p3-insta485-clientside

EECS485 P3: Client-side Dynamic Pages

Due 11:59pm ET February 20th, 2022. This is a group project to be completed in groups of two to three.

Change log

Initial Release for W22

2/7: Remove reference to Linux file permissions in WSL section

2/8: Fix typo in `DELETE /api/v1/comments/<commentid>/` route section

2/9: Changed `"likeid": "6"` to `"likeid": 6`

2/9: Updated API table to be consistent with this semester's project implementation.

2/10: Fix typo in `DELETE /api/v1/likes/<likeid>/` route section

2/13: Fix Typo in `POST /api/v1/comments/?postid=<postid>` JSON example

2/16: Fix outdated server logs examples

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.

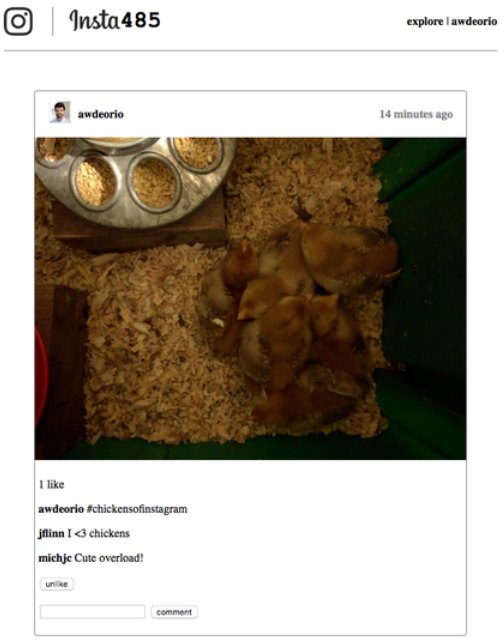
The learning goals of this project are client-side dynamic pages, JavaScript programming, asynchronous programming (AJAX), and REST APIs. You'll also gain more practice with the command line.

After a brief refresher on server-side vs. client-side dynamic pages, this spec will walk you through several parts:

1. Setup
2. REST API Specification
3. Client-side Insta485 specification
4. Testing
5. Deploy to AWS
6. Submitting and grading
7. FAQ

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.

```
@insta485.app.route('/')
def show_index():
    # Get posts from the database
    cur = connection.execute(...)
    context = cur.fetchall()

    # Fill out template and return HTML formatted string
    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
<a href="/u/awdeorio/' <a href="/u/awdeorio/' 5 days <!-- top --> <img src="/uploads/9887e06812ef434d291e493641 <!-- middle -->	▼ ▼ <img src="/uploads/e1a7c5c32973862 ▼ awdeorio ▼ 5 days ago

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.

```
class Post extends React.Component {
  componentDidMount() {
    // Get data from REST API
    fetch('/api/v1/posts/')
    .then(response => response.json());
    .then(json => this.setState(...));
  }
  render() {
    // Use data to modify the DOM
    return <p>{this.state.postid}</p>;
  }
}
```

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-elem -->`
`<script type="text/javascript" src=`
 	
`  </body>`
`</html>`
	▼ `<span class="usericon">`
	▼ `<a href="/u/awdeorio/">`
	`<img src="/uploads/e1a7c5c32973862`
	`</a>`
	`</span>`
	▼ `<span class="username">`
	`<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 (instructions). Your folder location might be different.

```
$ pwd
/Users/awdeorio/src/eecs485/p3-insta485-clientside
```

Verify that the entire project **folder path contains no spaces**. Spaces cause problems with some local tool installations.

```
$ pwd | grep ' ' && echo "ERROR: found a space" || echo "OK: no spaces"
```

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 GitLab repository.

```
$ pwd
/Users/awdeorio/src/eecs485/p3-insta485-clientside
$ git status
On branch main
Your branch is up-to-date with 'origin/main'.

nothing to commit, working tree clean
$ git remote -v
origin  https://gitlab.eecs.umich.edu/awdeorio/p3-insta485-clientside.git (fetch)
origin  https://gitlab.eecs.umich.edu/awdeorio/p3-insta485-clientside.git (push)
```

You should have a `.gitignore` file (instructions).

```
$ pwd
/Users/awdeorio/src/eecs485/p3-insta485-clientside
$ head .gitignore
This is a sample .gitignore file that's useful for EECS 485 projects.
...
```

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`).

```
$ pwd
/Users/awdeorio/src/eecs485/p3-insta485-clientside
$ ls -d env
env
$ echo $VIRTUAL_ENV
/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 10 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.

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

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

```
$ sudo apt-get install sqlite3 curl httpie
```

Linux

```
$ sudo apt-get install sqlite3 curl httpie
```

MacOS

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

Starter files

Download and unpack the starter files.

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

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

```
$ pwd
/Users/awdeorio/src/eecs485/p3-insta485-clientside
$ mv starter_files/* .
$ mv starter_files/.eslintrc.js .
$ rm -rf starter_files starter_files.tar.gz
```

You should see these files.

```
$ tree --matchdirs -I 'env|__pycache__'
.
├── package-lock.json
├── package.json
├── requirements.txt
├── setup.py
├── sql
```

```
│   └── uploads
│       ...
│       └── e1a7c5c32973862ee15173b0259e3efdb6a391af.jpg
├── tests
│   ...
│   └── utils.py
└── webpack.config.js
```

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

`package-lock.json`	JavaScript packages with dependencies
`package.json`	JavaScript packages
`requirements.txt`	Python package dependencies matching autograder
`setup.py`	Insta485 python package configuration
`sql/uploads/`	Sample image uploads
`tests/`	Public unit tests
`webpack.config.js`	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:

```
$ pwd
/Users/awdeorio/src/eecs485/p3-insta485-clientside
$ tree --matchdirs -I 'env|__pycache__'
.
├── bin
│   ├── insta485db
│   ├── insta485run
│   └── insta485test
├── insta485
```

```
│   ├── __init__.py
│   ├── api
│   │   ├── __init__.py
        ...
│   ├── config.py
│   ├── model.py
│   ├── static
│   │   ├── css
│   │   │   └── style.css
│   │   ├── images
│   │   │   └── logo.png
│   │   └── js
│   │       └── bundle.js
│   ├── templates
        ...
│   │   └── index.html
│   └── views
│       ├── __init__.py
├── package-lock.json
├── package.json
├── setup.py
├── sql
│   ├── data.sql
│   ├── schema.sql
│   └── uploads
        ...
│       └── e1a7c5c32973862ee15173b0259e3efdb6a391af.jpg
├── tests
    ...
│   └── util.py
└── webpack.config.js
```

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

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

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

```
$ ./bin/insta485db reset
$ ./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.

```
{
  "created": "2017-09-28 04:33:28",
```

```
  "imgUrl": "/uploads/122a7d27ca1d7420a1072f695d9290fad4501a41.jpg",
  "owner": "awdeorio",
  "ownerImgUrl": "/uploads/e1a7c5c32973862ee15173b0259e3efdb6a391af.jpg",
  "ownerShowUrl": "/users/awdeorio/",
  "postShowUrl": "/posts/1/",
  "url": "/api/v1/posts/1/"
}
```

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.

```
$ http \
  -a awdeorio:password \
  "http://localhost:8000/api/v1/posts/1/"
HTTP/1.0 200 OK
...
{
  "created": "2017-09-28 04:33:28",
  "imgUrl": "/uploads/122a7d27ca1d7420a1072f695d9290fad4501a41.jpg",
  "owner": "awdeorio",
  "ownerImgUrl": "/uploads/e1a7c5c32973862ee15173b0259e3efdb6a391af.jpg",
  "ownerShowUrl": "/users/awdeorio/",
  "postShowUrl": "/posts/1/",
  "url": "/api/v1/posts/1/"
}
```

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.

```
$ ls -d node_modules
node_modules
$ echo $VIRTUAL_ENV
/Users/awdeorio/src/eecs485/p3-insta485-clientside/env
$ node --version
v15.2.1
$ npm --version
7.0.12
```

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

```
$ npx webpack --version
4.41.5
$ npx eslint --version
v6.8.0
```

You can check the style of your code using `eslint`.

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

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

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

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

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.

After completing the tutorial, you should have Google Chrome and Chrome Driver installed. The first part of version should match (`79` in this example). While your versions should match, they might be different than this example.

```
$ google-chrome --version
Google Chrome 79.0.3945.130
$ chromedriver --version
ChromeDriver 79.0.3945.36 (3582db32b33893869b8c1339e8f4d9ed1816f143-refs/branch-heads/3945@{#614})
```

Run an end-to-end test provided with the starter files.

```
$ pytest -v --log-cli-level=INFO tests/test_index.py::test_anything
...
INFO     autograder:conftest.py:77 Setup test fixture 'app'
INFO     autograder:conftest.py:130 Setup test fixture 'base_driver'
INFO     autograder:conftest.py:160 IMPLICIT_WAIT_TIME=10
INFO     autograder:conftest.py:192 Setup test fixture 'driver'
INFO     werkzeug:_internal.py:122  * Running on http://localhost:50504/ (Press CTRL+C to quit)
INFO     werkzeug:_internal.py:122 127.0.0.1 - - [22/Jan/2020 08:25:52] "GET / HTTP/1.1" 302 -
INFO     werkzeug:_internal.py:122 127.0.0.1 - - [22/Jan/2020 08:25:52] "GET /accounts/login/ HTTP/1.1
INFO     werkzeug:_internal.py:122 127.0.0.1 - - [22/Jan/2020 08:25:52] "GET / HTTP/1.1" 302 -
INFO     werkzeug:_internal.py:122 127.0.0.1 - - [22/Jan/2020 08:25:52] "GET /accounts/login/ HTTP/1.1
INFO     werkzeug:_internal.py:122 127.0.0.1 - - [22/Jan/2020 08:25:52] "GET /static/css/style.css HTT
INFO     werkzeug:_internal.py:122 127.0.0.1 - - [22/Jan/2020 08:25:52] "GET /static/images/logo.png H
PASSED                                                                                    ...
```

Install script

Installing the tool chain requires a lot of steps! Write a bash script `bin/insta485install` to install your app. Don't forget to check for shell script pitfalls.

- Remember the shebang

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

- Stop on errors, print commands

```
set -Eeuo pipefail
set -x
```

- Create a Python virtual environment

```
python3 -m venv env
```

- Activate Python virtual environment

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

- Install back end

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

- Install front end

```
npm ci .
```

- Install the latest `chromedriver` using `npm`. This package will automatically install the appropriate `chromedriver` executable for your current OS and Google Chrome version. The `--no-save` argument is passed so that `npm` does not modify the `package.json` file. Note: this **must** be done after `npm ci .` is called.

```
npm install chromedriver --detect_chromedriver_version --no-save
```

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.

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

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

```
$ pwd
/Users/awdeorio/src/eecs485/p3-insta485-clientside
$ 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.

```
$ tree sql
sql
├── data.sql
├── schema.sql
└── uploads
    ...
    └── e1a7c5c32973862ee15173b0259e3efdb6a391af.jpg
```

Database management shell script

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

```
$ insta485db create
$ insta485db destroy
$ insta485db reset
$ 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.

```
SHUF=shuf
# If shuf is not on this machine, try to use gshuf instead
if ! type shuf 2> /dev/null; then
  SHUF=gshuf
fi
DB_FILENAME=var/insta485.sqlite3
FILENAMES="122a7d27ca1d7420a1072f695d9290fad4501a41.jpg
           ad7790405c539894d25ab8dcf0b79eed3341e109.jpg
           9887e06812ef434d291e4936417d125cd594b38a.jpg
           2ec7cf8ae158b3b1f40065abfb33e81143707842.jpg"
for i in `seq 1 100`; do
  # echo $FILENAMES      print string
```

```
    # shuf -n1              select one random line from multiline input
    # awk '{$1=$1;print}'   trim leading and trailing whitespace

    # Use '${SHUF}' instead of 'shuf'
    FILENAME=`echo "$FILENAMES" | ${SHUF} -n1 | awk '{$1=$1;print}'`
    OWNER="awdeorio"
    sqlite3 -echo -batch ${DB_FILENAME} "INSERT INTO posts(filename, owner) VALUES('${FILENAME}','${OWNE
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. You might find this tutorial on REST APIs using Python/Flask helpful. 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.

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

HTTP Method	Example URL	Action
`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.

```
$ http "http://localhost:8000/api/v1/"
HTTP/1.0 200 OK
...
{
  "comments": "/api/v1/comments/",
  "likes": "/api/v1/likes/",
  "posts": "/api/v1/posts/",
  "url": "/api/v1/"
}
```

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.

```
$ http -a awdeorio:password "http://localhost:8000/api/v1/posts/"
HTTP/1.0 200 OK
...
{
  "next": "",
  "results": [
    {
      "postid": 3,
      "url": "/api/v1/posts/3/"
    },
    {
      "postid": 2,
      "url": "/api/v1/posts/2/"
    },
    {
      "postid": 1,
      "url": "/api/v1/posts/1/"
    }
  ],
  "url": "/api/v1/posts/"
}
```

Authentication

HTTP Basic Authentication should work. This is true for every route with the exception of `/api/v1/`.

```
$ http -a awdeorio:password "http://localhost:8000/api/v1/posts/"
$ 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/`.

```
$ http \
  --session=./session.json \
  --form POST \
  "http://localhost:8000/accounts/" \
  username=awdeorio \
  password=password \
  operation=login
$ http \
  --session=./session.json \
  "http://localhost:8000/api/v1/posts/"
$ 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.

```
$ http -a awdeorio:password "http://localhost:8000/api/v1/posts/?postid_lte=2"
HTTP/1.0 200 OK
...
{
  "next": "",
  "results": [
    {
      "postid": 2,
      "url": "/api/v1/posts/2/"
    },
    {
      "postid": 1,
      "url": "/api/v1/posts/1/"
    }
  ],
  "url": "/api/v1/posts/?postid_lte=2"
}
```

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

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

```
  "url": "/api/v1/posts/?size=1"
}
```

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

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

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

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

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)`.

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

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

> **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.

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

Return the details for one post. Example:

```
$ http -a awdeorio:password "http://localhost:8000/api/v1/posts/3/"
HTTP/1.0 200 OK
...
{
  "comments": [
    {
      "commentid": 1,
      "lognameOwnsThis": true,
      "owner": "awdeorio",
      "ownerShowUrl": "/users/awdeorio/",
      "text": "#chickensofinstagram",
      "url": "/api/v1/comments/1/"
    },
    {
      "commentid": 2,
      "lognameOwnsThis": false,
      "owner": "jflinn",
      "ownerShowUrl": "/users/jflinn/",
      "text": "I <3 chickens",
      "url": "/api/v1/comments/2/"
    },
    {
      "commentid": 3,
      "lognameOwnsThis": false,
      "owner": "michjc",
      "ownerShowUrl": "/users/michjc/",
      "text": "Cute overload!",
      "url": "/api/v1/comments/3/"
    }
  ],
  "created": "2021-05-06 19:52:44",
  "imgUrl": "/uploads/9887e06812ef434d291e4936417d125cd594b38a.jpg",
  "likes": {
    "lognameLikesThis": true,
    "numLikes": 1,
    "url": "/api/v1/likes/6/"
  },
  "owner": "awdeorio",
  "ownerImgUrl": "/uploads/e1a7c5c32973862ee15173b0259e3efdb6a391af.jpg",
  "ownerShowUrl": "/users/awdeorio/",
  "postShowUrl": "/posts/3/",
  "postid": 3,
  "url": "/api/v1/posts/3/"
}
```

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:

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

```
{
  ...
  "likes": {
    "lognameLikesThis": true,
    "numLikes": 2,
    "url": "/api/v1/likes/4/"
  },
  "url": "/api/v1/posts/2/"
  ...
}
```

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`.

```
{
  ...
  "likes": {
    "lognameLikesThis": false,
    "numLikes": 1,
    "url": null
  },
  "url": "/api/v1/posts/3/"
  ...
}
```

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

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

```
$ http -a awdeorio:password \
  POST \
  "http://localhost:8000/api/v1/likes/?postid=3"
HTTP/1.0 201 CREATED
...
{
  "likeid": 6,
  "url": "/api/v1/likes/6/"
}
```

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

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

`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`.

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

`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()`.

```
$ http -a awdeorio:password \
  POST \
  "http://localhost:8000/api/v1/comments/?postid=3" \
  text='Comment sent from httpie'
HTTP/1.0 201 CREATED
...
{
  "commentid": 8,
  "lognameOwnsThis": true,
  "owner": "awdeorio",
  "ownerShowUrl": "/users/awdeorio/",
  "text": "Comment sent from httpie",
  "url": "/api/v1/comments/8/"
}
```

The new comment appears in the list now.

```
$ http -a awdeorio:password "http://localhost:8000/api/v1/posts/3/"
HTTP/1.0 200 OK
...
{
  "comments": [
    {
      "commentid": 1,
      "lognameOwnsThis": true,
      "owner": "awdeorio",
      "ownerShowUrl": "/users/awdeorio/",
      "text": "#chickensofinstagram",
      "url": "/api/v1/comments/1/"
    },
    {
      "commentid": 2,
      "lognameOwnsThis": false,
      "owner": "jflinn",
      "ownerShowUrl": "/users/jflinn/",
      "text": "I <3 chickens",
```

```
      "url": "/api/v1/comments/2/"
    },
    {
      "commentid": 3,
      "lognameOwnsThis": false,
      "owner": "michjc",
      "ownerShowUrl": "/users/michjc/",
      "text": "Cute overload!",
      "url": "/api/v1/comments/3/"
    },
    {
      "commentid": 8,
      "lognameOwnsThis": true,
      "owner": "awdeorio",
      "ownerShowUrl": "/users/awdeorio/",
      "text": "Comment sent from httpie",
      "url": "/api/v1/comments/8/"
    }
  ],
  "created": "2021-05-06 19:52:44",
  "imgUrl": "/uploads/9887e06812ef434d291e4936417d125cd594b38a.jpg",
  "likes": {
    "lognameLikesThis": true,
    "numLikes": 1,
    "url": "/api/v1/likes/6/"
  },
  "owner": "awdeorio",
  "ownerImgUrl": "/uploads/e1a7c5c32973862ee15173b0259e3efdb6a391af.jpg",
  "ownerShowUrl": "/users/awdeorio/",
  "postShowUrl": "/posts/3/",
  "postid": 3,
  "url": "/api/v1/posts/3/"
}
```

`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`.

```
$ http -a awdeorio:password \
  DELETE \
  "http://localhost:8000/api/v1/comments/8/"
HTTP/1.0 204 NO CONTENT
...
```

The new comment should not appear in the list now.

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

```
      "commentid": 1,
      "lognameOwnsThis": true,
      "owner": "awdeorio",
      "ownerShowUrl": "/users/awdeorio/",
      "text": "#chickensofinstagram",
      "url": "/api/v1/comments/1/"
    },
    {
      "commentid": 2,
      "lognameOwnsThis": false,
      "owner": "jflinn",
      "ownerShowUrl": "/users/jflinn/",
      "text": "I <3 chickens",
      "url": "/api/v1/comments/2/"
    },
    {
      "commentid": 3,
      "lognameOwnsThis": false,
      "owner": "michjc",
      "ownerShowUrl": "/users/michjc/",
      "text": "Cute overload!",
      "url": "/api/v1/comments/3/"
    }
  ],
  "created": "2021-05-06 19:52:44",
  "imgUrl": "/uploads/9887e06812ef434d291e4936417d125cd594b38a.jpg",
  "likes": {
    "lognameLikesThis": true,
    "numLikes": 1,
    "url": "/api/v1/likes/6/"
  },
  "owner": "awdeorio",
  "ownerImgUrl": "/uploads/e1a7c5c32973862ee15173b0259e3efdb6a391af.jpg",
  "ownerShowUrl": "/users/awdeorio/",
  "postShowUrl": "/posts/3/",
  "postid": 3,
  "url": "/api/v1/posts/3/"
}
```


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=` .

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

```
$ http -a awdeorio:password localhost:8000/api/v1/posts/
HTTP/1.0 200 OK
...
{
    "results": [
        ...
}
```

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

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

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.

```
username = flask.request.authorization['username']
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:

```
{
  "message": "<describe the problem here>",
  "status_code": <int goes here>
  ...
}
```

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

```
$ http 'http://localhost:8000/api/v1/posts/'  # didn't send cookies
HTTP/1.0 403 FORBIDDEN
Content-Length: 52
Content-Type: application/json
Date: Thu, 11 Feb 2021 02:07:55 GMT
```

```
Server: Werkzeug/1.0.1 Python/3.8.5

{
    "message": "Forbidden",
    "status_code": 403
}
$ http "http://localhost:8000/api/v1/"  # didn't send cookies
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 50
Server: Werkzeug/1.0.0 Python/3.7.6
Date: Wed, 19 Feb 2020 13:43:43 GMT

{
  "comments": "/api/v1/comments/",
  "likes": "/api/v1/likes/",
  "posts": "/api/v1/posts/",
  "url": "/api/v1/"
}
```

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/`.

```
$ http 'http://localhost:8000/'
HTTP/1.0 302 FOUND
Content-Length: 239
Content-Type: text/html; charset=utf-8
Date: Thu, 11 Feb 2021 02:11:49 GMT
Location: http://localhost:8000/accounts/login/
Server: Werkzeug/1.0.1 Python/3.8.5

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<title>Redirecting...</title>
<h1>Redirecting...</h1>
<p>You should be redirected automatically to target URL: <a href="/accounts/login/">/accounts/login/</
```

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

```
$ http -a awdeorio:password "http://localhost:8000/api/v1/posts/1000/"
HTTP/1.0 404 NOT FOUND
...
{
  "message": "Not Found",
  "status_code": 404
}
$ http -a awdeorio:password "http://localhost:8000/api/v1/posts/1000/?size=1"
HTTP/1.0 404 NOT FOUND
...
{
  "message": "Not Found",
  "status_code": 404
}
```

Checking output style

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

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

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

Client-side Insta485 Specification

This project includes the same pages as project 2. The only modified page is the index `/`. All other user-facing pages are identical to project 2. This section applies only to the main page.

The main page displays the same feed of posts as in project 2. In this project, posts will be rendered client-side by JavaScript code. All posts, including comments, likes, photo, data about the user who posted, etc. must be generated from JavaScript. Continue to use server-side rendering (AKA HTML templates) for the navigation bar at the top of the page.

> **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:** Commit and push features to your git repo, one at a time. Follow the Version control for a team work flow.

HTML Modifications from Project 2

In order for the autograder to correctly navigate the insta485 site, you'll need to add a few HTML tags to your HTML forms. *This section only applies to the main page; all other pages and their HTML elements may be left as they were in project 2.*

The comment form must contain the class name attribute `comment-form`. You may use this HTML form code. Feel free to style it and add other HTML attributes.

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

The like button must contain the class name attribute `like-unlike-button`. You may use this HTML code. Feel free to style it and add other HTML attributes.

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

The delete comment button must contain the class name attribute `delete-comment-button`. You may use this HTML code. Feel free to style it and add other HTML attributes. 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.

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

Main page HTML

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.

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.

```
...
<body>
  <!-- Plain old HTML and jinja2 nav bar goes here -->

  <!--
    We will tell React to use this div as it's entry-point for rendering
    **NOTE**: Make sure to include the "Loading ..." in the div below.
    This will display before our React code runs and modifies the DOM!
  -->
  <div id="reactEntry">
    Loading ...
  </div>
  <!-- Load JavaScript -->
  <script type="text/javascript" src="{{ url_for('static', filename='js/bundle.js') }}"></script>
</body>
...
```

Note: Rendered html text should be in the appropriate tags.

```
// good
render(){
  <p>{this.state.some_bool ? 'Hello' : 'Goodbye'}</p>
}

// bad
render(){
  {this.state.some_bool ? 'Hello' : 'Goodbye'}
}
```

Note: Structure your HTML in a reasonable way, putting unique content in separate tags. This is better code style, and it will prevent issues with the autograder parsing your HTML. Example:

```
// good (comment owner and comment are in separate tags)
render(){
  <div>
    <p>{comment_owner}</p>
    <p>{comment}</p>
  </div>
}

// bad (comment owner and comment are in same tag)
render(){
  <p>{comment_owner}{comment}</p>
}
```

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.

Response time

The main page should load without errors (exceptions), even when the REST API takes a long time to respond. Keep in mind that the `render()` method may be called asynchronously, and may be called multiple times.

Put another way, `render()` will likely be called by the React framework *before any AJAX data arrives*. The page should still render without errors.

Asynchronous design

Use the asynchronous Fetch API to make REST API calls from JavaScript (Mozilla Fetch API documentation).

Do not use `jQuery`. Do not use `XMLHttpRequest`.

⚠ **Pitfall:** State Updates May Be Asynchronous. Use this pattern when a `state` update depends on the previous `state`. React docs

```
// Wrong
this.setState({
  posts: this.state.posts.concat(data.results),
});
```

```
// Correct
this.setState(prevState => ({
  posts: prevState.posts.concat(data.results),
}));
```

Likes and comments update

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

0:00 / 0:11

Likes demo video.

Create a comment by pressing the `enter` (`return`) key. The user interface shall not contain any comment submit button. The React docs on forms are very helpful for this feature.

0:00 / 0:06

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.

0:00 / 0:04

Delete comment demo video.

Pro-tip: Use a React *Controlled Component* for the comment input box. React Controlled Component docs.

Pro-tip: The React docs on Handling Events have a helpful example of a toggle button that's useful for the like/unlike button.

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.

0:00 / 0:04

Double click demo video, the heart animation is optional.

Pro-tip: If you decide to create a Likes or Comments components, make them React Function Components and use the Lifting State Up technique.

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

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`.

```
constructor(props) {
 super(props);
 // Don't do this!
 this.state = { numLikes: props.numLikes };
}
```

Pitfall: Avoid using React Refs

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.

0:00 / 0:06

Infinite scroll demo video.

We recommend the React Infinite Scroll Component, which is already included in `package.json`.

If infinite scroll has been triggered and more than 10 posts are present on the main page, reloading the page should only display the 10 most recent posts (including any new posts made before the reload).

> **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.)

Browser history

Don't break the back button. Here's an example:

1. awdeorio loads `/`, which displays 10 posts.
2. awdeorio scrolls, triggering the infinite scroll mechanism. Now the page contains 20 posts.
3. jflinn adds a new post, which updates the database.
4. awdeorio clicks on a post to view the post details at `/posts/<postid>/`.
5. awdeorio clicks the back button on his browser, returning to `/`.
6. The exact same 20 posts from step 2 are loaded. jflinn's new post *is not included*.
7. awdeorio refreshes the page. the 10 most recent posts are shown including jflinn's new post.

As you might notice in the previous example, we do not specify whether the comments and / or likes on the 20 posts (step 6) are updated after returning to the index page using the back button. This is intentional. It is the student's choice whether to update the comments and likes of each post after returning to the main page using the back button.

Here's an example of two accceptable scenarios depending on the student's choice:

Scenario 1:

1. awdeorio loads `/`, which displays 10 posts (post ids 11-20).
2. awdeorio scrolls, triggering the infinite scroll mechanism. Now the page contains 20 posts (post ids 1-20).
3. jflinn adds a new post, which updates the database.
4. jflinn likes post id 20 (displayed to awdeorio in step 2), which updates the database.
5. awdeorio clicks on a post to view the post details at `/posts/5/`.
6. awdeorio clicks the back button on his browser, returning to `/`.
7. The exact same 20 posts from step 2 are loaded. jflinn's new post is not included but **his like on post id 20 is shown**.
8. awdeorio refreshes the page. the 10 most recent posts are shown including jflinn's new post.

Scenario 2:

1. awdeorio loads `/`, which displays 10 posts (post ids 11-20).
2. awdeorio scrolls, triggering the infinite scroll mechanism. Now the page contains 20 posts (post ids 1-20).
3. jflinn adds a new post, which updates the database.
4. jflinn likes post id 20 (displayed to awdeorio in step 2), which updates the database.
5. awdeorio clicks on a post to view the post details at `/posts/5/`.
6. awdeorio clicks the back button on his browser, returning to `/`.

7. The exact same 20 posts from step 2 are loaded. jflinn's new post is not included and **his like on post id 20 is not shown**.
8. awdeorio refreshes the page. the 10 most recent posts are shown including jflinn's new post.

The same example could be given to illustrate the student's freedom by taking scenario 1 and 2 and replacing jflinn's "like" in step 4 with a comment instead. In this case, the student could choose whether or not to display the comment in step 7.

The Mozilla documentation on the history API will be helpful.

Hint: this requires very few code modifications! Use the History API to manipulate browser history and use the PerformanceNavigationTiming API to check how the user is navigating to and from a page. Don't use other libraries for this feature (they only make it harder). Do not modify the URL.

> ⚠ **Pitfall:** Test the history functionality in a Chrome family browser (Chrome, Chromium, Brave, etc.). At the time of this writing (winter 2021), Firefox restores the DOM state when using the back button, so you won't see this problem with the back button.

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:

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

Press the like button a couple of times:

```
127.0.0.1 - - [06/May/2021 16:51:50] "DELETE /api/v1/likes/6/ HTTP/1.1" 204 -
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.

```
127.0.0.1 - - [16/Feb/2022 23:20:04] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [16/Feb/2022 23:20:04] "GET /static/js/bundle.js HTTP/1.1" 200 -
127.0.0.1 - - [16/Feb/2022 23:20:04] "GET /api/v1/posts/ HTTP/1.1" 200 -
127.0.0.1 - - [16/Feb/2022 23:20:04] "GET /api/v1/posts/104/ HTTP/1.1" 200 -
127.0.0.1 - - [16/Feb/2022 23:20:04] "GET /api/v1/posts/100/ HTTP/1.1" 200 -
127.0.0.1 - - [16/Feb/2022 23:20:04] "GET /api/v1/posts/101/ HTTP/1.1" 200 -
127.0.0.1 - - [16/Feb/2022 23:20:04] "GET /api/v1/posts/102/ HTTP/1.1" 200 -
127.0.0.1 - - [16/Feb/2022 23:20:04] "GET /api/v1/posts/103/ HTTP/1.1" 200 -
```

```
127.0.0.1 - - [16/Feb/2022 23:20:04] "GET /api/v1/posts/99/ HTTP/1.1" 200 -
127.0.0.1 - - [16/Feb/2022 23:20:04] "GET /api/v1/posts/98/ HTTP/1.1" 200 -
127.0.0.1 - - [16/Feb/2022 23:20:04] "GET /api/v1/posts/96/ HTTP/1.1" 200 -
127.0.0.1 - - [16/Feb/2022 23:20:04] "GET /api/v1/posts/95/ HTTP/1.1" 200 -
127.0.0.1 - - [16/Feb/2022 23:20:04] "GET /api/v1/posts/97/ HTTP/1.1" 200 -
127.0.0.1 - - [16/Feb/2022 23:20:05] "GET /uploads/e1a7c5c32973862ee15173b0259e3efdb6a391af.jpg HTTP/1
127.0.0.1 - - [16/Feb/2022 23:20:05] "GET /uploads/2ec7cf8ae158b3b1f40065abfb33e81143707842.jpg HTTP/1
127.0.0.1 - - [16/Feb/2022 23:20:05] "GET /uploads/ad7790405c539894d25ab8dcf0b79eed3341e109.jpg HTTP/1
127.0.0.1 - - [16/Feb/2022 23:20:05] "GET /uploads/122a7d27ca1d7420a1072f695d9290fad4501a41.jpg HTTP/1
127.0.0.1 - - [16/Feb/2022 23:20:05] "GET /uploads/9887e06812ef434d291e4936417d125cd594b38a.jpg HTTP/1
```

Scroll to the bottom and infinite scroll is triggered.

```
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 -
127.0.0.1 - - [16/Feb/2022 23:21:40] "GET /api/v1/posts/91/ HTTP/1.1" 200 -
127.0.0.1 - - [16/Feb/2022 23:21:40] "GET /api/v1/posts/92/ HTTP/1.1" 200 -
127.0.0.1 - - [16/Feb/2022 23:21:40] "GET /api/v1/posts/94/ HTTP/1.1" 200 -
127.0.0.1 - - [16/Feb/2022 23:21:40] "GET /api/v1/posts/93/ HTTP/1.1" 200 -
127.0.0.1 - - [16/Feb/2022 23:21:40] "GET /api/v1/posts/89/ HTTP/1.1" 200 -
127.0.0.1 - - [16/Feb/2022 23:21:40] "GET /api/v1/posts/90/ HTTP/1.1" 200 -
127.0.0.1 - - [16/Feb/2022 23:21:40] "GET /api/v1/posts/88/ HTTP/1.1" 200 -
127.0.0.1 - - [16/Feb/2022 23:21:40] "GET /api/v1/posts/86/ HTTP/1.1" 200 -
127.0.0.1 - - [16/Feb/2022 23:21:40] "GET /api/v1/posts/87/ HTTP/1.1" 200 -
127.0.0.1 - - [16/Feb/2022 23:21:40] "GET /api/v1/posts/85/ HTTP/1.1" 200 -
```

Code style

As in project 2, all HTML should be W3C compliant, as reported by `html5validator`. Python code should be contain no errors or warnings from `pycodestyle`, `pydocstyle`, and `pylint`. Use `pylint --disable=cyclic-import --unsafe-load-any-extension=y --disable=assigning-non-slot` (Why `--unsafe-load-any-extension`? Because the `sqlite3` module is a C extension.).

All JavaScript source code should conform to the AirBnB javascript coding standard. Use `eslint` to test it. Refer to the setup / eslint tutorial.

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.

Can I disable any code style checks?

Do not disable any code style check from any python code style tool (`pycodestyle`, `pydocstyle`, `pylint`), besides the three exceptions specified in the Project 2 spec

Additionally, do not disable any `eslint` checks in your `.jsx` files. There are **no exceptions** to this rule.

Testing

Make sure that you've completed the end-to-end testing tutorial.

Several unit tests are published with the starter files. Make sure you've copied the `tests` directory. Note that your files may be slightly different.

```
$ pwd
/Users/awdeorio/src/eecs485/p3-insta485-clientside
```

```
$ ls tests/
conftest.py    test_rest_api.py  test_slow_server_index.py  utils.py
pytest.ini     test_scripts.py   test_style.py
test_index.py  test_scroll.py    testdata
```

Rebuild your javascript bundles by running webpack and then run the tests.

```
$ pwd
/Users/awdeorio/src/eecs485/p3-insta485-clientside
$ npx webpack
Hash: 94d02a55e475959ef08d
Version: webpack 2.6.1
Time: 4910ms
...
$ pytest -v --log-cli-level=INFO
```

Note: if you get deprecation warnings from third party libraries, check out the pytest tutorial - deprecation warnings to suppress them. These deprecation warnings are expected.

`insta485test`

Add JavaScript style checking you `insta485test` script from project 2. In addition to the tests run in project 2, `insta485test` should run `eslint` on all files within the `insta485/js/` directory. Refer back to the `eslint` instructions for direction on how to do this.

```
$ ./bin/insta485test
```

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.

```
$ pwd
/Users/awdeorio/src/eecs485/p3-insta485-serverside
$ hostname
awdeorio-laptop  # NOT AWS
$ curl \
  --request POST \
  --cookie-jar cookies.txt \
  --form 'username=awdeorio' \
  --form 'password=password' \
  --form 'operation=login' \
  "http://<Public DNS (IPv4)>/accounts/"
$ curl -v -b cookies.txt "http://<Public DNS (IPv4)>/" > deployed_index.html 2> deployed_index.log
$ curl -v -b cookies.txt "http://<Public DNS (IPv4)>/static/js/bundle.js" > deployed_bundle.js 2> depl
```

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.

```
$ tar \
  --disable-copyfile \
  --exclude '*__pycache__*' \
  -czvf submit.tar.gz \
  bin \
  insta485 \
  package-lock.json \
  package.json \
  setup.py sql \
  webpack.config.js \
  deployed_index.html \
  deployed_index.log \
  deployed_bundle.js \
  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. Your Project 2 score will not impact your Project 3 score.

Rubric

This is an **approximate** rubric.

Tests	Value
Public Unit tests	50%
Public Python, JS, and HTML style	15%
Hidden unit tests run after the deadline	30%
AWS deployment	5%

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 when React methods are called. This is called the React component lifecycle. Add `console.log()` messages to each React method (`constructor()`, `render()`, etc.).

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`:

```
$ eslint --ext jsx insta485/js/
  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 propType: 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.

Reconciliation and keys

When using a collection of React components, they need to have unique `key` properties. This enables the fast shadow DOM to real DOM update performed by react. More info here:

- https://facebook.github.io/react/docs/reconciliation.html#keys
- https://facebook.github.io/react/docs/lists-and-keys.html#keys

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.

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

Acknowledgements

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