# 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** . 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, Events, User authentication**

### Before we begin

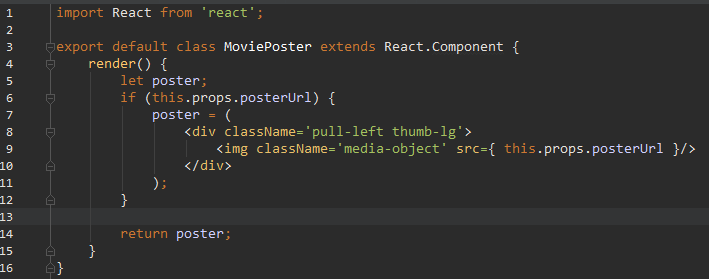
**Part 2** included a lot of refactoring, and a few mistakes in the document. I apologize for it, but what’s done is done and we should not worry ourselves with the past. For those that overcame the challenges in there, congratulations – you had the time and persistence to succeed. For the rest, just to be sure we’re on the same page, today’s **Part 3,** comes with my solution of **Part 2**. The same solution I build upon in this part.

**Important: Before starting to work on the project, open up a CMD in the project root directory and run gulp. You need to KEEP THIS BATCH OPEN, in order for gulp to automatically re-compile automatically.**

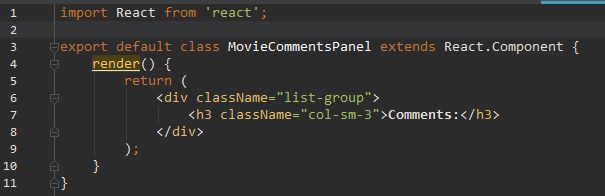
## Refactoring MovieCard (flashback)

Remember that last step from the previous lab? Congratulations to those who succeeded, but just so we are all on the same page, I will show them here

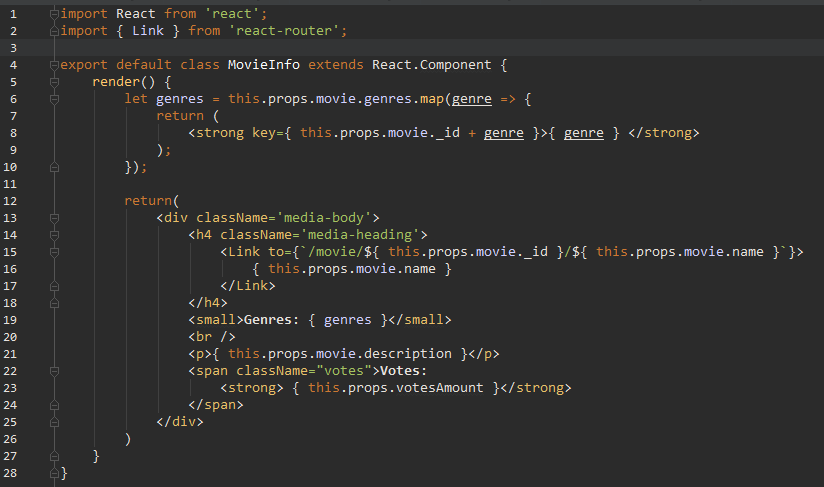
MoviePoster.js:



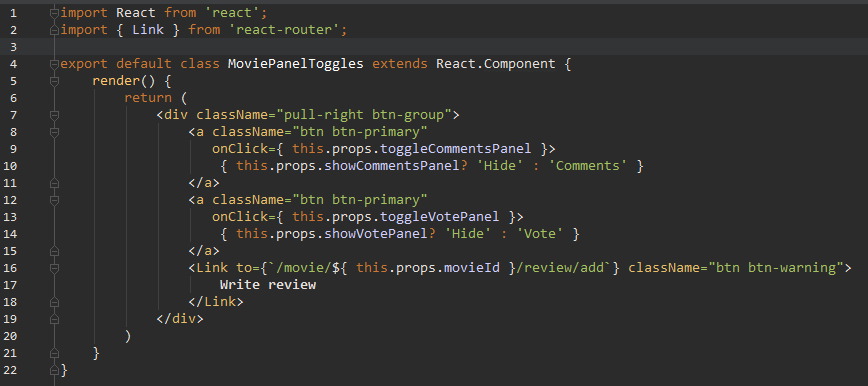
MovieCommentsPanel.js:



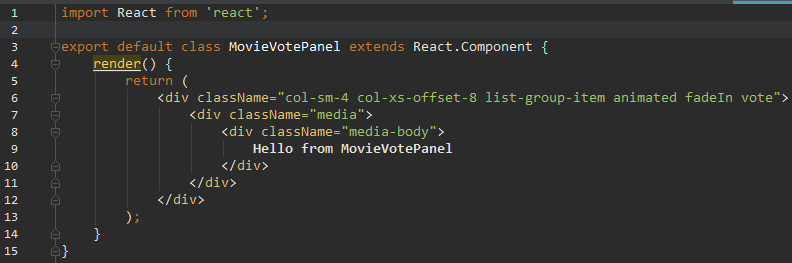
MovieInfo.js:



MoviePanelsToggle.js:

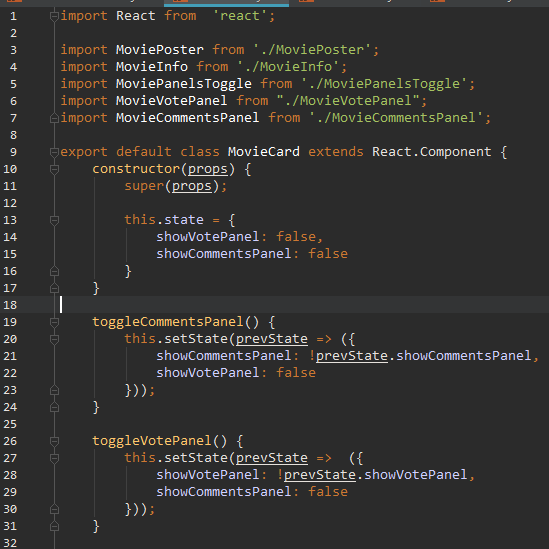


MovieVotePanel.js:



**TLDR:** We just broke the render of MovieCard.js into 5 separate **stateless** react components. While doing so we moved the conditional checks that were previously inside Helpers.js -> nodeMovieCard method inside each separate component. This code is now much more clean, reusable and natural.

Lastly we need to update MovieCard.js:



And it’s render:

render() {  
 return (  
 <div className='animated fadeIn'>  
 <div className='media movie'>  
 <span className='position pull-left'>{ this.props.index + 1 }</span>  
 <MoviePoster posterUrl={ this.props.movie.moviePosterUrl } />  
 <MovieInfo movie={ this.props.movie } />  
 <MoviePanelsToggle toggleCommentsPanel={ this.toggleCommentsPanel.bind(this) }  
 toggleVotePanel={ this.toggleVotePanel.bind(this) }  
 showCommentsPanel={ this.state.showCommentsPanel }  
 showVotePanel={ this.state.showVotePanel }  
 movieId={ this.props.movie.\_id } />  
 </div>  
 {this.state.showVotePanel? <MovieVotePanel movieId={ this.props.movie.\_id } /> : null}  
 {this.state.showCommentsPanel? <MovieCommentsPanel movieId={ this.props.movie.\_id } /> : null}  
 <div id="clear" />  
 </div>  
 );  
}

## User Register and Login

**Forms** may seem kinky and lengthly, and indeed they are. It takes some time to get them working, but that’s not, because **React**’s approach to **forms** is bad. It’s simply because **react** will not handle them instead of you. You have to manually connect a component’s state with **input fields.** Let’s finally make user **registeration** and **login.**

### Form components

Since we have number of different forms, let’s make it easy on ourselves – let’s break down a bootstrap-styled **form** into **statless** reactcomponents. Go to /source/client/components and create directory form.

Then inside of it, create Form.js, **create React** component and export it. Here’s the render:

render() {  
 return (  
 <div className='container'>  
 <div className='row flipInX animated'>  
 <div className='col-sm-8'>  
 <div className='panel panel-default'>  
 <div className='panel-heading'>{ this.props.title }</div>  
 <div className='panel-body'>  
 <form onSubmit={ this.props.handleSubmit }>  
 <div className={ `form-group ${ this.props.submitState }` }>  
 <span className={ `help-block` }>{ this.props.message }</span>  
 </div>  
 { this.props.children }  
 </form>  
 </div>  
 </div>  
 </div>  
 </div>  
 </div>  
 );  
}

Now create TextGroup.js:

render() {  
 return (  
 <div className={ 'form-group ' + this.props.validationState }>  
 <label className='control-label'>{ this.props.label }</label>  
 <input type={ this.props.type } className='form-control'  
 value={ this.props.value }  
 onChange={ this.props.handleChange } autoFocus={ this.props.autoFocus }/>  
 <span className='help-block'>{ this.props.message }</span>  
 </div>  
 );  
}

RadioGroup.js:

render() {  
 return (  
 <div className={ `form-group ${ this.props.validationState }` }>  
 <span className="help-block">{ this.props.message }</span>  
 { this.props.children }  
 </div>  
 );  
}

RadioElement.js:

render() {  
 return (  
 <div className='radio radio-inline'>  
 <input type='radio'  
 name={ this.props.groupName }  
 id={ this.props.value.toLowerCase() }  
 value={ this.props.value }  
 checked={ this.props.selectedValue === this.props.value }  
 onChange={ this.props.handleChange }/>  
 <label htmlFor={ this.props.value.toLowerCase() }>{ this.props.value }</label>  
 </div>  
 );  
}

Submit.js:

render() {  
 return <input type="submit" className={ `btn ${ this.props.type }` } value={ this.props.value } />;  
}

A lot of props going down. Most notably you will see that evey **html input field** has onChange listener attached, which fires a **callback** function, which we also receive with **props.** These **callbacks** will come from our **view component**’s **Actions**. We will implement it shortly.

### User Register

Go to /source/client/components and create UserRegister.js:

\

Now this is our top-most **component** (relative to this “page”). And as such it will operate **actions** and **store**. But first let’s take a look at the render:

render() {  
 return(  
 <Form title="Register"  
 handleSubmit={ this.handleSubmit.bind(this) }  
 submitState={ this.state.formSubmitState }  
 message={ this.state.message }>  
  
 <TextGroup type="text"  
 label="Username"  
 value={ this.state.username }  
 autoFocus= { true }  
 handleChange={ FormActions.handleUsernameChange }  
 validationState={ this.state.usernameValidationState }  
 message={ this.state.message } />  
  
 <TextGroup type="password"  
 label="Password"  
 value={ this.state.Password }  
 handleChange={ FormActions.handlePasswordChange }  
 validationState={ this.state.passwordValidationState }  
 message={ this.state.message } />  
  
 <TextGroup type="password"  
 label="Confirm Password"  
 value={ this.state.confirmPassword }  
 handleChange={ FormActions.handleConfirmedPasswordChange }  
 validationState={ this.state.passwordValidationState }  
 message={ this.state.message } />  
  
 <TextGroup type="text"  
 label="First Name"  
 handleChange={ FormActions.handleFirstNameChange }  
 value={ this.state.firstName } />  
  
 <TextGroup type="text"  
 label="Last Name"  
 handleChange={ FormActions.handleLastNameChange }  
 value={ this.state.lastName } />  
  
 <TextGroup type="number"  
 label="Age"  
 handleChange={ FormActions.handleAgeChange }  
 value={ this.state.age } />  
  
 <RadioGroup validationState={ this.state.genderValidationState }  
 message={ this.state.message }>  
 <RadioElement groupName='gender'  
 value='Male'  
 selectedValue={ this.state.gender }  
 handleChange={ FormActions.handleGenderChange } />  
  
 <RadioElement groupName='gender'  
 value='Female'  
 selectedValue={ this.state.gender }  
 handleChange={ FormActions.handleGenderChange } />  
 </RadioGroup>  
  
 <Submit type="btn-primary" value="Register" />  
 </Form>  
 );  
}

Long one. Atleast you can copy-paste develop it ☺. **TLDR**: we are imporing the **form** **components** we defined in the last section and we are passing down data as **props.** We are also passing down **actions,** as **callbacks**, which will fire upon onChange event, as we defined in our **form components**.

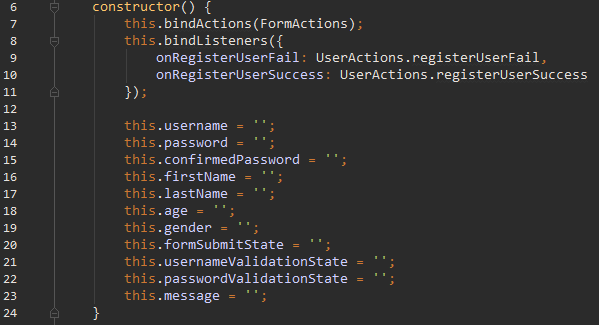
As you might guess, we are going to have multiple forms. Instead of creating separate **actions** and **stores** for each one of them, let’s create a **FormActions** and **FormStore,** which we are going to use from all of our **forms**.

Go to /source/client/actions and create FormActions.js:



All of the handleChange and validationFail actions, which we need for our **register** component.

Now inside /source/client/stores create FormStore.js:

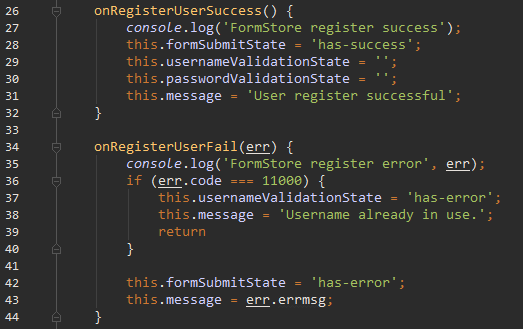


Don’t forget to import FormActions and UserActions.

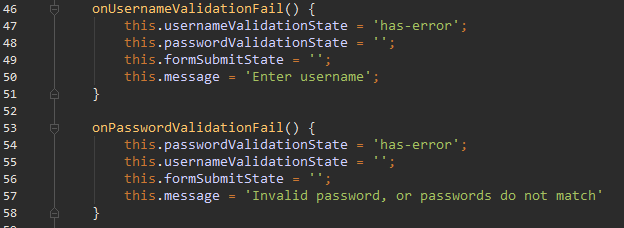
Constructor is quire lenghtly. And it is goting to get larger as we add new **forms** in our app and we need more fields for them. You may also notice we are calling another **Alt method**: bindListeners. It’s purpose is pretty self-explenatory: it attaches store **handlers** (onregisterUserFail) to specific **actions**. What **actions**? When we submit our form we want to register a new user to our database. This seems like user-related action thus we did not put it in FormActions.js. Let’s open UserActions.js and add it there:



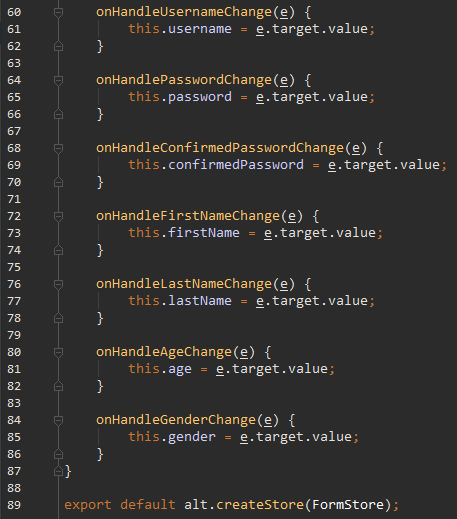
We are not going to change UserStore.js, because we already bound registerUserSuccess and registerUserFail inside FormStore.js. Why there? Because we want to show **success** or **error** message in our **form**, instead of manipulating user data. Let’s head back to FormStore.js:



Here we can see how our front-end validation will work. The required fields have a separate property, like usernameValidationState and we have message. If usernameValidationFail **action** is fired, we set usernameValidationState to ‘has-error’ and we put the **error** **message** inside this.message; Then if you look back through our **form components** you will see that these **validationStates** are actually **html selector** **classes**,which we then put on our **form-group DOM elements**. And that’s it. Remember, always reset the other validationStates or you are going to show more errors, than necessary. One thing to note is that we should always clear all validationState fields, on **successful action.** Another inresting line is err.code === 11000. That’s MongoDB code for **duplicate** **key** error. Since our usernames are unique, we want to show the user a messag, if such username already exists. Let’s go on:

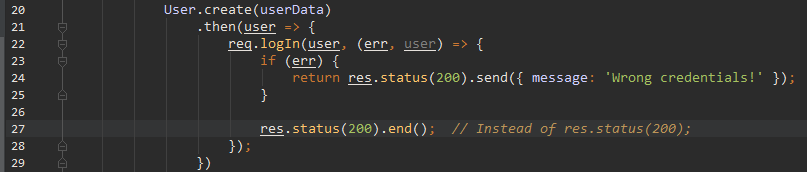


Next we have the **change handlers**:



And that it. Finaly. No, not the Lab, that was just the first **form**. It looks scarier then it actually is, but it is still a lot of work, compared to other **frameworks**. But, like it or not, that’s how **React forms** work and you have to deal with it.

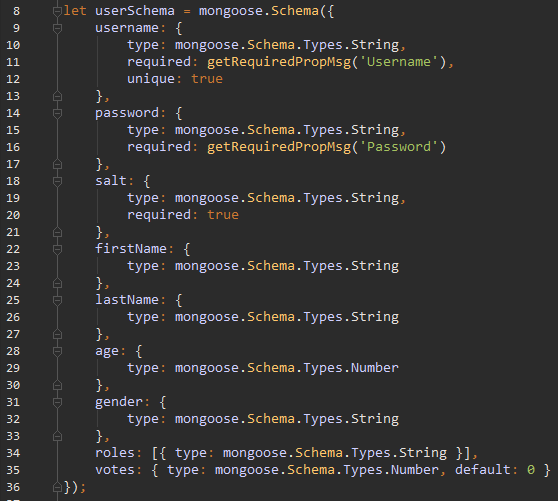
**Important: Before testing you are going to have to fix a bug in the back-end. I know, I know, you are not a back-ender, that’s why you study react. But the actuall back-ender messed up (he really did), so you are going to have to fix it – that’s life. Go to** /source/server/controllers/user.js, dive into register.post and change **line 27**, like this:



Go inside App.js and remove UserActions.login from componentDidMount:



**And another back-end blunder. As of now all input fields are required in the user model. Let’s keep the validation only on username and password, since only they are mandatory for our application’s work. Go to** /source/server/models/User.js **and change the** userSchema **like so:**



I also have a small **CSS** change to add:

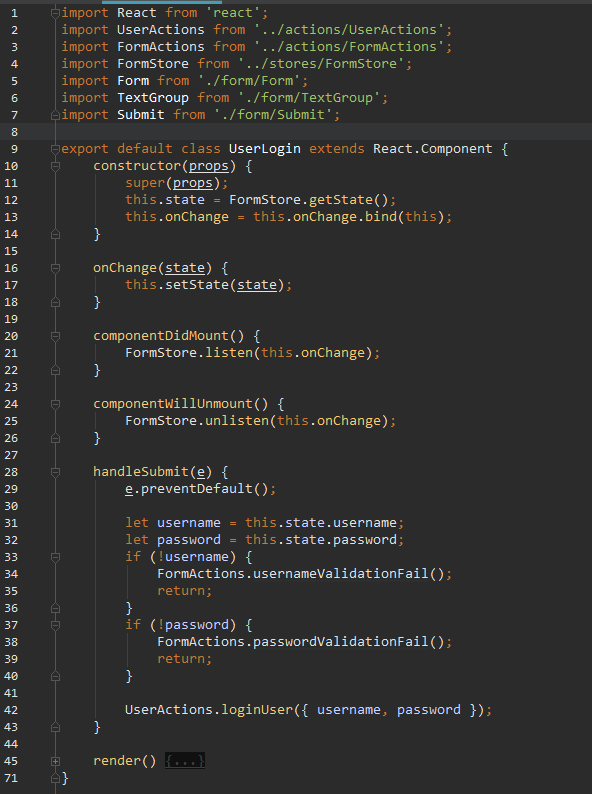
.help-block {  
 display:**none**;  
}  
.has-error .help-block,  
.has-success .help-block {  
 display: **inherit**;  
}

Go inside /source/client/styles and add this to main.less

Now add UserRegister component in /source/client/routes.js and test. If it works.

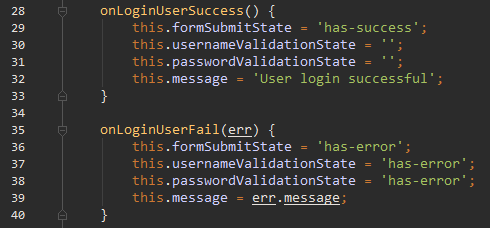
### User Login

Go to /source/client/components and create UserLogin.js:

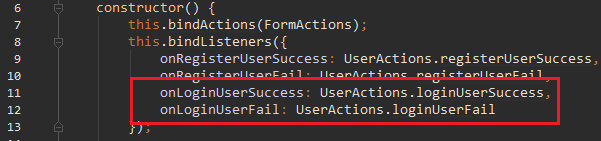


render() {  
 return(  
 <Form title="Login"  
 handleSubmit={ this.handleSubmit.bind(this) }  
 submitState={ this.state.formSubmitState }  
 message={ this.state.message }>  
  
 <TextGroup type="text"  
 value={ this.state.username }  
 label="Username"  
 handleChange={ FormActions.handleUsernameChange }  
 validationState={ this.state.usernameValidationState } />  
  
 <TextGroup type="password"  
 value={ this.state.password }  
 label="Password"  
 handleChange={ FormActions.handlePasswordChange }  
 validationState={ this.state.passwordValidationState }  
 message={ this.state.message } />  
  
 <Submit type="btn-primary"  
 value="Login" />  
  
 </Form>  
 )  
}

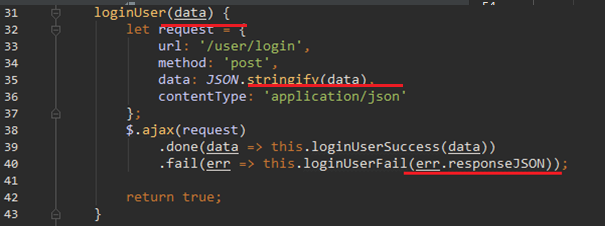
We are almost done now. Becase we have FormActions.js and FormStore.js already setup, we don’t have much work left. We only need to add success and fail **handlers** inside FormStore.js and bind them to userLoginSuccess, and loginUserFail **actions** respectively. Go to FormStore.js and add them:



And then bind them:



Also make this small change to UserActions.js:



One last thing – change the Login button to **link to** /user/login:



Now add UserLogin.js inside /source/client/routes.js. Refresh and test.

What should we do next? Well, why bother to make user registration and login if all the content is available to the “guest” user as well? You don’t really, but since we have already done it, let’s now implement **user authorization**.

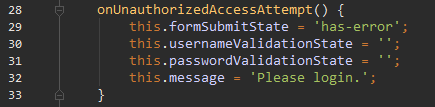
## User authorization

We will use **authorization** method, independent of the version of **react-router**. How will we do it? Remember those **container** components we mentioned? We are going to utilize such a component, which will listen UserStore to determine wheather the user is loged in, or not. If so, then it will render its child component. If user is not loged in – it will redirect to Login page. Go to /source/client/utilities and create Authorize.js:



This might seem confusing, but all we really do is expose (export) a function, authorize, which returns React component; authorize accepts single argument, representing the React component, which we want to render on this **route.** It is then rendered within the Authorization component. This way we are effectively creating a “middleware component”. More on this method [here](https://hackernoon.com/role-based-authorization-in-react-c70bb7641db4). Also, take a look at componentWillMount. If there is no logged in user (from UserStore) , we call a new **form action** – unauthorizedAccessAttempt.

**Go to** FormActions.js and create our new action. Then open FormStore.js and add the **handler**:



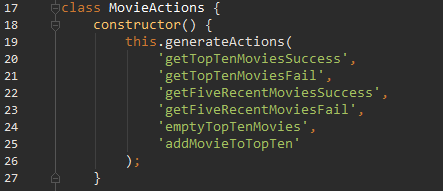
The **binding** is done by this.bindActions(FormActions), so that’s ready. The idea behind it, as you may have figured out, is to show a message when we redirect to Login page. We need one last step to get our authorization working – to add our **middleware** in routes.js, similar to how we added permissions **middleware** in **Express.** Go to /source/client/routes.js:



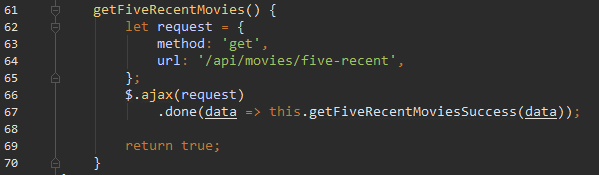
And that’s it. We call authorize, wherever we need **authorization** and pass our **view component** as argument, to be rendered by Authorization. Now refresh and test. If we got it all right, when you click on “Add Movie” (and you are not logged in) you should be redirected to Login page and get the message we defined in FormStore.js.

## Movie comments

In **Part 2** we added some toggle buttons to MovieCard.js. It’s time to use them. Let’s allow our users to write down comments on each movie. But first, we need to pass the comments down to the MovieCard component. Let’s have some good old refactoring – HomeActions.js and FooterActions.js both concern movie data. Same goes for the stores as well. Let’s merge them. Create a file MovieActions.js:



Now, getFiveRecentMovies, straight from FooterActions.js:

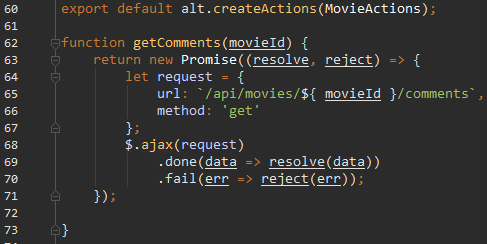


Then we have getTopTenMovies:



A lot has changed here. Do you remember how we used to have Promise.all, waiting for all requests to finish and then firing a single **action** to add all movies at once to the **store**. Well this system is no longer efficient, because now we have to add another array of promises, which will contain each movie’s comments. And later we would need a third for movie votes. That would make it overly complicated and ugly. Instead, now we are cycling through our movies and using nested requests (meaning that the second request only starts when the first is resolved) and in our last we fire addMovieToTopTen action for a single movie. It may sound like a bad performance solution, and indeed the requests will be a tad bit slower, but that’s really unoticable. At the same time, we add each movie one by one, which means that there wont be any long loading times (even if we handled significantly more movies).

Don’t forget to export with alt.createActions and to import RequesterTMDB.js. Then put the getComments function below:



Now you can safely delete HomeActions.js and FooterActions.js. Now let’s turn to the store. Go to /source/client/stores and create MovieStore.js:

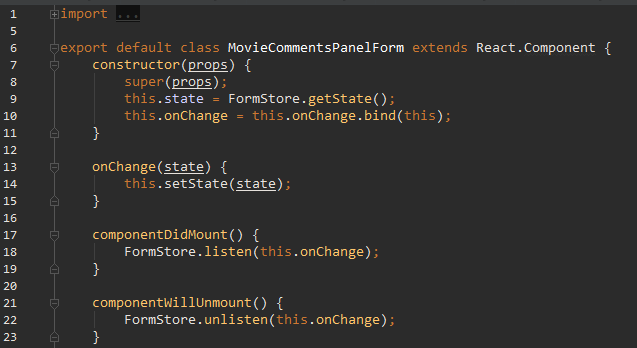


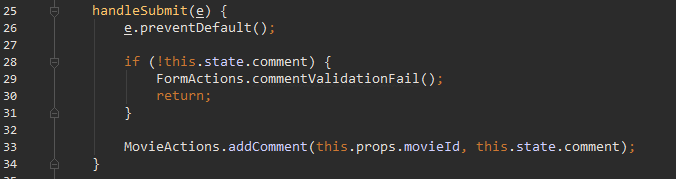
We have the new onAddMovieToTopTenand onEmptyTopTenMovies **handlers.** Delete HomeStore and FooterStore. Now fix import across the project – change them to use MovieActions and MovieStore instead.

Test it out. If you can see MovieCard components rendering, we are good. Open React dev tools and check the props of each MovieCard – you should now see comments under **props -> movies**. Let’s now make the **comments panel**. Go to /source/client/components/sub-components/MovieCommentsPanel.js and change it to:

import React from 'react';  
  
import CommentForm from './CommentsForm';  
  
export default class MovieCommentsPanel extends React.Component {  
 render() {  
 *console*.log('[MovieCommentsPanel]', this.props);  
 let comments = this.props.comments.map(comment => {  
 return (  
 <div key={comment.\_id} className="comment col-sm-9 list-group-item animated fadeIn">  
 <div className="media">  
 <div className="media-body">  
 <p>{ comment.content }</p>  
 </div>  
 </div>  
 </div>  
 );  
 });  
  
 return (  
 <div className="list-group">  
 <h3 className="col-sm-3">**Comments:**</h3>  
 { comments }  
 <div className="col-sm-6 col-xs-offset-6 list-group-item animated fadeIn">  
 <div className="media">  
 <CommentForm movieId={ this.props.movieId } />  
 </div>  
 </div>  
 </div>  
 );  
 }  
}

Now create CommentForm.js:



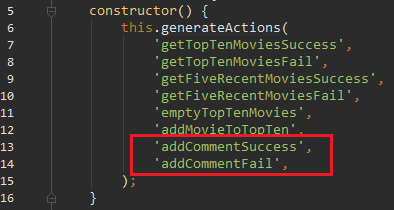


render() {  
 return (  
 <form onSubmit={ this.handleSubmit.bind(this) } >  
 <div className={ `form-group ${ this.state.commentValidationState }` }>  
 <label className="control-label" htmlFor="content">**Add comment**</label>  
 <textarea id="content"  
 className="form-control"  
 value={ this.state.comment }  
 onChange={ FormActions.handleCommentChange }  
 rows="5"/>  
 <span className={`help-block`}>{ this.state.message }</span>  
 </div>  
 <div className="form-group">  
 <input type="submit" className="btn btn-primary" value="Comment" />  
 </div>  
 </form>  
 )  
}

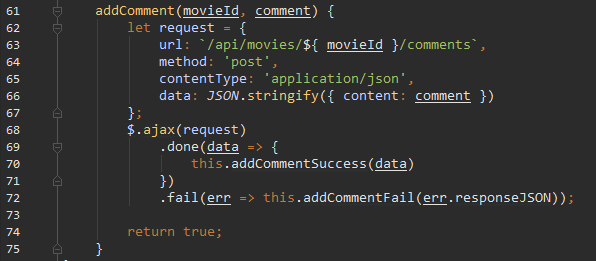
I know we had a rule, about only our **top-most** components to manage **actions** and **stores**, but that’s a form and this is cleanest way to implement form. Now let’s add the new actions to FormActions.js:



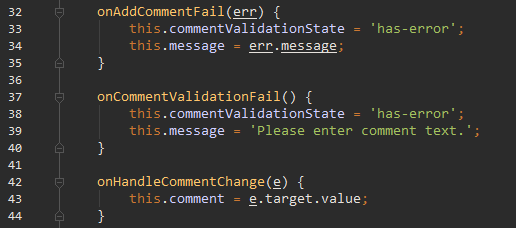
We also need to implement addComment. Go to MovieActions.js:



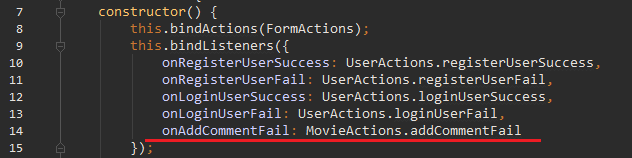
And the method itself, just bellow getFiveRecentMovies



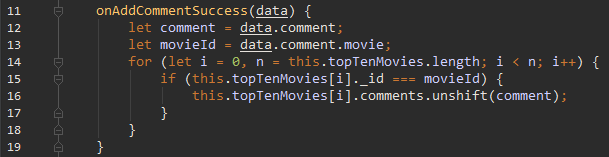
Our actions are ready. Let’s go to FormStore.js and add the corresponding handlers:



onCommentValidationFail and onHandleCommentChange are form actions, but onAddCommentFail has to be bound manually:



Now hop into MovieStore.js and add the success handler as well. Just bellow the constructor:



Last, but not least – we have to pass the comments from MovieCard to MovieCommentsPanel:



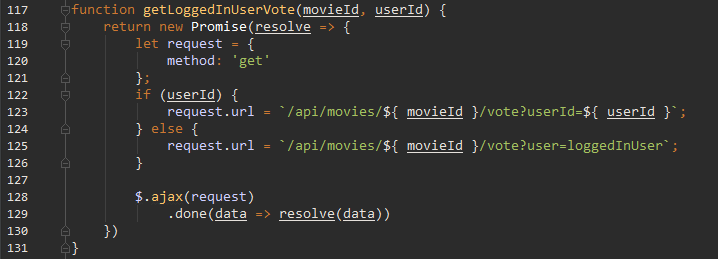
Refresh and test. If all goes well – you should be able to add comments.

## Movie Rating

Let’s repeat the same process we did for the comments. First we want to make another request in MovieActions.getTopTenMovies:



And put that function on the bottom of the file:



This function is called for each movie, to determine whether a user has voted on the movie, and if so, it retrieves his / hers vote. Now let’s move on to MovieVotePanel.js. Replace it with:



Again, we are violating **Flux’s** best practice guidelines, but that’s really the simplest design approach.

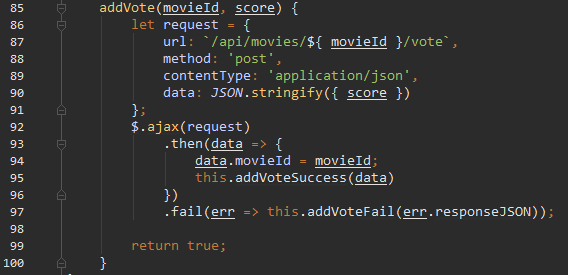
Now the render:

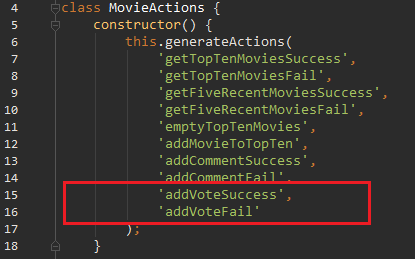
render() {  
 return (  
 <div className="col-sm-4 col-xs-offset-8 list-group-item animated fadeIn vote">  
 <div className="media">  
 <div className="media-body">  
 <div className={ `form-group ${ this.state.scoreValidationState }` }>  
 <span className="help-block">{ this.state.message }</span>  
 </div>  
 <form className="form-inline" onSubmit={ this.handleSubmit.bind(this) }>  
 <div className={ `form-group ${ this.state.scoreValidationState }` }>  
 <label className="control-label">**Score**</label>  
 <input className="form-control"  
 step="0.1"  
 type="number"  
 value={ this.state.score || this.props.loggedInUserScore }  
 onChange={ FormActions.handleScoreChange }/>  
 <input className="btn btn-primary" type="submit" value="Vote" />  
 </div>  
 </form>  
 </div>  
 </div>  
 </div>  
 );  
}

Unlike MovieCommentsPanel.js, this component represents only a single form. Thus we won’t be separating that form in standalone component. Now let’s go to FormActions.js and implement our new actions:

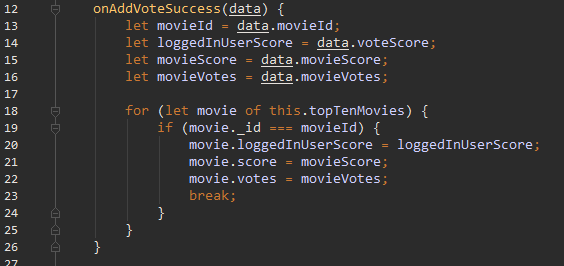


Then open MovieActions.js and create another method:

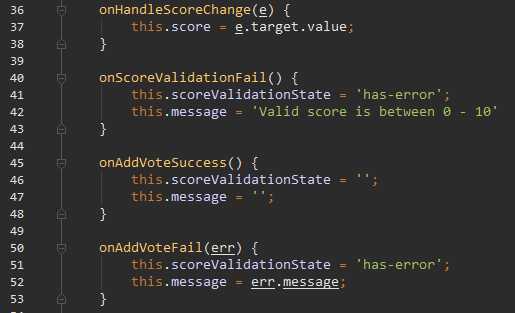


**POST ajax** to add the vote in our database. On success attach movieId to the response (data), because we will need it, to insert the comment in the concrete movie object, inside MovieStore.js. Now generate our new actions – addVoteSuccess and addVoteFail, in the constructor of MovieActions.js: 

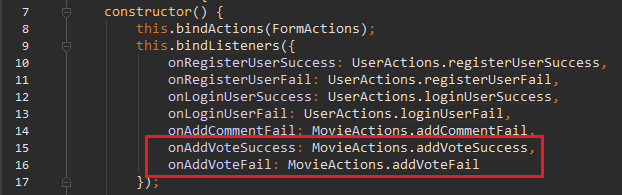
Now onto stores. Edit MovieStore.js to add our new **handler**:



Open FormStore.js and add the vote **handlers**:



Also initialize this.score, this.scoreValidationState and to bind onAddVoteSuccess, onAddVoteFail in the constructor:



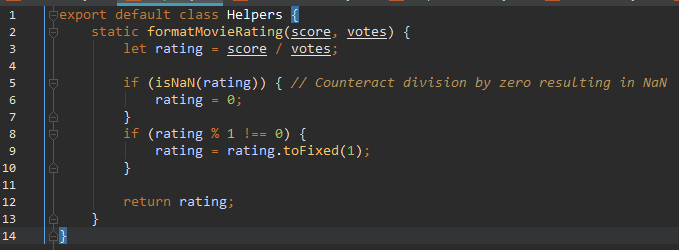
Now we need to show the rating we worked on so far – go to sub-components/MovieInfo.js chage it like this:

import React from 'react';  
import { Link } from 'react-router';

import Helpers from '../../utilities/Helpers';

export default class MovieInfo extends React.Component {  
 render() {  
 let genres = this.props.movie.genres.map(genre => {  
 return (  
 <strong key={ this.props.movie.\_id + genre }>{ genre } </strong>  
 );  
 });  
 let rating = Helpers.formatMovieRating(this.props.movie.score, this.props.movie.votes);  
  
 return(  
 <div className='media-body'>  
 <h4 className='media-heading'>  
 <Link to={`/movie/${ this.props.movie.\_id }/${ this.props.movie.name }`}>  
 { this.props.movie.name }  
 </Link>  
 </h4>  
 <small>**Genres:** { genres }</small>  
 <br />  
 <p>{ this.props.movie.description }</p>  
 <span className="votes">**Votes:** <strong> { this.props.movie.votes }</strong>  
 </span>  
 <span className='rating position pull-right'>  
 { rating }  
 <span className="badge badge-up">{ this.props.movie.loggedInUserScore }</span>  
 </span>  
 </div>  
 )  
 }  
}

Then open /source/client/utilities/Helpers.js, remove the old functions (we don’t use them since the refactoring of **Part 2**) and add this:



Now our voting should be working. But there is a trick. Because of the way the back-end works, it only allows for logged in users to vote. So before testing, you have to make sure you are logged in, or you will start seeing Internal server errors. We will fix that later.

## Authorization for VotePanel toggle button

Now we need to implement access control, but instead of restricting **URL** and redirecting, it is simply going to hide the Vote toggle button. Go to /source/client/utilities and open Authorize.js and add new class bellow the existing one:



Nearly identical logic as the above class, only difference is that we no longer redirect. Instead if user is not logged in, we simply render **null**.

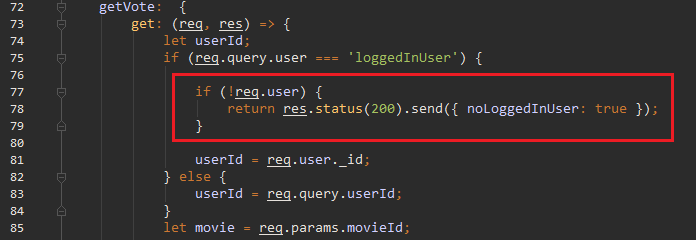
Now go to sub-components/MoviePanelsToggle.js and change it to:

import React from 'react';  
import { Link } from 'react-router';  
  
import { Concealer } from '../../utilities/Authorize';  
  
class VoteToggle extends React.Component {  
 render() {  
 return (  
 <a className="btn btn-primary"  
 onClick={ this.props.toggleVotePanel }>  
 { this.props.showVotePanel? 'Hide' : 'Vote' }  
 </a>  
 )  
 }  
}  
  
export default class MoviePanelToggles extends React.Component {  
 render() {  
 return (  
 <div className="pull-right btn-group">  
 <a className="btn btn-primary"  
 onClick={ this.props.toggleCommentsPanel }>  
 { this.props.showCommentsPanel? 'Hide' : 'Comments' }  
 </a>  
 <Concealer ChildComponent={ VoteToggle }  
 toggleVotePanel={ this.props.toggleVotePanel }  
 showVotePanel={ this.props.showVotePanel } />  
 <Link to={`/movie/${ this.props.movieId }/review/add`} className="btn btn-warning">  
 **Write review** </Link>  
 </div>  
 )  
 }  
}

It may feel wrong to have two classes in the same file, but VoteToggle is so insignificant, that it does not deseve a separate file, in my opinion. That said, however you are free to export it, if you so desire.

You can see how we replaced VoteToggle with Concealer, and passed VoteToggle and it’s depencendies as props. Nothing fancy here.

**Important: before you test you need fix yet another back-end blunder. Go to** /source/server/controllers/vote.js **and add this change:**

****

And now we are ready with Vote Authorization.

## Recap

I had other plans for this application, but this lab is already big enough for you, and for me as well. We learned a lot about **React**, we touched on various different problems and leaned how different tools can be used – **bower** and **gulp** for example. We used pure React, then refactored our code to use **Flux** architecture. **MDB** may not be the best Flux app out there, but it’s pretty good for a start.

Good luck going forward!

With pleasue**,**

**Alex**