flaskerize

Feb 22, 2022

Contents:

1 Quick Start Guide **3**

1.1 Step 1: Setting up Flaskerize 3

1.2 Step 2: Installing Flaskerize 3

1.3 Step 3: Creating a Flask API 4

1.4 Step 4: The Structure of your Flask API 9

1.5 Step 5: Adding Entities to an API 10

1.6 Step 6: Over To You 13

2 Contributing to flaskerize **15**

2.1 Contributing to the Source Code 15

2.2 Contributing to the Documentation 15

3 Glossary of Terms **17**

4 Indices and tables **19**

Index **21**

flaskerize is a code generation and project modification command line interface (CLI) written in Python and created for Python. It is heavily influenced by concepts and design patterns of the Angular CLI available in the popular JavaScript framework Angular. In addition to vanilla template generation, flaskerize supports hooks for custom run methods and registration of user-provided template functions. It was built with extensibility in mind so that you can create and distribute your own library of *schematics* for just about anything.

Use **flaskerize** for tasks including:

- Generating resources such as Dockerfiles, new **flaskerize** *schematics* , blueprints, yaml configs, SQLAlchemy entities, or even entire applications, all with functioning tests
- Upgrading between breaking versions of projects that provide flaskerize upgrade *schematics* with one command
- Bundling and serving static web applications such as Angular, React, Gatsby, Jekyll, etc within a new or existing Flask app.
- Registering Flask resources as new routes within an existing application
- Creating new *schematics* for your own library or organization

CHAPTER 1

Quick Start Guide

This guide is designed to get you up and running by showing you how to create a new Flask API using Flaskerize.

1.1 Step 1: Setting up Flaskerize

We're going to start from nothing, and over the course of this quickstart we'll end up with a simple API.

First, let's create a folder for our Flask API to live in.

```
mkdir flaskerize-example
cd flaskertize-example
```

Now, let's set up a virtual environment for our API project, activate it, and then upgrade `pip` within that environment.

```
python -m venv venv
source venv/bin/activate
pip install --upgrade pip
```

Note: The last command, `pip install --upgrade pip`, ensures that we have the latest version of `pip` installed.

1.2 Step 2: Installing Flaskerize

We're now ready to install Flaskeriez. Let's use *pip* to do just that. . .

```
pip install flaskerize
```

Once this command has completed we'll have installed Flaskerize along with its dependencies. If you want to see the packages that were installed, run the following command:

```
pip list
```

This should show you something like this…

```
$ pip list
Package      Version
------------ -------
appdirs      1.4.3
Click        7.0
Flask        1.1.1
flaskerize   0.12.0
fs           2.4.11
itsdangerous 1.1.0
Jinja2       2.10.1
MarkupSafe   1.1.1
pip          19.2.3
pytz         2019.2
setuptools   40.8.0
six          1.12.0
termcolor    1.1.0
Werkzeug     0.16.0
```

Note: The exact versions shown here may differ from the ones you see when you install **flaskerize**.

You should now have access to the `fz` command, verify this with `fz --help`, which should display something like the following:

```
$ fz --help
Flaskerizing...
usage: fz [-h] {attach,bundle,generate} [{attach,bundle,generate} ...]

positional arguments:
  {attach,bundle,generate}
                        Generate a new resource

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

1.3 Step 3: Creating a Flask API

You're we're now ready to create our Flask API, and we're going to use **flaskerize** to do most of this for us.

flaskerize has a number of generators that generate code and configuration for us. These generators use *schematics* to define exactly what code should be built. There are a number of *schematics* build into **flaskerize**.

We're going to start by using the `flask-api` generator to create a simple Flask API.

From the root of your project folder, run the following command:

```
fz generate flask-api my_app
```

You'll see output similar to the following:

```
$ fz generate flask-api my_app
Flaskerizing...

Flaskerize job summary:

        Schematic generation successful!
        Full schematic path: flaskerize/schematics/flask-api



        13 directories created
        40 file(s) created
        0 file(s) deleted
        0 file(s) modified
        0 file(s) unchanged

CREATED: flaskerize-example/.pytest_cache
CREATED: flaskerize-example/.pytest_cache/v
CREATED: flaskerize-example/.pytest_cache/v/cache
CREATED: flaskerize-example/my_app
CREATED: flaskerize-example/my_app/__pycache__
CREATED: flaskerize-example/my_app/app
CREATED: flaskerize-example/my_app/app/__pycache__
CREATED: flaskerize-example/my_app/app/test
CREATED: flaskerize-example/my_app/app/test/__pycache__
CREATED: flaskerize-example/my_app/app/widget
CREATED: flaskerize-example/my_app/app/widget/__pycache__
CREATED: flaskerize-example/my_app/commands
CREATED: flaskerize-example/my_app/commands/__pycache__
CREATED: .gitignore
CREATED: .pytest_cache/.gitignore
CREATED: .pytest_cache/CACHEDIR.TAG
CREATED: .pytest_cache/README.md
CREATED: .pytest_cache/v/cache/lastfailed
CREATED: .pytest_cache/v/cache/nodeids
CREATED: .pytest_cache/v/cache/stepwise
CREATED: my_app/README.md
CREATED: my_app/__pycache__/manage.cpython-37.pyc
CREATED: my_app/__pycache__/wsgi.cpython-37.pyc
CREATED: my_app/app/__init__.py
CREATED: my_app/app/__pycache__/__init__.cpython-37.pyc
CREATED: my_app/app/__pycache__/config.cpython-37.pyc
CREATED: my_app/app/__pycache__/routes.cpython-37.pyc
CREATED: my_app/app/app-test.db
CREATED: my_app/app/config.py
CREATED: my_app/app/routes.py
CREATED: my_app/app/test/__init__.py
CREATED: my_app/app/test/__pycache__/__init__.cpython-37.pyc
CREATED: my_app/app/test/__pycache__/fixtures.cpython-37.pyc
CREATED: my_app/app/test/fixtures.py
CREATED: my_app/app/widget/__init__.py
CREATED: my_app/app/widget/__pycache__/__init__.cpython-37.pyc
CREATED: my_app/app/widget/__pycache__/controller.cpython-37.pyc
CREATED: my_app/app/widget/__pycache__/interface.cpython-37.pyc
CREATED: my_app/app/widget/__pycache__/model.cpython-37.pyc
CREATED: my_app/app/widget/__pycache__/schema.cpython-37.pyc
CREATED: my_app/app/widget/__pycache__/service.cpython-37.pyc
```

(continues on next page)

(continued from previous page)

```
CREATED: my_app/app/widget/controller.py
CREATED: my_app/app/widget/interface.py
CREATED: my_app/app/widget/model.py
CREATED: my_app/app/widget/schema.py
CREATED: my_app/app/widget/service.py
CREATED: my_app/commands/__init__.py
CREATED: my_app/commands/__pycache__/__init__.cpython-37.pyc
CREATED: my_app/commands/__pycache__/seed_command.cpython-37.pyc
CREATED: my_app/commands/seed_command.py
CREATED: my_app/manage.py
CREATED: my_app/requirements.txt
CREATED: my_app/wsgi.py
```

Navigate into the *my_app* directory that was just created and list the files in that directory:

```
$ cd my_app
$ ls -al
total 32
drwxr-xr-x  9 bob  staff   288  4 Oct 15:01 .
drwxr-xr-x  6 bob  staff   192  4 Oct 15:01 ..
-rw-r--r--  1 bob  staff  1063  4 Oct 15:01 README.md
drwxr-xr-x  4 bob  staff   128  4 Oct 15:01 __pycache__
drwxr-xr-x  9 bob  staff   288  4 Oct 15:01 app
drwxr-xr-x  5 bob  staff   160  4 Oct 15:01 commands
-rw-r--r--  1 bob  staff   673  4 Oct 15:01 manage.py
-rw-r--r--  1 bob  staff   409  4 Oct 15:01 requirements.txt
-rw-r--r--  1 bob  staff   141  4 Oct 15:01 wsgi.py
```

As you can see, a number of files and folders have been created. One of the files that was just created is a `README.md` markdown file. If you open that file in a text editor find instructions on settng up your API. Those instructions are repeated here for convinience, but I'd recommend you take a look at `README.md` file regardless.

1.3.1 Following the Instructions from README.md

First, use `pip install` to install the requirements of your new API

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

Next, initialize the database

```
python manage.py seed_db
```

This step create a local SQLite database file.

Note: Type "Y" to accept the message. This check is there to prevent you accidentally deleting things.

1.3.2 Confirm your API is working

You're now ready to confirm that your API is working.

You can use the Flask command line interface to confirm that your Flask API is working by first using the `flask routes` command. This will print out all of the routes supported by your Flask API:

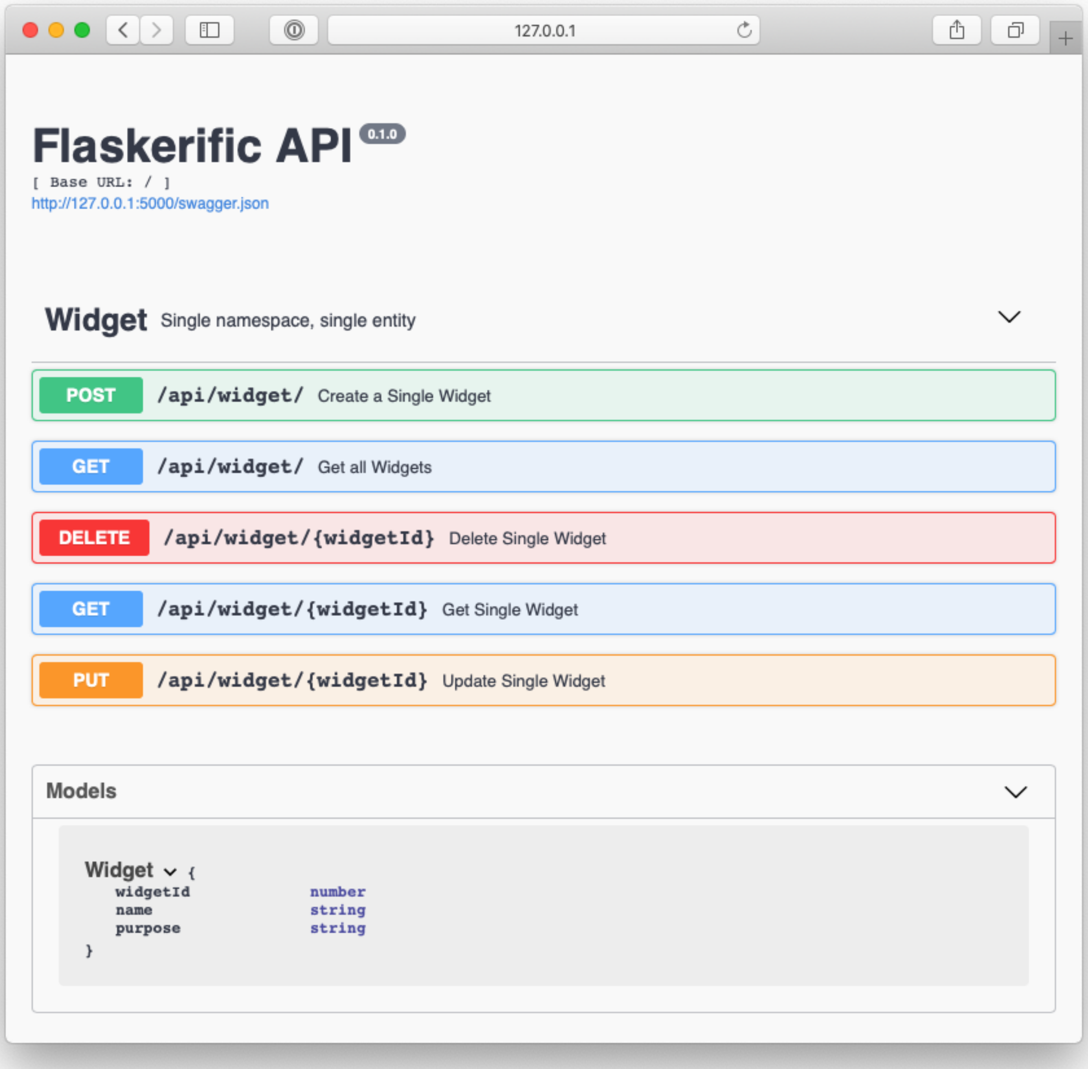
```
$ flask routes
Endpoint                     Methods           Rule
-------------------------    ----------------  ---------------------------
Widget_widget_id_resource    DELETE, GET, PUT  /api/widget/<int:widgetId>
Widget_widget_resource       GET, POST         /api/widget/
doc                          GET               /
health                       GET               /health
restx_doc.static          GET               /swaggerui/<path:filename>
root                         GET               /
specs                        GET               /swagger.json
static                       GET               /static/<path:filename>
```

As you can see, a number of routes have been generated.

Now, you can run your Flask API using `flask run` or by running `python wsgi.py`:

```
$ python wsgi.py
* Serving Flask app "app" (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: on
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger PIN: 304-898-518
```

While the Flask app is running, open http://127.0.0.1:5000/health within your favourite browser, and you should be greated with the Swagger documentation for your API.


Flaskerific API 0.1.0
[Base URL: /]
http://127.0.0.1:5000/swagger.json

Widget Single namespace, single entity

POST /api/widget/ Create a Single Widget
GET /api/widget/ Get all Widgets
DELETE /api/widget/{widgetId} Delete Single Widget
GET /api/widget/{widgetId} Get Single Widget
PUT /api/widget/{widgetId} Update Single Widget

Models

```
Widget {
    widgetId        number
    name            string
    purpose         string
}
```

You can use this UI to try getting all of the Widgets from your API. Alternatively, you can use the command line to call your api using `curl`. Execute the following command:

`curl -X GET "http://127.0.0.1:5000/api/widget/" -H "accept: application/json"`

This should return a JSON response, containing the entity details for the 3 Widgets currently stored in your SQL Lite database.

```
$ curl -X GET "http://127.0.0.1:5000/api/widget/" -H "accept: application/json"
[
    {
        "name": "Pizza Slicer",
        "widgetId": 1.0,
        "purpose": "Cut delicious pizza"
    },
    {
```

(continues on next page)

(continued from previous page)

```
        "name": "Rolling Pin",
        "widgetId": 2.0,
        "purpose": "Roll delicious pizza"
    },
    {
        "name": "Pizza Oven",
        "widgetId": 3.0,
        "purpose": "Bake delicious pizza"
    }
]
```

1.3.3 What Now?

flaskerize has very quickly set up a Flask API for you, including. . .

- the core API, and all the plumbing to set up routes
- an entity called “Widget”
- code to set up and seed a local database
- tests

In the next section we’ll dig deeper into what happened when you ran `fz generate flask-api my_app`, the structure of your Flask API, and what each of the generated files do.

1.4 Step 4: The Structure of your Flask API

In the previous step we created a Flask API using the **flaskerize** command `fz generate flask-api my_app`. This generated a number of file and folders, so let’s take a look at what you have.

The set of files and folders that were created are illustrated below:

```
.
├── README.md
├── app
│   ├── __init__.py
│   ├── config.py
│   ├── routes.py
│   ├── test
│   │   ├── __init__.py
│   │   └── fixtures.py
│   └── widget
│       ├── __init__.py
│       ├── controller.py
│       ├── interface.py
│       ├── model.py
│       ├── schema.py
│       └── service.py
├── commands
│   ├── __init__.py
│   └── seed_command.py
├── manage.py
├── requirements.txt
└── wsgi.py
```

Let's take a closer look at what these files do.

name	description
README.md	A markdown file containing instructions for setting up and running your Flask API
app	This folder contains your Flask API code
commands	This folder contains the code that seeds the database with data
manage.py	Exposes the database setup commands
requirements.txt	Contains the list of dependencies. Used for `pip install -r requirements.txt`
wsgi.py	Contains code that creates an instance of your Flask API

1.4.1 Entities

Within the `app`` folder you can see there's folder called ``widget`. This folder contains code related to the `widget` entity.

Each entity folder contains:

- `controller.py` - contains
- `interface.py` - contains
- `model.py` - contains
- `schema.py` - contains
- `service.py` - contains

You can read more about this structure in the following blog post:

http://alanpryorjr.com/2019-05-20-flask-api-example/

In the next part of this tutorial we will add an additional entity to our api.

1.5 Step 5: Adding Entities to an API

Over the previous steps we've built our Flask API. It already has a `widget` *entity*, but now we're going to add another *entity*.

We are going to add a `cake` *entity*.

To do this we're going to use another of **flaskerize's** *schematics*; the `entity` schematic.

From within the `my_app` folder we'll use the following command to generate our `cake` entity:

```
fz generate entity app/cake
```

This command will generate an entity, called cake, within the `app` folder.

```
$ fz generate entity app/cake
Flaskerizing...

Flaskerize job summary:

        Schematic generation successful!
        Full schematic path: flaskerize/schematics/entity



        1 directories created
        11 file(s) created
        0 file(s) deleted
        0 file(s) modified
        0 file(s) unchanged

CREATED: flaskerize-example/my_app/app/cake
CREATED: app/cake/__init__.py
CREATED: app/cake/controller.py
CREATED: app/cake/controller_test.py
CREATED: app/cake/interface.py
CREATED: app/cake/interface_test.py
CREATED: app/cake/model.py
CREATED: app/cake/model_test.py
CREATED: app/cake/schema.py
CREATED: app/cake/schema_test.py
CREATED: app/cake/service.py
CREATED: app/cake/service_test.py
```

So, what just happened?

- A folder named `cake` was created under the `app` folder. Everything related to the `cake` entity lives within this folder.
- A set of python files relating to the `cake` entity were created
- A set of tests, relating to the `cake` entity were also created

1.5.1 Wiring Up the New Cake Entity

If you run the `flask routes` command, or run `python wsgi.py`, you won't see any additional routes and you won't see your `cake` entity appear within the Swagger docs.

This is because there's some manual wire-up that you now need to do.

First, we need to edit the code within `my_app/app/routes.py`. Open this file in a text editor and add the following 2 lines of code (each addition has a comment starting with `ADD THE FOLLOWING LINE` above it):

```
def register_routes(api, app, root="api"):
    from app.widget import register_routes as attach_widget

    # ADD THE FOLLOWING LINE to import the register_routes function
```

(continues on next page)

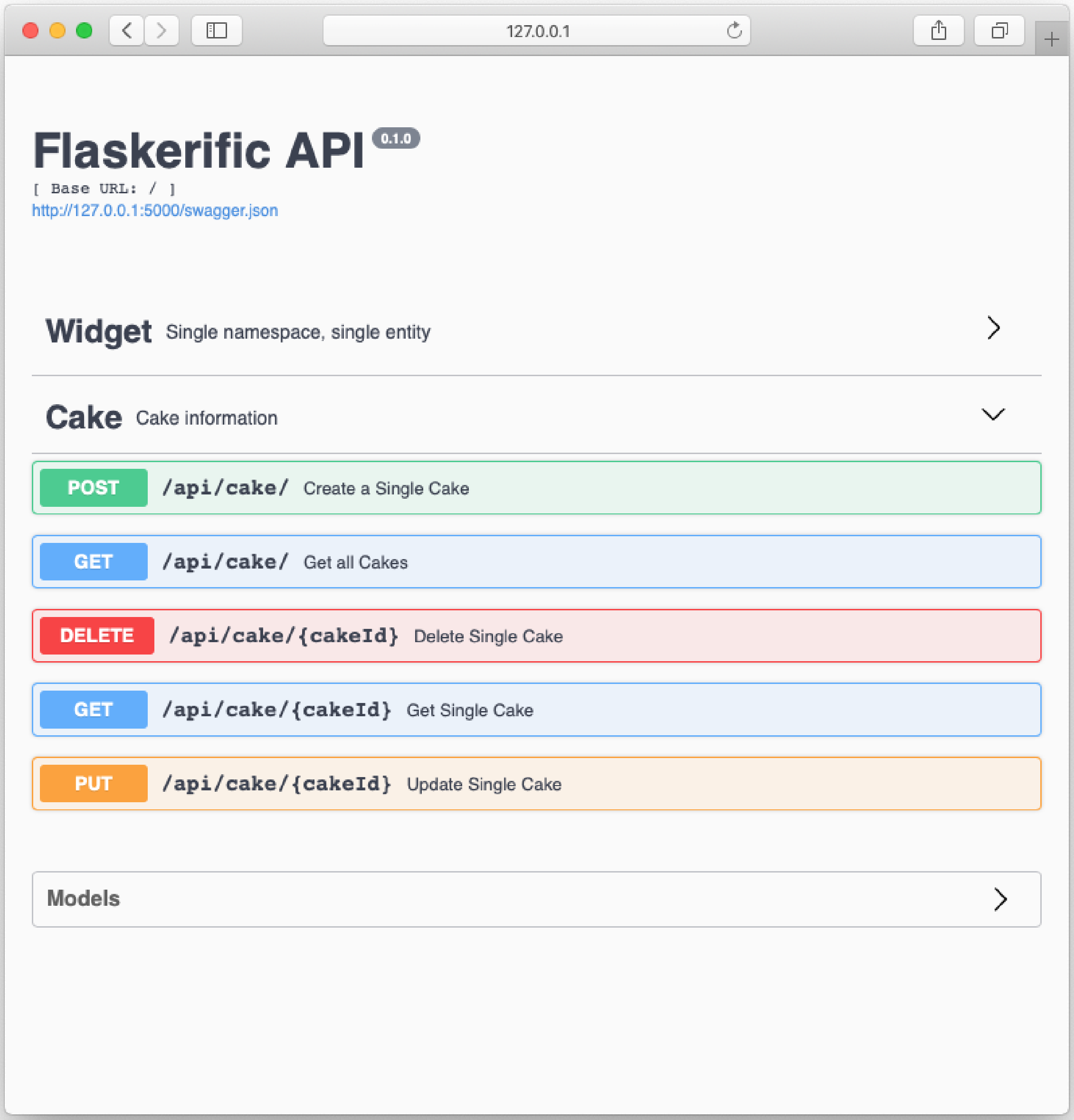
(continued from previous page)

```
    from app.cake import register_routes as attach_cake

    # Add routes
    attach_widget(api, app)

    # ADD THE FOLLOWING LINE to register the routes for the cake entity
    attach_cake(api)
```

Now, when you run `flask route` you'll see the additional routes for your `cake` entity. Additionally, you can now see the `cake` entity appear in the Swagger docs UI:



Flaskerific API 0.1.0

[Base URL: /]
http://127.0.0.1:5000/swagger.json

Widget Single namespace, single entity

Cake Cake information

POST /api/cake/ Create a Single Cake

GET /api/cake/ Get all Cakes

DELETE /api/cake/{cakeId} Delete Single Cake

GET /api/cake/{cakeId} Get Single Cake

PUT /api/cake/{cakeId} Update Single Cake

1.6 Step 6: Over To You

Over the last few steps you've created a Flask API, and added a new entity to it.

flaskerize has allowed you to quickly and easily generated code, and unit tests, for your API.

There are plenty of additional tasks for you to complete now, such as defining what your entity should look like, populating the database, writing meaningful tests etc. However, at least you now have a framework in which to write that code, and as you add more entities you'll use **flaskerize** to automate that job.

There are plenty **flaskerize** features that we've not covered here. This Quick Start was designed to give you just a brief taste of what's possible.

Good luck, and have fun using **flaskerize**!

1.6.1 Further Reading

Blog Post "Flask best practices"

http://alanpryorjr.com/2019-05-20-flask-api-example/

The flaskerize README

https://github.com/apryor6/flaskerize/

Schematics Build Into flaskerize

https://github.com/apryor6/flaskerize/tree/master/flaskerize/schematics

It's assumed that you have Python 3.7 installed. If not, go and install Python now.

https://www.python.org/downloads/

The instructions in this guide also assume you're comfortable using the command line. The illustrations that you'll see in this quick start are taken from a bash terminal. In general, the commands will work in your chosen terminal.

OK. . . .let's get started. . .

CHAPTER 2

Contributing to flaskerize

2.1 Contributing to the Source Code

TODO: Instructions here

2.2 Contributing to the Documentation

TODO: Instructions here

CHAPTER 3

Glossary of Terms

entity An entity is a combination of a Marshmallow schema, type-annotated interface, SQLAlchemy model, Flask controller, and CRUD service. It also contains tests and provides functionality for being registered within an existing Flask application via its register_routes method. This blog post gives more details on entities.

schematics Schematics generate code from parameterized templates. **flaskerize** ships with a bunch of built in schematics, listed here

CHAPTER 4

Indices and tables

- genindex
- search

Index

E

`entity`, **17**

S

`schematics`, **17**