The Flask Mega-Tutorial, Part I: Hello, World!

Posted by Miguel Grinberg *on* December 3, 2023 *under* Flask Programming Python

The Flask Mega-Tutorial Introduction

You are reading the 2024 edition of the Flask Mega-Tutorial. The complete course is also available to order in e-book and paperback formats from Amazon. Thank you for your support!

If you are looking for the 2018 edition of this course, you can find it here.

For your reference, here is the complete list of articles in this series:

- Chapter 1: Hello, World! (this article)
- Chapter 2: Templates
- Chapter 3: Web Forms
- Chapter 4: Database
- Chapter 5: User Logins
- Chapter 6: Profile Page and Avatars
- Chapter 7: Error Handling
- Chapter 8: Followers
- Chapter 9: Pagination
- Chapter 10: Email Support
- Chapter 11: Facelift

Welcome! You are about to start on a journey to learn how to create web applications with Python and the Flask framework. In this first chapter, you are going to learn how to set up a Flask project. By the end of this chapter you are going to have a simple Flask web application running on your computer!

All the code examples presented in this book are hosted on a GitHub repository. Downloading the code from GitHub can save you a lot of typing, but I strongly recommend that you type the code yourself, at least for the first few chapters. Once you become more familiar with Flask and the example application you can access the code directly from GitHub if the typing becomes too tedious.

At the beginning of each chapter, I'm going to give you three GitHub links that can be useful while you work through the chapter. The **Browse** link will open the GitHub repository for Microblog at the place where the changes for the chapter you are reading were added, without including any changes introduced in future chapters. The **Zip** link is a download link for a zip file including the entire application up to and including the changes in the chapter. The **Diff** link will open a graphical view of all the changes that were made in the chapter you are about to read.

The GitHub links for this chapter are: Browse, Zip, Diff.

Installing Python

If you don't have Python installed on your computer, go ahead and install it now. If your operating system does not provide you with a Python package, you can download an installer from the Python official website.

If you are using Microsoft Windows along with WSL or Cygwin, note that you will not be using the Windows native version of Python, but a UNIX-friendly version that you need to obtain from Ubuntu (if you are using WSL) or from Cygwin.

To make sure your Python installation is functional, you can open a terminal window and type `python3`, or if that does not work, just `python`. Here is what you should expect to see:

```
$ python3
Python 3.12.0 (main, Oct  5 2023, 10:46:39) [GCC 11.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> _
```

The Python interpreter is now waiting at an interactive prompt, where you can enter Python statements. In future chapters you will learn what kinds of things this interactive prompt is useful for. But for now, you have confirmed that Python is installed on your system. To exit the interactive prompt, you can type `exit()` and press Enter. On the Linux and Mac OS X versions of Python you can also exit the interpreter by pressing Ctrl-D. On Windows, the exit shortcut is Ctrl-Z followed by Enter.

Installing Flask

The next step is to install Flask, but before I go into that I want to tell you about the best practices associated with installing Python *packages*.

In Python, packages such as Flask are available in a public repository, from where anybody can download them and install them. The official Python package repository is called PyPI, which stands for Python Package Index (some people also refer to this repository as the "cheese shop"). Installing a package from PyPI is very simple, because Python comes with a tool called `pip` that does this work.

To install a package on your machine, you use `pip` as follows:

```
$ pip install <package-name>
```

Interestingly, this method of installing packages will not work in most cases. If your Python interpreter was installed globally for all the users of your computer, chances are your regular user account is not going to

have permission to make modifications to it, so the only way to make the command above work is to run it from an administrator account. But even without that complication, consider what happens when you install a package in this way. The `pip` tool is going to download the package from PyPI, and then add it to your Python installation. From that point on, every Python script that you have on your system will have access to this package. Imagine a situation where you have completed a web application using version 2 of Flask, which was the most current version of Flask when you started, but now it has been superseded by version 3. You now want to start a second application, for which you'd like to use version 3, but if you upgrade the version 1 that you have installed you risk breaking your older application. Do you see the problem? It would be ideal if it was possible to have Flask version 2 installed and accessible to your old application, while also install Flask version 3 for your new one.

To address the issue of maintaining different versions of packages for different applications, Python uses the concept of *virtual environments*. A virtual environment is a complete copy of the Python interpreter. When you install packages in a virtual environment, the system-wide Python interpreter is not affected, only the copy is. So the solution to have complete freedom to install any versions of your packages for each application is to use a different virtual environment for each application. Virtual environments have the added benefit that they are owned by the user who creates them, so they do not require an administrator account.

Let's start by creating a directory where the project will live. I'm going to call this directory *microblog*, since that is the name of the application:

```
$ mkdir microblog
$ cd microblog
```

Support for virtual environments is included in all recent versions of Python, so all you need to do to create one is this:

```
$ python3 -m venv venv
```

With this command, I'm asking Python to run the `venv` package, which creates a virtual environment named `venv`. The first `venv` in the command is an argument to the `-m` option which is the name of the Python virtual environment package, and the second is the virtual environment name that I'm going to use for this particular environment.

If you find this confusing, you can replace the second `venv` with a different name that you want to assign to your virtual environment. In general, I create my virtual environments with the name `venv` in the project directory, so whenever I `cd` into a project I find its corresponding virtual environment.

Note that in some operating systems you may need to use `python` instead of `python3` in the command above. Some installations use `python` for Python 2.x releases and `python3` for the 3.x releases, while others map `python` to the 3.x releases and do not have a `python3` command at all.

After the command completes, you are going to have a directory named *venv* where the virtual environment files are stored.

Now you have to tell the system that you want to use this virtual environment, and you do that by *activating* it. To activate your brand new virtual environment you use the following command:

```
$ source venv/bin/activate
(venv) $ _
```

If you are using a Microsoft Windows command prompt window, the activation command is slightly different:

```
$ venv\Scripts\activate
(venv) $ _
```

If you are on Windows but are using PowerShell instead of the command prompt, then there is yet another activation command you should use:

```
$ venv\Scripts\Activate.ps1
(venv) $ _
```

When you activate a virtual environment, the configuration of your terminal session is modified so that the Python interpreter stored inside it is the one that is invoked when you type `python`. Also, the terminal prompt is modified to include the name of the activated virtual environment. The changes made to your terminal session are all temporary and private to that session, so they will not persist when you close the terminal window. If you work with multiple terminal windows

open at the same time, it is perfectly fine to have different virtual environments activated on each one.

Now that you have a virtual environment created and activated, you can finally install Flask in it:

```
(venv) $ pip install flask
```

If you want to confirm that your virtual environment now has Flask installed, you can start the Python interpreter and *import* Flask into it:

```
>>> import flask
>>> _
```

If this statement does not give you any errors you can congratulate yourself, as Flask is installed and ready to be used.

Note that the above installation commands do not specify which version of Flask you want to install. The default when no version is specified is to install the latest version available in the package repository. This tutorial is designed for version 3 of Flask, but should also work with version 2. The above command will install the latest 3.x version, which should be appropriate for most users. If for any reason you prefer to follow this tutorial on a 2.x release of Flask, you can use the following command to install the latest 1.x version:

```
(venv) $ pip install "flask<3" "werkzeug<3"
```

A "Hello, World" Flask Application

If you go to the [Flask's quick start page](#), you are welcomed with a very simple example application that has just five lines of code. Instead of repeating that trivial example, I'm going to show you a slightly more elaborate one that will give you a good base structure for writing larger applications.

The application will exist in a *package*. In Python, a subdirectory that includes a *__init__.py* file is considered a package, and can be imported. When you import a package, the *__init__.py* executes and defines what symbols the package exposes to the outside world.

Let's create a package called `app`, that will host the application. Make sure you are in the *microblog* directory and then run the following command:

```
(venv) $ mkdir app
```

The *__init__.py* for the `app` package is going to contain the following code:

app/__init__.py: Flask application instance

```
from flask import Flask

app = Flask(__name__)

from app import routes
```

The script above creates the application object as an instance of class `Flask` imported from the flask package. The `__name__` variable passed to the `Flask` class is a Python predefined variable, which is set to the name of the module in which it is used. Flask uses the location of the module passed here as a starting point when it needs to load associated resources such as template files, which I will cover in Chapter 2. For all practical purposes, passing `__name__` is almost always going to configure Flask in the correct way. The application then imports the `routes` module, which doesn't exist yet.

One aspect that may seem confusing at first is that there are two entities named `app`. The `app` package is defined by the *app* directory and the *__init__.py* script, and is referenced in the `from app import routes` statement. The `app` variable is defined as an instance of class `Flask` in the *__init__.py* script, which makes it a member of the `app` package.

Another peculiarity is that the `routes` module is imported at the bottom and not at the top of the script as it is always done. The bottom import is a well known workaround that avoids *circular imports*, a common problem with Flask applications. You are going to see that the `routes` module needs to import the `app` variable defined in this script, so putting one of the reciprocal imports at the bottom avoids the error that results from the mutual references between these two files.

So what goes in the `routes` module? The routes handle the different URLs that the application supports. In Flask, handlers for the application routes are written as Python functions, called *view functions*. View functions are mapped to one or more route URLs so that Flask knows what logic to execute when a client requests a given URL.

Here is the first view function for this application, which you need to write in a new module named *app/routes.py*:

---

app/routes.py: Home page route

```python
from app import app

@app.route('/')
@app.route('/index')
def index():
    return "Hello, World!"
```

This view function is actually pretty short, it just returns a greeting as a string. The two strange `@app.route` lines above the function are *decorators*, a unique feature of the Python language. A decorator modifies the function that follows it. A common pattern with decorators is to use them to register functions as callbacks for certain events. In this case, the `@app.route` decorator creates an association between the URL given as an argument and the function. In this example there are two decorators, which associate the URLs `/` and `/index` to this function. This means that when a web browser requests either of these two URLs, Flask is going to invoke this function and pass its return value back to the browser as a response. If this does not make complete sense yet, it will shortly, when you run this application.

To complete the application, you need to have a Python script at the top-level that defines the Flask application instance. Let's call this script *microblog.py*, and define it as a single line that imports the application instance:

---

microblog.py: Main application module

```python
from app import app
```

Remember the two `app` entities? Here you can see both together in the same sentence. The Flask application instance is called `app` and is a member of the `app` package. The `from app import app` statement

imports the `app` variable that is a member of the `app` package. If you find this confusing, you can rename either the package or the variable to something else.

Just to make sure that you are doing everything correctly, below you can see a diagram of the project structure so far:

```
microblog/
  venv/
  app/
    __init__.py
    routes.py
  microblog.py
```

Believe it or not, this first version of the application is now complete! Before running it, though, Flask needs to be told how to import it, by setting the `FLASK_APP` environment variable:

```
(venv) $ export FLASK_APP=microblog.py
```

If you are using the Microsoft Windows command prompt, use `set` instead of `export` in the command above.

Are you ready to be blown away? You can run your first web application by typing the command `flask run`, as shown below:

```
(venv) $ flask run
 * Serving Flask app 'microblog.py' (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a productio
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

What happened here? The `flask run` command will look for a Flask application instance in the module referenced by the `FLASK_APP` environment variable, which in this case is *microblog.py*. The command sets up a web server that is configured to forward requests to this application.

After the server initializes it will wait for client connections. The output from `flask run` indicates that the server is running on IP address

127.0.0.1, which is always the address of your own computer. This address is so common that is also has a simpler name that you may have seen before: *localhost*. Network servers listen for connections on a specific port number. Applications deployed on production web servers typically listen on port 443, or sometimes 80 if they do not implement encryption, but access to these ports requires administration rights. Since this application is running in a development environment, Flask uses port 5000. Now open up your web browser and enter the following URL in the address field:

```
http://localhost:5000/
```

Alternatively you can use this other URL:

```
http://localhost:5000/index
```

Do you see the application route mappings in action? The first URL maps to `/`, while the second maps to `/index`. Both routes are associated with the only view function in the application, so they produce the same output, which is the string that the function returns. If you enter any other URL you will get an error, since only these two URLs are recognized by the application.
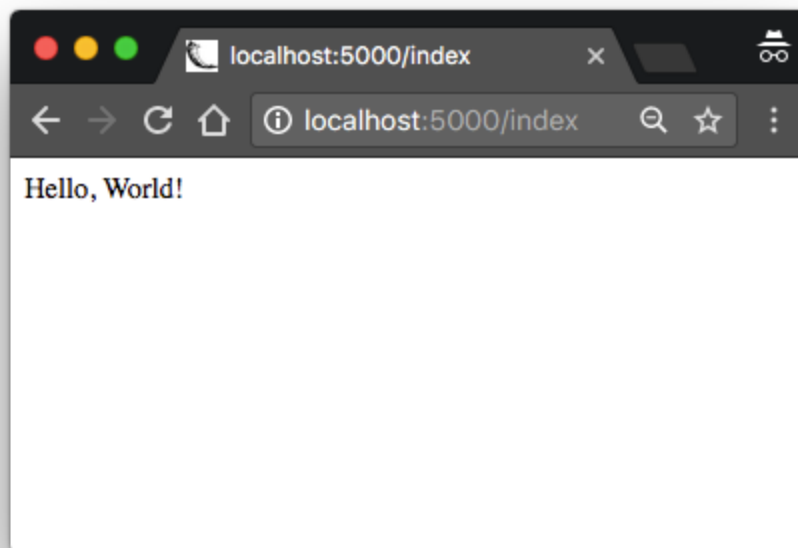
When you are done playing with the server you can just press Ctrl-C to stop it.

Congratulations, you have completed the first big step to become a web developer!

Did you have trouble running the Flask application? In most computers port 5000 is available, but there is a possibility that your computer is already running an application that uses this port, in which case the `flask run` command will fail with an "address already in use" or similar error. If you use a Macintosh computer, some versions of macOS run a service called "Airplay Receiver" on this port. If you are unable to figure out how to remove the software that uses port 5000, you can try running Flask on a different port. For example, here is how to start the server on port 5001:

```
(venv) $ flask run --port 5001
```

Before I end this chapter, I will show you one more thing. Since environment variables aren't remembered across terminal sessions, you may find it tedious to always have to set the `FLASK_APP` environment variable when you open a new terminal window to work on your Flask application. But luckily, Flask allows you to register environment variables that you want to be automatically used when you run the `flask` command. To use this option you have to install the *python-dotenv* package:

```
(venv) $ pip install python-dotenv
```

Now you can just write the environment variable name and value in a file named *.flaskenv* located in the top-level directory of the project:

```
.flaskenv: Environment variables for flask command

FLASK_APP=microblog.py
```

The `flask` command will look for the *.flaskenv* file and import all the variables defined in it exactly as if they were defined in the environment.

Continue on to the next chapter.

Buy me a coffee?

Thank you for visiting my blog! If you enjoyed this article, please consider supporting my work and keeping me caffeinated with a small one-time donation through [Buy me a coffee](#). Thanks!
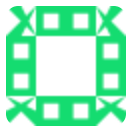

Share this post

54 comments

#1   Vladimir Kuzmenkov   said a year ago

Looks great, starting my learning journey.

#2   Radouane   said a year ago

I just bought 2022 edition book a month ago. Today, Dec 5th, 2023, I got the 2024 edition :)
I will keep both books. I think this is the least we can do to support your work. You have helped a lot of people to learn flask with your work :) Are you going to update the Udemy course too? Thanks

#3   Miguel Grinberg   said a year ago

@Radouane: Thanks! I'm currently not planning to update the video course.

#4   Ismet   said a year ago

Keep up the good work, Miguel!

#5 **Varoon** said a year ago

How do you deactivate python venv?

#6 Miguel Grinberg said a year ago

@Varoon: type `deactivate` at the prompt, or activate a different virtual environment to switch to another one.

#7 **Dan** said a year ago

Thank you Miguel, I hope to finish this book by end of Jan
I kept getting this error
Try 'flask run --help' for help.

Error: Could not import 'microblog'.

This worked for me
export FLASK_APP=app/microblog.py

#8 Miguel Grinberg said a year ago

@Dan: I recommend that you use the links to GitHub in the introduction in each chapter to see what the correct file structure for the project is. In this case, the microblog.py file should be in the top-level folder, not inside app like you have it.

#9 **Marco Renedo** said a year ago

Amazing resource!. I think it would be nice to point out that .flaskenv should be added to .gitignore.

#10 Miguel Grinberg said a year ago

@Marco: you are incorrect. The `.flaskenv` file should NOT be added to `.gitignore` and should be committed with the code. There shouldn't be any sensitive data stored in this file, so it is safe to be kept under source control. You are thinking of the

`.env` file, which is normally used to store secrets. That's a different story, that one is covered much later in the tutorial.

#11   **Ssali Jonathan**   said a year ago

Greetings Miguel, I would like to thank you for having taught me the Flask framework. A lot of speak against Flask saying it is a bit old-fashioned and useless to learn. I was surprised to get my first full-time position where we were using it for developing a microservice,

Thanks for the great work

#12   **mahima**   said a year ago

I tried following the official tutorial but couldnt quite figure it out and now i came across this!! thanks !! now i can actually follow a flask tutorial :)

#13   **Erik Javier Baaken**   said a year ago

So I went through every single step. I just cannot put my head around the very last part and do not understand what is meant with:
'Now you can just write the environment variable name and value in a file named .flaskenv located in the top-level directory of the project:
.flaskenv: Environment variables for flask command
FLASK_APP=microblog.py
The flask command will look for the .flaskenv file and import all the variables defined in it exactly as if they were defined in the environment.'

What am I supposed to do here?

#14   Miguel Grinberg   said a year ago

@Erik: when in doubt, you can always look at the working code for the chapter you are on using the link at the top of the article.

In this case, you have to write a file with the name `.flaskenv` and add the contents as indicated in the article.

#15   **Andrew May**   said a year ago

Hi, ive been following this using the ebook, really clear all the way through, thanks :) I'm trying to host it on App Engine on GCP, and have to change the SQLite db to mySQL as GAE doesnt support SQLite.

Ive set up the new mySQL db on Google App Engine, connected the flask GAE project to the mySQL db, and created a .env file with the DATABASE_URL to point to the mySQL db on google cloud platform, and put the env_variables into the app.yaml file needed.
I'm a bit stuck now, because when I try to run the db migrations using my local code, it still just updates the local sqlite db, not the mySQL db on App Engine. I can access the app on GAE, and get to the home page, but any interaction with the db results in the 'Unexpected Error has Occured' message. Is there something Ive missed ? The db migrations should update the mySQL databases, shouldnt they?
Hope you can help!

#16   **Miguel Grinberg**   said a year ago

@Andrew: It is really not possible for me to diagnose the problem with the little information you provided. I suggest you continue with this tutorial, which eventually shows how to switch to both Postgres and MySQL for different deployment options.

#17   **Dhiraj Kapse**   said a year ago

Starting my Web Development Journey with Flask! Thanks Miguel for providing this valuable resources for free. Also I request you to provide a tutorial on configuration of flask back-end with react front-end. <3 from India.

#18   **Vishwajeet Prasad**   said a year ago

Hi Miguel!

Just started out with this course, and it is good. Thanks for your effort.

I have one small question, what's the purpose of top-level python script that defines Flask application instance and also setting up FLASK_APP environment variable to point it to this script?

As I read the quick start tutorial from the flask's website, they have mentioned, that if your flask app file name is "app.py" or "wsgi.py", you can ignore --app option (which I think is analogous to FLASK_APP env var) and we actually used app.py for flask application. Then what's the further need of having additional top level script and setting up env var?

#19   **Miguel Grinberg**   said a year ago

@Vishwajeet: the main application script in this tutorial is called *microblog.py*, so it needs to be configured. You could move the functionality in this module somewhere inside the application package, and then it would be okay to not define `FLASK_APP`, since the `app` package would be found as a default.

#20   **Rickey**   said a year ago

I'm not that much of a internet reader to be honest but your blogs really nice, keep it up!
I'll go ahead and bookmark your website to come back down the road.
Cheers

#21   **William Smith**   said a year ago

I am planning to go through this series step by step and document my work on GitHub. GNU General Public License v3.0. I would prominently display credit directly to you and this series. Would this be OK with you?

#22   **Miguel Grinberg**   said a year ago

@William: Yes. The code is licensed with the MIT license. See the GitHub repository for details.

#23   **Gareth Thompson**   said a year ago

Hi Miguel,

For some reason I'm getting the following error when I try to run flask --
(venv) gareththompson@Gareths-MacBook-Pro microblog % flask run
Usage: flask run [OPTIONS]
Try 'flask run --help' for help.

Error: Failed to find Flask application or factory in module 'microblog'. Use 'microblog:name' to specify one.
(venv) gareththompson@Gareths-MacBook-Pro microblog %

Any help would be greatly appreciated, it's driving me insane 😂

#24   *Miguel Grinberg*   said a year ago

@Gareth: you did not create a microblog.py file, or you put it in the wrong place and Flask cannot find it. See the GitHub link near the top of the article to review the correct project structure for this chapter.

#25   **Petr**   said a year ago

Thank you so much for this resource!

Leave a Comment

Name

Email

Comment

Captcha

☐ I'm not a robot

reCAPTCHA
Privacy - Terms

Submit

The Flask Mega-Tutorial   **New 2024 Edition!**

If you would you like to support my work on my Flask Mega-Tutorial series on this blog and as a reward have access to the complete tutorial nicely structured as a book and/or a set of videos, you can now order it from my Courses site or from Amazon.

Click here to get the Book!
Click here to get the Video Course!

About Miguel

Welcome to my blog!

I'm a software engineer and technical writer, currently living in Drogheda, Ireland.

You can also find me on Github, LinkedIn, Bluesky, Mastodon, Twitter, YouTube, and Patreon.

Thank you for visiting!

Categories

AI *2*

Arduino *7*

Authentication *10*

Blog *1*

C++ *5*

CSS *1*

Cloud *11*

Database *23*

Docker *5*

Filmmaking *6*

Flask *128*

Games *1*

IoT *8*

JavaScript *35*

MicroPython *9*

Microdot *1*

Microservices *2*

Movie Reviews *5*

Personal *3*

Photography *7*

Product Reviews *2*

Programming *191*

Project Management *1*

Python *173*

REST *7*

Raspberry Pi *8*

React 18

Reviews 1

Robotics 6

Security 12

Video 22

WebSocket 2

Webcast 3

Windows 1