Jerry Cain
CS 106AX

Handout #25
November 22, 2024

Assignment #8: Flutterer

This assignment was developed by Ryan Eberhardt, Jonathan Kula, Esteban Rey, Suzanne Joh, and Anand Shankar in 2019 when CS106AX was first offered. ***Remember you're permitted to work with partners on this assignment!***

Due Date: Friday, December 6th, 5:00PM

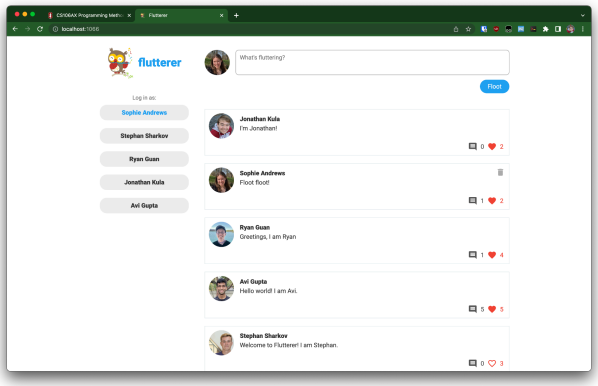
Owl image from pngtree.com

Welcome to Flutterer! In this assignment, you will implement a small but fully-functioning social network that emulates the design of (the admittedly deteriorating) X, formerly known as Twitter. The home page displays a list of posts—or rather, **floots**—made by yourself and friends and allows you to view and comment on them. As part of this, you'll be implementing a server to store and serve those floots to anyone who connects!

The hardest part of this assignment is less in the actual implementation and more in the process of architecting the various parts into a fully integrated application. In particular, this is the first program you'll have worked on that makes use of many separate files at once. In fact, Flutterer is implemented across **fourteen** different files: eight Python files on the backend (aka server-side), and six Javascript files on the frontend (aka client-side), all working in cooperation to present the user experience that is Flutterer. Don't worry about the size of the code base or even the size of this handout, as much of it is code we've already written to support you as you complete the assignment. On the backend, you'll need to implement one Python file, and read two or three others to understand how they work. On the frontend, you'll need to implement very small code segments across 5 JavaScript files.

When Flutterer is complete, you'll have implemented a full-stack application that is capable of:

- Displaying a list of floots to any computer, anywhere in the world
- Displaying comments on individual floots
- Allowing multiple users to post floots to the website that are later seen by everyone
- Allowing multiple users to comment on any floot
- Displaying all the above via an attractive, relatively modern UI



The finished product!
Note: The Flutterer in the assignment code includes the 106AX staff this year.

That may not seem like a lot at first glance, but hiding behind each of those elements are the complexities of

server-client interaction and dynamic content retrieval and modification. By the end of this assignment, you will have implemented something most likely unprecedented in an introductory computer science class, and something to be really proud of! 😊

If you'd like to go beyond the functionality above, Flutterer provides ample opportunity for extensions. We've detailed some ideas for extensions at the bottom of this handout. What's more? The best submission (as decided by Jerry, Ben, Rachel, and the section leaders) will be treated as a contest winner. That means your lowest assignment, midterm, or final exam grade will be dropped and replaced with a 100%.

Logistics

Taking one late day will extend the deadline to Monday, 12/9 at 5:00PM, and a second late day will extend the deadline to Wednesday, 12/11 at 5:00PM. **No submissions will be accepted past Wednesday, 12/11 at 5:00PM.**

Remember that you may work with a partner for this assignment. You should only submit one copy of the assignment and email the section leaders of each mentioning you worked together once one of the two of you submits.

Part 1: Implementing the Server in Python

For the first part of this assignment, we will focus on implementing a *server*. A server is a computer program that provides services—get it?—for other programs (called *clients*) over the Internet. In our case, the clients are web browsers (like Google Chrome) running the Javascript you will write in Part 2. The server maintains a database of floots and comments, and the client can ask the server to do things like create new floots, add comments, delete floots, and so on. Multiple clients can interact with the same server, so if one client asks the server to create a floot and then a different client asks the server for a list of all floots (for the purposes of populating its news feed), the second client will see the floot that the first client created (even though the two clients have no direct contact with each other).

The client and server communicate with each other via a specification called an *API*. The API specifies, in very clear terms, how the client and server should interact so that the server can understand what the client is asking it, and so the client can understand the server's responses. It defines a series of *endpoints*, where each endpoint corresponds to something the client might ask the server to do. For example, if the client wants to retrieve a list of all floots, it can send a request to the **`GET /api/floots`** endpoint. If it wants to delete a floot, it can send a request to the **`POST /api/floots/{flootId}/delete`** endpoint. For each endpoint, the API specifies what information the client needs to send in its request, and it specifies what the server's response should look like. For example, our API says that a client sending a request to **`POST /api/floots`** (an endpoint that creates new floots) needs to include the content of the floot, as well as the

username of the person posting. A server response to that request should contain a JSON-formatted string that includes a variety of information about the newly-created floot.

The full list of Flutterer API endpoints is described in the *API overview* section below. For now, don't worry about understanding all the API endpoints. Just focus on understanding the general ideas involved in this client-server communication.

You may remember learning about the HTTP protocol in lecture, along with various details such as how paths are specified, how headers are sent, and so on. For this assignment, you need to have an understanding of what is happening behind the scenes, but those details have already been handled for you. Your primary job is to write a handful of functions that implement the functionality of the Flutterer API endpoints. This work will be described more concretely in the next few sections.

Summary of starter code

Understanding and then modifying a large codebase can be intimidating, so we'll take some space here to explore Flutterer's files together. Although not all files need to be modified or even read, we still recommend looking through them. They can give you examples of what good decomposition looks like and catalyze an interest in Python concepts we haven't had the time to explore!

You will **not** need to modify the following files, but you will need to use the classes they define. You should take a look at the public methods (i.e., the ones that don't start with an underscore), read the documentation, and understand how they work. You don't need to read or understand the actual code in these files, although you are certainly encouraged to if you have the time.

- **`server/database.py`** defines a **`Database`** class that you will use to store information in your server so that the information persists, even if you restart the server or reboot your computer. There is a handy table describing all of the methods at the end of this handout (see Appendix), which you can print out as a quick reference if you'd like.
- **`server/floot.py`** defines a **`Floot`** class that you will need to use when interacting with the **`Database`**. This class is also described in the Appendix.
- **`server/floot_comment.py`** defines a **`FlootComment`** class that you will also need to use when interacting with the **`Database`**.
- **`server/error.py`** is very short and defines an **`HTTPError`** class that you can use to notify the client of any errors you encounter.

There are a few more files that are worth noting:

- **`server/data.json`** stores all of the information that has been saved to your **`Database`**. If you ever want to clear your database to start afresh, you can delete the file. (Other than

that, you won't need to interact with this file directly, as it's managed for you inside the `Database`.)

- `server/test_api.py` contains a series of tests you can use to ensure your code has been implemented correctly. You don't need to read or understand this file, but you will need to run it. This is covered in the *Testing your server* section. (Don't worry about it for now.)
- `run_server.py` launches a web server that accepts HTTP requests and runs your code to service API requests. Much of this code is complicated, but you don't need to read or understand it. You will, however, need to run this script starting in Part 2 (when you work on implementing the client).
- `server/serve.py` contains the core implementation of our HTTP server. You can read it if you're curious, but you don't need to read or understand anything here.

Lastly, note that `server/api.py` is the only backend file you'll be modifying. This file implements the responses to the various API endpoints defined in the API docs (described below).

How `api.py` works

Let's walk through `server/api.py` together, since this is where you will be doing the bulk of your work. Open the file in PyCharm, and you will see a number of functions with `# TODO` comments in them. These are the functions you need to implement for this first portion of the assignment. At the bottom of the file, you will find this code:

```
GET_ROUTES = [
    ("/api/floots", get_floots),
    (("/api/floots/(.*)", "floot_id"), get_floot),
    (("/api/floots/(.*?)/comments", "floot_id"), get_comments),
    (("(/.*)", "path"), serve_file),
]

POST_ROUTES = [
    (("/api/floots"), create_floot),
    (("/api/floots/(.*?)/comments", "floot_id"), create_comment),
    < 4 more routes omitted for brevity... >
]
```

These two variables define the API endpoints for Flutterer. The first line in `GET_ROUTES` defines the `GET /api/floots` endpoint, and tells the server that the `get_floots` function should be called whenever a client makes a request to that endpoint. The second line is a bit more complicated; it defines an endpoint that has a *path parameter*. If the client sends a request to the server for any URL matching the form `/api/floots/{`*some string*`}` (e.g. `/api/floots/abc` or `/api/floots/123`), our server code will extract the `{`*some string*`}` part from the requested URL (e.g. `abc` or `123`) and pass the extracted string as the `floot_id` parameter to the `get_floot` function. For this reason, do not change the parameter names in the API functions! (Or, if you

update the parameter names, be sure to update the corresponding parameter names in `GET_ROUTES` and `POST_ROUTES`.)

The endpoints defined in `POST_ROUTES` work the same way, with one twist: `POST` requests generally involve the client uploading some data to the server. For example, the `POST /api/floots` endpoint (for creating new floots) requires that the client upload the contents of the floot as part of the request body. Our server code takes the request body that was received from the client, parses it into a dictionary (since it is sent from the client to the server as a string), and passes that dictionary to the specified API function as the `request_body` parameter. For this reason, every API function that is mentioned in `POST_ROUTES` has a `request_body` parameter. To give a concrete example, see the `delete_floot` function. The `floot_id` parameter comes from the path parameter (as specified in `POST_ROUTES`), and the `request_body` parameter is a dictionary that comes from the request body sent by the client.

Your job is to implement the remaining API functions marked with `TODO` comments. As mentioned, these functions accept parameters based on the path parameters and request body; they should return a list or a dictionary, which gets converted to a JSON string by our server code. In some cases, you may want to return an `HTTPError` to the client to indicate that something has gone wrong. For example, in `get_floot`, you may be asked by the client to look up information about a particular floot. However, if the specified `floot_id` does not match that of any floot in the database, you should inform the client of the error by returning

```
HTTPError(404, "no floot with ID " + floot_id + " could be found").
```

Finally, as a last note, observe that a `Database` object has been created for you at the top of `api.py`:

```
db = Database()
```

You will need to use this `Database` object to get floots from the database, create new floots, and so on. You should read through the comments in `database.py` to see how to use this object, and there is also a quick reference at the very end of this handout, in the Appendix.

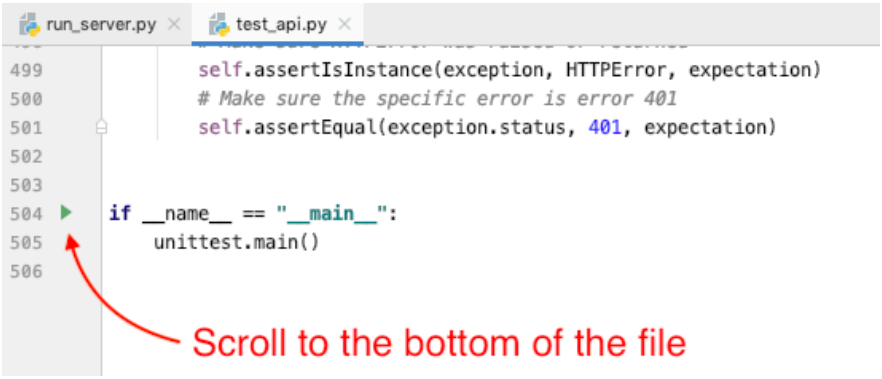
Testing your server

Your API functions are executed whenever a client makes a request to an API endpoint. However, you haven't implemented any client yet! So how are you supposed to test your API code?

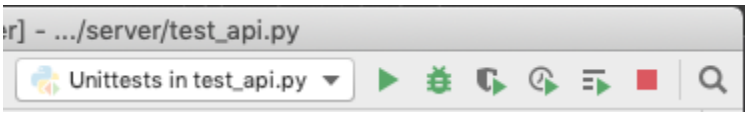
To solve this issue and to make your life simpler, we have provided a series of tests you can run to verify that your functions work correctly. These tests are fairly comprehensive, so, provided you haven't done anything especially bizarre, **passing all the tests means you should get 100% functionality for the server portion of this assignment.** Hopefully, in addition to giving you

some confidence for your functionality grade, these tests will allow you to focus on implementing the client in Part 2 without needing to worry about possible bugs in your server code.

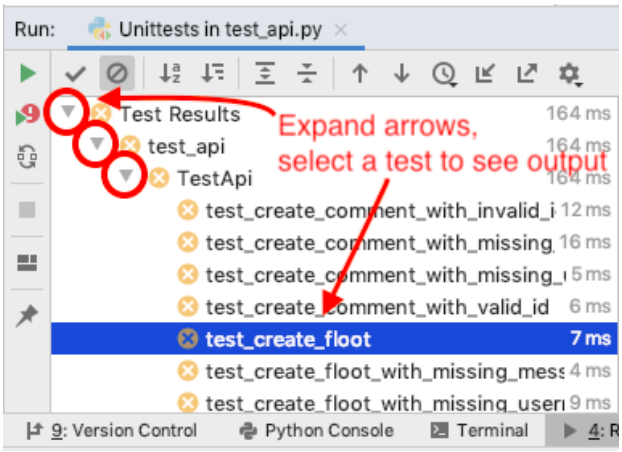
To run tests, open the **`server/test_api.py`** file, scroll to the bottom of the file, and click the "play" button to the left of **`if __name__ == "__main__"`**:



(Once you have done this at least once, you should also be able to launch the tests from the upper-right corner of PyCharm.)



This will run the entire test suite (21 tests) and report any errors it finds. PyCharm will show you the output like so:



Click the arrows to see the list of individual tests, then click on a particular test to see why it failed.

This is what a failure looks like:

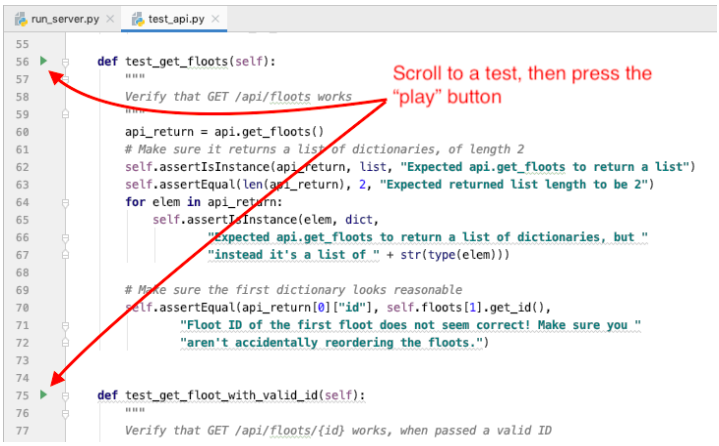
```
Failure
Traceback (most recent call last):
  File "/Library/Frameworks/Python.framework/Versions/3.8/lib/python3.8/unittest/case.py", line 60, in testPartExecutor
    yield
  File "/Library/Frameworks/Python.framework/Versions/3.8/lib/python3.8/unittest/case.py", line 676, in run
    self._callTestMethod(testMethod)
  File "/Library/Frameworks/Python.framework/Versions/3.8/lib/python3.8/unittest/case.py", line 633, in _callTestMethod
    method()
  File "/Users/reberhardt/Documents/Stanford/CS_106AX/flutterer/flutterer/starter/server/test_api.py", line 127, in
 test_create_floot
    self.assertIsInstance(output, dict,
  File "/Library/Frameworks/Python.framework/Versions/3.8/lib/python3.8/unittest/case.py", line 1335, in assertIsInstance
    self.fail(self._formatMessage(msg, standardMsg))
  File "/Library/Frameworks/Python.framework/Versions/3.8/lib/python3.8/unittest/case.py", line 753, in fail
    raise self.failureException(msg)
AssertionError: HTTPError(501, 'api.create_floot not implemented yet') is not an instance of <class 'dict'> : Expected
 api.create_floot to return a dictionary, but instead it returned a <class 'error.HTTPError'>
```

At the top, you'll see an error message that describes what we expected, and what your code did; then, it will summarize the difference. Read the last line first: In this case, **`api.create_floot`** was supposed to return a dictionary, but instead it returned an **`HTTPError`**. If your code outright crashes (instead of doing the wrong thing), the test failure will instead look like this:

```
Error
Traceback (most recent call last):
  File "/Library/Frameworks/Python.framework/Versions/3.8/lib/python3.8/unittest/case.py", line 60, in testPartExecutor
    yield
  File "/Library/Frameworks/Python.framework/Versions/3.8/lib/python3.8/unittest/case.py", line 676, in run
    self._callTestMethod(testMethod)
  File "/Library/Frameworks/Python.framework/Versions/3.8/lib/python3.8/unittest/case.py", line 633, in _callTestMethod
    method()
  File "/Users/reberhardt/Documents/Stanford/CS_106AX/flutterer/flutterer/starter/server/test_api.py", line 69, in
 test_get_floots
    api_return = api.get_floots()
  File "/Users/reberhardt/Documents/Stanford/CS_106AX/flutterer/flutterer/starter/server/api.py", line 46, in get_floots
    raise Exception("Uh oh, your code crashed!")
Exception: Uh oh, your code crashed!
```

In this case, you can read the stack trace from the bottom up to find where the error in your code occurred. (Here, we raised an **`Exception`** to induce an artificial crash for demo purposes.)

The tests are written in the same order as the things they exercise in **`api.py`**, so you can implement one function and then go to **`test_api.py`** and run the tests for just that function. You can execute a single test by clicking the "play" button to the left of the test:



Recommendations for completing the assignment

Our biggest recommendation for this part of the assignment is to **implement one API endpoint, test it, and only move to the next endpoint once the previous one functions correctly.** Do *not* implement all of the endpoints and then try testing things out all at once; your code will likely be broken, and it will take a lot longer to fix everything. We have given you the tests to make it easy to spot bugs early on, so please use them.

In addition, please reach out if you have questions about what's going on in this assignment. This assignment doesn't require you to write much code, but it's conceptually dense, and we want to make sure you understand how all the pieces work together.

API Overview

The server you will implement provides persistent, shared data storage of floots (which are like Twitter posts). Floots have information like the content of a message and who posted it, and are identified by a unique `id`, which looks like this:

```
cace4adc-34b9-49dd-90cd-32f552b7b72f
```

Every floot will have its own unique `id`, which can be used to identify a particular floot among many. Don't worry about generating this `id`; our starter code will take care of that for you.

Of course, this isn't all. Just as tweets can have replies, floots come with comments. Comments are linked to a particular floot (a-la Facebook, Instagram, TikTok, etc.) and are identified by both the floot they're attached to **and** by a separate unique `id` that takes the same format as above.

The API you need to implement is specified here: https://bit.ly/37wtWM7. This is industry-grade API documentation, and if you ever work with an API from Google, Facebook, Slack, or some other company, you will probably read similar documentation. In addition, we have written up a table with the most important information that you can print out and refer to if you'd like. This table is at the very end of this handout, in the Appendix.

Part 2: Implementing the Client in JavaScript

Now that you have a functioning server that is capable of storing data for multiple users, let's build the frontend of the website to interact with it!

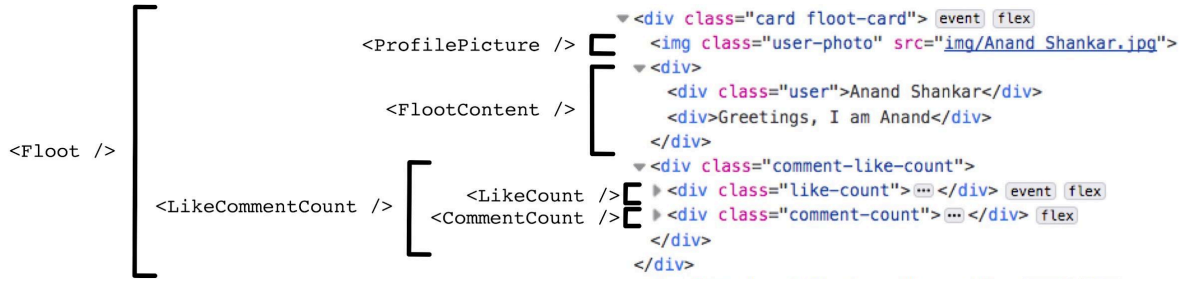
Components

Before we begin talking about the code, it is important to discuss how the code is organized, as well as why it is organized the way it is..

Modern websites are *huge*; my Facebook home page has 4,338 DOM nodes, and the Google Docs document I am currently typing in has 6,883. That's a *lot* of elements, and we need some way to manage the complexity that comes with it.

One way to manage the complexity is to organize DOM nodes into *components*. For example, say these are the DOM nodes needed to display a floot "card" on the screen, including the author's profile photo, the author's name, the contents of the floot, a "like" count, and a "number of comments" count:

We can group these nodes into logical "components":



Then, we can implement each component as a simple function that takes some parameters (e.g., **`ProfilePicture`** would need to know the username of the floot's author) and generates the DOM nodes that would be needed to add that component to the page. As a concrete example, we have implemented a **`ProfilePicture`** component that looks like this:

```
function ProfilePicture(name, imageUrl) {
    let image = document.createElement("img");
    image.src = imageUrl;
    image.className = "user-photo";
    image.alt = "User Profile Image for " + name;
    return image;
}
```

Then, using smaller components, building big components (like the floot as shown in the two pictures above) becomes much simpler:

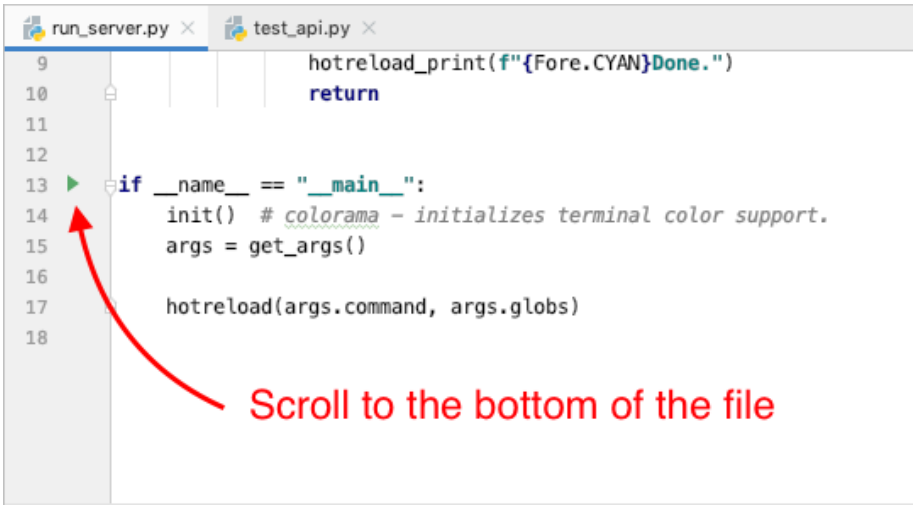
```
function Floot(floot, loggedInUser, actions) {
    let card = document.createElement("div");
```

```
        card.classList.add("card");
        card.classList.add("floot-card");
        card.appendChild(ProfilePicture(floot.username,
                                        "img/" + floot.username + ".jpg"));
        card.appendChild(FlootContent(floot.username, floot.message));
        card.appendChild(LikeCommentCount(floot, loggedInUser, toggleLike));
        return card;
    }
```

All of the DOM generation for this assignment is structured in terms of components. Throughout the rest of this handout, we will use the term "component" extensively, but it's important to remember that a component is *just a normal function* that takes some parameters and returns a DOM node as output.

Starting the server

Launch the server by opening **`run_server.py`** in PyCharm, scrolling to the bottom, and pressing the green "play" button to the left of **`if __name__ == "__main__"`**:



This will run the server and, upon any Python file change, will automatically restart the server so you're working with the latest version of your code. (You shouldn't need to make any changes to your Python code in this part of the assignment, but this will be useful if you want to implement an extension.) To exit, press the stop button.

Once the server is running, you can connect to it by going to http://localhost:1066/ in your web browser, all ready to go! When you make changes to your JavaScript/HTML/CSS files, changes will be reflected automatically – just refresh your browser! There's no need to restart the server.

You may run into an error that complains about "address already in use" or an "error binding port 1066" when running the server. This probably means the server is already running somewhere else (maybe another PyCharm *Run* tab). If it's not or you can't get it to work, you can edit the

`SERVER_PORT` constant in `server/serve.py` and change it to a different number between 1025 and 65534.

Note that unlike previous assignments where you could double-click index.html to run your Javascript, **you must launch the website by going to http://localhost:1066/!** If you open the website by double-clicking index.html, things might appear to work at first, but you will run into problems making `AsyncRequest`s. (If you try making an `AsyncRequest` for `/api/floots` after opening `index.html` directly, the browser will try looking for the file `/api/floots` on your computer instead of sending a request to your Python server.)

Milestone 1: Read the starter code

First, take some time to familiarize yourself with the code that has been provided to you. We wanted to give you a real-world problem for this assignment, but building web applications often involves a significant amount of tedious work (e.g. basic DOM manipulation) that we didn't feel was worth your time in Week 10. We have implemented most of the tedious work for you, we promise. While you don't need to understand how all of it works, you should skim the files to see what code we have already implemented and how to use it.

The following notable files are provided for you:

- `client/index.html`: This is the HTML file that browsers first read when they are loading Flutterer. You might notice that there is very little in this file, as the entire page is dynamically generated from JavaScript. You won't need to touch this file for this assignment; you'll manipulate the HTML by using JavaScript to manipulate the DOM.
- `client/style.css`: This file contains the CSS for Flutterer. There is quite a lot of CSS involved, and we have no expectation that you understand how it works. Feel free to skim this file if you are curious, but you don't need to do anything with this file for the purposes of this assignment.
- `client/img/`: This directory contains all the images that Flutterer can load. You're free to peruse these images, or to add your own! If you want to change the names of the users that show up in Flutterer, you can add new profile pictures in this directory. Just be sure for the name of your image file, that it matches your new username and replace any spaces with `%20`.
- `client/js/flutterer.js`: This is where you will need to write most of your code. The `Flutterer` function is called when the HTML file first loads. You will need to extend this and implement `MainComponent` to generate the contents of the page.
- `client/js/sidebar_components.js`: This file contains the `Sidebar`, `FluttererLogo`, and `AccountSelector` components, which make up the left side of the screen in the finished product. These components are fully implemented, and you won't need to change anything here except for one line in `AccountSelector`. You will need to use this code in Milestone 2.

- **`client/js/floot_components.js`**: This file contains components making up the right side of the screen: **`NewsFeed`**, **`NewFlootEntry`**, **`FlootList`**, **`Floot`**, and a few other small components. **`NewsFeed`** generates the entire right side of the screen via **`NewFlootEntry`** (the text box and submit button at the top of the screen that allow you to create a new floot) and **`FlootList`** (as you might expect, a list of **`Floot`**s). These components are mostly fully implemented, although you will need to make a few one-line changes for Milestones 5-7.
- **`client/js/comment_components.js`**: This file contains components making up the display and entry of comments on a floot: **`NewCommentEntry`**, **`CommentList`**, **`Comment`**, and **`CommentContent`**. Like **`floot_components.js`**, these components are mostly fully implemented. You'll have to make a few one-line changes for Milestones 8 and 9.
- **`client/js/modal_components.js`**: This file contains the single **`FlootModal`** component. A modal is a window-in-a-window; an example of this is the Google Docs "share" screen ("share" modal). In Flutterer, the **`FlootModal`** is responsible for displaying the details of a floot, including that floot's comments. The **`FlootModal`** is already implemented, but you'll have to make a couple changes for Milestone 7.
- **`client/js/general_components.js`**: This file contains several components used throughout the rest of Flutterer, including the **`ProfilePicture`**, **`CommentCount`**, and **`DeleteButton`** components, among others. These are already fully implemented for you.

Milestone 2: Display the basic interface

In this milestone, you'll implement your first component! In fact, it is the only component that we are requiring you to implement on your own. Make sure that you have read the earlier section called *Components*.

You should first implement **`MainComponent`**, which generates the nodes needed to show the entire page. To keep things simple starting out, we'll just generate a container **`div`** with the class **`primary-container`** that contains a **`Sidebar`** and a **`NewsFeed`**. First, create a **`div`** node using **`document.createElement()`**. Add the **`primary-container`** class to the **`div`**, so that it is styled properly using our provided CSS styles. Then, call the **`Sidebar`** and **`NewsFeed`** functions, each of which return a DOM node you can add to your **`div`** by calling *container*`.appendChild(`*sidebar or news feed node*`)`. (You may want to use the **`USERS`** constant when calling **`Sidebar`**.) Finally, return the container **`div`**.

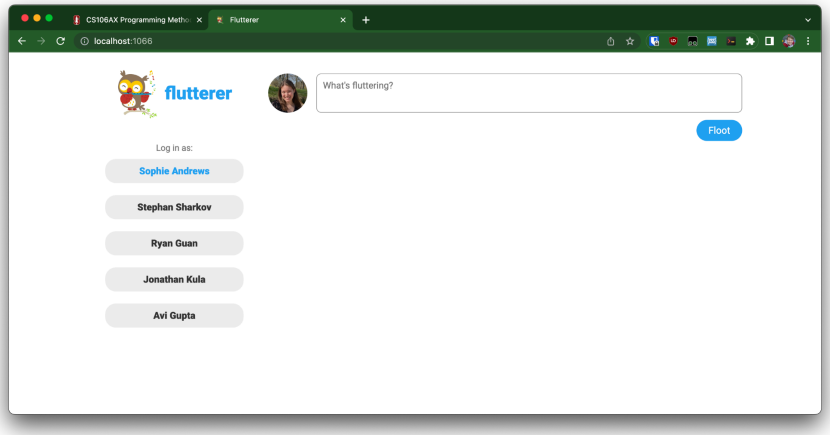
If you would like a reference for implementing **`MainComponent`**, take a look at **`Sidebar`** in **`sidebar_components.js`**. The two functions are very similar.

Once you have done this, update the **`Flutterer`** function to display **`MainComponent`** on the page:

```
document.body.appendChild(MainComponent(selected user, floots, actions));
```

For now, when calling **`MainComponent`**, you can pass **`USERS[0]`** as the selected user, **`[]`** as the list of floots, and **`{}`** as the actions aggregate. We will explain these parameters in milestones 3 and 4, and we can pass better values then.

At the end of this milestone, the left sidebar should appear with the first user highlighted, and the right side of the screen should be populated with a text box that allows you to enter a floot:



This is an exciting result to get for writing less than 20 lines of code! However, it's not magic: As you should have seen in Milestone 1, **`Sidebar`**, **`NewsFeed`**, and the components they depend on are simply doing a lot of mundane DOM manipulation similar to what you did in Assignment 7. CSS is being used to make the page look pretty. You should have a good sense of how everything on this page is working. Hopefully, you feel like you could use the same techniques to implement a different kind of web application without our scaffolding, if given sufficient time.

Be sure that your code generates the above web page before moving on to the next milestone.

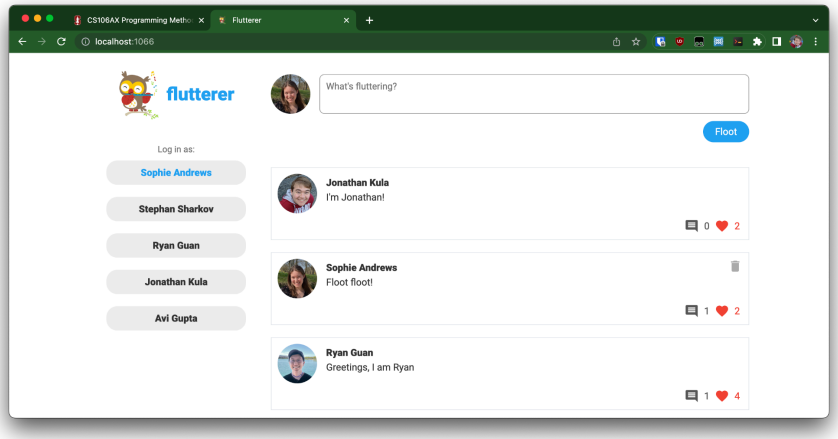
Milestone 3: Load the news feed

In Milestone 2, you passed the empty array (**`[]`**) as the list of floots to **`MainComponent`**. Let's pass in something real! For this milestone, we will add code to **`Flutterer`** to load the list of floots from the **`/api/floots`** API endpoint that you implemented in the first half of this assignment; then, we will pass this to **`MainComponent`**.

Modify the **`Flutterer`** function to create and send an **`AsyncRequest`** to **`/api/floots`**, following lecture and section examples for making asynchronous requests. When the request is completed and the response comes back from the server, use **`JSON.parse()`** to convert the string returned from *response*`.getPayload()` into an array of aggregates. Pass that array to **`MainComponent`** to display the floots returned from the server.

If you run into errors in sending the **`AsyncRequest`**, make sure you are loading the website through http://localhost:1066/ as described earlier in *Starting the server*.

When you are finished, you should see a list of demo floots displayed nicely on the screen:



Again, this leverages a lot of existing code we wrote for you in **`floot_components.js`**, but there are no new concepts in how these DOM nodes are generated, and you should have a good sense of how everything is working.

Milestone 4: Implement user switching

This looks great so far! However, clicking different names in the "Log in as" sidebar has no effect. Let's fix that.

Remember that components are simply functions that take some parameters and return DOM nodes that can be shown on the page. The **`selectedUser`** parameter passed to **`MainComponent`** determines which user is highlighted in the left sidebar and which profile picture appears to the left of the "What's fluttering?" text box in the top right of the page. As such, if we want to change which user is shown as being logged in, one way to do so is to delete all the elements on the page (i.e. remove all the children of **`document.body`**), and then reconstruct the page by passing a different **`selectedUser`** to **`MainComponent`** and displaying the returned elements (by appending the returned DOM node to **`document.body`** as you did in Milestone 2). This process of "clear the screen, regenerate the page, and display it again" happens so fast that the user can't perceive anything was deleted. The only changes they can see to the screen are that the highlighted user and "What's flooting?" profile picture have changed.

Side note: This "let's destroy everything and regenerate everything from scratch" approach might feel lazy and inefficient to you, and I would agree. However, modern web applications are so complicated and have so many things on the page that handling updates to the page in the most efficient way tends to yield extremely complicated code. Tools have been developed over the last

10 years to make this "functional" paradigm ("given some inputs, generate the output DOM from scratch") more efficient so that we can spend more time writing simple functions and less time debugging really complicated code.

To implement this approach, define a function inside **`Flutterer`** that should be called when someone selects a different user to log in as. (You could also define this function as a top-level function, but it will be much easier to define it as a closure function inside **`Flutterer`**.) This function should take one parameter (the name of the user being switched to), and it should:

- Clear anything being currently shown on the page:
```
while (document.body.lastChild != null) {
    document.body.removeChild(document.body.lastChild)
}
```
- Generate a new tree of DOM nodes by calling **`MainComponent`** with the new username
- Show the new nodes using **`document.body.appendChild()`**

You will need to define a top-level variable inside the **`Flutterer`** function that stores the list of floots retrieved from the server in Milestone 3. Otherwise, you won't have a good value to pass as the **`floots`** parameter to **`MainComponent`**.

Once you have done this, the only task that remains is to call this function when one of the users is clicked in the left sidebar. We have installed a click-handling function on each sidebar button (see **`buttonClicked`** inside of **`AccountSelector`** in **`sidebar_components.js`**). You only need to add a single line of code, calling the function you wrote inside buttonClicked. However, this is easier said than done; the function you wrote is a (hopefully) a closure function inside **`Flutterer`**, and it isn't accessible from inside **`AccountSelector`**.

To solve this problem, we have included an **`actions`** parameter in several of the components. This aggregate is intended to contain functions that small components (like **`AccountSelector`**) can use to send information up to **`Flutterer`**, and so that **`Flutterer`** can re-render the page with updated information. Inside **`Flutterer`**, put your function inside of an **`actions`** aggregate:

```
let actions = {
    changeSelectedUser: function(username) {
        // your code here
    },
};
```

Pass this aggregate as the third argument to **`MainComponent`** (instead of passing **`{}`**, as you did in Milestone 2). Then, you should be able to access this function inside of **`buttonClicked`** in **`AccountSelector`**. (You can verify this by doing **`console.log(actions);`** in **`buttonClicked`**.) The function can be retrieved as **`actions.changeSelectedUser`**, so you can call it and pass the clicked username by calling **`actions.changeSelectedUser(username)`**.

By the end of this milestone, you should be able to click different users in the left sidebar. The sidebar should update to reflect the selected user, and the profile picture to the left of the "What's fluttering?" text box should also update.

Milestone 5: Post new floots

In the first half of this assignment, you implemented a **`POST /api/floots`** API endpoint that allows the client to add new floots to the database. For this milestone, when the "Floot" button is clicked, you should make an **`AsyncRequest`** to that endpoint; then, make another **`AsyncRequest`** to **`GET /api/floots`** to get an updated list of floots (including the one that was just created). Use this updated list to create a new MainComponent and update the user's view.

Let's walk through all of that step by step. First, in **`Flutterer`**, add another function to the **`actions`** aggregate that takes the contents of the floot (as a string) as the **`message`** parameter. For now, just put a **`console.log()`** call inside the function so that we can see that it's being called:

```
let actions = {
    changeSelectedUser: /* code from Milestone 4 */,
    createFloot: function(message) {
        console.log("actions.createFloot was called, message:", message);
    },
};
```

Then, let's wire up the code so that function gets called when the "Floot" button is clicked. Have a look at the **`NewFlootEntry`** component inside **`floot_component.js`**. This component creates the "What's fluttering?" text box, as well as the "Floot" button that submits the floot. When the "Floot" button is clicked, the **`postFloot`** function is called. You can access the **`createFloot`** function you just created via the **`actions`** aggregate. Add a line to **`postFloot`** to call that function, passing it **`textbox.value`** (i.e. the contents of the "What's fluttering?" text box).

At this point, you should be able to type some text into the "What's fluttering?" text box, click the "Floot" button, and see that text appear in your browser console. (Try it out, and make sure everything is working. If it doesn't work, **`console.log`** the **`actions`** object at various points to make sure **`actions.createFloot`** is being passed properly.)

Next, add to your **`actions.createFloot`** function so that it sends an **`AsyncRequest`** to **`/api/floots`** using the **`POST`** method. When you are creating the **`AsyncRequest`**, you will need to call *request*`.setMethod("POST");` to make sure the **`POST`** method is used. If you don't do this, the server will think you are trying to **`GET`** a list of floots, and it will send you back a list of all floots (without creating any new floot). Then, you will need to call *request*`.setPayload(`*request body*`);` in order to specify the data for the request (i.e. the **`username`** and **`message`**). Note that **`setPayload`**'s request body parameter takes a string, *not* an object, so you will need to use **`JSON.stringify()`** to convert an aggregate/dictionary to a string that can be sent to the server:

```
request.setPayload(JSON.stringify({
    username: username,
    message: message,
}));
```

When the request finishes (i.e. the callback passed to `request.setSuccessHandler(`*callback*`)` is called), that means the server has saved the new floot. (You should be able to manually refresh the page and see the floot appear.) In the success handler, you should load an updated list of floots (i.e. make another `AsyncRequest` to `GET /api/floots`), and when that second request succeeds, you should reconstruct `MainComponent` to re-render the page with the updated list of floots. Throughout this process, try to find ways to consolidate your code with the code you wrote for Milestone 3 (loading and displaying the news feed) and Milestone 4 (re-rendering the page by creating a new `MainComponent`). Our solution decomposes into three functions:

- A function that loads the list of floots from the server (`GET /api/floots`). This function is called on initial load and whenever a new floot is posted (and it is also called several times for the later milestones).
- A function that serves as the `AsyncRequest` success handler for the previous function (i.e. is called whenever the server responds to `GET /api/floots`). This function extracts the list of floots from the server's response.
- A function that re-renders the `MainComponent` (i.e. clears everything in `document.body` and then adds a new `MainComponent`). This is called from the previous function (i.e. when the server responds with an updated list of floots), but it is also called when the logged-in user is changed, and it is called again in Milestone 7.

You can decompose your code any way that makes the most sense to you; we are only providing this as a reference.

Note that this code is pretty conceptually complicated. This is probably the first time you're working with chains of 3 callback functions, so there is a lot to think about. If things don't work, add gratuitous `console.log()` statements to try to build an intuition for what your program is doing and where it might be deviating from what you expect.

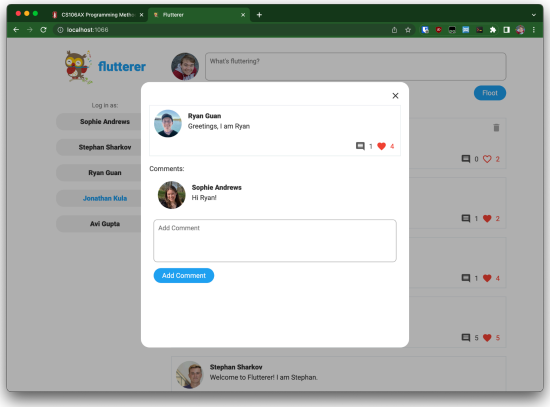
Milestone 6: Delete floots

Next, we'll be giving your users the opportunity to do something seldom allowed by today's social networks: deleting content! You'll notice that your Floots already show a trash can icon on deletable floots (i.e. floots that were posted by the currently logged-in user). Your job is to connect this trash can to the action of deleting a Floot, which you implemented in the `POST /api/floots/{id}/delete` API endpoint on the server. This is extremely similar to the work you did in Milestone 5. To get you started, here are some concrete things you'll need to do:

- Add another function to `actions` that can be called to delete a floot.

- Call this function from inside the **`deleteFloot`** event handler in the **`Floot`** component (**`floot_components.js`**).
- In your function, make a **`POST`** request to **`/api/floots/{id}/delete`**. Note that this endpoint takes a *path parameter* (the ID of the floot you're trying to delete). Make sure you don't do **`let request = AsyncRequest("/api/floots/{id}/delete")`**; you need to build up a target URL string that substitutes **`{id}`** out for the appropriate floot ID. Fortunately, the floot ID is embedded in the **`flootInfo`** parameter passed to the **`Floot`** component function.
- When that request succeeds, refresh the display so that the deleted floot disappears, just like you did in the previous milestone. Make sure to decompose your code well, since this does the exact same thing as Milestone 5.

Milestone 7: Show comments in a FlootModal



When the user clicks on a floot, a *modal* (kind of like a friendly, in-page popup window) appears, displaying the floot's comments and allowing the user to add a new one. We have already implemented the components necessary to create the modal, so your job is to make it appear when a floot is clicked.

To make a modal appear on the page, you can modify **`MainComponent`** to create a **`FlootModal`** (**`FlootModal`** is defined in the file **`modal_components.js`**):

main component container`.appendChild(`
    `FlootModal``(`*floot object*`, `*logged-in username*`, actions));`

Of course, you only want to show a modal if the user has clicked a floot. Somehow, you need to indicate to **`MainComponent`** whether or not a floot has been clicked, so that it can decide whether or not to create a **`FlootModal`**. If a floot has been clicked, you also need to provide that floot object to **`MainComponent`** so that it can be passed to **`FlootModal`** (so that the floot's comments can be displayed).

To address this, we recommend adding a **`selectedFloot`** parameter to **`MainComponent`** containing **`null`** if no floot has been clicked, or a floot object if the modal should be displayed. You can then change your implementation of **`MainComponent`** to include a **`FlootModal`** if **`selectedFloot`** is not **`null`**.

Once this is done, you need to update **`Flutterer`** so that it passes a meaningful value to **`MainComponent`** for this new parameter. You should add two new functions to **`actions`**:

```
let actions = {
    // Earlier code here...
    openFlootInModal: function(flootObject) {
        // Re-render the page, passing flootObject to MainComponent
    },
    closeModal: function() {
        // Re-render the page, passing null as the selectedFloot parameter
        // to MainComponent
    },
}
```

Then, wire up the rest of the code so that the modal opens/closes appropriately:

- In the **`handleCardClick()`** function of the **`Floot`** component (**`floot_components.js`**), call **`actions.openFlootInModal`** to display the given floot.
- In the **`FlootModal`** component (**`modal_components.js`**), after the code that creates **`closeBtn`**, add an event listener to **`closeBtn`** that calls **`actions.closeModal`** on **`click`**.
- Also in the **`FlootModal`** component, towards the very end of the function (the part that calls **`modal.addEventListener`**), add a line that calls **`actions.closeModal()`**. This will close the modal if the user clicks outside the modal.

Milestone 8: Create and delete comments

For the last piece of functionality, you will add the ability to add and remove comments to floots. This is similar to the work you completed for Milestones 5 and 6.

- You need to add functions to **`actions`** that make **`POST`** requests to **`/api/floots/{flootId}/comments`** (to create a comment), in addition to **`/api/floots/{flootId}/comments/{commentId}/delete`** (to delete a comment). When these requests succeed, reload the list of floots from the server and re-render the page. When re-rendering the page, make sure that you also update the **`selectedFloot`** object being passed to MainComponent. If you don't do this, the modal won't update to show the updated list of comments after a user adds a comment, so it will look like commenting isn't working (even though the new comments are being successfully sent to the server).
- In the **`NewCommentEntry`** component (**`comment_components.js`**), you will need to change **`submitComment`** to call your comment-creating function in **`actions`**.
- In the **`Comment`** component (also **`comment_components.js`**), you will need to change **`deleteComment`** to call your comment-deleting function in **`actions`**.

Congratulations! You're done!

General advice

You don't need to write much code for this part of the assignment, but the code you do write can be pretty complicated. We strongly recommend adding `console.log()` statements throughout so that when things go wrong, you can get a better sense of what your program is doing. However, we would like to add two caveats:

- Make your `console.log` statements *descriptive.* If you have too many `console.log` calls, and the messages printed don't explain what they are printing, you will have a hard time interpreting the output.
- Please remove extra `console.log` statements before submitting your assignment. (It's not a big deal if you forget, but removing them makes grading easier.)

Conclusion

By the end of this assignment, you should have a minimal but fully-functioning social network site! You can even use the site from other computers (including your phone). Try identifying your local IP address (Mac, Windows). Pretend your IP address is 12.34.100.200; then, if you have another computer connected to the same WiFi network, you should be able to go to http://12.34.100.200:1066/ and see Flutterer load. (Note that for security reasons, some WiFi networks prevent computers from talking directly to each other, so if this does not work, that may be the reason.)

We hope you have enjoyed seeing how strings, arrays, dictionaries, and objects can interact to form something of this complexity. As it turns out, *every* web application is built using the concepts presented in this class and in this assignment. Some of them use fancy database technologies, some of them use fancy server frameworks, and some of them use fancy client rendering libraries, but fundamentally, they all do the same thing: A server implements some functionality and presents an API that clients can use to interact with it; then, a client written in Javascript will make network requests to exercise that API, and will subsequently update the DOM to display information to the user. We hope this assignment has given you a better sense of how large applications are structured and demystified some of the magic of Google Docs, YouTube, and other common websites.

Extensions

This assignment has many opportunities for creativity. What additions can you add to your version of Flutterer? As mentioned at the beginning of this handout, the best submission (as judged by Jerry, Ben, Rachel, and the section leaders) will be treated as a contest winner.

If you submit an extension, please make a copy of the project and submit a version without any extensions. This way, we can grade your base submission and not penalize you for any

mistakes (functionality or style) that arise from your extension. Also, please add a comment in your extension documenting what changes you made, so that it is easy for us to find your extra features.

Here are some ideas to get you started, sorted (very roughly) in order from least difficulty to greatest difficulty. These are Javascript/CSS-heavy extensions:

- **Show post timestamps:** When each floot is created, a timestamp is saved indicating when the floot was posted. The floots are already sorted by timestamp, but you can modify the `Floot` component to display the timestamp as text on the screen. You may want to add a little bit of CSS to make this pretty.
- **Adding a "like" button:** There are two API endpoints already defined in `api.py` that you can implement, and much of the frontend code for a like button is already provided. You will need to do a bit of work to wire up all the pieces and make everything functional.
- **Adding a "like" button to comments:** If you do the previous extension, you can take this one step further and add a like button to comments. This will probably require a bit of DOM manipulation and CSS, since this was never implemented in our solution.
- **Reacts:** Instead of having a simple like button, you can allow users to react to a floot in several different ways. This will require some extra DOM manipulation and CSS.
- **More secure login:** Add a way for a user to sign up on the website with a username and password. For login, require that they submit the correct password.
- **Other ideas:** Try implementing stickers, "refloots," floot search, hashtags, followers, mentions, notifications, more...

Here are some Python-heavy extension ideas:

- **Floot sort order:** Instead of sorting the floots by the time at which they were posted, experiment with some more interesting criteria. For example, Facebook sorts posts by "relevance" (upweighting posts from friends that you talk to a lot, and upweighting posts with keywords like "congratulations" or "life update").
- **Do some data analysis on the floots:** For those interested in data processing and visualization, you might try something like generating diagrams of which users interact the most, times of day where posts are heavy, etc.

Appendix: Printable Reference

List of API endpoints

Method	Path	Description
GET	`/api/floots`	Returns a list of Floots from the database.

POST	`/api/floots`	Given a new Floot to post (in the request body), puts it into the database and returns it.
		
		Request Body:
		`{`
		`  "message": "string",`
		`  "username": "string"`
		`}`

GET	`/api/floots/{id}`	Returns a particular Floot, identified by the given id. Returns an HTTPError 404 if that id doesn't exist.

POST	`/api/floots/{id}/delete`	Deletes a particular Floot, identified by the given id. Returns an HTTPError 404 if that id doesn't exist, or an HTTPError 403 if the user isn't allowed to delete that Floot (by trusting the username they send in the body). *Security Note: This is clearly terrible security practice; you should never trust that the user is who they say they are without some kind of verification. However, for the sake of simplicity, we decide to simply trust the client.
		
		Request Body:
		`{`
		`  "username": "string"`
		`}`

GET	`/api/floots/{floot_id}/comments`	Returns a list of comments on a Flood identified by floot_id. Returns an HTTPError 404 if the floot isn't found.

POST	`/api/floots/{floot_id}/comments`	Given a new comment to post (in the request body) on a particular Floot (identified by the floot_id parameter), adds that comment to the Floot, then returns the new comment (including its new id). Returns an HTTPError 404 if the floot isn't found.
		
		Request Body:
		`{`
		`  "message": "string",`
		`  "username": "string"`

		`}`

POST	**`/api/floots/{floot_id}/comments/{id}/delete`**	Given a particular comment (identified by the floot_id it's a part of, and the id of the comment), deletes that comment. Returns "OK" if successful. Returns an HTTPError 404 if the floot isn't found, or an HTTPError 403 if the user isn't allowed to delete that comment.
		
		Request Body:
		`{`
		`  "username": "string"`
		`}`

How to use the `Database`

We have provided the **`Database`** class to do persistent data storage for you, so that saved data stays saved even if you restart your computer. When the Database class is constructed, it loads all of the previously-created floots, comments, users, etc. from **`server/data.json`**. Then, every time you call **`db.save_floot(`***floot*`)`**, the floot is saved back to the hard drive so that it can be loaded again later.

Here are some methods you may find helpful:

Member Name	*Description*
`db = Database()`	Calls the constructor to create a new **`Database`**. You won't need to call this yourself; a **`Database`** is already created for you at the top of **`api.py`**.
`floots = db.get_floots()`	Returns a list of **`Floot`** objects (see below) stored in the database, sorted from newest to oldest.
`present = db.has_floot(`*id*`)`**	Returns **`True`** if a **`Floot`** with the provided id exists in the database; returns **`False`** otherwise.
`floot = db.get_floot_by_id(`*id*`)`**	Returns the **`Floot`** object with the provided id. Raises a **`KeyError`** if the id is invalid.
`db.save_floot(`*floot*`)`**	Saves the provided **`Floot`** object into the database. You can call this multiple times on the same floot to replace old data and save the floot's latest version. Important: you'll need to call this method to re-save a **`Floot`** if you add or remove comments from that **`Floot`**.
`db.delete_floot_by_id(`*id*`)`**	Deletes the **`Floot`** object with the provided id. Raises a

	`KeyError` if the id is invalid.
`db.delete_floot(`*floot***`)`**	Deletes the provided **`Floot`** object. Raises a **`KeyError`** if the provided **`Floot`** is not in the database.

We also provide a **`Floot`** class in **`server/floot.py`**, outlined below:

Member Name	*Description*
`floot = Floot(`*message*`,`** ***username*`)`**	Calls the constructor to create a new **`Floot`**.
`timestamp = floot.get_timestamp()`	Returns a timestamp (string) of when the **`Floot`** was created.
`comments = floot.get_comments()`	Returns a list of **`FlootComments`** associated with this **`Floot`**.
`floot_id = floot.get_id()`	Returns the id (string) that uniquely identifies this **`Floot`**.
`username = floot.get_username()`	Returns the username of this **`Floot`**'s creator
`floot.delete_comment(`*comment*`,`** ***username*`)`**	Deletes the provided **`FlootComment`** if it was written by the provided username. Raises a **`KeyError`** if the provided comment isn't present in this **`Floot`**, and raises a **`PermissionError`** if the provided comment wasn't written by the provided username. Make sure to call **`db.save_floot(floot)`** after deleting the comment, so that the change gets saved to the database.
`floot.create_comment(`*comment*`)`**	Adds the provided **`FlootComment`** to this **`Floot`**. Make sure to call **`db.save_floot(floot)`** after adding the comment, so that the comment gets saved to the database.
`floot.set_liked(`*username*`,`** ***liked*`)`**	Accepts a username (string) and boolean value and notes that the given user likes (or doesn't like) this **`Floot`**. Make sure to call **`db.save_floot(floot)`** after changing the liked status.
`liked_by = floot.get_liked_by()`	Returns a list of usernames (strings) that like this **`Floot`**.
`num_likes = floot.get_num_likes()`	Returns the number of users that like this **`Floot`**.

`as_dict = floot.to_dictionary()`	Returns a dictionary where the keys are field names and the values are the values of the fields. Use this if you want a dictionary representing a **`Floot`**, which is useful since dictionaries are JSON-encodable.

There's also a **`FlootComment`** class, which we've opted to omit here. Take a look at **`server/floot_comment.py`** to learn more!