Flask Atomic

Keith Byrne

Dec 27, 2020

CONTENTS

Flask Atomic leverages the power of Flask Blueprints & SQLAlchemy to provide a plugin application builder. Blueprints are essentially mini Flask applications that are registered to the Flask application. Encapsulated within these Blueprints are all routes, templates etc…

When developing Flask applications and working with large amounts of models where boilerplate CRUD operations are required. With well-defined code, Flask Atomic has the opportunity to render potentially hundreds of lines of redundant.

This project was heavily influenced by repetitive efforts to get quick and dirty APIs up and running, with the bad practice of re-using a lot of code, over and over again. Instead of relying on throwaway efforts, Flask Atomic provides a very simply means to abstract away code and enable RESTful API best practices that are often esoteric and difficult to engineer for small projects.

This library also contains a few helpers and other utilities that I have found helpful.

CHAPTER ONE

INSTALLATION

Note: Package was developed using Python 3.6. No guarantee is given for backwards compatibility with older versions of Python. Especially Python < 3.x. If you have to ask if this works with Python 2, perhaps you should consider migrating.

Warning: This guide assumes a foundation understanding of Flask and Blueprints. Blueprints are essentially smaller application units that fit into the larger Flask application instance. This is akin to ‘apps’ in Django.

1.1 Installation

From the PyPi repository

```
pip install Flask-Atomic
```

CHAPTER

TWO

BASIC USAGE

Lets start with a basic usage example and illustrate the core features Flask Atomic offers.

2.1 Architect Blueprint

The Architect Blueprint is itself yes, you guessed it a Flask Blueprint. It looks, smells and sounds just the same as a normal Flask Blueprint but does a little magic for you.

All that's needed to start is a SQLAlchemy Model to get things started.

```
class SomeModel(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.Datetime, default=datetime.now())
    username = db.Column(db.String(50))
    alias = db.Column(db.String(50))
```

When we have models like this we typically will have a GET route, POST route maybe DELETE and PUT depending on the application. Typically, these routes will have similar behaviours across different application: define a route, take some data, create a model, save to database.

```
from flask_atomic import Architect

from .models import SomeModel

monitor_blueprint = Architect(SomeModel)

app = Flask(__name__)
app.register_blueprint(monitor_blueprint, url_prefix='example')

if __name__ == '__main__':
    app.run()
```

What has been created?

| :Route | Method | Comment | | ———————————— | ——— | ——————————————————————————————————
| | /example | GET | Index endpoint for the model, fetches all data. | | /example/ | GET | Get one endpoint. Defaults to model primary key. | | /example// | GET | Get one endpoint, returning field. Can be relationships. | | /example | POST | Post endpoint that allows for creation of new records. | | /example/ | DELETE | Delete endpoint, defaults to deleting based on PK lookup. | | /example/ | PUT | Put endpoint that allows for making modifications to a record. | | /example/ | HEAD | Simply checks a record exists and returns HTTP 200 if so, 404 if not. |

Making a curl request to your API and see it in action:

```
curl localhost:5000/example
```

Using the example model above, lets check posting new records:

```
curl --header "Content-Type: application/json" \
  --request POST \
  --data '{"name":"xyz","username":"xyz","alias":"xyz}' \
  http://localhost:5000/example
```

CHAPTER

THREE

FLASK ATOMIC RECIPES

CHAPTER

FOUR

API

4.1 Indices and tables

- genindex
- modindex
- search