Server-Side Rendering

Goal

The goal of this lab is to serve a prerendered React application to the browser.

The Mission

In this lab you will be prerendering a React application. The goal is to display the UI faster to the user and help a search engine index the page better.

The starter application in this exercise contains a working movie viewer and editor. The application is a traditional React application which is completely rendered in the browser. In this lab you will use either server-side rendering or prerendering to serve the application in a, partially, rendered state from the server.

You can start the application by running the **start.bat** file in the root folder. When you do a small Node.js server will start. Next the default browser will start to display the **index.html** page. Please note that the first time you start this will take some time as number of NPM packages will download. This requires an internet connection where the NPM registry is not blocked by a firewall.

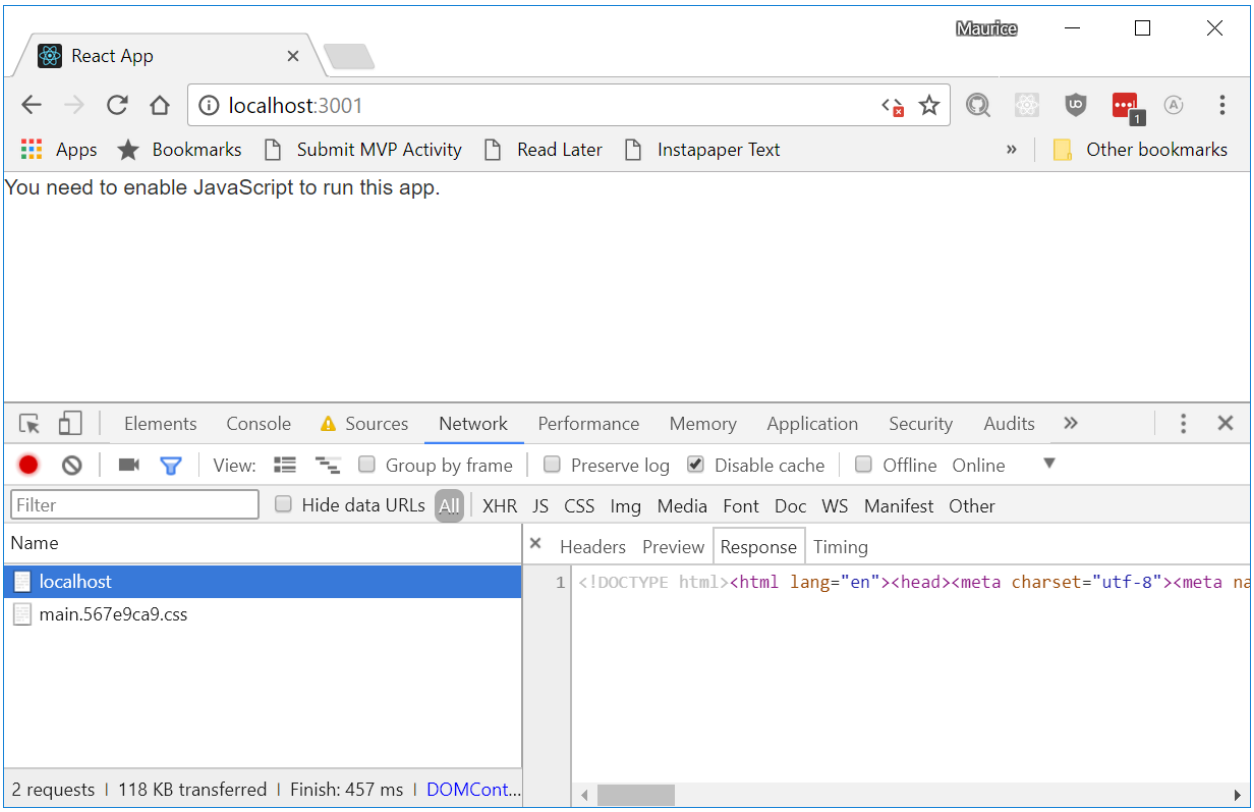
Note: The **FINISHED-PRERENDER** folder contains a finished version of this Prerendered lab as a reference. The **FINISHED-SSR** folder contains a finished version of this Server Side Rendered lab as a reference.

Type it out by hand?
Typing it drills it into your brain much better than simply copying and pasting it. Because you're forming new neuron pathways that are going to help you in the future, help them out now!

Please choose between assignment one and two for this exercise.

Before you start: Observe the default React behavior

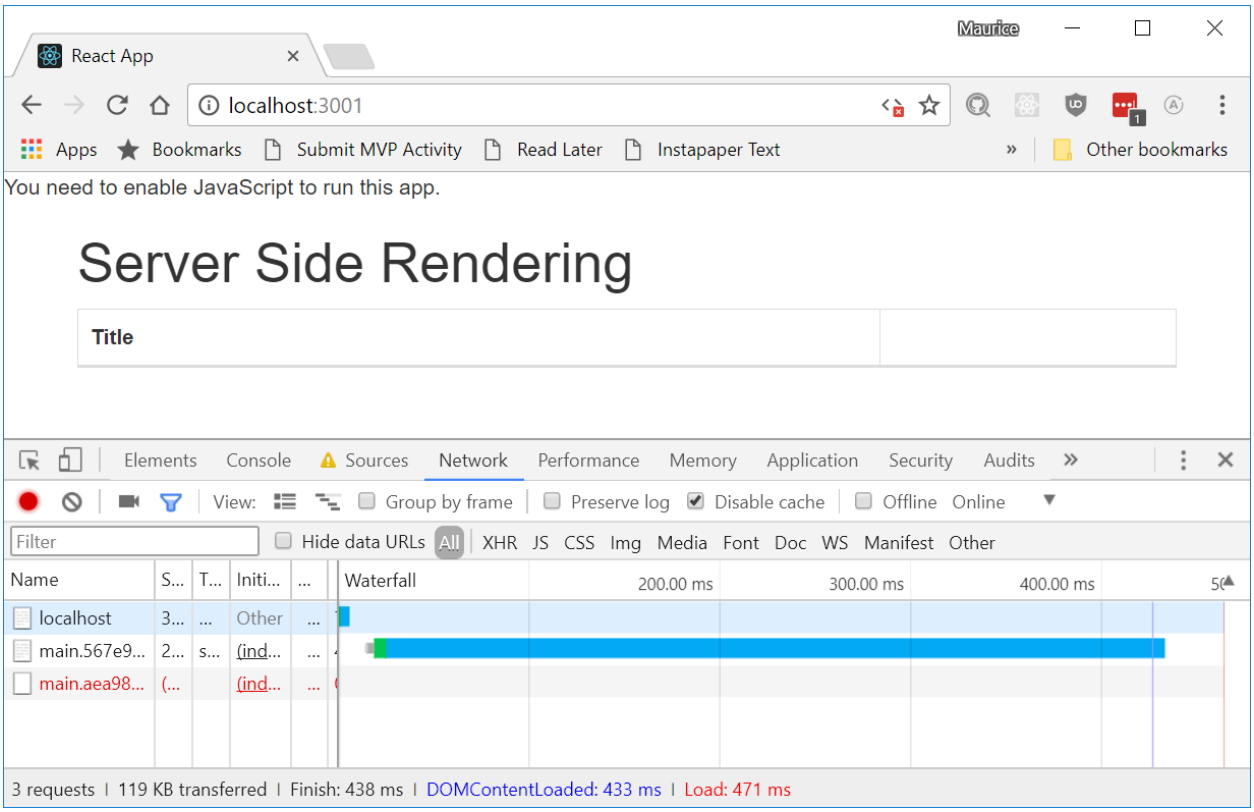
Start the application in the **BEGIN** folder and familiarize yourself with the functionality. Build the application using *npm run build* and start the application using *npm start*. The pre-build application is available at http://localhost:3001. Open the browsers debugger and inspect the initial html page being served to the browser. Observe that this page contains very little markup. You can also disable JavaScript in the browser and observe the resulting page.



Now choose on how to fix this using either build time prerendering or server-side rendering.

Assignment 1 - Prerendering

Using build time prerendering is the easy option. It will render the static parts of the application into the **INDEX.HTML** and serve that to the browser. Any dynamic content, like the list of movies, will only be rendered at runtime in the browser.



Start by installing react-snap as a development dependency using:

npm install --dev react-snap

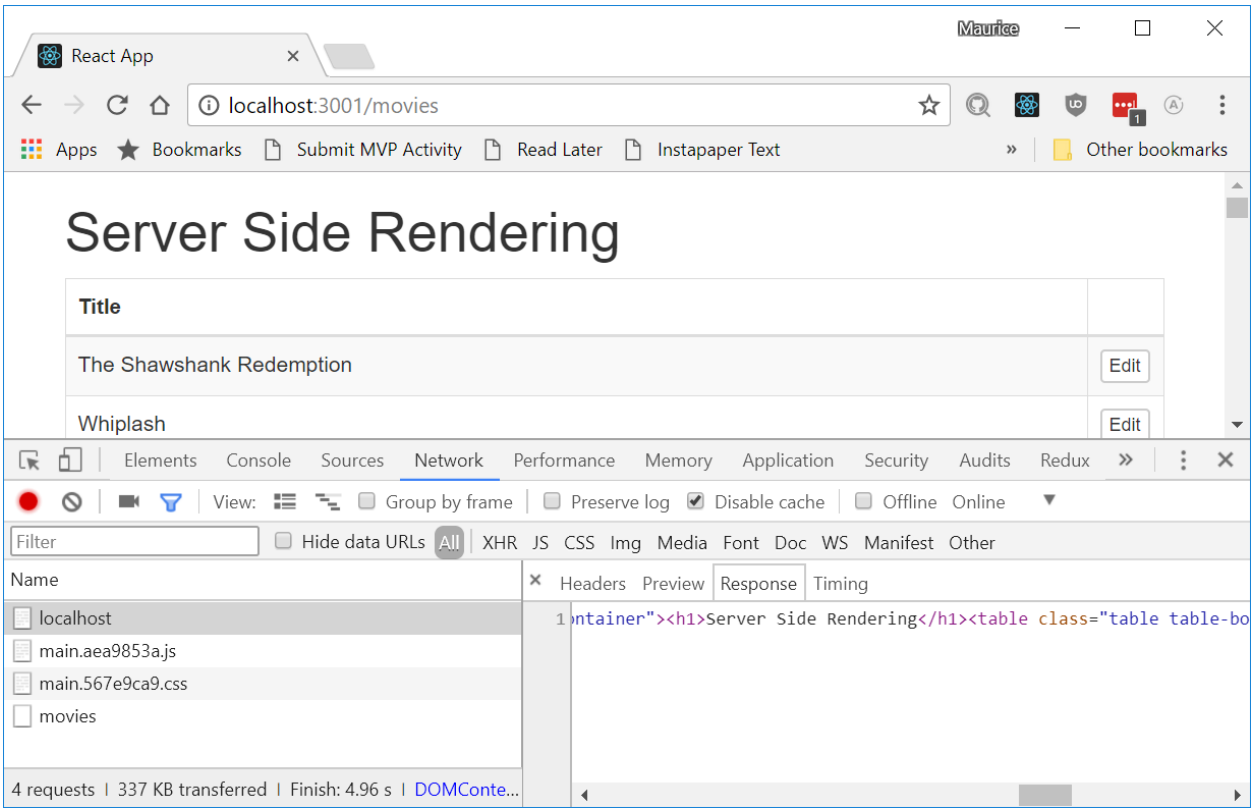
Next update the *build* script in the **./PACKAGE.JSON** to execute *react-snap* after each build.

```
"scripts": {
  "build": "react-scripts build && react-snap",
},
```

Open ./SRC/INDEX.JS and replace the rendering of the application with a check if there are DOM child elements. Use *ReactDOM.hydrate()* if there are children and *ReactDOM.render()* if there are none.

```
const rootElement = document.getElementById("root");
if (rootElement.childElementCount) {
  hydrate(application, rootElement);
} else {
  render(application, rootElement);
}
```

Now build use application using *npm run build* and run the application using *npm start*. If you navigate to http://localhost:3001 you will see the prerendered application.



Optional assignment 2 – Server-side rendering using Next.js

Create a basic Next.js based React application using the following steps:

1. Create a new folder named **ssr** under the **lab** folder and cd into the **ssr** folder
2. Create a package.json using: **npm init**
3. Install Next and React using: **npm install next react react-dom**
4. Add the **dev** script to the **package.json** to execute the **next** command
5. Add a folder named **pages**
6. Add an **index.js** in the **pages** folder with an export default of a simple React component
7. Run the application using **npm run dev**
8. Open the browser at http://localhost:3000/

Use NPM to add the concurrently and json-server packages.

Add the **db** folder to the project and add the following script to the **package.json**:

```
"start:db": "json-server ./db/movies.json --routes ./db/routes.json --port 3001",
```

Update the **dev** script to run both the JSON server and the Next.js server

```
"dev": "concurrently \"npm run start:db\" \"next\"",
```

With the basic application setup in place you can add a **movies.js** in the **/pages** folder. Add the **MovieList** component to this file. Add a folder named **/components** and add the **MovieRow** component to it. Remove the usages of Redux and use the Next.js specific **getInitialProps()** lifecycle method to load the list of movies. Note that Next.js doesn't have an HTTP proxy by default to use the complete URL to **http://localhost:3001/api/movies**. Update the **MovieRow** to use the Next.js **Link** component and change the URL to use a query string parameter like **`/movie?id=${movie.id}`**

Use NPM to install **bootstrap**. Add an **_app.js** file to the **/pages** folder and use it to load bootstrap and the main page layout using a **div** with the class **container** and a page header.

Add a **movie.js** to the **/pages** folder and add the **MovieEdit** component to it. Add the required **InputText** and **TextArea** to the **/components** folder. Remove the usages of Redux and use the Next.js specific **getInitialProps()** lifecycle method to load the movie with the ID specified in the context parameters **query.id**. Use the **getDerivedStateFromProps()** lifecycle function to store the movie in the component state and update the **onChange** to update this state. Update the **save** function to use Next.js **Router** and the **Cancel** button to use the Next.js **Link** components.

Make sure the application is still working as expected.