Camosun College

ICS 221 - Web Applications

Lab 2 - A Universal/Isomorphic JavaScript Message Board: Part 1

DEMO Due Date: Lab Demo Due by Thurs., Feb. 7 @ 4:50 PM

QUIZ Due Date: Lab Quiz Due by Fri., Feb. 8 @ 11:59 PM

Preparation

Over the next couple of labs, you will build an Isomorphic JavaScript (very basic) Message Board. A Message board is a great choice to introduce these concepts - its fairly simple but does require our application to be able to post messages to a page and list messages that have been posted. For the first time, you will see the benefits of Server-Side rendering (SSR) the first page of your application. Then, once loaded in the client browser, you will go full SPA and only make data requests back to the Server.

Here is how the Universal JavaScript Application works:

1. The Client browser navigates to our application's domain and makes the initial page request.

2. The Server receives the request, generates the HTML for the page from React Components, data, and the Pug View Template Engine. It then sends the HTML to the Client browser.

3. The Client browser makes additional requests for static assets, as required. So far, this is exactly what you did in Lab 1.

4. Now, the SPA behavior of our application takes over. Any subsequent Client browser requests will be for storing or reading data. Our application will send and receive JSON data and update only those Components necessary. This is what you did last semester. The key thing with Universal JavaScript though is the same Component code base will be used for both SSR and CSR.

Eventually, we want to add a fully database-driven *real* RESTful API Server. This we can do in a future lab. For now, in this lab, we'll concentrate on just getting the first 3 items above working.

To prepare for this lab, review what you did in Lab 1. Much of the concepts and code from that lab will be used in this lab.

ES5 CommonJS vs ES6 Modules

One thing you may have noticed different about Lab 1 and this lab compared to when you used Create React App in ICS 211, is the way we are sharing modules in our Application. In ICS 211, for your project, you used `import` and `export`. In this Lab and Lab 1, you're using `require()` and `module.exports` (a system called CommonJS). Node.js doesn't yet support ES6 Modules. We could use Babel to transpile ES6 to ES5 but it would have to work for the entire application. Right now, we're only using Babel to do our JSX to JS transpiling, on-the-fly. Using the ES5 syntax ensures our code is truly universal: it will work on both the Node.js server and the client browser.

Express Generator

Express Generator is a npm package that performs the same purpose for Express that Create React App did for React. It creates skeleton code and provides zero-configuration for starting an Express-based application. For example, after you complete Task 1 step 1, your Express app will have the following folder structure:

```
ics221-lab2-mb/
├── bin/
    ├── www
├── public/
    ├── images/
    ├── javascripts/
    ├── stylesheets/
        ├── style.css
├── routes/
    ├── index.js
    ├── users.js
├── views/
    ├── error.pug
    ├── index.pug
    ├── layout.pug
├── .gitignore
├── app.js
└── package.json
```

The biggest difference between this and what you did in Lab 1 is the addition of a *bin/* folder and a file called *www*. Express generator moves the Node.js web server code into this file (the `http.createServer` code you did in Lab 1) as well as add some network error handling. Two routers have been auto-generated as examples: *index.js* and *users.js*. Some Pug view templates have also been auto-generated: *layout.pug*, *index.pug* and *error.pug*. In *app.js*, some middleware has been automatically added. There's middleware to process JSON data, URLEncoded data, cookies, and, like lab 1, static content. Some extra error handling code has also been automatically generated.

The Express Generator mixes the routing code and the application logic. We'll need to fix that to make it true MVC. You'll do that in Task 2.

Front-End Design: Bootstrap

This course concentrates on back-end web app implementation but we can't ignore front-end design. We still need to make our Apps somewhat presentable! In ICS 211, you used CSS with your React

Components and you were also introduced to Ant Design (although this was optional). We could ignore UI Libraries altogether and just do straight CSS. But that's time consuming for prototype-level labs and projects, especially when we want to concentrate on the back-end coding. We could use a React UI Library like Ant Design or reactstrap, but you can't use these outside of a React Component (like in Pug, for example). They also use ES6 modules which isn't compatible with Node.js. The easiest thing to do is to use the Bootstrap CSS Library, the most common used and one you should get familiar with.

By using Bootstrap, you instantly make it easier to lay out your website. Bootstrap also takes care of making your web app reactive - the term used to describe a website that works on any device. Bootstrap uses a 12-column grid system for layout and is described here. It is based on flexbox. Bootstrap divides the viewport up into twelve equal columns, no matter what device you are on, or how big the viewport is. Obviously, the width of those columns will differ depending on the width of the current viewport. Bootstrap comes with five pre-defined classes that you can use to customize your layout for different viewports. The following table outlines the different viewport sizes (breakpoints) available in Bootstrap:

Breakpoint Name	Class Prefix	Example Device	Pixel Width
Extra-small devices	col-	Phones (Portrait)	less than 576
Small devices	col-sm-	Phones (Landscape)	576 or more
Medium devices	col-md-	Tablets	768 or more
Large devices	col-lg-	Laptops	992 or more
Extra large devices	col-xl-	Desktop Monitor	1200 or more

Which of these you use depends on your Application and its layout. For example, suppose you have some content in a `div` tag. On Phones, you want the content to take up the entire width of the viewport (all twelve columns) but on Tablets and larger devices, you want it to use only half the viewport (six columns). Based on the previous table, you would define the div tag as follows:

```
<div class="col-12 col-md-6"> . . . </div>
```

If you did `class="col-12"` only, then it would use 12 columns, no matter what device.

Now suppose that you have two divs that are declared the same way:

```
<div class="col-12 col-md-6">CONTENT ONE</div>
<div class="col-12 col-md-6">CONTENT TWO</div>
```

And you view the page on a device that has a viewport with a width of 767 pixels or less:

```
---------------
| CONTENT ONE |
| CONTENT TWO |
---------------
```

Since the viewport is smaller than the *medium devices* breakpoint, the first class value applies. In this case, each `div` will take up the entire twelve columns and thus will appear on their own row. On the other hand, if you view the page on a device with a viewport of 768 pixels or greater:

```
---------------------------
| CONTENT ONE CONTENT TWO |
---------------------------
```

In this case, the second class value applies. This means that each `div` will use six columns and will now appear in the same row, side by side.

As can be seen by the above example, you use Bootstrap by applying pre-defined class values to your tags.

Bootstrap has excellent documentation. To use Bootstrap and any of its UI Components, refer to the documentation.

nodemon

Recall that the V8 engine, the JavaScript JIT compiler behind Node.js, actually compiles JavaScript to machine code just before execution. This machine code is held in memory. That means that if you make changes to any of your back-end JavaScript code (including React code), the Node.js Server won't re-compile it until it is re-started. This can be a bit of pain while developing. Fortunately, nodemon is a tool that detects changes in your code and will automatically re-start the Node.js Web Server for you. It won't, however, automatically refresh the browser. That you'll have to do manually or you can use a tool like Browsersync or LiveReload.

Tasks

Task 1 - Initializing the Project

1. This time, to create your project, you are going to use a tool similar to what *create-react-app* did for React. *express-generator* creates skeleton code for an Express-based application. In a bash shell where you want your project to be located, run the following command:

   ```
   npx express-generator --view=pug --git ics221-mb
   ```

   This will create a skeleton Express App inside a folder called *ics221-mb*. It sets the view template engine to pug and also create a *.gitignore* file for creating a git repository (but don't create it yet).

2. Change into the *ics221-mb/* folder. You'll need to add React and Babel packages to the project:

   ```
   npm install react react-dom --save
   npm install @babel/core @babel/preset-react @babel/register --save-dev
   ```

3. Now **run `npm install`** to install the packages listed in *package.json* (this will create a *node_modules/* folder and a *package-lock.json* file).

4. Finally, run the Application by issuing the following command in a bash shell:

   ```
   DEBUG=ics221-mb:* npm start
   ```

The `DEBUG=ics221-mb:*` is for extra debugging when developing an Express application. It is explained [here](). `npm start` looks in the *package.json* file's *scripts* object to determine how to start the application. If you look in your *package.json* file, you will see that it starts your application by executing *node ./bin/www*.

5. If you navigate your browser to http://localhost:3000 you should see the output of the *index* router which is a simple 'Welcome to Express' message.

Task 2 - Making Your Application MVC

Like in lab 1, you are going to make your Application MVC. To do so, your Project's directory structure should look like that below:

```
ics221-mb/
├── app_server/
    ├── components/
    ├── controllers/
    ├── routes/
    ├── views/
        ├── error.pug
        ├── index.pug
        ├── layout.pug
├── bin/
    ├── www
├── node_modules/
    ├── (module dependencies)
├── public/
    ├── images/
    ├── javascripts/
    ├── stylesheets/
        ├── style.css
├── .gitignore
├── app.js
├── package-lock.json
└── package.json
```

You can delete both the *index.js* and *users.js* Routers currently under the *routes/* folder. Remove any reference to them in *app.js*. Also, fix the *views/* path in *app.js* (you can refer to Lab 1 for how it should look).

Task 3 - Designing Your Project (no steps in this task, read only)

Before you start your implementation, you need to think about how your project should be designed. This is very simple App, but we should still think about how we are going to partition the UI into React Components. You did this same thing last semester:

```
-------------
|  Header   |
-------------
| NewMsg    | <---|
-------------     |----> MsgBoard
| MsgList   | <---|
-------------
```

```
| Footer    |
-------------
```

We'll split our UI into *five* Components: `Header`, `NewMsg`, `MsgList`, `MsgBoard`, and `Footer`. `MsgBoard` will be a *Composite Component* with two child Components: `NewMsg` and `MsgList`.

Next we need to think about URL paths our Application will need. There's only one: '/'. Easy Peasy.

Task 4 - Add the App Router

Using Lab 1 as a guide, add an *app_router.js* file to your project. You'll need to import it in *app.js* and add it as a route (only one - the home page). In *app_router.js*, import a `msgController` from a *msg.js* file and pass control to an `index` handler for the '/' route. Refer to your code for Lab 1 as a guide.

Task 5 - Add a *msg.js* Controller

Create a new folder under *app_server/* called *controllers/*. Inside this folder, create a file called *msg.js*. Inside this file, you will need to do the following (use the *main.js* controller file from Lab 1 as a guide):

1. `require()` React and ReactDOMServer.

2. You will need to **transpile** and `require` three JSX components: `Header`, `Footer`, and `MsgBoard`.

3. For now, we're going to hard-code the data directly in the Controller. That eliminates the need to worry about a database and RESTful API at this point. It does mean that our data will be read-only. Why do it this way? We're breaking down the problem. By hard-coding the data (the Model) in the Controller, we can fully test and debug the View-Controller part of our application without worrying about all the Model code. In fact, we could have even hard-coded the data right in the View further breaking down the problem. In any case, putting the data here makes troubleshooting easier. Below your React Components' `createFactory()` statements, add the following global array declaration:

```
// temp hard-coded data
const msgs = [
  { id: 1, name: 'Bill', msg: 'Hi All!' },
  { id: 2, name: 'Ann', msg: 'ICS 221 is fun!' },
  { id: 3, name: 'John', msg: 'Howdy!' },
  { id: 4, name: 'Barb', msg: 'Hi' },
  { id: 5, name: 'Frank', msg: 'Who\'s tired?' },
  { id: 6, name: 'Sarah', msg: 'I heart React' }
];
```

   We'll eventually move this data to a database and fetch it via a RESTful API.

4. Next, you will need to add an `index` handler:

```
// index handler
const index = (req, res) => {
  res.render('index', {
    title: 'ICS 221 Universal JS Msg Board',
    header: ReactDOMServer.renderToString(Header()),
    footer: ReactDOMServer.renderToString(Footer()),
    msgBoard: ReactDOMServer.renderToString(MsgBoard(
      { messages: msgs }
    )),
    props: '<script>let messages=' + JSON.stringify(msgs.reverse()) +
     '</script>'
  });
};
```

Like in Lab 1, `res.render()` is the function that is going to generate the HTML and send it to the Client Browser. We use `ReactDOMServer.renderToString()` to render our React Components on the Server. We pass in the data (the msgs array) to `MsgBoard` as props. The `props` key below `msgBoard` renders a `script` tag with the `msgs` array reversed as JSON. This is for passing `messages` to the client-side code on initial render. It's reversed because the client-side `MsgBoard` will render it that way and they have to match.

5. You need to export this function. Add the following statement at the bottom of the file:

   ```
   module.exports = { index };
   ```

   The function in the exports must have curly braces around it to work.

   Next, we need to make our React Components and the Pug index template.

Task 6 - Create the Header and Footer React Stateless Components (ICS 211 Review)

1. Let's start with the Header Component. Like in Lab 1, create a *components/* folder under *app_server/*. Inside this folder, create a file called *Header.jsx*. You should be able to make this Stateless Component yourself! Using the *About.jsx* Component from Lab 1 as a guide, this Component only needs to render a `h1` tag with the inner text 'ICS 221 Message Board App'.

2. Next, create a file called *Footer.jsx*. It should simply render a `p` tag with `&copy;2019 your name` as the inner text.

Task 7 - The MsgBoard Stateful Component (ICS 211 Review)

Next, make a *Stateful* MsgBoard Component. This Component should have a `messages` state variable that is set to `this.props.messages` in the Constructor. In its `return()` statement in its `render()` method, you should render your `MsgList` Component, passing in the state variable as props (name the props `messages`).

Don't forget you need to import both the `MsgList` Component and the React library (using ES5 `require()`). You also need to export the Component (using ES5 `module.exports`).

We'll ignore the `NewMsg` Component for now.

Task 8 - The MsgList Stateless Component (ICS 211 Review + Bootstrap)

You need to make a *Stateless* MsgList Component. The goal of this Component is to render a formatted (using Bootstrap) table of the messages.

1. `require()` in the React Library, declare the Component as an Arrow Function and start a `return()` statement (all the following steps will be inside this statement).

2. Add a `table` tag with `className="table table-striped table-bordered"`. Remember, in React Components, the `class` attribute becomes `className`. What these class attribute values do are described in Bootstrap's Table Documentation.

3. Next, is your `thead` tag. Inside this should be a `tr` with three `th` tags (refer to online HTML documentation if you don't remember HTML tables) for three columns. The first column's title will just be # (msg number). The second column's title is **Name**. The third column's is **Message**. Each `th` tag should have a `scope="col"` attribute. This is for Bootstrap tables. In addition, add the following attribute based on the column:

   - First Column: `className="w-25"`
   - Second Column: `className="w-25"`
   - Third Column: `className="w-50"`

   This is using Bootstrap's Sizing Utility. The numbers are percentages. The first two columns will each take up 25% of the width with the last column 50% of the total available width.

4. After closing off your `tr` and `thead` tags, start a `tbody` tag. This is the code you need between your opening and closing `tbody` tags:

```
{props.messages.sort((a, b) => a.id - b.id).
  reverse().map( (message, index) =>
  <tr key={message.id}>
    <td>{index+1}</td>
    <td>{message.name}</td>
    <td>{message.msg}</td>
  </tr>
)}
```

   `MsgList` gets the `messages` array passed in as props from `MsgBoard`. We sort (by `id`) and reverse the array because we want the newest messages on top. Like in ICS 211, we use the `map()` function to "JSX-ize" each element. Recall that `map()` applies the function given as an argument to each element in the array. The anonymous arrow function takes two arguments: the first is each element of the array (each message) and the second is the current index of the array. In the first `td` cell, we output `index+1` since the array's index starts at 0 but we want the msg# to start at 1. The next cell outputs the message's name and the last the actual message. Finally, recall that React needs a `key` attribute in the top-level DOM element.

5. Now add a closing `tbody` tag and close off the table with a closing `table` tag.

6. Close off the `return()` statement and the Component. `module.exports` out the `MsgList` Component.

Task 9 - The Pug View Templates

1. With the React Components out-of-the-way, the last thing we need to do is set up the View Templates. The *layout.pug* file is already written for you and available in D2L. It's based on Bootstrap's Starter template (but Pugarized). Download this file and save it in your *views/* folder, overwriting the existing file.

2. Make sure you understand the contents of this file. Open *layout.pug* in VSC. We're using CDNs for Bootstrap as outlined in its Starter template. The `.container-fluid` creates a `div` tag with a `class="container-fluid"` attribute. `container-fluid` is Bootstrap. It means use the entire available viewport, add some padding, and make the contents responsive. We then have three divs with `class="row"`. That means each will be row. The `.col-12` means each of our rows will use the entire 12 columns of the viewport, no matter its size. The first row will contain the HTML of the `Header` Component, the last row will contain the HTML of the `Footer` Component, and the middle row is defined elsewhere (the `block content`).

3. Replace the existing contents of the *index.pug* view template with that below:

   ```
   extends layout

   block content
     #msg-board.col-12 !{msgBoard}
   ```

   This template is a child of *layout.pug* and will insert `#msg-board.col-12 !{msgBoard}` in the `block content` of *layout.pug*. The `#` is how you declare a div with an `id` attribute. The `msg-board` id is for rendering the `MsgBoard` Component with the client-side code. The `!{msgBoard}` is for rendering the same Component server-side. Funky.

Task 10 - Installing and Setting Up nodemon

1. nodemon will make it way easier to debug problems. In a shell, run:

   ```
   npm install -g nodemon
   ```

2. In your project's *package.json* file, replace:

   ```
   node ./bin/www
   ```

   with:

   ```
   nodemon -e js,jsx ./bin/www
   ```

   for the value of `start` in the `scripts` object. That's it!

Task 11 - Summarizing What You Have Done and Testing If it Works

At this point, you should have done the following:

- Initialized the project using `express-generator` (Task 1)
- Changed the project to be MVC (Task 2)

- Coded the single Router, *app_router.js* (Task 4)
- Coded the single Controller, *msg.js*, which includes the data embedded in it (Task 5)
- Coded all the React Components (except NewMsg): `Header`, `Footer`, `MsgBoard` and `MsgList` (Tasks 6 - 8)
- Added the *layout.pug* and coded *index.pug* View Templates (Task 9)

This should be everything you need for SSR (no client-side rendering yet). To test it:

1. Open a bash shell in your project's root folder (where the *app.js* file is) and type:

   ```
   DEBUG=ics221-mb:* npm start
   ```

2. If no errors, in a new bash shell, install Browsersync:

   ```
   npm install -g browser-sync
   ```

3. Type the following in the shell after it has finished installing:

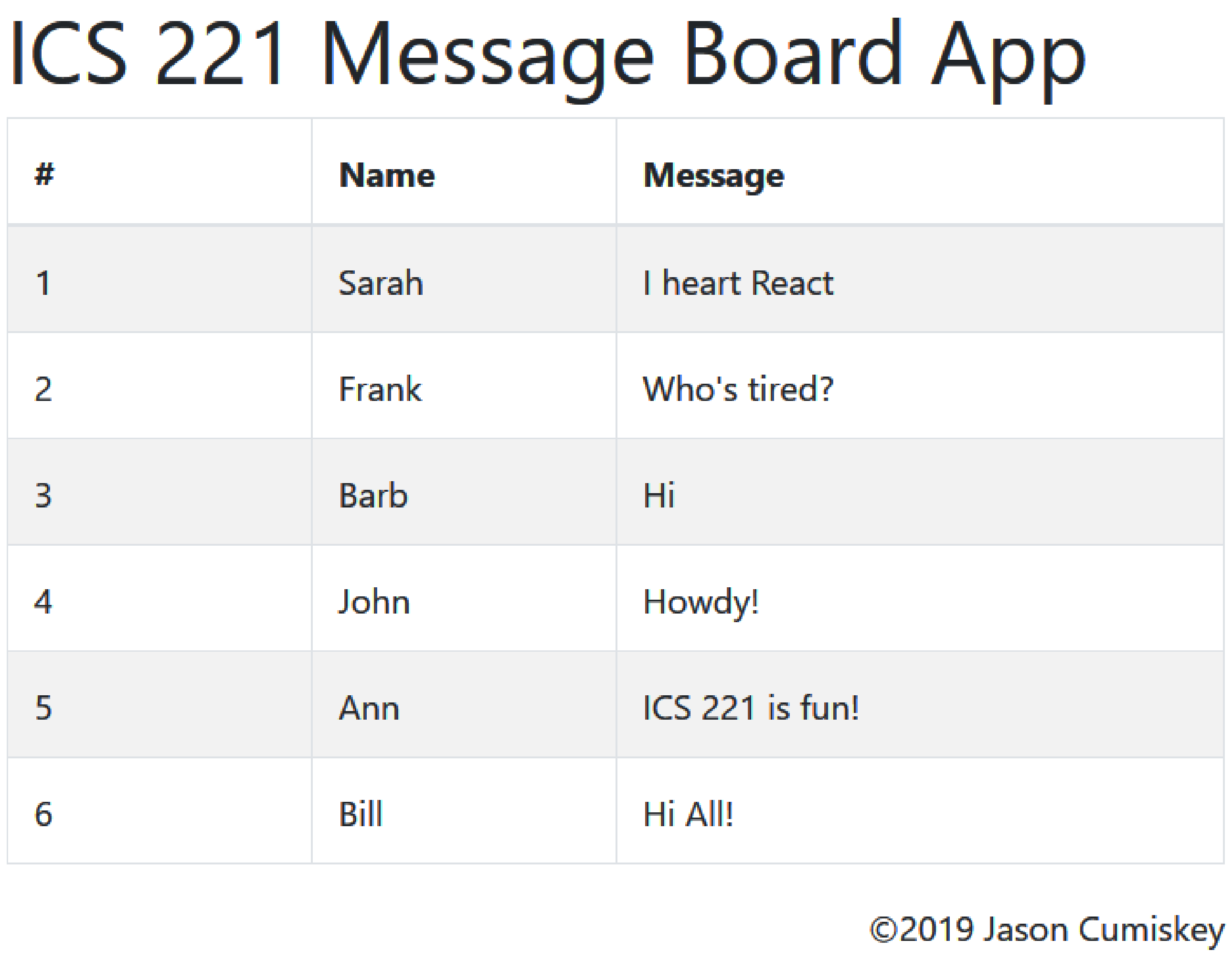
   ```
   browser-sync start --proxy localhost:3000 --reload-delay 2000 --files "**/*.pug,
   **/*.css, **/*.js, **/*.jsx"
   ```

   This will automatically open the default browser to your project. You should keep both Browsersync and nodemon running.

   Note: You may want to make this an alias in bash. You set aliases by using the command `alias` in a bash configuration file. For example:

   ```
   alias bs='browser-sync start --proxy localhost:3000 --reload-delay 2000 --files
   "**/*.pug, **/*.css, **/*.js, **/*.jsx"'
   ```

4. In the end, you should see something like:



ICS 221 Message Board App

#	Name	Message
1	Sarah	I heart React
2	Frank	Who's tired?
3	Barb	Hi
4	John	Howdy!
5	Ann	ICS 221 is fun!
6	Bill	Hi All!

©2019 Jason Cumiskey

Task 12 - Committing to Git

1. Once you've confirmed it works, time to commit it. Create a new git repository:

   ```
   git init
   ```

2. Add everything: `git add .`
3. Make the initial commit: `git commit -m "initial commit"`

Lab Submission (see top of lab for due date)

1. Demo to the Lab Instructor (15 marks):
   - The rendered Message Board App in the browser.
2. Do the Lab 2 Quiz in D2L (5 marks).