --no-requirements    Defaults to --no-requirements
  --requirements-file [``CWD``]         Requirements file location
  --target [``tight.yml::vendor_dir``]  Target directory.
  --help                                Show this message and exit.
```

Typically, after generating an app you'll want to run `tight pip install --requirements` from the application root directory. This will install the dependencies to the `app/vendored` directory and then remove the `boto3` and `botocore` packages; these libraries should not be shipped woth your app since they are provided by AWS in the default Lambda environment.

As you are developing a Tight app, you will undoubtedly need to install additional `pip` packages. You have two options for installing new dependencies. You can either add the dependency to `requirements-vendor.txt` and re-run `tight pip install --requirements` or you can run `tight pip install PACKAGE_NAME`, which will install the dependencies to `app/vendored` and then append `PACKAGE_NAME` to `requirements-vendor.txt`.

7.3 tight dynamo

One of Tight's primary goals is to make it quick and easy to scaffold RESTful APIs. To help achieve this goal, `tight-cli` provides a group of commands that helps you manage, run, and test interactions with DynamoDB.

7.3.1 tight dynamo installdb

Run this command to download and expand the latest stable version of DynamoDB. The downloaded tarball will be extracted to the directory `dynamo_db`.

7.3.2 tight dynamo rundb

This command will run the version of DynamoDB which was downloaded via `tight dynamo installdb`. This command runs dynamo using a shared database file which is written to `dynamo_db/shared-local-instance.db`.

This file is deleted on startup if it exsits.

Additionally, this command will traverse the `app/models` directory and automatically generate tables for models. Models should be instances of Flywheel models.

Before executing this command, you should have run `tight generate env` or otherwise have defined `app/env.yml`.

```
$ tight dynamo rundb

Initializing DynamoDB Local with the following configuration:
Port:       8000
InMemory:   false
DbPath:     ./dynamo_db
SharedDb:   true
shouldDelayTransientStatuses:       false
CorsParams: *

This engine has the following tables [u'my-service-dev-accounts']
```

As demonstrated in the example above, the command will report on the tables generated from auto-discovered model classes.

7.3.3 tight dynamo generateschema

This command will generate CloudFormation compatible DynamoDB resources from Flywheel models.

Given the following model:

```
from flywheel import Model, Field, Engine
import os

# DynamoDB Model
class Account(Model):
    __metadata__ = {
        '_name': '%s-%s-accounts' % (os.environ['NAME'], os.environ['STAGE']),
        'throughput': {
            'read': 1,
            'write': 1
        }
    }

    id = Field(type=unicode, hash_key=True)

    # Constructor
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, id):
        self.id = id
```

Running `tight dynamo generateschema` will write a YAML file to `app/schemas/dynamo`:

```
$ tight dynamo generateschema
$ cd schemas/dynamo
$ ls
-rw-r--r--  1 user  group   265B Dec 30 15:55 accounts.yml
```

The contents of `acounts.yml` will be:

```
Properties:
  AttributeDefinitions:
  - {AttributeName: id, AttributeType: S}
  KeySchema:
  - {AttributeName: id, KeyType: HASH}
  ProvisionedThroughput: {ReadCapacityUnits: 1, WriteCapacityUnits: 1}
  TableName: my-service-dev-accounts
Type: AWS::DynamoDB::Table
```

CHAPTER 8

tight

8.1 tight.providers

The tight package currently supports only one provider: AWS.

8.1.1 tight.providers.aws.lambda_app.app

Use this package to create an entry point module. Typical usage is quite simple, as demonstrated in the following code example. Read on to learn about how module internals work.

```
from app.vendored.tight.providers.aws.lambda_app import app
app.run()
```

When creating an app using tight-cli a file, *app_index.py*, will be automatically generated which will contain code like the snippet above.

tight.providers.aws.lambda_app.app.**collect_controllers**()

" Inspect the application directory structure and discover controller modules.

Given the following directory structure, located at TIGHT.APP_ROOT:

```
-app_index.py
-app/
---functions/
-----controller_a/
-------handler.py
-----controller_b/
-------handler.py
-----not_a_controller/
-------some_module.py
```

Descend into TIGHT.APP_ROOT/app/functions and collect the names of directories that contain a file named handler.py. The directory structure above would produce the return value:

```
['controller_a', 'controller_b']
```

Return type list

Returns A list of application controller names.

`tight.providers.aws.lambda_app.app.`**`create`**`(`*`current_module`*`)`

Attach functions to the app entry module.

Introspect the application function root and create function attributes on the provided module that map to each application controller. An application controller is defined as any directory in the app root that contains a *handler.py* file. The name of the controller is the enclosing directory.

Given the following app structure:

```
-app_index.py
-app/
---functions/
-----controller_a/
-------handler.py
-----controller_b/
-------handler.py
-----not_a_controller/
-------some_module.py
```

The controller names collected would be:

`controller_a` and `controller_b`

Notice that *not_a_controller* is omitted because there is no *handler.py* file in the directory.

Assuming that `app_index.py` is the module from which `create` is called, the result would be that `app_index.py` will behave as if it had been statically defined as:

```
def controller_a(controller_module_path, controller_name, event, context):
    controller_module_path # 'app.functions.controller_a.handler'
    controller_name # controller_a
    callback = importlib.import_module(controller_module_path, 'handler')
    return callback.handler(event, context, **kwargs)

def controller_b(controller_module_path, controller_name, event, context):
    controller_module_path # 'app.functions.controller_b.handler'
    controller_name # controller_b
    callback = importlib.import_module(controller_module_path, 'handler')
    return callback.handler(event, context, **kwargs)
```

This means that the handler value provided to lambda can follow the format:

```
'app_index.controller_a'
'app_index.controller_b'
```

So long as `app.functions.controller_a.handler` and `app.functions.controller_b.handler` define functions that are decorated by `tight.providers.aws.controllers.lambda_proxy_event` the call to `app_index.controller_a` or `app_index.controller_b` will in turn call the correct handler for the request method by mapping `event['httpMethod']` to the correct module function.

`tight.providers.aws.lambda_app.app.`**`run`**`()`

Call create on `sys.modules['app_index']` and catch any errors.

Typical usage would be to import this module and call `run` immediately:

```
from app.vendored.tight.providers.aws.lambda_app import app
app.run()
```

8.2 tight.core.logger

tight.core.logger.**error**(**args*, ***kwargs*)

tight.core.logger.**info**(**args*, ***kwargs*)

Log a message using the system logger.

Parameters

- **args** –
- **kwargs** –

Returns None

tight.core.logger.**warn**(**args*, ***kwargs*)

Python Module Index

t

tight.core.logger, 31
tight.providers.aws.lambda_app.app, 29

Index

C

collect_controllers() (in module tight.providers.aws.lambda_app.app), 29
create() (in module tight.providers.aws.lambda_app.app), 30

E

error() (in module tight.core.logger), 31

I

info() (in module tight.core.logger), 31

R

run() (in module tight.providers.aws.lambda_app.app), 30

T

tight.core.logger (module), 31
tight.providers.aws.lambda_app.app (module), 29

W

warn() (in module tight.core.logger), 31