# Lab: Movie Database

This lab is part of [“ReactJSFuncamentals” course @ SoftUni](https://softuni.bg/trainings/1643/reactjs-fundamentals-june-2017). The lab will consist of several parts with each step building on the previous one. The goal is to develop a sample (and simple ☺) application about movies – something like [IMDB](http://www.imdb.com/), or [RottenTomatoes](https://www.rottentomatoes.com/). We are going to have standard **User** **login** / **logout** and **authorization** with several **roles** (like “**Admin**” and “**Critic**). Also we are going to make **forms** to add **movies**, **comments**, **reviews** and also each **user** will be able to **vote** on the movies he **likes** / **dislikes**.

### Table of Contents

* **Part I – JSX, Components, Lifecycle, State** 
  + And routing (tomorrow)
* Part II – Thinking in the “React way” – Flux architecture with alt.
* Part III – Forms

## Project Specification

Design and implement a **“Movie Database” web application** (containing single page, dynamically updating with React) using JSX, CSS3, ReactJS, MongoDB and Node.js. The server configuration comes out of the box with the template archive. At the end our app should have the following functionality:

### Functionality

* **User Login**
  + Login in current application using username and password of already registered user.
* **User Register**
  + Register a new user by providing username and password.
* **User Logout**
  + Logouts from the application.
* **Movie add**
  + Create a new movie entry and save it to the database
  + Data must be saved in some sort of database.
* **Home**
  + List top-ten movies by user rating.
  + Add movie comments
  + Add movie vote. Each user can only vote once (1 – 10 scale)
  + Add movie review.
* **Find Movie**
  + Search movies by their name, description and genres
* **User Profile**
  + List information about the user – Name, roles, info, likes, reviews

|  |
| --- |
| This is how your app should look like: |

|  |  |  |
| --- | --- | --- |
| This is how your app should look like (with some movies): | | |
|  | | |
|  | |  |
| More screenshots: | | |
|  |  | |

### Routing

There will be different views which are displayed based on the routing (the URL). **Remember! We don’t want to refresh here!** How does that happen? **React** has provided us with **react-router.** But more on that tomorrow.

## Prerequisites

All of the pictures containing any sort of programming code (HTML and CSS too) are taken within **Webstorm**, but you really can use anything you like.

You will also need **Node.js** and **NPM.**

And **MongoDB.**

Keep in mind that this guide is divided into parts – each part covers a specific (single) lecture from [“ReactJSFuncamentals”](https://softuni.bg/trainings/1643/reactjs-fundamentals-june-2017) course. Thus said the first part (lecture) won’t cover the implementation of all the specified above functionalities and also some of the code will later be refactored or improved, based on the contents of the given lecture (example: user register and login forms will be included in the last part, concerning **forms**).q

# Part I – JSX, Components, Lifecycle, State

## Initial Setup

First extract **“template.zip”** file

Take a look inside package.json**.** A lot of stuff right? Since the project structure is not generated with **react-create-app**, but instead with manual **gulp** configuration, it steps on a lot of 3rd party libraries, which are all listed in **devDepencencies**.

Open a **cmd** or **terminal** window **in the root directory of the project** and type:

npm install

This can take a while. While everything is installing let’s take a look at bower.json. Bower is dependency management package for client side – basically the same as package.json, but only for the client. More info on bower [here](https://www.npmjs.com/package/bower).

So what’s in there? Just the stuff you would expect on the client: **jQuery**, **Bootstrap** and others. We need to install those dependencies as well. But first we need to install **bower globally**:

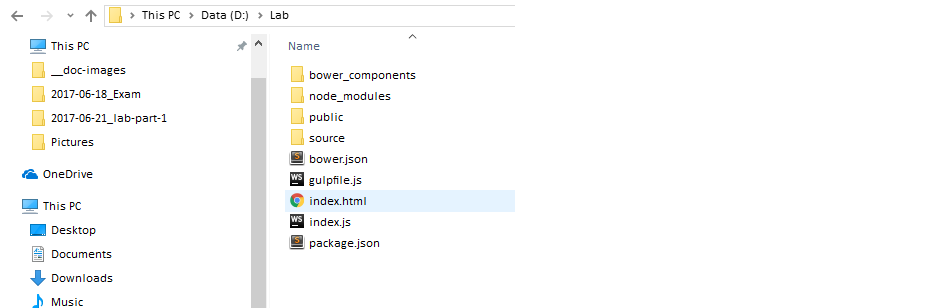
npm install –g bower

Now we can do this

bower install

And it will install the packages in /bower\_components

By now our project should look like this:



Now let’s start the **database**. Let’s create a directory for our **database**: C:\data\db. Once you have done so, open up a new cmd window and type:

mongod

**Tip:** If you don’t want to flood your **C: drive** you can create your directory in a custom location and then use:

mongod –-dbpath {path-to-directory}.

### Optionally: look through the server setup.

You can go to “**/source/server**” and look through the configuration. It is very similar with what we did in the [“ExpressJSFundamentals”](https://softuni.bg/trainings/1642/expressjs-fundamentals-may-2017) course, with some notable differences.

## Start the Project

Now to start our application we have to follow two steps:

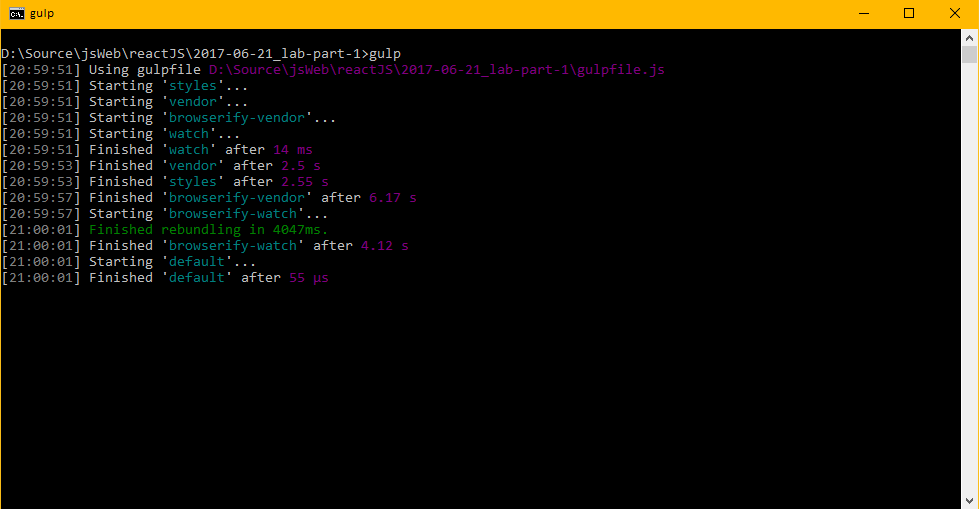
1. **Transpile** and bundle our **client**-**side** (**React**)
2. Start the server.

For the first step we are going to use **gulp**. You can take a look at the configuration in the gulpfile.js. Additional information on gulp [here](https://www.npmjs.com/package/gulp). There are two main tasks configured on the bottom: “**default**” and “**build”**. Both of them **compile**, **tanspile** and then **bundle** our **client-side** into minified ES5 javascript code, which will be exposed to the browser under the /publicdirectory. We also supply **source maps.** They reference the bundled code to our source code and allow the browser to show us relevant errors. Once the **bundling** is complete, you can take a look in the files under /public/scripts. Good luck debugging that ☺. More on source maps [here](https://developer.mozilla.org/en-US/docs/Tools/Debugger/How_to/Use_a_source_map).

Let’s go ahead and start the “**default**” gulp task. Inside the **project root** open a new cmd window and type:

gulp

Then we wait. When it is finished it should look something like this:



Do you recall we said there are two **gulp** tasks? “**build**” will bundle our client once and that’s it, while “**default**” will start a [**watchify**](https://www.npmjs.com/package/watchify) **batch job.** That means that every time when we make changes to the **client** files (**scripts** and **styles**) it should automatically trigger a **re**-**bundle** and update our /pubic dir.

If you now take a look under/publicyou should see the new bundled .jsfiles.

**Beware! In a perfect world the wachify task would work all the time. But we don’t live in a perfect word ☹. I don’t know on what basis and why, but sometimes it does not re-bundle after some small changes. When this occurs you can see react throwing checksum errors, or it may simply re-run your old code. Keep this in mind, to save some time debugging phantom issues. Use browser’s inspector tools to see if your changes are updated (Remember the source maps?)**

**With that being said – we are now ready to run the server. If you open up** package.json **again you will see “start” and “watch” scripts. Similar to our gulp tasks, “run” starts our application once, while “watch” listens for change and restarts automatically. If you plan to use “watch” make sure to install** [nodemon](https://www.npmjs.com/package/nodemon) **also:**

**npm install –g nodemon**

**Now lets finally start this app:**

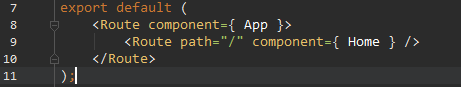
npm run watch

**And there we are. It may seem like a lot, but you’ll warm up quickly. If all works correct you should see a “Page Not Found”. Not very exciting. Let’s move on then.**

|  |  |
| --- | --- |
|  |  |

## Display Home Page

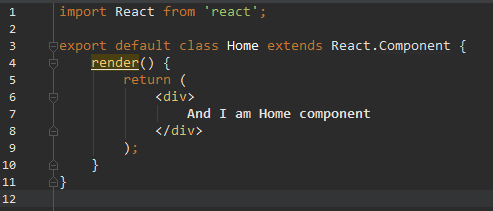
**If you open up** /source/client **you will see main.js and route.js files. Open up** main.js **and take a look. We have our base dependencies and we serve Router with** ReactDOM.render**. More on the router topic - tomorrow. We also import routes variable from** ./routes.js**. Take a look inside: we have a single routes component, but I don’t see any routes being implemented. Now we are getting a little ahead of ourselves, since we have not yet studied about React router, but we can’t really build anything without routes, so bear with me. Similar to how we defined routes in Express react routes consist of path and component, which will be visualized on that path. Let’s create our Home path:**

****

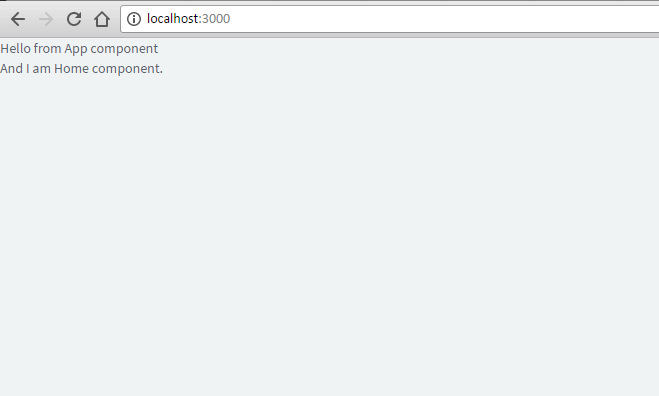
**Also don’t forget to import Home:**

****

Now let’s go to ./components and create file **Home.js**

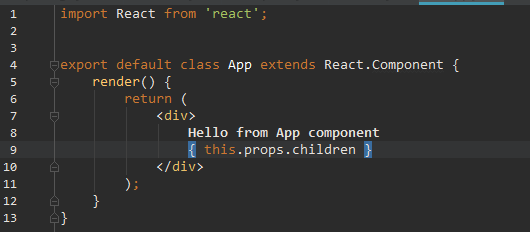
****

This code is pretty self-explanatory – it will write some text inside **<div>** tag. If you refresh the page you should see something like this:



Remember if you see **checksum** **errors**, that means **gulp** did not re-bundle after our changes. I usually resolve this problem by focusing the cmd window and clicking on it with left / right mouse buttons. Its sound lame, but usually it works. I think that happens, because when you focus a window the **OS** priorities it over other processes and that triggers the re-bundle. In extreme cases you will have to restart the **gulp task**, by pressing **ctrl + c** to terminate this batch and run “**gulp”** again. **Keep that in mind, since I think I have written it enough times already ☺ and will avoid to mention it further.**

Now let’s take a look at the App.js:



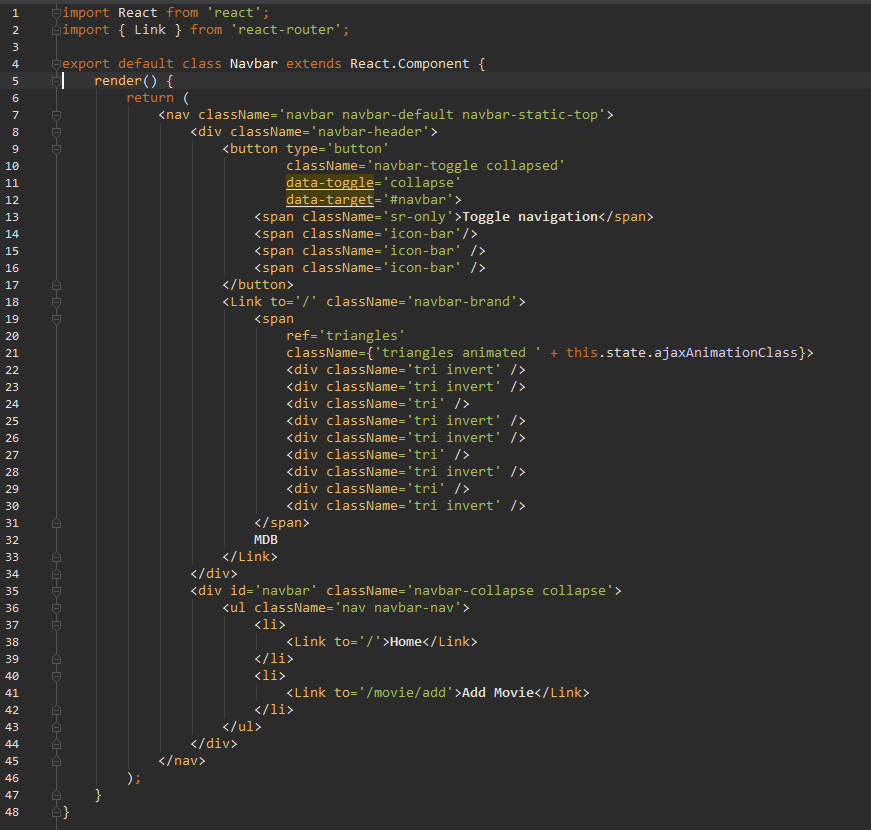
Now we can see where that text came from, but what is this { this.props.children }. Let’s go back in routes.js. You can see how the Route tag with path=”/” is indented. This means that our **home** route is child to our **app**, similarly to **HTML tags.** Back to App.js now. Does { this.props.children } make more sense now? It is simply a variable containing the equivalent **component** (or an array of **components**) of our **URL**, as defined in routes.js. It is very similar to the **templating engines** we used previously. Remember how we used to have one layout.\* file, and then multiple **views** extending this **layout. App component** is our **layout** and its children **components** are the separate **views.** If this is still somewhat unclear – no worries, **React router** will be the topic of tomorrow’s lecture.

## Default layout

Now let’s design our **layout**. We will implement two components **Navbar** and **Footer.**

In /source/client/components create a file Navbar.js

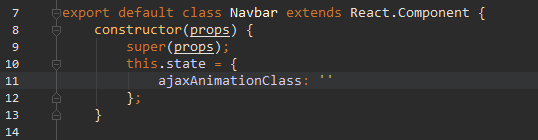
It is goint to contain our **navigation menu** and **user menu.**



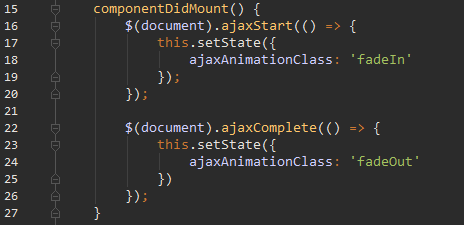
A lot of **JSX**, but I am sure most of you are familiar with **bootstrap.** I just decided to make it a bit pretty. Some interesting statements here:

1. <Link /> tag. This simply changes the **URL**, similarly to the standard **anchor** tag, onlywithout refreshing the page. We will use it a lot.
2. className is the equivalent of **class** in standard **HTML.** On **line 21:** we change this **className** dynamically**,** based on current **state.**

However as of now we have not implemented **component state.** Let’s do that:

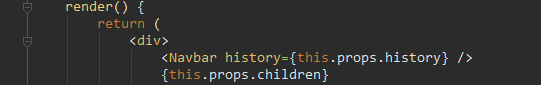


Frist we call super(props). This is mandatory in every class we implement as extension to React.Component**.** This calls the parent **constructor** and allows us to use the **props** variable. More on props later. Then we initialize the **component state** or simply **state.** In this caseour state is quite simple: it’s going to be either empty string, ‘fadeIn’ or ‘fadeOut’, depending on the **ajax status.** But so far our **state** is never updated. Let’s make our last **method**, which will change our **state**:

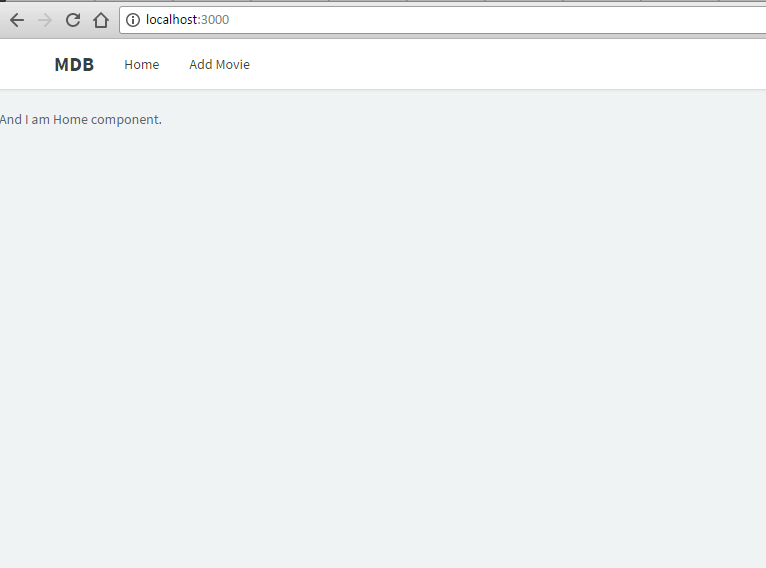
****

componentDidMount is a special **method**, which is called internally by R**eact** and is part of its **lifecycle.** More info on **lifecycle** [here](https://facebook.github.io/react/docs/state-and-lifecycle.html). In short words: componentDidMount is called when our **component** is rendered. So what happens inside? We attach ajaxStart and ajaxComplete **events**, which call this.setState, which then updates our **state** and triggers **re-render** of our **component.**

Finaly let’s import it in App.js



Now let’s refresh and woohoo:



If you followed the instructions you should see something like the above screenshot. Since we haven’t implemented any other routes then /, our links will not be working yet, don’t worry. Next is the **Footer component**

Let’s create Footer.js in /source/client/components directory:



So far it is very similar to Navbar.js. This time before return in the render method we see a variable mostResentMovies. That’s because we are going to show links for the five most recently added movies in our **footer**. this.state.recentMovies will hold an **array** of movie **documents,** taken from our database. Now let’s initialize our **state** and setup **ajax request** to get the data we need from the server.

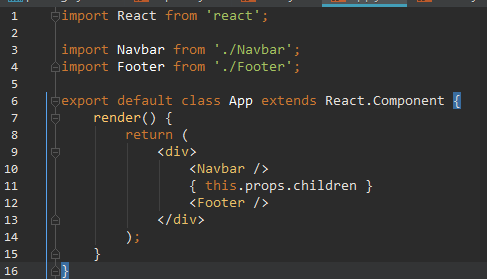


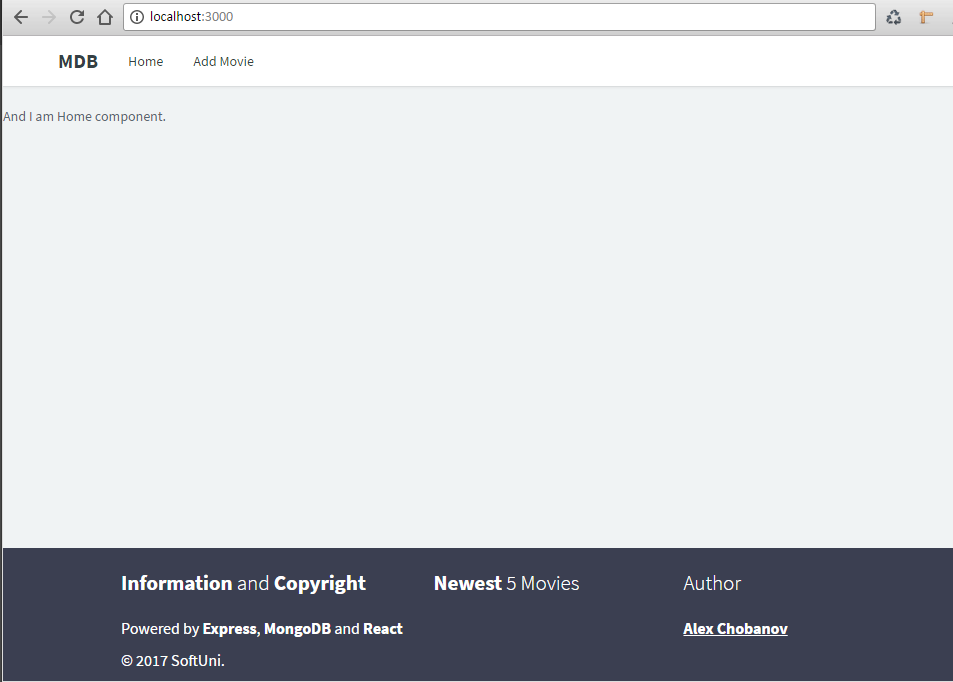
Constructor should be familliar, the only new **statement** we see is on **line 12**. This **binding** is making sure each time getFiveRecentMovies is called, it is called in the **context** of this class and this keyword will always point to this class. This is required because of the way **JavaScript prototypes** work and has nothing to do with **React.** The topic is quite complicated and there is no need to delve in it, just remember that **ANY CUSTOM METHOD** you write should be **bound** to the context of its **class** in such a mannor**.** This is not needed for componentDidMount, componentWillUnmount and render, because they are already **bound internally**, inside React.Component parent **class**.

Speaking of **internal** components, we see a new one – componentWillUnmount. I think it name does a good job explaining its purpose. In this case we need it, because we are calling getFiveRecentMovies on inside setInterval. And it is always mandatory to clear our **intervals**, when they are no longer needed, to prevent unnecessary server strain.

As for getFiveRecentMovies – it is a simple **ajax request** to our server. If you are curious as to how is this handled on the **back-end**, take a look under /source/server/config/server-routes.js.

Finally let’s import our **footer** inside /source/components/App.js. Are you ready? Here is how it should look like now:



Then refresh and you should see the **footer**: 

Hooray! We are ready with our default layout. Now we will focus on building each individual **view**.

## Add Movie view

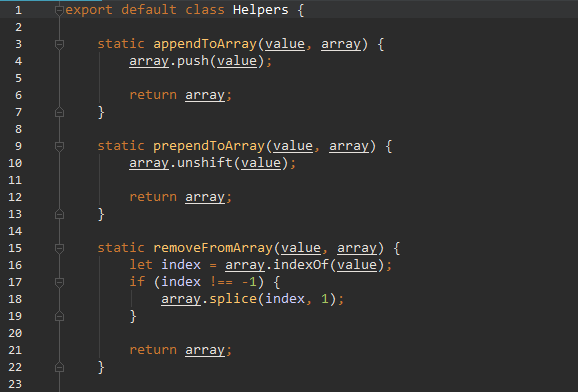
Go into /source/client/components and create MovieAdd.js:

import React from 'react';  
  
import Helpers from '../utilities/Helpers';  
  
export default class AddMovie extends React.Component {  
 constructor(props) {  
 super(props);  
  
 this.state = {  
 name: '',  
 description: '',  
 genres: [],  
 genresValidationState: '',  
 nameValidationState: '',  
 posterValidationState: '',  
 helpBlock: ''  
 };  
 }  
  
 handleSubmit(e) {  
 e.preventDefault();  
  
 let name = this.state.name.trim();  
 let genres = this.state.genres;  
 if (!name) {  
 this.setState({  
 nameValidationState: 'has-error',  
 helpBlock: 'Please enter Movie name!'  
 });  
 }  
 if (genres.length === 0) {  
 this.setState({  
 genresValidationState: 'has-error',  
 helpBlock: 'Please enter Movie name!'  
 });  
 }  
  
 if (name) {  
 let data = {  
 name: this.state.name,  
 description: this.state.description,  
 genres: this.state.genres  
 };  
  
 let request = {  
 url: '/api/movies/add',  
 method: 'POST',  
 data: *JSON*.stringify(data),  
 contentType : 'application/json'  
 };  
 $.ajax(request)  
 .done(() => {  
 this.props.history.pushState(null, '/');  
  
 })  
 .fail(() => *console*.log('movie post fail.'));  
 }  
 }  
  
 handleNameChange(e) {  
 let name = e.target.value;  
 this.setState({  
 name: name  
 });  
 }  
  
 handleDescriptionChange(e) {  
 let description = e.target.value;  
 this.setState({  
 description: description  
 });  
 }  
  
 handleGenresChange(e) {  
 let genreValue = e.target.value;  
 *console*.log('MovieAdd state', this.state);  
 if (this.state.genres.indexOf(genreValue) === -1) {  
 this.setState(prevState => ({  
 genres: Helpers.appendToArray(genreValue, prevState.genres)  
 }));  
 } else {  
 this.setState(prevState => ({  
 genres: Helpers.removeFromArray(genreValue, prevState.genres)  
 }));  
 }  
 }  
  
 render() {  
 return (  
 <div className='container'>  
 <div className='row flipInX animated'>  
 <div className='col-sm-8'>  
 <div className='panel panel-default'>  
 <div className='panel-heading'>**Add Movie**</div>  
 <div className='panel-body'>  
 <form onSubmit={ this.handleSubmit.bind(this) }>  
 <div className={ 'form-group ' + this.state.nameValidationState }>  
 <label className='control-label'>**Name**</label>  
 <input type='text' className='form-control' ref='nameTextField'  
 value={ this.state.name }  
 onChange={ this.handleNameChange.bind(this) } autoFocus/>  
 <span className='help-block'>{ this.state.helpBlock }</span>  
 </div>  
 <div className='form-group'>  
 <label className='control-label'>**Description**</label>  
 <textarea className='form-control'  
 rows="5"  
 value={ this.state.description }  
 onChange={ this.handleDescriptionChange.bind(this) } />  
 </div>  
 <div className={ 'form-group ' + this.state.genresValidationState }>  
 <div className='checkbox checkbox-inline'>  
 <input type='checkbox' name='genres' id='action' value='Action'  
 checked={ this.state.genres.indexOf('Action') !== -1 }  
 onClick={ this.handleGenresChange.bind(this) }/>  
 <label htmlFor='action'>**Action**</label>  
 </div>  
 <div className='checkbox checkbox-inline'>  
 <input type='checkbox' name='genres' id='horror' value='Horror'  
 checked={ this.state.genres.indexOf('Horror') !== -1 }  
 onChange={ this.handleGenresChange.bind(this) }/>  
 <label htmlFor='horror'>**Horror**</label>  
 </div>  
 <div className='checkbox checkbox-inline'>  
 <input type='checkbox' name='genres' id='sci-fi' value='Sci-fi'  
 checked={ this.state.genres.indexOf('Sci-fi') !== -1 }  
 onChange={ this.handleGenresChange.bind(this) }/>  
 <label htmlFor='sci-fi'>**Sci-fi**</label>  
 </div>  
 <div className='checkbox checkbox-inline'>  
 <input type='checkbox' name='genres' id='fantasy' value='Fantasy'  
 checked={ this.state.genres.indexOf('Fantasy') !== -1 }  
 onChange={ this.handleGenresChange.bind(this) }/>  
 <label htmlFor='fantasy'>**Fantasy**</label>  
 </div>  
 <div className='checkbox checkbox-inline'>  
 <input type='checkbox' name='genres' id='romance' value='Romance'  
 checked={ this.state.genres.indexOf('Romance') !== -1 }  
 onChange={ this.handleGenresChange.bind(this) }/>  
 <label htmlFor='romance'>**Romance**</label>  
 </div>  
 <div className='checkbox checkbox-inline'>  
 <input type='checkbox' name='genres' id='thriller' value='Thriller'  
 checked={ this.state.genres.indexOf('Thriller') !== -1 }  
 onChange={ this.handleGenresChange.bind(this) }/>  
 <label htmlFor='thriller'>**Thriller**</label>  
 </div>  
 <div className='checkbox checkbox-inline'>  
 <input type='checkbox' name='genres' id='adventure' value='Adventure'  
 checked={ this.state.genres.indexOf('Adventure') !== -1 }  
 onChange={ this.handleGenresChange.bind(this) }/>  
 <label htmlFor='adventure'>**Adventure**</label>  
 </div>  
 </div>  
 <button type='submit' className='btn btn-primary'>**Submit**</button>  
 </form>  
 </div>  
 </div>  
 </div>  
 </div>  
 </div>  
 );  
 }  
  
}

You do not see the entirety of the file here, but this time you can **copy-paste** it, because we are cheating a little bit. This **component** is all about **forms** which work very differently with React. We will study **forms** in details in our last lecture on **29th of June** - **Events&Forms.**

You may notice that we have imported **Helpers** from a file inside /source/client/utilities. This is a file containing helper functions, which we need on different places across our project. I have created it primarily to keep the react **components** clean and crisp. In this component **Helpers** is used inside handleGenresChange method, modify our genres field of this.state. Let’s implement those helper functions.

Go to /source/client/utilities and create a file Helpers.js:

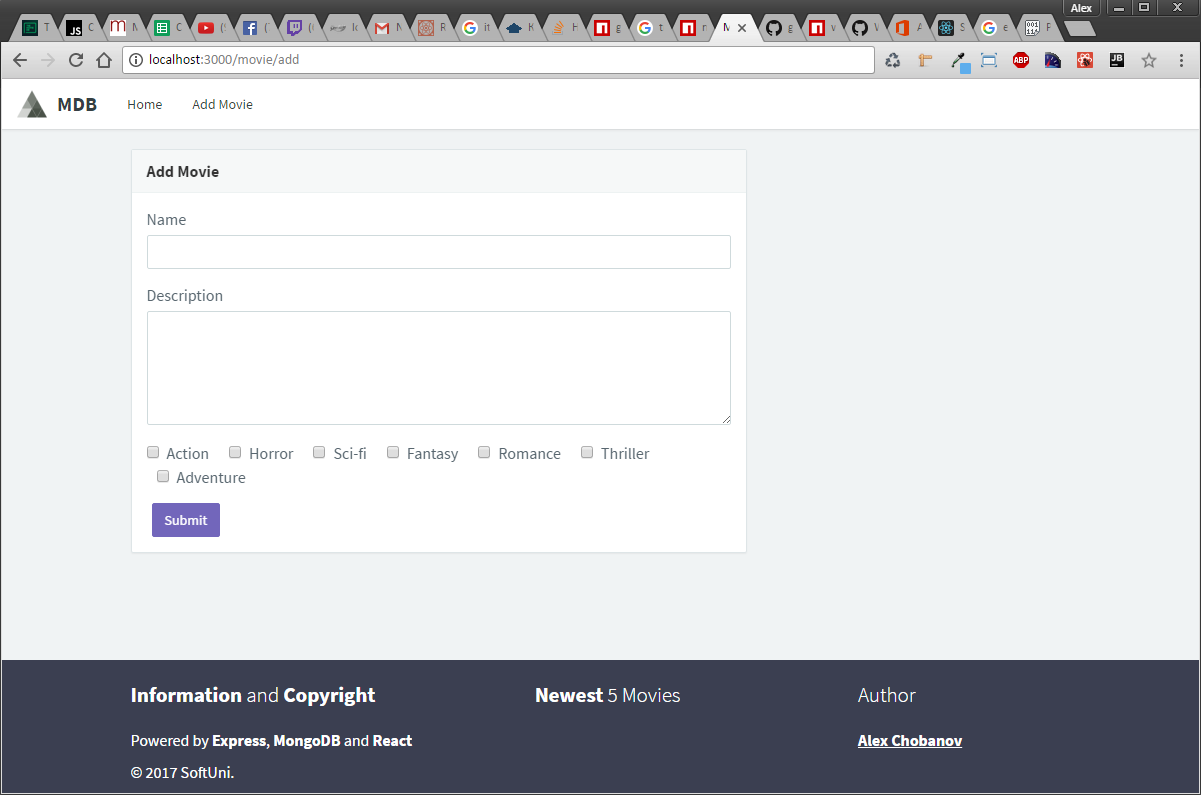


Now we need to modify routes.js to include our MovieAdd **component.** Go to /source/client/routes.js and insert our new route, just under **home** route:



**There is one last thing you need to do, but that is up to you…**

And when you are ready – refresh and click on “Add Movie” in our **Navbar:**

****

If you followed my directions closely, submitting the **form** should **redirect** to **home.** Now that we have this **component** working I have a few **tasks** for you:

1. Take a closer look at the code you just **copy-pasted.** Ignore all the **form** stuff for now and read through the **state** and **JSX**.
2. Look at handleSubmit function. Skip through the validation and straight to the **ajax.** You can see that on successful request, inside the done() callback we say: this.props.history.pushState(null, ‘/’). This is how we can programmatically change the **URL** (without refreshing, of course). But as stated previously this is the topic of **tomorrow’s** lecture, so won’t delve in too much.

Great job so far! Staring to like React already?

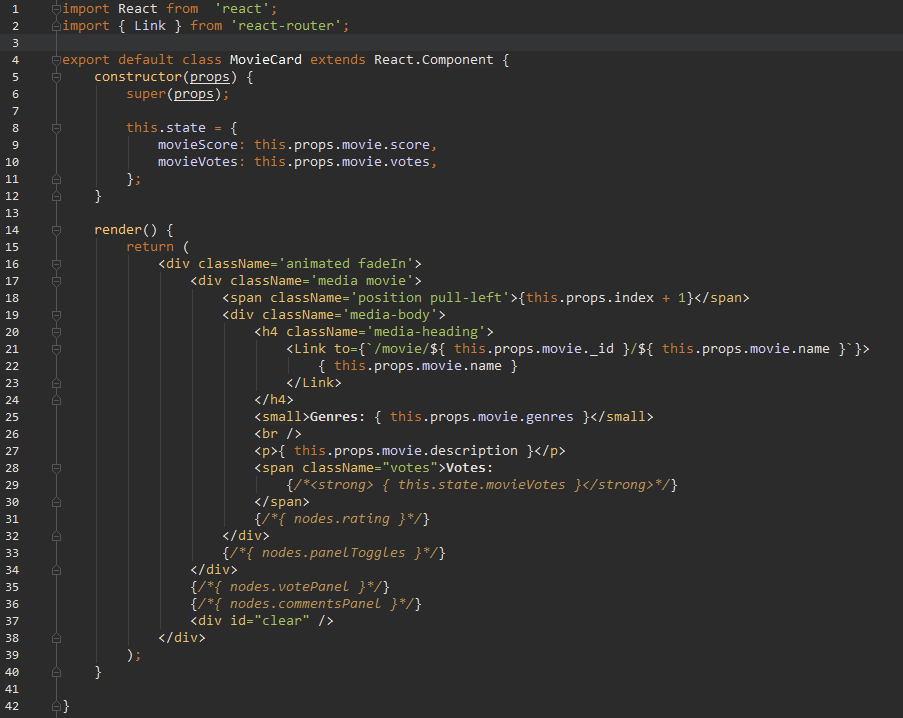
## Introducing the notion of “sub-components”

**DISCLAIMER:** The term **sub-components** is of my own making, since I find it good for the project structure and the code quality over-all if I make this general distinction between a **component** (which plays the role of a **view**) and **sub-components** (as in the building blocks of the standard **components**). I don’t know of it is considered a good practice by the majority out there. But in the context of this lab, you are stuck with me ☺. That being said you are free to do whatever you like, just keep in mind that if you stray from my structure, you will have to do some refactoring. Alex Chobanov is not responsible, neither accountable for any not-working code, if instructions were not followed strictly.

So far we have been designing our app in a way that each **component** is corresponding to a **view page**. But this is no way to design react application. Let explain briefly what do we want to achieve and why do we need **sub-components.** We want to list the movies we add in our **Home component**, but we will also list them later under **UserProfile** page. What we could do to achieve this is to create two separate **components** - **Home** and **UserProfile**, which would have the same logic to list our movies. But since we are trying to avoid unnecessary code duplication, we are instead going to create a **sub-component: MovieCard**, which will be responsible for the display of a single movie, and we are going to render **array** of **MovieCard components** in both **Home** and **UserProfile**

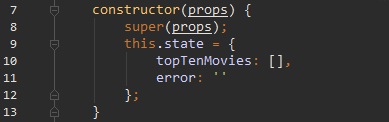
### MovieCard sub-component

Go to /source/client/components/sub-components and create MovieCard.js:

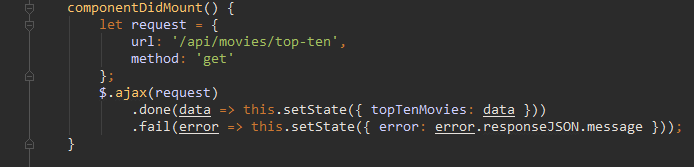


We will need the **commented-out** sections for later. Now this **sub-component** should display a Movie card, with data it receives from its parent **component,** in our case – Home.js. You notice a lot of this.props.movie being used, but how do we pass down the movie data? Let’s go to Home.js and see. Right now it looks very plain. Let’s get to work:

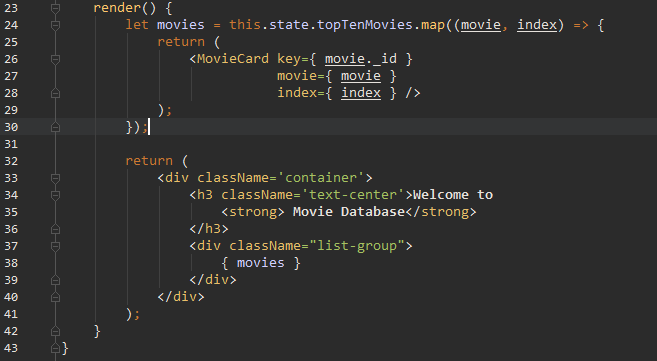
In order to display some movies we need to get **data** from the **database.** But before that we need to set **component state.** Change the constructor of Home.js to:

Why topTenMovies I have pre-configured the **back-end controller,** to query the **database** for the 10 highest-rated movies. Thus topTenMovies.

The following picture show how we are going to get the data:

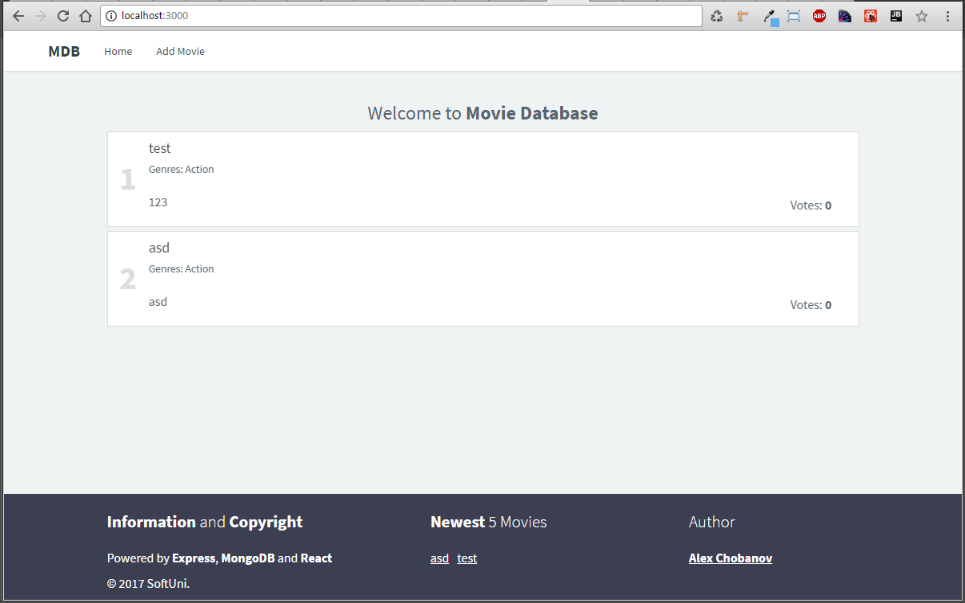


And now let’s update the render method as well:



this.state.topTenMovies will hold **array** of movie **documents.** You can see how we are mapping this data to an **array** of **MovieCard sub-components** and store it in movies variable. You can also see how we are passing **props** to **sub-component.** It is very similar to **HTML attributes.** All this data will be available to **MovieCard** through this.props.

Now go ahead and add one or two test movies and your **home** page should look like this:

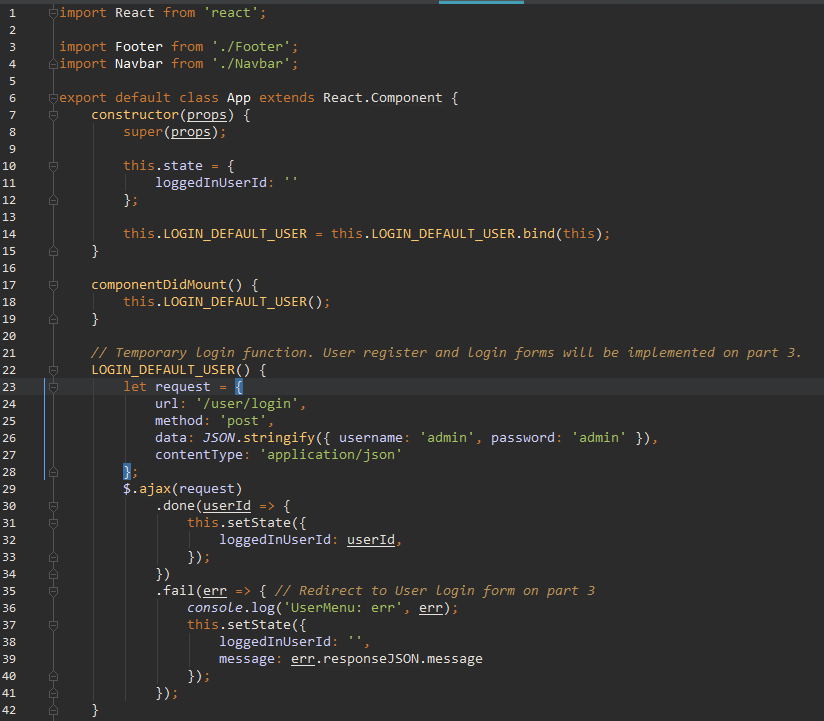


Right now these are not very attractive. We said they are listed by highest user votes. But as of now we cannot vote at all, or see any rating for that matter. However, since this is going to involve some **form** action we are going to leave it for the last part of this lab.

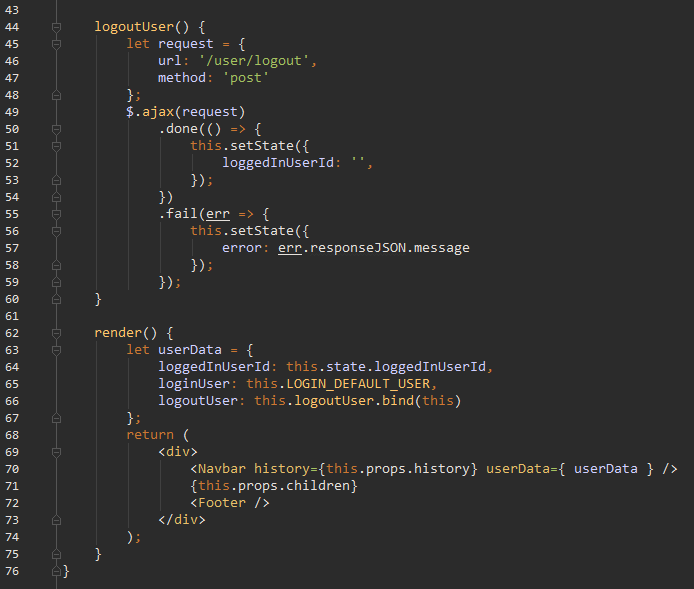
### Pseudo Login and UserMenu sub-component

Before we continue with the next step, which is going to be **User profile,** we have to implement some form of **register / login.** Our back-end is already configured to seed admin user – nothing fancy, but if you want you could take a look inside /source/server/config/database.js. We could implement **login** and **register forms,** but again that involves, well… you guessed it – **forms.** So for now we are going to hardcode a **login** function.

Let’s go to App.js and change it like so:

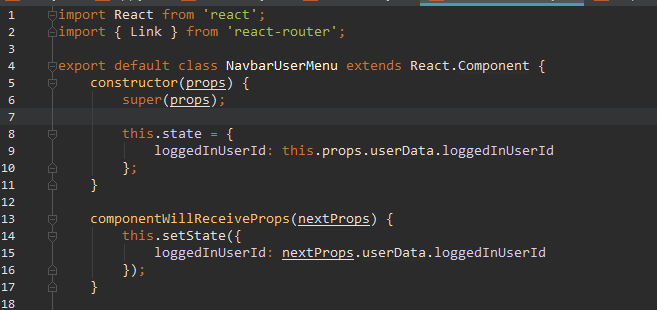


There is that **temporary login function.** Now we add logout and render:



In the render we define userDate, which we then pass to our **Navbar component.** Why **Navbar**? Because the **user menu** (**login**, **logout**, **register**, **profile**) is going to be located in the **Navbar** and it needs to have access to the **login** and **logout** functions. Speaking of **user menu**, why not implement it?

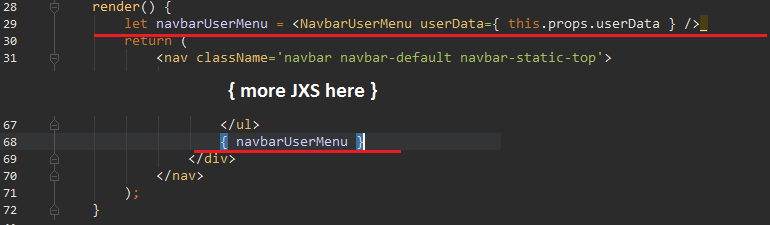
Go to /source/client/components/sub-components and create a file NavbarUserMenu.js:





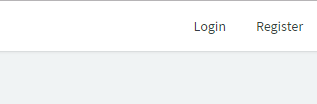
A very simple component. Most notably we can see a new member of the **react lifecycle methods –** componentWillReceiveProps. It only tigers if its **component** receives new **props**, go figure… In our case, when user logs in, it will triggeran **update of state** inour top-most parent **component –** App.js. When the **state is updated** react triggers render of App.js, which on its own turn will pass the updated loggedInUserID down to **Navbar**, and then finally down to **UserMenu component**, as next properties. Thus we catch and use these **props** in the componentWillMount(nextProps) method.

Now we need to import our new **sub-component.** Go to Navbar.js, import it at the top, then add it inside the render method:



Notice how we don’t have componentWillReceiveProps here? That’s because this **component**’s state is not dependent on its **props.** It only passes it down to UserMenu.js, thus it does not need any updated props.

If everything wet according to plan you should see our **user menu** in the top right side of **Navbar:**

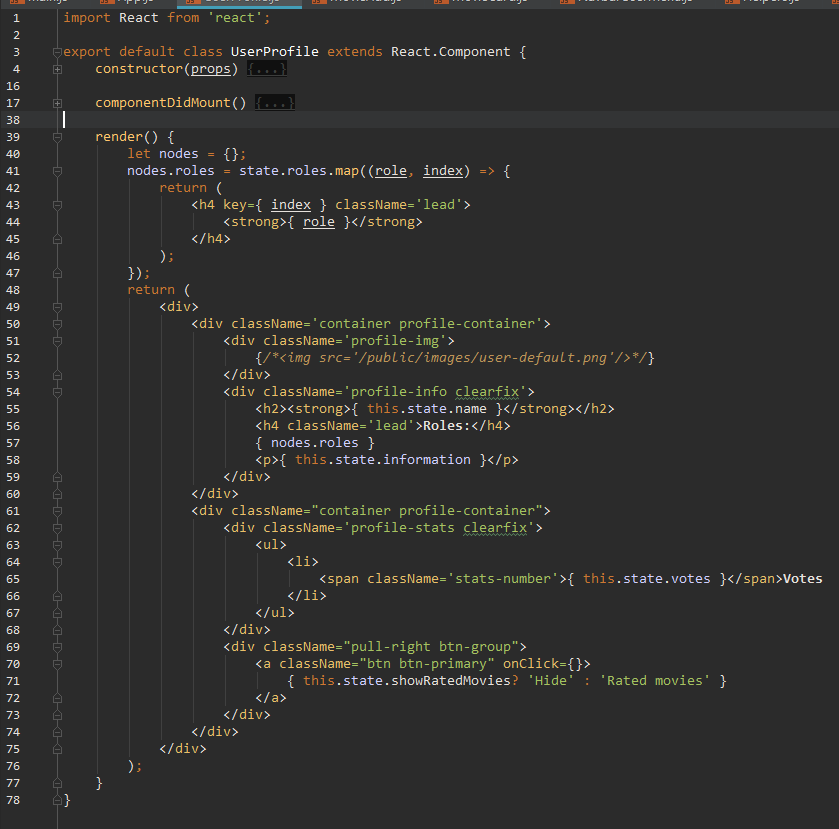
** **

You should get logged in automatically, because we call our login function in componentDidMount in App.js, and you should also see **login** and **register** buttons, if you log out. We are now ready to proceed with **User profile.**

## User profile page

Go to /source/client/components and create a file UserProfile.js:

What do we want to put in there? Well some profile information about the user, his or her roles, but also we are later on going to list all the movies the user has voted on and all the reviews he/she has written (if he/she has a role of “Critic”). For now we are going to create the view and show some basic user data:



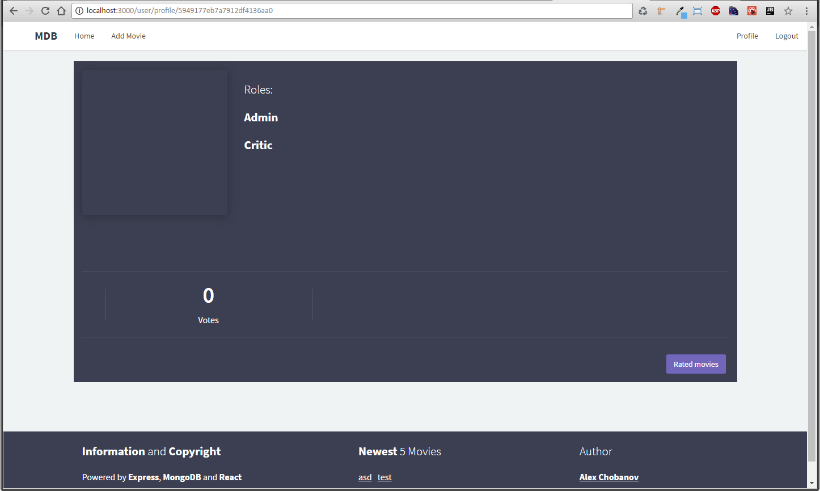
Now let’s initialize the **state** and send **ajax request** to get out data.:



Notice how we send our **request** to /api/user/${ this.props.params.userId }. This should look a bit familiar. Its equivalent in **Express** would be: req.params.userId. This means that we are getting the **user ID** as **URL parameter.** Let’s go ahead and add this **route:**

****

Now let’s refresh and click on **Profile**. We should see something like this:

Or should we? I have left one or two bugs crawling about this **component**. React does quite good job of showing errors, so this should not be hard for you.

That’s it for the day. Next time we going to do some **refactoring** – I am going to demonstrate a better, more structured way of building React apps with **Flux** architecture and a simple library [Alt](http://alt.js.org/guide/).