| **Type** | **Name** | **Latest commit message** | **Commit time** |
| --- | --- | --- | --- |
|  | [assets](https://github.com/DevMountain/simulation-3/tree/master/assets) | [Saved style guide in xd file](https://github.com/DevMountain/simulation-3/commit/488111aaa7ccd8d0a9205ee9435ed0decd153d90) | a year ago |
|  | [.gitignore](https://github.com/DevMountain/simulation-3/blob/master/.gitignore) | [Base for readme](https://github.com/DevMountain/simulation-3/commit/2699c8d038f822893fcb10a4e08997e40e645e0e) | a year ago |
|  | [PLANNING\_SHEET.md](https://github.com/DevMountain/simulation-3/blob/master/PLANNING_SHEET.md) | [Made readme and planning sheet rely on correct repo](https://github.com/DevMountain/simulation-3/commit/41e82bb07fca400cc84e08110ee58cd5db1304fe) | a year ago |
|  | [README.md](https://github.com/DevMountain/simulation-3/blob/master/README.md) | [fix formatting, replace body-parser](https://github.com/DevMountain/simulation-3/commit/3180cdfe475388ba3fb0c93bef12cf913cf588d3) | 2 months ago |

**README.md**

[[](https://camo.githubusercontent.com/d9cf9ee31887a3901288b88af700393f523b2901/68747470733a2f2f73332e616d617a6f6e6177732e636f6d2f6465766d6f756e7461696e2f726561646d652d6c6f676f2e706e67)](https://camo.githubusercontent.com/d9cf9ee31887a3901288b88af700393f523b2901/68747470733a2f2f73332e616d617a6f6e6177732e636f6d2f6465766d6f756e7461696e2f726561646d652d6c6f676f2e706e67)

**Project Summary**

This project is designed to give you an opportunity to build something from scratch and to teach you how to connect all the pieces of an application together. All of the instructions give you an idea of what order to do things in, but there won't be any guidance or solutions on how to write the code itself. The styling of the project is not included in the instructions at all and should be completed at your discretion.

This project is broken into three parts. The setup instructions are more detailed and are designed to get you started. The parts have varying levels of detail, with the newer concepts explained more. This gives you a chance to practice your skills on your own. Your mentors have also been asked to provide only minimal guidance. They can point you in the right direction, but cannot help you code. This project is a chance for you to combine and showcase the skills you've learned so far.

Good luck and work hard!

**Setup**

This section will help you create the files and install the packages you need.

**Color Palette & Font**

[](https://github.com/DevMountain/simulation-3/blob/master/assets/style_guide.png)

[**Google Font - Open Sans**](https://fonts.google.com/specimen/Open+Sans?selection.family=Open+Sans)

**Repository**

Do NOT clone this repository. You will be creating your own.

1. Navigate in the terminal to the folder you would like to store your simulation in, and then run create-react-app helo. cd into the folder to get started.
2. create-react-app starts your new project out as a local git repository, so you don't have to set that up. You can start using git commands right away.
3. Open up your Github profile page and click on 'Repositories'.
4. Click on the 'New' button. Name your repo (we suggest 'helo', the same name as your local folder). Do NOT initialize the repo with a README.
5. Now go back to your terminal and run git remote add origin [INSERT-GITHUB-URL-HERE] with the url from the remote repo you just created.
6. And finally, run git push origin master -u to push your local files to your remote repo for the first time.

Make sure to commit and push your code often. It's not fun to lose hours of work.

**React**

You have already created a React application as part of setting up the Github repo, so now you will start adding packages and files to that project.

1. Run npm i axios react-router-dom redux react-redux.
2. Create a Components folder inside of src.
3. Inside your Components folder create a folder for each component you will be using (Auth, Dashboard, Form, Post, Nav)
4. Inside each of these folders create a JavaScript file named the same thing. Make sure to capitalize the first letter!
5. Create a simple class component in the Auth, Dashboard, Form, Post, and Nav files. For now just return a div containing the component's name from the render method.
6. Now render the Nav, Auth, Dashboard, Form, and Post components in App.
7. Create a routes.js file inside the src folder. We will use this for our routing later.
8. Create a folder called ducks inside of src.
9. Create a store.js file and a reducer.js file inside of that folder.
10. Run npm start to make sure everything is working. You should see the names of all the components displayed.

**Server**

1. Run npm i express
2. Create a folder called server at the root of the project.
3. Create an index.js file and a controller.js file inside of that folder.
4. Open index.js and require your packages and the controller file.
5. Setup a basic Express server (you will add endpoints later, just get the server ready to run).
6. Open your package.json. Add your main property (so nodemon will work) and your proxy (so our axios requests will work).
   * Your main should look like "main": "server/index.js"
   * Your proxy should look like "proxy": "http://localhost:4000" using whatever port your server is setup to run on (the port should not be 3000 because that is what React will be running on).
7. Run nodemon and make sure your server runs.

**Database**

1. Run npm i massive dotenv
2. Create an .env file at the root of the project.
3. Open your .gitignore and add the .env file to it.
4. Open server/index.js and require masssive and dotenv (make sure to invoke config on dotenv).
5. Go to [Heroku](https://heroku.com/) and create a database (you can also use a database you already have created, but just be careful not to name your tables for Helo the same thing as any of the tables that already exist in your database)
6. Copy the connection URI for your new or existing database and save it in your .env file (make sure you put ?ssl=trueon the end of the string).
7. Create a folder called db at the root of the project.
8. Set up massive in your server using the connection string you saved in your .env file.
9. Make sure to run nodemon again and make sure your database is connecting.
10. Copy the connection string from your .env file into SQLTabs and create the users table and the posts table.
11. It's helpful to insert some dummy data into your database at this point to help you test as you go along.

**Part 1**

**Live example**[**here**](https://helo.devmountain.com/v2/part1/#/)**. Filled out planning sheet**[**here**](https://github.com/DevMountain/simulation-3/blob/master/PLANNING_SHEET.md)

In the first part you will set up routing and the ability to login/register.

Functionality of the Authentication View:

* A user should be able to enter a username and password into the input boxes.
* A user should be able to click the 'Login' button.
  + This should fetch the user's information from the database.
  + This should direct the user to the dashboard.
* A user should be able to click the 'Register' button.
  + This should create a new user in the database with a username, password and profile picture (You can use <https://robohash.org/> to generate the pictures).
  + This should send the new user information to the frontend.
  + This should direct the user to the dashboard.
* A user should not see the navbar.

Functionality of the Dashboard View:

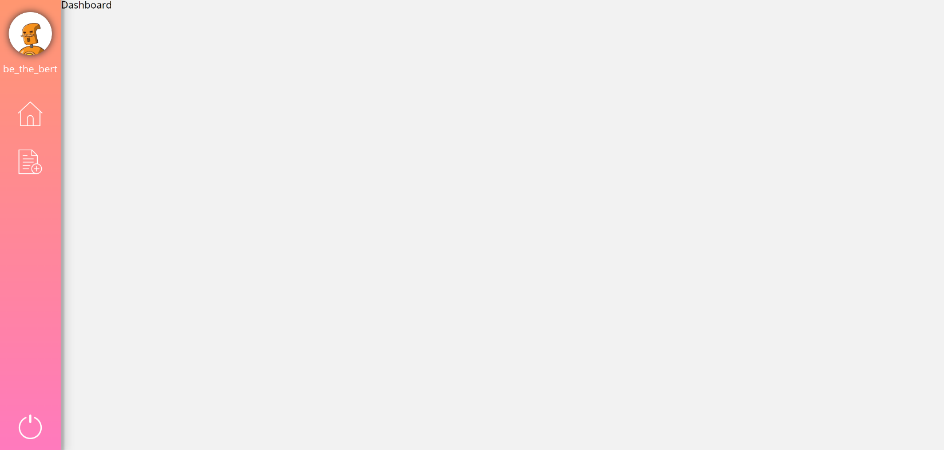
* A user should see their username and profile picture displayed in the navbar.
* A user should be able to able to navigate between the Dashboard and New Post views.
* A user should be able to click on the 'Logout' button.
  + This should cause the frontend to forget the user information.
  + This should direct the user to the Auth view.

**Design**

Authentication View

[](https://github.com/DevMountain/simulation-3/blob/master/assets/views/helo_redo_auth.png)

Dashboard View

[](https://github.com/DevMountain/simulation-3/blob/master/assets/views/helo_redo_pt1_dashboard.png)

**Step 1**

You are going to begin by setting up the routing.

* Open src/index.js and bring in the HashRouter from react-router-dom. Wrap App with HashRouter.
* Now open routes.js. Bring in Switch and Route from react-router-dom. Also bring in the Auth, Dashboard, Form, and Post components.
* Set up the Switch component as the default export of the file.
* Inside the Switch, setup a Route for the components you brought in.
  + The '/' path should render the Auth component.
  + The '/dashboard' path should render the Dashboard component.
  + The '/post/:postid' path should render the Post component.
  + The '/new' path should render the Form component.
* Open App.js and change what you're bringing into the component.
  + Remove Auth, Dashboard, Form, and Post from the component.
  + Instead bring in routes from routes.js and render it instead of the other components.
  + The Nav component should remain, as this will show on almost every view.
  + The Nav component should not render if the current view is the Auth view.
    - Use the location object found on props (this.props.location.pathname) to programatically check which view the user is currently on. If the path is '/', the Nav component should not return anything.
    - The location object is put onto props by react-router kind of like the match object, where you can find your routing parameters.
    - **IMPORTANT: Try to learn about this.props.location.pathname on your own, but if you get stuck the mentors can help explain what it is.**
  + If you open your application in the browser you should see the navbar in every view but Auth.
* Create the 'Home', 'New Post', and 'Logout' buttons in the Nav component.
  + 'Home' should navigate to the Dashboard view.
  + 'New Post' should navigate to the New Post view.
  + 'Logout' should navigate to the Auth view.

**Step 2**

Now that routing is set up, the first thing you will do is set up the authentication in the Auth view.

* Setup the username and password input boxes to update state.
  + Create initial state.
  + Setup the input boxes to update state.
* Create the 'Login' and 'Register' buttons in the Auth view.
* Write a POST endpoint in your server for registering.
  + The endpoint should pull the username and password off of the body.
  + The endpoint create a new user in the database.
  + The endpoint should respond with the newly created user.
* Write a method in Auth that sends an axios request to the endpoint you just wrote.
  + The axios request should take the username and password off of state and put them in the body of the request.
  + Once the response comes back, navigate to the Dashboard view.
  + Set up the 'Register' button to fire the method.
* Write a POST endpoint in your server for logging in.
  + The endpoint should pull the username and password off of the body.
  + The endpoint should pull the user with the matching username and password out of the database.
  + The endpoint should respond with the user.
* Write a method in Auth that sends an axios request to the endpoint you just wrote.
  + The axios request should take the username and password off of state and put them in the body of the request.
  + Once the response comes back, navigate to the Dashboard view.
  + Set up the 'Login' button to fire the method.

**Step 3**

Your users can now register and login, but your front end immediately forgets who just logged in. Let's fix that with Redux. First we will get Redux set up and sending values to the Nav component.

* Open reducer.js and create an object called initialState. This object should store the username, id, and profile picture for your user.
* Export a function named reducer. This function should take in two parameters: state (with the default value of initialState), and action.
* Set up a switch statement inside the reducer based on the action type. For now just setup a default case that returns state.
* Now open store.js and bring in createStore from redux and the reducer from reducer.js.
  + Create a store using the reducer you just brought in.
  + Export that store.
* Open src/App.js and bring in the Provider from react-redux and the store from store.js.
  + Wrap the Provider component around the HashRouter component.
  + Pass the store to the Provider.
* Go to Nav and bring in connect from react-redux.
  + Write the mapStateToProps function at the bottom of the file.
  + Take the username and profile picture off of the Redux state.
  + Now invoke connect, passing in mapStateToProps. Immediately invoke it again passing in the name of the component.
* Now if you console.log props inside the Nav component you should see the values coming from the Redux state.
* Set up the user profile picture and username to display. This will be pretty boring looking until we update these values in the next step

**Step 4**

Finally you will setup your Auth component to update Redux state.

* In reducer.js write an action builder that takes in a parameter for the user id, username, and profile picture.
* The function should return an action object with two properties: a type and a payload.
  + The type should be a string that describes what this action is supposed to do. These action type strings are usually stored in a constant outside the function.
  + The payload should be an object with a property for every parameter that was passed into the function.
  + The function should be exported.
* In your reducer function, add a case to the switch statement.
  + The case should match the action type you just wrote.
  + This case should return an object with all the same properties you set in initialState.
  + The values of the object should be based on the values of the action payload.
* In Auth, bring in connect from react-redux and the action builder you just wrote.
* Invoke connect at the bottom.
  + Pass in null for the first argument. This is because we don't need to use any values from Redux state.
  + Pass in an object for the second argument. Add the action builder you just brought into Auth as a value to this object.
* Now update both the register and login methods.
  + Once the user information has come back from the server, invoke the action builder and pass the information in.
  + The profile picture and username should now display in the navbar.

**Part 2**

**Live example**[**here**](https://helo.devmountain.com/v2/part2/#/)

In this part you will add the ability to view posts and create new ones.

Functionality of the Dashboard View:

* A user should be able to see all the posts created on Helo.
  + They should be able to choose if their own posts are included in the feed.
* A user should be able to search through the posts by title.
  + They should be able to click the 'Reset' button to clear the search term.
* A user should be able to click any of the posts to be taken to new page showing the post details.

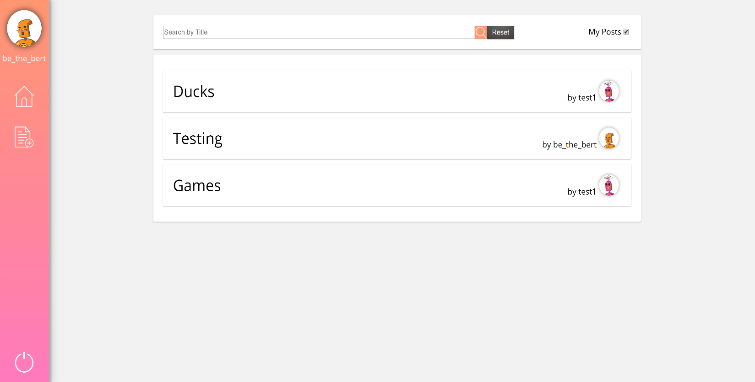
Functionality of the New Post View:

* A user should be able to enter title, image URL, and content values into the form.
* A user should be able to click the 'Post' button to create the post.
  + If the user has not logged in, an alert should pop up instructing them to log in and the post should not be created.
  + The user should be redirected to the Dashboard once the post has been created.

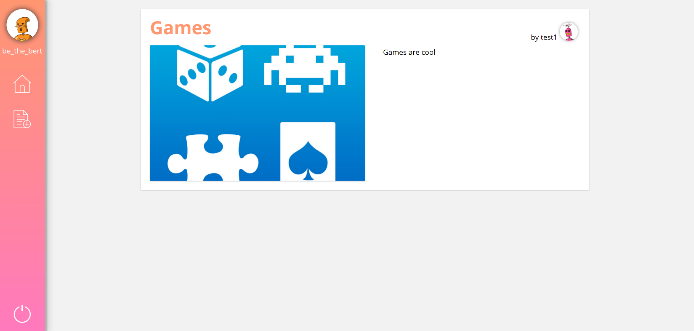
**Design**

Authentication View

[](https://github.com/DevMountain/simulation-3/blob/master/assets/views/helo_redo_auth.png)Dashboard View

[](https://github.com/DevMountain/simulation-3/blob/master/assets/views/helo_redo_pt2_dashboard.png)

PostView

[](https://github.com/DevMountain/simulation-3/blob/master/assets/views/helo_redo_post.png)

New Post View

[](https://github.com/DevMountain/simulation-3/blob/master/assets/views/helo_redo_new_post.png)

**Step 1**

First create the layout of the Dashboard.

* Set up an input box for the search functionality. Make sure to store the value in state.
* Create the 'Search' and 'Reset' buttons.
* Set up a checkbox to include the user's posts labeled 'My Posts'.
  + Make sure to store the value in state.
  + The value should be true intially.
* Store the list of posts in state and map over the list.
  + Each post should display the post title, and the author's username and profile picture.

**Step 2**

Then write the GET endpoint to retreive all posts. This endpoint is going to accept some queries: userposts(boolean) and search(string).

* The endpoint should return the post information and the author information for each post (*HINT:* Use a join)
* The endpoint should have a parameter for the user id.
* If userposts is true AND there is a search string, the endpoint should respond with all the posts where...
  + The title contains the search string.
* If userposts is false AND there is no search string, the endpoint should respond with all the posts where...
  + The current user is NOT the author.
* If userposts is false AND there is a search string, the endpoint should respond with all the posts where...
  + The current user is NOT the author.
  + The title contains the search string.
* If userposts is true AND there is no search string, the endpoint should respond with all the posts.

**Step 3**

Now set up Dashboard to hit the endpoint you just wrote.

* First you will need to connect to Redux and pull the user id off of Redux state.
  + Bring in the connect method from react-redux.
  + Write the mapStateToProps function at the bottom of the file. Pull the user id off of Redux State.
  + Invoke the connect method, passing in the mapStateToProps function and then invoking it with the component name as an argument.
* Write a method that sends an axios request to the endpoint you just wrote.
  + No matter the combination of queries, the request should send the user id from Redux state as a parameter.
  + If the user has entered a search term into the input box, send that string as a query.
  + If the 'My Posts' checkbox has been selected, send a userposts query with the value of true.
  + Once the request comes back, update state with the list of posts.
* Use a lifecycle hook to fire this method when the Dashboard first loads.
* Write a method to reset the search. This method will hit the same endpoint.
  + Send the user id as a parameter.
  + If the 'My Posts' checkbox has been selected, send a userposts query with the value of true.
  + Once the request comes back, update state with the list of posts and set the value of the search input to an empty string.

**Step 4**

Then you will add the navigation functionality so the user can view any of the posts shown in the Dashboard View.

* First write a GET endpoint in your server to retrieve a single post.
  + The endpoint should use a parameter to determine which post should be pulled from the database.
  + The endpoint should respond with the post title, image, and content for that post, as well as the username and profile picture of the post author (*Hint:* Use a join).
* Set up the Post component state to store the post title, image, content, and the username and profile picture of the post author.
* Write the JSX to display the values on state.
* Update where you are mapping over the list of posts in Dashboard to include a Link.
  + The Link should route the user to the Post view.
  + The Link should include the id of the post in the path as a parameter.
* Write a method in Post to hit the endpoint you just wrote.
  + The axios request should include the id of the desired post as a parameter.
  + The post id can be taken from the browser URL using the match object found on props.
  + Once the response comes back, update state with the post values.

**Step 5**

Now you will add the ability to add a new post.

* Set up input boxes for title, image URL, and content in the Form component.
  + Each input box should update state.
  + The image URL input box should populate an image preview.
* Create the 'Post' button.
* Next you will need to connect to Redux and pull the user id off of Redux state.
  + Bring in the connect method from react-redux.
  + Write the mapStateToProps function at the bottom of the file. Pull the user id off of Redux State.
  + Invoke the connect method, passing in the mapStateToProps function and then invoking it with the component name as an argument.
* Write a POST endpoint in your server.
  + The endpoint should accept a parameter for the user id.
  + The endpoint should pull the post title, image URL, and content off of the request body.
  + The endpoint should respond with the 'all good' status code once it has added the post to the database.
* Write a method in Form that sends a request to the endpoint you just wrote.
  + The axios request should include the user id as a parameter.
  + The request should send all the values stored in state in the body.
  + Once the response comes back from the server, redirect the user to the Dashboard.

**Part 3**

**Live example**[**here**](https://helo.devmountain.com/v2/part3/#/)

It's really cool that we can store data in Redux, but it's kind of a pain to keep sending the user's id along with all the axiosrequests, and if the user refreshes the page they have to log in again. So to solve both these problems, you will be implementing sessions in this part.

Additional functionality:

* A user should be able to refresh the page and still be logged into the site.
* A user should be able to logout.

**Design**

The changes for this part will deal exclusively with refactoring the data transfer, so the application's appearance will not change.

**Step 1**

In this step you will get sessions set up in your server.

* Run npm i express-session.
* Open server/index.js and require express-session.
* Set up the session (*Hint:* This is a top-level middleware, so you should use app.use)
  + Be sure to keep the session secret in your .env file.
  + Don't forget to set resave to false and saveUninitialized to true.

**Step 2**

Now you can use the session to store information! For this application all you will need to store is the user id.

* Modify your register endpoint. Before sending the user object to the front end, set req.session.userid equal to the user id you just pulled out of the database.
* Modify your login endpoint in the exact same way.

The session is automatically remembered by the user's browser, so you will be able to use the user id in future endpoints without storing it in your front end code. You still want to send the username and profile picture in the response for both of these endpoints however, because your front end code is going to display these values. You need an additional endpoint now to logout.

* Write a POST endpoint in your server.
  + It should have the URL '/api/auth/logout'.
  + This endpoint should destroy the session using req.session.destroy.

It may seem a little counter intuitive to use a POST endpoint for logging out, but authentication endpoints traditionally use POST because it is more secure and it can help avoid some unexpected behavior in some kinds of websites.

**Step 3**

Then you can update your existing endpoints to use the session information.

* Remove the parameter from your GET endpoint that returns multiple posts. You will be using the session instead.
  + Update the database queries to use req.session.userid instead of req.params.userid.
* Remove the parameter from your POST endpoint that creates a new post. You will be using the session instead.
  + Update the database query to use req.session.userid instead of req.params.userid.

**Step 4**

Next you can remove all instances of the user id in your front end code. It will all be taken care of in the back by sessions!

* Open reducer.js and remove the userId property from initialState, your reducer, and your action builders.
* Now go to the Dashboard component and remove the connection to Redux.
* Also remove the userId from the axios GET requests.
* Open the Form component and remove the connection to Redux.
* Also remove the userId from the axios POST request.

At this point your application should function just the same as before, but you have much cleaner code doing it.

**Step 5**

Finally, you will be sprucing up your Nav component so the user can refresh the page and still be logged in. Note: Because you have implemented sessions, the backend will remember your user when they refresh the page, but the front end won't look like it does. That's what you're going to fix in this step.

* Nav should be connected to Redux already, but you need to bring in an additional action builder.
  + Import the action builder from reducer.js that updates the user information.
  + Add the action builder you just brought in as a value to the object in the connect method.
* Write a GET endpoint in your server.
  + This endpoint should have the URL string '/api/auth/me'.
  + This endpoint should query the database for the username and profile picture using the user id stored in session.
  + This endpoint should respond with that user information.
* Write a method in Nav that hits the endpoint you just wrote.
  + Once the response comes back, invoke the action builder that you just connected, passing in the user information from the response.
* Use a lifecycle hook to fire this method as soon as the component loads.

Basically every time the navbar first loads, it is going to make a request to the backend to see if there is user information for the user id stored in sessions.

**Step 6**

The last piece is to set up the 'Logout' button to actually logout.

* Write a method in Nav to hit the '/api/auth/logout' endpoint.
  + Once the response comes back, invoke the action builder that clears the user information from Redux state.
* Update the 'Logout' button to fire this method, instead of firing the action builder directly.

**Congratulations! You've completed 31 competencies and built your third full-stack application!**

**Competencies**

Competencies covered by this project

**Setup**

["Student can use git to create, manage, and synchronize commits locally and remotely (Local and remote repository in-sync, .gitignore)"](https://github.com/DevMountain/simulation-3/blob/master/README.md#database)   
["Student can apply ES6 constructs in React for better code (import, export, destructuring)"](https://github.com/DevMountain/simulation-3/blob/master/README.md#react)   
["Student can use class based components in react and it's features (render, JSX, nested components)"](https://github.com/DevMountain/simulation-3/blob/master/README.md#react)   
["Student can create Node servers using the Express package (Server running)"](https://github.com/DevMountain/simulation-3/blob/master/README.md#server)   
["Student can create tables in a database"](https://github.com/DevMountain/simulation-3/blob/master/README.md#database)   
["Student can connect to their database in their NodeJS servers using Massive"](https://github.com/DevMountain/simulation-3/blob/master/README.md#database)

**Part 1**

**Step 1**   
["Student can add ReactRouter to their code base (HashRouter)"](https://github.com/DevMountain/simulation-3/blob/master/README.md#step-1)   
["Student can add ReactRouter to their code base (Switch, Route, component)"](https://github.com/DevMountain/simulation-3/blob/master/README.md#step-1) 

["Student can add ReactRouter to their code base (Link)"](https://github.com/DevMountain/simulation-3/blob/master/README.md#step-1)   
**Step 2**   
["Student can apply ES6 constructs in React for better code (arrow functions)"](https://github.com/DevMountain/simulation-3/blob/master/README.md#step-2)   
["Student can use class based components in react and it's features (state, setState, constructors)"](https://github.com/DevMountain/simulation-3/blob/master/README.md#step-2)   
["Student can use class based components in react and it's features (events)"](https://github.com/DevMountain/simulation-3/blob/master/README.md#step-2)   
["Student can interact with the web via axios and REST"](https://github.com/DevMountain/simulation-3/blob/master/README.md#step-2)   
["Student can create a RESTful API (Status codes)"](https://github.com/DevMountain/simulation-3/blob/master/README.md#step-2)   
["Student can create a RESTful API (express.json)"](https://github.com/DevMountain/simulation-3/blob/master/README.md#step-2)   
["Student can create a RESTful API (POST endpoint)"](https://github.com/DevMountain/simulation-3/blob/master/README.md#step-2)   
["Student can create SQL statements to manipulate data in their databases (Select)"](https://github.com/DevMountain/simulation-3/blob/master/README.md#step-2)   
["Student can create SQL statements to manipulate data in their databases (Insert)"](https://github.com/DevMountain/simulation-3/blob/master/README.md#step-2)   
["Student can run SQL commands in their NodeJS servers using Massive"](https://github.com/DevMountain/simulation-3/blob/master/README.md#step-2)   
**Step 3**   
["Student can utilize Redux in their code to manage state (store, reducer)"](https://github.com/DevMountain/simulation-3/blob/master/README.md#step-3)   
["Student can utilize Redux in their code to manage state (connect, mapStateToProps, share state)"](https://github.com/DevMountain/simulation-3/blob/master/README.md#step-3)   
**Step 4**   
["Student can utilize Redux in their code to manage state (actions, action builders, mapDispatchToProps object)"](https://github.com/DevMountain/simulation-3/blob/master/README.md#step-4)

**Part 2**

**Step 2**   
["Student can create a RESTful API (params)"](https://github.com/DevMountain/simulation-3/blob/master/README.md#step-2-1)   
["Student can create a RESTful API (queries)"](https://github.com/DevMountain/simulation-3/blob/master/README.md#step-2-1)   
["Student can create a RESTful API (GET endpoint)"](https://github.com/DevMountain/simulation-3/blob/master/README.md#step-2-1)   
["Student can utilize SQL Patterns in their databases (one-many)"](https://github.com/DevMountain/simulation-3/blob/master/README.md#step-2-1)   
["Student can create SQL statements to manipulate data in their databases (join)"](https://github.com/DevMountain/simulation-3/blob/master/README.md#step-2-1)   
**Step 3**   
["Student can use componentDidMount in their code"](https://github.com/DevMountain/simulation-3/blob/master/README.md#step-3-1)   
**Step 4**   
["Student can add ReactRouter to their code base (match object)"](https://github.com/DevMountain/simulation-3/blob/master/README.md#step-4-1)

**Part 3**

**Step 3**   
["Student can store data in sessions"](https://github.com/DevMountain/simulation-3/blob/master/README.md#step-3-2)

Don't worry too much about the competencies; they will be covered as you build this project.