Lecture 7

Last times

- Last time, we learned about Python, a programming language that comes with many features and libraries. Today, we'll use Python to generate HTML for webpages, and see how separations of concerns might be applied.
- A few weeks ago, we learned about web requests in HTTP, which might look like this:

```
GET / HTTP/1.1
Host: www.example.com
...
```

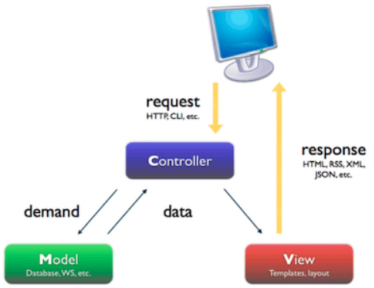
  - Hopefully, a server responds with something like:

```
HTTP/1.1 200 OK
Content-Type: text/html
...
```

    - The `...` is the actual HTML of the page.

Flask

- Today, we'll use Flask, a microframework, or a set of code that allows us to build programs without writing shared or repeated code over and over. (Bootstrap, for example, is a framework for CSS.)
- Flask is written in Python and is a set of libraries of code that we can use to write a web server in Python.
- One methodology for organizing web server code is MVC, or Model-View-Controller:



  - Thus far, the programs we've written have all been in the Controller category, whereby we have logic and algorithms that solve some problem and print output to the terminal. But with web programming, we also want to add formatting and aesthetics (the View component), and also access data in a more organized way (the Model component). When we start writing our web server's code in Python, most of the logic will be in the controllers.
  - By organizing our program this way, we can have separation of concerns.
- Today, we'll build a website where students can fill out a form to register for Frosh IMs, freshman year intramural sports.
- We can start by opening the CS50 IDE, and write some Python code that is a simple web server program, `serve.py`:

```
from http.server import BaseHTTPRequestHandler, HTTPServer

class HTTPServer_RequestHandler(BaseHTTPRequestHandler):

    def do_GET(self):

        self.send_response(200)

        self.send_header("Content-type", "text/html")
        self.end_headers()

        self.wfile.write(b"<!DOCTYPE html>")
        self.wfile.write(b"<html lang='en'>")
        self.wfile.write(b"<head>")
        self.wfile.write(b"<title>hello, title</title>")
        self.wfile.write(b"</head>")
        self.wfile.write(b"<body>")
        self.wfile.write(b"hello, body")
        self.wfile.write(b"</body>")
        self.wfile.write(b"</html>")


port = 8080
server_address = ("0.0.0.0", port)
httpd = HTTPServer(server_address, HTTPServer_RequestHandler)

httpd.serve_forever()
```

  - We already know how to write a hello, world HTML page, but now we're writing a program in Python to actually generate and return an HTML page.
  - Most of this code is based on the `http` library that we can import that handles the HTTP layer, but we have written our own `do_GET` function that will be called every time we receive a GET request. As usual, we need to look at the documentation for the library to get a sense of what we should write, and what we have available for us. First, we send a 200 status code, and send the HTTP header indicating that this is an HTML page. Then, we write (as ASCII bytes) some HTML, line by line, into the response.
  - Notice that we set the server to use port 8080 (since the IDE itself is using port 80), and actually create and start the server (based on

- Notice that we set the server to use port 8080 (since the IDE itself is using port 80), and actually create and start the server (based on documentation we found online).
- Now, if we run `python serve.py`, we can click CS50 IDE > Web Server, which will open our IDE's web server in another tab for us, and we'll see the hello, world page we just wrote.

- We can see that reimplementing many common functions of a web server can get tedious, even with an HTTP library, so a framework like Flask helps a lot in providing abstractions and shortcuts that we can reuse.
- With Flask, we can write the following in an `application.py` file:

```
from flask import Flask, render_template, request

app = Flask(__name__)

@app.route("/")
def index():
    return "hello, world"
```

  - With `app = Flask(__name__)`, we initialize a Flask application for our `application.py` file. Then, we use the `@app.route("/")` syntax to indicate that the function below will respond to any requests for `/`, or the root page of our site. We call that function `index` by convention, and it will just return "hello, world" as the response, without any HTML.
  - Now, we can call `flask run` from the terminal in the same folder as our `application.py`, and the resulting URL will show a page that reads "hello, world" (which our browser displays even without HTML).
- We can change the `index` function to return a template, or a file that has HTML that we've written, that acts as the View.

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

  - In a `templates` folder, we'll have an `index.html` file with the following:

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <meta name="viewport" content="initial-scale=1, width=device-width">
        <title>hello</title>
    </head>
    <body>
        hello,
    </body>
</html>
```

  - We see a new feature, ``, like a placeholder. So we'll go back and change the logic of `index`, our controller, to check for parameters in the URL and pass them to the view:

```
return render_template("index.html", name=request.args.get("name", "world"))
```

    - We use `request.args.get` to get a parameter from the request's URL called `name`. (The second argument, `world`, will be the default value that's returned if one wasn't set.) Now, we can visit `/?name=David` to see "hello, David" on the page. Now, we can generate an infinite number of webpages, even though we've only written a few lines of code.
- In `froshims0`, we can write an `application.py` that can receive and respond to a POST request from a form:

```
from flask import Flask, render_template, request

app = Flask(__name__)


@app.route("/")
def index():
    return render_template("index.html")


@app.route("/register", methods=["POST"])
def register():
    if not request.form.get("name") or not request.form.get("dorm"):
        return render_template("failure.html")
    return render_template("success.html")
```

  - For the default page, we'll return an `index.html` that contains a form:

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

{% block body %}
    <h1>Register for Frosh IMs</h1>
    <form action="/register" method="post">
        <input autocomplete="off" autofocus name="name" placeholder="Name" type="text">
        <select name="dorm">
            <option disabled selected value="">Dorm</option>
            <option value="Apley Court">Apley Court</option>
            <option value="Canaday">Canaday</option>
            <option value="Grays">Grays</option>
            <option value="Greenough">Greenough</option>
            <option value="Hollis">Hollis</option>
            <option value="Holworthy">Holworthy</option>
            <option value="Hurlbut">Hurlbut</option>
            <option value="Lionel">Lionel</option>
            <option value="Matthews">Matthews</option>
            <option value="Mower">Mower</option>
            <option value="Pennypacker">Pennypacker</option>
            <option value="Stoughton">Stoughton</option>
            <option value="Straus">Straus</option>
            <option value="Thayer">Thayer</option>
            <option value="Weld">Weld</option>
            <option value="Wigglesworth">Wigglesworth</option>
        </select>
        <input type="submit" value="Register">
    </form>
{% endblock %}
```

        - We have an HTML form, with an `input` tag for a student to type in their name, and a `select` tag to create a dropdown list for them to select a dorm. Our form will be submitted to a route we call `/register`, and we'll use the POST method to send the form's information.
        - Notice that our template is now using a new feature, `extends`, to define blocks that will be substituted themselves in another file, `layout.html`:

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <meta name="viewport" content="initial-scale=1, width=device-width">
        <title>froshims0</title>
    </head>
    <body>
        {% block body %}{% endblock %}
    </body>
</html>
```

            - Now, if we have other pages on our site, they can easily share the common markup we would want on every page. The `{% block body %}{% endblock %}` syntax is a placeholder block in Flask, where other pages, like `index.html`, can provide HTML that will be substituted into that block.
        - In our `register` function, we'll indicate that we're listening for a POST request, and inside the function, just make sure that we got a value for both `name` and `dorm`. `request.form` is an abstraction provided by Flask, such that we can access the arguments, or parameters, from the request's POST data.
- When we run our application with `flask run`, and visit the URL, sometimes we might see an Internal Server Error. And if we come back to our terminal, where our Flask server is running, we'll see an error message that provides us clues to what went wrong. We can press Control+C to stop our web server, make changes that will hopefully fix our error, and start our web server again. And even if nothing is broken but we made a change, sometimes we need to quit Flask and start it again, for it to notice those changes.
- We also need a `success.html` and `failure.html` in our `templates` directory, which might look like:

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

{% block body %}
    You are registered! (Well, not really.)
{% endblock %}
```

    - Our `register` function will return that, with the template fully rendered, if we provided both a name and dorm in the form.
    - With `layout.html`, we didn't need to copy and paste the same `<head>` and other shared markup, making it easier for us to make changes across all the pages we have at once.
- The failure page, too, will share the same layout but send a different message:

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

{% block body %}
    You must provide your name and dorm!
{% endblock %}
```

    - The `{% %}` syntax is actually called Jinja, a templating language that Flask is able to understand and put together.
- And all of this Python code lives on our server in the CS50 IDE, generating a completed HTML page each time and sending it to the browser as a response. We can see that by right-clicking the page in Chrome, clicking View Source, and seeing the full HTML that users will get.
- Now let's actually do something with the submitted form information. In `froshims1/application.py`, we'll create a list to store all the registered students:

```
from flask import Flask, redirect, render_template, request

# Configure app
app = Flask(__name__)

# Registered students
students = []


@app.route("/")
def index():
    return render_template("index.html")


@app.route("/registrants")
def registrants():
    return render_template("registered.html", students=students)


@app.route("/register", methods=["POST"])
def register():
    name = request.form.get("name")
    dorm = request.form.get("dorm")
    if not name or not dorm:
        return render_template("failure.html")
    students.append(f"{name} from {dorm}")
    return redirect("/registrants")
```

    - We create an empty list, `students = []`, and when we get a name and dorm in `register`, we'll use `students.append(f"{name} from {dorm}")` to add a formatted string with that name and dorm, to the `students` list.
    - In the `registrants` function, we'll pass in our `students` list to the template of `registered.html`:

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

```
{% block body %}
    <ul>
        {% for student in students %}
            <li>{{ student }}</li>
        {% endfor %}
    </ul>
{% endblock %}
```

  - Notice that, with Jinja, we can have simple concepts like a `for` loop to generate HTML based on variables passed into the template. (We need an `endfor` since, in HTML, indentation is only needed for stylistic purposes, so we need to specify when a loop ends.) Here, we're creating an `<li>` for each `student`, or string, in the `students` variable that was passed in by the controller, `application.py`. And notice that the markup, or formatting of the list, is in this template, or view.

- If we stop our server, and restart it, we'll have lost all of the data we've collected, since the `students` variable is only created and stored as long as our program is running.
- In `froshims2/application.py`, we use a new library:

```
import os
import smtplib
from flask import Flask, render_template, request

# Configure app
app = Flask(__name__)


@app.route("/")
def index():
    return render_template("index.html")


@app.route("/register", methods=["POST"])
def register():
    name = request.form.get("name")
    email = request.form.get("email")
    dorm = request.form.get("dorm")
    if not name or not email or not dorm:
        return render_template("failure.html")
    message = "You are registered!"
    server = smtplib.SMTP("smtp.gmail.com", 587)
    server.starttls()
    server.login("jharvard@cs50.net", os.getenv("PASSWORD"))
    server.sendmail("jharvard@cs50.net", email, message)
    return render_template("success.html")
```

  - The SMTP (Simple Mail Transfer Protocol) library allows us to use abstractions for sending email, and here, every time we get a valid form, we'll send an email. By reading the documentation for `smtplib` and for Gmail, we can figure out the lines of code needed to log in to Gmail's server programmatically, and send an email to the email address from our form.

- We can also save the registration data to a CSV on our server, which can then be opened even after our server is stopped:

```
from flask import Flask, render_template, request
import csv

app = Flask(__name__)


@app.route("/")
def index():
    return render_template("index.html")


@app.route("/register", methods=["POST"])
def register():
    if not request.form.get("name") or not request.form.get("dorm"):
        return render_template("failure.html")
    file = open("registered.csv", "a")
    writer = csv.writer(file)
    writer.writerow((request.form.get("name"), request.form.get("dorm")))
    file.close()
    return render_template("success.html")


@app.route("/registered")
def registered():
    file = open("registered.csv", "r")
    reader = csv.reader(file)
    students = list(reader)
    return render_template("registered.html", students=students)
```

  - We import the `csv` library, and open a file called `registered.csv` to append or read from. If we received a form in the `register` route, we'll open the file with `a`, to append. Then, we create a `csv.writer` (based on the documentation for the library), and use the `writerow` function to write the name and dorm to the file. Finally, we'll close the file.
  - The `registered` route will open the file for reading, and create a list of lists based on the file. Then, in `registered.html`, we can iterate over each list in the list (each row), and print the first item (the name) and the second item (the dorm):

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

{% block body %}
    <h1>Registered</h1>
    <ul>
        {% for student in students %}
            <li>{{ student[0] }} from {{ student[1] }}</li>
        {% endfor %}
    </ul>
{% endblock %}
```

- With a language we'll look at next week, SQL, we'll be able to work with data more easily than we can with a CSV file.

- In `froshims6/templates/index.html`, we use JavaScript in our template to check the input immediately:

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

{% block body %}
    <h1>Register for Frosh IMs</h1>
    <form action="/register" method="post">
        <input autocomplete="off" autofocus name="name" placeholder="Name" type="text">
        <select name="dorm">
            <option disabled selected value="">Dorm</option>
            <option value="Apley Court">Apley Court</option>
            <option value="Canaday">Canaday</option>
            <option value="Grays">Grays</option>
            <option value="Greenough">Greenough</option>
            <option value="Hollis">Hollis</option>
            <option value="Holworthy">Holworthy</option>
            <option value="Hurlbut">Hurlbut</option>
            <option value="Lionel">Lionel</option>
            <option value="Matthews">Matthews</option>
            <option value="Mower">Mower</option>
            <option value="Pennypacker">Pennypacker</option>
            <option value="Stoughton">Stoughton</option>
            <option value="Straus">Straus</option>
            <option value="Thayer">Thayer</option>
            <option value="Weld">Weld</option>
            <option value="Wigglesworth">Wigglesworth</option>
        </select>
        <input type="submit" value="Register">
    </form>

    <script>

        document.querySelector('form').onsubmit = function() {
            if (!document.querySelector('input').value) {
                alert('You must provide your name!');
                return false;
            }
            else if (!document.querySelector('select').value) {
                alert('You must provide your dorm!');
                return false;
            }
            return true;
        };

    </script>

{% endblock %}
```

  - With JavaScript on the page, the user can get feedback immediately since it runs in the browser. And we should still validate the input on our server, since someone might disable JavaScript or try to send bad requests programmatically. With libraries like Bootstrap, we can make validation pretty and really improve a user's experience, or UX.
  - In this example, we have a function that will be called when the `form` on the page is submitted, and checks that there's a value for both the `input` and the `select`. If there is no value for one of them, we'll create an alert and `return fallse` to stop the form from being submitted. Otherwise, our function will `return true` if both are present, allowing the form to be submitted by the browser.
  - We could also factor out the JavaScript code into a `.js` file and include it, but since we don't have very many lines of code yet, we can make a design decision to include our JavaScript code directly in our template. Frameworks like React will organize view code, like the HTML and JavaScript, in particular ways, so that we can maintain consistent patterns in more complicated web applications.

Words

- Let's create a website where someone can search for words that start with some string, much like how we might want to have autocomplete. We'll need a file called `large` that's a list of dictionary words, and in `words0/application.py` we'll have:

```
from flask import Flask, render_template, request

app = Flask(__name__)

WORDS = []
with open("large", "r") as file:
    for line in file.readlines():
        WORDS.append(line.rstrip())

@app.route("/")
def index():
    return render_template("index.html")


@app.route("/search")
def search():
    words = [word for word in WORDS if word.startswith(request.args.get("q"))]
    return render_template("search.html", words=words)
```

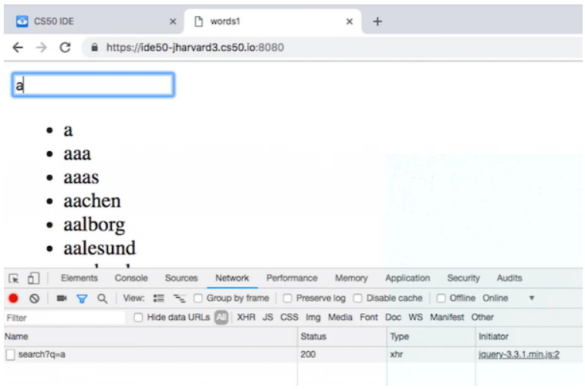
  - When our server starts, we'll create a `WORDS` list from reading in each line of the `large` file, removing the new line with `rstrip`, and storing that in our list.
  - In our `index` function, we'll render `index.html`, which is just a form:

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

{% block body %}
    <form action="/search" method="get">
        <input autocomplete="off" autofocus name="q" placeholder="Query" type="text">
        <input type="submit" value="Search">
    </form>
{% endblock %}
```

        - Our form will use the `get` method, since we want the query to be in the URL.
    - In our `search` route, we create a list, `words`, which is a list of every `word` in our global `WORDS` list (that we read in earlier) that start with the value of the parameter `q`. It's equivalent to:

```
words = []
q = request.args.get("q")
for word in WORDS:
    if word.startswith(q):
        words.append(word)
```

        - Once we have a list of words that match, we'll pass it to our template, `search.html` that will display each one with markup.
    - We can run our server with `flask run`, and when we visit the URL, we see a form that we can type some input into. If we type in the letter `a` or `b`, we can click submit and be taken to a page with all the words in our dictionary that start with `a` or `b`. And we notice that our route is something like `/search?q=a`, though we could have changed `q` (for query) to anything we'd like. We can even change the URL with some other value for `q`, and see our results displayed.
- In `words1`, we'll get the results list immediately with JavaScript. And we can infer how that example works, before looking at the code, by running it in the IDE. We can visit the URL, and use the Network tab in Developer Tools by right-clicking the page in Chrome:



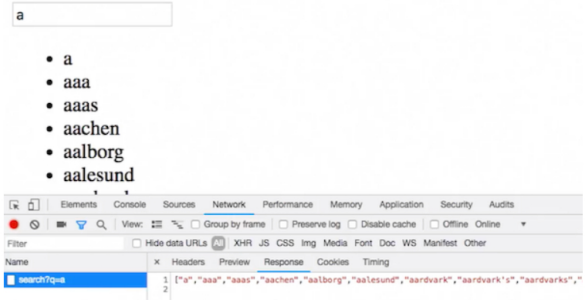
    - We see that our browser is making a request every time we type into the input box, and if we click on the request and then Response, we can see that our browser got some fragment of HTML with our results.
- We can click on View Source on the page, and see that our page has a bit of JavaScript after the HTML:

```
<input autocomplete="off" autofocus placeholder="Query" type="text">

<ul></ul>

<script src="https://code.jquery.com/jquery-3.3.1.min.js"></script>
<script>

    let input = document.querySelector('input');
    input.onkeyup = function() {
        $.get('/search?q=' + input.value, function(data) {
            document.querySelector('ul').innerHTML = data;
        });
    };

</script>
```

    - Here, we're using a JavaScript library called jQuery, which provides us with some abstractions. We're selecting the `input` element, and every time the `keyup` event occurs, we want to change the page. The `keyup` event will happen when we press a key in the input box, and let go. We use jQuery's `$.get` function to make a GET request to our server at the `/search?q=` route, with the value of the input box appended. When we get some `data` back, the `$.get` function will call an anonymous function (a callback) to set the `innerHTML` of the `ul` on our page to that `data`.
    - And notice that we provided an empty opened and closed `<ul>` element in our template, but we'll change the HTML inside with what our server responds with.
- On our server-side code, our `search` route is the mostly the same as before, but the template, `search.html`, will only have `<li>` elements, one for each matching word:

```
{% for word in words %}
    <li>{{ word }}</li>
{% endfor %}
```

    - Since we don't extend a `layout.html`, this route will only return an incomplete fragment of HTML. But that still works because our JavaScript code is putting it inside a complete page, our `index.html`.
- With `words2`, we have our server return data more efficiently, in a format called JSON, JavaScript Object Notation:



    - Then, in our JavaScript code on the page, we'll write each of them as an `<li>`, generating the markup in the browser instead of on our server.
    - The Python code in `application.py` uses a `jsonify` function to return a list as a JSON object:

```
@app.route("/search")
def search():
    q = request.args.get("q")
    words = [word for word in WORDS if q and word.startswith(q)]
    return jsonify(words)
```

- And our `index.html` has the JavaScript to append each word as an `<li>` element:

```
let input = document.querySelector('input');
input.onkeyup = function() {
    $.get('/search?q=' + input.value, function(data) {
        let html = '';
        for (word of data) {
            html += '<li>' + word + '</li>';
        }
        document.querySelector('ul').innerHTML = html;
    });
};
```

- In fact, since the browser can run JavaScript that can search a list, we can write all of this in JavaScript, without making a request to a server:

```
let input = document.querySelector('input');
input.onkeyup = function() {
    let html = '';
    if (input.value) {
        for (word of WORDS) {
            if (word.startsWith(input.value)) {
                html += '<li>' + word + '</li>';
            }
        }
    }
    document.querySelector('ul').innerHTML = html;
};
```

- When we get input from the user, we'll just iterate over a `WORDS` array and append any `word` string that starts with the input's value to the page as an `<li>` element.
- We'll also have to include a `large.js` file that creates that global variable, `WORDS`, which starts with the following:

```
let WORDS = [
  "a",
  "aaa",
  "aaas",
  "aachen",
  "aalborg",
  "aalesund",
  "aardvark",
  ...
```

- Even with a relatively simple example, we see how there can be a few different approaches to solving the same problem. With version 0, our server sent back entire, complete pages on every search. With version 1, we used JavaScript to make requests without navigating to another page, getting back data with markup from the server. With version 2, we used JavaScript, but only got back data from the server, that we then marked up in the browser. Finally, with version 3, we used JavaScript and the word list to accomplish the same results, but all within the browser. Each approach has pros and cons, so depending on what tradeoffs we value, one solution might be better than the rest.