14. Web Applications

14.1. Web Applications

14.2. How the Web Works

14.3. How Web Applications Work

14.4. Web Applications and HTML Forms

14.5. Writing Web Applications With Flask

14.6. More About Flask

14.7. Input For A Flask Web Application

14.8. Web Applications With a User Interface

14.1. Web Applications

In this chapter, you will learn how to create web applications in Python. There are two kinds of web applications – “client side” web applications and “server side” web applications. You’ve probably used both kinds in your travels on the World-Wide Web.

Client-side web applications are programs that are downloaded to a web browser and executed on the user’s local machine. Client-side applications are typically written in JavaScript and embedded in web pages.

Server-side web applications are programs that run on web server computers, rather than on the user’s local machine. Typically, a server-side web application displays a form with text boxes and other data collection mechanisms. The user fills out the form, clicks a submit button, and the browser sends the form to the web application on the server, which processes the request, and responds with another web page. If you’ve ever shopped online, you’ve used a server-side web application.

In this chapter, you will learn to build server-side web applications, which I will refer to simply as “web applications.”

14.2. How the Web Works

Before learning how to write a web application, you need to understand a bit about how web browsers work, and how web applications interact with users.

A web browser, at its core, is a fairly simple application. Basically, web browsers

Request files from web servers.

The World-Wide Web is composed of thousands of web servers connected to the Internet. Each web server contains lots of different kinds of files that web browsers can request: HTML pages, image files, audio files, and other resources. When you click a link on a web page, the web browser sends a request to the web server, which transmits the requested file back to the browser.

Process the downloaded files appropriately.

Once the web browser has downloaded the requested file, it needs to do something with it. Web browsers know how to render an HTML document, show images, play audio files, and so on. If the web browser doesn’t know what to do with a file, it usually prompts the user to save the file, so the user can do something with it.

Let’s take a specific example. Use your browser to access the following URL:

https://docs.python.org/3/library/index.html

Note

A URL (“Uniform Resource Locator”) is the address of a resource on the Web. It has three sections: the protocol (ex. https:) the browser uses to request the resource, the server where the document is located (ex. docs.python.org), and the path to the requested resource on the server (ex. /3/library/index.html).

When you click on this link, here’s what happens:

The browser opens a network connection to the web server named docs.python.org

The browser requests a file located on the server at /3/library/index.html

The web server transmits the HTML file back to the browser

The browser renders the HTML document

../\_images/webrequest.jpg

If you want to see the file transmitted by the web server to the browser, right-click in the browser window and choose View Page Source (your browser’s option to view the source may be slightly different). The browser shows you the file it downloaded from the web server.

14.3. How Web Applications Work

Most of the time, when your browser requests a file from a web server, the server simply transmits the contents of the file back to the browser. But sometimes, the “file” your browser requests isn’t really a file at all.

Try typing the following URL into your browser:

https://google.com/search?q=Microsoft

You’ll get back a page of search results about Microsoft from the Google search engine (at least, you will unless Google has changed how it performs searches since this chapter was written). How did this happen?

Well, your browser did what it always does when you type in a URL:

The browser opened a network connection to the web server named google.com

The browser requested the “file” named /search?q=Microsoft from the web server

What the web server did at this point is different than the example above. There’s no “file” named “/search?q=Microsoft” on the Google web server. Instead, the web server ran a web application to search through Google’s massive database of websites for pages that mention “Microsoft”. The web application dynamically generated an HTML document containing the search results, and the web server transmitted that document back to the browser.

The browser rendered the HTML document

As far as your browser is concerned, there is no difference between requesting a “static” HTML file from a web server, and requesting a dynamically generated HTML file. It’s up to the web server to examine the request submitted by the web browser to determine whether it should serve up a regular document, or run a web application to generate a response.

Anytime you’re browsing the web, and you notice that the URL of the page you’re viewing has a question mark (?), you can be fairly certain that the page was generated “on the fly” by a web application on a web server. By the way, the portion of the URL that comes after the ? is called the “query string,” and contains input for the web application. Try changing the query string by substituting “Firefox” for “Microsoft” to see what I mean.

In summary, a (server-side) web application is a program that is run by a web server to produce output in response to an incoming request from a web browser.

14.4. Web Applications and HTML Forms

Perhaps you’re thinking, “I don’t usually perform searches by typing in URL’s — I fill out a search form.” True — if web applications forced users to interact with them by entering query strings, the World-Wide Web would be a much less popular place.

Let’s explore the relationship between forms and query strings a bit. Bring up the Google home page (I’ll wait):

https://google.com

Now, type in your query. When I type in “Microsoft” and click Search, here is what I see:

../\_images/googlesearchresults.png

Now, take a good look at the URL in the title bar — notice the query string? It’s a bit more complicated than the one I had you create by hand earlier. But you can probably pick out the “q=Microsoft” if you look closely. How did all of that get there? Well, when you clicked Search, the browser took the information you typed into the form, packaged it up into a query string, and transmitted it to the Google web server. You see, when you fill out a form on a web page and click Submit, the browser uses the form data to construct a URL, and then sends a normal request to the web server.

Even if you’re a novice at writing HTML pages, it’s not hard to learn to create HTML forms. Take a look at this simplified version of the Google home page:

<html>

<head>

<title>Google</title>

</head>

<body>

<div align="center">

<img src="https://www.google.com/images/logo.png"><br><br>

<form action="https://google.com/search">

Enter your search words: <input type="text" name="q"><br><br>

<input type="submit" name="btnG" value="Google Search">

</form>

</div>

</body>

</html>

Focus on the region of this example in between the <form> tags. Here’s a quick overview of this part of the page:

The form is the region of the page in between the <form> and </form> tags.

The form can contain a mixture of text, regular HTML formatting tags, and form <input> tags

Each <input> tag has a type and a name attribute. The type attribute specifies what kind of input area it is (“text” for a text box, “submit” for submit button, etc.). The name attribute specifies a name for the input area.

When the user fills out the form and clicks the submit button, the browser constructs a URL by taking the form’s action attribute (https://google.com/search), appending a ?, and constructing the query string using the names of the form input areas, together with the data entered by the user.

Try it out! Using Notepad, type in this example, and save it as googleform.html. Open it in your browser; you should see something like this:

../\_images/googleform.png

Fill out the form, and, if Google still works as it did when this chapter was written, you should see search results appear in your browser.

For more information about creating HTML forms, you might take a look at the excellent tutorial at w3schools.com.

14.5. Writing Web Applications With Flask

In this section, you will learn how to write web applications using a Python framework called Flask.

Here is an example of a Flask web application:

from flask import Flask

from datetime import datetime

app = Flask(\_\_name\_\_)

@app.route('/')

def hello():

return """<html><body>

<h1>Hello, world!</h1>

The time is """ + str(datetime.now()) + """.

</body></html>"""

if \_\_name\_\_ == "\_\_main\_\_":

# Launch the Flask dev server

app.run(host="localhost", debug=True)

The application begins by importing the Flask framework on line 1. Lines 6-11 define a function hello() that serves up a simple web page containing the date and time. The call to app.run() on Line 14 starts a small web server. The run() method does not return; it executes an infinite loop that waits for incoming requests from web browsers. When a web browser sends a request to the Flask web server, the server invokes the hello() function and returns the HTML code generated by the function to the web browser, which displays the result.

To see this in action, copy the code above into a text editor and save it as flaskhello.py (or whatever name you like). Then, download the Flask framework and install it on your computer. In many cases, you can accomplish this using the pip command included with your Python distribution:

pip install flask

Next, execute your flaskhello.py program from the command line:

python flaskhello.py

Note

If you are using a Mac or Linux computer, use the following command to install flask:

pip3 install flask

and execute your flaskhello.py program using the following command:

python3 flaskhello.py

When you launch the program, you should see a message similar to the following appear on the console:

\* Serving Flask app "sample" (lazy loading)

\* Environment: production

WARNING: Do not use the development server in a production environment.

Use a production WSGI server instead.

\* Debug mode: on

\* Restarting with stat

\* Debugger is active!

\* Debugger PIN: 244-727-575

\* Running on http://localhost:5000/ (Press CTRL+C to quit)

Note

If you get an error message of some sort, it is possible that your computer may be running a server application that is using the port number that Flask wants to use. See the next section, “More About Flask,” for a discussion of port numbers and how to address this issue.

Once the Flask server is running, use your browser to navigate to the following URL:

http://localhost:5000/

Your browser sends a request to the Flask server, and you should see a “Hello, world!” message appear:

../\_images/flaskhello.png

To send the request again, press the Reload button in your browser. You should see the date and time change.

Scratch Activecode

Help

This Chapter

14.6. More About Flask

When you executed the flaskhello.py program in the previous section and used a web browser to access it with the url http://localhost:5000/, in addition to seeing a “Hello, world!” message in the browser, you should also have observed a log message like the following in the console:

127.0.0.1 - - [21/Apr/2016 08:02:28] "GET / HTTP/1.1" 200 -

Every time the Flask server receives a request from a browser, it writes a log message to the console. The message contains information such as the IP address of the computer that sent the request (127.0.0.1 is a special address indicating the request came from the browser on the same computer that the Flask server is running on); the date and time of the request; the path of the incoming request (“/” in this case); and the status of the result (here, 200 indicates the request was successfully processed).

The Flask server continues running until you press Ctrl-C to stop it. At that point, if you try to send a request to the application from the browser, the browser will display an error message indicating that it cannot contact the server. Go ahead and try this, so you can recognize what the error message looks like in your particular browser.

Recall that every URL has at least three components: the protocol, server, and the path. In our case the URL http://localhost:5000/ has the server name localhost, the path /, and an additional component: the port number, 5000. Let’s discuss some details about each of these.

Server name

When you use the name localhost in a URL, the browser attempts to connect to a server program running on your computer. This is the usual scenario when you are developing a web application: the browser and the server application are both running on the same computer. When you deploy the application to be hosted on an actual server, you will use the name of the server in the URL instead of the name localhost. If you want to experiment with deploying Flask applications to a public web server, check out pythonanywhere.com, which (at the time of writing) provides free hosting for Flask web applications.

Port number

Each server application running on a computer is assigned a distinct port number so that clients can connect to it. Port numbers range from 0 to 65,535. Web servers generally are assigned port number 80, and when the URL does not contain a port number, the web browser attempts to connect to a web server listening on port 80. But the default port number for Flask applications is 5000, so the URL must include that port number. You can specify a different port number for your Flask application in the line that launches the Flask server like this:

app.run(host="localhost", port=5001, debug=True)

Here, the Flask server binds to port 5001, and you would need to use that port number instead of 5000 in the URL in the browser.

Path

When the Flask receives an incoming request, it examines the path and uses it to determine which function in your program should be executed to handle the request and generate a response. A Flask application can contain one or more of these request handler functions, which are decorated by a line immediately preceding the function that looks like this:

@app.route('/')

The path in the quotes is matched to the path of the incoming request from the browser. If the incoming path from the browser does not match any of the handler function paths defined by @app.route() decorators, an error occurs. For example, try entering the following URL into your browser when the flaskhello.py program in the last section is running:

http://localhost:5000/blah

You will see an error message appear in the browser, and the log message that appears in the console will have the number 404 after the path, indicating that the path did not match, as shown below:

127.0.0.1 - - [21/Apr/2016 08:02:51] "GET /blah HTTP/1.1" 404 -

Here’s another version of the flaskhello.py program that has two different pages. The first page displays a “Hello world” message and invites the user to click a link to view the time. When the user clicks the link, the time appears.

from flask import Flask

from datetime import datetime

app = Flask(\_\_name\_\_)

@app.route('/')

def hello():

return HELLO\_HTML

HELLO\_HTML = """

<html><body>

<h1>Hello, world!</h1>

Click <a href="/time">here</a> for the time.

</body></html>

"""

@app.route('/time')

def time():

return TIME\_HTML.format(datetime.now())

TIME\_HTML = """

<html><body>

The time is {0}.

</body></html>

"""

if \_\_name\_\_ == "\_\_main\_\_":

# Launch the Flask dev server

app.run(host="localhost", debug=True)

Here’s how it works:

To begin, the user enters the URL http://localhost:5000, and the browser sends the request to the application. The Flask server matches that path “/” to the hello() function, invokes the function and returns the response to the browser.

The user clicks the link, which triggers the browser to send a request with the URL http://localhost:5000/time to the Flask server. The server matches the path “/time” to the time() function, invokes the function and returns a response containing the time to the browser.

Note that the user does not have to click the link in order to display the time. For example, the user could enter the URL http://localhost:5000/time directly into the browser to bypass the greeting page and get directly to the page showing the time.

The example above used the format() method to build an HTML string. For more information on format(), see String Format Method.

Also, notice how the example above defines separate HELLO\_HTML and TIME\_HTML variables to hold the HTML. This helps reduce cluttering the handler functions with HTML code, and separating the Python logic from the HTML also improves the overall readability and maintainability of the code.

14.7. Input For A Flask Web Application

In this section, we will design a web application that obtains input from the user. In the example in this section, the user must encode the input directly into the URL. In the next section, we’ll provide a more user-friendly approach for obtaining input.

The URL used to interact with a web application can contain input data in addition to the path. This input data is typically encoded into the URL in the form of a query string. Here’s an example of a URL containing a query string:

http://www.bing.com/search?q=python+flask&go=Submit

The query string is the portion that comes after the ? symbol:

q=python+flask&go=Submit

It contains a set of query variables and values, each query variable/value pair separated from the others by the & symbol. This example has a query variable named q whose value is python+flask, and a variable named go whose value is Submit.

Flask applications can access query variables using a dictionary named request.args (dictionaries are discussed in detail in Dictionaries). When a browser sends a request to a Flask application that contains a query string, the data in the query string is placed in the request.args dictionary, where it can be retrieved by the application. For example, in the Bing search URL above, if Bing were a Flask application, it could access the values in the query string like this:

q = request.args['q']

go = request.args['go']

This would retrieve the values ‘python flask’ and ‘Submit’ from the query string and store them, respectively, in q and go.

Here is an enhanced version of the original flaskhello.py program that gets the user’s name from the query string and uses it to greet the user:

from flask import Flask, request

from datetime import datetime

app = Flask(\_\_name\_\_)

@app.route('/')

def hello():

name = request.args['name']

return HELLO\_HTML.format(

name, str(datetime.now()))

HELLO\_HTML = """

<html><body>

<h1>Hello, {0}!</h1>

The time is {1}.

</body></html>"""

if \_\_name\_\_ == "\_\_main\_\_":

# Launch the Flask dev server

app.run(host="localhost", debug=True)

To test this example, you would need to enter the following URL into the browser:

http://localhost:5000/?name=Frank

If the name parameter is omitted, the application will crash when it attempts to retrieve the query parameter from the dictionary, because indexing a dictionary with a key that is not present in the dictionary is illegal. To make the application more robust, we could change line 8 to check to see if the name parameter was submitted:

if 'name' in request.args:

name = request.args['name']

else:

name = 'World'

The test 'name' in request.args is True if ‘name’ was present in the query parameters, and False if not.

A shorter way to handle a missing query parameter is to use the dictionary get() method, which allows us to supply a default value to use in case the user omits the query parameter. The if statement above could be rewritten with a single line of code:

name = request.args.get('name', 'World')

This line does the same check as the if statement. If ‘name’ is present in the query parameters, its value is stored in name. Otherwise, the value ‘World’ is stored in name if no name parameter was supplied.

14.8. Web Applications With a User Interface

This section builds on the material in the preceding sections to present a web application that prompts the user to provide input, performs some processing, and displays results.

from flask import Flask, request

app = Flask(\_\_name\_\_)

@app.route('/')

def home():

return HOME\_HTML

HOME\_HTML = """

<html><body>

<h2>Welcome to the Greeter</h2>

<form action="/greet">

What's your name? <input type='text' name='username'><br>

What's your favorite food? <input type='text' name='favfood'><br>

<input type='submit' value='Continue'>

</form>

</body></html>"""

@app.route('/greet')

def greet():

username = request.args.get('username', '')

favfood = request.args.get('favfood', '')

if username == '':

username = 'World'

if favfood == '':

msg = 'You did not tell me your favorite food.'

else:

msg = 'I like ' + favfood + ', too.'

return GREET\_HTML.format(username, msg)

GREET\_HTML = """

<html><body>

<h2>Hello, {0}!</h2>

{1}

</body></html>

"""

if \_\_name\_\_ == "\_\_main\_\_":

# Launch the Flask dev server

app.run(host="localhost", debug=True)

The program is organized as follows:

Lines 6-8 define the home() function, which defines the starting page for the application. It displays a form that prompts for the user’s name and favorite food.

The form’s action attribute on Line 13 specifies that the form submission will be directed to the path /greet. Processing for this path is defined by the greet() function on lines 20-31.

Lines 22-29 extract the information submitted on the form and compute a response message.