CS 61A SU26

Lecture 25: Web Applications

August 4, 2026

Presented by Rebecca Dang & Abby Brooks-Ramirez
Special thanks to Emma Zhong

We'll begin at Berkeley time (10 minutes after), as per tradition!

Introducing: Abby Brooks-Ramirez

Hi folks! I'm Abby (she/her)

- **Education:** UC Berkeley BA in CS, MS in EECS
- **Teaching:** introductory + sociotechnical education
  - Data 6 (Computational Thinking with Data + Society)
  - CS375 (Teaching Techniques for Computer Science)
  - CS195 (Social Implications of Computing)
  - Data C104 (Human Contexts and Ethics of Data)
  - CS61B (Data Structures)
  - CS61A (this!)
- **Professionally:**
  - I <3 science museums: Lawrence Hall, Exploratorium
  - Spotify → IBM → (soon) Amplify
- **Interests**: reading, critical theory, sewing, my cat, and nature

Potpourri

2 min

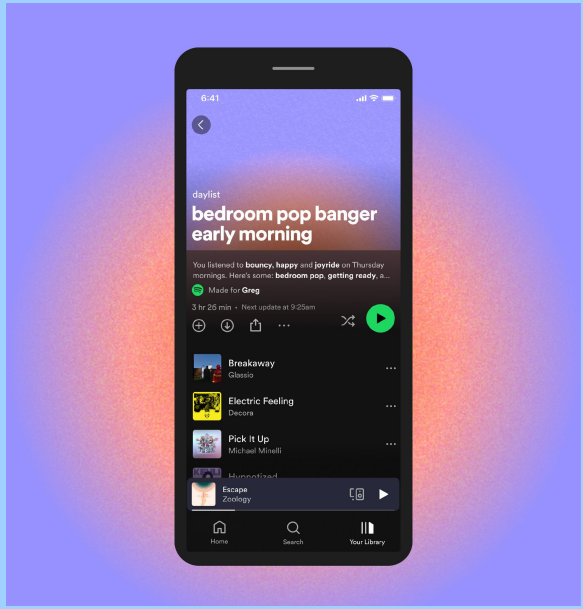
- Announcements

Announcements

- **Final Review Sessions** are going on throughout this week (and are recorded)
- **Correction on number of data centers in the US from yesterday's lecture:** As of April 2026, according to Pew Research, there are ~3000 data centers in the US (not including 1,500+ at various stages of development)
- Plugging the AMA thread again!

We Have Spotify at Home

10 min

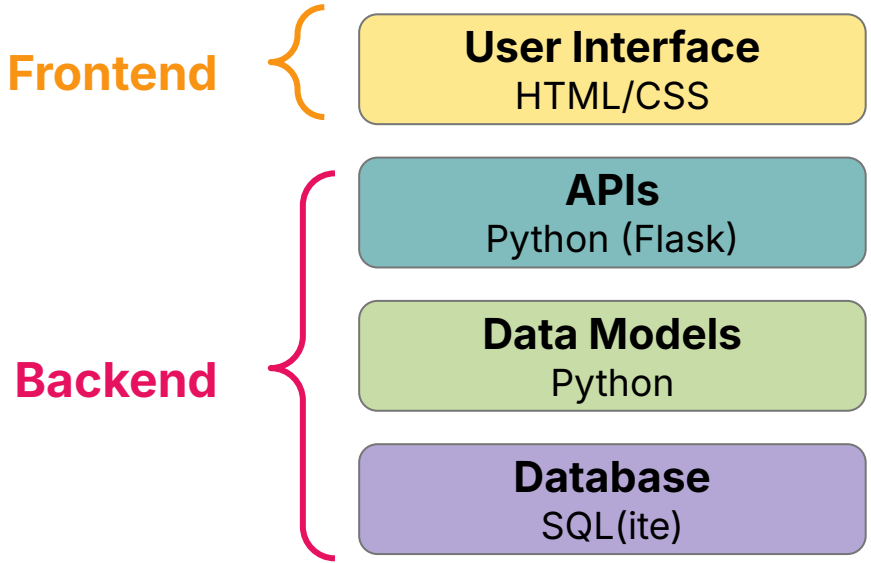


Today we will learn about **full stack web apps** and create a mini version of a popular one: the Spotify Daylist

Full Stack Development

A tech “stack” refers to the set of technologies used in order to create a product. “Full stack” development incorporates technologies from various disciplines.

Our app today will have the following layers:



Our Daylist (1 of 2)

- To create our daylist, we will use a dataset containing information about popular songs, which we gathered using Spotify's API

  → an **Application Programming Interface** (API) allows you to request and send data between websites/organizations. In our case, we used Spotify's API to *request* data.

  → More on APIs later!

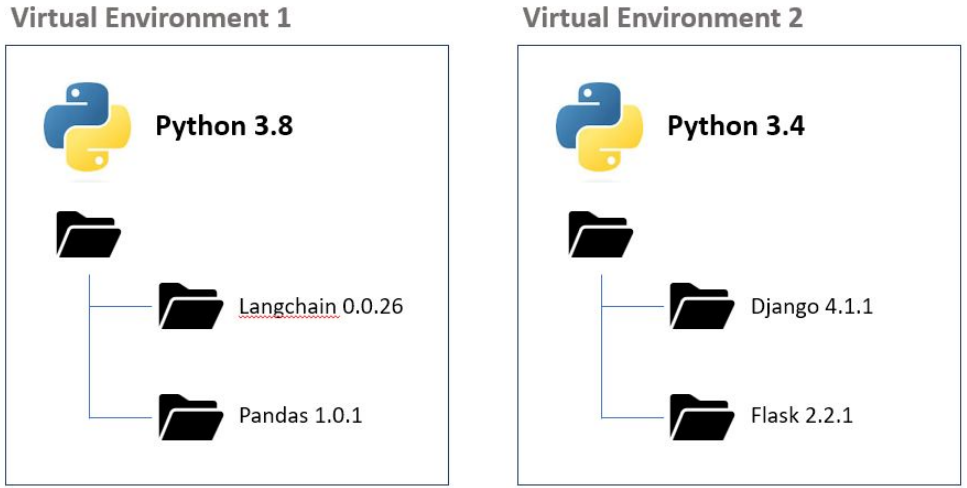
- Our dataset contains the following fields:
  - `id (INTEGER)` — unique ID for each song, specific to our database
  - `song_link (TEXT)` — URL of the song on Spotify
  - `song_id (TEXT)` — unique ID for each song, used by Spotify
  - `track_name (TEXT)` — song title
  - `artist_name (TEXT)` — artist name
  - `album_name (TEXT)` — name of the album the song is from
  - `album_image_url (TEXT)` — image URL of the album
  - `duration (FLOAT)` — length of the song in seconds

Our Daylist (2 of 2)

```
project/
├── data/              # Where our dataset lives
├── templates/         # Where our frontend lives
├── __init__.py        # Code to initialize our web app
├── api.py             # Where we create API routes
├── dao.py          # Where we will access our database
├── db.py               # Where our database lives
├── main.py            # Where we render the appropriate templates
└── schema.sql         # Where we define our database schema
```

Managing Dependencies

- We will use a command line interface called uv for this project
- uv allows software developers to manage Python versions, projects, and dependencies
- uv creates a **virtual environment** in which it will install packages and libraries. These won't be installed (or override) existing packages on your local computer!



Task 0: Setup

1. Go to the starter code GitHub repository: https://github.com/abigailrb03/daylist-at-home-starter
2. Download a .zip file of the repo or `git clone` it (if you know how)
3. Follow the instructions under Task 0: Setup

Task 0: Setup (Detailed)

1. Install `uv`, a command line interface (CLI) to manage Python versions, projects, and dependencies
   a. MacOS/Unix: `curl -LsSf https://astral.sh/uv/install.sh | sh`
   b. Windows: `powershell -ExecutionPolicy ByPass -c "irm https://astral.sh/uv/install.ps1 | iex"`
2. Set up your **virtual environment** and *initialize* the app's database
   a. Software engineers often use virtual environments to manage dependencies and isolate it from other systems on their local machine
   b. `uv run flask --app daylist init-db`
3. Verify that:
   a. You have a new directory called `.venv` (this is your virtual environment)
   b. You have a new directory called `instance` (this is your database)

Task 1: OOP Design

25 min

- Demo: Local Development
- Task 1A
- Task 1B
- Solution Walkthrough

Demo: Local Development

1. Start the local development server (reference):

   ```
   uv run flask --app daylist --debug run
   ```

2. View the web app in your browser by going to:

   ```
   http://127.0.0.1:5000
   ```

3. Run all unit tests (reference):

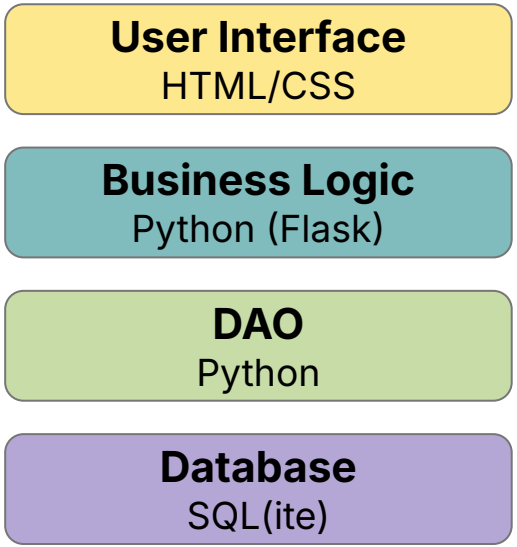
   ```
   uv run pytest
   ```

Aside: What is `127.0.0.1:5000`?

- All websites are uniquely identified by their IP address
- `127.0.0.1` is a special IP address called **`localhost`** which just refers to your own computer (it's not accessible to anyone else)
- The `:5000` means that the web app is running on **port 5000**.

Task 1A (1 of 2)

- So far in this class, we've learned about OOP, inheritance, and composition
  - Classes/attributes/methods are usually well-scoped for you
  - But in the real world, you will be expected to **design** your own classes to fulfill certain **requirements** → we will do this in the next slide!
- A common **design pattern**: Store persistent data in a database and have an abstraction layer (class) called a **data access object (DAO)** which is used by the higher-level business logic code to query the database
  - Prevents abstraction barrier violations!



Task 1A (2 of 2)

[5 min] In this exercise, **work with a partner to write a design doc (you can use the provided template)** to fulfill the requirements for 2 classes in the dao.py file: `DataAccessObject` and `Song`

Notes:

- The `db` parameter of the `DataAccessObject.__init__` method is an object that lets you query the database; you don't need to know more than this to write the design doc.
- The AI design critique is optional; focus on working with your partner.

Hints:

- What information about a `Song` do you need to keep track of?
- In `DataAccessObject`, consider the pros/cons of these 2 approaches:
  - When a `DataAccessObject` instance is created, query the database and store all songs as an instance attribute, then use this attribute when someone calls the instance methods
  - Query the database as needed when some calls the instance methods

Task 1B (1 of 3)

Before we implement our design, here's some examples of how to query the database using Python's `sqlite3` module:

```
# Suppose `conn` is an already defined sqlite3.Connection object
# Fetch all rows
>>> query_result = conn.execute("SELECT name, email FROM contacts")
>>> rows = query_result.fetchall() # suppose there are 3 rows
>>> for row in rows:
...     print(f"{row['name']}: {row['email']}")
John Doe: john@berkeley.edu
Jane Doe: jane@berkeley.edu
Oski Bear: oski@berkeley.edu
```

Task 1B (2 of 3)

```
# Fetch one row
>>> query_result = conn.execute("SELECT name, email FROM contacts")
>>> row = query_result.fetchone()
>>> print(f"{row['name']}: {row['email']}")
John Doe: john@berkeley.edu
# Parameterized query (to prevent SQL injection)
>>> query = """
...         SELECT is_favorite FROM contacts
...         WHERE name = ? AND email = ?
...         """
>>> query_result = conn.execute(query, ('Oski Bear', 'oski@berkeley.edu'))
>>> row = query_result.fetchone()
>>> print(row['is_favorite'])
True
```

Task 1B (3 of 3)

- Now that you have a design, let's implement it in code!
- Refer to the full spec for this task
- Run tests on this task: `uv run pytest tests/test_dao.py`

Solution Walkthrough (Task 1)

Task 2: Create an API Endpoint

15 min

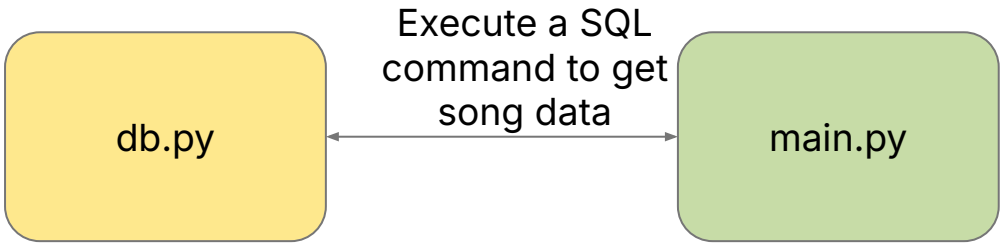
- Blueprints and Routes
- The /songs route
- APIs
- Task 2

Blueprints and Routes

- A **route** is a URL path that is used to navigate to different parts of a webpage. Each route executes different logic to render the page we see!
  - https://cs61a.org/**office-hours/**
  - https://cs61a.org/**staff/**
- Flask **Blueprints** are structures that allow developers (us) to organize our *routes*. By grouping related routes together, we can make our app more modular and easier to read (putting every single route in the same file can get messy fast)
  - `daylist/main.py` and `daylist/api.py` define the `main` and `api` Blueprints
  - `main.py` defines the routes our webpage will reach
  - `api.py` defines the routes that our API will reach (more on this soon)

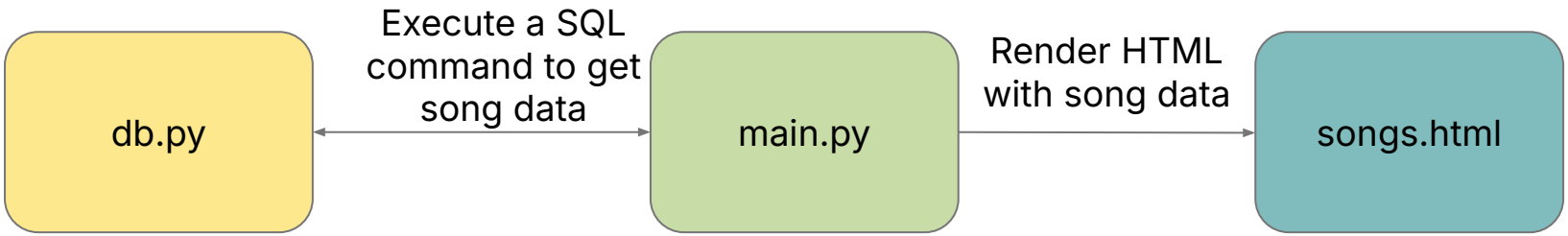
The /songs route (1 of 2)

- When you navigate to http://127.0.0.1:5000/**songs**, the app will reference the function `@bp.route('/songs')` in `daylist/main.py` which:
  - Selects all songs from the songs table



The /songs route (2 of 2)

- When you navigate to http://127.0.0.1:5000/**songs**, the app will reference the function `@bp.route('/songs')` in `daylist/main.py` which:
  - Selects all songs from the songs table
  - Calls the `render_template` function which renders the HTML template in `daylist/templates/songs.html`
  - The `songs.html` template then iterates over the songs passed to it through `main.py`



APIs (1 of 3)

- An **API** allows programmers to retrieve and send data
  - An analogy: in a restaurant, the customer orders food by sharing their order with the waiter, the waiter interfaces with the chefs who fulfil the order. The waiter then delivers the order to the customer!
  - The customer is like the programmer, the waiter is the API, and the chef is like the backend (e.g. the database)
- APIs are typically described by HTTP request methods
  - **HTTP**: a protocol for sending messages (like data requests) across the internet
  - Request method: the purpose of the request
    - `GET`: retrieving data from a resource (like a database)
    - `POST`: submitting data to a resource
    - `DELETE`: Deleting data from a resource
    - And many others!
    - Today we will work with `GET`ting data

APIs (2 of 3)

- API requests will return a **JSON object** and an **HTTP status code**
- GET requests will return a response in JSON format
  - JSON is a file format often used for transferring data across the web, ex:

```
{"id": 101,
  "name": "Alice Smith"}
```

  - For the purpose of this project, treat it like a Python dictionary
- HTTP status codes tell programmers if something went wrong (fun):

Status Code	Name	Description
200	OK	Request successful
400	BAD REQUEST	Something went wrong on the programmer's side, e.g. they provided bad input parameters
404	NOT FOUND	The resource the programmer requested is not found in the backend

APIs (3 of 3)

APIs can also take in **query parameters** through their URL. For instance:

```
GET /api/endpoint?param1=value1&param2=value2
```

- In this example, `param1` is set to `value1` and `param2` is set to `value2`
- Query parameters begin with **?** and are separated by **&**
- `endpoint`**:** In this project, "**endpoint**" and "**route**" mean the same thing. An endpoint is the specific path your program uses to run the logic needed to return the data an API request is asking for

Task 2 (1 of 3)

Let's look at an example API request:

```
GET /api/users?id=12345
```

We can write code so that this endpoint executes logic to retrieve information about the user with `id 12345`

- Let's assume we have created a `DataAccessObject` which has the same `__init__` logic as the `DataAccessObject` we've already created
  - It contains a method `get_user(id)` which returns a `user` object
  - The `user` object contains the attributes `name` and `email`

Task 2 (2 of 3)

```
@bp.route("/api/user", methods=["GET"])
def get_user():
    user_id = request.args.get("id", default=None)
    dao = DataAccessObject(db.get_db()) # create our DAO object
    user = dao.get_user(id) # get the user object from the database
    if user:
        return (
          jsonify({"user": user.name, "email": user.email}), # return response as a JSON
          HTTPStatus.OK # include HTTP status
        )
    else:
        return (
            jsonify({
                "status": "error",
                "message": f"user with id {user_id} not found"
            }),
            HTTPStatus.NOT_FOUND
        )
```

Task 2 (3 of 3)

- Implement Task 2
- Refer to the full spec for this task
- Run tests on this task: `uv run pytest tests/test_api.py -k test_track_image`

Solution Walkthrough (Task 2)

Task 3:
`GET /api/daylist`

15 min

- Pseudorandom Number Generation
- Task 3

Pseudorandom Number Generation (1 of 3)

- In software, we sometimes may want to have some form of randomization
  - Ex: Randomly sample or shuffle songs in a playlist (fun fact: Spotify's shuffle algorithm is not 100% random)
- Genuine random number generation is hard to do in hardware → instead computers will typically use ***pseudorandom*** number generators (PRNGs)
  - One benefit of using PRNGs is that you can set a **seed** so that the generated numbers occur deterministically (useful for unit testing)
- Python has a `random` module which allows you to do this

Pseudorandom Number Generation (2 of 3)

```
>>> import random
>>> lst = [1, 2, 3, 4, 5]
>>> random.sample(lst, 2)
[5, 1]
>>> random.sample(lst, 2)
[5, 2]
>>> exit()
```

```
# New interpreter session
# -> different sample
>>> import random
>>> lst = [1, 2, 3, 4, 5]
>>> random.sample(lst, 2)
[3, 5]
>>> random.sample(lst, 2)
[4, 1]
```

Pseudorandom Number Generation (3 of 3)

```
>>> import random
>>> random.seed(61)
>>> lst = [1, 2, 3, 4, 5]
>>> random.sample(lst, 2)
[4, 2]
>>> random.sample(lst, 2)
[5, 2]
>>> exit()
```

```
# New interpreter session with
# same seed -> same sample!
>>> import random
>>> random.seed(61)
>>> lst = [1, 2, 3, 4, 5]
>>> random.sample(lst, 2)
[4, 2]
>>> random.sample(lst, 2)
[5, 2]
```

Task 3

- [5 min] Implement the `GET /api/daylist` endpoint (spec)
  - One of the query parameters is an optional seed to randomly sample songs in the playlist
- Run tests for this task:

```
uv run pytest tests/test_api.py -k test_daylist_playlist
```

Solution Walkthrough (Task 3)

Task 4: Playlist Title Generation

10 min

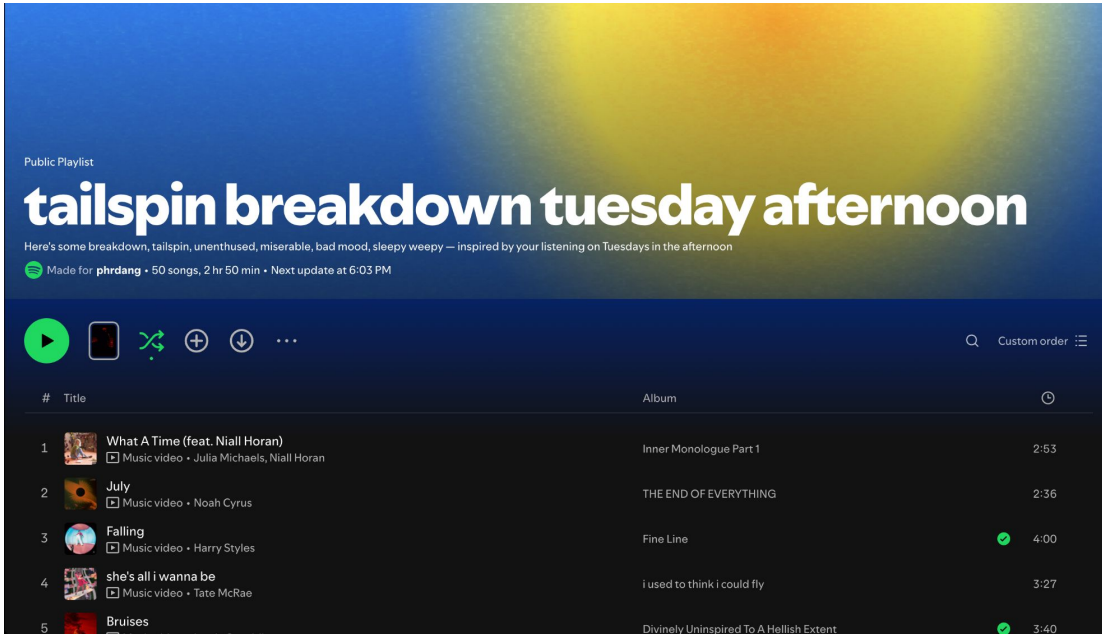
- Ollama and Prompt Engineering
- `datetime` module
- Task 4

Ollama and Prompt Engineering

- Our daylist now works! But what if we want to have a dynamically generated playlist title, instead of a hardcoded one?
  - One possible solution is to generate one with an LLM!
- Enter **Ollama**: An app and command line interface that lets you download and run LLMs. Why use it?
  - It's a unified platform to plug and play different models, which is cool for experimentation
  - It runs locally → data privacy
  - It's easy to install and use
- In Task 4, you'll be doing some **prompt engineering** to ask a model of your choice (we recommend Llama 3.2) to generate a playlist title for you
  - Recall: Prompt engineering is just manually adjusting/trying out different prompts to get the best results

datetime module (1 of 2)

- But remember that LLMs are non-deterministic. Is there part of the playlist title we know for sure will always be there?
  - The day of the week (e.g. Mon/Tue/Wed)
  - The time of day (e.g. morning/afternoon/evening)



datetime module (2 of 2)

Enter: Python's datetime module, which lets us get the current date and time

```
>>> from datetime import datetime
>>> now = datetime.now()
>>> now
datetime.datetime(2026, 8, 4, 15, 4, 48, 971283)
>>> now.hour
15
>>> now.weekday()
1
```

Task 4

- [5 min] Implement Task 4 so that your playlist title is generated dynamically (spec)
- Run tests for this task:

```
uv run pytest tests/test_api.py -k test_daylist_title
```

- Challenge: Generate the playlist title based on the songs sampled for the daylist

Solution Walkthrough (Task 4)

Learn More

If you liked this lecture, check out these courses:

- **CS 160:** User Interface Design & Development
- **CS 169A/L:** Software Engineering