Server Side Programming Tutorial

by Andrew Benson for WebDevWeeks by ScottyLabs

Prerequisities

- A laptop running Windows, Mac OS X, or a normal version of Linux.
- Basic Python knowledge on the order of 15-110. We'll use a few object-oriented concepts, but you can always ask for help if you need to.
- A bit of terminal experience. We can help you, but you'll have a better time if you know how to 'cd' and 'ls'.
- Google Chrome or Mozilla Firefox. It's not that other browsers are bad (except IE6-8), but it's simpler for everyone if we can agree on a browser for this session.
- Sublime Text. You can use any text editor you feel comfortable with, but we recommend Sublime Text.
- Completion of the "Setting up your Python Development Environment" for your operating system.

What is server-side programming?

Server-side programming is what powers Outlook.com, Wolfram-Alpha, and Google Docs. It's writing programs on a server that respond to a person interacting with a webpage. It's more powerful than just some JavaScript on a page - server-side programs can store data for long periods of time and for other computers, and can do more complex operations.

Why are web apps so great?

- They're cross-platform. Web apps are accessible on any platform with a modern web browser, including desktops, tablets, phones, game consoles - just about anything.
- Web apps are always up-to-date. You can deploy your changes to your server, and users will immediately receive the updated code next time they use your webpage.

What are we building?

A blog. You can see an example running at webdevblog.herokuapp.com. There's not a lot of features, but it will be a great introduction to server-side programming concepts.

Helpful Resources:

- Python Reference: https://docs.python.org/2.7/
- Flask Reference: http://flask.pocoo.org/
- Flask-SQLAlchemy Reference: http://pythonhosted.org/Flask-SQLAlchemy/
- Templating (Jinja2) Reference: http://jinja.pocoo.org/
- Example website: http://webdevblog.herokuapp.com/

Part 1: What's going on behind the scenes of a website?

I'll be giving a brief lecture on this topic, but here's a summary for future reference.

Webpages are just files that web browsers like Firefox interpret. Really. These files include:

- HTML files, which define **WHAT** stuff is on the page and links to all the other files
- Images and other media, which look **PRETTY**
- CSS files, which define how the stuff **LOOKS**
- JavaScript files, which define how the stuff **INTERACTS** with itself and the user

Only a single HTML file is required (browsers have some defaults for how stuff looks, so we can use their CSS instead of writing our own), but good, modern webpages use all of these.

When you go to a website, your browser requests a webpage. It receives the HTML file first and reads it, putting what it can on the screen and sending out another request when it needs more information (for example, if there is an embedded image).

So what do I mean by 'request'?

Requests are messages sent by the browser to another computer that accepts such messages. These messages have a certain format that abides by a protocol called HTTP (HyperText Transfer Protocol) (there are other protocols, such as SSH and BitTorrent). We call the computer with the browser that's sending HTTP requests the "client", and the computer that responds to HTTP requests the "server". Servers respond to HTTP requests by generating and sending back a HTTP response.

Here's an example HTTP request:

```
    GET / HTTP/1.1
    Host: webdevblog.herokuapp.com
    User-Agent: Mozilla/5.0 (Windows NT 6.3; WOW64; rv:33.0) Gecko/20100101
Firefox/33.0
    Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
    Accept-Language: en-US,en;q=0.5
    Accept-Encoding: gzip, deflate
    DNT: 1
    Connection: keep-alive
    Cache-Control: max-age=0
```

The first line is special. The first part of the line is the HTTP method, which is important. GET is used anytime you just want a file, which is almost every time. POST is used when you want to send information for the server to store, like when you complete a survey and press submit (the information will be included at the end of the request). There's more, which you can see at the

W3’s specification at http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html, but these two are the only important ones for our blog.

The second part is the URL minus the website’s name. So for example, this would be “/search?q=cats” for https://www.google.com/search?q=cats, and “/create” for webdevblog.herokuapp.com/create.

The third part is the protocol version. Don’t worry about this.

Everything else is a line with the format "Header: value". These are called HTTP headers, and they give more information, like the browser you're using (see User-Agent).

Here's an example HTTP response:

```
HTTP/1.0 200 OK
Connection: keep-alive
Server: gunicorn/19.0.0
Date: Sun, 03 Aug 2014 23:51:20 GMT
Content-Type: text/html; charset=utf-8
Content-Length: 1302

<here lies all the HTML you requested>
```

A lot is similar. The first line declares the HTTP version and the status code, which summarizes the status of the request (like the famous 404 Bad Request; see https://en.wikipedia.org/wiki/List_of_HTTP_status_codes for a full list). Underneath that special first line are HTTP headers, just server-specific ones. Further below that is the HTML we requested.

Server-side programming is concerned with how the server takes a HTTP request and creates an appropriate HTTP response. If you were writing a server from scratch, understandably this would be quite a bit of work (if you take 15-213, your last lab involves writing parts of a similar server). Fortunately, people have created frameworks that you can take advantage of to do the mundane things like parsing requests so you can focus on the important parts.

Popular frameworks you may have heard of are Ruby on Rails, Express, and Django. We'll be using a Python framework called Flask instead of those mainstream ones.

All Flask asks you to do is write Python functions (that return HTML) that should execute when a HTTP request of a certain HTTP method for a particular URL comes in. This makes Flask ideal for beginners.

Part 2: Writing a basic Flask server

Create a folder for your server in an appropriate location (perhaps ~/projects/webdevblog, or C:\Users\<username>\projects\webdevblog).
Go into the folder. Since we'll be doing Python development in here, let's set up a virtual environment. Run in Command Prompt (not Powershell)/Terminal (hereafter referred to as terminal):

```
virtualenv venv
```

That will create a folder called venv here that contains the virtual environment. Activate it by running

```
source venv/bin/activate
```

(Windows users - run
`.\venv\Scripts\activate`
)

You should now see a (venv) appear before or after each command.
Later, you can deactivate this virtual environment at any time by running

```
deactivate
```

Now let's install Flask and the Flask extension for SQLAlchemy (for our database later on). Run

```
pip install flask flask-sqlalchemy
```

If you decide to deploy your website to an external server in the future, it'll be helpful to have a list of Flask modules you use in your website. You can do this by running

```
pip freeze > requirements.txt
```

Great. Now you're ready to start.

Start Sublime Text and open this directory. Create a file called main.py. I'll give you the starting code:

```
from flask import Flask

app = Flask(__name__)
app.debug = False

@app.route("/")
def home():
  return "Hello World"

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

Now open terminal in this directory and run

```
python main.py
```

Now open a web browser and go to localhost:5000. You should see a message waiting for you!

Remember how I mentioned that there are clients and servers? Your web browser is the client here. You're also running a server at localhost, which you wrote in main.py. Yes, you're running both the client and the server on one computer! How cool is that?

Please note: every time you make changes to your code, you will have to stop your server with CTRL-C and restart it. Alternatively, you could change app.debug to True, and you shouldn't have to restart your server. (For Flask, never set app.debug to True in a production environment because its debugging capabilities allow anyone who gets an error message to execute any code they want.)

So what's going on? We first import Flask from the flask module, and use it to create an application object that represents our webapp. Then we use a python decorator (the `@app.route("/")`) to tell our webapp that requests to "`/`" should be dealt with by running the "`home`" function, which simply returns "Hello World". At the end, we run the app.

Exercise:

Rewrite main.py so that when you navigate to "localhost:5000/about", you see the message "This is a website for my blog".

So a simple text message is kinda boring. If we returned HTML instead, it would be cooler, wouldn't it?

Exercise:

Rewrite main.py so that when you navigate to "localhost:5000/about", you see the message "This is a website for my blog", where the word "blog" is in bold.

Since this is a SERVER-side programming session, not a CLIENT-side one, I've written some HTML for you. Download the webdevblog-handout.zip available at https://github.com/anbenson/webdevblog-resources/raw/master/webdevblog-handout.zip. Extract it, and you'll find a folder called "templates" in there. Move that folder to your working directory.

How is our server going to send back an HTML file? Flask provides a convenient function called `render_template` to do this. If you call `render_template("someHTMLfile.html")`, Flask will look specifically in your templates folder (the name “templates” is special) for a file called "someHTMLfile.html" and render it into a string, and then return the string.

So

```
  return render_template("index.html")
```

will return a string containing the HTML that we should return.

In order to use "render_template", though, you're going to need to import it from the flask module as well. So modify your first line to be

```
  from flask import Flask, render_template
```

Exercise:

Rewrite "home" so that going to localhost:5000 displays the page "index.html" describes.

Exercise:

Write code so that going to localhost:5000/about displays the page "about.html" describes.

Aside:

If you examine what's really in the templates folder, you'll notice that there are a lot of mysterious curly braces and bits of python code. That’s because templates are actually mixtures of HTML and Python code that allow us to create pages with dynamic elements, like the title of a blog post, while keeping the rest of the page the same.

You might also notice "base.html". Since the top bar of each webpage is going to be the same, I've put that common HTML in "base.html" and wrote the other webpages to "extend" base.html. This removes some of the duplicated code, and makes for cleaner templates.

You may have also noticed that you got unstyled web pages. That's because your server hasn’t been given the CSS and JavaScript files the webpage needs, and thus can’t serve them. We call these "static files" because we can't dynamically change them at run time like we can do with templates. As such, Flask has a special folder called "static" that is meant for static files. If you put CSS and JS files in here, Flask will automatically serve them.

Exercise:

Find a folder in the handout called "static". Now go make localhost:5000 and localhost:5000/about serve styled pages (you'll need to create a directory called "static" in your project's home directory).

Aside:

Links to CSS and JS usually appear in HTML. You might notice in the templates that I've used the `url_for` function provided by Flask to generate a link to the appropriate static file. It's wrapped in curly braces to mark it as Python code that `render_template` should execute. Using `url_for` is better than hard coding the link because if you move the template around somehow, the link will adjust appropriately.

Exercise:

We're about to create the form that allows users to add blog posts! The corresponding template is named "create.html". Now add code so that when you go to localhost:5000/create, you see what "create.html" describes.

Part 3: Completing our Blog

The form in "create.html" doesn't do much right now. If you take a look in the template, you'll use two attributes for the form tag: "method" and "action". Method refers to the HTTP method you want the form to send when you press submit. Action refers to the URL you want the HTTP request to go to.

Exercise:

Fill out the method attribute in the form tag inside "create.html". We've only talked about two (GET and POST), so it's going to be one of those. Remember that this form will be used to send information for the server to store.

Exercise:

Let's have the form send its requests to localhost:5000/newpost. With that in mind, fill out the action attribute in the form tag inside "create.html".

Great, so now clicking submit will send requests to /newpost. We should put a handler for that in main.py. But by default "app.route" will respond to all HTTP methods. We'll need to restrict it to just the single HTTP method that the form will send.

You can restrict a handler by specifying `methods = ['<method name>']` as an additional argument to `app.route`. For example,

```
@app.route("/example", methods=["GET"])
def example():
    pass
```

is how you would keep the "example" function from running in respond to any method except for GET.

Exercise:

Write code that responds to requests to /newpost, but only respond to requests with the method you used in the form. For now, you can pass or return a simple string in the function body.

The information that the user puts into the form is also going to be inside the request. Flask gives you access to this through the request object, which you'll need to import.

Exercise:

Modify your import statement to import request from flask as well.

Take another look at "create.html". You'll notice that each input tag has an attribute called "name". When the HTTP request is sent, it'll include the information from the input elements in key-value form, like

```
post-name=blogAuthor&post-title=blogTitle&post-text=randomtext
```

Fortunately, you don't have to parse this. All this information is represented by Flask as a dictionary called `request.form` where the names are keys and the values are, er, values. So you can get the value of "post-text" above with

```
request.form["post-text"]
```

which returns `"randomtext"`.

What are we going to do with this info? Well, we're going to have to store it somewhere so that future users can see it. Let's examine our options for storage.

Idea 1: Store data in a in-memory variable.

So this would be pretty easy - all we need to do is to create some global dictionary or list in main.py and store things in there. But although it's convenient, there are a lot of problems with this approach.

Critical Thinking:

What would happen to all of our stored blog entries if our server crashed, or if we restarted the server?

Critical Thinking:

A typical laptop nowadays has around 4 to 8 gigabytes of memory (though servers will probably have a bit more). Why might that be a problem for a website like Facebook?

Anyway, in-memory storage is a pretty terrible way to store data long-term. We need to store it separately from our random-access memory.

Idea 2: Store data in a text file on the server.

This addresses the problems we encountered with in-memory storage. Data persists even when we restart the server, and there's usually terabytes of hard disk memory on a server (or at least much more than there is RAM). But a text file is awfully annoying to use - you'll have to decide on a format with which to write your data so that you'll be able to write a good parser to read it.

Idea 3: Store data in a *database* on the server.

People have already thought about this issue, though, and have already written hundreds of papers on the topic. The industry-standard solution is to use a database, which stores data in files but provides a friendly interface for accessing them. We're going to use a relational database, which is usually accessed through the Structured Query Language (SQL, pronounced "sequel"). However, we're going to avoid it since Flask has a nice extension for SQLAlchemy, which provides a Python interface for a database system called SQLite.

First let's talk about relational databases in general. Data in a database is organized into tables, which hold rows of similar data. Let's see this with an example.

Say I'm trying to make a database to store information about the members of ScottyLabs. What I would do is create a new table called "ScottyLabs Members" and define some columns for the table, each of which represents some attribute that each contact has (perhaps their first and last name, major, etc). Then you would add rows to the table, each of which represents each person.

`ScottyLabs Members`

ID	First name	Last name	Major	Specialty
1	Taylor	Poulos	IS	Design/CSS
2	Tom	Shen	CS	Servers
3	Jake	Zimmerman	CS	JavaScript

Let's take a look at the code that creates and declares our database (you'll copy it into your python file later):

```
from datetime import datetime
from flask.ext.sqlalchemy import SQLAlchemy
```

```
app.config["SQLALCHEMY_DATABASE_URI"] = "sqlite:////tmp/blog.db"

db = SQLAlchemy(app)

class BlogEntry(db.Model):
  ID = db.Column(db.Integer, primary_key=True)
  author = db.Column(db.String(50))
  title = db.Column(db.String(100))
  text = db.Column(db.String(5000))
  date = db.Column(db.DateTime())

  def __init__(self, author, title, text):
    self.author = author
    self.title = title
    self.text = text
    self.date = datetime.utcnow()

db.create_all()
```

First we import objects from modules - nothing special there. Then we set a configuration variable for our webapp that tells SQLAlchemy where to store its database, which is in the /tmp directory on your computer. Then we create our database object, and declare our table (called "BlogEntry").

BlogEntry inherits from db.Model, which is a representation of a table provided by SQLAlchemy. Note how we define a BlogEntry to be composed of an ID, author, title, text, and date, which respectively are an Integer, 50-char string, 100-char string, 5000-char string, and a datetime object.

Since this is still a standard Python class declaration, we can create instances of BlogEntry, which are the rows of the table "BlogEntry". Read that again - the table on the whole is called BlogEntry by the database, and we create BlogEntry's rows by creating instances of the Python class BlogEntry.

Our table might conceptually look something like this:

BlogEntry

ID	author	title	text	date
1	Andrew	Hello World	This is the first post...	2014-07-21 01:06:20.200564
2	Ben	Awesome site	I think this site is cool...	2014-07-22 04:03:45.506525

and so forth.

At the end of that code, we ask the database to initialize itself.

Exercise:

Copy the code we just discussed to the top of main.py, but just under "app.debug = False" (the imports, though, should go at the top). Your code might be a bit messy now - it might be a good idea to refactor it into organized sections at this point.
(IMPORTANT NOTE: If you're using Windows, you must replace `"sqlite:////tmp/blog.db"` with `"sqlite:///tmp/blog.db"` (There's one less slash). Then open Command Prompt in your project directory and enter `mkdir tmp`, then `cd tmp`, then `copy /y nul blog.db`. You can return to your project directory with `cd ..`)

Now that we're done talking about how to store our data, we can finish writing the handler for /newpost. To review, we can extract information from the request object with `request.form[<name>]` using the name attribute in the input element, and we can create BlogEntry objects by creating instances of the BlogEntry class with `BlogEntry(<author>,<title>,<text>)`. The last thing you need to know is how to add these BlogEntry objects to the database. You can do that with

```
db.session.add(<database object>)
```

When you're done with adding objects in a handler, you should use

```
db.session.commit()
```

Exercise:

Write code that handles requests to /newpost. Your code should do the following things:
- get the author, title, and text from the request object using `request.form` (if you don't remember how, review page 8 carefully)
- create a BlogEntry object from these parameters
- add the BlogEntry database object to the database
- commit your database changes

What should our handler return? (In other words, what webpage should users see after clicking submit?) That might be a bit of a design decision, whether to return a success message or to redirect to another page. I think we should redirect to the home page so that users can immediately see the new blog entry that they've created.

Flask provides the `redirect` function, which handles the redirection and the appropriate HTTP responses. It takes as an argument the URL you want to redirect to as a string. Well, Flask has a function to generate URLs too. It's called `url_for`, and you give it the name of the function in main.py that corresponds to the handler you're interested in. So for "/", that's "home", if you've been following this tutorial exactly.

Exercise:

Add a return statement to your /newpost handler that redirects users to the home page. (You should return the result of calling redirect). Don't forget to import "redirect" and "url_for" from flask.

Go ahead and try out using your form at /create. Hopefully if you type something into each input field and click submit, you'll be redirected to the home page without error.

Now try using your form again, but this time leave one field blank. What happens when you click submit? What do you think should have happened?

We probably don't want to create incomplete blog entries. Let's create a function in main.py that checks whether the parameters are okay. Better yet, let's create a helper function that creates the blog entry if the parameters are acceptable, or returns `None` otherwise.

Exercise:

Create a function in main.py called `createPost` which takes author, title, and text as parameters and returns a BlogEntry object if the parameters are okay and None otherwise. Parameters are valid if they are not `None`, not the empty string, and abide by the length restrictions that our database had (50 chars for author, 100 for title, and 5000 for text).

Exercise:

Rewrite your /newpost handler so that it uses `createPost` to create a BlogEntry and checks the result, adding it to the database and redirecting if it's valid, and simply rendering the "create.html" page again if it's not. If this were a more professional website, we might display the create.html page with a nice error message, but um...it's not.

When we go to the home page, we'd like to see all of the posts that are in the database. In order to display all of our posts, we'll need to "query" the database on our table, which returns a list of BlogEntry's. We'll also need to write some Python code that will iterate through these and display them nicely.

First, let's see how to query our database. Since we want all the data in the BlogEntry table, we'll direct our query there. We'll also tell the database we want our data to be sorted in reverse order of time created. Here's the expression for this query:

```
  BlogEntry.query.order_by(BlogEntry.date.desc())
```

(To construct more complex queries, take a look at the SQLAlchemy Query API Reference at http://docs.sqlalchemy.org/en/rel_0_9/orm/query.html.)

Now we want to write some code to iterate through these and display them in the webpage. Since this involves working with HTML, we're going to write this code directly in the template, which Flask's templating engine allows you to do.

You'll first have to pass the list of BlogEntry's you obtained from your query to your template. You can do this by adding another argument to your `render_template` function for the home page.

Exercise:

Rewrite your handler for the homepage so that it queries your database for a list of BlogEntry's and returns the result of a call to render_template with the list as a second argument. If your list of BlogEntry's is called blogEntries, then your call to render_template will look something like

```
render_template("index.html", blogEntries = blogEntries)
```

Now we can write Python code in our template that uses the blogEntries variable. Open up “index.html”. Inspect the HTML code that makes up the example blog entry. What you're going to want to do is to loop through blogEntries and write out this HTML once for each iteration, with the appropriate variables inserted in the HTML. Since that probably wasn't too clear, here's an example that will result in all the entries’ titles displaying:

```
{% for entry in blogEntries %}
<h2>{{entry.title}}</h2>
{% endfor %}
```

As you might have inferred, text between {% and %} is executed as Python code, and text between { and } is interpreted as a Python expression. Remember that your entries have title, author, date, and text fields.

Exercise:

Write similar templating code in index.html so that when you navigate to your website's homepage, you see all of your blog entries displayed.

Here’s a helpful hint if you want to format your dates prettily. Datetime objects in Python have a method called “strftime” that will probably be helpful. Here’s the code I used, where entry is a blogEntry:

```
{{ entry.date.strftime("%I:%M %p (UTC) %b %d, %Y") }}
```

Congratulations! You’ve just finished programming a website! If you want to make this blog even better, here are some ideas:

- Allow users to click on a post to see more details in a separate webpage.
- Add authentication so that only you can add posts.
- Deploy your website to an external server so that the whole world can see it. Tom Shen will be hosting a session on deployment soon.