Material for Flask Masterclass

Abdur-Rahmaan Janhangeer

Material for Flask Masterclass

Abdur-Rahmaan Janhangeer

This book is for sale at http://leanpub.com/courses/leanpub/flask-masterclass

This version was published on 2023-02-20

Contents

About The Factory Pattern **1**
- Factory Pattern 1
- Implementing the pattern 1
- Why such a pattern? 2

Configuration Profiles **3**
- Inheritance in Python 3
- App structure 3
- Purposeful profiles 4
- Immediate benefits 5
- Switching between profiles 5
- Good to know 6

Secrets Management **7**
- Overview of the instance folder 7
- Setting up the instance folder 7
- Testing the configuration 9
- Setting up the instance folder from scratch 11
- Environment variables 11

Commandline Arguments **14**
- Implement command-line arguments using `app.cli.command` 14
- The Flask shell 15
- Customizing the Flask shell 16
- Implementing manage.py 17

Pattern For Avoiding Application Factory Errors **18**
- File for declarations 18

Section 1 Quiz **21**

flask-sqlalchemy **22**
- Advantages of an ORM 22
- Differences between SQLAlchemy and Flask-SQLAlchemy 22
- Setting up Flask-SQLAlchemy 22

Creating the database . . . 25
SQLAlchemy tables . . . 26
Refs . . . 27

flask-migrate . . . 28
Setting up Flask-Migrate . . . 28
Using Flask-Migrate . . . 30

flask-login . . . 32
Flask-Login requirements . . . 32
Overview of the app . . . 32
Setting up configurations . . . 33
Implementing our models . . . 34
Dealing with authentication . . . 36
Why Bcrypt (or Flask-Bcrypt) is not needed for this demo? . . . 36
What is a blueprint? . . . 36
Writing templates . . . 37
Setting up views . . . 38
Tweaking the app.py . . . 39
Running the app . . . 41

flask-admin . . . 43
Adding an admin view . . . 43
Implementation . . . 44

flask-reuploaded . . . 48
Flask-Uploads concepts . . . 48
Implementation . . . 48
Good to know . . . 50

flask-wtforms . . . 52
Flask-WTForms concepts . . . 52
Demo app . . . 53
Good to know . . . 55

marshmallow-sqlalchemy . . . 56
Why is Marshmallow-Sqlalchemy useful? . . . 56
Implementing Flask-marshmallow . . . 56
Choosing what fields to display . . . 59

wtforms-sqlachemy . . . 60
Our app . . . 60
Implementation . . . 60
Homepage . . . 62
Edit page . . . 64

Dealing with the CSRF Token 65

flask-mailman 66
Implementation 66
Watch emails arrive 67

Section 2 Quiz 71

Pushing The Modular Approach Further 72
Benefits of the modular pattern 72
Caveats of modular patterns 73
Conclusion 73

Enhancing The Modular Approach 74
Registering blueprints on the fly 74
Automating the creation of modules 76
The facultative nature of module elements 76
Conclusion 76

Assets Management 77
Modular assets management 77
Implementing `collectstatic` serving 78
Conclusion 78

More Project Management Options 79
Implementing theming 79
Spawning new projects 80

Section 3 Quiz 81

Introduction To pytest 82
Getting started with PyTest 82
Dealing with multiple files 82

Defining Flask Fixtures 84
Flask fixtures 84

Documenting With Sphinx 86
Set up Sphinx 86
Bonus 87

Logging 88

Section 4 Quiz 89

APIs With Flask 90
Available frameworks for developing API apps 90

flask-restx Fundamentals **91**

Advanced flask-restx Concepts **96**
Custom input validation 96
Defining default response messages for status codes 96
Namespaces 98
Using with the application factory pattern. 100
Explicitly adding models to the documentation 101
Nested and List fields in models 101
Returning a list of items 101

Auth, JWT, Errors and CORS **103**
Documenting authorizations 103
Cors 104
Crafting responses 104
Decouple the business logic from the endpoint 105
The convenience of tracking operation success 105
Pattern for JWT validation 107

Relationship Between Flask And Werkzeug **108**
What is Flask made up of? 108
The problem that Werkzeug sometimes poses 109

How Context Processing Works **110**
Request Context 110
Flask signals concerning request context 110
Proxies 111
Application context 111
Flask's g 111

Deploying To Shared Hosting **112**
What to look for? 112
How to proceed? 112

New In Flask 2.0 . . . . . **114**
Loading any type of configuration file 114
Nested blueprints 114
Named methods 115
Async requests 115

Flask Community Workgroup **116**
About the need for a community workgroup 116
About the Pallets-eco 116

Shopyo: The Framework Behind The Course **117**

Other Useful Extensions **118**
Flask-Limiter 118
Flask-Babel 118
Flask-Multipass 118
Flask-Appbuilder 119
Flask-SocketIO 119

Further Readings **120**

About The Factory Pattern

This course assumes that you have basic knowledge of Flask. Let's get started with some more advanced Flask concepts.

Factory Pattern

A widely used design pattern is the Factory Pattern. An oft-quoted definition taken from Design Patterns: Elements of Reusable Object-Oriented Software. (1994) is:

"Define an interface for creating an object, but let subclasses decide which class to instantiate. The Factory method lets a class defer instantiation it uses to subclasses."

We can clarify it as follows:

- Define an interface for creating an object.
- Have a base class and have other classes inherit from it.
- An object is created when the Factory method is called.

This uses the idea that a factory can create different objects. If you need many cars, don't create cars but create a factory that produces automobiles.

Implementing the pattern

The actual implementations vary but here is one simple example.

```
class Car:
    '''Base class'''
    def print_make(self):
        pass


class Nissan(Car):
    '''Inherited class'''
    def print_make(self):
        print('Nissan')


```

```
class Toyota(Car):
    '''Inherited class'''
    def print_make(self):
        print('Toyota')


class Factory:

    @classmethod
    def create_car(cls, make):
        car = None

        if make == 'nissan':
            car = Nissan()

        if make == 'toyota':
            car = Toyota()

        return car


car1 = Factory.create_car('nissan')
car1.print_make()
```

This is the same pattern that Flask uses for the Application Factory pattern.

```
def create_app(app_name):
    app = Flask(app_name)

    return app
```

Why such a pattern?

Application Factory pattern allows us to configure our apps according to the values passed.

It comes in handy while configuring our app for production, development, and testing. Moreover, it can also be used to test the app with different settings.

We will use this pattern in the coming lessons.

Configuration Profiles

Please look at this section's code[1] to follow along.

Configuration profiles allow us to have different settings for different environments, typically development, testing, and production.

Inheritance in Python

Here's a simple snippet demonstrating how inheritance is achieved in Python

```
1  class Car:
2      wheels = 4
3
4
5  class Nissan(Car):
6      pass
7
8
9  print(Nissan.wheels)
```

That's why we defined classes in Python 2 using the syntax `class Car(object)` since the class `Car` is inherited from the class `object`. This will help understand the usefulness of this pattern in configuration profiles.

App structure

We have two files

- **app.py**: used for app logic
- **config.py**: used for holding configurations

[1] https://github.com/FlaskMasterClass/s1-c2

```python
class Config:
    TESTING = False

class ProductionConfig(Config):
    FAV_FLOWER = 'rose'

class DevelopmentConfig(Config):
    FAV_FLOWER = 'sunflower'

class TestingConfig(Config):
    FAV_FLOWER = 'moonlight petal'
    TESTING = True
```

```python
from flask import Flask
from config import ProductionConfig

def create_app():
    # factory pattern not useful for now
    app = Flask(__name__)

    return app


if __name__ == '__main__':
    app = create_app()
    app.config.from_object(ProductionConfig())
    app.run()
```

Here we are loading the `ProductionConfig` instead of the `DevelopmentConfig`. The contents of the Config class need not to be redefined. In `ProductionConfig`, `DevelopmentConfig`, or `TestingConfig` we only redefine what changes in that particular setting.

Purposeful profiles

Now, let's tweak our **app.py** to show the usefulness of the Factory pattern and the flexibility of having configuration profiles.

```python
from flask import Flask
from config import ProductionConfig
from config import DevelopmentConfig
from config import TestingConfig


profiles = {
    'development': DevelopmentConfig(),
    'production': ProductionConfig(),
    'testing': TestingConfig()
}


def create_app(profile):
    app = Flask(__name__)
    app.config.from_object(profiles[profile])
    return app

app = create_app('production')

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

Immediate benefits

From what we can see with the Application Factory pattern is that

1. We don't import apps everywhere. We import `create_app` and then create an app wherever we want. This reduces a lot of circular import errors.
2. While creating an app, we can determine what settings to load. This allows us to specify the type of app to use.

Switching between profiles

Since it is recommended to have a default config and to load the configuration profile to use from an environment variable, we can set our app running to

```
if __name__ == '__main__':
    flask_env = os.environ.get("FLASK_ENV", default="development")
    app = create_app(flask_env)
    app.run()
```

and doing

```
export FLASK_ENV=development
```

before running our app.

Good to know

- from_object[2] should only be used to hold default values. Values prone to change will be covered in the next lesson.
- With *from_object*, only values written in caps will be considered. Take for example

```
class DevelopmentConfig(Config):
    FAV_FLOWER = 'sunflower'
    fav_cake = 'mellow'
```

In that case, `fav_cake` will not be read and will not be available as a variable to the app.

[2] https://flask.palletsprojects.com/en/2.0.x/api/#flask.Config.from_object

Secrets Management

Please look at this section's code[3] to follow along.

There are some values that we may want to keep secret, be it API keys or passwords. It is recommended, first and foremost, to secure access to the server. After that, we deal with loading secrets. The first intuition to load secret values is to use a package like python-dotenv[4]. However, it's worth considering what Flask already provides and recommends in this regard.

Overview of the instance folder

Flask provides loading values from a folder named instance which **should be included in our .gitignore**. Our app looks like this:

```
.
├── app.py
├── config.py
└── instance
    └── config.py
```

In **instance/config.py** we can add

```
AMAZON_KEY = "superseckey"
```

And the `AMAZON_KEY` variable will be available for us to use in our app.

Setting up the instance folder

Modifying last lesson's **app.py**:

[3] https://github.com/FlaskMasterClass/s1-c3
[4] https://pypi.org/project/python-dotenv/

```
import os
from flask import Flask
from config import ProductionConfig
from config import DevelopmentConfig
from config import TestingConfig


profiles = {
    'dev': DevelopmentConfig(),
    'production': ProductionConfig(),
    'testing': TestingConfig()
}


def create_app(profile):
    app = Flask(__name__, instance_relative_config=True)
    app.config.from_object(profiles[profile])
    app.config.from_pyfile("config.py", silent=True)

    try: # make sure the config folder exists
        os.makedirs(app.instance_path)
    except OSError:
        pass
    return app


flask_env = os.environ.get("FLASK_ENV", default="development")
app = create_app(flask_env)

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

There are two bits that need explaining. The first one is

```
app = Flask(__name__, instance_relative_config=True)
```

and the second one is

```
app.config.from_pyfile("config.py", silent=True)
```

Flask, by default, looks for the **instance** folder in the same path as **app.py**. `instance_relative_config=True` asks Flask to look for configurations inside the **instance** folder.

`app.config.from_pyfile("config.py", silent=True)` asks Flask to load configurations from the file named **config.py**.

If we look closely at our app, though we have two **config.py**,

```
.
├── app.py
├── config.py # 1st
└── instance
    └── config.py # 2nd
```

we are not loading them both. Specifying `app.config.from_object(profiles[profile])` loads a class with values and not **./config.py**. `instance_relative_config` specifies the config file in the instance folder and `app.config.from_pyfile("config.py ..."` asks to look for a **config.py**. So, only the second one is used.

```
.
├── app.py
├── config.py
└── instance
    └── config.py # only this config.py used
```

```
## Skip instance config loading on testing

It is recommended to skip loading instance variables when testing. To this end, we c\
an do:

```python
if profile != "testing":
    # load the instance config, if it exists, when not testing

    app.config.from_pyfile("config.py", silent=True)
```

Testing the configuration

In **instance/config.py** add the following line.

```
INSTANCE_VAR = 'abcd'
```

Then modify **app.py** as given.
```

```
import os
import pathlib
import flask
from flask import Flask
from flask import current_app

from flask.cli import load_dotenv
dotenv_path = os.path.join(os.path.dirname(__file__), '.env')
load_dotenv(dotenv_path)


from config import ProductionConfig
from config import DevelopmentConfig
from config import TestingConfig


profiles = {
    'development': DevelopmentConfig(),
    'production': ProductionConfig(),
    'testing': TestingConfig()
}

def create_app(profile):
    app = Flask(__name__, instance_relative_config=True)
    app.config.from_object(profiles[profile])
    app.config.from_pyfile("config.py", silent=True)

    if profile != "testing":
        # load the instance config, if it exists, when not testing
        app.config.from_pyfile("config.py", silent=True)

    try: # make sure the config folder exists
        os.makedirs(app.instance_path)
    except OSError:
        pass
    return app


if __name__ == '__main__':
    flask_env = os.environ.get("FLASK_ENV", default="development")
    app = create_app(flask_env)
    @app.route('/check')
    def var_check():
```

```
44          return current_app.config['INSTANCE_VAR']
45      app.run(host='0.0.0.0', port=5001, debug=True)
```

If you go to http://127.0.0.1:5000/check you should see `abcd`

Setting up the instance folder from scratch

It is recommended to put the **instance** folder outside the project directory. This eliminates the possibility of committing the folder by mistake. In such a case, the folder is defined as

```
1  app = Flask(__name__, instance_path='/path/to/instance/folder')
```

But this is not the most popular option and the flaskr[5] tutorial as of 2.0 uses relative config. That's why we covered it first.

Environment variables

Flask provides a way to load environment variables. This requires installing python-dotenv[6].

```
1  pip install python-dotenv
```

Then, to have the following structure,

```
1  .
2  ├── app.py
3  ├── config.py
4  ├── .env
5  ├── .flaskenv
6  └── instance
7      └── config.py
```

.flaskenv is used to hold variables starting with `FLASK_` like `FLASK_ENV`. **.env** is for other variables.
Note: Make sure that both files are added to .gitignore and applied.

We can modify the **config.py** as

[5] https://flask.palletsprojects.com/en/2.0.x/tutorial/
[6] https://pypi.org/project/python-dotenv/

```python
import os

class Config:
    TESTING = False
    ENV_VAR = os.environ.get('ENV_VAR', 'default')
    FLASK_ENV = os.environ.get('FLASK_ENV', 'default')
    FLASK_APP = os.environ.get('FLASK_APP', 'default')
    FLASK_CONFIG = os.environ.get('FLASK_CONFIG', 'default')

class ProductionConfig(Config):
    FAV_FLOWER = 'rose'

class DevelopmentConfig(Config):
    FAV_FLOWER = 'sunflower'

class TestingConfig(Config):
    FAV_FLOWER = 'moonlight petal'
    TESTING = True
```

and our **app.py** as:

```python
import os
import pathlib
import flask
from flask import Flask
from flask import current_app

from flask.cli import load_dotenv
dotenv_path = os.path.join(os.path.dirname(__file__), '.env')
load_dotenv(dotenv_path)


from config import ProductionConfig
from config import DevelopmentConfig
from config import TestingConfig

# rest of file same
```

.env

```
ENV_VAR=somevar
```

.flaskenv

```
FLASK_ENV=development
FLASK_APP=app.py
FLASK_CONFIG=development
```

The difference between Flask's `load_dotenv` and python-dotenv's implementation is that the Flask's one loads .**flaskenv** as well.

Commandline Arguments

Please look at this section's code[7] to follow along.

We can always get away with implementing command-line arguments using `sys.argv` but Flask provides a more wholesome and integrated way to get going with command-line arguments.

Implement command-line arguments using `app.cli.command`

Using `app.cli.command` allows you to pass arguments after the flask command. For example you have `flask routes` which is included by default. Here's how we can implement a `flask greet` command. Modify the **app.py** as given.

```
import click
# ...
def create_app(profile):
    app = Flask(__name__, instance_relative_config=True)
    app.config.from_object(profiles[profile])
    app.config.from_pyfile("config.py", silent=True)

    if profile != "testing":
        # load the instance config, if it exists, when not testing
        app.config.from_pyfile("config.py", silent=True)

    try: # make sure the config folder exists
        os.makedirs(app.instance_path)
    except OSError:
        pass

    @app.cli.command("greet", short_help="Greet Flask users")
    def greet():
        click.echo('Greetings Flask users all over the world!')
        return 0

    return app
```

[7] https://github.com/FlaskMasterClass/s1-c4

> **Note**: click[8] comes installed with Flask.

Now run `flask greet`. It would return

```
Greetings Flask users all over the world!
```

This is particularly practical for seeding the database or doing some other useful operations.

> **Note**: We are using `click.echo` instead of `print` to ensure that we manage our output configurations through click.

Normal development follows *click* patterns. Adding flags for example,

```
@app.cli.command("greet", short_help="Greet Flask users")
@click.option("--verbose/--no-verbose", default=False)
def greet():
```

The Flask shell

Inside the virtual environment, if we type `python`, we will get something similar to the following

```
Python 3.9.5 (default, May 19 2021, 11:32:47)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

However, if we want to modify models or something that needs to be aware of the request context, we'd have to do something like the following each time

```
...
>>> from app import app
>>> from somewhere import db
>>> from models import Model
>>> with app.app_context():
....     model = Model()
....     db.session.add(model)
....     db.session.commit()
```

but, tying `flask shell` gives something similar to the following

[8] https://click.palletsprojects.com/en/8.0.x/

```
Python 3.9.5 (default, May 19 2021, 11:32:47)
[GCC 9.3.0] on linux
App: app [production]
Instance: /home/appinv/play/flaskcourse/s1/c4/instance
>>>
```

which saves us querying the context each time.

```
...
>>> from somewhere import db
>>> from models import Model
>>> model = Model()
>>> db.session.add(model)
>>> db.session.commit()
```

Customizing the Flask shell

The example above is tedious as we have to import our classes and objects. We can avoid this by passing in our imports to load in the following way

```
def create_app(profile):
    app = Flask(__name__, instance_relative_config=True)
    # ...

    @app.shell_context_processor
    def shell():
        return {
            "x": 1,
            "y": 2,
            "z": 3
        }

    return app
```

so that when we type `flask shell` we can do the following.

```
Python 3.9.5 (default, May 19 2021, 11:32:47)
[GCC 9.3.0] on linux
App: app [production]
Instance: /home/appinv/play/flaskcourse/s1/c4/instance
>>> x
1
>>> y
2
>>> z
3
```

Instead of passing `x`, `y` or `z`, we can pass in a db or a model. Hence, we won't need to import them.

Implementing manage.py

Let's say we want to implement `python manage.py runserver` for our Flask app. The way to do it is just to use *click* to pass the command and then either pass `raw flask run` as subprocess command or call `app.run()`, something similar to the following.

```
# manage.py

from app import app
import click


@click.Group
def cli():
    pass

@cli.command('runserver')
def runserver():
    app.run()

if __name__ == '__main__':
    cli()
```

Pattern For Avoiding Application Factory Errors

Please look at this section's code[9] to follow along.

A really annoying error that comes when dealing with Flask is the circular import error. Implementing the Factory pattern means tweaking a few things in your app

File for declarations

A great way of avoiding the circular error is to define extension objects in a separate file so that there are no imports from **app.py** anywhere in the project. We can name it **init.py**.

Here's the structure.

```
.
├── app.py
├── config.py
├── .env
├── .flaskenv
├── init.py
└── instance
    └── config.py
```

In case, we want to define `flask_cors` in the **init.py** file,

```
# init.py

from flask_cors import CORS


cors = CORS()
```

then in **app.py**

[9] https://github.com/FlaskMasterClass/s1-c5

```
import os
import pathlib
import flask
from flask import Flask
import click


from init import cors # new line

from flask.cli import load_dotenv
dotenv_path = os.path.join(os.path.dirname(__file__), '.env')
load_dotenv(dotenv_path)


from config import ProductionConfig
from config import DevelopmentConfig
from config import TestingConfig


profiles = {
    'development': DevelopmentConfig(),
    'production': ProductionConfig(),
    'testing': TestingConfig()
}

def create_app(profile):
    app = Flask(__name__, instance_relative_config=True)
    app.config.from_object(profiles[profile])
    app.config.from_pyfile("config.py", silent=True)

    cors.init_app(app) # new line

    if profile != "testing":
        app.config.from_pyfile("config.py", silent=True)

    try:
        os.makedirs(app.instance_path)
    except OSError:
        pass

    @app.cli.command("greet", short_help="Greet Flask users")
    def greet():
        click.echo('Greetings Flask users all over the world!')
```

```
        return 0

    @app.shell_context_processor
    def shell():
        return {
            "x": 1,
            "y": 2,
            "z": 3
        }

    return app

flask_env = os.environ.get("FLASK_ENV", default="development")
app = create_app(flask_env)

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

then in **app.py**, `cors.init_app(app)` is used as opposed to the usual `cors = Cors(app)` as it does not play along well if you need to use the variable cors in a completely new folder.

```
.
├── app.py
├── config.py
├── .env
├── .flaskenv
├── init.py
├── instance
│   └── config.py
└── new_folder
    └── file.py
```

In case we want to use cors in **file.py** shown above, we can do so by writing `from init import cors`.

Section 1 Quiz

Take this quiz online[10]

Take this quiz online[11]

Take this quiz online[12]

Take this quiz online[13]

Take this quiz online[14]

[10] http://leanpub.com/courses/leanpub/flask-masterclass/quizzes/quiz-1
[11] http://leanpub.com/courses/leanpub/flask-masterclass/quizzes/quiz-2
[12] http://leanpub.com/courses/leanpub/flask-masterclass/quizzes/quiz-3
[13] http://leanpub.com/courses/leanpub/flask-masterclass/quizzes/quiz-4
[14] http://leanpub.com/courses/leanpub/flask-masterclass/quizzes/quiz-5

flask-sqlalchemy

Please look at this section's code[15] to follow along.

SQLAlchemy is a fascinating library. It's one of the most robust and well-executed ORMs across all languages. An ORM allows us to deal with and create SQL tables using regular codes. We can query and delete rows using only pure code without touching SQL.

Advantages of an ORM

One valid reason for not using an ORM is speed, especially when we know that our query will be better than the one auto-generated by the library. Otherwise, using an ORM like SQLAlchemy will help us to:

- easily set up database
- set up migrations
- avoid SQL injections
- keep your code readable, easy to understand
- switch DB if need be (provided we do not use any db-specific syntax)
- easily set up tests
- easily build queries

Differences between SQLAlchemy and Flask-SQLAlchemy

Flask-SQLAlchemy makes it easier to integrate with Flask by abstracting SQLAlchemy details and provides methods like `.paginate` or `_or_404`. SQLAlchemy also recommends using Flask-SQLAlchemy when needed. Read more[16].

Setting up Flask-SQLAlchemy

First, install Flask-SQLAlchemy (make sure you are in a virtual environment).

[15] https://github.com/FlaskMasterClass/s2-c1
[16] https://docs.sqlalchemy.org/en/14/orm/contextual.html

```
pip install Flask-SQLAlchemy==2.5.1
```

Structure

```
.
├── app.py
├── config.py
├── .env
├── .flaskenv
├── init.py
└── instance
    └── config.py
```

In **init.py** add

```
from flask_sqlalchemy import SQLAlchemy


db = SQLAlchemy()
```

In **instance/config.py** add

```
SQLALCHEMY_TRACK_MODIFICATIONS = False
SQLALCHEMY_DATABASE_URI = 'sqlite:///app.db'
```

We are using SQLite[17] as db. `SQLALCHEMY_DATABASE_URI` values come in different flavors depending on the type of db we are using. SQLite allows us to have our db in a single file. This is great for demo apps as well as in situations where we don't want to worry about setting up a database. However, for high-traffic sites, it does not play well. Moreover, if we create and test our demo app in SQLite and want to change to MySQL, there is no guarantee that our code will still work. This is because SQLite is very permissive and will not strictly check what we wrote. When switching to another db, we might find that what we wrote is, in fact, terrible SQL logic. For information on MySQL, Postgres, etc, refer to this link here[18].

`SQLALCHEMY_TRACK_MODIFICATIONS` is a Flask configuration that needs to be defined, otherwise, we will get the following warning.

[17] https://www.sqlite.org/index.html
[18] https://docs.sqlalchemy.org/en/14/core/engines.html#database-urls

```
FSADeprecationWarning: SQLALCHEMY_TRACK_MODIFICATIONS adds significant overhead and \
will be disabled by default in the future.  Set it to True or False to suppress this\
 warning.
```

Note: Refer to this link[19] to learn more about it.

app.py

```python
import os
import pathlib
import flask
from flask import Flask
from flask import current_app
import click


from init import db # new line

from flask.cli import load_dotenv
dotenv_path = os.path.join(os.path.dirname(__file__), '.env')
load_dotenv(dotenv_path)


from config import ProductionConfig
from config import DevelopmentConfig
from config import TestingConfig


profiles = {
    'development': DevelopmentConfig(),
    'production': ProductionConfig(),
    'testing': TestingConfig()
}

def create_app(profile):
    app = Flask(__name__, instance_relative_config=True)
    app.config.from_object(profiles[profile])
    app.config.from_pyfile("config.py", silent=True)


    db.init_app(app) # new line
```

[19] https://flask-sqlalchemy.palletsprojects.com/en/2.x/signals/

```
34
35      if profile != "testing":
36          app.config.from_pyfile("config.py", silent=True)
37
38      try:
39          os.makedirs(app.instance_path)
40      except OSError:
41          pass
42
43      @app.shell_context_processor
44      def shell():
45          return {
46              "db": db, # new line
47          }
48
49      return app
50
51
52  flask_env = os.environ.get("FLASK_ENV", default="development")
53  app = create_app(flask_env)
54
55  if __name__ == '__main__':
56      app.run()
```

If we were not using the Application Factory pattern, it would be like the following.

```
1  db = SQLAlchemy(app)
```

However, since we are using the Factory pattern, we will use the `init_app` method.

```
1  db.init_app(app)
```

Creating the database

Using bare-bones Python in the shell, we will type `python` (ensure that you are in a virtual environment).

```
Python 3.9.5 (default, May 19 2021, 11:32:47)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

then

```
>>> from init import db
>>> from app import app
>>> with app.app_context():
...     db.create_all()
...
```

Here `app.app_context()` is used to use SQLAlchemy within the context of a Flask app.

We can also do

```
flask shell
```

which will fire up

```
Python 3.9.5 (default, May 19 2021, 11:32:47)
[GCC 9.3.0] on linux
App: app [production]
Instance: /home/appinv/play/flaskcourse/s2/c1/instance
>>>
```

then, just type

```
>>> db.create_all()
```

Both methods would work and you should see an **app.db** file in your directory.

> **Note**: The first approach is recommended as the second one can silently pass on errors.

SQLAlchemy tables

The simplest demo of a table we can get is by adding the following before `create_app`.

```
class User(db.Model):
    __tablename__ = "users"
    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
```

This tells SQLAlchemy to create a table named *users* having a column named *id*, which will automatically increment as new records are added.

Re-run `db.create_all` and use a db viewer mentioned below to see the change in database.

> **Note**: To view the database on Windows, use HeidiSQL[20]. Use DBeaver[21] (also cross-platform) for Linux or sqlitebrowser[22] for only SQLite use.

Refs

[1]

> The scoped_session object is a very popular and helpful object used by many SQLAlchemy applications. However, it is important to note that it presents only one approach to the issue of Session management. If you're new to SQLAlchemy, and especially if the term "thread-local variable" seems strange to you, we recommend that, if possible, yourself first familiarize with an off-the-shelf integration system such as Flask-SQLAlchemy or zope.sqlalchemy. (source[23])

[20] https://www.heidisql.com/
[21] https://dbeaver.io/
[22] https://sqlitebrowser.org/
[23] https://docs.sqlalchemy.org/en/14/orm/contextual.html

flask-migrate

Please look at this section's code[24] to follow along.

Database or schema migrations, also simply known as migrations, allow us to roll back to a previous version of our database schema. To enable a smooth experience, we in part, rely on the robustness of the tool used. Alembic[25] is an excellent migration tool used in conjunction with SQLAlchemy. It is robust and maintained under the SQLAlchemy organization. Flask-Migrate[26] provides us Alembic configured for usage with Flask.

Setting up Flask-Migrate

Structure

```
.
├── app.py
├── config.py
├── init.py
└── instance
    └── config.py
```

Install Flask-Migrate

```
pip install Flask-Migrate==2.5.3
```

init.py

```
from flask_sqlalchemy import SQLAlchemy
from flask_migrate import Migrate # new line

db = SQLAlchemy()
migrate = Migrate() # new line
```

app.py

[24] https://github.com/FlaskMasterClass/s1-c2
[25] https://github.com/sqlalchemy/alembic
[26] https://flask-migrate.readthedocs.io/en/latest/

```
import os
import pathlib
import flask
from flask import Flask
from flask import current_app
import click


from init import db
from init import migrate # new line

from flask.cli import load_dotenv
dotenv_path = os.path.join(os.path.dirname(__file__), '.env')
load_dotenv(dotenv_path)


from config import ProductionConfig
from config import DevelopmentConfig
from config import TestingConfig


profiles = {
    'development': DevelopmentConfig(),
    'production': ProductionConfig(),
    'testing': TestingConfig()
}


class User(db.Model):
    __tablename__ = "users"
    id = db.Column(db.Integer, primary_key=True, autoincrement=True)

def create_app(profile):
    app = Flask(__name__, instance_relative_config=True)
    app.config.from_object(profiles[profile])
    app.config.from_pyfile("config.py", silent=True)


    db.init_app(app)
    migrate.init_app(app, db) # new line

    if profile != "testing":
        app.config.from_pyfile("config.py", silent=True)
```

```
44
45      try:
46          os.makedirs(app.instance_path)
47      except OSError:
48          pass
49
50      @app.shell_context_processor
51      def shell():
52          return {
53              "db": db,
54          }
55
56      return app
57
58
59  flask_env = os.environ.get("FLASK_ENV", default="development")
60  app = create_app(flask_env)
61
62  if __name__ == '__main__':
63      app.run()
```

Just calling `migrate.init_app` with proper parameters will enable us to use Flask-Migrate which we'll cover next.

Using Flask-Migrate

Delete **app.db** (if it exists). We will create our tables from `Flask-Migrate.Run` next.

```
1  flask db init
```

After running the above line, you will see a folder called `migrations`. Usually, we can edit the Alembic files in there. For applying the actual change, Flask-Migrate has some commands

```
1  flask db migrate -m "Initial migration."
```

followed by

```
1  flask db upgrade
```

We can change *Initial migration* each time with an appropriate message. The `flask migrate` and `flask upgrade` commands are needed each time we want to apply our migration to the db.

> **Note:** In the folder called migrations, we can edit the Alembic files. This is because there are times when we need to write what's changed. Alembic may not detect some changes automatically. So for changing table names or for some other use-cases[27], knowing Alembic well will come in handy. We may not need it in most cases but we must know when to do what. Reading the Alembic docs is the best way to know how Flask-Migrate fits in.
>
> **Note:** The app above will not work as there is no route and the whole purpose is to demo the Flask-Migrate integration structure.

Flask-Migrate does not impact the app at runtime. Flask-Migrate is useful to easily run migration commands before we run our app.

[27] https://alembic.sqlalchemy.org/en/latest/autogenerate.html#what-does-autogenerate-detect-and-what-does-it-not-detect

flask-login

Please look at this section's code[28] to follow along.

If you are using Flask's template system to build your website, Flask-Login[29] is a great framework to implement authentication. It has an excellent maintenance record and integrates nicely with other plugins like Flask-Admin. The only downside is that we'd need to code the logic and roles from the scratch. This, however, gives the possibility to go as far and as much as we need. It provides the basic elements we'd need straight away, including the *remember me* feature.

Flask-Login requirements

To get Flask-Login working, we need to:

- set `SECRET_KEY` config
- specify how to find the user
- specify login view
- define anonymous class

Overview of the app

Well, here is the structure of the app we are going to implement.

```
.
├── app.db
├── app.py
├── auth_views.py
├── secret_views.py
├── config.py
├── .env
├── .flaskenv
├── init.py
├── instance
│   └── config.py
├── models.py
└── templates
    ├── home.html
    └── secret.html
```

[28] https://github.com/FlaskMasterClass/s2-c3
[29] https://flask-login.readthedocs.io/en/latest/

Routes

Route	Purpose
/	homepage with login form
/auth/validate-login	check if login ok and redirect to secret page
/auth/logout	logout route
/secret/	the page that cannot be accessed except by login

We'll add `/auth/validate-login` and `/auth/logout` under a blueprint called *auth* and `/secret/` also under a blueprint.

Setting up configurations

The only thing that differs from the previous lessons is that we are now setting the `SECRET_KEY`.

config `.env`

```
SECRET_KEY = pass1234 # should be as random as possible
```

This allows us to use sessions which Flask-Login uses.

.flaskenv

```
FLASK_APP=app.py
FLASK_ENV=development
```

config.py

```
import os


class Config:
    TESTING = False
    ENV_VAR = os.environ.get("ENV_VAR")
    FLASK_ENV = os.environ.get("FLASK_ENV")
    SECRET_KEY = os.environ.get("SECRET_KEY") # new line


class ProductionConfig(Config):
    pass


```

```
15 class DevelopmentConfig(Config):
16     pass
17
18
19 class TestingConfig(Config):
20     TESTING = True
```

instance/config.py

```
1 INSTANCE_VAR = "abcd"
2 SQLALCHEMY_TRACK_MODIFICATIONS = False
3 SQLALCHEMY_DATABASE_URI = "sqlite:///app.db"
```

init.py

```
1 from flask_login import LoginManager # new
2 from flask_migrate import Migrate
3 from flask_sqlalchemy import SQLAlchemy
4
5 db = SQLAlchemy()
6 migrate = Migrate()
7 login_manager = LoginManager() # new
8 login_manager.login_view = "home" # new
```

Here we are creating a new login manager and defining the login page view so that Flask-Login can redirect as appropriate. *home* is the name of a view function we defined in our **app.py**

Implementing our models

models.py

```
1 from flask_login import AnonymousUserMixin
2 from flask_login import UserMixin
3 from sqlalchemy.ext.hybrid import hybrid_property
4 from werkzeug.security import check_password_hash
5 from werkzeug.security import generate_password_hash
6
7 from init import db
8 from init import login_manager
9
10
```

```
class AnonymousUser(AnonymousUserMixin):
    """Anonymous user class"""

    def __init__(self):
        self.username = "guest"
        self.email = "<anonymous-user-no-email>"

    def __repr__(self):
        return f"<AnonymousUser {self.username}>"


login_manager.anonymous_user = AnonymousUser

class User(UserMixin, db.Model):
    __tablename__ = "users"
    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    email = db.Column(db.String(120), unique=True, nullable=False)
    _password = db.Column(db.String(128), nullable=False)

    @hybrid_property
    def password(self):
        return self._password

    @password.setter
    def password(self, plaintext):
        # the default hashing method is pbkdf2:sha256
        self._password = generate_password_hash(plaintext)

    def check_password(self, password):
        return check_password_hash(self._password, password)


@login_manager.user_loader
def load_user(user_id):
    return User.query.get(user_id)
```

Flask-Login provides some mixins so that we can inherit from them. A mixin allows us to inject values and methods into our class. Flask-Login needs an anonymous class and a user class. `login_manager.anonymous_user = AnonymousUser` specifies the anonymous class and `@login_manager.user_loader` tells Flask-Login how to query the user.

Dealing with authentication

First and foremost, we do not store passwords in plain text in databases. This ensures that even if someone hacks into our database, they'll still have to figure out our encryption mechanism to know our passwords.

Here we set passwords to be a hashed string. Creating a user is done via `User(email='...', password='...')`. When saving to the db the actual column name is `_password`. hybrid_property[30] allows us to have password as a class attribute and `@password.setter` handles what exactly happens when setting `password=`. Then `user.check_password('...')` returns true or false whether or not the value we passed generates the same hash as we have in the db.

Why Bcrypt (or Flask-Bcrypt) is not needed for this demo?

We used `generate_password_hash` from *Werkzeug* which comes pre-installed with Flask. It uses the hashing algorithm and is comparable to Bcrypt. To step up our pbkdf2 game with Werkzeug, increase the number of iterations of the `generate_password_hash` function using `generate_password_hash('...', method='pbkdf2:sha256:80000')`. The Django[31] docs have a nice summary of why they choose the pbkdf2 as default and how we can tighten the security.

> **Note:** We will skip the discussion on salting in this lesson but it should be undertaken by the user. Please note that the function used above already generates salted passwords.

What is a blueprint?

A blueprint allows us to namespace our URLs. E.g., if we have `/fruit/apple` and `/fruit/pear/`, we can group them under a common namespace.

[30] https://docs.sqlalchemy.org/en/14/orm/extensions/hybrid.html
[31] https://docs.djangoproject.com/en/3.2/topics/auth/passwords/

```python
fruit_blueprint = Blueprint(
    "fruit",
    __name__,
    url_prefix="/fruit",
    template_folder="templates",
)


@fruit_blueprint.route("/apple")
def apple():
    ...


@fruit_blueprint.route("/pear")
def pear():
    ...

# in app.py
app.register_blueprint(fruit_blueprint)
```

Writing templates

templates/home.html

```html
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title></title>
</head>
<body>
    Please log in:
    <form action="{{ url_for('auth.validate_login') }}" method="POST">
        Email<br>
        <input type="email" name="email"><br>
        Password<br>
        <input type="password" name="password"><br>
        <input type="submit" name="" value="submit">
    </form>
</body>
</html>
```

templates/secret.html

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title></title>
</head>
<body>
    This is a secret page<br>
    <a href="{{ url_for('auth.logout') }}">log out</a>
</body>
</html>
```

Setting up views

auth_views.py

```
from flask import Blueprint
from flask import redirect
from flask import url_for
from flask import request
from flask_login import current_user
from flask_login import login_required
from flask_login import login_user
from flask_login import logout_user

from models import User

auth_blueprint = Blueprint(
    "auth",
    __name__,
    url_prefix="/auth",
    template_folder="templates",
)


@auth_blueprint.route("/validate-login", methods=["POST"])
def validate_login():
    email = request.form["email"]
    password = request.form["password"]

    user = User.query.filter(User.email == email).first()

```

```
    if user is None:
        return redirect("home")
    else:
        if user.check_password(password):
            login_user(user)
            return redirect(url_for("secret.message"))
        else:
            return redirect(url_for("home"))


@auth_blueprint.route("/logout", methods=["GET"])
def logout():
    logout_user()
    return redirect(url_for("home"))
```

secret_views.py

```
from flask import Blueprint
from flask import render_template
from flask_login import login_required

secret_blueprint = Blueprint(
    "secret",
    __name__,
    url_prefix="/secret",
    template_folder="templates",
)


@secret_blueprint.route("/")
@login_required
def message(methods=["GET"]):
    return render_template("secret.html")
```

`login_user` and `logout_user` define what happens when accessing a route protected with `@login_required`.

Tweaking the app.py

app.py

```
import os

import click
import flask
from flask import Flask
from flask import current_app
from flask import redirect
from flask import render_template
from flask.cli import load_dotenv

from auth_views import auth_blueprint
from secret_views import secret_blueprint
from init import db
from init import login_manager # new
from init import migrate
from models import User


dotenv_path = os.path.join(os.path.dirname(__file__), '.env')
load_dotenv(dotenv_path)


from config import DevelopmentConfig
from config import ProductionConfig
from config import TestingConfig

profiles = {
    'development': DevelopmentConfig(),
    'production': ProductionConfig(),
    'testing': TestingConfig()
}

def create_app(profile):
    app = Flask(__name__, instance_relative_config=True)
    app.config.from_object(profiles[profile])
    app.config.from_pyfile("config.py", silent=True)


    app.register_blueprint(auth_blueprint) # new
    app.register_blueprint(secret_blueprint) # new


    db.init_app(app)
```

```
    migrate.init_app(app, db)
    login_manager.init_app(app) # new

    if profile != "testing":
        app.config.from_pyfile("config.py", silent=True)

    try:
        os.makedirs(app.instance_path)
    except OSError:
        pass

    @app.shell_context_processor
    def shell():
        return {
            "db": db,
            "User": User
        }


    @app.route('/') # new
    def home():
        return render_template('home.html')

    return app


flask_env = os.environ.get("FLASK_ENV", default="development")
app = create_app(flask_env)

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

Running the app

Run the following command (make sure that you are in a virtual environment).

```
pip install Flask-Login==0.5.0
```

Create the tables and user by using the following command

```
flask shell
```

followed by

```
Python 3.9.5 (default, May 19 2021, 11:32:47)
[GCC 9.3.0] on linux
App: app [development]
Instance: /home/appinv/code/flaskcourse/s2/c3/instance
>>> db.create_all()
>>> admin = User(email='admin@domain.com', password='pass')
>>> db.session.add(admin)
>>> db.session.commit()
>>> exit()
```

lastly

```
python app.py
```

Login using email = ‘admin@domain.com’ and password = ‘pass’.

Here’s a terminal widget if you want to run the above commands

flask-admin

Please look at this section's code[32] to follow along.

Flask-Admin[33] can be used to have an admin view of the Flask app that would allow quick addition and modification of items.

Adding an admin view

The demo in this section builds upon the Flask-Login demo. It features a simple interface for CRUD operations protected by Flask-Login's auth. After adding the admin view, running `flask routes` would give the following output.

```
Endpoint             Methods    Rule
-------------------  ---------  --------------------------------
admin.index          GET        /admin/
admin.indexs         GET        /admin/dashboard
admin.static         GET        /admin/static/<path:filename>
auth.logout          GET        /auth/logout
auth.validate_login  POST       /auth/validate-login
home                 GET        /
secret.message       GET        /secret/
static               GET        /static/<path:filename>
user.action_view     POST       /admin/user/action/
user.ajax_lookup     GET        /admin/user/ajax/lookup/
user.ajax_update     POST       /admin/user/ajax/update/
user.create_view     GET, POST  /admin/user/new/
user.delete_view     POST       /admin/user/delete/
user.details_view    GET        /admin/user/details/
user.edit_view       GET, POST  /admin/user/edit/
user.export          GET        /admin/user/export/<export_type>/
user.index_view      GET        /admin/user/
```

By default, we see that Flask-Admin registers the admin routes and also the user routes where we asked to add CRUD operations for users (in the implementation below). This is the default working of Flask-Admin.

[32] https://github.com/FlaskMasterClass/s2-c4
[33] https://flask-admin.readthedocs.io/en/latest/

Implementation

```
import os

import click
import flask
from flask import Flask
from flask import current_app
from flask import redirect
from flask import render_template
from flask import url_for
from flask import request
from flask.cli import load_dotenv
from flask_login import AnonymousUserMixin
from flask_login import UserMixin
from flask_login import current_user

from flask_admin import Admin # new
from flask_admin.contrib import sqla as flask_admin_sqla # new
from flask_admin import AdminIndexView # new
from flask_admin import expose # new
from flask_admin.menu import MenuLink # new

from auth_views import auth_blueprint
from secret_views import secret_blueprint
from init import db
from init import login_manager
from init import migrate
from models import User


dotenv_path = os.path.join(os.path.dirname(__file__), '.env')
load_dotenv(dotenv_path)


from config import DevelopmentConfig
from config import ProductionConfig
from config import TestingConfig

profiles = {
    'development': DevelopmentConfig(),
    'production': ProductionConfig(),
```

```
    'testing': TestingConfig()
}

#
# Flask admin setup
#


class DefaultModelView(flask_admin_sqla.ModelView):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

    def is_accessible(self):
        return current_user.is_authenticated

    def inaccessible_callback(self, name, **kwargs):
        # redirect to login page if user doesn't have access
        return redirect(url_for("home", next=request.url))


class MyAdminIndexView(AdminIndexView):
    def is_accessible(self):
        return current_user.is_authenticated

    def inaccessible_callback(self, name, **kwargs):
        # redirect to login page if user doesn't have access
        return redirect(url_for("home", next=request.url))

    @expose("/")
    def index(self):
        if not current_user.is_authenticated:
            return redirect(url_for("home"))
        return super(MyAdminIndexView, self).index()

    @expose("/dashboard")
    def indexs(self):
        if not current_user.is_authenticated:
            return redirect(url_for("home"))
        return super(MyAdminIndexView, self).index()

def create_app(profile):
    app = Flask(__name__, instance_relative_config=True)
    app.config.from_object(profiles[profile])
```

```
    app.config.from_pyfile("config.py", silent=True)


    app.register_blueprint(auth_blueprint)
    app.register_blueprint(secret_blueprint)


    admin = Admin(
        app,
        name="My App",
        template_mode="bootstrap4",
        index_view=MyAdminIndexView(),
    ) # new
    admin.add_view(DefaultModelView(User, db.session)) # new
    admin.add_link(
        MenuLink(name="Logout", category="", url="/auth/logout")
    ) # new

    db.init_app(app)
    migrate.init_app(app, db)
    login_manager.init_app(app)

    if profile != "testing":
        app.config.from_pyfile("config.py", silent=True)

    try:
        os.makedirs(app.instance_path)
    except OSError:
        pass

    @app.shell_context_processor
    def shell():
        return {
            "db": db,
            "User": User
        }


    @app.route('/')
    def home():
        return render_template('home.html')

    return app
```

```
127
128
129 flask_env = os.environ.get("FLASK_ENV", default="development")
130 app = create_app(flask_env)
131
132 if __name__ == '__main__':
133     app.run()
```

Flask-Admin does not have any affiliation with Flask-Login but it does provide `is_accessible` and `inacessible_callback` methods to its views. This allows the user to enter when authenticated or when it has access according to Flask-Login. The Flask-Admin will react accordingly. `add_link` enables us to add a custom link, which we used to add a logout button.

Use credentials `admin@domain.com` and password `pass` to access the demo app below.

flask-reuploaded

Please look at this section's code[34] to follow along.

Flask-Uploads[35] makes life easier by quickly enabling file uploads. However, it is unmaintained. In this lesson, we are using Flask-Reuploaded[36]. The code remains the same; it is only the installation that changes. For documentation purposes, though the Flask-Uploads docs is still valid, it is recommended to look at the Flask-Reuploaded[37] docs.

Flask-Uploads concepts

Flask-Uploads has upload sets that we can customize to allow whatever file extensions we want and destination paths for them. Calling the `.save` method then saves the file.

Implementation

Structure

```
.
├── app.py
├── static
│   └── img
└── templates
    └── upload.html
```

HTML

[34] https://github.com/FlaskMasterClass/s2-c5
[35] https://pythonhosted.org/Flask-Uploads/
[36] https://github.com/jugmac00/flask-reuploaded
[37] https://flask-reuploaded.readthedocs.io/en/latest/

```
<!doctype html>
<html lang=en>
<head>
    <meta charset=utf-8>
    <title>Flask-Reuploaded Example</title>
</head>
<body>
    {% with messages = get_flashed_messages() %}
    {% if messages %}
    <ul class=flashes>
    {% for message in messages %}
        <li>{{ message }}</li>
    {% endfor %}
    </ul>
    {% endif %}
    {% endwith %}

<form method="POST" enctype="multipart/form-data" action="{{ url_for('upload') }}">
    <input type="file" name="photo">
    <button type="submit">Submit</button>
</form>
</body>
</html>
```

app.py

```
import os

from flask import Flask
from flask import flash
from flask import render_template
from flask import request
from flask_uploads import IMAGES
from flask_uploads import UploadSet
from flask_uploads import configure_uploads

app = Flask(__name__)
photos = UploadSet("photos", IMAGES)
app.config["UPLOADED_PHOTOS_DEST"] = "static/img"
app.config["SECRET_KEY"] = os.urandom(24)
configure_uploads(app, photos)


```

```
@app.route("/", methods=["GET", "POST"])
def upload():
    if request.method == "POST" and "photo" in request.files:
        photos.save(request.files["photo"])
        flash("Photo saved successfully.")
        return render_template("upload.html")
    return render_template("upload.html")


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

The first thing to note is that we are defining an upload set

```
photos = UploadSet("photos", IMAGES)
```

then, we are adding a config

```
app.config["UPLOADED_PHOTOS_DEST"] = "static/img"
```

in the format

```
UPLOADED_*_DEST
```

where we add in between our uploadset name in caps.

We, then, call the `save` method.

```
photos.save(request.files["photo"])
```

The following is used to disable auto-generating a link for the uploaded file as if our images are private. An attacker, by default, can see it by guessing the URL.

```
app.config["UPLOADS_AUTOSERVE"] = False
```

Good to know

Variable flask_uploads	Types
ARCHIVES	gz, bz2, zip, tar, tgz, txz, 7z
AUDIO	wav, mp3, aac, ogg, oga, flac
DATA	csv, ini, json, plist, xml, yaml, yml
DEFAULTS	txt, rtf, odf, ods, gnumeric, abw, doc, docx, xls, xlsx, pdf, jpg, jpe, jpeg, png, gif, svg, bmp, webp, csv, ini, json, plist, xml, yaml, yml
DOCUMENTS	rtf, odf, ods, gnumeric, abw, doc, docx, xls, xlsx, pdf
EXECUTABLES	so, exe, dll
IMAGES	jpg, jpe, jpeg, png, gif, svg, bmp, webp
SCRIPTS	js, php, pl, py, rb, sh
SOURCE	c, cpp, c++, h, hpp, h++, cxx, hxx, hdl, ada, rs, go, f, for, f90, f95, f03, d, dd, di, java, hs, cs, fs, cbl, cob, asm, s
TEXT	txt

flask-wtforms

Please look at this section's code[38] to follow along.

Flask-WTForms[39] allows us to create, manage and validate forms easily. It also has support for file uploads and reCaptcha.

Flask-WTForms concepts

Flask-WTForms inherits its form classes from `FlaskForm`.

```
class WishForm(FlaskForm):
    pass
```

Each field is defined by adding them as class attributes.

```
class WishForm(FlaskForm):
    name = StringField("name")
```

We can then call a form object in our Jinja template.

```
{{ form.name.label }} <br>
{{ form.name() }} <br>
```

The above results in the below snippet.

```
<label for="name">name</label> <br>
<input id="name" name="name" type="text" value=""> <br>
```

Then we can add validations to the field using functions provided by Flask-WTForms or define our own.

```
class WishForm(FlaskForm):
    name = StringField("name", validators=[DataRequired()])
```

`DataRequired` is provided by default in Flask-WTForms and adds the `required` directive in the HTML tag.

To validate if the input values pass our validations, `form` objects have a `.validate_on_submit()` method.

[38] https://github.com/FlaskMasterClass/s2-c6
[39] https://flask-wtf.readthedocs.io/en/0.15.x/

```
if form.validate_on_submit():
  ...
```

Here's an app that applies these concepts.

Demo app

We'll now build an app that echoes what we have learned. We will deliberately throw in invalid values to read error messages.

Our app structure looks like this

```
.
├── app.py
└── templates
    └── home.html
```

```
# ...

# "name" is called the label of the field

class WishForm(FlaskForm):
    name = StringField("name", validators=[DataRequired()])
    email = StringField("email", validators=[DataRequired(), Email()])
    message = TextAreaField("message", validators=[DataRequired()])

# ...

@app.route("/", methods=["GET", "POST"])
def home():
    form = WishForm()
    if request.method == "POST":

        if form.validate_on_submit():
            name = form.name.data
            email = form.email.data
            message = form.message.data

            flash("Posted message:")
            flash(name)
            flash(email)
```

```
            flash(message)
            return redirect(url_for("home"))
        else:
            flash_errors(form)
            return redirect(url_for("home"))
    return render_template("home.html", form=form)


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

`flash_errors` is a custom function that flashes form errors by looping over error items. It gives the user an idea of what field in the form did not pass validation.

```
def flash_errors(form):
    """Flashes form errors"""
    for field, errors in form.errors.items():
        for error in errors:
            flash(
                u"Error in the %s field - %s"
                % (getattr(form, field).label.text, error),
                "error",
            )
```

In our template, we call the form and display the flashed messages.

```
<body>
    {% with messages = get_flashed_messages() %}
    {% if messages %}
    <ul class=flashes>
    {% for message in messages %}
        <li>{{ message }}</li>
    {% endfor %}
    </ul>
    {% endif %}
    {% endwith %}

<form action="{{ url_for('home') }}" method="POST">
    Write your wish<br><br>
    {{ form.name.label }} <br>
    {{ form.name() }} <br>
    {{ form.email.label }} <br>
```

```
17      {{ form.email() }} <br>
18      {{ form.message.label }} <br>
19      {{ form.message() }} <br>
20      {{ form.csrf_token() }}
21      <input type="submit" value="submit">
22  </form>
23  </body>
```

Calling form elements occur in the pattern:

`form.attribute.label` and `form.attribute()`

For the form attribute `name`, in the jinja template we call ‘ form.name.label `and` form.name()‘.

Note:

We also need to install the `email-validator` library if we are using the default mail validation function as it is not installed by default.

Good to know

Here are some more fields from WTForms:

- BooleanField
- DateField
- DateTimeField
- DecimalField
- FieldList
- FileField
- FloatField
- FormField
- HiddenField
- IntegerField
- MultipleFileField
- PasswordField
- RadioField
- SelectField
- SelectMultipleField
- StringField
- SubmitField
- TextAreaField
- TextField
- TimeField

Note: Refer to this link[40] for more on `flash_errors` function.

[40] https://stackoverflow.com/questions/13585663/flask-wtfform-flash-does-not-display-errors

marshmallow-sqlalchemy

Please look at this section's code[41] to follow along.

Flask-Marshmallow[42] integrates marshmallow[43] with Flask. It also ships with marshmallow-sqlalchemy[44]. It requires flask-sqlalchemy[45] to work. Marshmallow-Sqlalchemy allows the conversion of Sqlalchemy models into API responses.

Why is Marshmallow-Sqlalchemy useful?

Marshmallow is a framework-agnostic library for converting complex data types, such as objects, to and from native Python datatypes. In short, marshmallow schemas can be used to:

- Validate input data.
- Deserialize input data to app-level objects.
- Serialize app-level objects to primitive Python types. The serialized objects can then be rendered into standard formats such as JSON in an HTTP API.

Marshmallow makes use of schemas to define structures.

```
1 class Author(Schema):
2     name = fields.Str()
```

Marshmallow-Sqlalchemy allows us to convert our model to schemas easily from where we can convert it to JSON. As an example, it can save us the pain of rewriting schemas to match the column types of our models and updating the schemas when the models update.

Implementing Flask-marshmallow

[41] https://github.com/FlaskMasterClass/s2-c7
[42] https://flask-marshmallow.readthedocs.io/en/latest/
[43] https://marshmallow.readthedocs.io/en/stable/
[44] https://marshmallow-sqlalchemy.readthedocs.io/en/latest/
[45] https://flask-sqlalchemy.palletsprojects.com/en/2.x/

```
pip install marshmallow==3.14.1
pip install marshmallow-sqlalchemy==0.26.1
pip install flask-marshmallow==0.14.0
```

We are using the same structure as the first lesson of this section on Flask-SQLAlchemy.

We configure our `init.py` as follows.

```
from flask_sqlalchemy import SQLAlchemy
from flask_marshmallow import Marshmallow

db = SQLAlchemy()
ma = Marshmallow() # always after sqlalchemy definition
```

Let's modify our User class to have a name field.

```
class User(db.Model):
    __tablename__ = "users"
    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    name = db.Column(db.String)
```

And write our schema.

```
class UserSchema(ma.SQLAlchemyAutoSchema):
    class Meta:
        model = User
        include_fk = True
```

Let's create the DB and users.

```
Python 3.9.5 (default, May 19 2021, 11:32:47)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from app import app
>>> from app import User
>>> from init import db
>>> with app.app_context():
...     db.create_all()
...
>>> u1 = User(name='Adam')
>>> u2 = User(name='Aon')
>>> with app.app_context():
```

```
...     db.session.add(u1)
...     db.session.add(u2)
...     db.session.commit()
...
>>>
```

And the endpoint to dump our JSON

```
if __name__ == '__main__':
    @app.route('/<int:user_id>', methods=['GET'])
    def user_id(user_id):
        user = User.query.get(user_id)
        user_schema = UserSchema()
        return jsonify(user_schema.dump(user))
    # code to run app
```

querying

```
/1
```

fetches

```
{
  "id": 1,
  "name": "Adam"
}
```

`jsonify` is a normal Flask function and `user_schema.dump(user)` dumps the json

We can also get a list of all users.

```
@app.route('/all', methods=['GET'])
def user_all():
    user_schema = UserSchema()
    return jsonify(user_schema.dump(User.query.all(), many=True))
```

We can pass in any query instead of `User.query.all`. `many=True` returns a list of items.

Querying

```
/all
```

returns

```
[
  {
    "id": 1,
    "name": "Adam"
  },
  {
    "id": 2,
    "name": "Aon"
  }
]
```

Choosing what fields to display

We can also specify what fields to display.

```
class UserSchema(ma.SQLAlchemyAutoSchema):
    class Meta:
        model = User
        include_fk = True
        fields = ['name']
```

The same query returns the response with only the specified fields when we run it.

```
[
  {
    "name": "Adam"
  },
  {
    "name": "Aon"
  }
]
```

Here's the code:

wtforms-sqlachemy

Please look at this section's code[46] to follow along.

To easily manage forms, we have the wtforms package. But if we have to edit and create sqlalchemy objects, we still have to write form codes that match the sqlalchemy models. And try not to forget to update form codes when the model changes. Konsta Vesterinen wrote the life-saving wtforms-alchemy[47] library to solve this.

Our app

Our app builds on the app in the Flask-Wtforms chapter. It looks like this:

```
.
├── app.db
├── app.py
├── config.py
├── init.py
├── instance
│   └── config.py
└── templates
    ├── edit.html
    └── home.html
```

Note: We also require Flask-WTForms.

```
pip install WTForms-Alchemy==0.17.0
```

Implementation

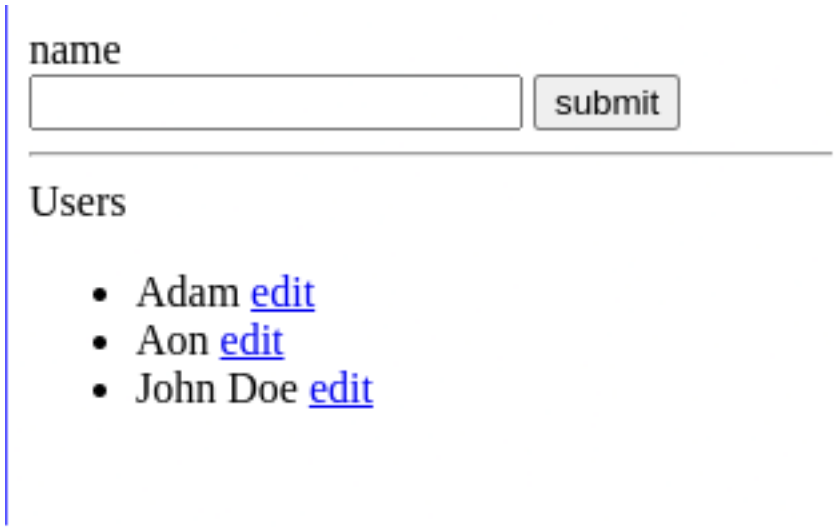
Forms using Wtforms-alchemy need to inherit from a `FlaskForm` class. Here's what our init.py looks like:

[46] https://github.com/FlaskMasterClass/s1-c8
[47] https://wtforms-alchemy.readthedocs.io/en/latest/

```
from flask_sqlalchemy import SQLAlchemy
from flask_wtf import FlaskForm
from wtforms_alchemy import model_form_factory

db = SQLAlchemy()


BaseModelForm = model_form_factory(FlaskForm)

class ModelForm(BaseModelForm):
    @classmethod
    def get_session(self):
        return db.session
```

Our app will have two pages.

home page:



edit page:

We note that creating our form from a model is incredibly easy.

```
class UserForm(ModelForm):
    class Meta:
        model = User
```

We can go ahead and create our db as usual.

```
Python 3.9.5 (default, May 19 2021, 11:22:33)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from app import app
>>> from app import User
>>> from init import db
>>> with app.app_context():
...     db.create_all()
...
>>>
```

Homepage

In our routes, just include the form we defined.

```
    @app.route("/", methods=["GET", "POST"])
    def home():
        form = UserForm()
        users = User.query.all()
        if request.method == "POST":

            if form.validate_on_submit():
                u = User(name=form.data["name"])
                db.session.add(u)
                db.session.commit()
                return redirect(url_for("home"))
            else:
                flash_errors(form)
                return redirect(url_for("home"))
        return render_template("home.html", form=form, users=users)
```

Our page

```
<body>
    <!-- flash message code goes here -->

<form action="{{ url_for('home') }}" method="POST">
    {% for field in form %}
        {{ field.label }}<br>
        {{ field }}
    {% endfor %}
    <input type="submit" value="submit">
</form>
<hr>
Users
<ul>
    {% for user in users %}
        <li> {{ user.name }} <a href="{{ url_for('edit', user_id=user.id) }}">edit</\
a> </li>
    {% endfor %}
</ul>
</body>
```

We see that we did not have anything to write.

```
{% for field in form %}
    {{ field.label }}<br>
    {{ field }}
{% endfor %}
```

Just looping over the form object generates our form. However, the real power of wtforms-alchemy comes when editing objects.

Edit page

Route

```
@app.route("/edit/<int:user_id>", methods=["GET", "POST"])
    def edit(user_id):
        user = User.query.get(user_id)
        form = UserForm(obj=user)
        if request.method == "POST":

            if form.validate_on_submit():
                form.populate_obj(user)
                db.session.commit()
                return redirect(url_for("home"))
            else:
                flash_errors(form)
                return redirect(url_for("home"))
        return render_template("edit.html", form=form, user=user)
```

Html page

```
<body>
    {% with messages = get_flashed_messages() %}
    {% if messages %}
    <ul class=flashes>
    {% for message in messages %}
        <li>{{ message }}</li>
    {% endfor %}
    </ul>
    {% endif %}
    {% endwith %}
Edit
<form action="{{ url_for('edit', user_id=user.id) }}" method="POST">
```

```
    {% for field in form %}
        {{ field.label }}<br>
        {{ field }}
    {% endfor %}
    <input type="submit" value="submit">
</form>
</body>
```

The code is nearly the same. However, we see that:

- 1. the form is pre-filled.

We do it by specifying `obj=`.

```
user = User.query.get(user_id)
form = UserForm(obj=user)
```

- 2. It magically edits our user.

This is because we update our user with the form details then save.

```
if form.validate_on_submit():
    form.populate_obj(user)
```

Dealing with the CSRF Token

We can filter out the csrf label by checking if `field.id` is equal to `'csrf_token'`, but we do include it at the end without a label.

```
{% for field in form %}
    {% if field.id not in ['csrf_token'] %}
    {{ field.label }}<br>
    {{ field }}
    {% else %}
        {{field()}}
    {% endif %}
{% endfor %}
```

flask-mailman

Please look at this section's code[48] to follow along.

A common package for sending mails in Flask is Flask-Mail[49] but it's unmaintained. Flask-Mailman[50] is a port of Django's email implementation with awesome docs[51]. Flask-Mailman allows us to send mails using the SMTP provider we specify.

Implementation

In our init.py file

```
from flask_mailman import Mail

mail = Mail()
```

It expects some config keys to be set

```
class Config:
    ...
    MAIL_SERVER = ''
    MAIL_PORT = 0
    MAIL_USE_TLS = False
    MAIL_USE_SSL = False
    MAIL_USERNAME = ''
    MAIL_PASSWORD = ''
    MAIL_DEFAULT_SENDER = ''
```

Then sending a mail is as simple as

[48] https://github.com/FlaskMasterClass/s2-c9
[49] https://github.com/mattupstate/flask-mail
[50] https://github.com/waynerv/flask-mailman
[51] https://waynerv.github.io/flask-mailman/

```
from flask_mailman import EmailMessage

from init import mail
...
    ...
    @app.route("/send-mail", methods=["GET", "POST"])
    def home():
        msg = EmailMessage(
            f'Hello Test {random.randint(0, 100)}',
            f"You choose {random.choice(['Mango', 'Pear'])}",
            'from@educative.example',
            ['to1@educative.example', 'to2@educative.example'],
            ['bcc@educative.example'],
            reply_to=['another@example.com'],
            headers={'Message-ID': 'foo'},
        )
        msg.send()
        return 'sending mail ...'
```

Each time we call `/send-mail`, an email will be sent!

Watch emails arrive

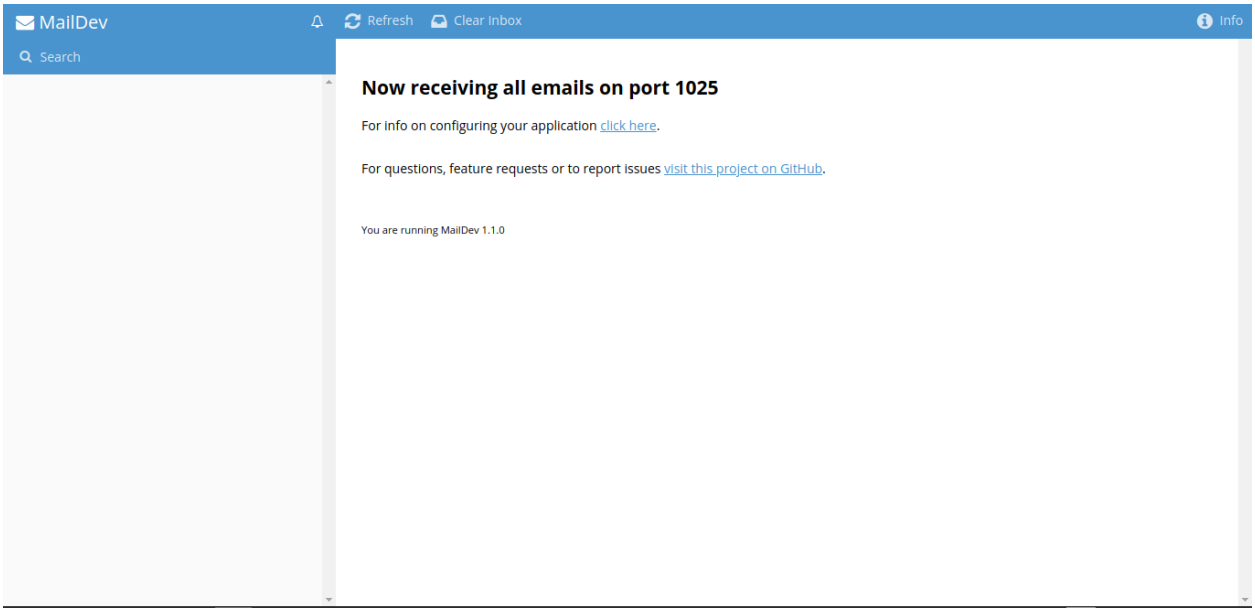
One of the main reasons we covered this extension is how to configure the mail dev environment. In case we have NodeJs and npm installed, we'll be using the maildev package. First, we install it.

```
npm install -g maildev
```

Then run `maildev`, giving something like that

```
$ maildev
MailDev webapp running at http://0.0.0.0:1080
MailDev SMTP Server running at 0.0.0.0:1025
```

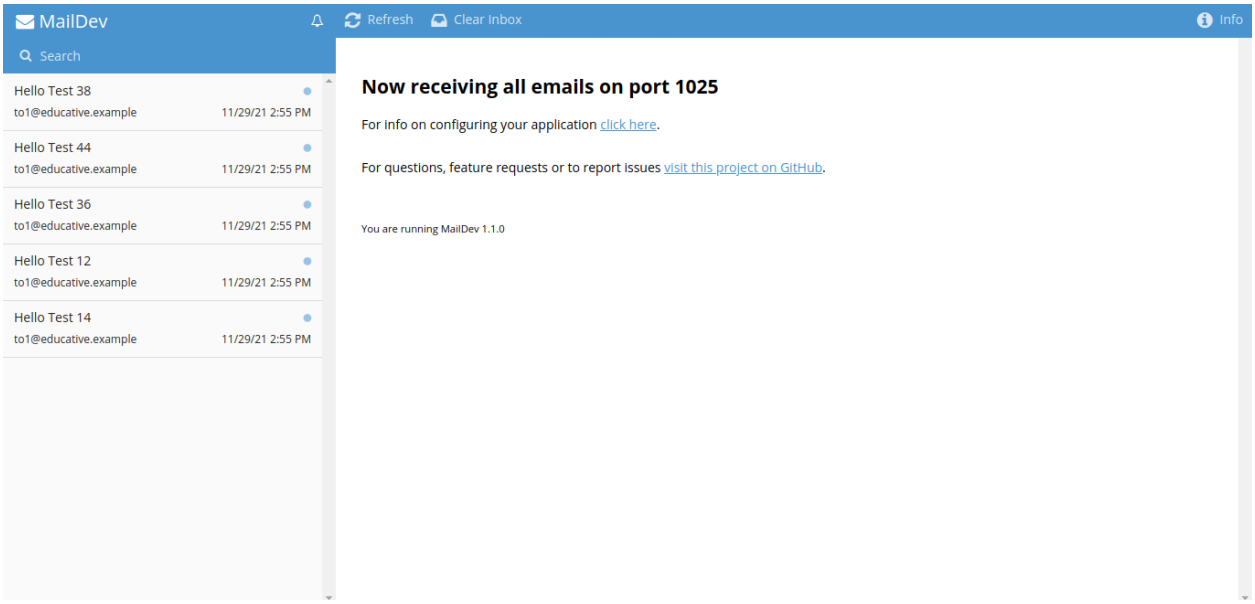
Opening the web app URL gives:



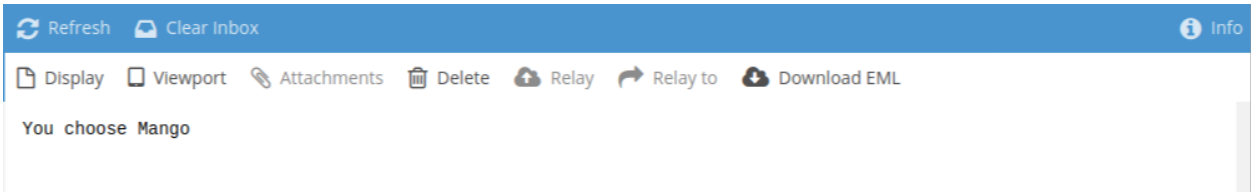
Then we set our config to match maildev's.

```
class Config:
    # ...
    MAIL_SERVER = 'localhost'
    MAIL_PORT = 1025
    MAIL_USE_TLS = False
    MAIL_USE_SSL = False
    MAIL_USERNAME = '' # os.environ.get("MAIL_USERNAME")
    MAIL_PASSWORD = '' # os.environ.get("MAIL_PASSWORD")
    MAIL_DEFAULT_SENDER = 'ma@mail.com' # os.environ.get("MAIL_DEFAULT_SENDER")
```

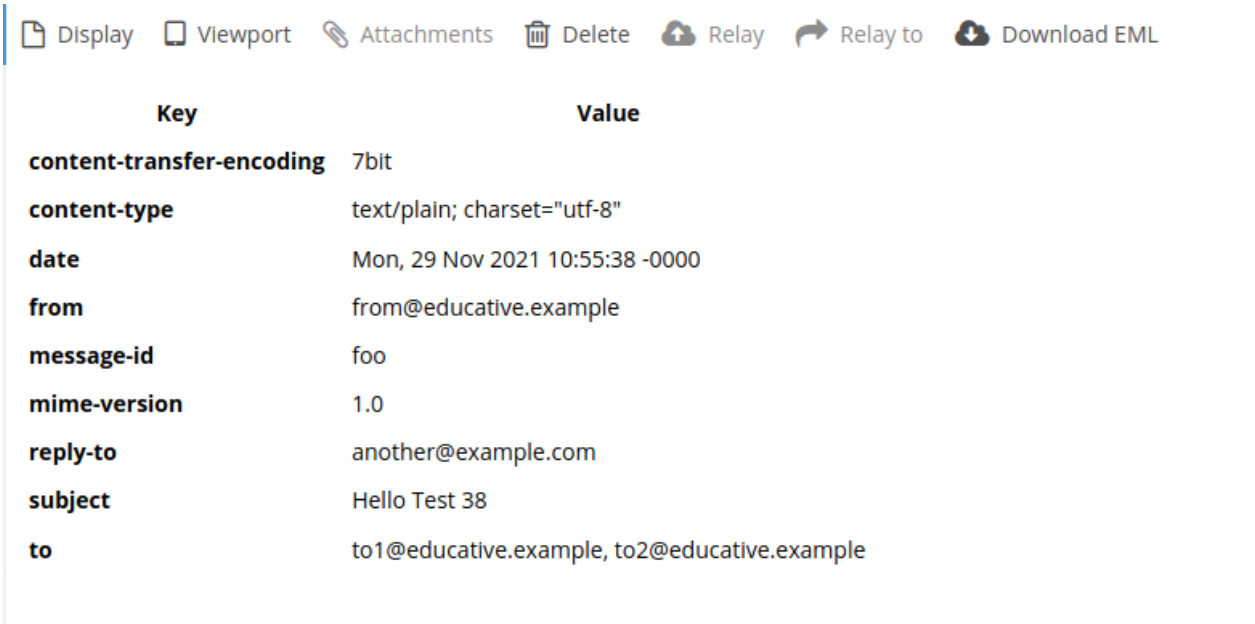
Running the app and calling `send-mail` a few times results in new mails arriving



Which reveals the mail content when opened.



And has options to view the headers under Display.



In shared hosting, `MAIL_SERVER` is often `localhost`.

Section 2 Quiz

Take this quiz online[52]

Take this quiz online[53]

Take this quiz online[54]

Take this quiz online[55]

Take this quiz online[56]

Take this quiz online[57]

Take this quiz online[58]

Take this quiz online[59]

Take this quiz online[60]

Take this quiz online[61]

Take this quiz online[62]

Take this quiz online[63]

Take this quiz online[64]

[52] http://leanpub.com/courses/leanpub/flask-masterclass/quizzes/quiz-6
[53] http://leanpub.com/courses/leanpub/flask-masterclass/quizzes/quiz-7
[54] http://leanpub.com/courses/leanpub/flask-masterclass/quizzes/quiz-8
[55] http://leanpub.com/courses/leanpub/flask-masterclass/quizzes/quiz-9
[56] http://leanpub.com/courses/leanpub/flask-masterclass/quizzes/quiz-10
[57] http://leanpub.com/courses/leanpub/flask-masterclass/quizzes/quiz-11
[58] http://leanpub.com/courses/leanpub/flask-masterclass/quizzes/quiz-12
[59] http://leanpub.com/courses/leanpub/flask-masterclass/quizzes/quiz-13
[60] http://leanpub.com/courses/leanpub/flask-masterclass/quizzes/quiz-14
[61] http://leanpub.com/courses/leanpub/flask-masterclass/quizzes/quiz-15
[62] http://leanpub.com/courses/leanpub/flask-masterclass/quizzes/quiz-16
[63] http://leanpub.com/courses/leanpub/flask-masterclass/quizzes/quiz-17
[64] http://leanpub.com/courses/leanpub/flask-masterclass/quizzes/quiz-18

Pushing The Modular Approach Further

The organization of components plays a great role in the maintainability of a project. We'll discuss why the modular approach works great for a web developer.

Benefits of the modular pattern

There is a tendency to group related activities under the same folder in the Flask world.

```
app.py

routes/
    payment.py
    auth.py

forms/
    forms.py

models/
    payment.py
    auth.py

tests/
        auth.py
        payment.py
```

This pattern generally evolves as a side effect of refactoring on the fly. A modular approach would group related components.

```
app.py

payment/
    view.py
    models.py
    forms.py
    test_payment.py

auth/
    view.py
    models.py
    forms.py
    test_auth.py
```

This pattern has the following advantages:

- **Separation of concerns**: We can work on features separately. If we have many developers, we can separate the area of work.
- **Reuse**: We can use a module developed for one project in another.
- **Consistency**: Developers navigating the codebase have a clear idea of what to expect in each module, and the collection of modules gives a clearer picture of how the system operates

We don't have to visit some five folders to remove a feature. We only remove the module.

Caveats of modular patterns

If our modules depend a lot on each other, we'd have trouble tracking what's going on, reducing the separation efficiency provided.

Conclusion

Using a modular approach helps immensely in not rewriting from scratch, thus outperforming forecasted delivery dates. But there is one last sauce missing for modular success: practice standardization.

Enhancing The Modular Approach

One of the reasons people stick to Flask is that it gives back the joy of coding. Being able to code our app the way we want brings extraordinary motivation. But, a mere do-as-we-wish approach might prove to be tiring if it translates into more unnecessary work. That's why we need to ace a winning formula. The modular pattern is such a formula. It is proven to work, but we need to detail and customize it in the Flask context. The approach described in the last lesson is already a very nice improvement over most projects we can potentially encounter. Here are some ideas which can help boost productivity.

Registering blueprints on the fly

Let us consider the pattern we have.

```
app.py

payment/
    view.py
    models.py
    forms.py
    test_payment.py

auth/
    view.py
    models.py
    forms.py
    test_auth.py
```

We'd still need to write blueprint registering codes. It will look something like this:

```
from payment.view import payment_blueprint
from auth.view import auth_blueprint

app.register_blueprint(payment_blueprint)
app.register_blueprint(auth_blueprint)
```

One way to automate it is to group all modules under a folder. Then we iterate over them, find their **<text>view.py</text>**, and import the blueprint from there.

```
app.py

modules/
    payment/
        view.py
        models.py
        forms.py
        test_payment.py

    auth/
        view.py
        models.py
        forms.py
        test_auth.py
```

Our pseudocode will look like this.

```
for folder in modules_folder:
    blueprint = importlib.import(f'modules.{folder}.view.{folder}_blueprint')
    app.register_blueprint(blueprint)
```

The above works if the module folder and blueprint name prefix stay the same. If we don't want to group all modules inside a folder, we need a way to differentiate how modules are named by requiring a prefix.

```
app.py

module_payment/
    view.py
    models.py
    forms.py
    test_payment.py

module_auth/
    view.py
    models.py
    forms.py
    test_auth.py
```

Or, we can add a constant LOAD_APPS = ['payment', 'auth'] in the config file. Afterwords, we load blueprints from folders specified in the list so that our app knows what to look for.

Automating the creation of modules

Since each module will have more or less the same structure, we can use the lesson on Commandline Arguments to create a new module with the required files and file contents.

```
flask createmodule auth
```

In the example above, `createmodule` is the command while `auth` is the parameter.

The facultative nature of module elements

When loading blueprints, it's up to us to validate elements found in an app. Should **<text>forms.py</text>** be made optional? Or even **<text>view.py</text>**? Those are questions up to us to answer.

Conclusion

This lesson and the upcoming lesson aim to provide readers with enough seed materials to fine-tune the modular experience. We have the flexibility to choose elements and setups which ring well with us, in line with the spirit of Flask!

Assets Management

Sooner or later, we'll feel the need to serve assets. But, it is against best practices to serve them from a Flask app. It's better to delegate the task to a proper server like Nginx or Apache. We'll discover how to manage it in relation to modules.

Modular assets management

We will be serving our assets from the static folder. Flask serves assets by default on `/static/` routes. In production, we will configure it such that `/static/img/sand.jpg` is served from a folder called **img** found inside the **static** folder. Nginx will intercept the request and serve it.

Now, a module with assets might look like this

```
payment/
    view.py
    models.py
    forms.py
    test_payment.py
    static/
        img.jpg
```

Our goal is to move it to the static folder

```
app.py
static/
    modules/
        payment/ # needs to be here
            img.jpg
modules/
    payment/
        view.py
        models.py
        forms.py
        test_payment.py
        static/
            img.jpg
```

This is why Django has the `collectstatic` command. Running `collectstatic` means copying assets from each module into the desired folder in the static folder at the root of the app.

Implementing `collectstatic` serving

We have two choices:

1. Run `collectstatic` after every change in assets so that our `static` folder always contains all assets. Then we serve from `/static/modules/<modulename>/<static_path>`
2. Have two urls for serving images. `/static/modules/<modulename>/<static_path>` for production and `/mycustomroute/<modulename>/<static_path>` for development which uses `send_from_directory` to return the asset from path **/modules/modulename/static/static_path**.

Conclusion

The `collectstatic` command requires some considerate implementation. Serving assets from our web server works fine with low-traffic apps. We might want to explore cloud providers or dedicated companies' offerings for higher-volume sites in terms of asset serving.

More Project Management Options

We might want to give our site some fresh looks time and again. Or, we might wish to have custom designs based on the time of the year. Knowing how to implement a theming system might help customize the looks of our site.

Implementing theming

It's straightforward to implement theming, and all we need is to have custom locations for our template files. We typically achieve this feat with the code below.

```
# BASE_DIR is the app's root directory path
with app.app_context():
    theme_dir = os.path.join(
        app.config["BASE_DIR"], "static", "themes"
    )
    my_loader = jinja2.ChoiceLoader(
        [
            app.jinja_loader,
            jinja2.FileSystemLoader([theme_dir]),
        ]
    )

    app.jinja_loader = my_loader
```

Inside **/static/** we can have our themes.

```
static/
    snowman/
        landing_page.html
    margeurita/
        landing_page.html
```

We need to have a configuration called `CURRENT_THEME`. Then a typical view render would look like this

```
# ...
theme = current_app.config['CURRENT_THEME']
return render_template(f'{theme}/landing_page.html')
```

Spawning new projects

Flask projects need a proper structure to get started for serious development. But each time, we have to create some files and folders with content. We'll learn how to get started on new projects quickly.

Cookiecutter[65] is a command-line utility to create projects from templates. We can create files and folders manually, but, cookiecutter saves time by changing variables in the template itself. Loading a template from Github is as simple as

```
cookiecutter https://github.com/audreyr/cookiecutter-pypackage
```

The README[66] has more pieces of information as to its actual usage.

[65] https://pypi.org/project/cookiecutter/
[66] https://github.com/cookiecutter/cookiecutter

Section 3 Quiz

Take this quiz online[67]

[67] http://leanpub.com/courses/leanpub/flask-masterclass/quizzes/quiz-19

Introduction To pytest

Please look at this section's code1[68], code2[69] to follow along.

There are many different testing frameworks around. Python has an in-built one called `unittest`. But Pytest is a much more convenient framework with advanced features. We will now cover how to get started.

Getting started with PyTest

Let's say we have a maths function

```
def add(a, b):
        return a + b
```

We can test it as follows.

```
def test_add():
        assert add(1, 2) == 3
```

Dealing with multiple files

PyTest is a potent tool, and it allows us to focus on writing tests and not worry about anything, not even structure.

If we have tests in different places and want to run all of them, profile the file names with `test_`.

```
feature1/
    test_feature1.py
feature2/
    test_feature2.py
```

We can also group them in a folder called tests. The way to run them is no different.

[68] https://github.com/FlaskMasterClass/s3-c1-1
[69] https://github.com/FlaskMasterClass/s3-c1-2

```
feature1/
feature2/
tests/
    test_feature1.py
    test_feature2.py
```

It does not matter where our file is. We can just run in our root folder as shown below.

```
python -m pytest .
```

Running the below commands should give an output like this.

```
============================= test session starts ==============================
platform linux -- Python 3.9.5, pytest-6.2.5, py-1.11.0, pluggy-1.0.0
rootdir: /path/to/folder/
collected 2 items

feature1/test_feature1.py .                                              [ 50%]
feature2/test_feature2.py .                                              [100%]

============================== 2 passed in 0.02s ===============================
```

Pytest gives a list of files it discovered. Then, the dot represents tests that passed. The percentage of test passed is also given, here 100%.

Defining Flask Fixtures

Please look at this section's code[70] to follow along.

Flask provides a default client for use in tests. But fixtures help us write elegant functions.

Flask fixtures

Flask natively provides the `app.test_client()` invocation for tests. An example usage is

```
assert 'x' in app.test_client().get("/").data
```

We can define the following fixtures

```
import pytest
from app import create_app

@pytest.fixture()
def app():
    app = create_app('testing')

    yield app


@pytest.fixture()
def client(app):
    return app.test_client()
```

Which enable us not to repeat ourselves in tests

[70] https://github.com/FlaskMasterClass/s3-c2

```
def test_get(client):
    response = client.get("/")
    assert b"abcd" in response.data


def test_post(client):
    response = client.post("/post_here", data={
        "fruit":"apple"
    })
    assert response.status_code == 200
```

The same goes for command-line arguments.

```
# conftest.py
@pytest.fixture()
def runner(app):
    return app.test_cli_runner()

# actual test
def test_cli(runner):
    result = runner.invoke(args=["greet"])
    assert "Greetings Flask users all over the world!" in result.output
```

Documenting With Sphinx

Please look at this section's code[71] to follow along.

Sphinx[72] is a powerful documentation tool. It was started by the same group of developers who started Flask. It is stable, mature, and is used as the default documentation tool for Python itself.

Set up Sphinx

First, we need to install Sphinx

```
python -m pip install sphinx
```

Then the `sphinx-quickstart` executable will be made available to us. Though we can create a project by ourselves, `sphinx-quickstart` helps us by providing a base to start with. We can configure it later.

Running `sphinx-quickstart` in your project directory runs through a setup like this

```
Welcome to the Sphinx 3.2.1 quickstart utility.

Please enter values for the following settings (just press Enter to
accept a default value, if one is given in brackets).

Selected root path: .

You have two options for placing the build directory for Sphinx output.
Either, you use a directory "_build" within the root path, or you separate
"source" and "build" directories within the root path.
> Separate source and build directories (y/n) [n]:

The project name will occur in several places in the built documentation.
> Project name: MyProject
> Author name(s): ARJ
> Project release []: 0.0.1

If the documents are to be written in a language other than English,
you can select a language here by its language code. Sphinx will then
```

[71] https://github.com/FlaskMasterClass/s3-c3
[72] https://www.sphinx-doc.org/en/master/

```
translate text that it generates into that language.

For a list of supported codes, see
https://www.sphinx-doc.org/en/master/usage/configuration.html#confval-language.
> Project language [en]:

Creating file ./docs/conf.py.
Creating file ./docs/index.rst.
Creating file ./docs/Makefile.
Creating file ./docs/make.bat.

Finished: An initial directory structure has been created.

You should now populate your master file ./docs/index.rst and create other documenta\
tion
source files. Use the Makefile to build the docs, like so:
   make builder
where "builder" is one of the supported builders, e.g. html, latex or linkcheck.
```

Then you can build the html docs using the command

```
make HTML
```

Then in your browser you you can navigate to `docs/_build/html`

Bonus

Sphinx by default uses the RST format for documentation. In case you want to add actual codes from your project, the syntax will look like this

```
.. literalinclude:: ../file.py
   :language: python
   :linenos:
   :lines: 1-1
```

Logging

Please look at this section's code[73] to follow along.

We are not covering logging in its intricate details here as it's a vast topic in Python. Here we will only cover how to save log statements to a file.

Re-routing output to a file is not very difficult, we only need to specify the file.

```
logging.basicConfig(
    filename='record.log',
    level=logging.DEBUG,
    format=f'%(asctime)s %(levelname)s %(name)s %(threadName)s : %(message)s')
```

Then use as follows

```
app.logger.info('Info level log')
```

Run the app below. Then in the terminal, `ctrl + c` then run `cat record.log` to see the log.

[73] https://github.com/FlaskMasterClass/s3-c4

Section 4 Quiz

Take this quiz online[74]

Take this quiz online[75]

Take this quiz online[76]

[74] http://leanpub.com/courses/leanpub/flask-masterclass/quizzes/quiz-20
[75] http://leanpub.com/courses/leanpub/flask-masterclass/quizzes/quiz-21
[76] http://leanpub.com/courses/leanpub/flask-masterclass/quizzes/quiz-22

APIs With Flask

Using Flask's jsonify[77] function, you can craft responses. However, frameworks designed for building APIs also provide parameter validation and auto-documentation generation. Let us have a look at some frameworks for building APIs.

Available frameworks for developing API apps

We need to look at two criteria to see whether or not a framework is suited for APIS.

1. Input validation

Frameworks providing input validation save time writing manual codes for validating parameters and value types. Additionally, having a way to check for expected output guards against us crafting unintended responses.

2. Documentation generation

Developers appreciate Swagger-like interfaces are very much for viewing API docs. It also provides an interface for testing requests and notifies endpoints requiring token authentication. You can still write Swagger documentation by hand using typically `.yml` files. But, it is time-consuming, and Frameworks, instead, auto-generate the needed swagger docs.

Here are some frameworks satisfying the above conditions

- Flask-Smorest[78] uses Marshmallow for input and output validation. It autogenerates Swagger docs.
- Grey Li, a Flask maintainer, maintains APIFlask[79]. It is very promising and tries to be as thorough as possible.
- Miguel Grinberg maintains APIFairy[80]. It tries to give the user as much freedom as possible.
- Flask-RestX[81] is the maintained fork of Flask-Restful/Flask-RestPlus.

My personal choice is for Flask-Smorest. But we will cover Flask-RestX as it has the most resources and solved cases around. Since it is the maintained version of Flask-Restful, people have been documenting its usage for a really long time.

[77] https://flask.palletsprojects.com/en/2.0.x/api/#flask.json.jsonify
[78] https://flask-smorest.readthedocs.io/en/latest/
[79] https://apiflask.com/
[80] https://apifairy.readthedocs.io/en/latest/intro.html
[81] https://flask-restx.readthedocs.io/en/latest/

flask-restx Fundamentals

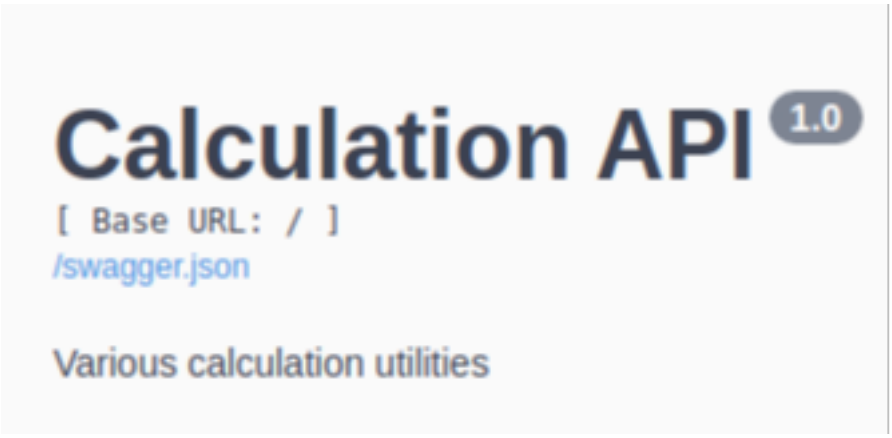
The first thing to do is to install Flask-RestX

```
python -m pip install flask-restx
```

Let us get started by a minimal example

```
from flask import Flask
from flask_restx import Resource, Api, fields, reqparse

app = Flask(__name__)
api = Api(app, version="1.0", title="Calculation API", description="Various calculat\
ion utilities")


answer = api.model('Answer', {
    'answer': fields.Integer
})
num_parser = reqparse.RequestParser()
num_parser.add_argument('number1', type=int, location='args', help='First number to \
be multiplied')
num_parser.add_argument('number2', type=int, location='args', help='Second number to\
 be multiplied')

@api.route('/multiply')
class Multiply(Resource):
    @api.response(200, 'Success', answer)
    @api.doc(parser=num_parser)
    def get(self):
        '''
        Multiply 2 numbers
        '''
        args = num_parser.parse_args()
        return {'answer': args['number1']*args['number2']}

if __name__ == '__main__':
```

Let's break it down. `API()` defines the Swagger informations



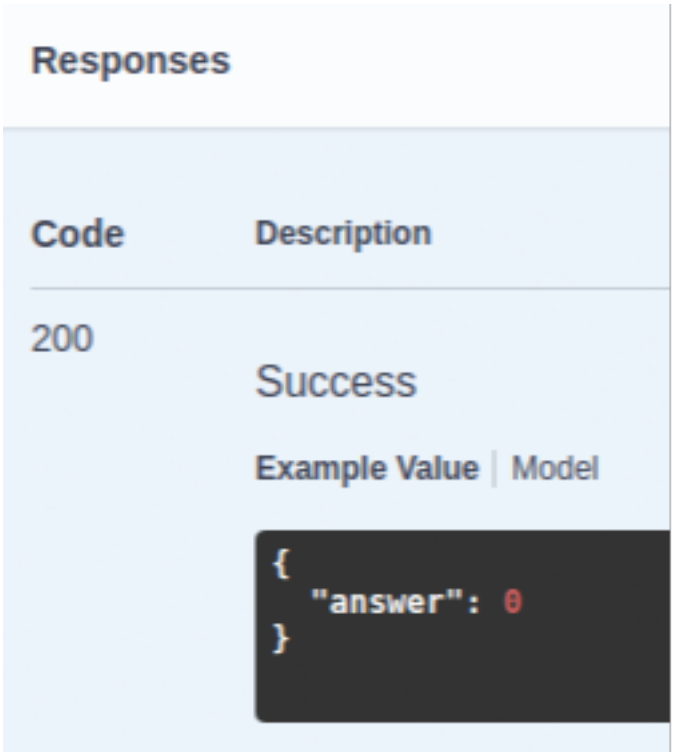

```
api = Api(app, version="1.0", title="Calculation API", description="Various calculat\
ion utilities")
```

```
num_parser = reqparse.RequestParser()
num_parser.add_argument('number1', type=int, location='args', help='First number to \
be multiplied')
num_parser.add_argument('number2', type=int, location='args', help='Second number to\
 be multiplied')
```

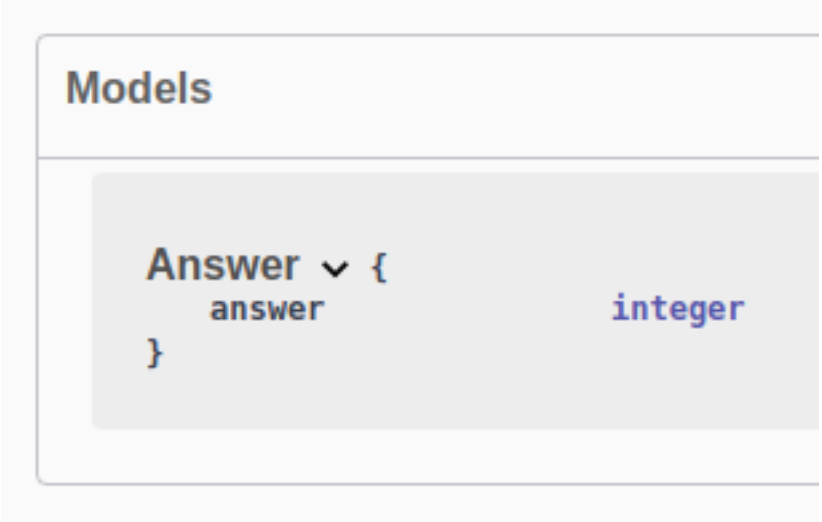
The `RequestParser` specifies the argument name, type and location. By location we mean in form or URL parameter or header etc. The `help` parameter specifies the parameter's description.

We also document how the response looks like when status code 200 is returned.

```
answer = api.model('Answer', {
    'answer': fields.Integer
})
# ...

@api.route('/multiply')
class Multiply(Resource):
    @api.response(200, 'Success', answer)
    # ...
    def get(self):
        ...
```



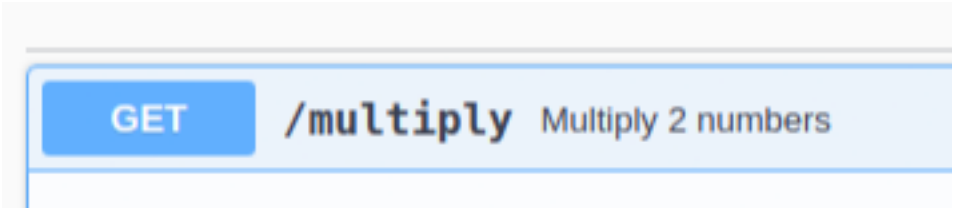
The response model is also listed with all models.



The docstring is used to describe the purpose of the endpoint.

```
    def get(self):
        '''
        Multiply 2 numbers
        '''
```

Which generates the following



Advanced flask-restx Concepts

Flask-RestX provides many features in terms of organization and better writing of responses. We'll cover them next.

Custom input validation

Let's say we are defining an input of type string.

```
from flask_restx import reqparse

login_parse = reqparse.RequestParser()
login_parse.add_argument("email", type=str, required=True)
```

But, if we want to, let's say, accept emails of a maximum length of 50, we would have to define our custom validators.

```
def str_max_length(max_length):
    def validate(s):
        if len(s) <= max_length:
            return s
        raise ValueError("String must be less or equal %i characters" % max_length)

    return validate
```

Then we replace `str`.

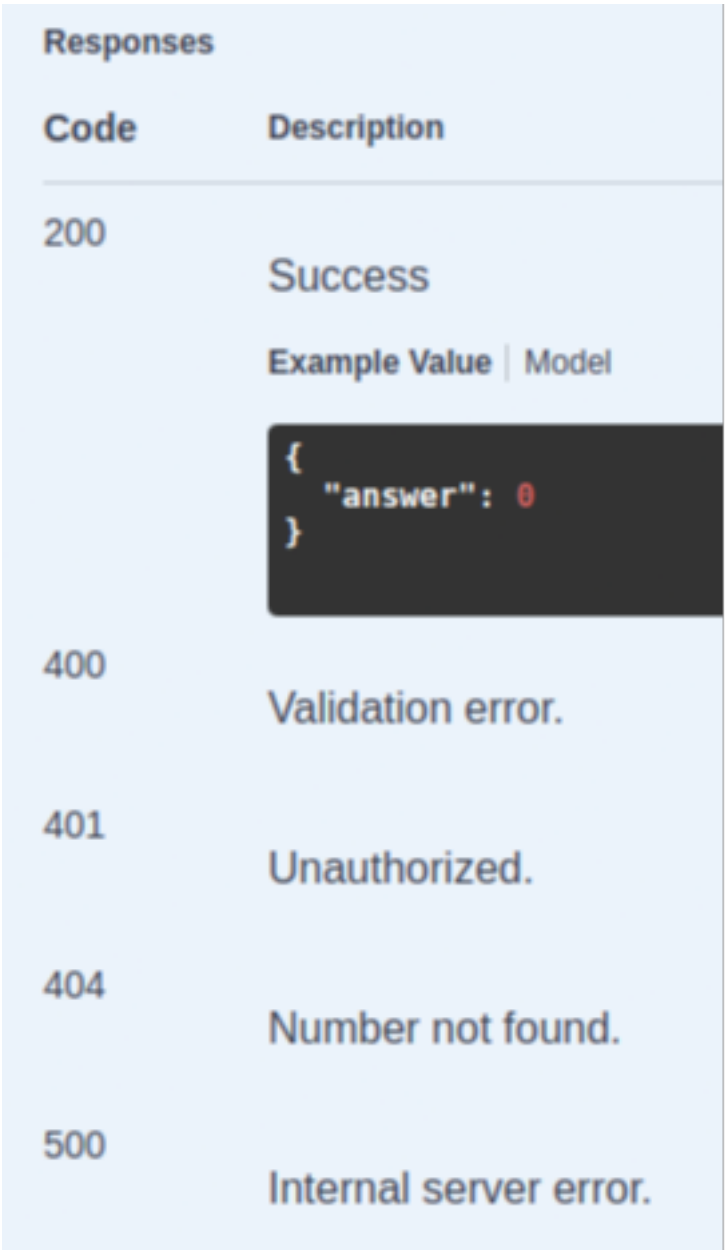
```
login_parse.add_argument("email", type=str_max_length(50), required=True)
```

Defining default response messages for status codes

Sometimes you need to document what status codes represent. Instead of using raw numbers like `200` and `404`, you can use the in-built `http` module with descriptive names for the codes. This makes reading the codes easier.

```
from http import HTTPStatus
# ...
@api.route('/multiply')
@api.response(int(HTTPStatus.NOT_FOUND), "Number not found.")
@api.response(int(HTTPStatus.BAD_REQUEST), "Validation error.")
@api.response(int(HTTPStatus.UNAUTHORIZED), "Unauthorized.")
@api.response(int(HTTPStatus.INTERNAL_SERVER_ERROR), "Internal server error.")
class Multiply(Resource):
    # ...
```

The above produces



This allows consumers to handle responses better.

Namespaces

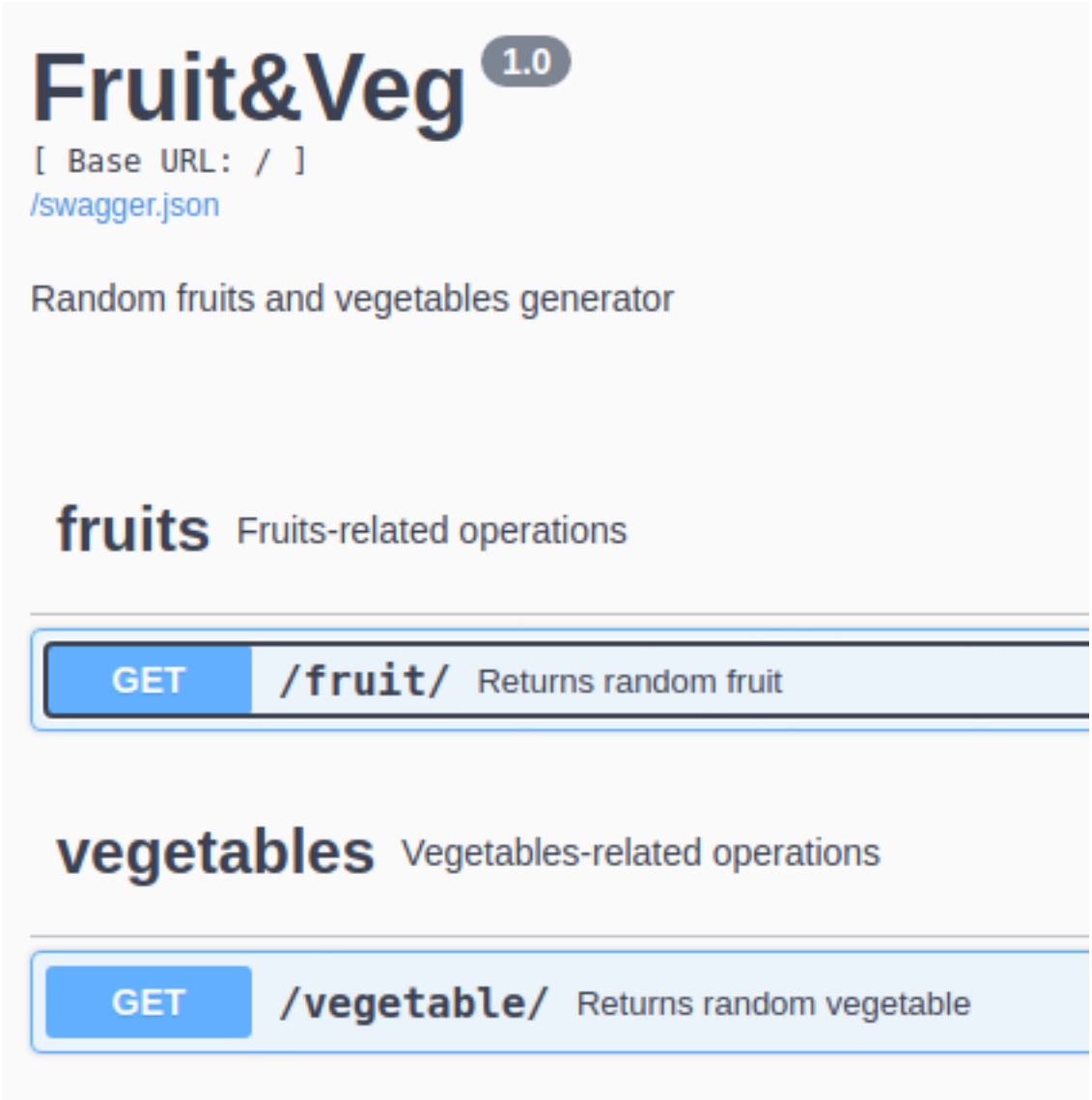
If we want to organise related functions under a common URL namespace, we can use namespaces. Here's a sample namespace definition.

```
from flask_restx import Namespace, Resource
import random

api = Namespace('vegetables', description='Vegetables-related operations')


vegs = ['Carrot', 'Egg plant']


@api.route('/')
class VegGen(Resource):
    # ...
```

Which you add to your main API definition

```
api = Api(app, version="1.0", title="Fruit&Veg", description="Random fruits and vege\
tables generator")


api.add_namespace(vegetable_api, path='/vegetable')
```

It produces the following documentation



Using with the application factory pattern.

If you are using Flask-RestX with the application factory pattern, define the API variable as follows

```
api = Api(version="1.0", title="Fruit&Veg", description="Random fruits and vegetable\
s generator")
```

Then in your`create_app` function add

```
    api.init_app(app)
```

Explicitly adding models to the documentation

If you use models in your code, it will get added to the documentation automatically. But, in case you want to explicitly add one response model to the documentation, here's how it's done

```
my_ns = Namespace(name="name of ns", validate=True)
my_ns.models[x_model.name] = x_model
```

Nested and List fields in models

In case we want to add nested fields in our response model, we use the Nested field. In case we want to define a list of items, we use the List field.

```
from flask_restx.fields import List, Nested, String

owner_model = Model("Owner",
    {
        "name": String
    }
)
door_model = Model("Door",
    {
        "color": String
    }
)
house_model = Model(
    "House",
    {
        "owner": Nested(owner_model, skip_none=True),
        "doors": List(Nested(door_model)),
    },
)
```

Returning a list of items

If we want to return a list of items we do

```
capi = Namespace('cats', description='Cats related operations')

cat = api.model('Cat', {
    'id': fields.String(required=True, description='The cat identifier'),
    'name': fields.String(required=True, description='The cat name'),
})

CATS = [
    {'id': 'felix', 'name': 'Felix'},
]

@capi.route('/')
class CatList(Resource):
    @api.doc('list_cats')
    @api.marshal_list_with(cat)
    def get(self):
        '''List all cats'''
        return CATS
```

Auth, JWT, Errors and CORS

This section covers common scenarios we encounter while using Flask-RestX.

Documenting authorizations

Please read this[82] document before proceeding. It lists authorization schemes to pass while defining the main API object.

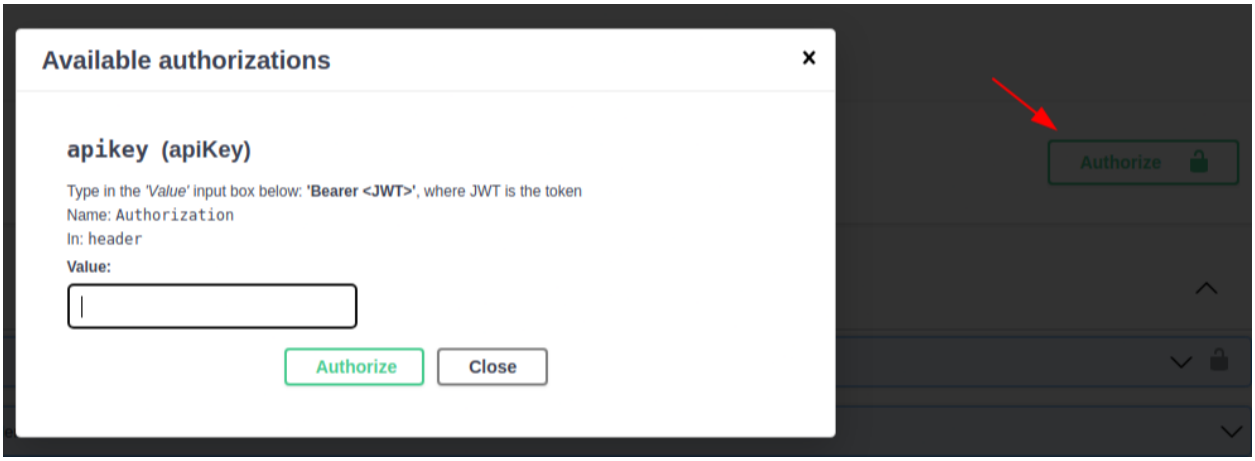
```
authorizations = {
    'apikey': {
        'type': 'apiKey',
        'in': 'header',
        'name': 'Authorization',
        'description': "Type in the *'Value'* input box below: **'Bearer &lt;JWT&gt;\
'**, where JWT is the token"
    }
}

capi = Namespace('cats', description='Cats related operations', authorizations=autho\
rizations)
```

Then to secure an endpoint in the documentation, you refer to a key defined in `authorizations`.

```
@capi.route('/')
class CatList(Resource):
    @capi.doc(security="apiKey") # here
    @capi.doc('list_cats')
    @capi.marshal_list_with(cat)
    def get(self):
        '''List all cats'''
        return CATS
```

A point of note is that Swagger docs do not authenticate the endpoint. The backend should do proper authentication. The docs will render like this

[82] https://swagger.io/docs/specification/2-0/authentication/



Cors

Flask, by default, enables a security policy that allows accepting requests coming from its origin. Accessing APIs from another origin causes refusal on the part of the server. There are three ways to deal with Cross-origin issues in Flask-RestX.

The first one is by using Flask-CORS[83]. The second one is by using Flask-RestX `cors` in the sense of `decorators=[cors.crossdomain(origin="*")]`[84]. The third way is by specifying the permission when returning the response.

```
    return {}, 200, {'Access-Control-Allow-Origin': '*'}
```

Crafting responses

You can elegantly specify the content, response code and headers using Flask's `jsonify` function

```
from flask import jsonify
from http import HTTPStatus

    def get():
        # ...
        response = jsonify(status="success", message='Item created')
        response.status_code = HTTPStatus.CREATED
        response.headers["X-Attempt"] = 3
        return response
```

`jsonify` also accepts a dictionary as parameter.

[83] https://flask-cors.readthedocs.io/en/latest/
[84] https://github.com/python-restx/flask-restx/issues/183#issue-664630621

Decouple the business logic from the endpoint

Normally we have the tendency to write the business logic inside the endpoint body.

```
@api.route('/multiply')
class Multiply(Resource):
    @api.response(200, 'Success', answer)
    @api.expect(num_parser)
    def get(self):
        '''
        Multiply 2 numbers
        '''
        args = num_parser.parse_args()
        return {'answer': args['number1']*args['number2']}
```

but, it is better to decouple the logic and the endpoint

```
def multiply_response(args):
    return {'answer': args['number1']*args['number2']}

@api.route('/multiply')
class Multiply(Resource):
    @api.response(200, 'Success', answer)
    @api.expect(num_parser)
    def get(self):
        '''
        Multiply 2 numbers
        '''
        args = num_parser.parse_args()
        return multiply_response(args)
```

This allows us to test the endpoint and the logic component separately as well as keeping our endpoint file clean.

The convenience of tracking operation success

Let's say you code a result class

```
class Result:
    def __init__(self, value=None, success=True):
        self.success = success
        self.value = value

not_found = Result({'status_code': 404, 'message': 'Item not found'}, success=False)

assert not_found.success == False
assert not_found.value['status_code'] == 404
```

You can rewrite the above decoupling as follows

```
def multiply(args):
    if args['number1'] > 500:
        return Result(value=args['number1']*args['number2'], success=True)
    else:
        return Result(success=False)

def multiply_response(args):
    operation = multiply(args)
    if operation.success:
        response = jsonify({
        'answer': operation.value
        })
        response.status_code = 200
    else:
        response = jsonify({
        'message': 'validation error'
        })
        response.status_code = 403
   return response

@api.route('/multiply')
class Multiply(Resource):
    @api.response(200, 'Success', answer)
    @api.expect(num_parser)
    def get(self):
        '''
        Multiply 2 numbers
        '''
        args = num_parser.parse_args()
        return multiply_response(args)
```

Pattern for JWT validation

An awesome pattern to check for JWT validation is to use a decorator on the function containing the business logic

```
@token_required
def multiply(args):
    # ...
```

We define it as

```
def token_required(f):
    """Execute function if request contains valid access token."""

    @wraps(f)
    def decorated(*args, **kwargs):
        token_payload = _check_access_token()
        for name, val in token_payload.items():
            setattr(decorated, name, val)
        return f(*args, **kwargs)

    return decorated

def _check_access_token():
    # if all ok
    return {
        'public_id': '',
        'token': '',
        }
    # else abort with required codes
```

The `setattr` ensures that we can use what is returned in `_check_access_token` as attributes. For example

```
@token_required
def multiply(args):
    user_public_id = multiply.public_id
```

Relationship Between Flask And Werkzeug

For experienced users, it's good to know what Flask is and how it operates. This section covers the role of Werkzeug.

What is Flask made up of?

Flask is a web framework. But, it does not maintain all the parts it uses. Flask is made up of

- Werkzeug: Provides routing and WSGI utilities

- Jinja: Templates parsing and rendering
- Click: Provides commandline utilities
- Python-dotenv: For reading config files
- Blinker: For signal[85] processing
- Cryptography: For Ad-hoc SSL certificates
- Asgiref: When using Async handlers

Flask is a wrapper around Werkzeug. It's more intuitive than using Werkzeug by itself.

Here is a snippet from the official Werkzeug tutorial[86]

```
def on_new_url(self, request):
    error = None
    url = ''
    if request.method == 'POST':
        url = request.form['url']
        if not is_valid_url(url):
            error = 'Please enter a valid URL'
        else:
            short_id = self.insert_url(url)
            return redirect(f"/{short_id}+")
    return self.render_template('new_url.html', error=error, url=url)
```

A Flask user can already tell what the code does as `request`, `redirect`, and `render_template` are used in Flask. There are also instances[87] where Werkzeug inspired itself from Flask.

[85] https://flask.palletsprojects.com/en/2.0.x/signals/
[86] https://werkzeug.palletsprojects.com/en/2.0.x/tutorial/#step-5-the-first-view
[87] https://werkzeug.palletsprojects.com/en/2.0.x/utils/#werkzeug.utils.send_file

The problem that Werkzeug sometimes poses

Since Flask uses Werkzeug internally,
breaking changes in Werkzeug always puts Flask at risk of breaking. In fact, after the most recent[88] such incidents, though Flask took care[89] of it, if we were using an older version of Flask but forgot to pin the Werkzeug version, our project was effectively broken. People tend to forget that complete dependencies version pinning is important even if Flask is the only dependency.

Flask recommends[90] using `check_password_hash` and `generate_password_hash` from `werkzeug.security`. Once again, any change in the API leaves a project broken. This is not Flask-specific and can happen to any project. But, it's good to be notified of some possible sources of failures.

[88] https://twitter.com/mitsuhiko/status/1225758711009902592
[89] https://github.com/pallets/flask/commit/17d7353d39100b5eca00f769e2564ca95457d053
[90] https://flask.palletsprojects.com/en/2.0.x/tutorial/views/#create-a-blueprint

How Context Processing Works

Flask has three contexts which it processes

- Template context
- App context
- Request context

We covered context processing for templates under the Building A Theming System lesson.

Request Context

When Flask receives a request, it pushes a request context on a stack[91] it manages. It also pushes an app context. The object is tied to a thread. It uses Werkzeug's Context Locals to ensure thread-safe global objects manipulations. Whatever context exists is destroyed when the request ends.

Before each request is processed, the before_request[92] function is called. Then after_request[93]. Independently of errors, teardown_request[94] and teardown_appcontext[95] are called.

Flask signals concerning request context

If signals_available[96] is true, the following signals are sent

- request_started[97] is sent before the `before_request()` functions are called.
- request_finished[98] is sent after the `after_request()` functions are called.
- got_request_exception[99] is sent when an exception begins to be handled, but before an errorhandler()[100] is looked up or called.
- request_tearing_down[101] is sent after the `teardown_request()` functions are called.

The signal[102] mechanism allows us to be notified when events happen. Though they are similar to event functions, they occur in unspecified order and do not modify data. Event functions can abort, for example while signal consumers only read the data.

[91] https://flask.palletsprojects.com/en/2.0.x/api/#flask._request_ctx_stack
[92] https://flask.palletsprojects.com/en/2.0.x/api/#flask.Flask.before_request
[93] https://flask.palletsprojects.com/en/2.0.x/api/#flask.Flask.after_request
[94] https://flask.palletsprojects.com/en/2.0.x/api/#flask.Flask.teardown_request
[95] https://flask.palletsprojects.com/en/2.0.x/api/#flask.Flask.teardown_appcontext
[96] https://flask.palletsprojects.com/en/2.0.x/api/#flask.signals.signals_available
[97] https://flask.palletsprojects.com/en/2.0.x/api/#flask.request_started
[98] https://flask.palletsprojects.com/en/2.0.x/api/#flask.request_finished
[99] https://flask.palletsprojects.com/en/2.0.x/api/#flask.got_request_exception
[100] https://flask.palletsprojects.com/en/2.0.x/api/#flask.Flask.errorhandler
[101] https://flask.palletsprojects.com/en/2.0.x/api/#flask.request_tearing_down
[102] https://flask.palletsprojects.com/en/2.0.x/signals/

Proxies

Proxies are objects which you can access in the same way for each worker thread. `current_app`, for example, is a proxy. To query the exact object referred to by the proxy, we use `_get_current_object()`. For example `current_app._get_current_object()`.

Application context

The Flask app object has useful associated objects like `app.config`. To access them, one has to import the app object and then use. But, using the app object within views can cause circular import errors among others. Flask pushes a context on the app stack[103]. Details of the context can be accessed through the `current_app` context. The error

```
RuntimeError: Working outside of application context.
```

Typically, this means we are accessing some of the Flask app functionalities, but we have not pushed a context to the stack. The code below, famously used, automatically pushes an app context.

```
with app.app_context():
    ...
```

Flask's g

Sharing global objects among threads is a problem that needs lots of code and understanding to tackle. Coupled with the request processing cycle, getting this right is a reasonably tedious problem. This is why Flask provides the `g` object to manage resources during a request. You can read more[104] about it.

[103] https://flask.palletsprojects.com/en/2.0.x/api/#flask._app_ctx_stack
[104] https://flask.palletsprojects.com/en/2.0.x/appcontext/#storing-data

Deploying To Shared Hosting

We don't need to buy a Virtual Private Server to host our Python apps. We can use a normal CPanel-based hosting which costs very cheap. If our app does not need custom software running like redis etc, this might be a sensible option.

What to look for?

In order to deploy your app on shared CPanel-based hosting, you must ensure that

- The CPanel offer has support for Python apps
- You have terminal access from CPanel

These two are crucial to get started. Git comes pre-installed normally. We use the terminal to update our app.

How to proceed?

Go to CPanel

Choose the terminal icon.

Navigate to the folder you will clone your project (typically at domain.com folder or subdomain.domain.com folder). See File manager for an overview.

Then git clone your project.

Now on control panel open `Setup Python app`

Set the following values

Python version: 3.7 works

Application root: domainfolder/myproject_folder

Application URL: choose subdomain or adjust as needed

Application startup file: wsgi.py

Application Entry point: application

Now it will override your wsgi.py, edit the file using file manager and add the following

```
import os
import sys

sys.path.append(os.getcwd())
from app import create_app

# on shell do pwd to get a path like this:'/home2/folder/myproject_folder' set path \
to this
path = ""
if path not in sys.path:
    sys.path.insert(0, path)

application = create_app("production")
```

Set `path = ''` in wsgi.py to what you get when on terminal you navigate to folder/shopyo/shopyo and type pwd It needs the absolute file path.

You also get the path when editing wsgi.py, add everything except the last paer, that is /wsgi.py

Now initialise app on the Python app module. On the python app page it will give you an instruction to copy to activate virtual env

Paste in terminal and press enter

```
python3 -m pip install --upgrade pip

python3 -m pip install -r requirements.txt

export FLASK_APP=app.py
```

Go to setup python app then restart app.

Go to your URL.

New In Flask 2.0

Flask 2.0 introduced many features which were in the oven for a long time. It's a nice illustration of the reaction of maintainers in response to evolving standards.

Loading any type of configuration file

You can specify the file name and loader .

```
import toml

from flask import Flask

app = Flask('__name__')
app.config.from_file('config.toml', toml.load)
```

with content

```
DEBUG = true
SECRET_KEY = "development key"
```

or JSON

```
import json

from flask import Flask

app = Flask('__name__')
app.config.from_file('config.json', json.load)
```

with content

```
{
    "DEBUG": true,
    "SECRET_KEY": "development key"
}
```

Nested blueprints

```
fruit = Blueprint('fruit', __name__, url_prefix='/fruit')
apple = Blueprint('apple', __name__, url_prefix='/apple')
banana = Blueprint('banana', __name__, url_prefix='/apple')
fruit.register_blueprint(apple)
fruit.register_blueprint(banana)
app.register_blueprint(fruit)
```

Named methods

The `get`, `post`, `patch`, `put` and `delete` request types have their named methods. This

```
@app.get('/'):
def x():
    # ...
```

is an alternative to this

```
@app.route('/', methods=['GET']):
def x():
    # ...
```

Async requests

Flask supports using asynchronous constructs inside of views

```
@app.route('/')
async def hello():
    return await my_async_func()
```

It is to be noted that Flask still uses WSGI, the handling of asynchronicity is limited.

Quart[105] and aioflask[106] are attempts at introducing ASGI compatibility in Flask. Both are by Pallets maintainers.

[105] https://gitlab.com/pgjones/quart
[106] https://github.com/miguelgrinberg/aioflask

Flask Community Workgroup

Flask currently needs community support to maintain plugins more than ever. Here is an extract as to why the Flask Community Workgroup[107] was set up.

About the need for a community workgroup

Flask is a great library with great people and great extensions, but it can be even greater with a structured approach to the community aspect. Being around for 10 years and yet, still maintaining relevance is a feat of it's own. It's simple to get started with all while being a battle-proven framework. We also notice that it's a smooth flow from a proof of concept to production. It also influences other frameworks upto this day. Technically it is sound, no doubt about it.

However, there are many Flask-related issues that deserve some attention. The fix to those issues lies not in mere merges or code fixes. One such example is the plugins/extensions ecosystem. A recurrent and commonly observed pattern is that a project rises in popularity and becomes the de-facto reference for a specific task then the maintainer goes offline, cut off from the project. Sometimes other maintainers themselves have no merge right, leaving the project as good as dead.

Due to many factors, the projects soon become obsolete, less efficient or outright broken. But people still use them sometimes unknowingly exposing their applications to vulnerabilities. One help point from the WG is keep an eye on extensions and prevent those kinds of failures. One concrete initiative is to set up an organization (github) to care about extensions. It will have a minimum required number of developers who are willing to look after those extensions. This can also further the Flask education cause by having mentoring, much like Google Code-In ... .

About the Pallets-eco

The Pallets-eco[108] Github organisation was set up to help maintain important but abandoned plugins. You can find the criteria including plugins here[109].

[107] https://flaskcwg.github.io/
[108] https://github.com/pallets-eco
[109] https://flaskcwg.github.io/pallets-eco/

Shopyo: The Framework Behind The Course

I maintain shopyo[110] which is a framework built on top of Flask to provide a neat development experience. Though created for personal use, it now has some nice download metrics[111].

I talked about it at several conferences like EuroPython, Conf42 etc. The framework is the most advanced attempt at a Flask base. It emulates many Django features and introduces concepts of it's own. It uses many packages covered under the plugins section. It is designed for use in building big Flask apps.

The project is the result of years of freelancing. OpenSource helped to improve it. One of the core developers now work for Amazon. The combined effects of the above enabled me to share my knowledge by writing this course. Hope you enjoyed it. You can write tesimonials and feedback here[112] or contact me on my mail arj.python@gmail.com. Thank you!

[110] https://github.com/shopyo/shopyo
[111] https://pepy.tech/project/shopyo
[112] https://github.com/FlaskMasterClass/feedback

Other Useful Extensions

Some extensions have specific use-cases, which are good to know. Over the past years, they achieved a high maintenance index, and the Flask community respects them. We'll cover some of these!

Flask-Limiter

One challenge of API development is knowing how to assign quotas to consumers. We need to limit the number of requests per time unit to preserve our bandwidth for other users. Flask-Limiter[113] allows us to rate-limit our endpoints. It also has an intuitive API.

```
app = Flask(__name__)
limiter = Limiter(
    app,
    key_func=get_remote_address,
    default_limits=["200 per day", "50 per hour"]
)

@app.route("/slow")
@limiter.limit("1 per day")
def slow():
    return ":("
```

It integrates greatly with Redis and Memcached.

Flask-Babel

Internationalization dramatically improves the accessibility of a website. Having pages in different languages allows us to reach a bigger audience. Flask-Babel[114] integrates the babel library with Flask. We can specify translations using `.po` files.

Flask-Multipass

Authentication schemes can be a hassle to handle and set up correctly. Flask-Multipass[115] allows us to simultaneously have LDAP (Lightweight Directory Access Protocol), OAuth, and similar schemes for our apps.

[113] https://flask-limiter.readthedocs.io/en/stable/
[114] https://python-babel.github.io/flask-babel/
[115] https://flask-multipass.readthedocs.io/en/latest/

Flask-Appbuilder

Building a fully-featured site requires us to repeatedly go through the same steps, laying out the structure or setting up access control. Flask-AppBuilder[116] is one of the best Flask bases ever. By default, it provides authentication support, database plugins, internationalization, and charts. Apache Airflow, a popular data pipeline orchaestration tool uses it.

Flask-SocketIO

For real-time communication, socketIO is a great library to use. Flask-SocketIO[117] gives Flask applications access to low latency bi-directional communications between the clients and the server. The crucial point is getting[118] the corresponding JavaScript library and Flask-SocketIO versions right.

[116] https://flask-appbuilder.readthedocs.io/en/latest/
[117] https://flask-socketio.readthedocs.io/en/latest/
[118] https://flask-socketio.readthedocs.io/en/latest/intro.html#version-compatibility

Further Readings

Here are some further readings that may come in handy at one time or the other

Topic	Explanation
Flask and React Deployment on NginX[119]	Deploying a front-end and a back-end on the same VPS is a useful pattern
Optimise Service Performance and Reduce Costs With Quart[120]	How an ASGI-compatible version of Flask can make a drastic difference
Multi-tenancy with Flask and SQLAlchemy[121]	Another performance improvement pattern

[119] https://blog.miguelgrinberg.com/post/how-to-deploy-a-react--flask-project

[120] https://medium.com/snaptravel/how-we-optimized-service-performance-using-the-python-quart-asgi-framework-and-reduced-costs-by-1362dc365a0

[121] https://hamza-senhajirhazi.medium.com/how-to-handle-schema-multi-tennancy-with-python-flask-sqlalchemy-postgres-7000dda10749