Abby Pennington
LIS452: Final Project

Working with Django and Flask Web Development Frameworks

Great open source software almost always comes about because one or more clever developers had a problem to solve and no viable or cost effective solution available.

--Nigel George, *Mastering Django: Core*

DJANGO

History of Django

Django is a web development tool well suited to creating "content" sites such as Amazon.com, but is equally suitable for creating any type of website (George xxxiv). Django was developed by Adrian Holovaty and Simon Willison in 2003, and released as open source software with the contributions of Jacob Kaplan-Moss in 2005 (George xxxiv). It was created in response to increasingly complex website design at a time when each web page had to be written and revised manually (George xxxiii).

The impetus for the development Django is multifold. As mentioned above, programmers needed a more efficient method for creating and revising websites. This issue was addressed by the work of NCSA (the National Center for Supercomputing) which created the first graphical browser that let the "Web Server spawn external programs that could dynamically generate HTML" (George xxxiii). This protocol is called the Common Gateway Interface (CGI). CGI allows web developers to design dynamic websites--complex functionality, more easily revised, more engaging for users, and more discoverable-- that work with web pages as "resource generated on demand" rather than page by page as was necessary before ("Static vs. Dynamic Websites"; George xxxiii).

It is due to the complexities of CGI that PHP/FI (Personal Home Page/Forms Interpreter) emerged as a tool for simplifying the web development process (George xxxiii; Wikipedia). However, PHP is not without problems, not the least of which are that "its ease of use encourages sloppy, repetitive, ill-conceived code" and its security vulnerabilities (George xxxiv).

These shortcomings led to the creation of Django. At the time of its development, Holovaty and Willison were working as part of the web development team for the *Lawrence Journal-World* newspaper (George xxxiv). The chaotic environment of the news business demanded quick turnaround for website enhancements and revisions. In response, Holovaty and Willison developed Django (named after the jazz guitarist Django Reinhardt) (George xxxiii). The features that made Django so useful in the environment of the newsroom, make it well-suited to "content" sites such as cragslist.org that "offer dynamic, database-driven information" (George xxxiii).

Django: Functionality/Feature Overview

George notes that another important aspect of Django's history is its origins in "real-world code, rather than being an academic exercise or commercial product" and narrow focus on real-world web development issues (xxxv). As a result, Django enables developers to create "deep, dynamic, interesting sites in an extremely short time," and includes features such as:

- High-level abstractions of common web development patterns
- Shortcuts for frequent programming tasks, and
- Clear conventions on how to solve problems

(George (xxxv).

An object-oriented mapper enables users to describe their database layout using Python code. A dynamic administration interface--automatically created--"lets authenticated users add, change, and delete objects" (Django). Django enables "clean, elegant URL" schemes with "simple mapping between URL patterns and Python callback functions" (Django).

According to the Django website, the benefits of using this web development application include short "concept to launch" time, low hassle Web development, in a free, open source development environment. Django provides the core tools for Web development, including:

- User authentication
- Content administration
- Site maps
- Rss feeds
- Object-relational mapper (for database layout description)
- Data-model syntax (for representing models)
- Scalable

Django: Accessibility to New Users

I was not able to get a website up and running with Django. In fact, after attempting one online tutorial and two online book tutorials, I was unable to even get a quarter of the way through any of them. Part of the problem is that the tutorials and books are out of date and so don't reflect the latest versions of Python or Django.

FLASK

History of Flask

Flask is a web development tool that is referred to as a "micro framework" because it does not rely on "particular tools and libraries (*Wikipedia*) and is accessible to the learning user (Grinberg). It was developed by Armin Ronacher in 2004 of Pocoo, an international group of Python programmers. Pocoo built the foundation upon which Flask is based, Werkzeug Web Server Gateway Interface (WSGI) (an "interface between web servers and web applications or frameworks for the Python programming language"), and the Jinja2 template engine (which "provides Python-like expressions while ensuring that the templates are evaluated in a sandbox") ("Jinja (Template Engine)").

Flask: Functionality Overview

As mentioned, Flask is a micro or small framework. However, part of its power is derived from its extensibility atop a strong foundation of tools (Grinberg). For instance, services such as native support for accessing databases, “validating web forms, authenticating users, or other high-level tasks” (Grinberg) are met by “hooks to customize its behavior” (Ronacher). Users can add the most suitable extensions, thereby creating for a given project, a “lean stack that has no bloat and does exactly what you need” (Grinberg).

Due to its simple development process, Flask is a good fit for startups, which need to move from concept to product in a short period of time (Copperwaite, Leifer). Further, Flask has a robust set of libraries for facilitating design of the most common development task, including:

- URL routing that makes it easy to map URLs to your code
- Template rendering with Jinja2, one of the most powerful Python template engines
- Session management and securing cookies
- HTTP request parsing and flexible response handling
- Interactive web-based debugger
- Easy-to-use, flexible application configuration management

(Copperwaite, Leifer)

Flask: Accessibility to New Users

DuPlain explains that Flask provides “clear, simple interfaces packed with useful documentation.” While maintaining that Flask is fun to develop code with, it can also be challenging when working on large applications (DuPlain).

As with Django, due to outdated material, I was not able to complete the two book tutorials that I attempted. I got the furthest with the Lynda.com “Flask Tutorial,” detailed below.

USING THE LYNDA.COM FLASK TUTORIAL

In the end, the Lynda.com “Flask Tutorial” was best starting point for someone like me, a beginning programmer with a some Python coding experience. It has its own problems with being out of date in relation to the current software version, but has far fewer flaws than the other books and tutorials that I tried. While most appropriate for at least intermediate level programmers, it is still a worthwhile investment of time for those with at least some Python experience.

What I had hoped after finishing a tutorial is to develop a simple application that enabled user to develop regex skills. Both the scope and focus of the project changed. First, it seemed simpler to develop a quick look-up for BASH commands. Second, and most importantly, I am not at a point with Flask where I can develop the level of code that would be required to enable a user to click on a BASH command and, in response, display syntax details and examples, as I envisioned. That said, what is working is user registration, login, the display of some BASH commands to give a sense of what I envisioned, a new database, and the About page.

Instructions for Using the Website

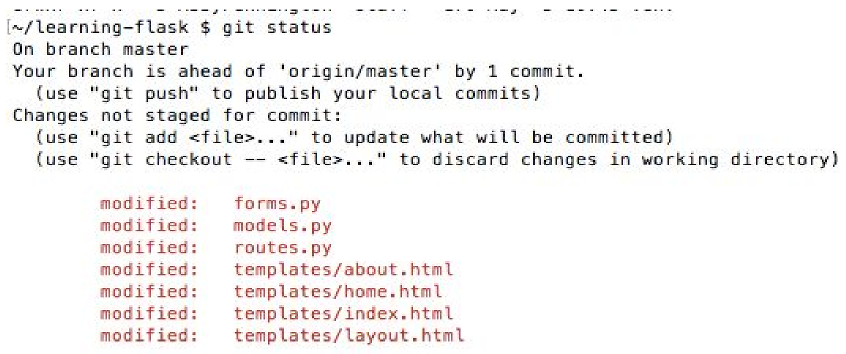
At the index page (which displays a “Signup” button and a “Learn More” button), click “Signup” to register or “Learn More” to read the brief about page. Once registered, you will be taken to the Home page (the search button is not functional).

About the Flask Development Environment

For the tutorial, in addition to Python, Flask, and HTML, there are three pieces of software utilized to provide for version control and backups, create and maintain databases, and develop and launch the website: Git, PostgreSQL, Heroku, respectively.

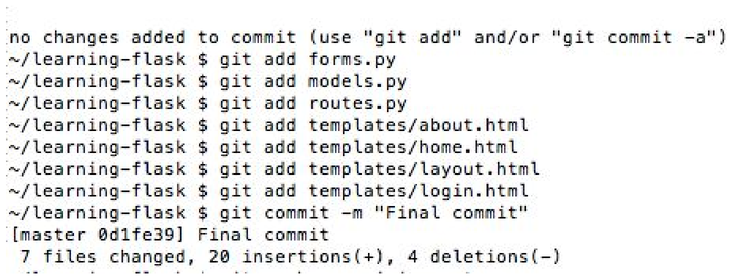
GitHub for Version Control and Backup

Git is configured and run in the Terminal. In the example below, the **git status** command displays a list of files that have been modified since the last backup (it would list new directories if there were any).



```
~/learning-flask $ git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   forms.py
        modified:   models.py
        modified:   routes.py
        modified:   templates/about.html
        modified:   templates/home.html
        modified:   templates/index.html
        modified:   templates/layout.html
```

The user can then use **git add** to specify which files need to be uploaded to GitHub. The **git commit -m “Name_of_Commit”** command assigns a title to the build and transfers the files to the GitHub repository, as shown below. To “push” these files to GitHub, **git push origin master** is executed.



```
no changes added to commit (use "git add" and/or "git commit -a")
~/learning-flask $ git add forms.py
~/learning-flask $ git add models.py
~/learning-flask $ git add routes.py
~/learning-flask $ git add templates/about.html
~/learning-flask $ git add templates/home.html
~/learning-flask $ git add templates/layout.html
~/learning-flask $ git add templates/login.html
~/learning-flask $ git commit -m "Final commit"
[master 0d1fe39] Final commit
 7 files changed, 20 insertions(+), 4 deletions(-)
```

PostgreSQL for Database Creation and Management

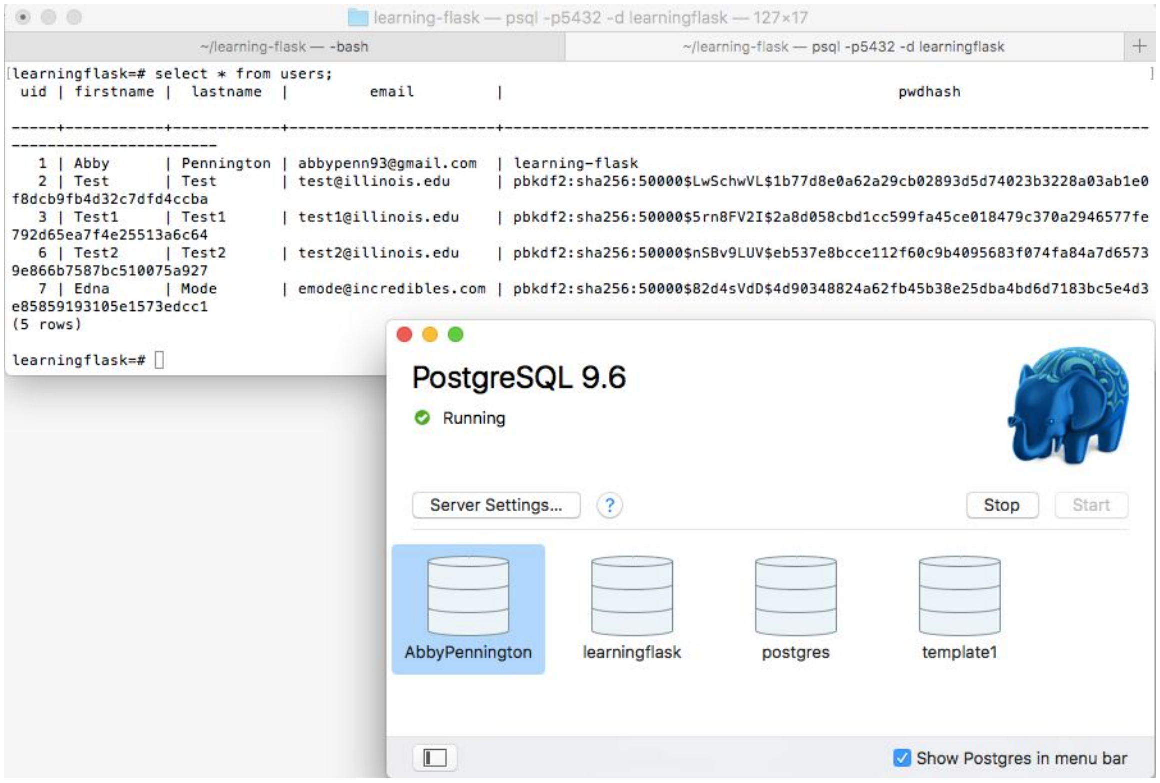
PostgreSQL is a relation database used to store the web application’s user information. This tutorial uses the PostgreSQL app for Mac. It is a simple process to create a database and define its schema:

```
CREATE TABLE users (uid serial PRIMARY KEY, firstname VARCHAR(100) not null,
firstname VARCHAR(100) not null, email VARCHAR(100) not null, email VARCHAR(100) not
null unique,
    pwdhash VARCHAR(100) not null);
```

VARCHAR sets the maximum length for the firstname, lastname and email address. The **unique** parameter ensures that each email address that is entered in the table is unique. All this user supplied data needs to be completed before the user can register for the site, as indicated by the **not null** parameter. **pwdhash** ensures that all passwords are encrypted before being stored in the database.

Once the schema is defined, the table can be populated, either by using the website or in the Terminal, as shown below.

```
INSERT INTO users (firstname, lastname, email, pwdhash) VALUES (Edna, Mode,
    ‘emode@theincredibles.org’, #######).
```



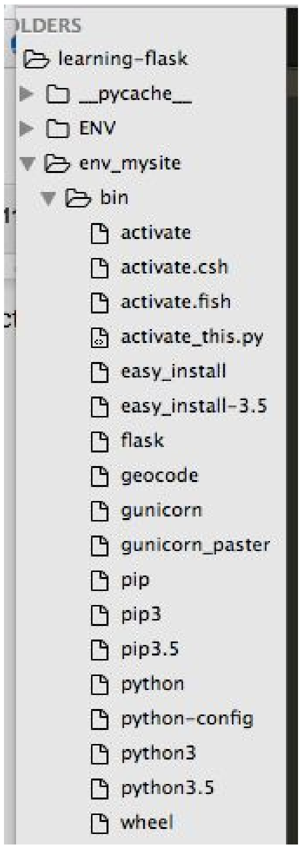
learning-flask — psql -p5432 -d learningflask — 127×17
~/learning-flask — -bash
~/learning-flask — psql -p5432 -d learningflask
PostgreSQL 9.6
Running
Server Settings...
?
Stop
Start
AbbyPennington
learningflask
postgres
template1
Show Postgres in menu bar

```
learningflask=# select * from users;
 uid | firstname |  lastname  |         email          |                                  pwdhash

-----+-----------+------------+------------------------+--------------------------------------------------------------------------
------------------------
   1 | Abby      | Pennington | abbypenn93@gmail.com   | learning-flask
   2 | Test      | Test       | test@illinois.edu      | pbkdf2:sha256:50000$LwSchwVL$1b77d8e0a62a29cb02893d5d74023b3228a03ab1e0
f8dcb9fb4d32c7dfd4ccba
   3 | Test1     | Test1      | test1@illinois.edu     | pbkdf2:sha256:50000$5rn8FV2I$2a8d058cbd1cc599fa45ce018479c370a2946577fe
792d65ea7f4e25513a6c64
   6 | Test2     | Test2      | test2@illinois.edu     | pbkdf2:sha256:50000$nSBv9LUV$eb537e8bcce112f60c9b4095683f074fa84a7d6573
9e866b7587bc510075a927
   7 | Edna      | Mode       | emode@incredibles.com  | pbkdf2:sha256:50000$82d4sVdD$4d90348824a62fb45b38e25dba4bd6d7183bc5e4d3
e85859193105e1573edcc1
(5 rows)

learningflask=# 
```

Heroku for Website Deployment Environment

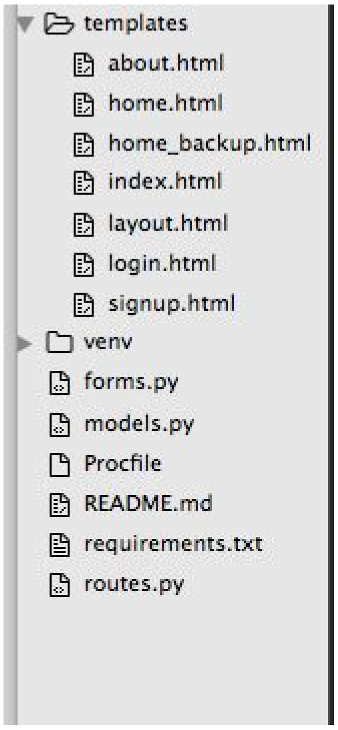
Heroku is a “container-based cloud Platform” for web application development and launch (“About Heroku”). **heroku create** creates a new sub-domain for the website. Heroku Toolbelt makes the website visible online. The screenshot below shows the URL created for this web app.

File Structure for Flask Projects



```
LDERS
learning-flask
  __pycache__
  ENV
  env_mysite
    bin
      activate
      activate.csh
      activate.fish
      activate_this.py
      easy_install
      easy_install-3.5
      flask
      geocode
      gunicorn
      gunicorn_paster
      pip
      pip3
      pip3.5
      python
      python-config
      python3
      python3.5
      wheel
```

The *env_mysite/bin* directory holds the local development environment. Keeping a local copy of the these files (what I believe are considered executables) enables developers to store only the modules that are pertinent to the project (reducing the likelihood of any overlap that might occur when unused modules are present and keeping the environment streamlined).



```
templates
  about.html
  home.html
  home_backup.html
  index.html
  layout.html
  login.html
  signup.html
venv
forms.py
models.py
Procfile
README.md
requirements.txt
routes.py
```

The *templates directory* holds all the .html files. Template files define the text and other visual data that is displayed on the pages of a website. The base .html file is *layout.html*. All the other html files augment *layout.html* and inherit the *layout.html* content.

- *forms.py* creates classes for each form, each with a list of fields, and each "represented by an object" (Grinberg 38).
- *models.py* maps classes to items in the database.
- *Procfile* installs the web server.
- *requirements.txt* holds a list of versions of the software used by the website.

- *routes.py* is the Terminal executable.

learning-flask/ Files

Forms.py defines the classes for the signup form, login form, address form (not functional), and the command form (not functional). For instance, the SignupForm class creates the fields for users to register for the site by entering their first name, last name, email address, and password. If no data is entered, the **validators** return a text string asking the user to complete the field. The actual layout of the forms is handled by the .html forms, outlined below.

models.py imports the SQLAlchemy so that the program can map classes to items in the database. A model in Flask refers to “persistent entities used by the application,” typically a Python class with “attributes that match the columns of a corresponding database table” (Grinberg). Variable db is assigned an instance of this class. The classes *User* and *Command* are defined (note that class Command isn’t fully functional so it is commented out; *Place* is not functional, due to the tutorial being out of sync with current software specifications).

As you can see, the *User* class consists of five fields which correspond to columns in the *users* table (assigned with __tablename__= 'users'). String length is assigned for the field that will store the first name, last name, email address, and password (e.g. the first_name field can have a maximum of 100 characters). The primary key (assigned automatically) is stored in the uid field. The set_password and check_password are Werkzeug "helpers" that generate and deal with encrypted passwords so that the real password is not stored directly in the database, making accounts vulnerable to hacking. A partial screenshot of *Models.py* is show above.

procfile starts up the web server (**web: gunicorn routes:app**).

requirements.txt simply specifies the version number for each software tool.

routes.py is the main program. First, it imports Flasks and Flask extensions, imports classes from models.py, and imports the form classes. An instance of the Flask class is created with app = Flask(__name__). On line 9, Flask is connected to the database storing user information, and after that, the flask application is initialized. App.secret_key creates secure forms (Lynda). @app.* creates web pages. For instance, @app.route("/") loads the first page that the user sees when the visit the site (configured in index.html). The same process establishes web pages for the About, Signup, Login (has not been developed), and Home pages. In the segment of route.py shown below, the **GET** option receives information from the browser and the **POST** receives and validates information received from the browser (DuPlain). The POST below is displaying the Signup form and the GET is receiving the new user information.

```
if request.method == "POST":
        if form.validate() == False:
                return render_template('signup.html', form=form)
        else:
                newuser = User(form.first_name.data, form.last_name.data, form.email.data,
form.password.data)
                db.session.add(newuser)
                db.session.commit()

                session['email'] = newuser.email
```

```
            return redirect(url_for('home'))

    elif request.method == "GET":
            return render_template('signup.html', form=form)
```

For the signup process, routes.py adds and commits the new user information to the database and loads the Home page. For the login process, if the user is already logged in, they are navigated to the Home page. If they are not logged in, their username and password are verified (I believe that's what **user = User.query.filter_by(email=email).first()** is doing). If either the email address or the password are incorrect, the user is returned to the login page.

@app.route('logout') deletes the cookie (session.pop('email', None), generated when they logged in and navigates the user back to the index page.

@app.route('home') checks to see if the user is logged in, and if not, moves the user to the login page.

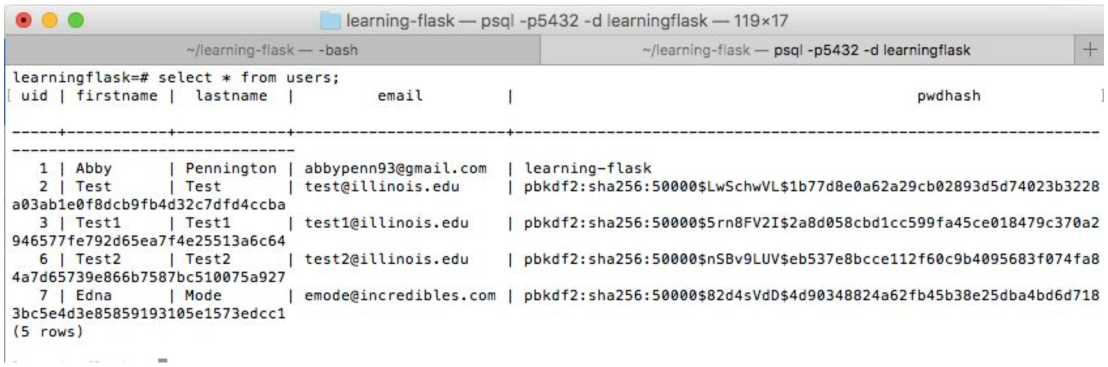
The commented out code is a rough draft of the BASH command syntax functionality. Upon navigating to the home page, and as mentioned earlier, the page would have a list of BASH commands that the user could click on to research their syntax and to reference examples. I had hoped to get the program to look up whatever command the user clicked on and post syntax and examples on that same page. It would first check to see if the command was in the database and then display the contents of the syntax and example fields from the database.

The last set of the code, for places, goes with the non-functional Lynda map portion of the web application.

Tables in the learning-flask Database

The *users* Table

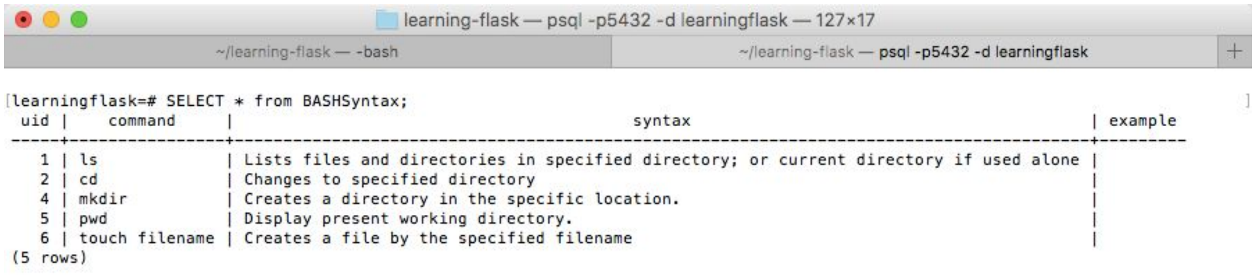
The table *users* is added to each time a user registers for the site.


learning-flask — psql -p5432 -d learningflask — 119×17
~/learning-flask — -bash
~/learning-flask — psql -p5432 -d learningflask

```
learningflask=# select * from users;
 uid | firstname |  lastname  |         email          |                              pwdhash

-----+-----------+------------+------------------------+---------------------------------------------------------------
------------------------------
   1 | Abby      | Pennington | abbypenn93@gmail.com   | learning-flask
   2 | Test      | Test       | test@illinois.edu      | pbkdf2:sha256:50000$LwSchwVL$1b77d8e0a62a29cb02893d5d74023b3228
a03ab1e0f8dcb9fb4d32c7dfd4ccba
   3 | Test1     | Test1      | test1@illinois.edu     | pbkdf2:sha256:50000$5rn8FV2I$2a8d058cbd1cc599fa45ce018479c370a2
946577fe792d65ea7f4e25513a6c64
   6 | Test2     | Test2      | test2@illinois.edu     | pbkdf2:sha256:50000$nSBv9LUV$eb537e8bcce112f60c9b4095683f074fa8
4a7d65739e866b7587bc510075a927
   7 | Edna      | Mode       | emode@incredibles.com  | pbkdf2:sha256:50000$82d4sVdD$4d90348824a62fb45b38e25dba4bd6d718
3bc5e4d3e85859193105e1573edcc1
(5 rows)
```

The *BASHSyntax* Table

If the BASH lookup functionality were to be fully implemented, this table would be used to lookup detailed syntax and examples for each BASH command.


learning-flask — psql -p5432 -d learningflask — 127×17
~/learning-flask — -bash
~/learning-flask — psql -p5432 -d learningflask

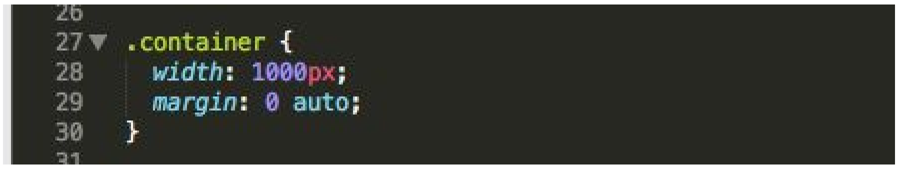
```
learningflask=# SELECT * from BASHSyntax;
 uid |    command     |                                  syntax                                   | example
-----+----------------+---------------------------------------------------------------------------+---------
   1 | ls             | Lists files and directories in specified directory; or current directory if used alone |
   2 | cd             | Changes to specified directory                                            |
   4 | mkdir          | Creates a directory in the specific location.                             |
   5 | pwd            | Display present working directory.                                        |
   6 | touch filename | Creates a file by the specified filename                                  |
(5 rows)
```

Static Files

learning-flask/static/css/main.css

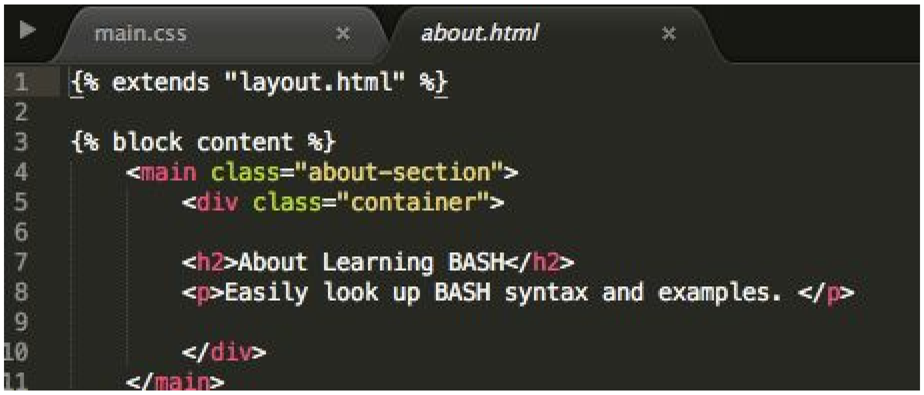
main.css defines the physical layout and appearance of the forms. For instance, in *about.html*, the first part of the code specifies **class= "container."** This command string references content in *main.css*, as shown below. This code specifies the location of "About Learning BASH" header and associated paragraph that is displayed below.

***main.css* (partial)**



```
.container {
  width: 1000px;
  margin: 0 auto;
}
```

about.html



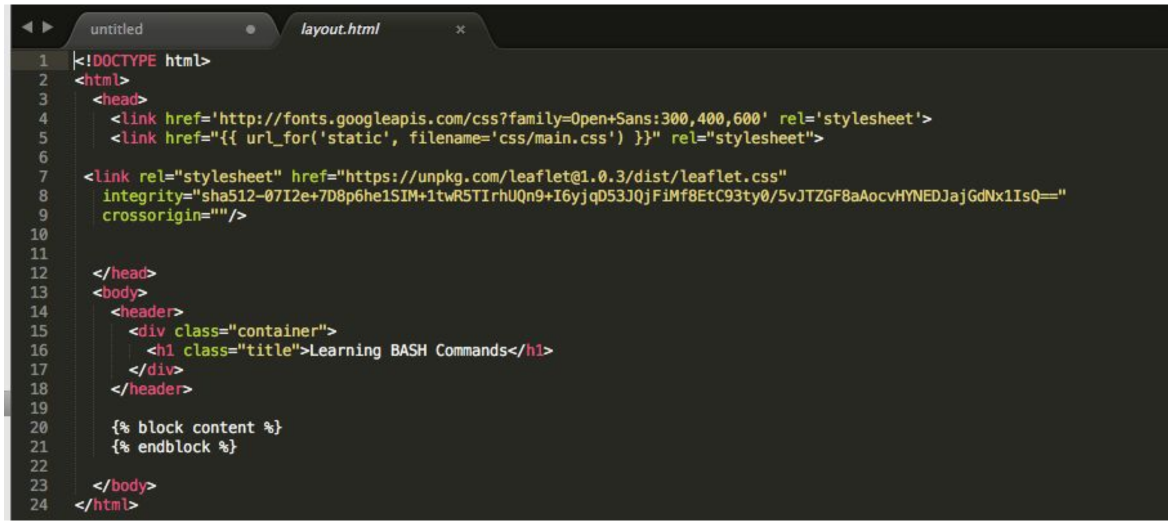
```
{% extends "layout.html" %}

{% block content %}
    <main class="about-section">
        <div class="container">

        <h2>About Learning BASH</h2>
        <p>Easily look up BASH syntax and examples. </p>

        </div>
    </main>
```

learning-flask/static/js and learning-flask/static/img

The js folder is empty and the img folder holds the map that the Lynda Flask Tutorial home page displays.

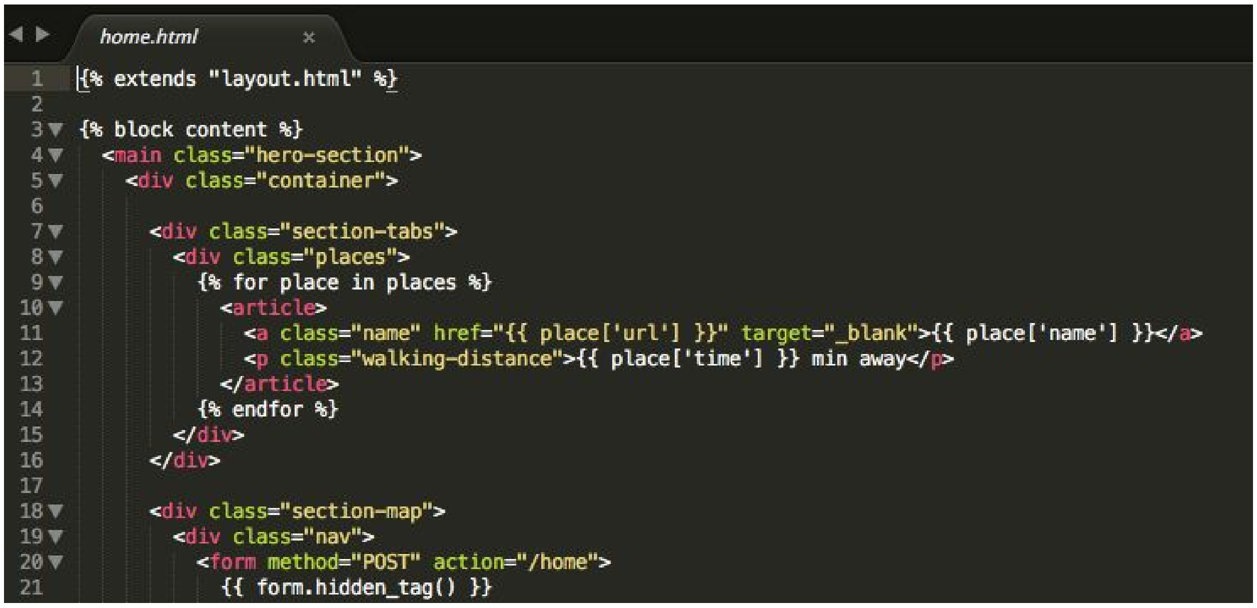
Template Files

Template files contain the text and other visual data that is displayed on the pages of a website. As mentioned, layout.html file (shown below) defines the basic elements for the entire website. DuPlain says that base templates allow the developer to “write focused pages which share a common structure and style.” For instance, title text “Learning BASH Commands” appears on all the pages of the website. It references *main.css* for how data should be displayed and where it should be located. This is exemplified by the **class = “container”** command string; the formatting for the contents of this block will follow the specification in home.css (discussed more below).



```
<!DOCTYPE html>
<html>
  <head>
    <link href='http://fonts.googleapis.com/css?family=Open+Sans:300,400,600' rel='stylesheet'>
    <link href="{{ url_for('static', filename='css/main.css') }}" rel="stylesheet">

 <link rel="stylesheet" href="https://unpkg.com/leaflet@1.0.3/dist/leaflet.css"
   integrity="sha512-07I2e+7D8p6he1SIM+1twR5TIrhUQn9+I6yjqD53JQjFiMf8EtC93ty0/5vJTZGF8aAocvHYNEDJajGdNx1IsQ=="
   crossorigin=""/>


  </head>
  <body>
    <header>
      <div class="container">
        <h1 class="title">Learning BASH Commands</h1>
      </div>
    </header>

    {% block content %}
    {% endblock %}

  </body>
</html>
```

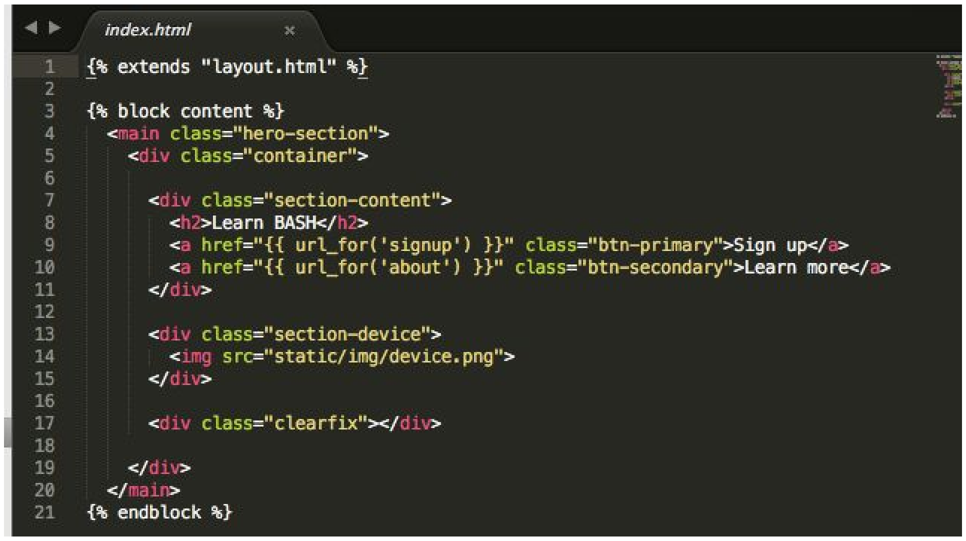
home.html extends *layout.html* by defining text and a window for data entry. It is originally set up to allow the user to enter an address. The Lynda Flask Tutorial was meant to search for business near a supplied address. The text defined below, showing BASH commands and descriptions does not have any correlating functionality. What I had imagined is creating list of commands that could be clicked on to expand into a full explanation of syntax and examples.



home.html

```
{% extends "layout.html" %}

{% block content %}
  <main class="hero-section">
    <div class="container">

      <div class="section-tabs">
        <div class="places">
          {% for place in places %}
            <article>
              <a class="name" href="{{ place['url'] }}" target="_blank">{{ place['name'] }}</a>
              <p class="walking-distance">{{ place['time'] }} min away</p>
            </article>
          {% endfor %}
        </div>
      </div>

      <div class="section-map">
        <div class="nav">
          <form method="POST" action="/home">
            {{ form.hidden_tag() }}
```

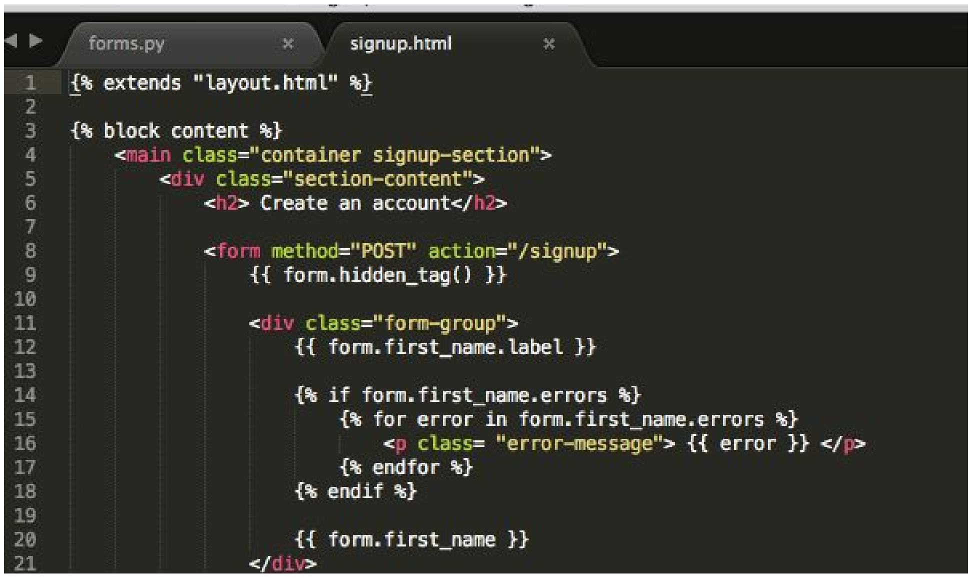
index.html creates the Signup and About buttons (e.g., **class= "btn-primary">Sign up>/a>**)on the initial page for the website. Clicking on "Signup" button loads *signup.html* and clicking on the "Learn more" button loads *about.html*.



index.html

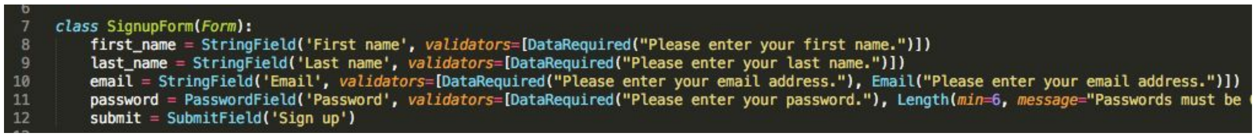
```
{% extends "layout.html" %}

{% block content %}
  <main class="hero-section">
    <div class="container">

      <div class="section-content">
        <h2>Learn BASH</h2>
        <a href="{{ url_for('signup') }}" class="btn-primary">Sign up</a>
        <a href="{{ url_for('about') }}" class="btn-secondary">Learn more</a>
      </div>

      <div class="section-device">
        <img src="static/img/device.png">
      </div>

      <div class="clearfix"></div>

    </div>
  </main>
{% endblock %}
```

signup.html allows the user to create an account. This form contains the user name fields and the email and password fields. The label for each field, specified by **form.field_name.label** is referenced from the SignupForm class in *forms.py*. For example, **form.first_name.errors** uses the label held in **StringField ('First name')**. If a field is empty when the user presses the "Submit" button, an error message, sourced from **validators=[DataRequired**("Please enter your email address."] in *forms.py*, is displayed. A section of *signup.html* is shown here.



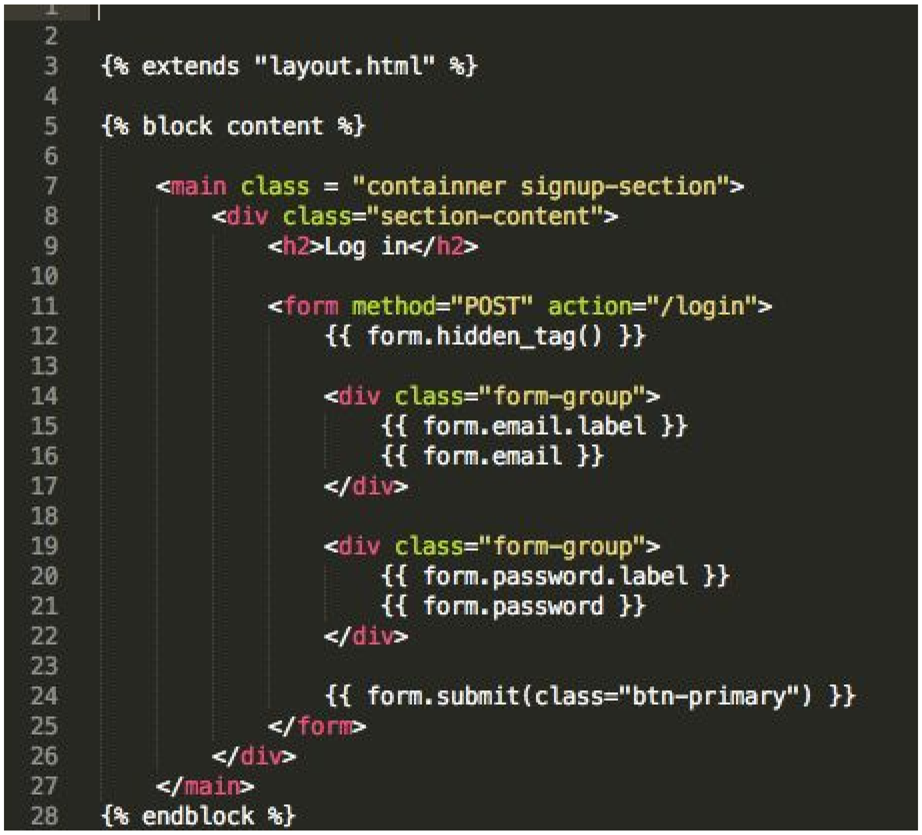
```
{% extends "layout.html" %}

{% block content %}
    <main class="container signup-section">
        <div class="section-content">
            <h2> Create an account</h2>

            <form method="POST" action="/signup">
                {{ form.hidden_tag() }}

                <div class="form-group">
                    {{ form.first_name.label }}

                    {% if form.first_name.errors %}
                        {% for error in form.first_name.errors %}
                            <p class= "error-message"> {{ error }} </p>
                        {% endfor %}
                    {% endif %}

                    {{ form.first_name }}
                </div>
```

Here is the corresponding code for the signup process in *forms.py*.



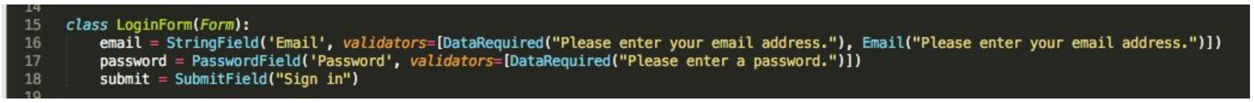
```
class SignupForm(Form):
    first_name = StringField('First name', validators=[DataRequired("Please enter your first name.")])
    last_name = StringField('Last name', validators=[DataRequired("Please enter your last name.")])
    email = StringField('Email', validators=[DataRequired("Please enter your email address."), Email("Please enter your email address.")])
    password = PasswordField('Password', validators=[DataRequired("Please enter your password."), Length(min=6, message="Passwords must be
    submit = SubmitField('Sign up')
```

login.html defines the appearance of the login page. It creates fields for the registered user to enter their email address and password. Similar to the "Signup" form, this form references the *forms.py,* in this case using the Loginform class, to display the appropriate text and error messages for each field.



```

{% extends "layout.html" %}

{% block content %}

    <main class = "containner signup-section">
        <div class="section-content">
            <h2>Log in</h2>

            <form method="POST" action="/login">
                {{ form.hidden_tag() }}

                <div class="form-group">
                    {{ form.email.label }}
                    {{ form.email }}
                </div>

                <div class="form-group">
                    {{ form.password.label }}
                    {{ form.password }}
                </div>

                {{ form.submit(class="btn-primary") }}
            </form>
        </div>
    </main>
{% endblock %}
```

Here is the corresponding code for login in *forms.py*.



```python
class LoginForm(Form):
    email = StringField('Email', validators=[DataRequired("Please enter your email address."), Email("Please enter your email address.")])
    password = PasswordField('Password', validators=[DataRequired("Please enter a password.")])
    submit = SubmitField("Sign in")
```

Annotated Bibliography

Aggarwal, Shalabh. *Flask Framework Cookbook*. Packt Publishing, 2014. *Safari Books Online*. Web. 7 May 2017.

The audience for this book is experienced Flask programmers who want to develop applications and "scale them with industry-standard practices" (Aggarwal). This book is most appropriate for developers who are already familiar with Flask's major extensions. The "cookbook" aspect of the book centers on "recipes" that help the reader gain an understanding of Flask's powerful features and extensions. It also covers debugging and error handling.

Bissex, Paul, Jeff Forcier, and Wesley Chun. *Python Web Development with Django*. Addison-Wesley Professional, 2008. *Safari Books Online*. Web. 7 May 2017.

Python Web Development with Django is directed at experienced Python coders who want to learn Flask. It hits the ground running by guiding the reader through the process of building a blog application (rather than the canonical "Hello, World" project). It then moves into core Flask concept including models, forms, views, and templates, working with URLs and HTTP mechanisms. Lastly, it guides the reader through the steps of developing a content management system, photo gallery, and pastebin.

Cirella, David. "Going the Third Way: Developing Custom Software Solutions for Your Library." *Computers in Libraries.* vol. 36 no. 2, Mar. 2016, pp. 12–16. search.ebscohost.com/login.aspx?direct=true&db=a9h&AN=113772831&site=ehost-live. Accessed 7 May 2017.

This article addresses the often lengthy and expensive process for extending library web services. The author proposes using web development applications such as Flask to develop and implement tailored solutions to common tasks, such as generating a listing of reserved items.

Copperwaite, Matt. Charles Leifer. *Learning Flask Framework*. Packt Publishing. 26 Nov. 2015. proquest.safaribooksonline.com.proxy2.library.illinois.edu/book/web-design-and-development/9781783983360. Accessed 10 Apr. 2017.

Directed at any level programmer, but best suited to those with at least intermediate level Python coding experience. Flask employs Python design principles which can simplify the process of learning web development. This book uses a tutorial to teach the core components of Flask, including templates, views, URL routes, models, extensions, inheritance, and static files. It also covers advanced issues, including security, debugging, testing, relational databases, Ajax requests, and APIs. *Learning Flask Framework* offers readers a blogging and tagging sample project.

"Declaring Models." *Flask*. //flask-sqlalchemy.pocoo.org/2.1/models/. Accessed 11 May 2017.

Detailed explanation of Web application models used for mapping classes to items in a database. This webpage provides a simple example and descriptions of how to define one-to-one, one-to-many, and many-to-many relationships.

"Django at a Glance." *Django*. docs.djangoproject.com/en/1.11/intro/overview/. Accessed 25 Apr. 2017.

This webpage provides an overview of how to develop a database-driven Web application using Django. It provides the technical specifications for how to design and install a database schema, access the database content using the Python API, administrative interface (for user registration and authentication), and designing URLs, templates, static files, and views.

DuPlain, Ron. *Instant Flask Web Development*. Packt Publishing. 26 Aug. 2013. proquest.safaribooksonline.com.proxy2.library.illinois.edu/book/web-design-and-development/9781783983360. Accessed 15 Apr. 2017.

Comprehensive guide with instructions on how to install Flask, create a simple website, route URLs, work with databases and records, design and work with templates, and work with user input. The table of contents contains notes about which chapters are appropriate for "simple," "intermediate", and "advanced" users.

"Flask" *Wikipedia*. 20 Jan. 2017. en.wikipedia.org/wiki/Flask_(web_framework). Accessed 27 Apr. 2017.

"Flask Web Development, One Drop at a Time." 21 Mar. 2017. flask.pocoo.org/docs/0.12/.latex/Flask.pdf. Accessed 10 May 2017.

This detailed document is the user guide to Flask Web application development. It represents a compilation of the Flask webpages. This detailed instruction is most appropriate for programmers experienced in resolving problems that arise due to different development environments and cryptic error messages. Covers all the core Flask concepts, including templates, views, models, and forms, with heavy emphasis on developing secure applications.

George, Nigel. *Mastering Django: Core*. GNW Independent Publishing. 2016.

Comprehensive Django guide with installation instructions, basic instructions on how to create a simple website, develop templates, perform database queries, work with URLs, and create more complex websites. Appropriate for the experienced programmer.

Grinberg, Miguel. *Flask Web Development*. O'Reilly. 2014.

Guide with installation instructions, basic instructions on how to create a simple website, an overview of application structures, example Web applications, and instructions for how to deploy websites. Appropriate for users with some Python experience.

"Jinja (Template Engine)" *Wikipedia*. 21 Apr. 2017. en.wikipedia.org/wiki/Jinja_(template_engine). Accessed 27 Apr. 2017.

"Learning Flask." *Lynda.com.* 27 Accessed Apr. 2017.

A video Flask tutorial for users with Python, SQL, HTML, and CSS experience. This tutorial provides detailed explanations of Flask functionality and implementations, along with helpful visualizations. It has a few flaws due to outdated documentation, however it is a worthwhile investment for the interested Flask learner.

Perras, Joël. *Flask Blueprints*. Packt Publishing, 2015. *Safari Books Online*. Accessed 7 May 2017.

This book guides those with some Python and Flask development experience through the steps of creating more complex web applications. The reader is presented with Flask projects that increase in complexity using Flask extensions and external Python libraries. The guide employs virtualenv to establish an isolated development environment, Flask-Login extension, Flaks-WFT (for forms), and Flask-SQLAlchemy (database interactions). It addresses how to build Jinja templates, timeline application, testing, and version control using GitHub. Finally, the reader will learn how to develop a "photo timeline application" that employs running Celery tasks.

"PHP." *Wikipedia*. 27 Apr. 2017. en.wikipedia.org/wiki/PHP#History. Accessed 27 Apr. 2017.

"Static Vs Dynamic websites – what's the difference?." 22 Feb. 2016. *EDInteractive*. edinteractive.co.uk/static-vs-dynamic-websites-difference/. Accessed 24 Apr. 2017.

Brief article comparing static websites with dynamic websites, with emphasis on pros and cons of developing each.

Stouffer, Jack. *Mastering Flask*. Packt Publishing, 2015. *Safari Books Online*. proquest.safaribooksonline.com.proxy2.library.illinois.edu/9781784393656. Accessed 7 May 2017.

Resource for web development using Flask, Python, Git (for version control), SQLAlchemy, and Alembic (database migrations). It provides a "step-by-step" guide with emphasis on best coding practices. It provides readers with guidance on how to develop a custom Flask extension and create asynchronous tasks. Advanced topics include transforming an app to utilize a

Model-View-Controller (MVC) architecture, user registration and login, unit testing, a lengthy discussion of the different platforms that Flask apps can be deployed on.

“Tutorial.” *Flask*. flask.pocoo.org/docs/0.12/tutorial/. Accessed 30 Apr. 2017.
A step-by-step guide to developing Flask Web applications from the developers of Flask. Addresses Flask project file structure, basic components such as templates, models, views, databases, as well as adding formatting and testing a simple application. Best suited to experienced programmers, with some knowledge of Web development. Note that some aspects of the tutorial are outdated due to changes in Flask functionality or syntax.

“Web Server Gateway Interface.” *Wikipedia*. 6 Apr. 2017.
en.wikipedia.org/wiki/Web_Server_Gateway_Interface. Accessed 27 Apr. 2017.