This is CS50x

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://orcid.org/0000-0001-5338-2522)
(https://www.quora.com/profile/David-J-Malan)
(https://www.reddit.com/user/davidjmalan) (https://twitter.com/davidjmalan)

Lecture 9

Web programming

- Today we'll create more advanced web applications by writing code that runs on the server.
- Last week, we used `http-server` in the CS50 IDE as a **web server**, a program that listens for connections and requests, and responds with web pages or other resources.
- An HTTP request has headers, like:

```
GET / HTTP/1.1
...
```

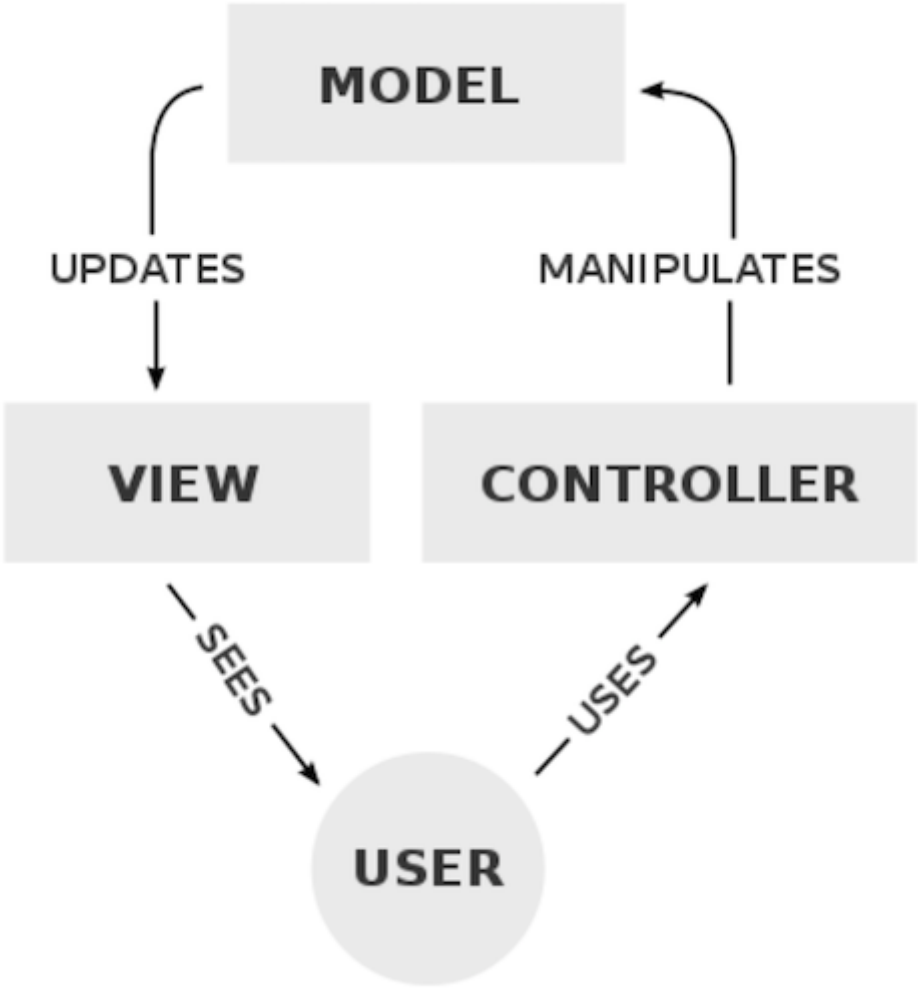
    - These headers can ask for some file or page, or send data from the browser back to the server.
- While `http-server` only responds with static pages, we can use other web servers that parses, or analyzes request headers, like `GET /search?q=cats HTTP/1.1`, to return pages dynamically.

Flask

- We'll use Python and a library called **Flask** to write our own web server, implementing additional features. Flask is also a **framework**, where the library of code also comes with a set of conventions for how it should be used. For example, like other libraries, Flask includes functions we can use to parse requests individually, but as a framework, also requires our program's code to be organized in a certain way:

```
application.py
requirements.txt
static/
templates/
```

    - `application.py` will have the Python code for our web server.
    - `requirements.txt` includes a list of required libraries for our application.
    - `static/` is a directory of static files, like CSS and JavaScript files.
    - `templates/` is a directory for files that will be used to create our final HTML.
- There are many web server frameworks for each of the popular languages, and Flask will be a representative one that we use today.
- Flask also implements a particular **design pattern**, or way that our program and code is organized. For Flask, the design pattern is generally **MVC**, or Model–view–controller:



  - The controller is our logic and code that manages our application overall, given user input. In Flask, this will be our Python code.
  - The view is the user interface, like the HTML and CSS that the user will see and interact with.
  - The model is our application's data, such as a SQL database or CSV file.
- The simplest Flask application might look like this:

```
from flask import Flask

app = Flask(__name__)

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

  - First, we'll import `Flask` from the `flask` library, which happens to use a capital letter for its main name.
  - Then, we'll create an `app` variable by giving our file's name to the `Flask` variable.
  - Next, we'll label a function for the `/` route, or URL with `@app.route`. The `@` symbol in Python is called a decorator, which applies one function to another.

    - We'll call the function `index`, since it should respond to a request for `/`, the default page. And our function will just respond with a string for now.
- In the CS50 IDE, we can go to the directory with our application code, and type `flask run` to start it. We'll see a URL, and we can open it to see `hello, world`.
- We'll update our code to actually return HTML with the `render_template` function, which finds a file given and returns its contents:

```
from flask import Flask, render_template

app = Flask(__name__)


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

    - We'll need to create a `templates/` directory, and create an `index.html` file with some content inside it.
    - Now, typing `flask run` will return that HTML file when we visit our server's URL.
- We'll pass in an argument to `render_template` in our controller code:

```
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"
```

    - It turns out that we can give `render_template` any named argument, like `name`, and it will substitute that in our template, or our HTML file with placeholders.
        - In `index.html`, we'll replace `hello, world` with `hello,` to tell Flask where to substitute the `name` variable:

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <title>hello</title>
    </head>
    <body>
        hello, {{ name }}
    </body>
</html>
```

- We can use the `request` variable from the Flask library to get a parameter from the HTTP request, in this case also `name`, and fall back to a default of `world` if one wasn't provided.
- Now, when we restart our server after making these changes, and visit the default page with a URL like `/?name=David`, we'll see that same input returned back to us in the HTML generated by our server.

- We can presume that Google's search query, at `/search?q=cats`, is also parsed by some code for the `q` parameter and passed along to some database to get all the results that are relevant. Those results are then used to generate the final HTML page.

Forms

- We'll move our original template into `greet.html`, so it will greet the user with their name. In `index.html`, we'll create a form:

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <title>hello</title>
    </head>
    <body>
        <form action="/greet" method="get">
            <input name="name" type="text">
            <input type="submit">
        </form>
    </body>
</html>
```

  - We'll send the form to the `/greet` route, and have an input for the `name` parameter and one for the submit button.
  - In our `applications.py` controller, we'll also need to add a function for the `/greet` route, which is almost exactly what we had for `/` before:

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


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

    - Our form at `index.html` will be static since it can be the same every time.

- Now, we can run our server, see our form at the default page, and use it to generate another page.

POST

- Our form above used the GET method, which includes our form's data in the URL.
- We'll change the method in our HTML: `<form action="/greet" method="post">`. Our controller will also need to be changed to accept the POST method, and look for the parameter somewhere else:

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

  - While `request.args` is for parameters in a GET request, we have to use `request.form` in Flask for parameters in a POST request.
- Now, when we restart our application after making these changes, we can see that the form takes us to `/greet`, but the contents aren't included in the URL anymore.

Layouts

- In `index.html` and `greet.html`, we have some repeated HTML code. With just HTML, we aren't able to share code between files, but with Flask templates (and other web frameworks), we can factor out such common content.
- We'll create another template, `layout.html`:

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <title>hello</title>
    </head>
    <body>
        {% block body %}{% endblock %}
    </body>
</html>
```

  - Flask supports Jinja, a templating language, which uses the `{% %}` syntax to include placeholder blocks, or other chunks of code. Here we've named our block `body` since it contains the HTML that should go in the `<body>` element.
- In `index.html`, we'll use the `layout.html` blueprint and only define the `body` block with:

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

{% block body %}

    <form action="/greet" method="post">
        <input autocomplete="off" autofocus name="name" placeholder="Name" typ
        <input type="submit">
    </form>

{% endblock %}
```

- Similarly, in `greet.html`, we define the `body` block with just the greeting:

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

{% block body %}

    hello, {{ name }}

{% endblock %}
```

- Now, if we restart our server, and view the source of our HTML after opening our server's URL, we see a complete page with our form inside our HTML file, generated by Flask.
- We can even reuse the same route to support both GET and POST methods:

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

  - First, we check if the `method` of the `request` is a POST request. If so, we'll look for the `name` parameter and return HTML from the `greet.html` template. Otherwise, we'll return HTML from the `index.html`, which has our form.
  - We'll also need to change the form's `action` to the default `/` route.

Frosh IMs

- One of David's first web applications was for students on campus to register for "frosh IMs", intramural sports.
- We'll use a `layout.html` similar to what we had before:

```
<!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>
```

  - A `<meta>` tag in `<head>` allows us to add more metadata to our page. In this case, we're adding a `content` attribute for the `viewport` metadata, in order to tell the browser to automamtically scale our page's size and fonts to the device.

- In our `application.py`, we'll return our `index.html` template for the default `/` route:

```
from flask import Flask, render_template, request

app = Flask(__name__)

SPORTS = [
    "Dodgeball",
    "Flag Football",
    "Soccer",
    "Volleyball",
    "Ultimate Frisbee"
]

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

- Our `index.html` template will look like this:

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

{% block body %}
    <h1>Register</h1>

    <form action="/register" method="post">

        <input autocomplete="off" autofocus name="name" placeholder="Name" typ
        <select name="sport">
            <option disabled selected value="">Sport</option>
            <option value="Dodgeball">Dodgeball</option>
            <option value="Flag Football">Flag Football</option>
            <option value="Soccer">Soccer</option>
            <option value="Volleyball">Volleyball</option>
            <option value="Ultimate Frisbee">Ultimate Frisbee</option>
        </select>
        <input type="submit" value="Register">
```

```
    </form>
{% endblock %}
```

- We'll have a form like before, and have a `<select>` menu with options for each sport.

- In our `application.py`, we'll allow POST for our `/register` route:

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

  if not request.form.get("name") or not request.form.get("sport"):
      return render_template("failure.html")

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

- We'll check that our form's values are valid, and then return a template depending on the results, even though we aren't actually doing anything with the data yet.

- But a user can change the form's HTML in their browser, and send a request that contains some other sport as the selected option!
- We'll check that the value for `sport` is valid by creating a list in `application.py`:

```
from flask import Flask, render_template, request

app = Flask(__name__)

SPORTS = [
    "Dodgeball",
    "Flag Football",
    "Soccer",
    "Volleyball",
    "Ultimate Frisbee"
]

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

...
```

- Then, we'll pass that list into the `index.html` template.

- In our template, we can even use loops to generate a list of options from the list of strings passed in as `sports`:

```
...
<select name="sport">
    <option disabled selected value="">Sport</option>
```

```
    {% for sport in sports %}
        <option value="{{ sport }}">{{ sport }}</option>
    {% endfor %}
</select>
...
```

- Finally, we can check that the `sport` sent in the POST request is in the list `SPORTS` in `application.py`:

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

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

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

- We can change the select menu in our form to be checkboxes, to allow for multiple sports:

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

{% block body %}
    <h1>Register</h1>

    <form action="/register" method="post">

        <input autocomplete="off" autofocus name="name" placeholder="Name" typ
        {% for sport in sports %}
            <input name="sport" type="checkbox" value="{{ sport }}"> {{ sport
        {% endfor %}
        <input type="submit" value="Register">

    </form>
{% endblock %}
```

  - In our `register` function, we can call `request.form.getlist` to get the list of checked options.
- We can also use radio buttons, which will allow only one option to be chosen at a time.

Storing data

- Let's store our registered students, or registrants, in a dictionary in the memory of our web server:

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

app = Flask(__name__)

REGISTRANTS = {}

...

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

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

    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")

    REGISTRANTS[name] = sport

    return redirect("/registrants")
```

- We'll create a dictionary called `REGISTRANTS`, and in `register` we'll first check the `name` and `sport`, returning a different error message in each case. Then, we can safely store the name and sport in our `REGISTRANTS` dictionary, and redirect to another route that will display registered students.
- The error message template, meanwhile, will just display the message:

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

{% block body %}
    {{ message }}
{% endblock %}
```

- Let's add the `/registrants` route and template to show the registered students:

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

- In our route, we'll pass in the `REGISTRANTS` dictionary to the template as a parameter called `registrants`:

```
{% 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 %}
```

  - Our template will have a table, with a heading row and row for each key and value stored in `registrants`.

- If our web server stops running, we'll lose the data stored, so we'll use a SQLite database with the SQL library from `cs50`:

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

app = Flask(__name__)

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

...
```

  - In the IDE's terminal, we can run `sqlite3 froshims.db` to open the database, and use the `.schema` command to see the table with columns of `id`, `name`, and `sport`, which was created in advance.

- Now, in our routes, we can insert and select rows with SQL:

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

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

```
    db.execute("INSERT INTO registrants (name, sport) VALUES(?, ?)", name, spo

    return redirect("/registrants")


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

  - Once we've validated the request, we can use `INSERT INTO` to add a row, and similarly, in `registrants()`, we can `SELECT` all rows and pass them to the template as a list of rows.
- Our `registrants.html` template will also need to be adjusted, since each row returned from `db.execute` is a dictionary. So we can use `registrant.name` and `registrant.sport` to access the value of each key in each row:

```
<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.id }}"
                    <input type="submit" value="Deregister">
                </form>
            </td>
        </tr>
    {% endfor %}
</tbody>
```

- We can even email users with another library, `flask_mail`:

```
import os
import re

from flask import Flask, render_template, request
from flask_mail import Mail, Message

app = Flask(__name__)
app.config["MAIL_DEFAULT_SENDER"] = os.getenv("MAIL_DEFAULT_SENDER")
app.config["MAIL_PASSWORD"] = os.getenv("MAIL_PASSWORD")
app.config["MAIL_PORT"] = 587
app.config["MAIL_SERVER"] = "smtp.gmail.com"
app.config["MAIL_USE_TLS"] = True
```

```
app.config["MAIL_USERNAME"] = os.getenv("MAIL_USERNAME")
mail = Mail(app)
```

- We've set some sensitive variables outside of our code, in the IDE's environment, so we can avoid including them in our code.
- It turns out that we can provide configuration details like a username and password and mail server, in this case Gmail's, to the `Mail` variable, which will send mail for us.

- Finally, in our `register` route, we can send an email to the user:

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

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

    message = Message("You are registered!", recipients=[email])
    mail.send(message)

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

- In our form, we'll also need to ask for an email instead of a name:

```
<input autocomplete="off" name="email" placeholder="Email" type="email">
```

- Now, if we restart our server and use the form to provide an email, we'll see that we indeed get one sent to us!

Sessions

- **Sessions** are how web servers remembers information about each user, which enables features like allowing users to stay logged in.
- It turns out that servers can send another header in a response, called `Set-Cookie`:

```
HTTP/1.1 200 OK
Content-Type: text/html
Set-Cookie: session=value
...
```

    - **Cookies** are small pieces of data from a web server that the browser saves for us. In many cases, they are large random numbers or strings used to uniquely identify and track a user between visits.
    - In this case, the server is asking our browser to set a cookie for that server, called `session` to a value of `value`.
- Then, when the browser makes another request to the same server, it'll send back cookies that the same server has set before:

```
GET / HTTP/1.1
Host: gmail.com
Cookie: session=value
```

- In the real world, amusement parks might give you a hand stamp so you can return after leaving. Similarly, our browser is presenting our cookies back to the web server, so it can remember who we are.
- Advertising companies might set cookies from a number of websites, in order to track users across all of them. In Incognito mode, by contrast, the browser doesn't send any cookies set from before.
- In Flask, we can use the `flask_session` library to manage this for us:

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

app = Flask(__name__)
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("/")
```

- We'll configure the session library to use the IDE's filesystem, and use `session` like a dictionary to store a user's name. It turns out that Flask will use HTTP cookies for us, to maintain this `session` variable for each user visiting our web server. Each visitor will get their own `session` variable, even though it appears to be global in our code.
- For our default `/` route, we'll redirect to `/login` if there's no name set in `session` for the user yet, and otherwise show a default `index.html` template.
- For our `/login` route, we'll set `name` in `session` to the form's value sent via POST, and then redirect to the default route. If we visited the route via GET, we'll render the login form at `login.html`.
- For the `/logout` route, we can clear the value for `name` in `session` by setting it to `None`, and redirect to `/` again.
- We'll also generally need a `requirements.txt` that includes the names of libraries we want to use, so they can be installed for our application, but the ones we use here have been preinstalled in the IDE.

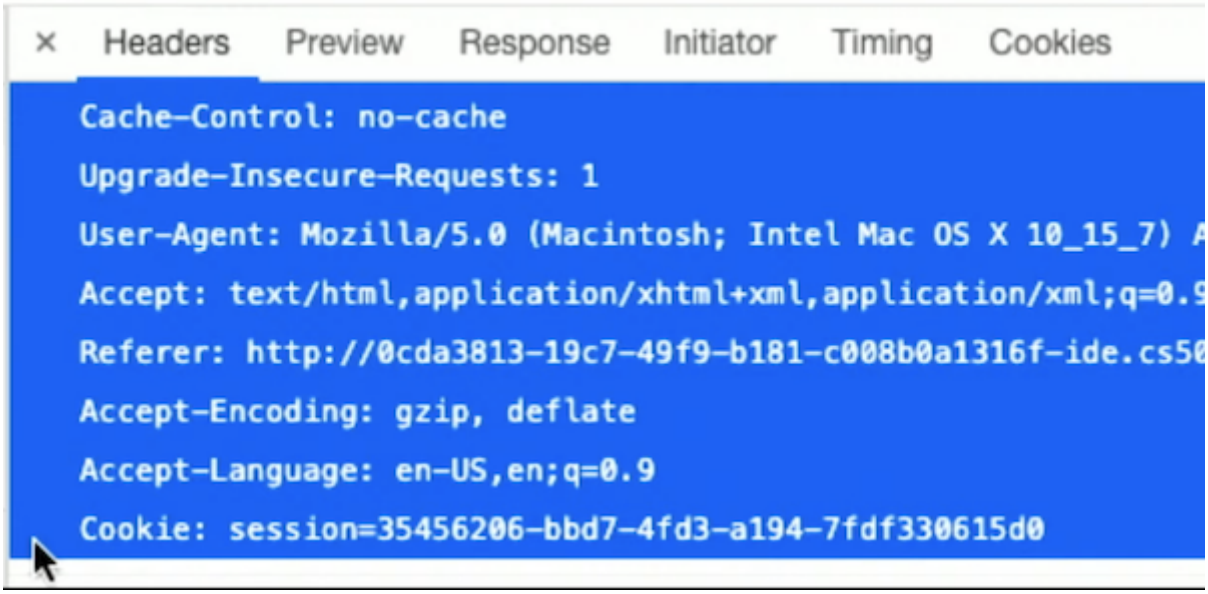
- In our `login.html`, we'll have a form with just a name:

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

{% block body %}

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

{% endblock %}
```

- And in our `index.html`, we can check if `session.name` exists, and show different content:

```
{% 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 %}
```

- When we restart our server, go to its URL, and log in, we can see in the Network tab that our browser is indeed sending a `Cookie:` header in the request:



store, shows

- We'll look through an example, `store` (https://cdn.cs50.net/2020/fall/lectures/9/src9/store/):
  - `application.py` initializes and configures our application to use a database and sessions. In `index()`, the default route renders a list of books stored in the database.
  - `templates/books.html` shows the list of `books`, as well as a form that allows us to click "Add to Cart" for each of them.
  - The `/cart` route, in turn, stores an `id` from a POST request in the `session` variable in a list. If the request used a GET method, however, `/cart` would show a list of books with `id`s matching the list of `id`s stored in `session`.
- So, "shopping carts" on websites can be implemented with cookies and session variables stored on the server.
- When we view the source generated by our default route, we see that each book has its own `<form>` element, each with a different `id` input that's hidden and generated. This `id` comes from the SQLite database on our server, and is sent back to the `/cart` route.
- We'll look at another example, `shows`, where we can use both JavaScript on the **front-end**, or side that the user sees, and Python on the **back-end**, or server side.
- In `application.py` here, we'll open a database, `shows.db`:

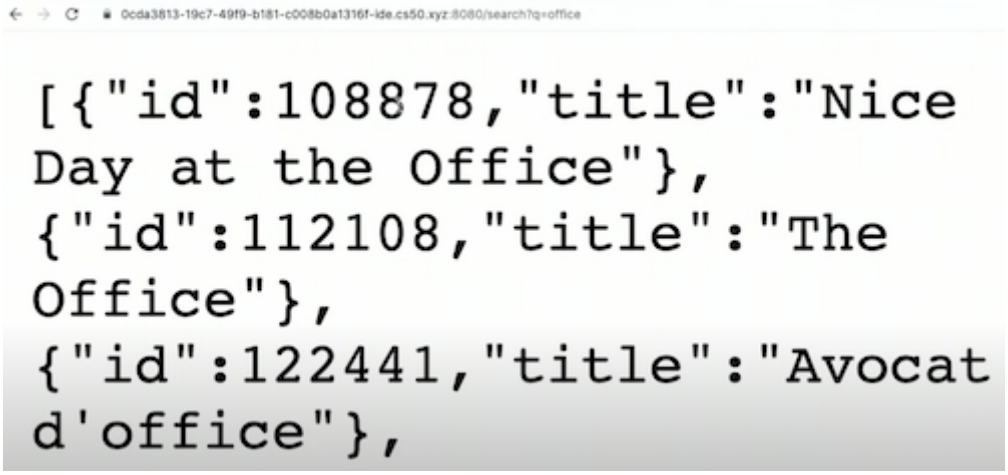
```
from cs50 import SQL
from flask import Flask, render_template, request

app = Flask(__name__)

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

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


@app.route("/search")
def search():
    shows = db.execute("SELECT * FROM shows WHERE title LIKE ?", "%" + request
    return render_template("search.html", shows=shows)
```

  - The default `/` route will show a form, where we can type in some search term.
  - The form will use the GET method to send the search query to `/search`, which in turn will use the database to find a list of shows that match. Finally, a `search.html` template will show the list of shows.
- With JavaScript, we can show a partial list of results as we type. First, we'll use a function called `jsonify` to return our shows in the JSON format, a standard format that JavaScript can use.

```
@app.route("/search")
def search():
    shows = db.execute("SELECT * FROM shows WHERE title LIKE ?", "%" + request
    return jsonify(shows)
```

  - Now we can submit a search query, and see that we get back a list of dictionaries:

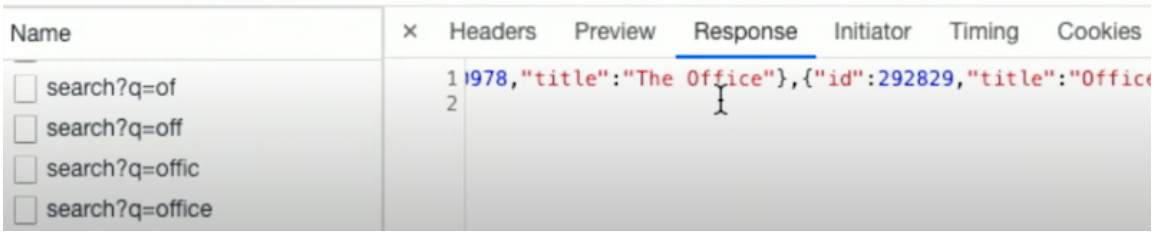


- Then, our `index.html` template can convert this list to elements in the DOM:

```
<!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 crossorigin="anonymous" integrity="sha256-9/aliU8dGd2tb6OSsuzi
        <script>

            let input = document.querySelector('input');
            input.addEventListener('keyup', function() {
                $.get('/search?q=' + input.value, function(shows) {
                  let html = '';
                  for (let id in shows)
                  {
                      let title = shows[id].title;
                      html += '<li>' + title + '</li>';
                  }

                  document.querySelector('ul').innerHTML = html;
                });
            });

        </script>

    </body>
</html>
```

- We'll use another library, JQuery, to make requests more easily.
- We'll listen to changes in the `input` element, and use `$.get`, which calls a JQuery library function to make a GET request with the input's value. Then, the response will be passed to an anonymous function as the variable `shows`, which will set the DOM with generated `<li>` elements based on the response's data.
- `$.get` is an **AJAX** call, which allows for JavaScript to make additional HTTP requests after the page has loaded, to get more data. If we open the Network tab again, we can indeed see that each key we pressed made another request, with a response:



- Since the network request might be slow, the anonymous function we pass to `$.get` is a **callback** function, which is only called after we get a response from the server. In the meantime, the browser can run other JavaScript code.

- That's it for today!