Build Your Python Flask Application

Bhavani Ravi Follow

Mar 17, 2019 · 10 min read

Step-by-Step tutorial to build your first web application with Python.

Why this tutorial?

I constantly hold my urge to write how-tos on basic 101 topics, since I strongly feel that there is already enough content on the internet.

However, after seeing people reading through tutorials after tutorials, watching video after videos, I've noticed that they still don't feel that *"They are good enough" or "They know enough"* to build an industry-level application. These courses walk you through a structured environment where there is a lesser possibility to make mistakes. Hence when placed in a real-time environment to build their own application you often don't know where to start.

In this blog, I am going to take you through a step-by-step guided tutorial on ***Building a ToDo List App using Flask***. We are not just going to code together but also think through and solve problems we come across.

But if you already know Flask and just want to take a look at the code. Here you go.

Non-Technical Requirements

1. Your Time

Now that you have landed on this blog, you need some time to finish it to the end. If you don't have time right away, bookmark the blog and come back when you have some. Trust me reading/scanning through is not going to help.

2. Laptop/Computer — Coding on your phone is a complete No.

3. A quiet place — Not just for this blog; learning anything needs your complete focus. So make sure you are in a place with fewer distractions.

Why Python?

Python, the most favorite language amongst beginners, is known for its simplicity and its clean and concise way of implementing things. Despite the hype around machine learning, Python has done enough to conquer the web application domain as well.

It is one of the most preferred languages to build backend applications. Even when facing good old PHP and full stack Javascript, Python has its place because of the ease of implementation, and a supportive community that keeps beginners hooked.

Why Flask?

Just like how people get confused with which Python version to start with, Django vs Flask is another age-old debate. Though people say Flask is simple and easy to get started with while Django is heavy for building web applications, there is another important reason why you should choose Flask over Django.

We, as developers in the era of the cloud, are moving away from monolithic applications. With microservices architecture in place, running multiple Django servers is going to make your services heavy because of all the built-in apps it comes with.

Requirements & Setup

1. Of course, we need Python — 3.7 would be great

2. Pycharm community IDE because it's awesome & free

3. Once you have the IDE, create a project. I named mine `todo-flask`. *How creative!!*

4. virtualenv — if you're using Pycharm you don't have to worry about this.

5. Installing flask using `pip install flask`

If you know the basics of Flask and wanna jump right in and build the app. jump here.

Hello World

Let's just check if our setup is fine by writing a "hello world" program. Create an `app.py` file under your project. You can name it anything other than calling your file `flask.py` since it will create a conflict with the actual flask package we installed.

```
todo-flask
    |_ app.py
```

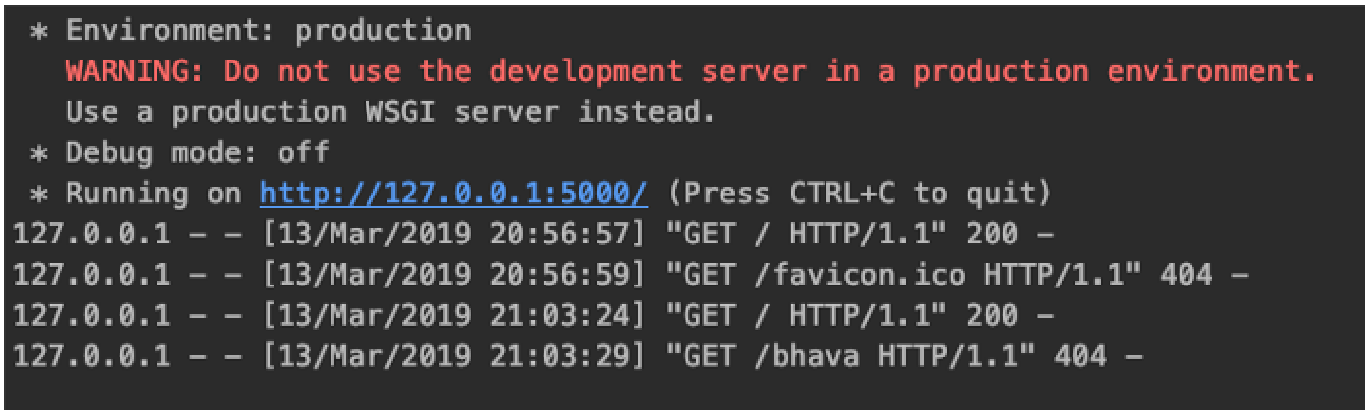
The code. Read through each line as it comes with an explanation of what each line means

```
# app.py

from flask import Flask                  # import flask
app = Flask(__name__)                    # create an app instance

@app.route("/")                          # at the end point /
def hello():                             # call method hello
    return "Hello World!"                # which returns "hello world"

if __name__ == "__main__":               # on running python app.py
    app.run()                            # run the flask app
```

Running the Application

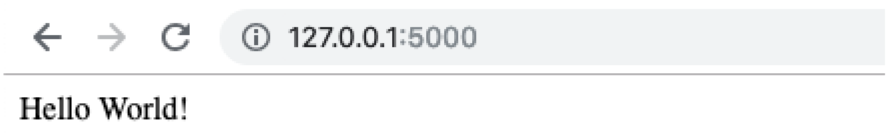
Run the application by running the `app.py` file. By default, flask runs a local server at port 5000.

```
python app.py
```



```
 * Environment: production
   WARNING: Do not use the development server in a production environment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
127.0.0.1 - - [13/Mar/2019 20:56:57] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [13/Mar/2019 20:56:59] "GET /favicon.ico HTTP/1.1" 404 -
127.0.0.1 - - [13/Mar/2019 21:03:24] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [13/Mar/2019 21:03:29] "GET /bhava HTTP/1.1" 404 -
```

On hitting the URL from your browser you will see your first hello world program alive.



Hello World!

Hello With Your Name

```
@app.route("/<name>")                 # at the end point /<name>
def hello_name(name):                 # call method hello_name
    return "Hello "+ name             # which returns "hello + name
```

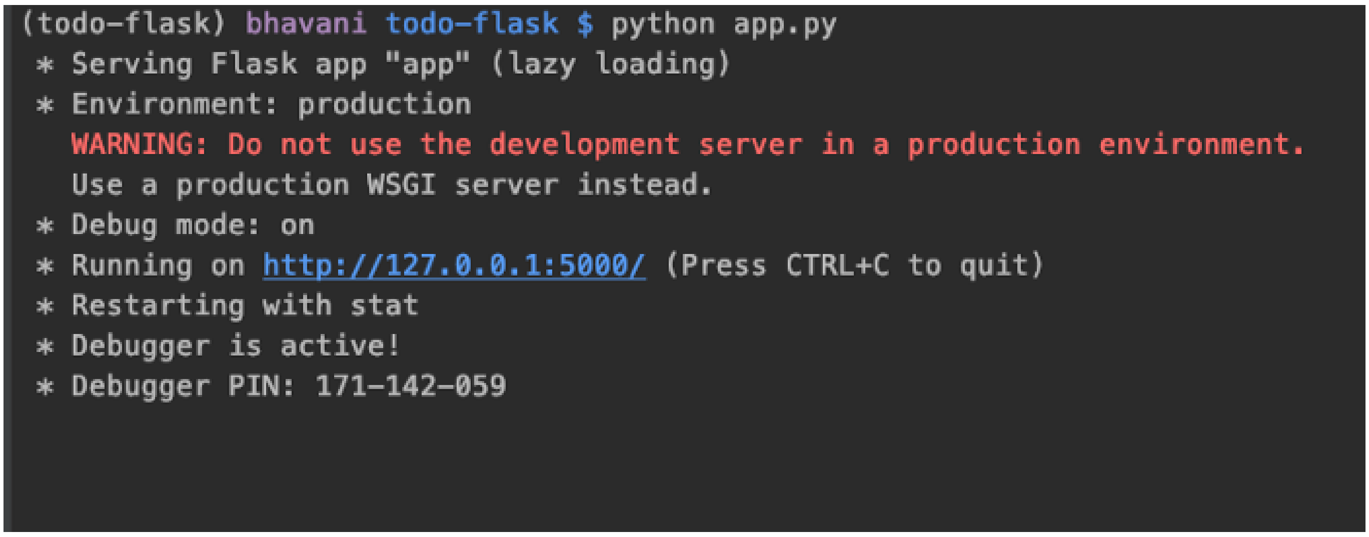
Let’s hit `http://localhost:5000/bhava.` we will get a 404 error.

Debug Mode

This is because we are running the server in a production mode. For development purposes, we use something called as `debug` mode. When `debug=True` is set the server restarts as we add new code to our Flask Application. In order to set the debug mode do the following

1. Modify the line `app.run()` to `app.run(debug=True).`
2. Stop the running server and restart it again.

You will see “debugger is active” which means that the debug mode has taken effect. Now you can go on and edit your file as much as you want, and the server will be restarted.



```
(todo-flask) bhavani todo-flask $ python app.py
 * Serving Flask app "app" (lazy loading)
 * Environment: production
   WARNING: Do not use the development server in a production environment.
   Use a production WSGI server instead.
 * Debug mode: on
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 171-142-059
```

Let’s Build that ToDo list Application

Now that you have the setup ready and working, let’s jump right in and start building our application. You may be pumped to dive right in and start coding. Me too. Let’s control that urge together for some time and do some planning.

What can the users do?

Let’s say you are building this application and giving it to users. What are the operations you expect them to do?

1. Create an item
2. Delete an item
3. Mark an item done
4. Update an item

These are basic functionalities that make it a ToDo app. But you can go fancy on the features and provide

1. Categorizing ToDo items into work, personal etc.,
2. Tagging items
3. Prioritizing items
4. Remainder when an item is not done in the estimated time

How does the data look?

Now that we have figured out the actions that need to be performed let's look at how to data is going to look like pertaining to each action

1. **Create an Item —** Title, Description, CreatedOn, DueDate
2. **Delete an Item** — _is_deleted

Now we gotta decide whether you want to delete the record permanently or just set a flag saying it is deleted. It is called a soft delete. Soft Delete is a common practice in customer-facing systems.

3. Mark an Item Done — _is_done

4. **Update an Item** — We are not editing any info other than the ones we created. So no additional parameters here.

Designing the Schema

For our database, we are going to use the SQLite relational DB which means we are going to store our data in the form of tables.

We need a table to store the ToDo list created by users, which makes our first table with columns.

```
ToDo_Items
- Id                    Primary Key
— Title                 Text
```

```
- Description         Text
- CreatedOn           Date
- DueDate             Date
- _is_deleted         Boolean
- _is_done            Boolean
```

Now, this looks good to go if we are designing a system for just one person, but we are not. We are going to give it to a lot of people, which means we need to have a list of all the users who signed up into the system and maintain a reference to the user associated with each ToDo item in our ToDo items table.

```
User
- Name             Text
- Email            Email
- Id               Primary Key

ToDo_Items
   ...
   ...
   CreatedBy       ForeginKey(User)
```

Structuring Our Code

One of the common mistakes that beginners do is dumping all the code into one file. While this is completely acceptable since you are a beginner, it is always good to have a sense of best practices and why we do them. This would give you a sense of building real software rather than building a quick hack.

Now for our application, we are going to split the code into 3 parts

1. app.py — the entry & exit point to our application
2. service.py — converts the request into a response.
3. models.py — handles everything that involves a Database.

Why do we need 3 separate files?

Standards — One of the important things that is missed in a beginner tutorial is the standards or patterns that need to be followed. By separating the app into 3 different files we are separating the business logic(service.py), the application layer(app.py), and the data(models.py). In technical terms, this is called an **MVC — Model View Controller Pattern.**

Code Maintainability —Let’s say years down the line you decide MongoDB is better than Sqlite and you are determined to change it. In a single file application, you will be fiddling with almost every line of the file. But with our MVC model in place, all you need to do is to dump the models.py file and rewrite it using a bunch of MongoDB queries.

> Want to learn more about such Industry standards and build your own app from scratch? Join the Online Bootcamp

Let's Get Our Hands Dirty

From here on we are going to jump right on to our code and learn as we move along. Before we move on let's go through a quick primer of SQLite.

```
# 1.import sqlite
import sqlite

# 2. create a connection to DB
conn = sqlite3.connect('todo.db')

# 3. Write your sql query
query = "<SQLite Query goes here>"

# 4. execute the query
result = conn.execute(query)
```

Throughout the `models.py` file you see this pattern repeating for creating, deleting, updating items in our todo list.

Creating Tables

Let's keep aside Flask for now and look at the Pythonic way of creating these tables. To handle all the DB related operations we are going to create a separate file called `models.py`

It consists of two parts

1. **Schema — Where are the *DB Tables* are created and maintained.**

```
class Schema:
    def __init__(self):
        self.conn = sqlite3.connect('todo.db')
        self.create_user_table()
        self.create_to_do_table()
        # Why are we calling user table before to_do table
        # what happens if we swap them?

    def create_to_do_table(self):
        query = """
        CREATE TABLE IF NOT EXISTS "Todo" (
          id INTEGER PRIMARY KEY,
          Title TEXT,
          Description TEXT,
          _is_done boolean,
          _is_deleted boolean,
          CreatedOn Date DEFAULT CURRENT_DATE,
          DueDate Date,
          UserId INTEGER FOREIGNKEY REFERENCES User(_id)
        );
        """

        self.conn.execute(query)

    def create_user_table(self):
        # create user table in similar fashion
        # come on give it a try it's okay if you make mistakes
        pass
```

Now, these tables have to be created when you start the application which means it is going to be triggered before our `app.run` command.

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

2. Todo — Operations related to ToDo table

The lines in the ToDoModel are kept this way for the tutorial purposes. In future blogs, we will be switching to an ORMs where you don't have to write SQL Queries at all. Not using an ORM may result in SQLInjections.

You can think of models as a binding that associates DB tables with associated functions. Hence we create a class with 4 methods `Create, update, delete and list` which correspond to 4 common SQL queries `CREATE, UPDATE, DELETE and SELECT`

```
class ToDoModel:
    TABLENAME = "TODO"

    def __init__(self):
        self.conn = sqlite3.connect('todo.db')

    def create(self, text, description):
        query = f'insert into {TABLENAME} ' \
                f'(Title, Description) ' \
                f'values ("{text}","{description}")'
        result = self.conn.execute(query)

        # return result
        # uncomment above line and check what error occurs and why

        return result.lastrowid
    # Similarly add functions to select, delete and update todo
```

The Service Methods

We are separating view methods from service method because it enables you to test these functions easily.

```
# Service.py
class ToDoService:
    def __init__(self):
        self.model = ToDoModel()

    def create(self, params):
        return self.model.create(params["Title"],
params["Description"])
```

The View Functions

The view functions as I already mentioned is the entry and exit point of the system. The kind of decisions that we take in this file include

1. Type of input we expect from the user — JSON, File, free text etc.,
2. Type of output we are providing the user — JSON, file, HTML page etc.,
3. Authentication of the requests
4. Logging the requests

```
# app.py

@app.route("/todo", methods=["POST"])
def create_todo():
    return ToDoService().create(request.get_json())
```

Final Magic

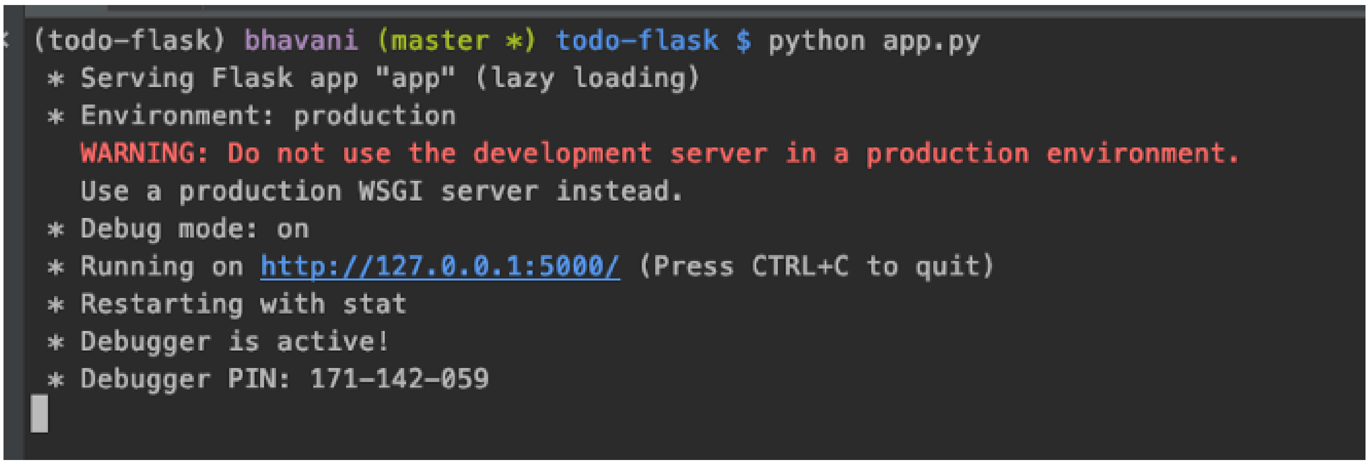
Now we have all our functions in place talking to each other. All we got to do is hit the API endpoint, create some ToDo Items and see it all working in action.

Throughout this blog, I have never talked about how are we going to build an interface for people to enter the Todo items and where they mark them done etc., As I said, in the beginning, we are learning not just Flask but also the practices followed in the industry.

Most of the software out there works with completely separated backend and frontend. What we have created is a backend system.

To see it working we can build a front-end, which we will do with React in the next blog. But for now, I am going to show you how you can consume it with Python or Postman.

1. Let your app.py be running in one console. Now as you see it is running in my localhost at port 5000.



```
(todo-flask) bhavani (master *) todo-flask $ python app.py
 * Serving Flask app "app" (lazy loading)
 * Environment: production
   WARNING: Do not use the development server in a production environment.
   Use a production WSGI server instead.
 * Debug mode: on
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 171-142-059
```

2. Let's create our first todo. Open another terminal. I use `request` python package to consume APIs with Python. You can also use postman

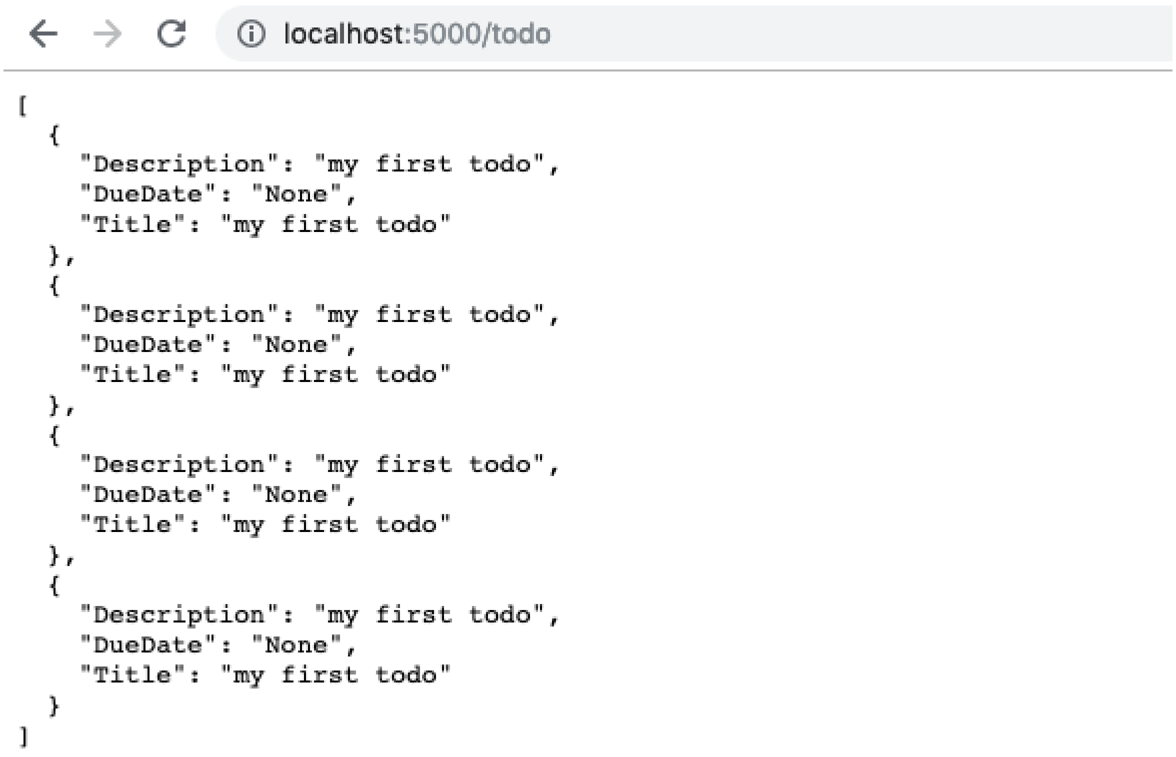
```
pip install requests
```

3. Open python shell and hit the API

```
$ python
>>> import requests
>>> requests.post("http://localhost:5000/todo",
                  json={"Title":"my first todo",
                        "Description":"my first todo"})
<Response [200]>
```

`requests.post` calls the post API we just created with the data required to create a ToDo Item. We get a response 200 which means it is successfully created.

I have run the code 4 times and on hitting the `/todo` API with GET method I get the following response.



```
[
  {
    "Description": "my first todo",
    "DueDate": "None",
    "Title": "my first todo"
  },
  {
    "Description": "my first todo",
    "DueDate": "None",
    "Title": "my first todo"
  },
  {
    "Description": "my first todo",
    "DueDate": "None",
    "Title": "my first todo"
  },
  {
    "Description": "my first todo",
    "DueDate": "None",
    "Title": "my first todo"
  }
]
```

. . .

If you have gotten this far into the blog give yourself a pat on the back because guess what? You're awesome. This is just the starting point. The whole working repository is available on GitHub.

One last thing, How about we build your app from scratch just like this with guidance and code review, check this out.

. . .

Did you like what you read?

Support my content to make it free for everyone

Thanks to Krithika.

Python Flask Web Development Rest Programming