Analytics Vidhya

Building REST APIs using Flask-RESTPlus, SQLAlchemy & Marshmallow

Suman Das
Dec 18, 2020 · 13 min read

flask-Restplus with SqlAlchemy

Python is an interpreted, high-level, and general-purpose programming language. It can be used to develop business applications as well as system scripts, has data science related capabilities, desktop GUIs, and also offers many choices for web and internet development.

Flask is a micro web framework written in Python that helps you build web applications easily. It does not require tools or libraries and has no database abstraction layer, form validation, serializers, user management, or built-in internationalization. However, it supports extensions that can add application features as if they were implemented in Flask itself. An aspiring Flask developer must then choose the right extensions and combine them together to get the right set of functions, as applicable.

The aim of this tutorial is working with Flask extensions that help us create a production environment ready Python application and to integrate with Swagger UI without a hitch. We will learn to build a Rest API using Flask extensions such as ***Flask-RestPlus***, ***Flask-Marshmallow***, and ***Flask-SQLAlchemy*** and share the ***API*** using **Swagger UI**.

Flask and Flask-RestPlus

Flask is a lightweight web server and framework. Although, we can create a Web API directly with `flask`, the Flask-RESTPlus extension makes it simpler to build Rest APIs.

- It supports namespaces, which are ways of creating prefixes and structuring the code.
- It has a full solution for parsing and validating the input parameters. This means that we have an easy way of dealing with endpoints that require several parameters and validate them.
- Its best feature is the ability to automatically generate interactive documentation for our API using **Swagger UI**.

Flask-Marshmallow

Marshmallow is an ORM/ODM/framework-agnostic library developed to simplify the process of serialization and deserialization. The Flask-Marshmallow extension acts as a thin integration layer for Flask and Marshmallow that adds additional features to Marshmallow, including URL and hyperlink fields for HATEOAS-ready APIs. It also integrates with **Flask-SQLAlchemy** and reduces some boilerplate code.

Flask-SQLAlchemy

SQLAlchemy is a library that facilitates the communication between Python programs and databases that gives application developers the full power and flexibility of SQL. The Flask-SQLAlchemy extension simplifies the usage of SQLAlchemy with the Flask application by providing useful defaults and extra helpers that make it easier to accomplish common tasks. It provides an ORM for us to modify application data by easily creating defined models.

Swagger UI

Swagger UI helps to generate interactive documentation that makes it much easier to test the Rest API as well as share the API documentation with other users. Hence, without a Swagger UI the Flask API is incomplete.

. . .

Following are the steps required to create a sample Flask-based API for an Item & Store management application:

1. Setup and Installation
2. Integrate Flask-Marshmallow
3. Integrate Flask-SQLAlchemy
4. Create Models
5. Create Schemas
6. Create Resources
7. Application Entry Point
8. Test the API

Prerequisites

We require Python 3 with **Pipenv** and Git installed. Pipenv is a package and a virtual environment manager which uses `PIP` under the hood. It provides more advanced features like version locking and dependency isolation between projects.

1. Setup and Installation

Once the prerequisites are in place we can begin creating the application.

a) Create a Sample Item & Store Management Flask Application

To begin with, our application, create a folder called `python-sample-flask-application` in any directory on the disk for our project.

```
$ cd /path/to/my/workspace/
$ mkdir python-sample-flask-application
$ cd python-sample-flask-application
```

Navigate to the project folder.

b) Activate Virtual Environment

Once we are inside the project folder, execute the following commands to activate the VirtualEnv.

```
pipenv shell
```

The virtual environment will now be activated, which will provide the required project isolation and version locking.

c) Install Dependencies

Next, install all the required dependencies using Pipenv as shown.

```
pipenv install flask-restplus
pipenv install flask-marshmallow
pipenv install flask-sqlalchemy
pipenv install marshmallow-sqlalchemy
```

After we execute the above commands, the required dependencies will be installed.

We can see now two files, which have been created inside our project folder, namely, `Pipfile` and `Pipfile.lock`.

- `Pipfile` contains all the names of the dependencies we just installed.
- `Pipfile.lock` is intended to specify, based on the dependencies present in `Pipfile`, which specific version of those should be used, avoiding the risks of automatically upgrading dependencies that depend upon each other and breaking your project dependency tree.

All the dependency versions that were installed while executing the above commands are as shown below.

```
flask-restplus = "==0.13.0"
flask-marshmallow = "==0.14.0"
flask-sqlalchemy = "==2.4.4"
marshmallow-sqlalchemy = "==0.24.1"
```

The dependency version will depend upon the latest version available at that point in time. If required, we can also update or specify the version number of any dependency in the `Pipfile` using the syntax `flask-restplus = "==0.13.0"`

d) Update 'Werkzeug' Dependency Version

When we install `flask-restplus`, **werkzeug**, which is a comprehensive WSGI web application library, is installed automatically with version `1.0.1`. We can locate that entry in `Pipfile.lock` file as follows:

```
"werkzeug": {
    "hashes": [

"sha256:2de2a5db0baeae7b2d2664949077c2ac63fbd16d98da0ff71837f7d1dea3
fd43",

"sha256:6c80b1e5ad3665290ea39320b91e1be1e0d5f60652b964a3070216de83d2
e47c"
    ],
    "version": "==1.0.1"
}
```

However, **flask-restplus** is currently not compatible with extension "**werkzeug=1.0.1**". Hence, we need to downgrade the version to `werkzeug==0.16.1` as suggested in the Github: https://github.com/noirbizarre/flask-restplus/issues/777#issuecomment-583235327.

Accordingly, let us downgrade the version of `werkzeug` in `Pipfile` and update the installed dependencies.

```
[[source]]
name = "pypi"
url = "https://pypi.org/simple"
verify_ssl = true

[dev-packages]

[packages]
flask-restplus = "==0.13.0"
flask-marshmallow = "==0.14.0"
werkzeug = "==0.16.1"
flask-sqlalchemy = "==2.4.4"
marshmallow-sqlalchemy = "==0.24.1"

[requires]
python_version = "3.7"
```

To update the installed dependencies, execute the following command.

```
pipenv update
```

Now, we are all set to write some code for our application.

2. Integrate Flask-Marshmallow

To integrate Flask-Marshmallow with our application, create a file `ma.py` with the following content.

```
from flask_marshmallow import Marshmallow
ma = Marshmallow()
```

Here, we have imported Marshmallow from `flask_marshmallow`. Later on, we will use this Marshmallow instance `ma` to integrate with the flask application using the command `ma.init_app(app)`.

3. Integrate Flask-SQLAlchemy

For Flask-SqlAlchemy integration, create a file called `db.py` having the following content.

```
from flask_sqlalchemy import SQLAlchemy
db = SQLAlchemy()
```

Here, we have imported SQLAlchemy from `flask_sqlalchemy`. We will also use this SQLAlchemy instance `db` to integrate with the flask application using the command `db.init_app(app)`.

4. Create Database Models

Next, we will create database models for our data storage and organization. For our application, we need to create two database models Item and Store. We will be using `db`, the instance of the SQLAlchemy from Flask-SQLAlchemy, which we created earlier (Step 3) to create our models.

The `db` instance contains all the functions and helpers from both **`sqlalchemy`** and sqlalchemy.orm. It provides a class called `Model` that is a declarative base, which can be used to declare our models.

Create the `models` package and add two files named `Item.py` and `Store.py`.

4.1 Item.py

The Item.py file should contain the following content.

The above code in `item.py` does the following:

- We started off by creating the ItemModel class in `line 5`.
- In `line 6`, we declared the table name `items` where this model will be mapped to.
- From `line 8 to 10`, we defined the table columns along with their dataType.
- We defined the relationship with StoreModel in `line 12 and 13.`
- From `line 26` to `36`, we added some helper methods to perform search operation on `items` table.
- From `line 38` to `40`, we defined the `save_to_db` method to save the items in the database.
- From `line 42` to `44`, we defined the `delete_from_db` method to delete the items from the database.

4.2 Store.py

The Store.py file should contain the following content.

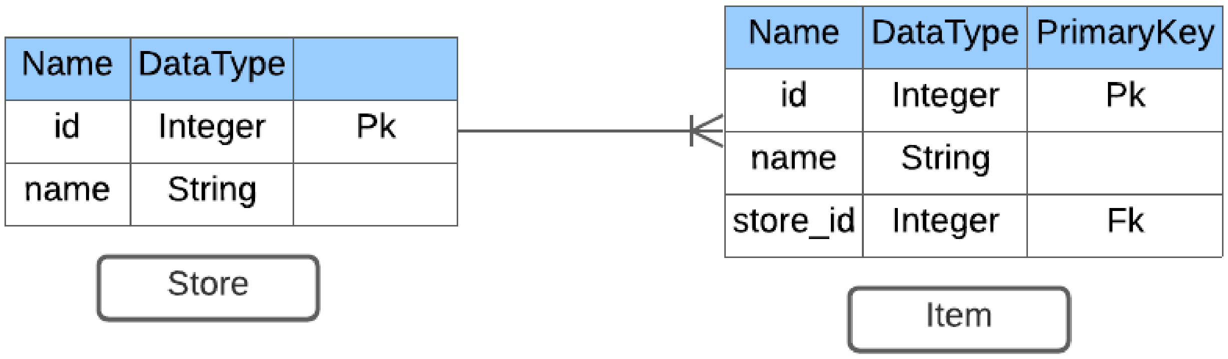
Store.py

The above code in `store.py` does the following:

- We started off by creating the StoreModel class in `line 5`.
- In `line 6`, we declared the table name `stores` where this model will be mapped to.
- From `line 7 to 8`, we defined the table columns along with their dataType.
- We defined the relationship with ItemModel in `line 10.`

- From `line 17` to `27`, we added some helper methods to perform search operation on `stores` table.
- From `line 29` to `31`, we defined the `save_to_db` method to save the stores in the database.
- From `line 33` to `35`, we defined the `delete_from_db` method to delete the stores from the database.

The `Item` and `Store` classes inherit from `db.Model` class, which declares the class as a model for SQLAlchemy. The required Columns are defined in the Model class. Also, we define the relationship between Item and Store. The Store has a one-to-many relationship with Item.



Entity Diagram

5. Create Schemas

Create the Marshmallow `Schemas` from our models defined earlier using `SQLAlchemyAutoSchema`. We will be using `ma`, the instance of the Marshmallow from Flask-Marshmallow, which we created earlier (Step 2) to create our schemas.

Create the `schemas` package and add two files named `Item.py` and `Store.py`.

5.1 Item.py

The Item.py file should contain the following content.

ItemSchema.py

5.2 Store.py

The Store.py file should contain the following content.

StoreSchema.py

We need to create one schema per Model class, which we defined earlier (Step 4). Once we have defined the Schemas, we can now use these schemas to dump and load the ORM objects.

6. Create Resources

Flask-RESTPlus provides many ways to organize our application and maintain them at a granular level, such as Resources, Namespaces and Blueprints. More about this as we go along.

Resources

The main building blocks provided by Flask-RESTPlus are resources. Resources are built on top of Flask pluggable views, giving us easy access to multiple HTTP methods just by defining methods on a resource.

A `resource` is a class whose methods are mapped to an API/URL endpoint. It can have multiple methods, but each one must be named after one of the accepted HTTP verbs. In case, we need more than one `GET` or `POST` methods for our API, then we need to create multiple resource classes and put each method in the corresponding resource class.

Now, let's start creating the Resources for our application. Create the `resources` package, and add two files named `Item.py` and `Store.py`.

6.1 Item.py

The Item.py file should contain the following content.

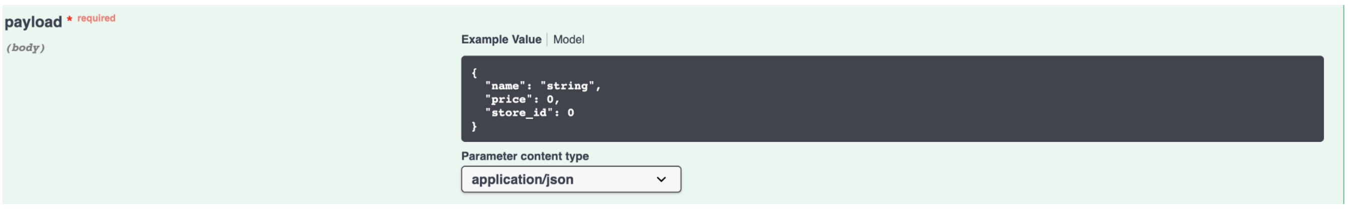
Item-Resource.py

In the `item.py` file, we have defined two resources, namely, `Item` and `ItemList`.

- `Item` is used to perform operations on the `Item` table using the id field.
- `ItemList` is used to create an `Item` or return all the items present in the `Item` table.

API Model for Swagger-UI Support

Currently, Flask-RestPlus does not support ModelSchema or Model, directly, as the request body to be displayed in Swagger-UI, which is a bit of an overhead. Hence, as a workaround we can create API models to be used along with the `expect` annotation to achieve our goal. Flask-RESTPlus has the ability to automatically document and validate the format of incoming JSON objects by using API models. This model is only required for display in Swagger-UI.



```
{
  "name": "string",
  "price": 0,
  "store_id": 0
}
```

request-body

To send or receive information (JSON objects) we can create API models. A RESTPlus API model defines the format of an object by listing all the expected fields. Each field has an associated type, such as, `String`, `Integer`, `DateTime`, which determines what field values will be considered valid.

In this case, we have defined an item model by specifying the information it expects and the properties of each expected value, as follows:

```
item = items_ns.model('Item', {
    'name': fields.String('Name of the Item'),
    'price': fields.Float(0.00),
    'store_id': fields.Integer
})
```

This item model will be used as a body in `Put` method of `Item` Resource and `Post` method of `ItemList` Resource.

By using the `expect` annotation, for every HTTP method we can specify the expected *model* of the payload body to be displayed in Swagger-UI.

```
@item_ns.expect(item)
```

Namespaces

Namespaces are optional, and add a bit of additional organizational touch to the API, mainly, from a documentation point of view. A namespace allows you to group related

Resources under a common root. Create the namespaces as follows:

```
item_ns = Namespace('item', description='Item related operations')
items_ns = Namespace('items', description='Items related
operations')
```

Here, we created a separate Namespace for individual Resource. The `Namespace()` function creates a new namespace with a URL prefix. The `description` field will be used in the Swagger UI to describe this set of methods.

6.2 Store.py

The Store.py file should contain the following content.

Store-Resource.py

In the `store.py` file, we have defined two resources, namely, `Store` and `StoreList`.

- `Store` is used to perform operations on `Store` table using the id field.
- `StoreList` is used to create a `Store` or return all the stores present in the `Store` table.

API Model for Swagger-UI Support

Here too, we have created a store model by specifying the information it expects and the properties of each expected value, as follows:

```
store = stores_ns.model('Store', {
    'name': fields.String('Name of the Store')
})
```

This store model will be used as a body in the `Post` method of `StoreList` Resource.

We will be using the `expect` annotation as follows:

```
@stores_ns.expect(store)
```

Namespaces

Similar to Item Resources, here also we created separate Namespaces.

```
store_ns = Namespace('store', description='Store related
operations')
stores_ns = Namespace('stores', description='Stores related
operations')
```

7. Application Entry Point

Now, let us create our application entry point, which brings us to blueprints.

Blueprints

Flask Blueprints helps us to organize our application by splitting common functionality into distinct components. We can organize our application components based on their function or the division of the app they cater to.

In the root directory of the project, create a file named `app.py` with the following content:

The above code within `app.py` does the following:

7.1 WSGI (Web Server Gateway Interface):

- We started off by creating a Flask WSGI application in `line 10`.

7.2 Flask Blueprints:

- In `line 11`, we created a blueprint instance by passing `name` , `import_name` and `url_prefix`. **Flask blueprint,** helps us to host the API under the `/api` URL prefix. This enables us to separate the API part of our application from other parts. It's a popular way of designing modular applications.
- `API` is the main entry point for the Flask-RESTPlus application resources. Hence, we have initialized it with the `blueprint` in `line 12`. Here, we have overridden the default Swagger UI documentation location by passing an additional parameter `doc=/doc` to the initializer.
- The next step is to register the Blueprint with our Flask app in `line 13`.

7.3 SQLAlchemy

- From `line 14` to `16` we have added some configuration related to SQLAlchemy. Here, we are using SQLite DB. These configurations are required to link SQLite DB with SQLAlchemy. `data.db` is the name of the DB File.

7.4 Namespaces

- From `line 18` to `21` we added all the namespaces defined earlier to the list of namespaces in the `API` instance. If you build the API using the namespaces composition, it becomes simpler to scale it to multiple APIs, when you have to maintain multiple versions of an API.

7.5 Database

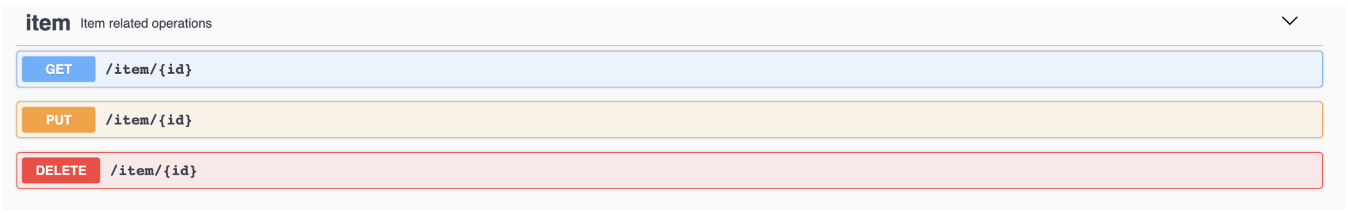
- From `line 24` to `26` we have used the db instance (from file `db.py`) to create the DB file along with the tables before the user accesses the server.

7.6 Exception Handling

- The `@api.errorhandler` decorator allows us to register a specific handler for a given exception (or any exceptions inherited from it), in the same manner that we can do with Flask/Blueprint `@errorhandler` decorator.

7.7 Add Resources to Namespace

- From `line 34` to `37` we called the `add_resource` method on the namespaces to bring the required resources under a given namespace. We should also ensure that the name of the namespace replaces the name of the resource, so endpoints can simply refer to the URL specified as an argument. Here, for example `item_ns.add_resource(Item,'/<int:id>')` Item resource is available at the endpoint `/item/{id}` name of `item_ns` which is `item` followed by `/id`. We can notice the changes in the Swagger UI as follows:



Item-Swagger-UI

7.8 SQLAlchemy and Marshmallow

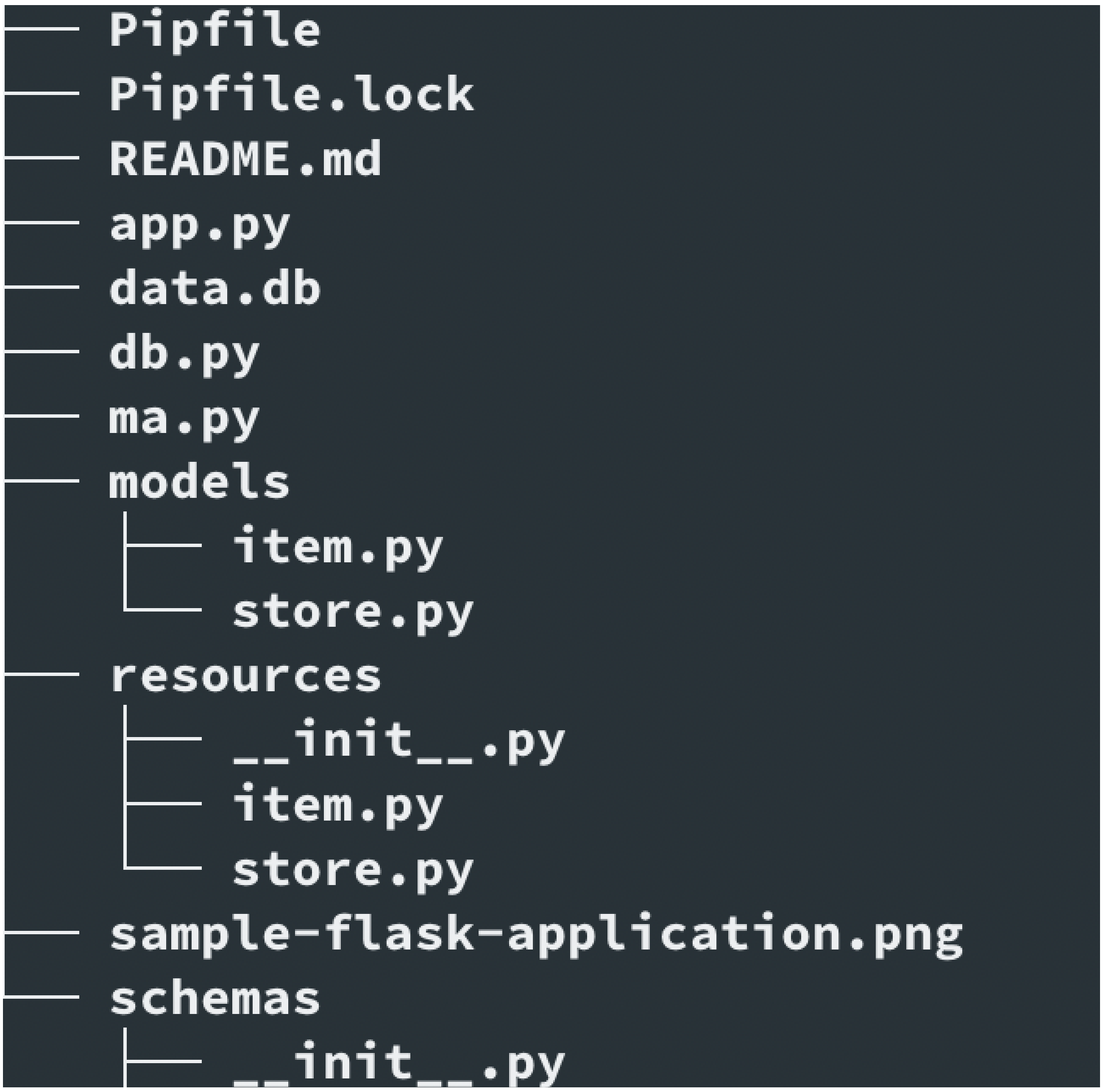
- In `line 40` and `line 41` integrate SQLAlchemy and Marshmallow.

7.9 **Start** the Application

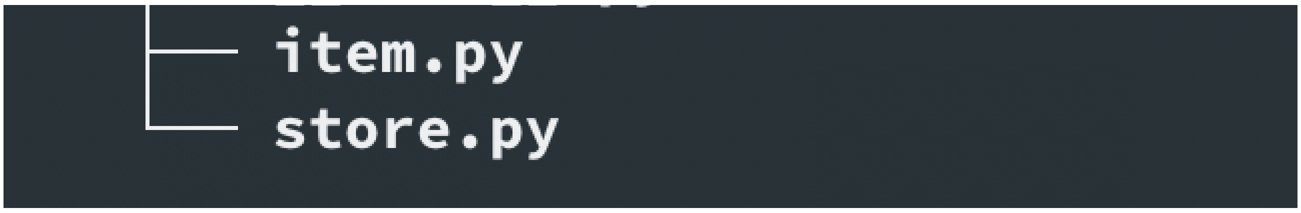
- Finally, in `line 42` we start the application at port=5000 by executing the following command.

```
python app.py
```

We are done with all the coding part and it's testing time. Our project structure now looks as below.



```
├── Pipfile
├── Pipfile.lock
├── README.md
├── app.py
├── data.db
├── db.py
├── ma.py
├── models
│   ├── item.py
│   └── store.py
├── resources
│   ├── __init__.py
│   ├── item.py
│   └── store.py
├── sample-flask-application.png
├── schemas
│   ├── __init__.py
```

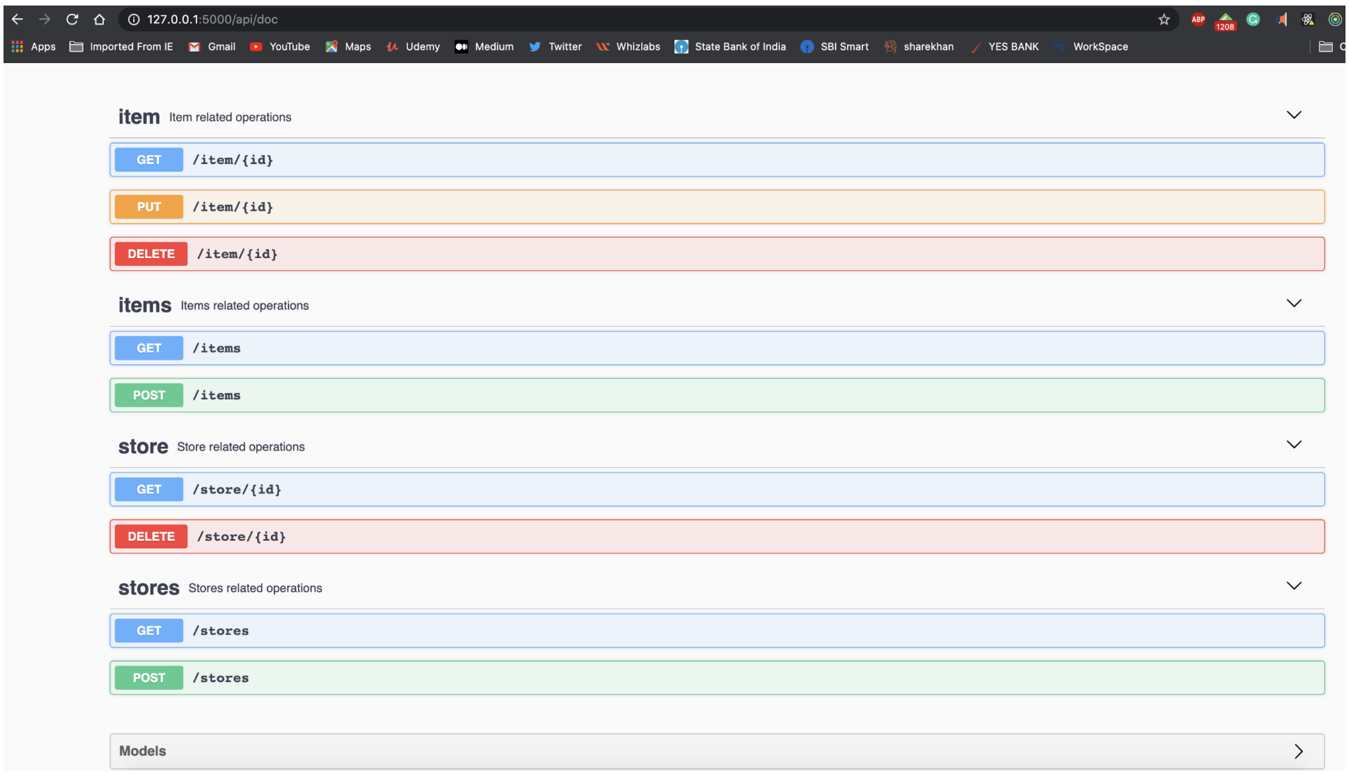


```
├── item.py
└── store.py
```

Project Structure

8. Test the API

We can now test our application to ensure that everything is working fine. We can open the URL `http://127.0.0.1:5000/api/doc` in our browser. We should be able to see the swagger documentation as below.



item Item related operations

- GET `/item/{id}`
- PUT `/item/{id}`
- DELETE `/item/{id}`

items Items related operations

- GET `/items`
- POST `/items`

store Store related operations

- GET `/store/{id}`
- DELETE `/store/{id}`

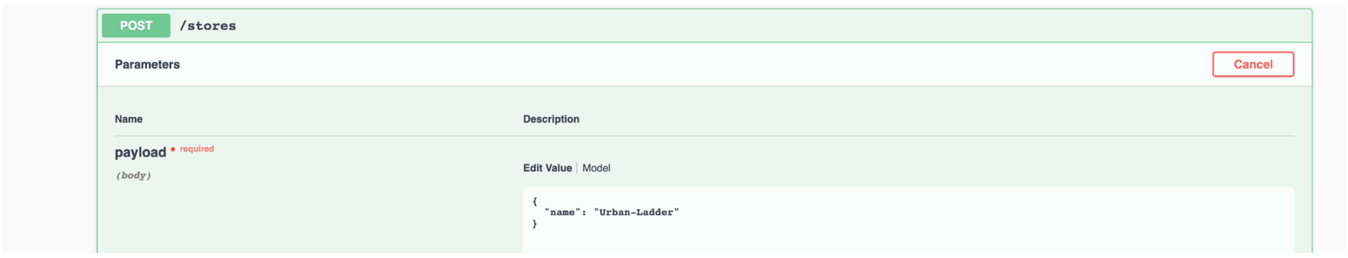
stores Stores related operations

- GET `/stores`
- POST `/stores`

Models

Swagger-UI

Let's test the create Store endpoint using the swagger.

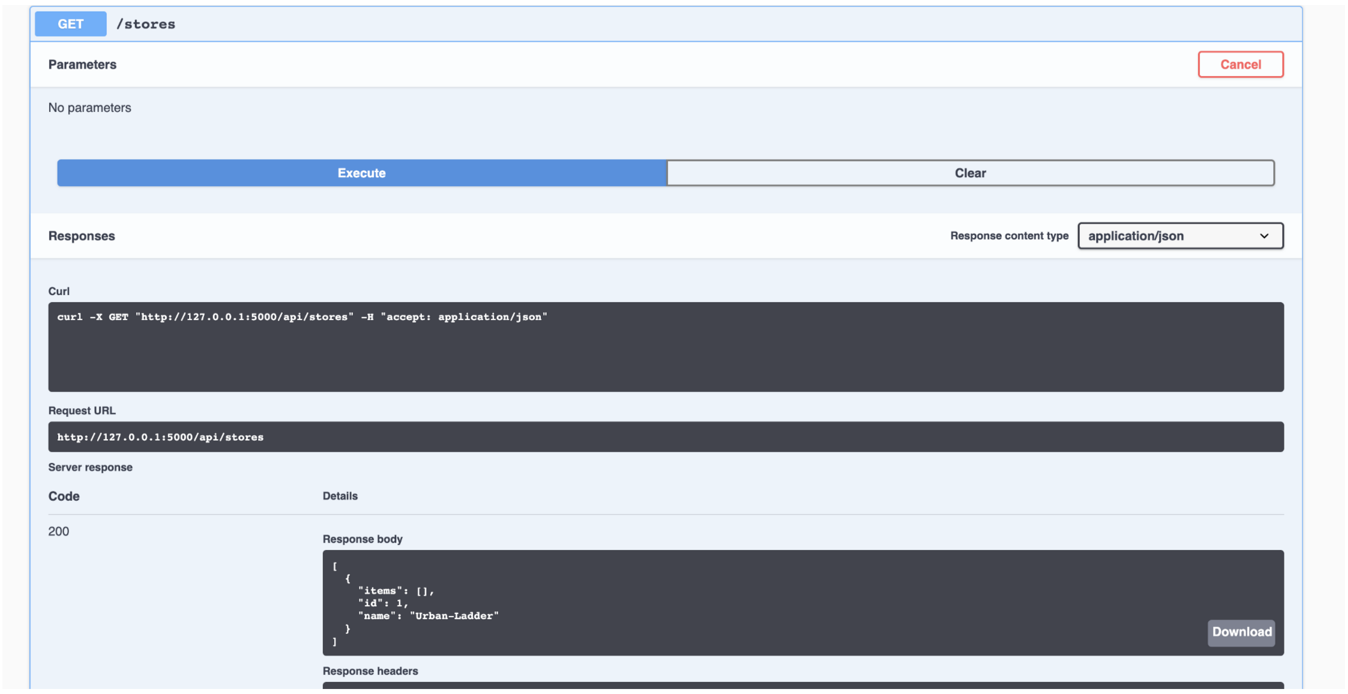


Name	Description
payload * required (body)	

```
{
  "name": "Urban-Ladder"
}
```

Store-Create

Let's fetch all the stores.



No parameters

Response content type: application/json

Curl

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

Request URL

```
http://127.0.0.1:5000/api/stores
```

Server response

Code	Details
200	Response body

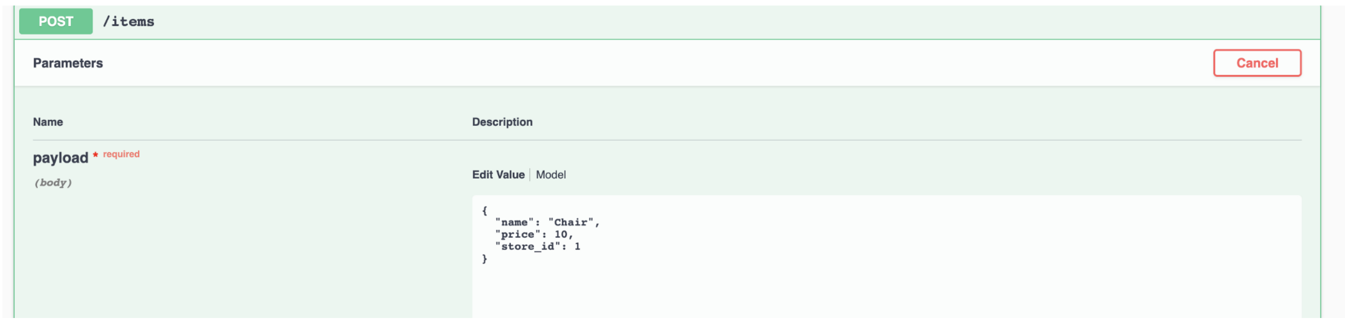
```
[
  {
    "items": [],
    "id": 1,
    "name": "Urban-Ladder"
  }
]
```

Response headers

Fetch-Stores

As we can see in the response that there are no items in the store. So let’s start by adding some items to the stores.

Add an Item to the store.

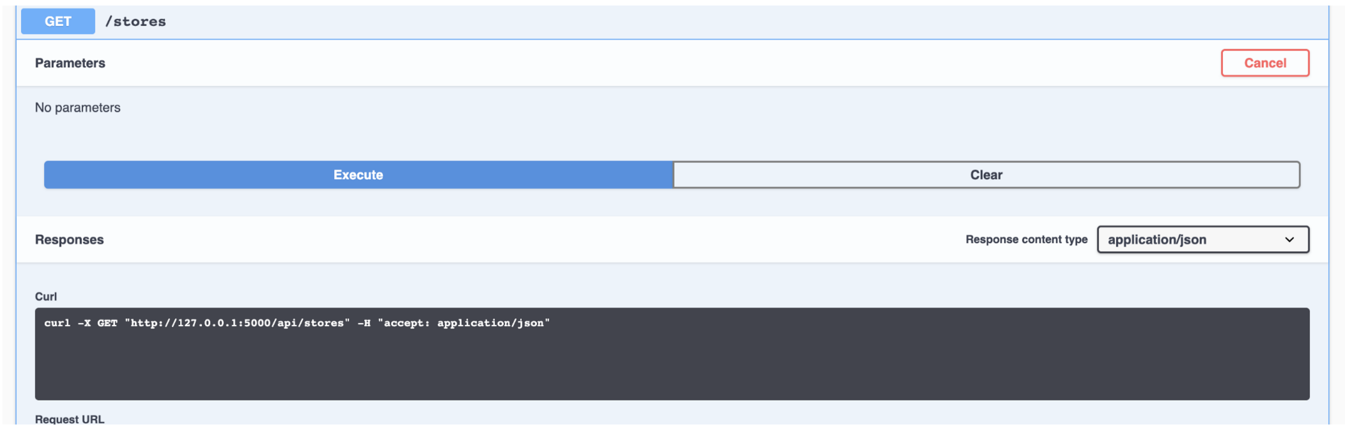


Name	Description
payload * required (body)	

```
{
  "name": "Chair",
  "price": 10,
  "store_id": 1
}
```

Add-Item

Now, if we fetch the stores again, we can see the item that we added in the earlier step in the store.



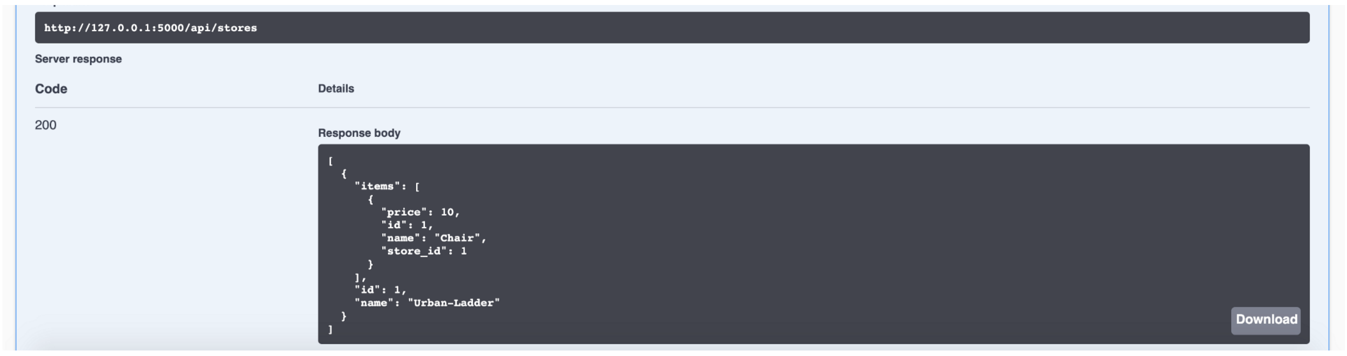
No parameters

Response content type: application/json

Curl

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

Request URL



```
http://127.0.0.1:5000/api/stores
```

Server response

Code	Details
200	Response body

```
[
  {
    "items": [
      {
        "price": 10,
        "id": 1,
        "name": "Chair",
        "store_id": 1
      }
    ],
    "id": 1,
    "name": "Urban-Ladder"
  }
]
```

fetch-stores

Similarly, we can test other operations using Swagger-UI.

If you would like to refer to the full code, do check:

https://github.com/sumanentc/python-sample-flask-application

References & Useful Readings

Flask-RESTPlus

Flask-RESTPlus is an extension for Flask that adds support for quickly building REST APIs. Flask-RESTPlus encourages...

flask-restplus.readthedocs.io

noirbizarre/flask-restplus

You can't perform that action at this time. You signed in with another tab or window. You signed out in another tab or...

github.com

SQLAlchemy 1.4 Documentation

About this document This tutorial covers the well known SQLAlchemy ORM API that has been in use for many years. As of...

docs.sqlalchemy.org

Quickstart - Flask Documentation (1.1.x)

A minimal Flask application looks something like this: So what did that code do? First we imported the class. An...

flask.palletsprojects.com

Flask-SQLAlchemy - Flask-SQLAlchemy Documentation (2.x)

See the SQLAlchemy documentation to learn how to work with the ORM in depth. The following documentation is a brief...

flask-sqlalchemy.palletsprojects.com

Flask-Marshmallow: Flask + marshmallow for beautiful APIs - Flask-Marshmallow 0.14.0 documentation

Flask-Marshmallow is a thin integration layer for Flask (a Python web framework) and marshmallow (an object...

flask-marshmallow.readthedocs.io

Flask Flask Restful Flask Sqlalchemy Python3