The New and Improved

Flask Mega-Tutorial

(2024 Edition)

Miguel Grinberg

The New and Improved Flask Mega-Tutorial (2024 Edition)

Miguel Grinberg

Jan 09, 2025

Contents

Preface

Back in 2012, I decided to start a software development blog. Because I am a do-it-yourselfer at heart, instead of using Blogger or WordPress, I sat down and wrote my own blog engine, using a then little known web framework called Flask. I knew I wanted to code it in Python, and I first tried Django, which was the most popular Python web framework at the time. But unfortunately Django seemed too big and too structured for my needs. I've found that Flask gave me as much power, while being small, unopinionated and unobtrusive.

Writing my own blog engine was an awesome experience that left me with a lot of ideas for topics I wanted to blog about. Instead of writing individual articles about all these topics, I decided to write a long, overarching tutorial that Python beginners can use to learn web development. And just like that, the Flask Mega-Tutorial was born!

The book that you have in your hands has been revised several times since 2012 to keep up with changes in the Python and Flask ecosystems. I released a first major revision, update and expansion of the original tutorial in 2018 thanks to the support of almost 600 Kickstarter backers. In 2021 I refreshed the content again after the release of Flask 2.0. This last revision, which I'm calling the "2024 Edition", includes support for Flask 3.0.

Who This Book Is For

This book will take you on a journey through a realistic web development project, from start to end. If you have just a little bit of experience coding in Python and understand how the web works at a high-level, you should have no trouble using this book to learn how to develop your own web applications using Python and Flask.

The tutorial assumes that you are familiar with the command line in your operating system. If you aren't, then I recommend that you learn how to execute programs, create directories, copy files, etc. using the command line before you begin.

If you have learned Flask with my original Mega-Tutorial, this new edition will introduce you to new features in Flask that did not exist when I wrote the original articles, as well as give you an updated look at important topics such as authentication, full-text search and internationalization. In addition to the revised content, this version of the tutorial includes new chapters that cover topics that have become relevant in recent times, such as APIs, background jobs and containers.

Requirements

The example code that accompanies this book can be used on any platform on which Python runs, so macOS, Linux and Microsoft Windows are all valid choices. I have tested all the code extensively on Python versions as far back as 3.5 and continue to do so as new versions come out, so any recent version of Python should work fine.

If you are using a Microsoft Windows computer, you probably know that the world of web development is dominated by UNIX-based workflows, and you may rightly feel that you are at a disadvantage. That should not be a major concern when you work with this book, because when necessary, specific instructions that apply to Windows users are noted. My assumption is that if you are working on Windows you will be using the command prompt to work with your application. If you prefer to use PowerShell, you will need to translate commands to the appropriate syntax for that shell.

This may be hard to accept if you work on Windows, but I think you will have a better experience if you force yourself to learn UNIX, which can be done right on your Windows computer without making any drastic configuration changes. My recommendation is that you install UNIX tools on your Windows system and adopt the UNIX workflow. If you are interested in doing this, one option is the Windows Subsystem for Linux (WSL)[1], an officially supported feature of Windows 10 and 11 that adds a Linux system that runs in parallel with your Windows operating system and includes a UNIX version of Python. If your system is not compatible with WSL, then another very good option is Cygwin[2], an open-source POSIX emulation layer that includes Windows ports of a large number of UNIX tools, including Python. I have worked with Python under both WSL and Cygwin and find them perfectly adequate for web development work.

About The Example Application

The application that I'm going to develop as part of this tutorial is a nicely featured microblogging server that I decided to call *Microblog*. Pretty creative, I know.

Just so that you have some idea of what you will learn if you follow this tutorial, these are some of the topics that I will cover:

- User management, including secure password handling, logins, user profiles and avatars.
- Database management and database migration support
- Handling of user input via web forms

[1] https://msdn.microsoft.com/en-us/commandline/wsl/about
[2] https://cygwin.org

- Pagination of long lists of items
- Full-text search
- Email notifications to users
- HTML templates
- Working with dates and times
- Internationalization and localization
- Installation on a production server
- Working with Docker containers
- Application Programming Interfaces
- Push notifications
- Background jobs

I hope this application will serve as a template that you can use for writing your own web applications.

How To Work With The Example Code

I have released the complete source code for this project on the following GitHub repository: https://github.com/miguelgrinberg/microblog. There is a commit in this repository for each chapter.

The way I envision you will work through this tutorial is by writing the application on your own, based on the instructions provided in the text, at least for the first few chapters. You can certainly copy and paste portions of code from the text or from GitHub to save some typing, but I think it is important that you familiarize yourself with the task of coding a Flask application by writing the code yourself, instead of just downloading the files from GitHub (unless explicitly instructed to do so).

The GitHub repository can serve as a reference if you get lost and can't get the application to work. You can compare your files against the code in the repository link provided with each chapter if you get stuck with a problem you can't solve.

Conventions Used In This Book

This book frequently includes commands that you need to type in a terminal session. For these commands, a $ will be shown as a command prompt. This is a standard prompt for many Linux shells, but may look unfamiliar to Microsoft Windows users. For example:

```
$ python hello.py
hello
```

In a lot of the terminal examples, you are going to be required to have an activated *virtual environment* (do not worry if you don't know what this is yet, you will find out very soon!). For those examples, the prompt will appear as `(venv) $`:

```
(venv) $ python hello.py
hello
```

You will also need to interact with the Python interactive interpreter. Examples that show statements that need to be entered in a Python interpreter session will use a >>> prompt, as in the following example:

```
>>> print('hello!')
hello
```

In all cases, lines that are not prefixed with a $ or >>> prompt, are output printed by the command, and should not be typed.

Acknowledgements

This project would not have been possible without the amazing support of my Kickstarter backers. My deepest thanks go to Dhritiman Sagar, Alex Anderson, Bahrom Matyakubov, Dave Finnegan, John Gann, John W. O'Brien, Kojo Idrissa, Mark Anders, Raph, Fredrik Dahlgren, Jorge García García, Todd Twiggs, Pietro P Peterlongo, Chris Davis, Alexandre Harano, Bob Jordan, Chris Dent, Chris Jones, CptJason, Daniel Abeles, Daniel Plas Rivera, Dipanjan Sarkar, Eric Chou, Eric Ho, Graham Williamson, jiho Bak, John Sobanski, Kai Mies, Len Sumnler, Marc P. Rostock, Michael Sim, Nick Brandaleone, Nnamdi E. Anyanwu, R. Da Costa Faro, Reimund Klain, Scott Strattner, SNC Cloud Dev (twitter.com/snc_clouddev), T81, Tobias Siebenlist, Viet Le, Ed Wachtel, Shivas Jayaram, JVA, GenLots.com, Martin Thorsen Ranang, DFW Python, Allan Swanepoel, Andrej Stabenow, Anthony Bourguignon, Aron Filbert, Auke Bakker, Bryson Tyrrell, Chuck Woodraska, Colin R. Crossman, Dario Varotto, Dax Morrow, Eric G. Barron, Everett Toews, Fisherworks, flasky mcflaskface, Iain Hunter, Jeremy Barisch Rooney, Jesse Liles, Jindrich K. Smitka, Jing Sheng Pang, Karthik Ramakrishnan, Kevin Porterfield (KP), Leonel Decunta, Martynas Budvytis, Mathew Divine, Matt Makai (Full Stack Python), Matt Trentini, Michael from Talk Python, Nana B Okyere, Nathan Sanders, Nduka Obinna Azubuike, Neal Duncan, Philip Penquitt, Rémi Debette, Romer Ibo, Ryan Hagan, Scott Andrew Underwood, Stephan Simon, Steve Bartell, Timothy DAuria, Vitaly Popovich, Yi Luo and the remaining 484 backers.

1. Hello, World!

Welcome! You are about to start on a journey to learn how to create web applications with Python[1] and the Flask[2] framework. In this first chapter, you are going to learn how to set up a Flask project. By the end of the 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[3], Zip[4], Diff[5].

1.1. 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[6]. 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.

[1] https://python.org
[2] http://flask.pocoo.org
[3] https://github.com/miguelgrinberg/microblog/tree/v0.1
[4] https://github.com/miguelgrinberg/microblog/archive/v0.1.zip
[5] https://github.com/miguelgrinberg/microblog/compare/v0.0...v0.1
[6] http://python.org/download/

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 macOS versions of Python you can also exit the interpreter by pressing Ctrl-D. On Windows, the exit shortcut is Ctrl-Z followed by Enter.

1.2. 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[7], 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.

[7] https://pypi.python.org/pypi

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"
```

1.3. A "Hello, World" Flask Application

If you go to the Flask's quick start page[8], 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:

[8] https://flask.palletsprojects.com/en/3.0.x/quickstart/

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

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

Listing 1.1: *app/__init__.py*: Flask application instance

```python
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*:

Listing 1.2: *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:

Listing 1.3: *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 production deployment.
   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.

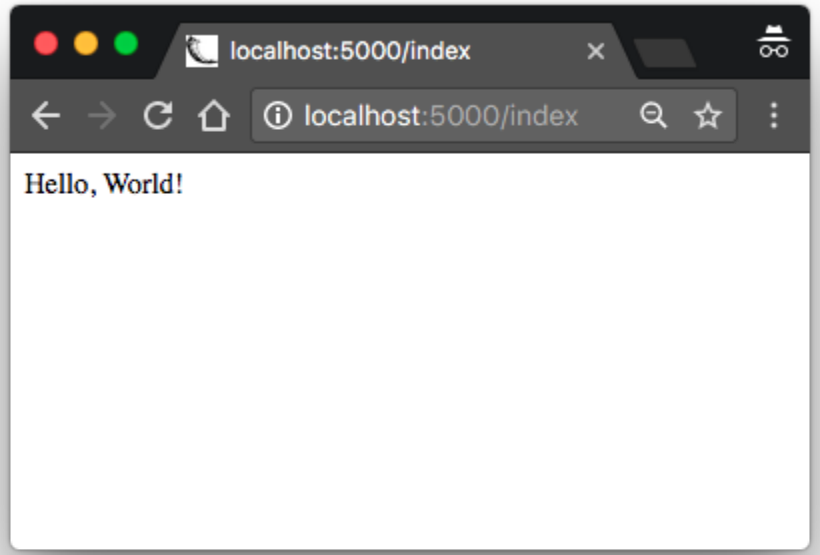
After the server initializes it will wait for client connections. The output from `flask run` indicates that the server is running on IP address 127.0.0.1, which is always the address of your own computer. This address is so common that is also has a simpler name that you may have seen before: *localhost*. 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.



Hello, World!

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:

Listing 1.4: *.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.

2. Templates

After completing *Chapter 1*, you should have a simple, yet functional web application that has the following file structure:

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

To run the application you set the `FLASK_APP=microblog.py` in your terminal session (or better yet, add a *.flaskenv* file with this variable), and then execute `flask run`. This starts a web server with the application, which you can open by typing the *http://localhost:5000/* URL in your web browser's address bar.

In this chapter you will continue working on the same application, and in particular, you are going to learn how to generate more elaborate web pages that have a complex structure and many dynamic components. If anything about the application or the development workflow so far isn't clear, please review *Chapter 1* again before continuing.

The GitHub links for this chapter are: Browse[1], *Zip*[2], *Diff*[3].

2.1. What Are Templates?

I want the home page of my microblogging application to have a heading that welcomes the user. For the moment, I'm going to ignore the fact that the application does not yet have the concept of users, as this is going to come later. Instead, I'm going to use a *mock* user, which I'm going to implement as a Python dictionary, as follows:

[1] https://github.com/miguelgrinberg/microblog/tree/v0.2
[2] https://github.com/miguelgrinberg/microblog/archive/v0.2.zip
[3] https://github.com/miguelgrinberg/microblog/compare/v0.1...v0.2

```
user = {'username': 'Miguel'}
```

Creating mock objects is a useful technique that allows you to concentrate on one part of the application without having to worry about other parts of the system that don't exist yet. I want to design the home page of my application, and I don't want the fact that I don't have a user system in place to distract me, so I just make up a user object so that I can keep going.

The view function in the application returns a simple string. What I want to do now is expand that returned string into a complete HTML page, maybe something like this:

Listing 2.1: *app/routes.py*: Return complete HTML page from view function

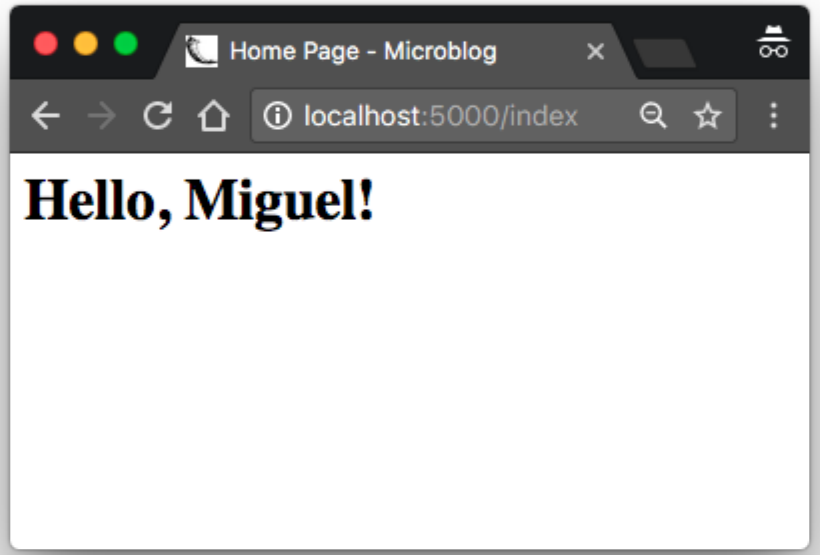
```
from app import app

@app.route('/')
@app.route('/index')
def index():
    user = {'username': 'Miguel'}
    return '''
<html>
    <head>
        <title>Home Page - Microblog</title>
    </head>
    <body>
        <h1>Hello, ''' + user['username'] + '''!</h1>
    </body>
</html>'''
```

If you are not familiar with HTML, I recommend that you read HTML Markup[4] on Wikipedia for a brief introduction.

Update the view function as shown above and run the application again to see how it looks in your browser.

[4] https://en.wikipedia.org/wiki/HTML#Markup



Hello, Miguel!

I hope you agree with me that the solution used above to deliver HTML to the browser is not good. Consider how complex the code in this view function will become when you add blog posts from users, which are going to constantly change. The application is also going to have more view functions that are going to be associated with other URLs, so imagine if one day I decide to change the layout of this application, and have to update the HTML in every view function. This is clearly not an option that will scale as the application grows.

If you could keep the logic of your application separate from the layout or presentation of your web pages, then things would be much better organized, don't you think? You could even hire a web designer to create a killer website while you code the application logic in Python.

Templates help achieve this separation between presentation and business logic. In Flask, templates are written as separate files, stored in a *templates* folder that is inside the application package. After making sure that you are in the *microblog* directory, create the directory where templates will be stored:

```
(venv) $ mkdir app/templates
```

Below you can see your first template, which is similar in functionality to the HTML page returned by the `index()` view function above. Write this file in *app/templates/index.html*:

Listing 2.2: *app/templates/index.html*: Main page template

```
<!doctype html>
<html>
    <head>
        <title>{{ title }} - Microblog</title>
    </head>
    <body>
        <h1>Hello, {{ user.username }}!</h1>
    </body>
</html>
```

This is a standard, short HTML page. The only interesting thing in this page is that there are a couple of placeholders for the dynamic content, enclosed in `{{ ... }}` sections. These placeholders represent the parts of the page that are variable and will only be known at runtime.

Now that the presentation of the page was offloaded to the HTML template, the view function can be simplified:

Listing 2.3: *app/routes.py*: Use render template() function

```
from flask import render_template
from app import app

@app.route('/')
@app.route('/index')
def index():
    user = {'username': 'Miguel'}
    return render_template('index.html', title='Home', user=user)
```

This looks much better, right? Try this new version of the application to see how the template works. Once you have the page loaded in your browser, you may want to view the source HTML and compare it against the original template.

The operation that converts a template into a complete HTML page is called *rendering*. To render the template I had to import a function that comes with the Flask framework called `render_template()`. This function takes a template filename and a variable list of template arguments, and returns the same template, but with all the placeholders in it replaced with actual values.

The `render_template()` function invokes the Jinja[5] template engine that comes bundled with the Flask framework. Jinja substitutes `{{ ... }}` blocks with the corresponding values, given by the arguments provided in the `render_template()` call.

[5] http://jinja.pocoo.org