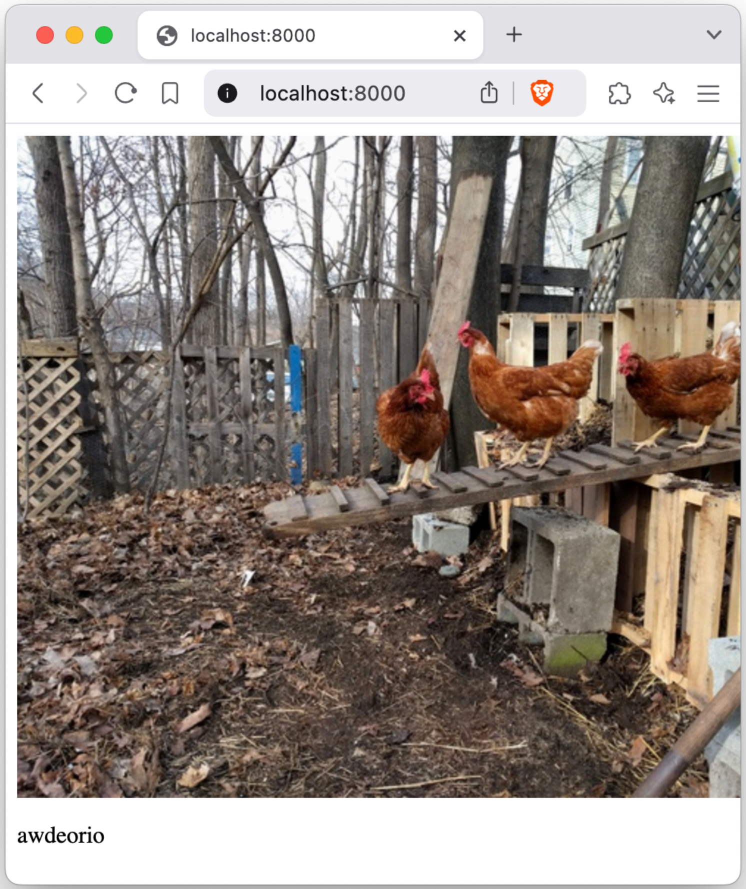
p3-insta485-clientside

React/JS Tutorial

This tutorial will walk you through a simple React/JS application that fetches from a REST API. The app will display a single social media post.



⚠ **Pitfall:** This tutorial is meant to be a supplement to the official React docs. Be sure to read them!

Prerequisites

You should have these configuration files from the starter files.

```
$ ls
package-lock.json  package.json  webpack.config.js  ...
```

You should have a minimally functional REST API from the Flask REST API Tutorial.

```
$ source env/bin/activate
$ flask --app insta485 --debug run --host 0.0.0.0 --port 8000
$ curl http://localhost:8000/api/v1/posts/1/
{
  "imgUrl": "/uploads/122a7d27ca1d7420a1072f695d9290fad4501a41.jpg",
  "owner": "awdeorio",
}
```

Install tool chain

We'll install these tools:

- Command line JavaScript interpreter `node`
- Package manager `npm`
- Third-party JavaScript libraries and frameworks like `React`
- Module bundler `webpack`
- Compiler `babel`
- Linter `eslint`
- Formatter `prettier`
- End-to-end testing framework `Cypress`

Node.js

Install the Node.js JavaScript interpreter and NPM package manager. The latest LTS version or higher is required for EECS 485.

macOS

Your versions may be different.

```
$ brew install node
$ node --version
v19.8.1
$ npm --version
9.5.1
```

Linux/WSL

Uninstall older versions of Node, then install the latest version from a third-party package repository maintained by NodeSource.

```
$ sudo apt remove nodejs
$ sudo apt autoremove
$ sudo apt update
$ sudo apt install -y ca-certificates curl gnupg
$ sudo mkdir -p /etc/apt/keyrings
$ curl -fsSL https://deb.nodesource.com/gpgkey/nodesource-repo.gpg.key | sudo
gpg --dearmor -o /etc/apt/keyrings/nodesource.gpg
$ NODE_MAJOR=23
$ echo "deb [signed-by=/etc/apt/keyrings/nodesource.gpg]
https://deb.nodesource.com/node_$NODE_MAJOR.x nodistro main" | sudo tee
/etc/apt/sources.list.d/nodesource.list
$ sudo apt update
$ sudo apt install nodejs -y
$ node --version
v20.7.0
$ npm --version
10.1.0
```

JavaScript packages

Install third-party packages like React. Package manager `npm` reads `package-lock.json` and `package.json` and installs into `./node_modules/`. You can ignore warnings about funding and vulnerabilities.

```
$ npm ci .
...
added 758 packages, and audited 759 packages in 4s
```

⚠ **WSL Pitfall:** `npm` may be slow or produce errors on network file shares. WSL uses a network file share between the Linux and Windows file systems. Use a folder that’s not a network file share.

Bad Example	Good Example
`/mnt/c/Users/awdeorio/`	`/home/awdeorio/`

Social Media Post App

This example is an app that displays a single social media post using React. The app fetches data from a REST API and displays it to the user.

Before continuing, read the React quick start.

Files

Start by creating an empty JavaScript package for our web app.

```
$ mkdir -p insta485/js/
$ mkdir -p insta485/templates/
$ touch insta485/js/main.jsx
$ touch insta485/js/post.jsx
$ touch insta485/templates/index.html
```

Your files should look like this. It's OK if you have other files copied from Project 2.

```
$ tree insta485/
insta485/
├── js
│   ├── main.jsx
│   └── post.jsx
└── templates
    └── index.html
```

`index.html`

Edit or create an HTML file, e.g., `insta485/templates/index.html`. If you copied your HTML from project 2, delete all the jinja template code that displays the feed, but keep the navigation bar.

Add an empty `div` with an id of `reactEntry` to your top level HTML file. Later, we'll write JavaScript code to add DOM nodes at this entry point.

Then load the `bundle.js`, which will contain JavaScript code for the app we're about to write.

insta485/templates/index.html

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


  <div id="reactEntry">
    Loading ...
  </div>
  <!-- Load JavaScript -->
  <script src="{{ url_for('static', filename='js/bundle.js') }}"></script>
</body>
</html>
```

Notice that the HTML code asks for `bundle.js`, which is the output of our module bundler and compiler. The inputs to the bundler and compiler are the JavaScript files in `insta485/js/`. The output is a single JavaScript file that is completely self-contained with no dependencies, `insta485/static/js/bundle.js`.

`main.jsx`

The `main.jsx` file includes `import` statements for React libraries and our custom `Post` component. It also connects the custom `Post` component to the `reactEntry` `div` from above in our `index.html`.

insta485/js/main.jsx

```
import React, { StrictMode } from "react";
import { createRoot } from "react-dom/client";
import Post from "./post";

// Create a root
const root = createRoot(document.getElementById("reactEntry"));

// Insert the post component into the DOM.  Only call root.render() once.
root.render(
  <StrictMode>
    <Post url="/api/v1/posts/1/" />
  </StrictMode>
);
```

In this example, we render one component, `Post`. In your project, you will render many, but still use only one `createRoot()` and `root.render()` call with a parent component that manages child components. See this section of the React Docs and Thinking in React docs.

Wrapping your React application in `<StrictMode>` helps catch bugs by triggering extra re-renders and Effects checks during development. See here for documentation.

`post.jsx`

The `post.jsx` file contains a React component called `Post` that represents one social media post.

insta485/js/post.jsx

```jsx
import React, { useState, useEffect } from "react";

// The parameter of this function is an object with a string called url inside it.
// url is a prop for the Post component.
export default function Post({ url }) {
  /* Display image and post owner of a single post */

  const [imgUrl, setImgUrl] = useState("");
  const [owner, setOwner] = useState("");

  useEffect(() => {
    // Declare a boolean flag that we can use to cancel the API request.
    let ignoreStaleRequest = false;

    // Call REST API to get the post's information
    fetch(url, { credentials: "same-origin" })
      .then((response) => {
        if (!response.ok) throw Error(response.statusText);
        return response.json();
      })
      .then((data) => {
        // If ignoreStaleRequest was set to true, we want to ignore the results of the
        // the request. Otherwise, update the state to trigger a new render.
        if (!ignoreStaleRequest) {
          setImgUrl(data.imgUrl);
          setOwner(data.owner);
        }
      })
      .catch((error) => console.log(error));

    return () => {
      // This is a cleanup function that runs whenever the Post component
      // unmounts or re-renders. If a Post is about to unmount or re-render, we
      // should avoid updating state.
      ignoreStaleRequest = true;
    };
  }, [url]);

  // Render post image and post owner
  return (
    <div className="post">
      <img src={imgUrl} alt="post_image" />
      <p>{owner}</p>
    </div>
  );
}
```

Props

The `Post` component is a pure function, always returning the same output for the same input.

Inputs are passed as *props*, which are function parameters. Props are read-only and immutable, so components cannot change their props. The `Post` component takes a single prop, `url`.

The output is a tree of DOM nodes described by JSX syntax. JSX is a JavaScript extension that compiles into JavaScript code for creating DOM nodes.

```
export default function Post({ url }) {
  // ...

  return (
    <div className="post">
      <img src={imgUrl} alt="post_image" />
      <p>{owner}</p>
    </div>
  );
}
```

Please read the Describing the UI chapter of the React Docs to learn more about using JSX to render React components.

State

For mutable values (values that change), we use *state*. When `state` changes, the component re-renders, the DOM changes, and the user can see the updated page.

The `Post` component changes two values: an image URL (`imgUrl`), and the creator of the post (`owner`). Initially, both values are set to an empty string.

Both *state* and *props* can appear in the output.

```
export default function Post({ url }) {
  const [imgUrl, setImgUrl] = useState("");
  const [owner, setOwner] = useState("");

  // ...

  return (
    <div className="post">
      <img src={imgUrl} alt="post_image" />
      <p>{owner}</p>
    </div>
  );
```

In the next section, we’ll use `setImgUrl()` and `setOwner()` to modify state with values from a REST API.

Please read the Adding Interactivity chapter of the React Docs to learn more about state.

Fetch from a REST API

For this example, the `Post` component will fetch from a REST API that returns post details like this. The REST API for your project will include more detail!

/api/v1/posts/1/

```json
{
  "imgUrl": "/uploads/122a7d27ca1d7420a1072f695d9290fad4501a41.jpg",
  "owner": "awdeorio",
}
```

After calling the above REST API with `fetch()`, `setImgUrl()` and `setOwner()` update their respective states, triggering a re-render.

```
export default function Post({ url }) {
  const [imgUrl, setImgUrl] = useState("");
  const [owner, setOwner] = useState("");

  useEffect(() => {
    // Call REST API to get the post's information
    fetch(url, { credentials: "same-origin" })
      .then((response) => {
        if (!response.ok) throw Error(response.statusText);
        return response.json();
      })
      .then((data) => {
        setImgUrl(data.imgUrl);
        setOwner(data.owner);
      }
      .catch((error) => console.log(error));

      // ...
  }, [url]);
}
```

The `fetch()` function is called inside an anonymous function passed to `useEffect()`. This anonymous function is called by React *after* the component renders. See our explanation of Data Fetching in React with useEffect for an in-depth explanation.

The line `}, [url]);` contains `useEffect`'s dependency array, which controls when the effect runs. If you don't pass in an array, it will run on every render. If you pass in an empty array, it will run only after the first render. And, if you include variables in the array, it will run after the first render and *whenever those variables change*.

Please read the Synchronizing With Effects section of the React Docs to learn more about side-effects.

Build and run

Run the module bundler `webpack`, which puts together our code with third-party library code. It also uses `babel` to compile modern JavaScript to a version supported by older web browsers.

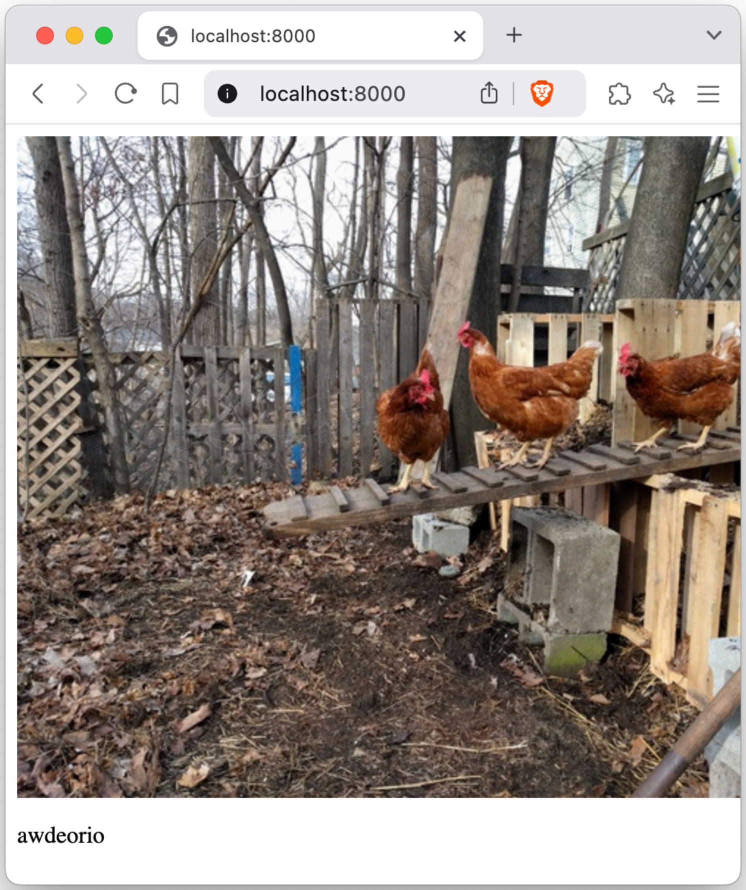
The inputs are the JSX files in `insta485/js/` and the JavaScript packages in `node_modules/`. The output is a single file `insta485/static/js/bundle.js`. The configuration is in `webpack.config.js`.

```
$ npx webpack
...
webpack 5.6.0 compiled successfully in 2290 ms
```

Run your Flask web server.

```
$ flask --app insta485 --debug run --host 0.0.0.0 --port 8000
```

Browse to http://localhost:8000/, and you will be able to see the Post component.



Developer Tools

In this section, we'll discuss using `eslint` and `prettier` to enforce coding style and a web browser extension that helps debug React applications.

`eslint`

We use `eslint` to enforce the AirBnB JavaScript coding standard. The configuration is in the `.eslintrc.js` provided with the starter files.

```
$ npx eslint insta485/js/post.jsx  # Check one file
$ npx eslint --ext jsx insta485/js/  # Check all files
```

`prettier`

We use `prettier` to enforce default formatting rules. You can also have `prettier` fix formatting automatically.

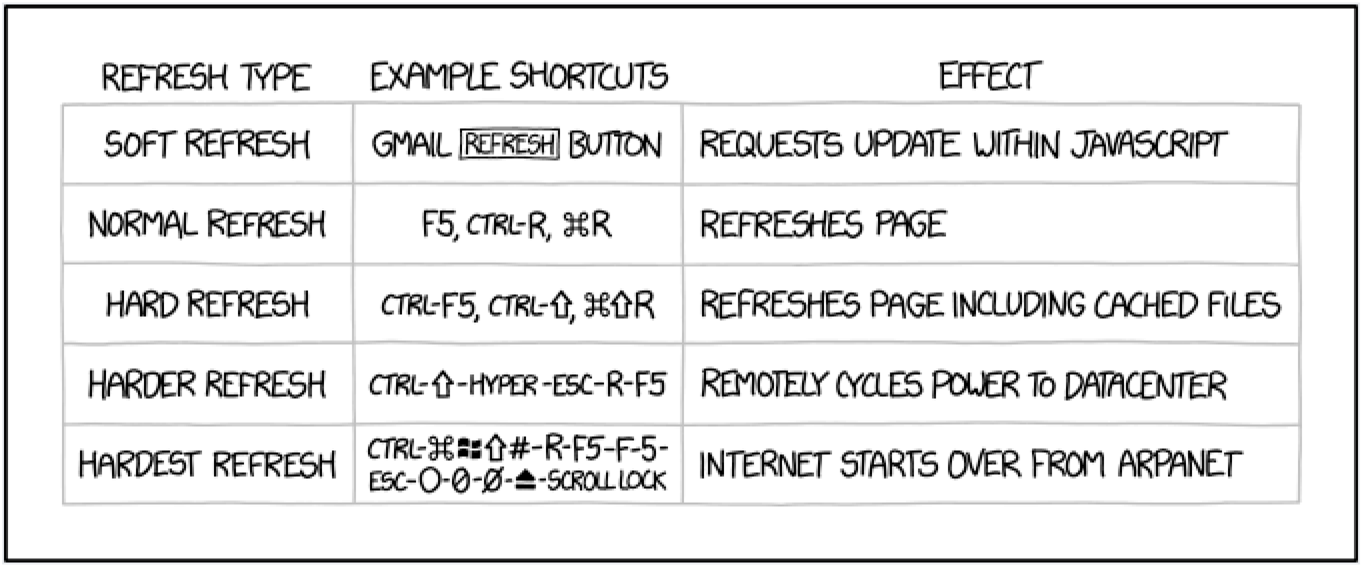
```
$ npx prettier --check insta485/js  # Check
$ npx prettier --write insta485/js  # Fix
```

React Developer Tools

React Developer Tools is a web browser extension that adds a “Components” tab to the developer tools. It lets you browse your React components organized in a way that looks a lot like your JSX code, rather than the complex DOM that results from your source code. See the React/JS Debugging Tutorial for an example of how to use this tool.

Browser refresh and JavaScript

JavaScript source code is sometimes cached by the web browser. If you change the source code, you need to tell your browser to clear the cache and reload the JavaScript using the hard refresh. The commands for a hard refresh are different based on your OS and browser so take a look on how to hard refresh with your system. If you are using Chrome, you can also disable caching by going to the network tab of the web inspector developer tool and clicking on the checkbox that says “Disable cache”. This comic “explains” (credit: xkcd.com).



REFRESH TYPE	EXAMPLE SHORTCUTS	EFFECT
SOFT REFRESH	GMAIL [REFRESH] BUTTON	REQUESTS UPDATE WITHIN JAVASCRIPT
NORMAL REFRESH	F5, CTRL-R, ⌘R	REFRESHES PAGE
HARD REFRESH	CTRL-F5, CTRL-⇧, ⌘⇧R	REFRESHES PAGE INCLUDING CACHED FILES
HARDER REFRESH	CTRL-⇧-HYPER-ESC-R-F5	REMOTELY CYCLES POWER TO DATACENTER
HARDEST REFRESH	CTRL-⌘■⇧#-R-F5-F-5-ESC-O-0-Ø-⏏-SCROLL LOCK	INTERNET STARTS OVER FROM ARPANET

Acknowledgments

Original document written by Andrew DeOrio awdeorio@umich.edu.