The Flask Mega-Tutorial Part I: Hello, World! (/post/the-flask-mega-tutorial-part-i-hello-world)

December 5 2017

Posted by Miguel Grinberg (/author/Miguel Grinberg) under Flask (/category/Flask), Programming (/category/Programming), Python (/category/Python).

Welcome! You are about to start on a journey to learn how to create web applications with Python (https://python.org) and the Flask (http://flask.pocoo.org) framework. The video above will give you an overview of the contents of this tutorial. 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!

For your reference, below is a list of the articles in this series.

- Chapter 19: Deployment on Docker Containers (/post/the-flask-mega-tutorial-part-xix-deployment-on-docker-containers)
- Chapter 20: Some JavaScript Magic (/post/the-flask-mega-tutorial-part-xx-some-javascript-magic)
- Chapter 21: User Notifications (/post/the-flask-mega-tutorial-part-xxi-user-notifications)
- Chapter 22: Background Jobs (/post/the-flask-mega-tutorial-part-xxii-background-jobs)
- Chapter 23: Application Programming Interfaces (APIs) (/post/the-flask-mega-tutorial-part-xxiii-application-programming-interfaces-apis)

Note 1: If you are looking for the legacy version of this tutorial, it's here (/post/the-flask-mega-tutorial-part-i-hello-world-legacy).

Note 2: If you would like to support my work on this blog, or just don't have patience to wait for weekly articles, I am offering the complete version of this tutorial packaged as an ebook or a set of videos. For more information, visit courses.miguelgrinberg.com (https://courses.miguelgrinberg.com).

All the code examples presented in this tutorial 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 (https://github.com/miguelgrinberg/microblog/tree/v0.1), Zip (https://github.com/miguelgrinberg/microblog/archive/v0.1.zip), Diff (https://github.com/miguelgrinberg/microblog/compare/v0.0...v0.1).

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 (http://python.org/download/). 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 3.5.2 (default, Nov 17 2016, 17:05:23)
[GCC 5.4.0 20160609] 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 (https://pypi.python.org/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 (in Python 2.7 `pip` does not come bundled with Python and needs to be installed separately).

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 as above. 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 0.11 of Flask, which was the most current version of Flask when you started, but now has been superseeded by version 0.12. You now want to start a second application, for which you'd like to use the 0.12 version, but if you replace the 0.11 version that you have installed you risk breaking your older application. Do you see the problem? It would be ideal if it was possible to install Flask 0.11 to be used by your old application, and also install Flask 0.12 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
```

If you are using a Python 3 version, virtual environment support is included in it, 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 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.

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

If you are using any version of Python older than 3.4 (and that includes the 2.7 release), virtual environments are not supported natively. For those versions of Python, you need to download and install a third-party tool called virtualenv (https://virtualenv.pypa.io) before you can create virtual environments. Once virtualenv is installed, you can create a virtual environment with the following command:

```
$ virtualenv venv
```

Regardless of the method you used to create it, you should have your virtual environment created. Now you have to tell the system that you want to use it, 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) $ _
```

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.

A "Hello, World" Flask Application

If you go to the Flask website (http://flask.pocoo.org/), 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 sub-directory 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 simply 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

(/post/the-flask-mega-tutorial-part-ii-templates). 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 workaround to *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 are the different URLs that the application implements. 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 your first view function, which you need to write in the new module named *app/routes.py*:

app/routes.py: Home page route

```
from app import app

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

This view function is actually pretty simple, 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 the return value of it back to the browser as a response. If this does not make complete sense yet, it will in a little bit 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

```
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 Microsoft Windows, use `set` instead of `export` in the command above.

Are you ready to be blown away? You can run your first web application, with the following command:

```
(venv) $ flask run
 * Serving Flask app "microblog"
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

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 require

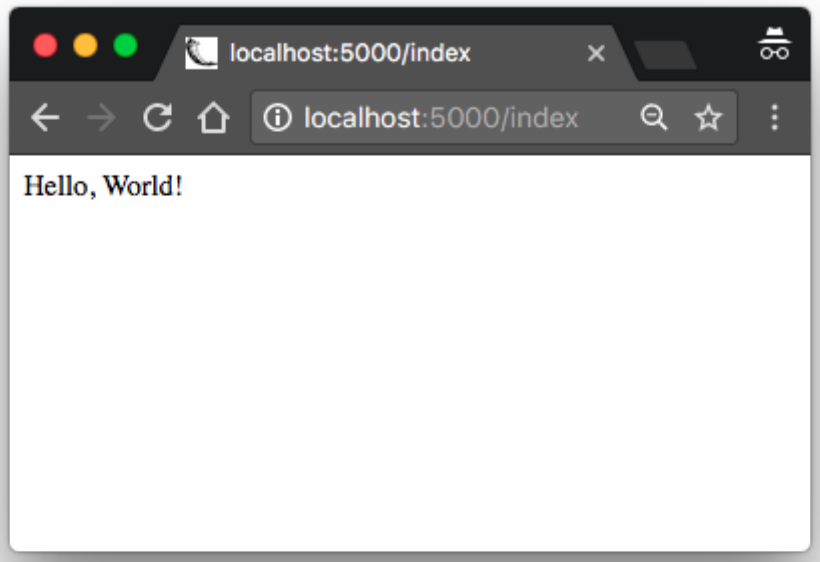
administration rights. Since this application is running in a development environment, Flask uses the freely available port 5000. Now open up your web browser and enter the following URL in the address field:

```
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!

Before I end this chapter, I want to mention one more thing. Since environment variables aren't remembered across terminal sessions, you may find tedious to always have to set the `FLASK_APP` environment variable when you open a new terminal window. Starting with

version 1.0, Flask allows you to register environment variables that you want to be automatically imported when you run the `flask` command. To use this option you have to install the *python-dotenv* package:

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

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

.flaskenv: Environment variables for flask command

```
FLASK_APP=microblog.py
```

Doing this is optional. If you prefer to set the environment variable manually, that is perfectly fine, as long as you always remember to do it.