# Flask and Intro to Deployment

Fall 2018 – Introduction to Full-Stack Development

Instructor: Matthew Fritter

matt@frit.me

# Stepping Back: Flask so far

- In Lab 6, we installed Flask and configured Apache to serve our application
  - Configuring Passenger to help act as a middleman in serving our Flask application
  - Configuring an Apache VirtualHost to route our web traffic towards Passenger
  - We covered the basics of routing and templates in Flask to serve basic dynamic information and static HTML pages

# Stepping Back: Flask so far

- In Lab 7, we covered some more of the capabilities that the Flask framework offers
  - How to pass data from our application to our templates using the Jinja templating system, and using the Markup() function to allow HTML to be passed
  - We looked at how to handle routes with variable names, and how to use those variables within our route function
  - Defining specific GET and POST routes
  - Handling request data from forms via POST requests

# Stepping Back: Flask so far

- With what we've covered so far, and a basic knowledge of the Python programming language, we should be able to build simple applications in the Flask framework
  - Having a knowledge of multiple web frameworks makes you more employable, and also makes learning further frameworks easier – the more experience you have, the more that carries over to other frameworks
  - Today, we'll be covering some of the more specialized functionalities within Flask

# Announcements

- I'll be handing back the marked quizzes for quiz 1 and quiz 2 tomorrow in the lab
- We'll be having a third quiz on the Flask framework next class (Tuesday, Nov 13th)
  - This will be closed book, and will focus on practical knowledge. It will be focused on the application, not configuration.
- Any questions on what we've covered in Flask so far?

# Additional Flask Functionality

# Handling JS/CSS Files

- Obviously, any serious web application is likely going to have additional CSS and JS elements incorporated into it for styling and user interaction
  - This poses a question – there is no /public folder that Flask serves out of like there is in Laravel – so how do we serve CSS and JS files that are used in our web pages?

# ★ Handling JS/CSS Files

- It's actually quite straightforward:
  - Create a folder called '`static`' within your Flask application directory. This is a specially named folder, like the `templates` folder, which Flask will automatically look for.
  - Place your static CSS/JS files within the `static` folder
  - To generate the URL for these files, you can use the url_for() function (must be imported), which accepts an endpoint name, and a filename:

```
url_for('static', filename='mystyle.css')
```

  - Keep in mind that the filename must match *exactly*. This will generate a string URL to the `style.css` file that you can then pass on to a template.
  - You can use this function within templates as well, using {{ }}

# ★ Handling Redirects

- Oftentimes, it will be useful to redirect from one route to another. For example: returning the user to the home page after successfully logging in.
  - This can be easily done using the `redirect()` function, which takes a URL input. Note you must import `redirect` from `flask`:

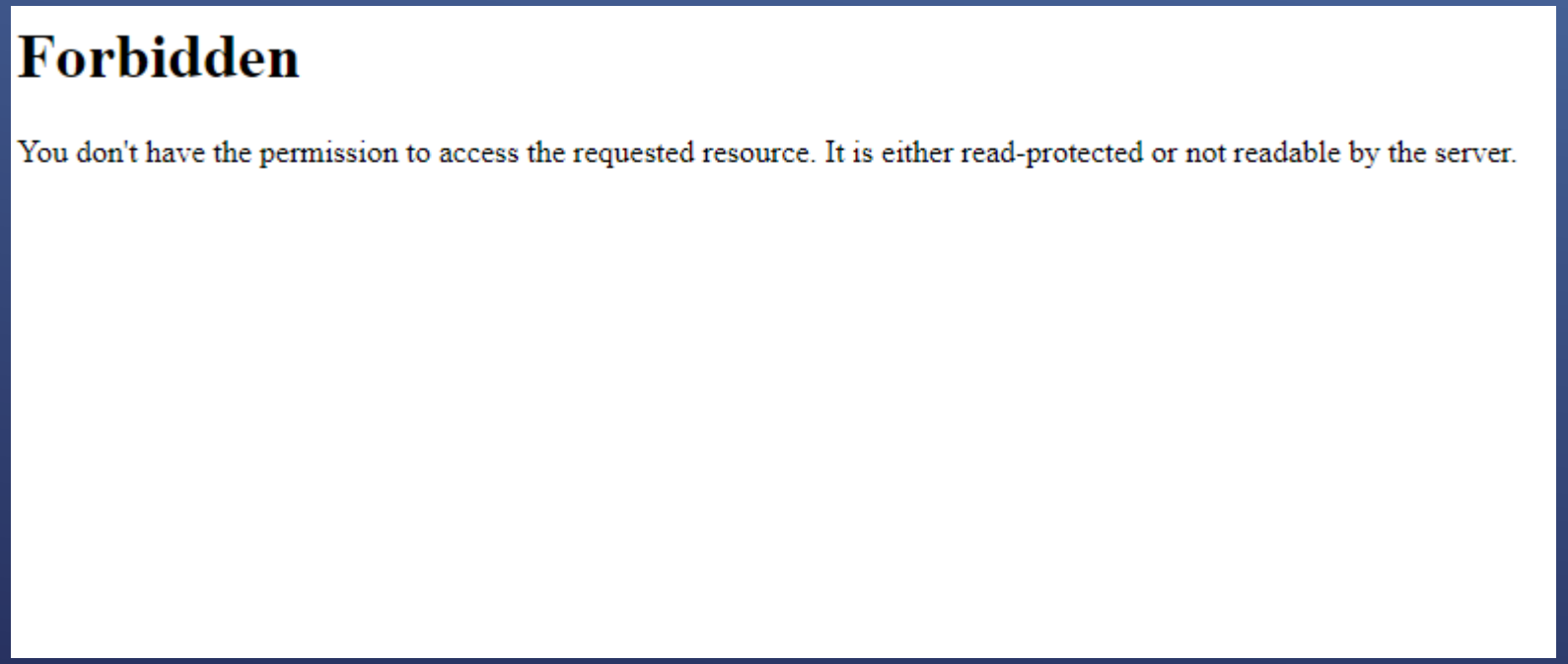
```
return redirect('/')
```

    - This would return the user to the home (root) page of the website
- You can also forcibly return error codes using the `abort()` function, such as if a form fails validation, or a user tries to access a restricted page. Like redirect, abort must be imported from flask. An abort looks like this:

```
abort(403)
```

# Pretty Error Pages

- By default, the `abort()` function will return pretty bland, boring error pages, similar to default Apache error pages:



# Pretty Error Pages

- Obviously, that's not a great look for a website. Especially for errors like 404 errors, which can be fairly common.
- We can create a special route that will handle a specific error, allowing us to define a custom template or action when that error occurs:

```
@MyApp.errorhandler(403)
def forbiddenerror():
    return render_template('403_page.html'), 403
```

- The 403 in the final statement after the render_template() call makes sure that the page is returned with a 403 status number
  - While this might not be particularly important for users, it's important to include the status number so that bots can identify what's happened and not archive the error-producing page

# ★ Sessions

- We previously looked at the concept of sessions in Laravel – they make up an integral part of many web applications
  - To rehash, the session is a user-specific set of data that *is stored between requests to the server!*
- Flask also includes Sessions, although it stores the session client-side in an encrypted cookie, rather than server side as a file, cache, or database entry
- Like most things in flask, you'll need to import `session` from the `flask` module at the top of the application script

# ★ Sessions – Encryption Key

- In order to make use of sessions, we'll need to supply an application encryption key. This is a secret key that will be used to sign cookies
  - Note: the encryption key itself is not used to sign, but rather a key that is a derivative of the secret key will be used
- This is done by simply adding this line, near the top of your file but below the `app = Flask(__name__)` line:
  - `MyApp.secret_key = "weRf46H6b3ld!3"`
  - This defines the secret key for your application. Note that if this key is compromised, it should be changed immediately – failure to do so could expose user sessions to malicious use.

# ★ Sessions – Data I/O

- Adding data to a session is extremely easy in Flask. Simply call session as if it was a variable with a specific key, and assign a value:

```
session['login'] = "frit"
```

- It's that easy!
- Removing a set session variable is also very easy. Simply use the session.pop() function to remove the variable:

```
session.pop('login', None)
```

  - Note, if you delete a session variable and then try to perform an action with the session variable, it will return an internal server error
- Once you've added the session variable, it will be available between requests. Note that because the sessions are cookie-based, deleting cookies client-side *will destroy all session variables*

# Flashing Data

- In Laravel, we looked briefly at the concept of *flashing* data with sessions, which is simply the case of temporarily holding onto some data until the request is made
- In Flask, flashing is done through the `flash` function, which must be imported from `flask`

```
flash('Login failed.')
```

  - This will save the message 'Login failed.' until the next request is made to the server.
- Like the Laravel equivalent, this system is designed for handily passing small bits of data between pages, such as notifications or warning messages

# Accessing Flash Data

- Flashed data can be accessed directly through the `get_flashed_messages()` function, which will return all flashed messages as list
- Most likely, you'll want to access it directly in a Jinja template, which you can do using something similar to the following:

```
{% with errors = get_flashed_messages() %}
    {% if errors %}
        {% for message in errors %}
            <p>{{message}}</p>
        {% endfor %}
    {% endif %}
{% endwith %}
```

# ★ Template Inheritance

- In Laravel, we saw that we could use the Blade templating system to create inheritance in templates using `@yield`
  - This is very useful to create a single "wrapper" of HTML that we can inject content into, allowing our pages to maintain identical elements without having to include them in each page
- This can also be done in Flask using the Jinja `block`
  - Concept: create a named block in our parent template, and then have the child inherit from the parent template and specify what content should go within that block

# ★ Template Inheritance - Example

main.html

```
<html>
        <head>
<title>{% block title %} {% endblock %}</title>
        </head>
        <body>
{% block content %} {% endblock %}
        </body>
</html>
```

child.html

```
{% extends “main.html” %}
{% block title %}
Homepage
{% endblock %}
{% block content %}
<h1>Homepage</h1>
<p>Welcome to my homepage</p>
{% endblock %}
```

# Databases in Flask

- You may notice that we haven't discussed accessing and manipulating databases in Flask yet
  - This is because handling databases in Flask is identical to the handling of databases in Python – i.e. quite simple
- Python comes by default with the sqlite3 module, which allows for very easy use of a local SQLite database
- Flask provides some helper functions, but they are largely unnecessary in this context

# Databases in Flask – sqlite3 example

```
import sqlite3
con = sqlite3.connect(‘test.db’)
cur = con.cursor()
cur.execute(‘’‘create table users (username text, pass text, email text)’’’)
cur.execute(“insert into users values (‘frit’,’p4ssw0rd’,’matt@frit.me’)”)
con.commit()
con.close()
```

# Databases in Flask – sqlite3 example

```
import sqlite3
con = sqlite3.connect('test.db')
cur = con.cursor()
for row in c.execute('select * from users')
       print(row[0])
con.close()
```

# Databases in Flask – Other options

- Since we're using Python, there are a multitude of other database options available to us
  - The MySQLdb module can be used to connect to local and remotely hosted MySQL databases
  - The Psycopg module allows you to use a PostgreSQL database
  - pyMongo is the official Python database driver for the MongoDB database, which is a NoSQL platform
- Options are basically only limited by your comfort level with different types of database – Experiment!

# Questions

- Does anyone have any remaining questions about Flask?
- Does anyone want to see any of the examples we looked at in class again?
- We'll be finishing up Flask here – You should have all the major components required to build a dynamic web application in Flask. Obviously we're not covering the entire depth of the Flask framework, but this is something that could have an entire course dedicated to it.
  - Note that the Flask documentation is very good, so if you want to know more, I highly recommend that you take a look

# Intro to Deployment

# ★ What is Deployment?

- Simply put, deployment is the act of making our website available
- This encompasses a whole range of requirements depending on the software in use:
  - Installing and configuring backend components (Apache, PHP/Flask/etc, databases)
  - Acquiring the codebase and/or database to be used
  - Ensuring that everything functions correctly
  - Doing this all as quickly and painlessly as possible

# Considering Potential Deployments

- Here are some deployment strategies, in terms of general worst-case to best-case. Note however this is only in general terms, and some situations may require different deployment strategies:
  1. Manually configuring components and copying over files from your desktop to your server via SSH/SFTP
  2. Manually configuring components and pulling your codebase from GitHub
  3. Automatically configuring components via script, automatically pulling changes from a clean GitHub build branch on a regular basis
  4. Using containerization/virtualization solution such as Docker

# ★ Why Deployment is Important

- Deployment is all about *scale*
- Manually configuring and deploying a codebase to a single server can be time consuming, but is ultimately fairly trivial
- Manually configuring and deploying a codebase to a hundred servers is a nightmare!
  - Automating this deployment process means less time spent typing the same commands into a console over and over, and more time writing code and being productive
  - Businesses are actively looking for people with skills in deployment and version control – writing good code is worthless if you can't take it live in an efficient manner

# Being Prepared

- Even if you don't think you need to think about a deployment solution, you probably should
- If your site goes viral for some reason, traffic is going to increase exponentially – this could mean a great deal of profit for you, through advertising revenue or sales
  - However, that revenue is lost if your site goes down – a very probable scenario if you're running a small server
  - By having a fast deployment option before you launch, you help ensure that you'll be able to keep up with traffic by deploying additional instances of your application through a load balancer

# ★ Using Git

- Git is a near-ubiquitous version control software native to the Linux environment, but available for both Windows and Mac platforms as well
- Many, if not all of you, have probably used it before
- Besides being useful for version control, Git can also provide a powerful means of rolling out changes to your code, or quickly deploying your codebase on a new server
- In particular, the use of online repository systems like GitHub allows for codebases to be quickly downloaded, and allows update rollouts to be performed in an automated manner

# ★ Using Git – A Refresher

- For those who haven't used Git in a while, or have never used Git, a brief refresher:
  - A repository is a directory in which Git is tracking changes to files (edits, creation of new files, deletions, etc)
  - A commit is saving the changes made to the repository at the current point in time. Commits can be rewound (undone).
  - A push is when the commits you have saved locally are forwarded to a remote repository, such as GitHub. The remote repository should then reflect your current repository.
  - A pull is when changes are downloaded from a remote repository to the local environment.

# ★ Using Git – The General Workflow

- The general workflow for doing full-stack development in Git using an online repository is as follows:
  1. Develop locally in a virtual machine or native server OS first, **committing** your changes locally.
  2. **Push** your commits to the online repository
  3. **Pull** your commits down from the online repository onto your server
- The benefits of this workflow are that you aren't working directly on your production server – you can catch errors and problems locally, before releasing your code to the public

# ★ Using Git – More Depth

- Often, you'll want to separate your repository into branches
  - You'll want a dev branch, which contains your newest, utterly broken code
  - You'll also want a production or stable branch, which contains the code that is currently considered "production" quality – error free and safe for release
  - Once you've tested your dev branch code satisfactorily and fixed any lingering problems, you can merge your dev branch into the production branch, and then pull the changes down onto your server
- It's a good idea to keep untested development code away from the actual production code

# ★ Using Git – Automation

- Git offers benefits even if you don't use automation, but it can be an extremely useful deployment tool combined with good automation
- First, you'll probably want to include some sort of script (i.e. a bash script) within your repository root that can be used to set up the operating environment (install web server software, update dependencies, etc)
- Secondly, you'll probably want to add a script that automatically performs a pull on the production branch of your Git on a timed interval
  - For example, a script that performs a pull every day at midnight – you can then guarantee that your server codebase is never out-of-date by more than 24 hours, when compared to the stable/production branch

# Shortcomings of Git + Automation

- Using Git and automation scripts for deployment do have some shortcomings:
  - It's usually dependent on the operating system. An automation script designed to run on CentOS will likely error in Ubuntu
  - Git can at times be unwieldy, especially if you have multiple people working on a project and there is potential for conflicts between commits
  - Your automation files might be susceptible to changes in dependencies, and for complex systems they might be untenable

# What Addresses these Shortcomings?

- Using virtualization or containerization systems!
- These would be your Docker, Solaris containers, etc
- Next lecture: discussion of Docker, containerization, and load balancing for your server

# Tomorrow's Lab

- Tomorrow we'll be covering some of the more in-depth functions of Flask that we discussed today.
- We'll also be taking a look at deployment – I recommend that if you don't have a GitHub account, that you go sign up for one now – you'll need it for the lab.

# That's All Folks!