This is CS50

CS50’s Introduction to Computer Science

OpenCourseWare

Donate (https://cs50.harvard.edu/donate)

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan) (https://www.instagram.com/davidjmalan/) (https://www.linkedin.com/in/malan/) (https://www.reddit.com/user/davidjmalan) (https://www.threads.net/@davidjmalan) (https://twitter.com/davidjmalan)

Lecture 9

Welcome!

- In previous weeks, you have learned numerous programming languages, techniques, and strategies.
- Indeed, this class has been far less of a *C class* or *Python class* and far more of a *programming class*, such that you can go on to follow future trends.
- In these past several weeks, you have learned *how to learn* about programming.
- Today, we will be moving from HTML and CSS into combining HTML, CSS, SQL, Python, and JavaScript so you can create your own web applications.

Static to Dynamic

- Up until this point, all HTML you saw was pre-written and static.
- In the past, when you visited a page, the browser downloaded an HTML page, and you were able to view it.
- Dynamic pages refer to the ability of Python and similar languages to create HTML files on-the-fly. Accordingly, you can have web pages that are generated by options selected by your user.
- You have used `http-server` in the past to serve your web pages. Today, we are going to utilize a new server that can parse out a web address and perform actions based on the URL provided.

Flask

- *Flask* is a third-party library that allows you to host web applications using the Flask framework within Python.
- You can run flask by executing `flask run`.
- To do so, you will need a file called `app.py` and a folder called `templates`.
- To get started, create a folder called `templates` and create a file called `index.html` with the following code:

```html
<!DOCTYPE html>

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

Notice the double `{{ name }}` that is a placeholder for something that will be later provided by our Flask server.

- Then, in the same folder that the `templates` folder appears, create a file called `app.py` and add the following code:

```
# Greets user

from flask import Flask, render_template, request

app = Flask(__name__)


@app.route("/")
def index():
    return render_template("index.html", name=request.args.get("name", "world"))
```

  Notice that this code defines `app` as the Flask application. Then, it defines the `/` route of `app` as returning the contents of `index.html` with the argument of `name`. By default, the `request.args.get` function will look for the `name` being provided by the user. If no name is provided, it will default to `world`.

- Finally, add a final file in the same folder as `app.py` called `requirements.txt` that has only a single line of code:

```
Flask
```

  Notice only `Flask` appears in this file.

- You can run this file by typing `flask run` in the terminal window. If Flask does not run, ensure that your syntax is correct in each of the files above. Further, if Flask will not run, make sure your files are organized as follows:

```
/templates
    index.html
app.py
requirements.txt
```

  Once you get it running, you will be prompted to click a link. Once you navigate to that webpage, try adding `?name=[Your Name]` to the base URL in your browser's URL bar.

- Improving upon our program, we know that most users will not type arguments into the address bar. Instead, programmers rely upon users to fill out forms on web pages. Accordingly, we can modify index.html as follows:

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <meta name="viewport" content="initial-scale=1, width=device-width">
        <title>hello</title>
    </head>
    <body>
        <form action="/greet" method="get">
            <input autocomplete="off" autofocus name="name" placeholder="Name" ty
            <button type="submit">Greet</button>
        </form>
    </body>
</html>
```

Notice that a form is now created that takes the user's name and then passes it off to a route called `/greet`.

- Further, we can change `app.py` as follows:

```
# Adds a form, second route

from flask import Flask, render_template, request

app = Flask(__name__)


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


@app.route("/greet")
def greet():
    return render_template("greet.html", name=request.args.get("name", "world"))
```

Notice that the default path will display a form for the user to input their name. The `/greet` route will pass the `name` to that web page.

- To finalize this implementation, you will need another template for `greet.html` as follows:

```
<!DOCTYPE html>

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

Notice that this route will now render the greeting to the user, followed by their name.

Layout

- Both of our web pages, `index.html` and `greet.html`, have much of the same data. Wouldn't it be nice to allow the body to be unique, but copy the same layout from page to page?
- First, create a new template called `layout.html` and write code as follows:

  ```
  <!DOCTYPE html>

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

  Notice that the `{% block body %}{% endblock %}` allows for the insertion of other code from other HTML files.
- Then, modify your `index.html` as follows:

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

  {% block body %}

      <form action="/greet" method="get">
          <input autocomplete="off" autofocus name="name" placeholder="Name" type="
          <button type="submit">Greet</button>
      </form>

  {% endblock %}
  ```

  Notice that the line `{% extends "layout.html" %}` tells the server where to get the layout of this page. Then, the `{% block body %}{% endblock %}` tells what code to be inserted into `layout.html`.
- Finally, change `greet.html` as follows:

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

  {% block body %}
      hello, {{ name }}
  {% endblock %}
  ```

  Notice how this code is shorter and more compact.

POST

- You can imagine scenarios where it is not safe to utilize `get`, as usernames and passwords would show up in the URL.
- We can utilize the method `post` to help with this problem by modifying `app.py` as follows:

```
# Switches to POST

from flask import Flask, render_template, request

app = Flask(__name__)


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


@app.route("/greet", methods=["POST"])
def greet():
    return render_template("greet.html", name=request.form.get("name", "world"))
```

  Notice that `POST` is added to the `/greet` route, and that we use `request.form.get` rather than `request.args.get`.
- This tells the server to look *deeper* in the virtual envelope and not reveal the items in `post` in the URL.
- Still, this code can be advanced further by utilizing a single route for both `get` and `post`. To do this, modify `app.py` as follows:

```
# Uses a single route

from flask import Flask, render_template, request

app = Flask(__name__)


@app.route("/", methods=["GET", "POST"])
def index():
    if request.method == "POST":
        return render_template("greet.html", name=request.form.get("name", "world
    return render_template("index.html")
```

  Notice that both `get` and `post` are done in a single routing. However, `request.method` is utilized to properly route based upon the type of routing requested by the user.

Frosh IMs

- Frosh IMs or *froshims* is a web application that allows students to register for intermural sports.
- Create a folder by typing `mkdir froshims` in the terminal window. Then, type `cd froshims` to browse to this folder. Within, create a directory called templates by typing `mkdir templates`. Finally, type `code app.py` and write code as follows:

```
# Implements a registration form using a select menu

from flask import Flask, render_template, request

app = Flask(__name__)

SPORTS = [
    "Basketball",
    "Soccer",
    "Ultimate Frisbee"
]


@app.route("/")
def index():
    return render_template("index.html", sports=SPORTS)


@app.route("/register", methods=["POST"])
def register():

    # Validate submission
    if not request.form.get("name") or request.form.get("sport") not in SPORTS:
        return render_template("failure.html")

    # Confirm registration
    return render_template("success.html")
```

  Notice that a `failure` option is provided, such that a failure message will be displayed to the user if the `name` or `sport` field is not properly filled out.

- Next, create a file in the `templates` folder called `index.html` by typing `code templates/index.html` and write code as follows:

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

{% block body %}
    <h1>Register</h1>
    <form action="/register" method="post">
        <input autocomplete="off" autofocus name="name" placeholder="Name" type="
        <select name="sport">
            <option disabled selected>Sport</option>
            {% for sport in sports %}
                <option value="{{ sport }}">{{ sport }}</option>
            {% endfor %}
        </select>
        <button type="submit">Register</button>
    </form>
{% endblock %}
```

- Next, create a file called `layout.html` by typing `code templates/layout.html` and write code as follows:

```
<!DOCTYPE html>

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

- Fourth, create a file in templates called `success.html` as follows:

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

{% block body %}
    You are registered!
{% endblock %}
```

- Finally, create a file in templates called `failure.html` as follows:

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

{% block body %}
    You are not registered!
{% endblock %}
```

- You can imagine how we might want to accept the registration of many different registrants. We can improve `app.py` as follows:

```
# Implements a registration form, storing registrants in a dictionary, with error

from flask import Flask, redirect, render_template, request

app = Flask(__name__)

REGISTRANTS = {}

SPORTS = [
    "Basketball",
    "Soccer",
    "Ultimate Frisbee"
]


@app.route("/")
def index():
    return render_template("index.html", sports=SPORTS)


@app.route("/register", methods=["POST"])
def register():

    # Validate name
    name = request.form.get("name")
    if not name:
        return render_template("error.html", message="Missing name")

    # Validate sport
    sport = request.form.get("sport")
    if not sport:
        return render_template("error.html", message="Missing sport")
    if sport not in SPORTS:
        return render_template("error.html", message="Invalid sport")

    # Remember registrant
    REGISTRANTS[name] = sport

    # Confirm registration
    return redirect("/registrants")


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

Notice that a dictionary called `REGISTRANTS` is used to log the `sport` selected by `REGISTRANTS[name]`. Also, notice that `registrants=REGISTRANTS` passes the dictionary on to this template.

- Further, create a new template called `registrants.html` as follows:

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

{% block body %}
    <h1>Registrants</h1>
    <table>
        <thead>
            <tr>
                <th>Name</th>
                <th>Sport</th>
            </tr>
        </thead>
        <tbody>
            {% for name in registrants %}
```

```
                <tr>
                    <td>{{ name }}</td>
                    <td>{{ registrants[name] }}</td>
                </tr>
            {% endfor %}
        </tbody>
    </table>
{% endblock %}
```

Notice that `{% for name in registrants %}...{% endfor %}` will iterate through each of the registrants. Very powerful to be able to iterate on a dynamic web page!

- Executing `flask run` and entering numerous names and sports, you can browse to `/registrants` to view what data has been logged.
- You now have a web application! However, there are some security flaws! Because everything is client-side, an adversary could change the HTML and *hack* a website. Further, this data will not persist if the server is shut down. Could there be some way we could have our data persist even when the server restarts?

Flask and SQL

- Just as we have seen how Python can interface with a SQL database, we can combine the power of Flask, Python, and SQL to create a web application where data will persist!
- To implement this, you will need to take a number of steps.
- First, modify `requirements.txt` as follows:

```
cs50
Flask
```

- Modify `index.html` as follows:

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

{% block body %}
    <h1>Register</h1>
    <form action="/register" method="post">
        <input autocomplete="off" autofocus name="name" placeholder="Name" type="
        {% for sport in sports %}
            <input name="sport" type="radio" value="{{ sport }}"> {{ sport }}
        {% endfor %}
        <button type="submit">Register</button>
    </form>
{% endblock %}
```

- Modify `layout.html` as follows:

```
<!DOCTYPE html>

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

- Ensure `failure.html` appears as follows:

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

{% block body %}
    You are not registered!
{% endblock %}
```

- Modify `registrants.html` to appear as follows:

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

{% block body %}
    <h1>Registrants</h1>
    <table>
        <thead>
            <tr>
                <th>Name</th>
                <th>Sport</th>
                <th></th>
            </tr>
        </thead>
        <tbody>
            {% for registrant in registrants %}
                <tr>
                    <td>{{ registrant.name }}</td>
                    <td>{{ registrant.sport }}</td>
                    <td>
                        <form action="/deregister" method="post">
                            <input name="id" type="hidden" value="{{ registrant.i
                            <button type="submit">Deregister</button>
                        </form>
                    </td>
                </tr>
            {% endfor %}
        </tbody>
    </table>
{% endblock %}
```

Notice that a hidden value `registrant.id` is included such that it's possible to use this `id` later in `app.py`

- Finally, modify `app.py` as follows:

```
# Implements a registration form, storing registrants in a SQLite database, with

from cs50 import SQL
from flask import Flask, redirect, render_template, request

app = Flask(__name__)

db = SQL("sqlite:///froshims.db")

SPORTS = [
    "Basketball",
    "Soccer",
    "Ultimate Frisbee"
]


@app.route("/")
def index():
    return render_template("index.html", sports=SPORTS)


@app.route("/deregister", methods=["POST"])
def deregister():

    # Forget registrant
    id = request.form.get("id")
    if id:
        db.execute("DELETE FROM registrants WHERE id = ?", id)
    return redirect("/registrants")


@app.route("/register", methods=["POST"])
def register():

    # Validate submission
    name = request.form.get("name")
    sport = request.form.get("sport")
    if not name or sport not in SPORTS:
        return render_template("failure.html")

    # Remember registrant
    db.execute("INSERT INTO registrants (name, sport) VALUES(?, ?)", name, sport)

    # Confirm registration
    return redirect("/registrants")


@app.route("/registrants")
def registrants():
    registrants = db.execute("SELECT * FROM registrants")
    return render_template("registrants.html", registrants=registrants)
```

Notice that the `cs50` library is utilized. A route is included for `register` for the `post` method. This route will take the name and sport taken from the registration form and execute a SQL query to add the `name` and the `sport` to the `registrants` table. The `deregister` routes to a SQL query that will grab the user's `id` and utilize that information to deregister this individual.

- You can read more in the Flask documentation (https://flask.palletsprojects.com).

Session

- While the above code is useful from an administrative standpoint, where a back-office administrator could add and remove individuals from the database, one can imagine how this code is not safe to implement on a public server.
- For one, bad actors could make decisions on behalf of other users by hitting the deregister button – effectively deleting their recorded answer from the server.
- Web services like Google use login credentials to ensure users only have access to the right data.
- We can actually implement this itself using *cookies*. Cookies are small files that are stored on your computer, such that your computer can communicate with the server and effectively say, "I'm an authorized user that has already logged in."
- In the simplest form, we can implement this by creating a folder called `login` and then adding the following files.
- First, create a file called `requirements.txt` that reads as follows:

```
Flask
Flask-Session
```

  Notice that in addition to `Flask`, we also include `Flask-Session`, which is required to support login sessions.

- Second, in a `templates` folder, create a file called `layout.html` that appears as follows:

```
<!DOCTYPE html>

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

Notice this provides a very simple layout with a title and a body.

- Third, create a file in the `templates` folder called `index.html` that appears as follows:

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

{% block body %}

    {% if session["name"] %}
        You are logged in as {{ session["name"] }}. <a href="/logout">Log out</a>
    {% else %}
        You are not logged in. <a href="/login">Log in</a>.
    {% endif %}

{% endblock %}
```

Notice that this file looks to see if `session["name"]` exists. If it does, it will display a welcome message. If not, it will recommend you browse to a page to log in.

- Fourth, create a file called `login.html` and add the following code:

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

{% block body %}

    <form action="/login" method="post">
        <input autocomplete="off" autofocus name="name" placeholder="Name" type="
        <button type="submit">Log In</button>
    </form>

{% endblock %}
```

Notice this is the layout of a basic login page.

- Finally, create a file in the `login` folder called `app.py` and write code as follows:

```
from flask import Flask, redirect, render_template, request, session
from flask_session import Session

# Configure app
app = Flask(__name__)

# Configure session
app.config["SESSION_PERMANENT"] = False
app.config["SESSION_TYPE"] = "filesystem"
Session(app)


@app.route("/")
def index():
    if not session.get("name"):
        return redirect("/login")
    return render_template("index.html")


@app.route("/login", methods=["GET", "POST"])
def login():
    if request.method == "POST":
        session["name"] = request.form.get("name")
        return redirect("/")
    return render_template("login.html")


@app.route("/logout")
def logout():
    session["name"] = None
    return redirect("/")
```

Notice the modified *imports* at the top of the file, including `session`, which will allow for you to support sessions. Most important, notice how `session["name"]` is used in the `login` and `logout` routes. The `login` route will assign the login name provided and assign it to `session["name"]`. However, in the `logout` route, the logging out is implemented by simply setting `session["name"]` to `None`.

- You can read more about sessions in the Flask documentation (https://flask.palletsprojects.com/en/2.2.x/api/?highlight=session#flask.session).

Store

- Moving on to a final example of utilizing Flask's ability to enable a session.
- We examined the following code for `store` in `app.py`. The following code was shown:

```
from cs50 import SQL
from flask import Flask, redirect, render_template, request, session
from flask_session import Session

# Configure app
app = Flask(__name__)

# Connect to database
db = SQL("sqlite:///store.db")

# Configure session
app.config["SESSION_PERMANENT"] = False
app.config["SESSION_TYPE"] = "filesystem"
Session(app)


@app.route("/")
def index():
    books = db.execute("SELECT * FROM books")
    return render_template("books.html", books=books)


@app.route("/cart", methods=["GET", "POST"])
def cart():

    # Ensure cart exists
    if "cart" not in session:
        session["cart"] = []

    # POST
    if request.method == "POST":
        id = request.form.get("id")
        if id:
            session["cart"].append(id)
        return redirect("/cart")

    # GET
    books = db.execute("SELECT * FROM books WHERE id IN (?)", session["cart"])
    return render_template("cart.html", books=books)
```

Notice that `cart` is implemented using a list. Items can be added to this list using the `Add to Cart` buttons in `books.html`. When clicking such a button, the `post` method is invoked, where the `id` of the item is appended to the `cart`. When viewing the cart, invoking the `get` method, SQL is executed to display a list of the books in the cart.

API

- An *application program interface* or *API* is a series of specifications that allow you to interface with another service. For example, we could utilize IMDB's API to interface with their database. We might even integrate APIs for handling specific types of data downloadable from a server.
- We looked at an example called `shows`.
- Looking at `app.py`, we saw the following:

```
# Searches for shows using Ajax

from cs50 import SQL
from flask import Flask, render_template, request

app = Flask(__name__)

db = SQL("sqlite:///shows.db")


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

```
@app.route("/search")
def search():
    q = request.args.get("q")
    if q:
        shows = db.execute("SELECT * FROM shows WHERE title LIKE ? LIMIT 50", "%"
    else:
        shows = []
    return render_template("search.html", shows=shows)
```

Notice that the `search` route executes a SQL query.

- Looking at `search.html`, you’ll notice that it is very simple:

```
{% for show in shows %}
    <li>{{ show["title"] }}</li>
{% endfor %}
```

Notice that it provides a bulleted list.

- Finally, looking at `index.html`, notice that *AJAX* code is utilized to power the search:

```
<!DOCTYPE html>

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

        <input autocomplete="off" autofocus placeholder="Query" type="search">

        <ul></ul>

        <script>

            let input = document.querySelector('input');
            input.addEventListener('input', async function() {
                let response = await fetch('/search?q=' + input.value);
                let shows = await response.text();
                document.querySelector('ul').innerHTML = shows;
            });

        </script>

    </body>
</html>
```

Notice an event listener is utilized to dynamically query the server to provide a list that matches the title provided. This will locate the `ul` tag in the HTML and modify the web page accordingly to include the list of the matches.

- You can read more in the AJAX documentation (https://api.jquery.com/category/ajax/).

JSON

- *JavaScript Object Notation* or *JSON* is text file of dictionaries with keys and values. This is a raw, computer-friendly way to get lots of data.
- JSON is a very useful way of getting back data from the server.
- You can see this in action in the `index.html` we examined together:

  ```
  <!DOCTYPE html>

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

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

          <ul></ul>

          <script>

              let input = document.querySelector('input');
              input.addEventListener('input', async function() {
                  let response = await fetch('/search?q=' + input.value);
                  let shows = await response.json();
                  let html = '';
                  for (let id in shows) {
                      let title = shows[id].title.replace('<', '&lt;').replace('&',
                      html += '<li>' + title + '</li>';
                  }
                  document.querySelector('ul').innerHTML = html;
              });

          </script>

      </body>
  </html>
  ```

  While the above may be somewhat cryptic, it provides a starting point for you to research JSON on your own to see how it can be implemented in your own web applications.
- You can read more in the JSON documentation (https://www.json.org/json-en.html).

Summing Up

In this lesson, you learned how to utilize Python, SQL, and Flask to create web applications. Specifically, we discussed…

- GET
- POST
- Flask
- Session
- AJAX
- JSON

See you next time for our final lecture!