

**This assignment is due by the due date on the dropbox.**

This exercise is the fourth part of a multipart assignment to build a small MVC-based login system with some basic per-user functionality.

In this section, you will be converting the app to use React.js. React allows us to create dynamic user interfaces easily and efficiently. Though everything React does can be done in JS, we use React to easily allow us to make complex HTML structure and put it on the page.

As a bonus, React has a related project called React Native that allows you to build native Android or iOS apps with React JS code. It does mean a difference in styling and display though (since the devices are not powered by HTML/CSS).

You will be starting where you left off in Part-C.

1. To keep your DomoMaker A-C separate from the work we will do for DomoMaker D, please make a new GitHub repo called DomoMaker-D. Clone this empty repo, and copy all your source code into this new folder (except the .git folder). It will likely be faster to not copy the node\_modules folder, and to instead rerun npm install in the new folder.
2. You will also need to create a new Heroku app in order to deploy this new repo. Keep in mind that you will also need to configure your MONGODB\_URI, REDIS\_CLOUD\_URL, and NODE\_ENV config vars for this new app.
3. Once you are up and running in the new folder, we need to start setting up the project to work with React. Install the following dev dependencies to your project:  
@babel/core @babel/preset-react babel-loader

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

Babel (<https://babeljs.io/>) is a code transpiler that converts code between different formats. Historically it was useful for converting ES6 code to browser friendly ES5. Even though almost all browsers still in use now support ES6, it can still be useful.

React makes use of a special syntax known as JSX. JSX is not supported by any modern browser, and so we need to use babel to convert it to actual ES6 that the browser can use.

## DOMOMAKER - D

- Now that we have these tools installed, we need to configure webpack to work properly with babel. Go to webpack.config.js in the root folder of the project. Update it to include the following module entry.

```
module.exports = {
  entry: './client/client.js',
  module: {
    rules: [
      {
        test: /\. (js|jsx) $/,
        exclude: /node_modules/,
        use: {
          loader: "babel-loader",
        },
      },
    ],
  },
  mode: 'production',
}
```

This entry tells webpack to use the babel-loader module to handle all js and jsx files being run through it (except any in the node\_modules folder). Babel-loader will ensure our code gets run through the babel transpiler.

- Next, open the .babelrc file in the root folder of the project. .babelrc is the global configuration file for babel within the project. Add “@babel/preset-react” to the list of presets to ensure the react preset is used when babel runs.

```
{
  "presets": [ "@babel/preset-react" ]
}
```

- Download the additional starter files for this part of the assignment, found here. <https://github.com/IGM-RichMedia-at-RIT/DomoMaker-D-Start>

Note that you do not need to clone or fork this repo. Simply click the “Code” button, and “Download ZIP”. Unzip this, and replace the views files in your new DomoMaker-D code with the views files provided in the starter files of this assignment.

- Remove the signup.handlebars view. We will no longer need this in our new implementation.

The reason we no longer need this file is that we are moving to a structure with dynamic views. We will be using React.js to create the appropriate view for us. The login and signup will be on the same page, and will dynamically show up when necessary.

8. Open the new login.handlebars and app.handlebars. Notice that we no longer have a form in our #content section. Additionally, we are now including scripts for react.js.
9. Open the Account.js controller. We need to make some changes to support our new structure.

Remove the signupPage controller function **and** its module export. We no longer need this page because our new login page will work both for logins and signups.

10. Inside of your Account controller, add a getToken function. Our react app will now request csrf tokens when it makes requests. This will allow our react app to get a one-time token each time it needs to send a form (signup or login).

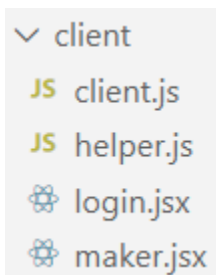
```
const getToken = (req, res) => {  
  return res.json({csrfToken: req.csrfToken()});  
};  
  
module.exports = {  
  loginPage,  
  login,  
  logout,  
  signup,  
  getToken,  
};
```

11. Now we need to update our routes.

In the router, remove the GET request to /signup. Replace it with a GET request to /getToken calling the getToken controller.

```
const router = (app) => {  
  app.get('/getToken', mid.requiresSecure, controllers.Account.getToken);  
  
  app.get('/login', mid.requiresSecure, mid.requiresLogout, controllers.Account.loginPage);  
  app.post('/login', mid.requiresSecure, mid.requiresLogout, controllers.Account.login);  
  
  app.post('/signup', mid.requiresSecure, mid.requiresLogout, controllers.Account.signup);  
  
  app.get('/logout', mid.requiresLogin, controllers.Account.logout);  
  
  app.get('/maker', mid.requiresLogin, controllers.Domo.makePage);  
  app.post('/maker', mid.requiresLogin, controllers.Domo.makeDomo);  
  
  app.get('/', mid.requiresSecure, mid.requiresLogout, controllers.Account.loginPage);  
};
```

12. Now go to our client folder. We will need to modify the client JS to use React. We're also going to make two different bundles for different pages, which we will configure in webpack in just a moment.
13. In the client folder, create helper.js, login.jsx, and maker.jsx. The .jsx extension is useful for react files, as it will properly enable the correct syntax highlighting for JSX.



We will be getting rid of client.js soon, but for now it will be useful to copy some code out of.

Maker.jsx will contain all of our code for the application that shows up once the user has logged in. Login.js will handle everything before the user is logged in, like the login and signup screens. Helper.js will contain a number of helpful functions that both files will import.

14. Copy the handleError and sendPost functions from client/client.js and paste them into helper.js. handleError will remain unchanged, but we will modify sendPost.

15. We want to change `sendPost` to make it a little more versatile. We will be modifying the version in `helper.js`. This new version will take in a third parameter called `handler`. This will be a function we can pass in to add functionality to handling requests. We call it at the bottom of `sendPost`, provided we passed something in for it.

```
const sendPost = async (url, data, handler) => {
  const response = await fetch(url, {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
    },
    body: JSON.stringify(data),
  });

  const result = await response.json();
  document.getElementById('domoMessage').classList.add('hidden');

  if(result.error) {
    handleError(result.error)
  }

  if(result.redirect) {
    window.location = result.redirect;
  }

  if(handler) {
    handler(result);
  }
};
```

16. We will also make a `hideError` helper function to hide the error popup. We can then export these functions from `helper.js`. Remember that we can use the same module require syntax in our client code as we do in our server code because webpack will convert the code for us.

```
const hideError = () => {  
  document.getElementById('domoMessage').classList.add('hidden');  
};  
  
module.exports = {  
  handleError,  
  sendPost,  
  hideError,  
};
```

You can now delete the client.js file from the client folder.

17. Speaking of webpack, we need to configure it a little bit more. Ideally, we want our client code to be broken up into two pieces: a “login bundle” which contains all the code needed to handle logging in and signing up, as well as an “app bundle” which contains all the code related to making domos.

Open webpack.config.js and modify it like so:

```
module.exports = {  
  entry: {  
    app: './client/maker.jsx',  
    login: './client/login.jsx',  
  },  
  module: {  
    rules: [  
      {  
        test: /\.js$/,  
        exclude: /node_modules/,  
        use: {  
          loader: "babel-loader",  
        },  
      },  
    ],  
  },  
  mode: 'production',  
  watchOptions: {  
    aggregateTimeout: 200,  
  },  
  output: {  
    path: path.resolve(__dirname, 'hosted'),  
    filename: '[name]Bundle.js',  
  },  
};
```

By defining multiple entry points, webpack will create multiple bundles. For the output filename, we use “[name]Bundle.js”. Webpack will pull in the name from the key in the

# DOMOMAKER - D

entry object, so we will generate appBundle.js and loginBundle.js.

18. Save the webpack.config.js. You can now run “npm run webpack” to bundle your code. You can also run “npm run webpackWatch” to have webpack run in watch mode. When in watch mode, any edit you make to the client code will trigger a rebuild of the appropriate bundles. Be aware that right now the bundles will have nothing in them, because the maker.jsx and login.jsx never import any code from helper and have no code of their own.
19. Inside of the login.jsx file in the client folder, add the following. First, we will require our helper code into this file (which we can do because of webpack). Then we will make a function for handling the submit event on the login form. Note that when we call helper.sendPost we do not pass in a handler function as the third param. This is because we do not need one. The built in redirect and error handling will work fine for us.

```
const helper = require('./helper.js');

const handleLogin = (e) => {
  e.preventDefault();
  helper.hideError();

  const username = e.target.querySelector('#user').value;
  const pass = e.target.querySelector('#pass').value;
  const _csrf = e.target.querySelector('#_csrf').value;

  if(!username || !pass) {
    helper.handleError('Username or password is empty!');
    return false;
  }

  helper.sendPost(e.target.action, {username, pass, _csrf});

  return false;
}
```

20. Similarly, add the following code in login.jsx in order to handle clicks to the signup button.

```
const handleSignup = (e) => {
  e.preventDefault();
  helper.hideError();

  const username = e.target.querySelector('#user').value;
  const pass = e.target.querySelector('#pass').value;
  const pass2 = e.target.querySelector('#pass2').value;
  const _csrf = e.target.querySelector('#_csrf').value;

  if(!username || !pass || !pass2) {
    helper.handleError('All fields are required!');
    return false;
  }

  if(pass !== pass2) {
    helper.handleError('Passwords do not match!');
    return false;
  }

  helper.sendPost(e.target.action, {username, pass, pass2, _csrf});

  return false;
}
```

21. Now that we have these event handlers, we can create our React components that will use them. Since this is a simple component that will not update when the user types into it, we will make a “functional stateless component” or FSC. Add the following to login.jsx. Note that we have proper syntax highlighting for the JSX since this is a .jsx file.

```
const LoginWindow = (props) => {
  return (
    <form id="loginForm"
      name="loginForm"
      onSubmit={handleLogin}
      action="/login"
      method="POST"
      className="mainForm"
    >
      <label htmlFor="username">Username: </label>
      <input id="user" type="text" name="username" placeholder="username" />
      <label htmlFor="pass">Password: </label>
      <input id="pass" type="password" name="pass" placeholder="password" />
      <input id="_csrf" type="hidden" name="_csrf" value={props.csrf} />
      <input className="formSubmit" type="submit" value="Sign in" />
    </form>
  );
};
```



## DOMOMAKER - D

22. We will create a similar function for rendering the signup page.

```
const SignupWindow = (props) => {
  return (
    <form id="signupForm"
      name="signupForm"
      onSubmit={handleSignup}
      action="/signup"
      method="POST"
      className="mainForm"
    >
      <label htmlFor="username">Username: </label>
      <input id="user" type="text" name="username" placeholder="username" />
      <label htmlFor="pass">Password: </label>
      <input id="pass" type="password" name="pass" placeholder="password" />
      <label htmlFor="pass2">Password: </label>
      <input id="pass2" type="password" name="pass2" placeholder="retype password" />
      <input id="_csrf" type="hidden" name="_csrf" value={props.csrf} />
      <input className="formSubmit" type="submit" value="Sign in" />
    </form>
  );
}
```

23. Now we need to initialize the page. The first part of that process will be to get our CSRF token from the server so that we can use it when we create our React components. Add the following to login.jsx.

```
const init = async () => {
  const response = await fetch('/getToken');
  const data = await response.json();
};

window.onload = init;
```

24. With our CSRF token retrieved, we can setup our event listeners. On the login.handlebars page there are two <a> tags styled as a Login and Signup button. When the user presses these buttons, we want to fill our 'content' section with the correct form. To accomplish this, we will add click event listeners to them and render out our react components when the user presses them.

```
const init = async () => {
  const response = await fetch('/getToken');
  const data = await response.json();

  const loginButton = document.getElementById('loginButton');
  const signupButton = document.getElementById('signupButton');

  loginButton.addEventListener('click', (e) => {
    e.preventDefault();
    ReactDOM.render(<LoginWindow csrf={data.csrfToken} />,
      document.getElementById('content'));
    return false;
  });

  signupButton.addEventListener('click', (e) => {
    e.preventDefault();
    ReactDOM.render(<SignupWindow csrf={data.csrfToken} />,
      document.getElementById('content'));
    return false;
  });
};
```

25. We also want to ensure that when the user first goes to the page, they see some content. Ideally the login screen. To accomplish this, we can also just render out the LoginWindow component immediately. Add the following to the bottom of the init() function.

```
signupButton.addEventListener('click', (e) => {
  e.preventDefault();
  ReactDOM.render(<SignupWindow csrf={data.csrfToken} />,
    document.getElementById('content'));
  return false;
});
```

```
ReactDOM.render(<LoginWindow csrf={data.csrfToken} />,
  document.getElementById('content'));
```

26. Ensure you have saved all your files and then run the “npm run webpack” command to build your bundle. You should see it build the loginBundle. It will also build the appBundle, but that will currently be empty.

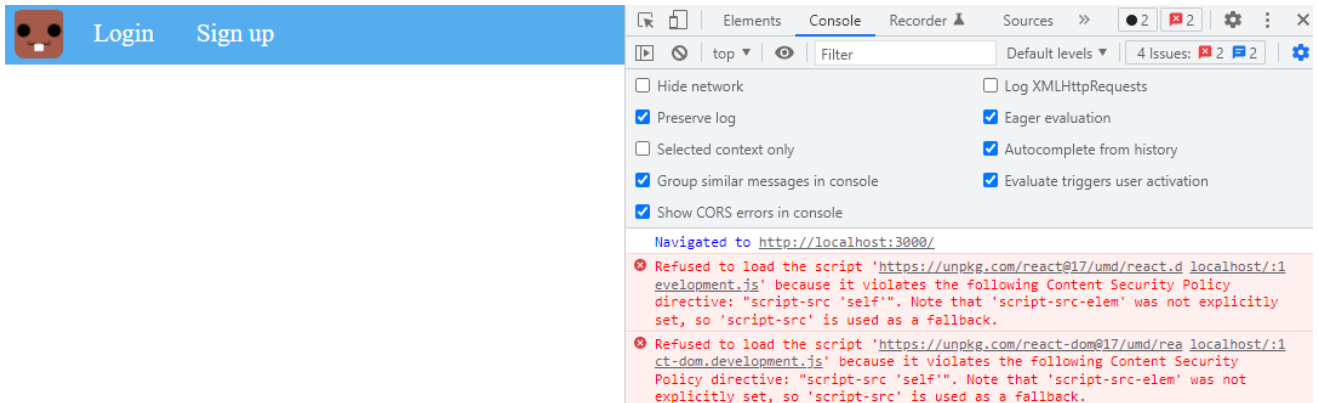
## RICH MEDIA WEB APP DEV II

# DOMOMAKER - D

27. Now that we have our login bundle, we can test it out. Start the server using “npm run nodemon”. Open your browser and navigate to localhost:3000

**Note: Make sure MongoDB and Redis are running or else the server will fail.**

When you launch the page you'll notice... nothing. The app doesn't appear to be working. If we open our client console, we will see a few errors.



These errors talk about the “Content Security Policy” or CSP. CSP is a very useful tool for securing our apps that can be configured from either the client html or from the server. CSP allows us to ensure our app is not loading in resources from places we wouldn't expect, and helps secure us against cross-site scripting attacks (among other things).

The downside is it also prevents us from loading resources we might want like the React and ReactDOM libraries, or images from image hosting sites.

CSP can be configured, and writing a good CSP can be a bit difficult. There are some websites (like this one <https://report-uri.com/home/generate>) that simplify the process greatly.

To help prevent headaches in the future for those of you that extend this assignment series for your project, we are going to opt to simply disable CSP for now.

28. CSP in our app is being configured by an npm package called Helmet. Helmet is an express middleware that does a bunch of work to secure our application from attacks. To disable CSP in helmet, we need to go to server/app.js. About halfway down the page, you should find `app.use(helmet());`; Change it to look like this.

```
const app = express();

app.use(helmet({
  crossOriginEmbedderPolicy: false,
  contentSecurityPolicy: false,
}));
```

29. Save the app.js file. Nodemon should restart your server. Now return to your browser and you should see your login page. If you don't be sure to use Ctrl+F5 to refresh your browser. This will ensure that the browser has not cached the version of the page with CSP enabled.



Username:	<input type="text" value="username"/>
Password:	<input type="password" value="password"/>
	<input type="button" value="Sign in"/>

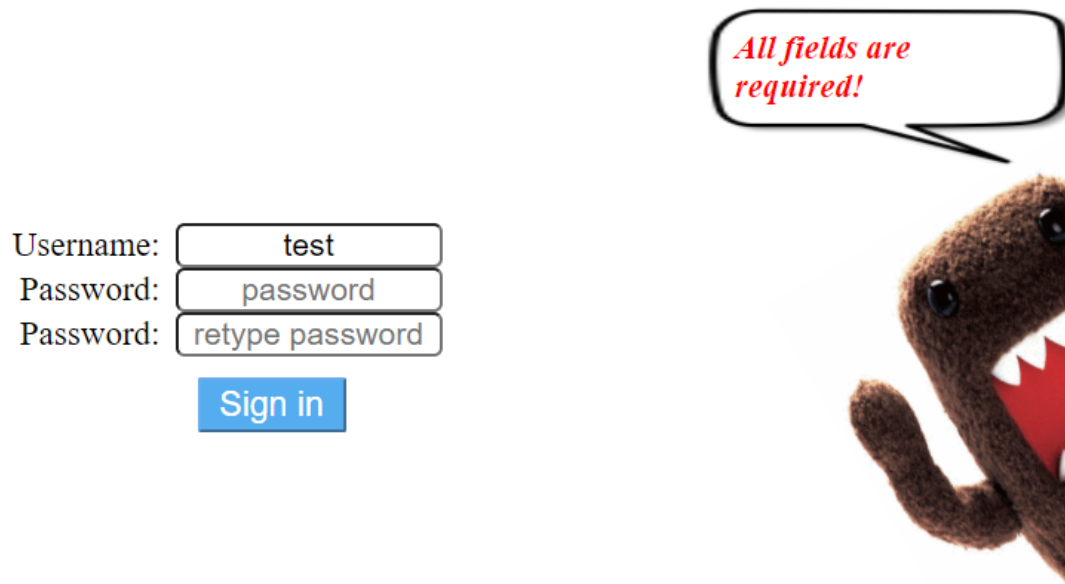
We want to ensure that this new React version of the login screen is working. Try the following:

30. Click the signup button on the page. It should instantly load the signup form without actually changing the page. Again, if there are errors check the Chrome console.

Username:	<input type="text" value="username"/>
Password:	<input type="password" value="password"/>
Password:	<input type="password" value="retype password"/>
	<input type="button" value="Sign Up"/>

## DOMOMAKER - D

31. Try signing up without a password. It should give you errors like before our react changes.



The image shows a login form with three input fields: 'Username:' containing 'test', 'Password:' containing 'password', and another 'Password:' containing 'retype password'. Below these fields is a blue 'Sign in' button. To the right of the form is a brown, fuzzy Domo character with a red mouth and white teeth. A speech bubble above the character contains the text 'All fields are required!' in red, italicized font.

32. Try signing up with two mismatching passwords.

33. Try to create an account that already exists.

34. Now try to create a new account. It should take you to the app if successful.

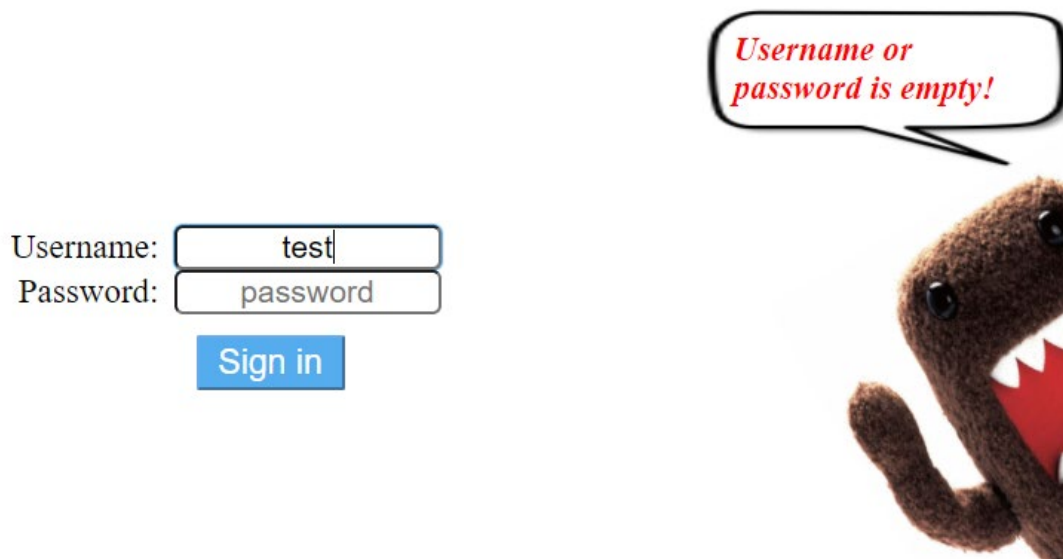
Notice there is nothing here. This is because we have not made the app React UI yet. We will do that shortly. Now we just have an app page without any components.

Once we add in the React components, this will work as it did before, but with much nicer interactions.



35. Click 'logout' and test logging in without a password. It should work like before React.

## DOMOMAKER - D



36. Sign in with a correct username and password. If successful, then our Login bundle is working correctly. Our react components allow us to easily render the necessary UI for the user's context. This means less work for the server, faster UI for the user and highly reusable code. We could now use these react components on other pages if desired.
37. We will now begin modifying the “maker” portion of the app to use React. Go to server/controllers/Domo.js. Simplify the makerPage function to simply render the app.handlebars page. In the next step we will write code to retrieve the list of domos for our client React code to render instead.

```
const makerPage = (req, res) => {  
  return res.render('app');  
};
```

38. Now we will make a getDomos function to retrieve all of the domos belonging to the logged in user.

This function will allow us to just get JSON responses of Domos for a user. This will allow our client app to update dynamically using React. We can pair the data on screen to the data from this function.

Once we do that, our app will update without changing pages. We'll be able to dynamically grab updates from the server and immediately update the UI on screen. This is a major improvement over our previous system that required the page to reload.

## DOMOMAKER - D

```
const getDomos = (req, res) => {
  return DomoModel.findByOwner(req.session.account._id, (err, docs) => {
    if(err) {
      console.log(err);
      return res.status(400).json({ error: 'An error occurred!' });
    }
    return res.json({ domos: docs });
  });
}

module.exports = {
  makerPage,
  makeDomo,
  getDomos,
};
```

39. Inside of the router, add a route for /getDomos that calls our getDomos controller.

```
const router = (app) => {
  app.get('/getToken', mid.requiresSecure, controllers.Account.getToken);
  app.get('/getDomos', mid.requiresLogin, controllers.Domo.getDomos);

  app.get('/login', mid.requiresSecure, mid.requiresLogout, controllers.Account.loginPage);
  app.post('/login', mid.requiresSecure, mid.requiresLogout, controllers.Account.login);

  app.post('/signup', mid.requiresSecure, mid.requiresLogout, controllers.Account.signup);

  app.get('/logout', mid.requiresLogin, controllers.Account.logout);

  app.get('/maker', mid.requiresLogin, controllers.Domo.makerPage);
  app.post('/maker', mid.requiresLogin, controllers.Domo.makeDomo);

  app.get('/', mid.requiresSecure, mid.requiresLogout, controllers.Account.loginPage);
};
```

40. Now open the client/maker.jsx file. We can require our helper file here. Then we will add our React components for our Domo app. Add the following:

```
const helper = require('./helper.js');

const handleDomo = (e) => {
  e.preventDefault();
  helper.hideError();

  const name = e.target.querySelector('#domoName').value;
  const age = e.target.querySelector("#domoAge").value;
  const _csrf = e.target.querySelector("#_csrf").value;

  if(!name || !age) {
    helper.handleError('All fields are required!');
    return false;
  }

  sendPost(e.target.action, {name, age, _csrf});

  return false;
}
```

41. Create a functional component in our client/maker.jsx to create our Add Domo form. This is very similar to our signup and login forms.

```
const DomoForm = (props) => {
  return (
    <form id="domoForm"
      onSubmit={handleDomo}
      name="domoForm"
      action="/maker"
      method="POST"
      className="domoForm"
    >
      <label htmlFor="name">Name: </label>
      <input id="domoName" type="text" name="name" placeholder="Domo Name" />
      <label htmlFor="age">Age: </label>
      <input id="domoAge" type="number" min="0" name="age" />
      <input id="_csrf" type="hidden" name="_csrf" value={props.csrf} />
      <input className="makeDomoSubmit" type="submit" value="Make Domo" />
    </form>
  );
}
```



42. Now we will create a component to display our list of Domos.

```
const DomoList = (props) => {
  if(props.domos.length === 0) {
    return (
      <div className="domoList">
        <h3 className="emptyDomo">No Domos Yet!</h3>
      </div>
    );
  }

  const domoNodes = props.domos.map(domo => {
    return (
      <div key={domo._id} className="domo">
        
        <h3 className="domoName"> Name: {domo.name} </h3>
        <h3 className="domoAge"> Age: {domo.age} </h3>
      </div>
    );
  });

  return (
    <div className="domoList">
      {domoNodes}
    </div>
  );
}
```

43. We also want a function that can load our list of domos from the server. When it gets a new list back, it should rerender the domoList component so that it is up to date with the data. We are keeping this outside of our domoList component because we will want to call it each time a new domo is created by the domoForm.

```
const loadDomosFromServer = async () => {
  const response = await fetch('/getDomos');
  const data = await response.json();
  ReactDOM.render(
    <DomoList domos={data.domos} />,
    document.getElementById('domos')
  );
}
```

## DOMOMAKER - D

44. Now that we have our DomoForm and our DomoList components, we simply need get a CSRF token and then render them to the page. To do this, we will use an init function.

```
const init = async () => {
  const response = await fetch('/getToken');
  const data = await response.json();

  ReactDOM.render(
    <DomoForm csrf={data.csrfToken} />,
    document.getElementById('makeDomo')
  );

  ReactDOM.render(
    <DomoList domos={[]} />,
    document.getElementById('domos')
  );

  loadDomosFromServer();
}
```

45. Now ensure you have saved all your files, and run “npm run webpack” to build the appBundle. Start your server.

46. Login to the DomoMaker app to test our changes. Our react components should render the form and Domos. If there are issues, check the Chrome console for errors.

*If this user does not yet have any Domos.*



The screenshot shows a web application interface. At the top, there is a blue header bar with a small avatar icon and a "Log out" link. Below the header, there is a form with two input fields: "Name:" with the placeholder text "Domo Name" and "Age:" with the placeholder text "Domo Age". To the right of these fields is a "Make Domo" button. Below the form, there is a large, empty rectangular box with a thin blue border, containing the text "No Domos yet".

No Domos yet

*If this user already has some Domos.*

 Log out

Name:

Age:

Make Domo

 Name: Bort Age: 25

 Name: Maryge Age: 35

47. Create a new Domo without an age. You should get the same errors as before React.

 Name: Bort Age: 25

 Name: Maryge Age: 35



All fields are required!

48. You will notice that when you add a domo, the entire page flickers before the new domo shows up in the list. This is because the page is reloading every time we create a domo, since that is what the makeDomo controller function tells it to do. We don't want this happening, as it defeats the purpose of using React.

Go to `/server/controllers/Domo.js`, and modify the `makeDomo` function. Rather than send a redirect command to the client on successful creation of a domo, we want to simply tell them it was a success by sending a 201 "Created" status code.

## DOMOMAKER - D

```

const makeDomo = async (req, res) => {
  if (!req.body.name || !req.body.age) {
    return res.status(400).json({ error: 'Both name and age are required!' });
  }

  const domoData = {
    name: req.body.name,
    age: req.body.age,
    owner: req.session.account._id,
  };

  try {
    const newDomo = new Domo(domoData);
    await newDomo.save();
    return res.status(201).json({ name: newDomo.name, age: newDomo.age });
  } catch (err) {

```

49. We also need to configure our client code so that it properly reloads the list of domos when one has been successfully created. Go to `/client/maker.jsx` and modify `handleDomo` to call `loadDomosFromServer` when it receives a response from `/maker`.

```

const handleDomo = (e) => {
  e.preventDefault();
  helper.hideError();

  const name = e.target.querySelector('#domoName').value;
  const age = e.target.querySelector("#domoAge").value;
  const _csrf = e.target.querySelector("#_csrf").value;

  if(!name || !age) {
    helper.handleError('All fields are required!');
    return false;
  }


  helper.sendPost(e.target.action, {name, age, _csrf}, loadDomosFromServer);

  return false;
}




```

## DOMOMAKER - D

50. Restart your server, refresh your browser, and test that creating a domo properly updates the domo list without the page flickering / reloading.

 Log out

Name:  Age:

	Name: Bort	Age: 25
	Name: Maryge	Age: 35
	Name: Hummer	Age: 40

Congrats, you've ported our DomoMaker to React.js. This was just a light intro to React. React has a lot of useful and reusable functionality. It provides a very powerful UI framework.

### **ESLint & Code Quality**

You are required to use ESLint with the AirBnB configuration for code quality checks. The `eslintrc` config file was given to you in the starter files. Your **server** code must pass ESLint to get credit for this part. Warnings are okay, but errors must be fixed. I won't require ESLint on the client side code in this assignment. ***If you cannot fix an error or need help, please ask us!***

You are also required to use the "pretest" script in order to run ESLint. I should be able to run "`npm test`" and have ESLint scan your code for me.

### **Continuous Integration**

You are required to use CircleCI for continuous integration. Your build must succeed and be handed in with the assignment. The `circle.yml` config file was given to you in the starter files. CircleCI will build based on your `npm test` and ESLint configuration.

For instructions on how to do this, look at the ***CircleCI Setup*** instructions on mycourses.

### **Build & Bundling Scripts**

Your app should have custom scripts to automatically restart node upon server changes (I suggest using nodemon) and create a client-side JS bundle (Webpack is installed and configured to do this with the "`npm run webpack`" command).

You should write your JS in separate files and create a bundled JS file from them. The only JS file that should go to the client is the bundled file.

RICH MEDIA WEB APP DEV II

# DOMOMAKER - D

## Rubric

DESCRIPTION	SCORE	VALUE %
<b>App works correctly</b> – all pages, forms and functions work correctly. App.js starts and runs correctly.		25
<b>Login/Signup Bundle</b> – Login/Signup is now done through React. The React UI switches pages between Login and Signup without reloading the web page. All Login/Signup functionality works.		30
<b>App Bundle</b> – The Domo app is now done through React. The React UI allows us to submit new Domos and dynamically update the DomoList without changing the page. All Domo functionality works.		35
<b>Bundle Creation</b> – App and Login bundle are properly generated by webpack and include the correct code.		10
<b>Git Penalty</b> – If your code is not in a Git repo or the link is not handed in, there will be a penalty.		-30% (this time)
<b>Heroku Penalty</b> – If your app is not functional on Heroku, there will be a penalty. This means you need to correctly hookup the addons on Heroku.		-30% (this time)
<b>CircleCI Penalty</b> – If your build is not successful on CircleCI, there will be a penalty.		-30% (this time)
<b>Build and Bundling Penalty</b> - If your package.json does not include properly working automatic build scripts for Node (nodemon or other) and client-side bundling scripts (babel, webpack, browserify, rollup or other), there will be a penalty.		-20% (this time)
<b>ESLint Penalties</b> – I should be able to run 'npm test' and have ESLint scan your code without errors. Warnings are acceptable. <b>You may not alter the rules in the .eslintrc file provided.</b>  <b>Check your code with ESLint often during development!</b>  <b>If you cannot fix an error, please ask us!</b>	1 Error	-10%
	2 Errors	-15%
	3 Errors	-20%
	4 Errors	-25%
	5+ Errors	-35%
<b>Additional Penalties</b> – These are point deductions for run time errors, poorly written code or improper code. There are no set values for penalties. The more penalties you make, the more points you will lose.		
<b>TOTAL</b>		100%

## Submission:

By the due date please submit a zip of your project to the dropbox. **Do not** include the node\_modules folder in the zip or in your git repo.

In the submission comments include the following:

- a link to your Heroku App (not the dashboard but the working web link from 'open app'),
- a link to your Github repo (if private, please add the TA and myself).
- a link to your circleCI build *in the submission comments*.