

EECS485 P3: Client-side Dynamic Pages

Due 11:59pm ET February 19th, 2023. This is a group project to be completed in groups of two to three.

Change log

Initial Release for W23

- 2023-01-31: Change required Node version from 10 to 12 in spec, fix `starter_files/package.json` and `starter_files/package-lock.json` to be compatible with Node 12.
- 2023-01-31: Update tree commands and add `cypress.config.js` to output.
- 2023-01-31: Fix `babel-loader` dependency in `starter_files/package.json` to be compatible with Node 12.
- 2023-01-31: Add a [new tutorial](#) about data fetching with `useEffect`.
- 2023-02-02: Add a pitfall in [code style](#) about using a dependency array and cleanup function. Add note about race conditions in [data fetching tutorial](#).
- 2023-02-06: Fix autograder issue that caused student `./bin/insta485run` to throw an error.
- 2023-02-07: Update grep command in the [React Tutorial](#)

Introduction

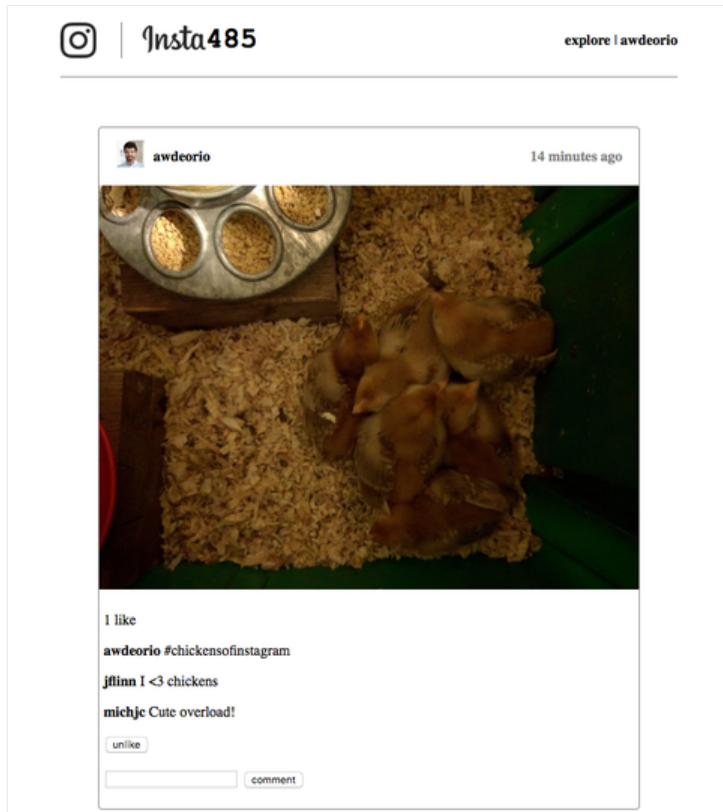
An Instagram clone implemented with client-side dynamic pages. This is the third of an EECS 485 three project sequence: a static site generator from templates, server-side dynamic pages, and client-side dynamic pages.

Build an application using client-side dynamic pages and a REST API. Reuse server-side code from project 2, refactoring portions of it into a REST API. Write a client application in JavaScript that runs in the browser and makes AJAX calls to the REST API.

The learning goals of this project are client-side dynamic pages, JavaScript programming, asynchronous programming (AJAX), and REST APIs. You'll also gain more practice with the command line.

Server-side vs Client-side Dynamic Pages

When you finish this project, the main page should look just like it did in [Project 2](#). While Project 2 used server-side dynamic pages, Project 3 will use client-side dynamic pages.



Server-side dynamic pages example

Project 2 used server-side dynamic pages. Each time a client made a request to the server, a Python function ran on the server. The output of the function was a string containing HTML. The client loads the HTML into the Document Object Model (DOM).

The Python function run by the server returns an HTML-formatted string.

```
1 @insta485.app.route('/')
2 def show_index():
3     # Get posts from the database
4     cur = connection.execute(...)
5     context = cur.fetchall()
6
7     # Fill out template and return HTML formatted string
8     return flask.render_template("index.html", **context)
```

With server-side dynamic pages, you'll see that the HTML source and the DOM are very similar.

HTML	DOM
------	-----

```

<span class="usericon"><a href="/u/awdeorio/"
<span class="username"><a href="/u/awdeorio/"
<span class="userinfo"><a href="/p/3/">5 days
</div><!-- top -->

<div class="middle">
  
  ▼ <a href="/u/awdeorio/">
    
  <a href="/u/awdeorio/">awdeorio</a>
</span>
▼ <span class="userinfo">
  <a href="/p/3/">5 days ago</a>
</span>

```

Client-side dynamic pages example

Project 3 uses client-side dynamic pages. The first time a client loads Insta485, the server responds with a small amount of HTML that links to a larger JavaScript program. The client then runs the JavaScript program, which modifies the DOM.

The JavaScript code run by the client gets data from a REST API and then uses that data to modify the DOM.

```

1  function Post() {
2    const [postId, setPostId] = useState("");
3    useEffect(() => {
4      // Get data from REST API
5      fetch('/api/v1/posts/')
6        .then(response => response.json());
7        .then(json => setPostId(...));
8
9      return () => {
10         // Cleanup effect
11         ...
12       };
13     });
14     // Use data to modify the DOM
15     return <p>{postId}</p>;
16   }

```

With client-side dynamic pages, you'll see that the HTML source is small and references a JavaScript program. You'll also see that the DOM looks a lot like it did with server-side dynamic pages. The difference is that it was created using JavaScript instead of with HTML.

HTML	DOM
------	-----

```

<div id="reactEntry">
  Loading feed ...
</div>
<!-- Load JavaScript -->
<!-- Put this at the end of body re:
dom-elm -->
<script type="text/javascript" src=

  </body>
</html>

```

```

▼ <span class="usericon">
  ▼ <a href="/u/awdeorio/">
    
  <a href="/u/awdeorio/">awdeorio</a>
  </span>
▼ <span class="userinfo">
  <a href="/p/3/">5 days ago</a>
  </span>

```

Why bother with client-side dynamic pages? We can implement some really nice user interface features that are impossible with server-side dynamic pages. Here are a few examples:

1. Click “like” or “comment” without a page reload or redirection
2. Infinite scroll
3. Double-click to like

Setup

Group registration

Register your group on the [Autograder](#).

AWS account and instance

You will use Amazon Web Services (AWS) to deploy your project. AWS account setup may take up to 24 hours, so get started now. Create an account, launch and configure the instance. Don’t deploy yet. [AWS Tutorial](#).

Project folder

Create a folder for this project. Your folder location might be different.

```

1  $ pwd
2  /Users/awdeorio/src/eecs485/p3-insta485-clientside

```

⚠ Pitfall: Avoid paths that contain spaces. Spaces cause problems with some command line tools.

Bad Example	Good Example
EECS 485/Project 3 Insta485 Client-side	eeecs485/p3-insta485-clientside

⚠️ WSL Pitfall: Avoid project directories starting with `/mnt/c/`. This shared directory is slow.

Bad Example	Good Example
<code>/mnt/c/ ...</code>	<code>/home/awdeorio/ ...</code>

Version control

Set up version control using the [Version control tutorial](#).

Be sure to check out the [Version control for a team](#) tutorial.

After you're done, you should have a local repository with a "clean" status and your local repository should be connected to a remote GitHub repository.

```

1  $ pwd
2  /Users/awdeorio/src/eeecs485/p3-insta485-clientside
3  $ git status
4  On branch main
5  Your branch is up-to-date with 'origin/main'.
6
7  nothing to commit, working tree clean
8  $ git remote -v
9  origin  https://github.com/awdeorio/p3-insta485-clientside.git (fetch)
10 origin  https://github.com/awdeorio/p3-insta485-clientside.git (push)

```

You should have a `.gitignore` file ([instructions](#)).

```

1  $ pwd
2  /Users/awdeorio/src/eeecs485/p3-insta485-clientside
3  $ head .gitignore
4  This is a sample .gitignore file that's useful for EECS 485 projects.
5  ...

```

Python virtual environment

Create a Python virtual environment using the Project 1 [Python Virtual Environment Tutorial](#).

Check that you have a Python virtual environment, and that it's activated (remember `source env/bin/activate`).

```
1 $ pwd
2 /Users/awdeorio/src/eecs485/p3-insta485-clientside
3 $ ls -d env
4 env
5 $ echo $VIRTUAL_ENV
6 /Users/awdeorio/src/eecs485/p3-insta485-clientside/env
```

⚠ WARNING Anaconda and `pip` don't play nice together. If you run into issues with your Python virtual environment, uninstall Anaconda completely and [restart](#) the Python virtual environment tutorial.

Install utilities

Windows Subsystem for Linux (WSL)

⚠ WARNING You must have WSL2 installed; WSL1 will not work for this project.

Start a Windows PowerShell. Verify that you are using WSL 2. Your Ubuntu version may be different.

```
1 PS > wsl -l -v
2      NAME                                STATE      VERSION
3 *  Ubuntu-20.04                          Running     2
```

Start a Bash shell and install other utilities needed for this project.

```
1 $ sudo apt-get install sqlite3 curl
2 $ pip install httpie
```

Linux

```
1 $ sudo apt-get install sqlite3 curl
2 $ pip install httpie
```

MacOS

```
$ brew install sqlite3 curl httpie coreutils
```

Starter files

Download and unpack the starter files.

```
1 $ pwd
2 /Users/awdeorio/src/eecs485/p3-insta485-clientside
3 $ wget https://eecs485staff.github.io/p3-insta485-clientside/starter_files.tar.gz
4 $ tar -xvzf starter_files.tar.gz
```

Move the starter files to your project directory and remove the original `starter_files/` directory and tarball.

```
1 $ pwd
2 /Users/awdeorio/src/eecs485/p3-insta485-clientside
3 $ mv starter_files/* .
4 $ mv starter_files/./* .
5 $ rm -rf starter_files starter_files.tar.gz
```

You should see these files.

```
1 $ tree -a -I '.git|env|__pycache__|*.egg-info|node_modules'
2 .
3 ├── cypress.config.js
4 ├── package-lock.json
5 ├── package.json
6 ├── requirements.txt
7 ├── pyproject.toml
8 ├── sql
9 |   └── uploads
10      ...
11 |   └── e1a7c5c32973862ee15173b0259e3efdb6a391af.jpg
12 ├── tests
13      ...
14 |   └── utils.py
15 ├── tsconfig.json
16 ├── .eslintrc.js
17 ├── .prettierrc.json
18 ├── .prettierignore
19 └── webpack.config.js
```

Here's a brief description of each of the starter files.

`cypress.config.js`

Config for the frontend testing framework Cypress

<code>package-lock.json</code>	JavaScript packages with dependencies
<code>package.json</code>	JavaScript packages
<code>requirements.txt</code>	Python package dependencies matching autograder
<code>pyproject.toml</code>	Insta485 python package configuration
<code>sql/uploads/</code>	Sample image uploads
<code>tests/</code>	Public unit tests
<code>tsconfig.json</code>	Optional TypeScript config
<code>.eslintrc.js</code>	Config for the JavaScript linter ESLint
<code>.prettierrc.json</code>	Config for the JavaScript formatter Prettier
<code>.prettierignore</code>	Lists files that Prettier should ignore
<code>webpack.config.js</code>	JavaScript bundler config

Before making any changes to the clean starter files, it's a good idea to make a commit to your Git repository.

Copy project 2 code

You'll reuse much of your code from project 2. Copy these files and directories from project 2 to project 3:

- Scripts
 - `bin/insta485db`
 - `bin/insta485run`
 - `bin/insta485test`
- Server-side version of Insta485
 - `insta485/`
- Database SQL files
 - `sql/schema.sql`
 - `sql/data.sql`

Do not copy:

- Virtual environment files `env/`
- Python package files `insta485.egg-info/`

Your directory should now look like this:

```
1  $ pwd
2  /Users/awdeorio/src/eecs485/p3-insta485-clientside
3  $ tree -a -I '.git|env|__pycache__|*.egg-info|node_modules'
4  .
5  ├── bin
6  |   ├── insta485db
7  |   ├── insta485run
8  |   └── insta485test
9  ├── cypress.config.js
10 ├── insta485
11 |   ├── __init__.py
12 |   ├── config.py
13 |   ├── model.py
14 |   ├── static
15 |   |   ├── css
16 |   |   |   └── style.css
17 |   |   └── images
18 |   |       └── logo.png
19 |   ├── templates
20 |   |   ...
21 |   |   └── index.html
22 |   └── views
23 |       ├── __init__.py
24 ├── package-lock.json
25 ├── package.json
26 ├── pyproject.toml
27 ├── requirements.txt
28 ├── sql
29 |   ├── data.sql
30 |   ├── schema.sql
31 |   └── uploads
32 |   |   ...
33 |   |   └── e1a7c5c32973862ee15173b0259e3efdb6a391af.jpg
34 ├── tests
35 |   |   ...
36 |   |   └── util.py
37 ├── tsconfig.json
38 ├── .eslintrc.js
39 ├── .prettierrc.json
40 ├── .prettierignore
41 └── webpack.config.js
```

Use pip to install the `insta485` package and the exact same third party packages as are installed on the autograder.

```
1 $ pip install -r requirements.txt
2 $ pip install -e .
```

Run your project 2 code and make sure it still works by navigating to <http://localhost:8000/>.

```
1 $ ./bin/insta485db reset
2 $ ./bin/insta485run
```

Commit these changes and push to your Git repository.

REST API

The [Flask REST API Tutorial](#) will show you how to create a small REST API with Python/Flask.

Run the Flask development server.

```
$ ./bin/insta485run
```

Navigate to <http://localhost:8000/api/v1/posts/1/>. You should see this JSON response, which is a simplified version of what you'll [implement later](#).

```
1 {
2   "created": "2017-09-28 04:33:28",
3   "imgUrl": "/uploads/122a7d27ca1d7420a1072f695d9290fad4501a41.jpg",
4   "owner": "awdeorio",
5   "ownerImgUrl": "/uploads/e1a7c5c32973862ee15173b0259e3efdb6a391af.jpg",
6   "ownerShowUrl": "/users/awdeorio/",
7   "postShowUrl": "/posts/1/",
8   "url": "/api/v1/posts/1/"
9 }
```

Commit these changes and push to your Git repository.

REST API tools

The [REST API Tools Tutorial](#) will show you how to use `curl` and HTTPie (the `http` command) to test a REST API from the command line.

You should now be able make a REST API call from the command line. The response below is a simplified version of what you'll [implement later](#).

```
1 $ http \
```

```
2  -a awdeorio:password \
3  "http://localhost:8000/api/v1/posts/1/"
4  HTTP/1.0 200 OK
5  ...
6  {
7    "created": "2017-09-28 04:33:28",
8    "imgUrl": "/uploads/122a7d27ca1d7420a1072f695d9290fad4501a41.jpg",
9    "owner": "awdeorio",
10   "ownerImgUrl": "/uploads/e1a7c5c32973862ee15173b0259e3efdb6a391af.jpg",
11   "ownerShowUrl": "/users/awdeorio/",
12   "postShowUrl": "/posts/1/",
13   "url": "/api/v1/posts/1/"
14 }
```

React/JS

The [React/JS Tutorial](#) will get you started with a development toolchain and a “hello world” React app.

After completing the tutorial, you will have local JavaScript libraries and tools installed. Your versions may be different.

```
1  $ ls -d node_modules
2  node_modules
3  $ echo $VIRTUAL_ENV
4  /Users/awdeorio/src/eecs485/p3-insta485-clientside/env
5  $ node --version
6  v15.2.1
7  $ npm --version
8  7.0.12
```

More tools written in JavaScript were installed via `npm`. Your versions may be different.

```
1  $ npx webpack --version
2  4.41.5
3  $ npx eslint --version
4  v6.8.0
5  $ npx prettier --version
6  2.7.1
```

Build the front end using `webpack` and then start a Flask development server.

```
1  $ npx webpack
2  $ ./bin/insta485run
```

i Pro-tip: Start `webpack` in watch mode to automatically rebuild the front end when changes are detected in the JavaScript source files.

```
$ npx webpack --watch
```

Browse to <http://localhost:8000/> where you should see the test “Post” React Component.

Commit these changes and push to your Git repository.

React/JS Debugging

Learn how to use a JavaScript debugger and a React debugging extension with [React/JS Debugging Tutorial](#).

End-to-end testing

The [End-to-end Testing Tutorial](#) describes how to test a website implemented with client-side dynamic pages. It goes through an example that doesn’t depend on your project.

Install script

Installing the tool chain requires a lot of steps! Write a bash script `bin/install` to install your app. Don’t forget to check for [shell script pitfalls](#).

- Remember the shebang

```
#!/bin/bash
```

- Stop on errors, print commands

```
1 set -Eeuo pipefail
2 set -x
```

- Create a Python virtual environment

```
python3 -m venv env
```

- Activate Python virtual environment

```
source env/bin/activate
```

- Install back end

```
1 pip install -r requirements.txt
```

```
2 pip install -e .
```

- Install front end

```
npm ci .
```

Remember to add `bin/insta485install` to your Git repo and push.

⚠ WARNING Do **not** commit automatically generated or binary files to your Git repo! They can cause problems when running the code base on other computers, e.g., on AWS or a group member's machine. These should all be in your `.gitignore`.

- `env/`
- `insta485.egg-info`
- `node_modules`
- `__pycache__`
- `bundle.js`
- `tmp`
- `cookies.txt`
- `session.json`
- `var`

Fresh install

These instructions are useful for a group member installing the toolchain after checking out a fresh copy of the code.

Check out a fresh copy of the code and change directory.

```
1 $ git clone <your git URL here>
2 $ cd p3-insta485-clientside/
```

If you run into trouble with packages or dependencies, you can delete these automatically generated files.

```
1 $ pwd
2 /Users/awdeorio/src/eecs485/p3-insta485-clientside
3 $ rm -rf env/ node_modules/ insta485.egg-info/ insta485/static/js/bundle.js
```

Run the installer created during the setup tutorial.

```
$ ./bin/insta485install
```

Activate the newly created virtual environment.

```
$ source env/bin/activate
```

That's it!

Database

Use the same database schema and starter data as in the [Project 2 Database instructions](#).

After copying `data.sql` and `schema.sql` from project 2, your `sql/` directory should look like this.

```
1 $ tree sql
2 sql
3 |— data.sql
4 |— schema.sql
5 |— uploads
6   ...
7   |— e1a7c5c32973862ee15173b0259e3efdb6a391af.jpg
```

Database management shell script

Reuse your same database management shell script (`insta485db`) from [project 2](#). Your script should already support these subcommands:

```
1 $ insta485db create
2 $ insta485db destroy
3 $ insta485db reset
4 $ insta485db dump
```

Add the `insta485db random` subcommand, which will generate 100 posts in the database each with owner `awdeorio` and a random photo (selected from the starter photos).

Here is a bash snippet that adds 100 posts to the database each with owner `awdeorio` and a random photo. **Note:** you will not need to modify this bash snippet, but you will need to add the `random` subcommand to your bash script.

```
1 SHUF=shuf
2 # If shuf is not on this machine, try to use gshuf instead
3 if ! type shuf 2> /dev/null; then
4     SHUF=gshuf
5 fi
6 DB_FILENAME=var/insta485.sqlite3
7 FILENAMES="122a7d27ca1d7420a1072f695d9290fad4501a41.jpg
```

```

8         ad7790405c539894d25ab8dcf0b79eed3341e109.jpg
9         9887e06812ef434d291e4936417d125cd594b38a.jpg
10        2ec7cf8ae158b3b1f40065abfb33e81143707842.jpg"
11    for i in `seq 1 100`; do
12        # echo $FILENAMES          print string
13        # shuf -n1                  select one random line from multiline input
14        # awk '{$1=$1;print}'       trim leading and trailing whitespace
15
16        # Use '${SHUF}' instead of 'shuf'
17        FILENAME=`echo "$FILENAMES" | ${SHUF} -n1 | awk '{$1=$1;print}'`
18        OWNER="awdeorio"
19        sqlite3 -echo -batch ${DB_FILENAME} "INSERT INTO posts(filename, owner)
VALUES('${FILENAME}', '${OWNER}');"
20    done

```

MacOS: the `insta485db` random code above using the `shuf` (or `gshuf`) command-line utility, which not installed by default. Install the `coreutils` package, which includes `gshuf`.

```
$ brew install coreutils
```

REST API Specification

This section describes the REST API implemented by the server. It implements the functionality needed to implement the main `insta485` page. Completing the REST API is a small portion of the time it takes to complete this project, so be sure to plan plenty of time for the client-side dynamic pages [portion](#) of the project.

Before beginning, make sure you've completed the [REST API Tutorial](#).

Access control

Most routes require an authenticated user. All REST API routes requiring authentication should work with either session cookies (like Project 2) or [HTTP Basic Access Authentication](#).

Every REST API route should return `403` if a user is not authenticated. The only exception is `/api/v1/`, which is publicly available.

⚠ Warning: Always use HTTPS for user login pages. Never use HTTP, which transfers a password in plaintext where a network eavesdropper could read it. For simplicity, this project uses HTTP only.

Routes

The following table describes each REST API method.

HTTP Method	Example URL	Action
GET	/api/v1/	Return API resource URLs
GET	/api/v1/posts/	Return 10 newest post urls and ids
GET	/api/v1/posts/?size=N	Return N newest post urls and ids
GET	/api/v1/posts/?page=N	Return N'th page of post urls and ids
GET	/api/v1/posts/?postid_lte=N	Return post urls and ids no newer than post id N
GET	/api/v1/posts/<postid>/	Return one post, including comments and likes
POST	/api/v1/likes/?postid=<postid>	Create a new like for the specified post id
DELETE	/api/v1/likes/<likeid>/	Delete the like based on the like id
POST	/api/v1/comments/?postid=<postid>	Create a new comment based on the text in the JSON body for the specified post id
DELETE	/api/v1/comments/<commentid>/	Delete the comment based on the comment id

GET /api/v1/

Return a list of services available. The output should look exactly like this example. Does not require user to be authenticated.

```
1 $ http "http://localhost:8000/api/v1/"
2 HTTP/1.0 200 OK
3 ...
4 {
5   "comments": "/api/v1/comments/",
6   "likes": "/api/v1/likes/",
7   "posts": "/api/v1/posts/",
8   "url": "/api/v1/"
```



```
9 }
```

You should now pass one unit test.

```
$ pytest -vv tests/test_rest_api_simple.py::test_resources
```

GET /api/v1/posts/

Return the 10 newest posts. The posts should meet the following criteria: each post is made by a user which the logged in user follows or the post is made by the logged in user. The URL of the next page of posts is returned in `next` . Note that `postid` is an int, not a string.

```
1 $ http -a awdeorio:password "http://localhost:8000/api/v1/posts/"
2 HTTP/1.0 200 OK
3 ...
4 {
5   "next": "",
6   "results": [
7     {
8       "postid": 3,
9       "url": "/api/v1/posts/3/"
10    },
11    {
12      "postid": 2,
13      "url": "/api/v1/posts/2/"
14    },
15    {
16      "postid": 1,
17      "url": "/api/v1/posts/1/"
18    }
19  ],
20  "url": "/api/v1/posts/"
21 }
```

Authentication

HTTP Basic Access Authentication should work (see [later section](#)). This is true for every route with the exception of `/api/v1/` .

```
1 $ http -a awdeorio:password "http://localhost:8000/api/v1/posts/"
2 $ pytest tests/test_rest_api_simple.py::test_http_basic_auth
```

Authentication with session cookies should also work. This is true for every route with the exception of `/api/v1/` .

```
1 $ http \
2   --session=./session.json \
3   --form POST \
4   "http://localhost:8000/accounts/" \
5   username=awdeorio \
6   password=password \
7   operation=login
8 $ http \
9   --session=./session.json \
10  "http://localhost:8000/api/v1/posts/"
11 $ pytest tests/test_rest_api_simple.py::test_login_session
```

Pagination

Request results no newer than `postid` with `?postid_lte=N`. This is useful later in the situation where a user adds a new post while another user is scrolling, triggering a REST API call via the infinite scroll mechanism. When `postid_lte` is not specified, default to the most recent `postid`.

```
1 $ http -a awdeorio:password "http://localhost:8000/api/v1/posts/?postid_lte=2"
2 HTTP/1.0 200 OK
3 ...
4 {
5   "next": "",
6   "results": [
7     {
8       "postid": 2,
9       "url": "/api/v1/posts/2/"
10    },
11    {
12       "postid": 1,
13       "url": "/api/v1/posts/1/"
14    }
15  ],
16  "url": "/api/v1/posts/?postid_lte=2"
17 }
```

Request a specific number of results with `?size=N`.

```
1 $ http -a awdeorio:password "http://localhost:8000/api/v1/posts/?size=1"
2 HTTP/1.0 200 OK
3 ...
4 {
5   "next": "/api/v1/posts/?size=1&page=1&postid_lte=3",
6   "results": [
```

```
7      {
8          "postid": 3,
9          "url": "/api/v1/posts/3/"
10     }
11 ],
12 "url": "/api/v1/posts/?size=1"
13 }
```

Request a specific page of results with `?page=N` .

```
1 $ http -a awdeorio:password "http://localhost:8000/api/v1/posts/?page=1"
2 HTTP/1.0 200 OK
3 ...
4 {
5     "next": "",
6     "results": [],
7     "url": "/api/v1/posts/?page=1"
8 }
```

Put `postid_lte` , `size` and `page` together.

```
1 $ http -a awdeorio:password "http://localhost:8000/api/v1/posts/?
  postid_lte=2&size=1&page=1"
2 HTTP/1.0 200 OK
3 ...
4 {
5     "next": "/api/v1/posts/?size=1&page=2&postid_lte=2",
6     "results": [
7         {
8             "postid": 1,
9             "url": "/api/v1/posts/1/"
10        }
11    ],
12    "url": "/api/v1/posts/?postid_lte=2&size=1&page=1"
13 }
```

Both `size` and `page` must be non-negative integers. Hint: let Flask coerce to the integer type in a query string like this: `flask.request.args.get("size", default=<some number>, type=int)` .

```
1 $ http -a awdeorio:password "http://localhost:8000/api/v1/posts/?page=-1"
2 HTTP/1.0 400 BAD REQUEST
3 ...
4 {
5     "message": "Bad Request",
```

```
6     "status_code": 400
7   }
8   $ http -a awdeorio:password "http://localhost:8000/api/v1/posts/?size=-1"
9   HTTP/1.0 400 BAD REQUEST
10  ...
11  {
12    "message": "Bad Request",
13    "status_code": 400
14  }
```

HINT: Use a SQL query with `LIMIT` and `OFFSET`, which you can compute from the `page` and `size` parameters.

i Pro-tip: Returning the newest posts can be tricky due to the fact that all the posts are generated at nearly the same instant. If you tried to order by timestamp, this could potentially cause 'ties'. Take advantage of the fact that `postid` is automatically incremented in the order of creation.

Some students get confused about when the `next` field should be blank. When a request returns fewer than `size` posts, `next` should be an empty string. Remember that `size` defaults to 10. Some examples:

- A `GET` request is made to `/api/v1/posts/`. There are 9 posts in the database that were made by the logged in user or by users they follow. The `next` field should be an empty string.
- A `GET` request is made to `/api/v1/posts/`. There are 10 posts in the database that were made by the logged in user or by users they follow. The newest post returned has a `postid` of 10. The `next` field should be `/api/v1/posts/?size=10&page=1&postid_lte=10`, even though a `GET` request to that url will return no posts.
- A `GET` request is made to `/api/v1/posts/`. There are 11 posts in the database that were made by the logged in user or by users they follow. The newest post returned has a `postid` of 11. The `next` field should be `/api/v1/posts/?size=10&page=1&postid_lte=11`. A `GET` request to the `next` url will return 1 post in this case.

GET `/api/v1/posts/<postid>/`

Return the details for one post. Example:

```
1 $ http -a awdeorio:password "http://localhost:8000/api/v1/posts/3/"
2 HTTP/1.0 200 OK
3 ...
```

```

4  {
5    "comments": [
6      {
7        "commentid": 1,
8        "lognameOwnsThis": true,
9        "owner": "awdeorio",
10       "ownerShowUrl": "/users/awdeorio/",
11       "text": "#chickensofinstagram",
12       "url": "/api/v1/comments/1/"
13     },
14     {
15       "commentid": 2,
16       "lognameOwnsThis": false,
17       "owner": "jflinn",
18       "ownerShowUrl": "/users/jflinn/",
19       "text": "I <3 chickens",
20       "url": "/api/v1/comments/2/"
21     },
22     {
23       "commentid": 3,
24       "lognameOwnsThis": false,
25       "owner": "michjc",
26       "ownerShowUrl": "/users/michjc/",
27       "text": "Cute overload!",
28       "url": "/api/v1/comments/3/"
29     }
30   ],
31   "comments_url": "/api/v1/comments/?postid=3",
32   "created": "2021-05-06 19:52:44",
33   "imgUrl": "/uploads/9887e06812ef434d291e4936417d125cd594b38a.jpg",
34   "likes": {
35     "lognameLikesThis": true,
36     "numLikes": 1,
37     "url": "/api/v1/likes/6/"
38   },
39   "owner": "awdeorio",
40   "ownerImgUrl": "/uploads/e1a7c5c32973862ee15173b0259e3efdb6a391af.jpg",
41   "ownerShowUrl": "/users/awdeorio/",
42   "postShowUrl": "/posts/3/",
43   "postid": 3,
44   "url": "/api/v1/posts/3/"
45 }

```

NOTE: "created" should not be returned as human-readable from the API.

HINT: `<postid>` must be an integer. Let Flask enforce the integer type in a URL like this:

```
1 @insta485.app.route('/api/v1/posts/<int:postid_url_slug>/')
2 def get_post(postid_url_slug):
```

Likes detail

A `likes` object, which is nested in a `post` object, corresponds to a database row (if any) in the likes table.

If the logged in user likes the post, then the `likeid` in the `url` should be the `likeid` corresponding to the database row storing the logged in user's like of the post. For example, logged in user `awdeorio` likes post id `2` and the `url` of the `likes` object is `/api/v1/likes/4/`.

```
1 {
2     ...
3     "likes": {
4         "lognameLikesThis": true,
5         "numLikes": 2,
6         "url": "/api/v1/likes/4/"
7     },
8     "url": "/api/v1/posts/2/"
9     ...
10 }
```

If the logged in user does not like the post, then the `like url` should be `null`. For example, the logged in user `jflinn` does not like post id `3` and the `url` of the `likes` object is `null`.

```
1 {
2     ...
3     "likes": {
4         "lognameLikesThis": false,
5         "numLikes": 1,
6         "url": null
7     },
8     "url": "/api/v1/posts/3/"
9     ...
10 }
```

POST `/api/v1/likes/?postid=<postid>`

Create one "like" for a specific post. Return 201 on success. Example:

```
1 $ http -a awdeorio:password \
2 POST \
```

```
3  "http://localhost:8000/api/v1/likes/?postid=3"
4  HTTP/1.0 201 CREATED
5  ...
6  {
7    "likeid": 6,
8    "url": "/api/v1/likes/6/"
9  }
```

If the “like” already exists, return the like object with a 200 response.

```
1  $ http -a awdeorio:password \
2    POST \
3    "http://localhost:8000/api/v1/likes/?postid=3"
4  HTTP/1.0 200 OK
5  ...
6  {
7    "likeid": 6,
8    "url": "/api/v1/likes/6/"
9  }
```

DELETE /api/v1/likes/<likeid>/

Delete one “like”. Return 204 on success.

If the `likeid` does not exist, return `404` .

If the user does not own the like, return `403` .

```
1  $ http -a awdeorio:password \
2    DELETE \
3    "http://localhost:8000/api/v1/likes/6/"
4  HTTP/1.0 204 NO CONTENT
5  ...
```

POST /api/v1/comments/?postid=<postid>

Add one comment to a post. Include the ID of the new comment in the return data. Return 201 on success.

HINT: sqlite3 provides a special function to retrieve the ID of the most recently inserted item: `SELECT last_insert_rowid()` .

```
1  $ http -a awdeorio:password \
2    POST \
3    "http://localhost:8000/api/v1/comments/?postid=3" \
```

```
4   text='Comment sent from httpie'
5   HTTP/1.0 201 CREATED
6   ...
7   {
8     "commentid": 8,
9     "lognameOwnsThis": true,
10    "owner": "awdeorio",
11    "ownerShowUrl": "/users/awdeorio/",
12    "text": "Comment sent from httpie",
13    "url": "/api/v1/comments/8/"
14  }
```

The new comment appears in the list now.

```
1  $ http -a awdeorio:password "http://localhost:8000/api/v1/posts/3/"
2  HTTP/1.0 200 OK
3  ...
4  {
5    "comments": [
6      {
7        "commentid": 1,
8        "lognameOwnsThis": true,
9        "owner": "awdeorio",
10       "ownerShowUrl": "/users/awdeorio/",
11       "text": "#chickensofinstagram",
12       "url": "/api/v1/comments/1/"
13     },
14     {
15       "commentid": 2,
16       "lognameOwnsThis": false,
17       "owner": "jflinn",
18       "ownerShowUrl": "/users/jflinn/",
19       "text": "I <3 chickens",
20       "url": "/api/v1/comments/2/"
21     },
22     {
23       "commentid": 3,
24       "lognameOwnsThis": false,
25       "owner": "michjc",
26       "ownerShowUrl": "/users/michjc/",
27       "text": "Cute overload!",
28       "url": "/api/v1/comments/3/"
29     },
30     {
```



```

31     "commentid": 8,
32     "lognameOwnsThis": true,
33     "owner": "awdeorio",
34     "ownerShowUrl": "/users/awdeorio/",
35     "text": "Comment sent from httpie",
36     "url": "/api/v1/comments/8/"
37 }
38 ],
39 "comments_url": "/api/v1/comments/?postid=3",
40 "created": "2021-05-06 19:52:44",
41 "imgUrl": "/uploads/9887e06812ef434d291e4936417d125cd594b38a.jpg",
42 "likes": {
43     "lognameLikesThis": true,
44     "numLikes": 1,
45     "url": "/api/v1/likes/6/"
46 },
47 "owner": "awdeorio",
48 "ownerImgUrl": "/uploads/e1a7c5c32973862ee15173b0259e3efdb6a391af.jpg",
49 "ownerShowUrl": "/users/awdeorio/",
50 "postShowUrl": "/posts/3/",
51 "postid": 3,
52 "url": "/api/v1/posts/3/"
53 }

```

DELETE /api/v1/comments/<commentid>/

Delete a comment. Include the ID of the comment in the URL. Return 204 on success.

If the `commentid` does not exist, return `404` .

If the user doesn't own the comment, return `403` .

```

1  $ http -a awdeorio:password \
2      DELETE \
3      "http://localhost:8000/api/v1/comments/8/"
4  HTTP/1.0 204 NO CONTENT
5  ...

```

The new comment should not appear in the list now.

```

1  $ http -a awdeorio:password "http://localhost:8000/api/v1/posts/3/"
2  HTTP/1.0 200 OK
3  ...
4  {

```

```
5  "comments": [  
6    {  
7      "commentid": 1,  
8      "lognameOwnsThis": true,  
9      "owner": "awdeorio",  
10     "ownerShowUrl": "/users/awdeorio/",  
11     "text": "#chickensofinstagram",  
12     "url": "/api/v1/comments/1/"  
13   },  
14   {  
15     "commentid": 2,  
16     "lognameOwnsThis": false,  
17     "owner": "jflinn",  
18     "ownerShowUrl": "/users/jflinn/",  
19     "text": "I <3 chickens",  
20     "url": "/api/v1/comments/2/"  
21   },  
22   {  
23     "commentid": 3,  
24     "lognameOwnsThis": false,  
25     "owner": "michjc",  
26     "ownerShowUrl": "/users/michjc/",  
27     "text": "Cute overload!",  
28     "url": "/api/v1/comments/3/"  
29   }  
30 ],  
31 "comments_url": "/api/v1/comments/?postid=3",  
32 "created": "2021-05-06 19:52:44",  
33 "imgUrl": "/uploads/9887e06812ef434d291e4936417d125cd594b38a.jpg",  
34 "likes": {  
35   "lognameLikesThis": true,  
36   "numLikes": 1,  
37   "url": "/api/v1/likes/6/"  
38 },  
39 "owner": "awdeorio",  
40 "ownerImgUrl": "/uploads/e1a7c5c32973862ee15173b0259e3efdb6a391af.jpg",  
41 "ownerShowUrl": "/users/awdeorio/",  
42 "postShowUrl": "/posts/3/",  
43 "postid": 3,  
44 "url": "/api/v1/posts/3/"  
45 }
```

HTTP Basic Access Authentication

[HTTP Basic Access Authentication](#) includes a username and password in the headers of every request. Every route that requires authentication should work with either session cookies or HTTP Basic Access Authentication.

Here's an example without HTTP Basic Access Authentication.

```
GET localhost:8000/api/v1/posts/ HTTP/1.1
```

Here's an example with HTTP Basic Access Authentication. It adds an `Authorization` header with the username `awdeorio` and password `password` encoded using [base 64](#), which looks like `YXdkZW9yaW86cGFzc3dvcmQ=`.

```
1 GET localhost:8000/api/v1/posts/ HTTP/1.1
2 Authorization: Basic YXdkZW9yaW86cGFzc3dvcmQ=
```

⚠ Warning: Always use HTTPS with Basic Access Authentication. Never use HTTP because the username and password are sent in cleartext where a network eavesdropper could read it. Base 64 is an encoding, not an encryption algorithm. For simplicity, this project uses HTTP only.

HTTP Basic Auth and HTTPie

Send HTTP basic auth credentials using HTTPie (`http` command). In this example, the username is `awdeorio` and the password is `password`.

```
1 $ http -a awdeorio:password localhost:8000/api/v1/posts/
2 HTTP/1.0 200 OK
3 ...
4 {
5     "results": [
6         ...
7     ]
}
```

The REST API should return `403` if the credentials are wrong. See the [HTTP response codes](#) section for more.

```
1 $ http -a awdeorio:wrongpassword localhost:8000/api/v1/posts/
2 HTTP/1.0 403 FORBIDDEN
3 ...
4 {
5     "message": "Forbidden",
6     "status_code": 403
}
```

```
7 }
```

HTTP Basic Auth and Flask

Here's an example of how to access the username and password sent via HTTP Basic Access Authentication headers from a Flask app. See the [Flask docs](#) for more info.

```
1 username = flask.request.authorization['username']
2 password = flask.request.authorization['password']
```

HTTP Response codes

The Flask documentation has a helpful section on [implementing API exceptions](#). Errors returned by the REST API should take the form:

```
1 {
2     "message": "<describe the problem here>",
3     "status_code": <int goes here>
4     ...
5 }
```

All routes require a login, except `/api/v1/`. Return 403 if user is not logged in.

```
1 $ http 'http://localhost:8000/api/v1/posts/' # didn't send cookies
2 HTTP/1.0 403 FORBIDDEN
3 Content-Length: 52
4 Content-Type: application/json
5 Date: Thu, 11 Feb 2021 02:07:55 GMT
6 Server: Werkzeug/1.0.1 Python/3.10.9
7
8 {
9     "message": "Forbidden",
10    "status_code": 403
11 }
12 $ http "http://localhost:8000/api/v1/" # didn't send cookies
13 HTTP/1.0 200 OK
14 Content-Type: application/json
15 Content-Length: 50
16 Server: Werkzeug/1.0.0 Python/3.7.6
17 Date: Wed, 19 Feb 2020 13:43:43 GMT
18
19 {
20     "comments": "/api/v1/comments/",
21     "likes": "/api/v1/likes/",
```

```
22     "posts": "/api/v1/posts/",
23     "url": "/api/v1/"
24 }
```

Note that requests to user-facing pages should still return HTML. For example, if the user isn't logged in, the `/` redirects to `/accounts/login/`.

```
1  $ http 'http://localhost:8000/'
2  HTTP/1.0 302 FOUND
3  Content-Length: 239
4  Content-Type: text/html; charset=utf-8
5  Date: Thu, 11 Feb 2021 02:11:49 GMT
6  Location: http://localhost:8000/accounts/login/
7  Server: Werkzeug/1.0.1 Python/3.10.9
8
9  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
10 <title>Redirecting...</title>
11 <h1>Redirecting...</h1>
12 <p>You should be redirected automatically to target URL: <a
    href="/accounts/login/">/accounts/login/</a>. If not click the link.
```

Post IDs that are out of range should return a 404 error.

```
1  $ http -a awdeorio:password "http://localhost:8000/api/v1/posts/1000/"
2  HTTP/1.0 404 NOT FOUND
3  ...
4  {
5      "message": "Not Found",
6      "status_code": 404
7  }
8  $ http -a awdeorio:password "http://localhost:8000/api/v1/posts/1000/?size=1"
9  HTTP/1.0 404 NOT FOUND
10 ...
11 {
12     "message": "Not Found",
13     "status_code": 404
14 }
```

Testing

At this point, the REST API autograder tests should pass.

```
$ pytest -v --log-cli-level=INFO tests/test_rest_api_*
```

All returned JSON should conform to standard formatting. Whitespace doesn't matter.

Client-side Insta485

You'll write only the Insta485 Feed on the main page (/) in this project. The other pages (user detail, following, etc.) are server-side dynamic pages copied from Project 2.

Before continuing, review our [React/JS code explained](#) and the [React quick start](#).

Feed

The main page displays a feed of posts. When you're done, it should look just like the server-side version from project 2.

The navigation bar should be rendered server-side, just like project 2. Include a link to / in the upper left hand corner. If not logged in, redirect to /accounts/login/. If logged in, include a link to /explore/ and /users/<user_url_slug> in the upper right hand corner.

The feed of posts should be rendered by client-side JavaScript code. All posts, including comments, likes, photo, data about the user who posted, etc. must be generated from JavaScript.

Here's an outline of the rendered HTML for the main page. Notice that there is no feed content. Rather, there is an entry point for JavaScript to add the feed content.

```
1
2  ...
3  <body>
4    <!-- Plain old HTML and jinja2 nav bar goes here -->
5
6    <!--
7      We will tell React to use this div as it's entry-point for rendering
8      **NOTE**: Make sure to include the "Loading ..." in the div below.
9      This will display before our React code runs and modifies the DOM!
10   -->
11   <div id="reactEntry">
12     Loading ...
13   </div>
14   <!-- Load JavaScript -->
15   <script type="text/javascript" src="{% url_for('static',
16     filename='js/bundle.js') %}"></script>
17
18   ...
19
```

Pro-tip: Start with a React/JS mock-up and hard coded data. Gradually add features, like retrieving data from the REST API, one at a time. See the [Thinking in React docs](#) for a good example.

Pro-tip: You already have a Post component; here's an example of how to [render a list of components](#).

Pitfall: Don't change state directly; [update objects \(or arrays\)](#) in state using a copy. [Spread syntax](#) is helpful here.

```
1 // Concatenate list of posts using spread syntax ("...")
2 setPosts([...posts, ...data.results]);
```

Pitfall: A component should render without errors even if fetched data has not yet arrived. The first render occurs *before fetched data arrives*.

For example, a `Post` component could show “Loading”, and should *not* show an image, like button, or any other data-dependent content until the fetched data arrives. Check out the React docs on [conditional rendering](#).

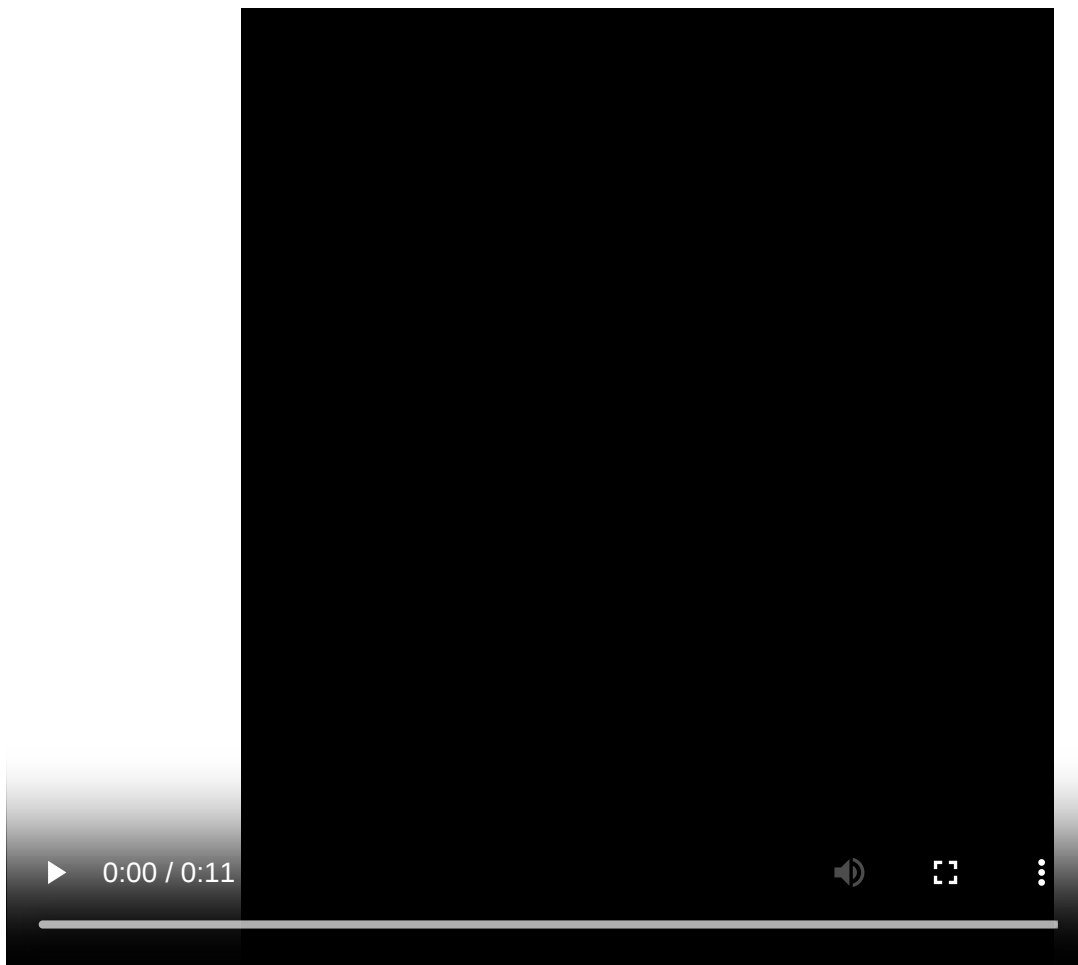
Human readable timestamps

The API call `GET /api/v1/posts/<postId>` returns the created time of a post in the format `Year-Month-Day Hour:Minutes:Seconds`.

Your React code should convert this timestamp into human readable form e.g `a few seconds ago`. You should only use the `moment.js` library, which is already included for you in `package.json`, to achieve this. Using any other libraries, including `react-moment`, will cause you to fail tests on the Autograder.

Likes update

Likes added or deleted by the logged in user should appear immediately on the user interface without a page reload.



[Likes demo video.](#)

i Pro-tip: The React docs on [state](#) have a helpful example of a toggle button that's useful for the like/unlike button. They use ideas from the React docs on [responding to Events](#) and [reacting to input with state](#).

i Pro-tip: If you decide to create a Likes or Comments component, use the [Lifting State Up](#) technique.

The parent Post component stores the state (number of likes) and a state setter function.

The child Likes component displays the number of likes, which is passed as props by the parent Post component.

The child Likes component uses a function reference passed as props by the parent Post component when the Like button is pressed.

⚠️ Pitfall: Avoid copying `props` to `state` ([React docs](#)). This anti-pattern can create bugs because updates to the `prop` won't be reflected in the `state`.

```
1 function Likes({ numLikes }) {  
2   const [numLikesVal, setNumlikesVal] = useState(numLikes); // Don't do  
   this!  
3 }
```

The like button must contain the class name attribute `like-unlike-button` in order for the autograder to find it. Feel free to style this sample code or add other HTML attributes. This applies only to the main page.

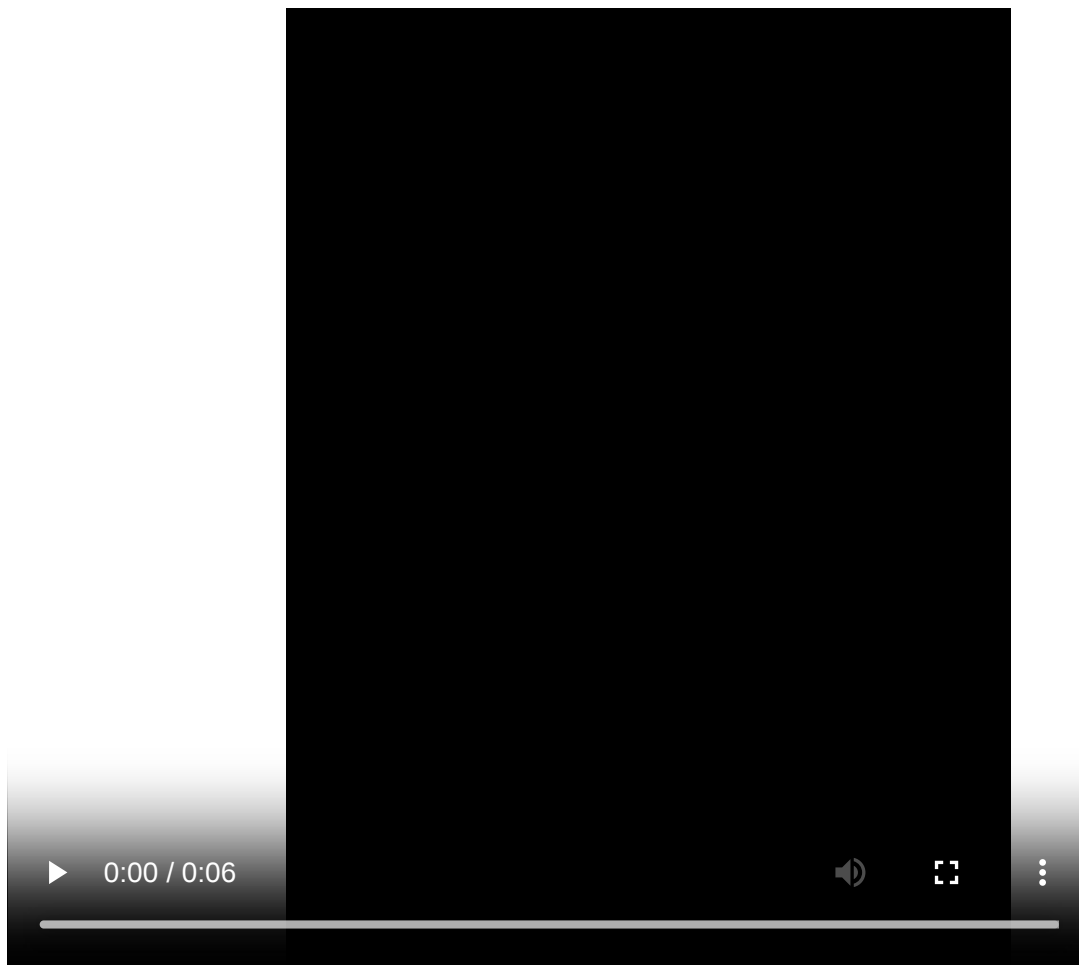
```
1 <button className="like-unlike-button">  
2   FIXME button text here  
3 </button>
```

Comment update

Comments added or deleted by the logged in user should appear immediately on the user interface without a page reload.

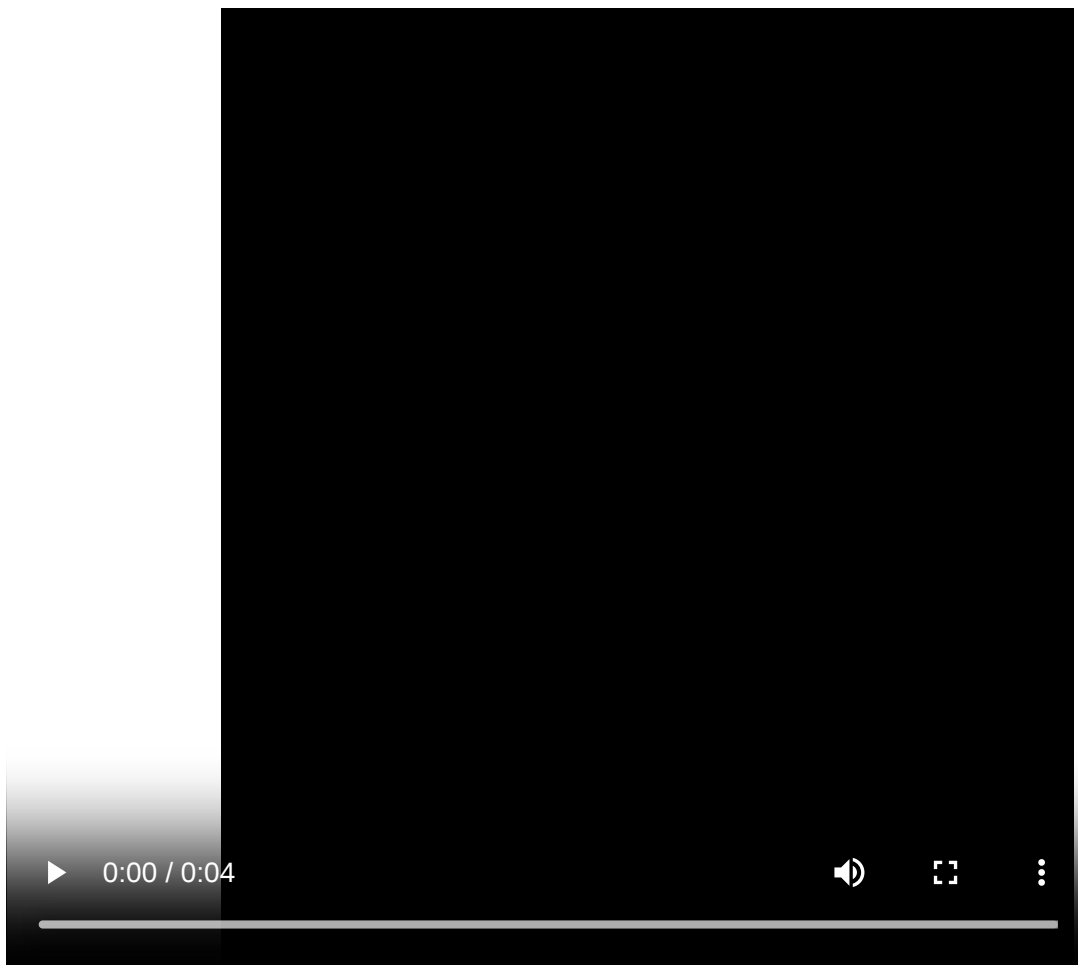
To prevent the page from reloading, check out the React docs' section on [event.preventDefault\(\)](#).

Create a comment by pressing the `enter` (`return`) key. Do not include a comment submit button on the user interface.



[Create comment demo video.](#)

Remove a comment by pressing the delete comment button. Only comments created by the logged-in user should display a delete comment button.



[Delete comment demo video.](#)

i Pro-tip: The React docs on [Reacting to Input with State](#) are helpful for the comment input box. It might also help to read the [API reference page for <input>](#) .

The comment form must contain the class name attribute `comment-form` in order for the autograder to find it. Feel free to style this sample code or add other HTML attributes. This applies only to the main page.

```
1 <form className="comment-form">
2   <input type="text" value=""/>
3 </form>
```

The comment text must contain the class name attribute `comment-text` . You may use this HTML code. Feel free to style it, add other HTML attributes, or use a different type of element instead of `` . This only applies to the main page.

```
<span className="comment-text">FIXME comment text here</span>
```

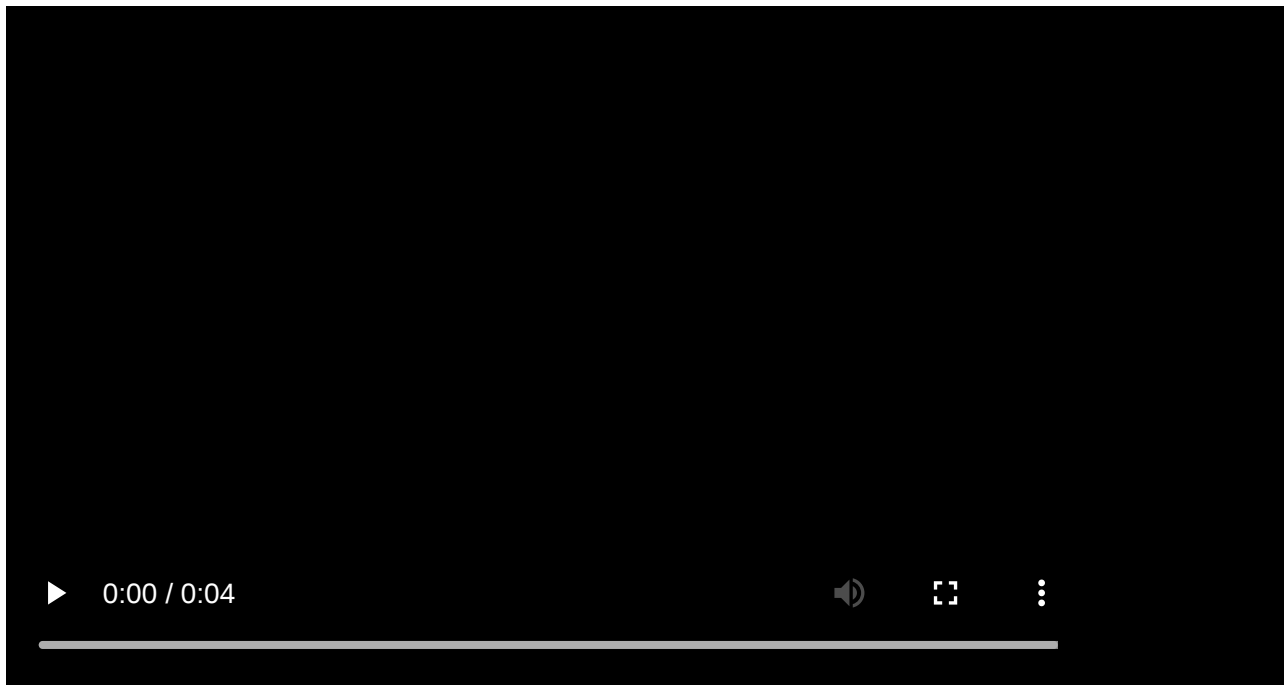
The delete comment button must contain the class name attribute `delete-comment-button` . Feel free to style this sample code or add other HTML attributes. This applies only to the main page. Only comments created by the logged in user should display a delete comment button. This is a new feature in project 3 and applies only to the main page.

```
1 <button className="delete-comment-button">
2   FIXME button text here
3 </button>
```

Double-click to like

Double clicking on an unliked image should like the image. Likes added by double clicking on the image should appear immediately on the user interface without a page reload. The like count and text on the like button should also update immediately.

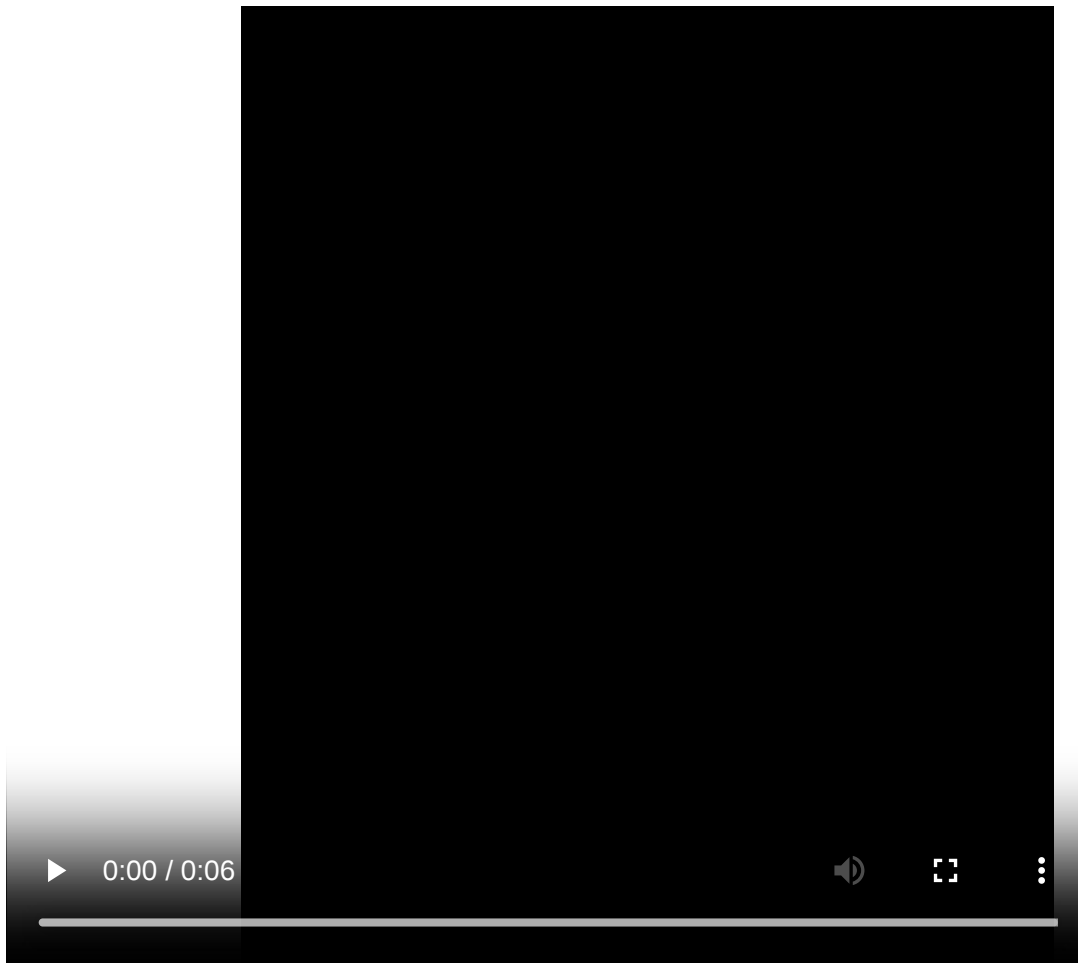
Double clicking on a liked image should do nothing. Do not unlike an image when it is double clicked.



[Double click demo video](#), the heart animation is optional.

Infinite scroll

Scrolling to the bottom of the page causes additional posts to be loaded and displayed. Load and display the next 10 posts as specified by the `next` parameter of the most recent API call to `/api/v1/posts/` . Do not reload the page. You do not need to account for the case of new posts being added while the user is scrolling.



[Infinite scroll demo video.](#)

We recommend the [React Infinite Scroll Component](#), which is already included in `package.json`.

The page should behave the same when reloaded (refreshed) as it does when a logged-in user first lands on the page, regardless of whether or not the infinite scroll mechanism was triggered previously:

1. The scroll position is set to the top of the page.
2. The 10 most recent posts are rendered, including any new posts made before the reload.

i Pro-tip to test this feature, you can use `insta485db random`.

(Note that in some visual demos, we use the same pictures multiple times, which might make you think that infinite scroll should at some point “cycle back to the start.” That’s NOT how it should work. Infinite scroll should keep scrolling until there are no more pictures available.)

REST API calls and logging

We're going to grade your REST API by inspecting the server logs. We'll also be checking that your client-side javascript is making the correct API calls by inspecting the server logs. Loading the main page with the default database configuration, while logged in as awdeorio should yield the following logs:

```
1 127.0.0.1 - - [16/Feb/2022 23:07:47] "GET / HTTP/1.1" 200 -
2 127.0.0.1 - - [16/Feb/2022 23:07:47] "GET /static/js/bundle.js HTTP/1.1" 200 -
3 127.0.0.1 - - [16/Feb/2022 23:07:47] "GET /api/v1/posts/ HTTP/1.1" 200 -
4 127.0.0.1 - - [16/Feb/2022 23:07:47] "GET /api/v1/posts/3/ HTTP/1.1" 200 -
5 127.0.0.1 - - [16/Feb/2022 23:07:47] "GET /api/v1/posts/2/ HTTP/1.1" 200 -
6 127.0.0.1 - - [16/Feb/2022 23:07:47] "GET /api/v1/posts/1/ HTTP/1.1" 200 -
7 127.0.0.1 - - [16/Feb/2022 23:07:47] "GET
/uploads/e1a7c5c32973862ee15173b0259e3efdb6a391af.jpg HTTP/1.1" 200 -
8 127.0.0.1 - - [16/Feb/2022 23:07:47] "GET
/uploads/9887e06812ef434d291e4936417d125cd594b38a.jpg HTTP/1.1" 200 -
9 127.0.0.1 - - [16/Feb/2022 23:07:47] "GET
/uploads/505083b8b56c97429a728b68f31b0b2a089e5113.jpg HTTP/1.1" 200 -
10 127.0.0.1 - - [16/Feb/2022 23:07:47] "GET
/uploads/ad7790405c539894d25ab8dcf0b79eed3341e109.jpg HTTP/1.1" 200 -
11 127.0.0.1 - - [16/Feb/2022 23:07:47] "GET
/uploads/122a7d27ca1d7420a1072f695d9290fad4501a41.jpg HTTP/1.1" 200 -
```

Press the like button a couple of times:

```
1 127.0.0.1 - - [06/May/2021 16:51:50] "DELETE /api/v1/likes/6/ HTTP/1.1" 204 -
2 127.0.0.1 - - [06/May/2021 16:51:50] "POST /api/v1/likes/?postid=3 HTTP/1.1"
201 -
```

Add a comment:

```
127.0.0.1 - - [06/May/2021 16:52:11] "POST /api/v1/comments/?postid=3 HTTP/1.1"
201 -
```

An example of infinite scroll. First, we load the main page from a database populated with 100 random posts.

```
1 127.0.0.1 - - [16/Feb/2022 23:20:04] "GET / HTTP/1.1" 200 -
2 127.0.0.1 - - [16/Feb/2022 23:20:04] "GET /static/js/bundle.js HTTP/1.1" 200 -
3 127.0.0.1 - - [16/Feb/2022 23:20:04] "GET /api/v1/posts/ HTTP/1.1" 200 -
4 127.0.0.1 - - [16/Feb/2022 23:20:04] "GET /api/v1/posts/104/ HTTP/1.1" 200 -
5 127.0.0.1 - - [16/Feb/2022 23:20:04] "GET /api/v1/posts/100/ HTTP/1.1" 200 -
6 127.0.0.1 - - [16/Feb/2022 23:20:04] "GET /api/v1/posts/101/ HTTP/1.1" 200 -
7 127.0.0.1 - - [16/Feb/2022 23:20:04] "GET /api/v1/posts/102/ HTTP/1.1" 200 -
8 127.0.0.1 - - [16/Feb/2022 23:20:04] "GET /api/v1/posts/103/ HTTP/1.1" 200 -
9 127.0.0.1 - - [16/Feb/2022 23:20:04] "GET /api/v1/posts/99/ HTTP/1.1" 200 -
10 127.0.0.1 - - [16/Feb/2022 23:20:04] "GET /api/v1/posts/98/ HTTP/1.1" 200 -
```

```

11 127.0.0.1 - - [16/Feb/2022 23:20:04] "GET /api/v1/posts/96/ HTTP/1.1" 200 -
12 127.0.0.1 - - [16/Feb/2022 23:20:04] "GET /api/v1/posts/95/ HTTP/1.1" 200 -
13 127.0.0.1 - - [16/Feb/2022 23:20:04] "GET /api/v1/posts/97/ HTTP/1.1" 200 -
14 127.0.0.1 - - [16/Feb/2022 23:20:05] "GET
   /uploads/e1a7c5c32973862ee15173b0259e3efdb6a391af.jpg HTTP/1.1" 200 -
15 127.0.0.1 - - [16/Feb/2022 23:20:05] "GET
   /uploads/2ec7cf8ae158b3b1f40065abfb33e81143707842.jpg HTTP/1.1" 200 -
16 127.0.0.1 - - [16/Feb/2022 23:20:05] "GET
   /uploads/ad7790405c539894d25ab8dcf0b79eed3341e109.jpg HTTP/1.1" 200 -
17 127.0.0.1 - - [16/Feb/2022 23:20:05] "GET
   /uploads/122a7d27ca1d7420a1072f695d9290fad4501a41.jpg HTTP/1.1" 200 -
18 127.0.0.1 - - [16/Feb/2022 23:20:05] "GET
   /uploads/9887e06812ef434d291e4936417d125cd594b38a.jpg HTTP/1.1" 200 -

```

Scroll to the bottom and infinite scroll is triggered.

```

1 127.0.0.1 - - [16/Feb/2022 23:21:40] "GET /api/v1/posts/?
   size=10&page=1&postid_lte=104 HTTP/1.1" 200 -
2 127.0.0.1 - - [16/Feb/2022 23:21:40] "GET /api/v1/posts/91/ HTTP/1.1" 200 -
3 127.0.0.1 - - [16/Feb/2022 23:21:40] "GET /api/v1/posts/92/ HTTP/1.1" 200 -
4 127.0.0.1 - - [16/Feb/2022 23:21:40] "GET /api/v1/posts/94/ HTTP/1.1" 200 -
5 127.0.0.1 - - [16/Feb/2022 23:21:40] "GET /api/v1/posts/93/ HTTP/1.1" 200 -
6 127.0.0.1 - - [16/Feb/2022 23:21:40] "GET /api/v1/posts/89/ HTTP/1.1" 200 -
7 127.0.0.1 - - [16/Feb/2022 23:21:40] "GET /api/v1/posts/90/ HTTP/1.1" 200 -
8 127.0.0.1 - - [16/Feb/2022 23:21:40] "GET /api/v1/posts/88/ HTTP/1.1" 200 -
9 127.0.0.1 - - [16/Feb/2022 23:21:40] "GET /api/v1/posts/86/ HTTP/1.1" 200 -
10 127.0.0.1 - - [16/Feb/2022 23:21:40] "GET /api/v1/posts/87/ HTTP/1.1" 200 -
11 127.0.0.1 - - [16/Feb/2022 23:21:40] "GET /api/v1/posts/85/ HTTP/1.1" 200 -

```

Code style

Rendered html text should be in the appropriate tags.

```

1 // good
2 return (<p>{some_bool ? 'Hello' : 'Goodbye'}</p>);
3
4 // bad
5 return (some_bool ? 'Hello' : 'Goodbye');

```

Put unique content in separate tags. This is better code style, and it will prevent issues with the autograder parsing your HTML.

```

1 // good (comment owner and comment are in separate tags)
2 return (
3   <div>

```

```

4     <p>{comment_owner}</p>
5     <p>{comment}</p>
6 </div>
7 );
8
9 // bad (comment owner and comment are in same tag)
10 return (<p>{comment_owner}{comment}</p>);

```

As in project 2, all HTML should be W3C compliant, as reported by `html5validator`. Python code should contain no errors or warnings from `pycodestyle`, `pydocstyle`, and `pylint`. Why `--unsafe-load-any-extension`? [Because](#) the `sqlite3` module is a C extension.).

```

1 $ pycodestyle -v insta485
2 $ pydocstyle -v insta485
3 $ pylint --disable=cyclic-import --unsafe-load-any-extension=y insta485

```

All JavaScript source code should conform to the Airbnb JavaScript coding standard. Use `eslint` to test it. Refer to the [React Tutorial eslint section](#).

```
$ npx eslint --ext jsx insta485/js/
```

All JavaScript source code should also conform to the default Prettier formatting rules. You can also have `prettier` fix formatting automatically. Refer to the [React Tutorial prettier section](#).

```

1 $ npx prettier --check insta485/js # Check
2 $ npx prettier --write insta485/js # Fix

```

You may only use JavaScript libraries that are contained in `package.json` from the starter files and the built-in [Web APIs](#).

You must use the `fetch` API for AJAX calls.

⚠️ Pitfall: Don't forget a *dependency array* and a *cleanup function*. Our [Data Fetching in React with useEffect](#) explains why. Use our [example code](#) as a starting point.

Testing

Make sure that you've completed the [end-to-end testing tutorial](#).

Several end-to-end tests are published with the starter files. Make sure you've copied the `tests` directory. Your files may be slightly different.

```
1 $ ls tests/cypress/e2e
```



```
2 test_index_public.cy.js test_scroll_public.cy.js
  test_slow_server_index_public.cy.js
```

Rebuild your JavaScript bundle and then run your back end development server. Your `./bin/insta485run` may [automatically rebuild](#) the front end bundle.


```
1 $ npx webpack
2 $ ./bin/insta485run
```


Run and debug front end tests. Run this command in a separate terminal from the back end. See the [Run a test](#) section of the End-to-end Testing Tutorial.

```
$ npx cypress open
```

Run all front end tests in headless mode. Run this command in a separate terminal from the back end. We recommend headless mode only as a sanity check before submitting to the autograder.

```
$ npx cypress run --browser chrome
```

 **Pitfall:** Make sure your back end dev server (`./bin/insta485run`) is running in a separate terminal before running Cypress tests.

 **Pro-tip:** In headless mode, Cypress records a video of the test in `tests/cypress/videos` . If a test fails, Cypress records a screenshot at the point of failure in `tests/cypress/screenshots` . Remember to use headless mode sparingly though: Cypress is most useful in the browser.

`insta485test` script

Add JavaScript style checking to your `insta485test` script from project 2. In addition to the tests run in project 2, `insta485test` should run `eslint` and `prettier` on all files within the `insta485/js/` directory. Refer back to the [eslint](#) and [prettier](#) instructions.

```
1 $ ./bin/insta485test
2 + npx eslint --ext jsx insta485/js
3 + npx prettier --check insta485/js
4 ...
```

Running Cypress from your `insta485test` script is not recommended.

Deploy to AWS

You should have already created an AWS account and instance ([instructions](#)). Resume the [Project 2 AWS Tutorial - Deploy a web app](#).

After you have deployed your site, download the main page along with a log. Do this from your local development machine, not while SSH'd into your EC2 instance.

```
1  $ pwd
2  /Users/awdeorio/src/eecs485/p3-insta485-serverside
3  $ hostname
4  awdeorio-laptop # NOT AWS
5  $ curl \
6    --request POST \
7    --cookie-jar cookies.txt \
8    --form 'username=awdeorio' \
9    --form 'password=password' \
10   --form 'operation=login' \
11   "http://<Public DNS (IPv4)>/accounts/"
12 $ curl -v -b cookies.txt "http://<Public DNS (IPv4)>/" > deployed_index.html
13 $ curl -v -b cookies.txt "http://<Public DNS (IPv4)>/static/js/bundle.js" > deployed_bundle.js
```

Be sure to verify that the output in `deployed_index.log` and `deployed_bundle.log` doesn't include errors like "Couldn't connect to server". If it does contain an error like this, it means `curl` couldn't successfully connect with your flask app. Verify that your logs have a `200 OK` status in them, not `302 REDIRECT`.

Also be sure to verify that the output in `deployed_index.html` looks like the `index.html` file you coded while `deployed_bundle.js` contains Javascript code.

Submitting and grading

One team member should register your group on the autograder using the *create new invitation* feature.


Submit a tarball to the autograder, which is linked from <https://eecs485.org>. Include the `--disable-copyfile` flag only on macOS and the `tsconfig.json` file only if you used TypeScript.

```
1  $ tar \
2    --disable-copyfile \
3    --exclude '*__pycache__*' \
```

```
4  -czvf submit.tar.gz \  
5  bin \  
6  insta485 \  
7  package-lock.json \  
8  package.json \  
9  pyproject.toml \  
10 sql \  
11 webpack.config.js \  
12 tsconfig.json \  
13 deployed_index.html \  
14 deployed_index.log \  
15 deployed_bundle.js \  
16 deployed_bundle.log
```

The autograder will run `pip install -e YOUR_SOLUTION` and `cd YOUR_SOLUTION && npm ci .`. The exact library versions in `requirements.txt` and `package-lock.json` are cached on the autograder, so be sure not to add extra library dependencies.

We won't run Project 2 test cases on your Project 3 code, but some of the Project 3 tests rely on a working server-side login.

 **WARNING** The autograder for this project can be slow to grade your submissions. Leave ample time for submitting this project to the autograder.

Requirements and restrictions

The main page should load without errors (exceptions), even when the REST API takes a long time to respond. Put another way, the React framework will render your components *before any AJAX data arrives*. The page should still render without errors, and should not display inaccurate content like an image without an `src` (see [this pitfall](#) for more).

Use the asynchronous Fetch API to make REST API calls from JavaScript ([Mozilla Fetch API documentation](#)). Do not use `jQuery`. Do not use `XMLHttpRequest`.

Rubric

This is an **approximate** rubric.

Tests	Value
Public Unit tests	60%

Tests	Value
Public Python, JS, and HTML style	15%
Hidden unit tests run after the deadline	15%
AWS deployment	10%

FAQ

My JavaScript code doesn't work. What do I do?

1. Make sure it's `eslint` clean. [Instructions here](#).
2. Make sure it's free from exceptions by checking the [developer console](#) for exception messages
3. Try the [React Developer tools](#) Chrome extension
4. Check your assumptions about [how state works](#).
5. Add `console.log()` messages to each relevant component and callback function to identify the problem and its causes.

Do trailing slashes in URLs matter to Flask?

Yes. Use them with the `route` decorator your REST API. See the “Unique URLs / Redirection Behavior” section in the [Flask quickstart](#). Here's a good example:

```
@insta485.app.route("/users/<username_url_slug>/", methods=["GET", "POST"])
```

Can we use `console.log()` ?

Yes. Ideally you should only log in the case of an error.

`eslint` Error ... “is missing in props validation”

You'll probably encounter this error while running `eslint` :

```
1 $ eslint --ext jsx insta485/js/
2 24:38 error 'url' is missing in props validation react/prop-types
```

With `prop-types` , you'll get a nice error in the console when a type property is violated at run time. For example,

```
“Warning: Failed propTypes: The prop url is marked as required in CommentInput , but its value is undefined . Check the render method of Comments .
```

More on the `prop-types` library: <https://www.npmjs.com/package/prop-types>.

Keys in an array of React components

When using a collection of React components, they need to have unique `key` attributes. Otherwise, you'll get an error in your browser console:

Warning: Each child in a list should have a unique "key" prop.

This enables the fast virtual DOM-to-real DOM update performed by React. More info [here](#).

How to fix `pylint` "Similar lines in 2 files"

The REST API shares some code in common with portions of `insta485`'s static pages that haven't been modified. For example, both the REST API and the static `/posts/<postid>/` read the comments and likes from the database. This could lead to `pylint` detecting copy-paste errors.

```
1 ***** Module insta485.views.user
2 R: 1, 0: Similar lines in 2 files
3 ==insta485.api.comments:30
4 ==insta485.views.post:42
```

A nice way to resolve this problem is by adding helper functions to your `model`. The canonical way to solve this problem is an Object Relational Model (ORM), but we're simplifying in this project.

History

In past semesters, a section of this project was "don't break the back button". Prior to the introduction of browser [back/forward cache](#), developers of apps with client-side dynamic pages had to manually code browser back button behavior in JavaScript.

Here's an example of what would happen if an app did *not* implement the browser back button behavior.

1. Load `/` and scroll until the infinite scroll mechanism kicks in, displaying 20 posts.
2. Click on a post to view the post details at `/posts/<postid>/`.
3. Click the back button, returning to `/`. You'll only see 10 posts.

This is not expected behavior for a few reasons. First, when you hit the back button you expect to see the same posts again. Second, they might not even be the same posts because somebody else might have created a new post.

In November 2021, Google Chrome version 96 implemented a [back/forward cache](#) that automatically handles browser history. This made it difficult for students to test if they had correctly implemented the history portion of the project because everything appeared fine when they tested it in their browser. For that reason, we have decided to remove this feature from the project. Long live “don’t break the back button!”

If you are interested in how we ensured correct behavior in past semesters, check out the MDN documentation on the [history API](#) for help in manipulating browser history and the [PerformanceNavigationTiming API](#) for help in checking how the user is navigating to and from a page.

Reach Goals

All reach goals are entirely optional.

TypeScript

If you choose to, you may write your React app in TypeScript instead of JavaScript. TypeScript is a superset of JavaScript (meaning that any valid JavaScript program is a valid TypeScript program), but with strong and static typing. To opt into this, just use `.tsx` and `.ts` file extensions instead of `.jsx` and `.js`.

We’ve already configured the TypeScript configuration file `tsconfig.json` for you, but you may modify it if you wish.

You can also choose to lint your code with TypeScript-specific ESLint rules. To do so, edit your `.eslintrc.js` file and add `"plugin:@typescript-eslint/eslint-recommended"` and `"plugin:@typescript-eslint/recommended"` to the `extends` array in the `overrides` section. Alternatively, add `"@typescript-eslint"` to the `plugins` array and add individual rules to the `rules` array. Then run ESLint with:

```
$ npx eslint --ext 'ts,tsx' insta485/js/
```

On the autograder we will not extend our ESLint configuration from this plugin even if you’re using TypeScript, but you may find some of its rules that we’re not using helpful.

If you’re not already familiar with TypeScript but want to learn it, see the [official TypeScript handbook](#).

EECS 485 staff members may not be able to help with typing issues.

Acknowledgments

Original project written by Andrew DeOrio awdeorio@umich.edu, fall 2017. Updated by the EECS 485 team, February 2019.

This document is licensed under a [Creative Commons Attribution-NonCommercial 4.0 License](https://creativecommons.org/licenses/by-nc/4.0/). You're free to copy and share this document, but not to sell it. You may not share source code provided with this document.