# 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

# Part II – Flux with Alt

Today we are not going to make a lot of progress. In the end of this second part our application will have very similar functionality. The goal today is to introduce **Flux** as a project structure wich will separate our components’ **responsibilities** into 3 parts:

* **View Component** – Will hold our **JSX** and will be responsible for **mounting**, **unmounting** (all other lifecycle actions), **rendering** and **listening for user input** (since this is primarily done in **JSX**)
* **Actions** – These will be fired by **component**’s lifecycle, or **JSX events** (more on events in the next part, after the **forms&events** lecture). After completion action will call the **Store:**
* **Store** – The store holds **component**’s state and is responsible for changing that **state**, depending on **actions**.

By design **Flux** is **unidirectional**, which means the **architecture**’s flow runs in one direction:

**View Component -> Actions -> Store -> View Component**.

User interacts with our **view component**, which triggers an **action**, wich changes the **store,** which triggers re-render of our **view component.** Simple, isn’t it?

For example: In Home.js **component** we make an **ajax request** to get topTenMovies. Let’s split that single component into **component, actions, store.** The **view component** will hold our render method, the **actions** will hold our **ajax request** and our **store** will hold the **state**, and also the functions wich we pass as done() and fail().

And what about **Alt**? Well alt uses the same principles you saw in the lecture. It uses the same **dispatcher** and calls the actions as you saw on video. It does it internaly, when we call **generateActions** and **bindActions**.

But enough with the theory. Let’s get our hands dirty:

## Alt setup

First we need to install the package:

npm install –-save alt

Let’s go to /source/client and create alt config file alt.js:



And that’s it. It really is. All we need to do is to create a new instance of Alt **class**.

**Disclaimer:** I don’t claim **Flux** is the best architecture, nor that **Alt** is the best libray for it. I have chosen them, because they are really, really simple and do the job well for the purpose of this tutorial. Popular alternative to **Flux** is [**Redux**](http://redux.js.org/). Popular alternative to **Alt** is [Reflux](https://www.npmjs.com/package/reflux).

## Refactoring

This will be the most lengthly step for today. We have a lot of **copy-page-change-this-and-that** work to do. It may be a bit annoying, but hopefuly by the end you would like the result ☺. Let’s now go step by step, refactoring one component at a time and then testing **immediately**, to make sure our **components** still wok. Let’s start at the top with App.js

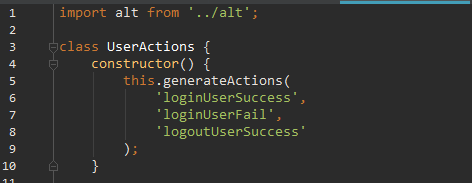
### Refactor App component

Let’s go to /source/client and create two directories: actions and stores.

#### Actions

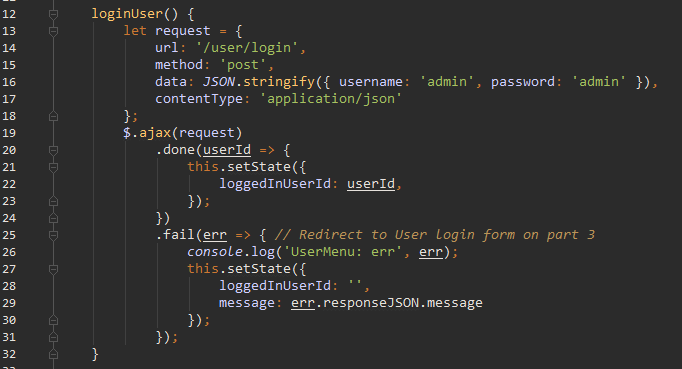
Our App.js component holds the login / logout logic. This is User related information so let’s create **UserStore** and **UserActions.**

Now inside /source/client/actions create UserActions.js:



First take note – we are importing the instance of Alt **class**, that we created in alt.js, **NOT** the library inside **node\_modules.** Then in the constructor we are generating our actions. Why 3 actions? Why **logoutUser** is one action, while **loginUser** has two actions: **success, fail**? Well unless we really messed up the user will never have a problem with **logging** **out**. But we can have **login** failiures, if invalid credentials are provided. In this case we would like the user to see an error message.

Now the **login** function – we can copy-paste it right under the **constructor**:



And the logout function:



Lastly we need to export our **actions**. Remember the **alt**, that we imported on top? Let’s use it:



Is this going to work now? What do you think? We’ll be back in a moment, but for now let’s implement our **store**

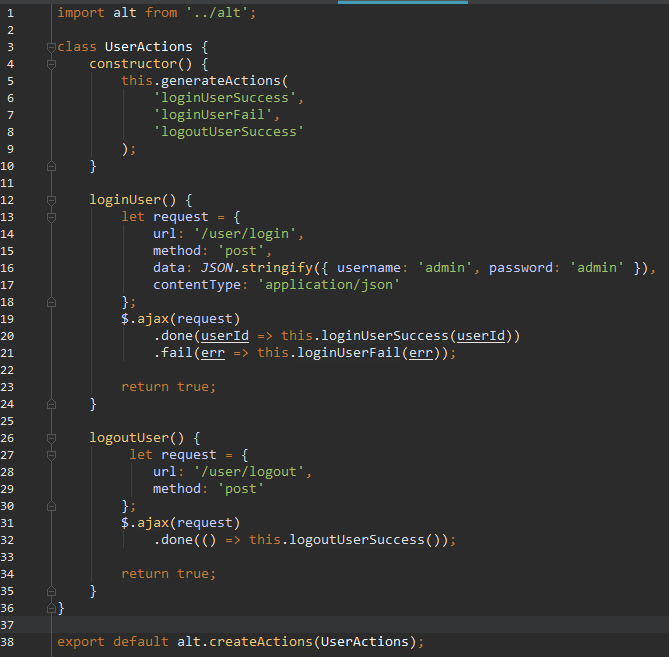
#### Store

Go to /source/client/stores and create UserStore.js



Doesn’t this remind you of something? onLoginUserSuccess, onLoginUserFail, onLogoutUser are doing the same thing we are going in the **ajax callbacks** of loginUser and logoutUser. Remember that in my short **Flux** description I wrote that **Store** will be responsible for making changes to the **component state**.

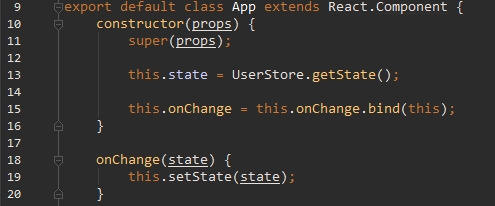
So lets now go back to UserActions.js and change it accordingly



Now instead of setState we are calling the actions, generated in the constructor.

#### **Refactoring Componen**t

We need to clear out some of the stuff from **App** component, that we put in the **actions** and **stores**. Since our **state** is inside UserStore.js, handling it from here is very easy.



But what triggers this onChange component?



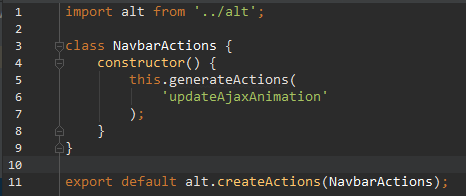
When our **component** moutns, it starts listening the **store** for change and passes onChange as **callback** function. Similar to setInterval, we want to make sure that we stop listeting after the **component** unmounts. render method stays the same. Don’t forget to import the **actions** and **store**, then let’s rebuld, refresh and make sure that our functionality is still there.

And there we have it – one **component** is already ready!

### Refactoring Navbar

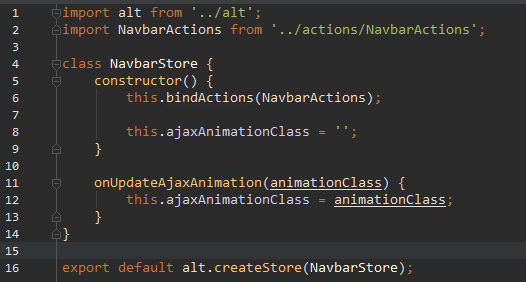
#### Actions

Create /source/client/actions/NavbarActions.js:



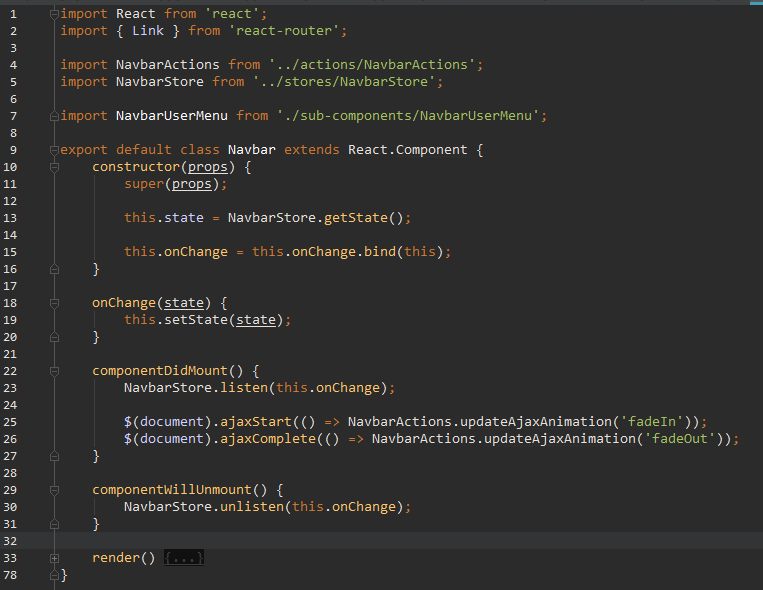
That was easy? Indeed it was. Because we only have two **ajax status** listeners and one **state field**, there really isn’t anything else we need as of now.

#### Store



**Store** isn’t overly complicate as well. Let’s move on the **component view**

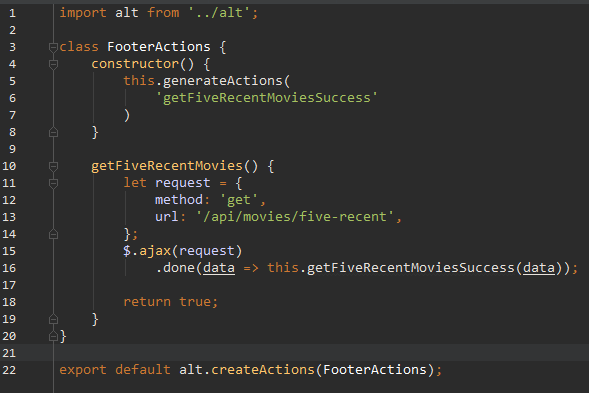
#### Component



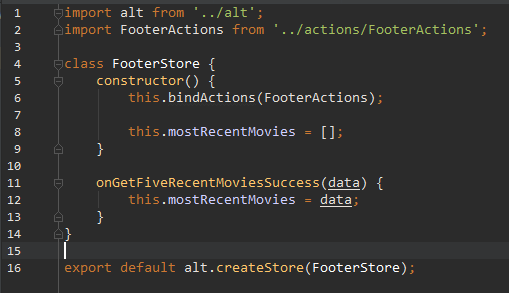
render is all the same. Again we have onChange, componentDidMount and componentWillUnmount combo. Get used to it. Don’t forget to test in the browser.

### Refactoring Footer

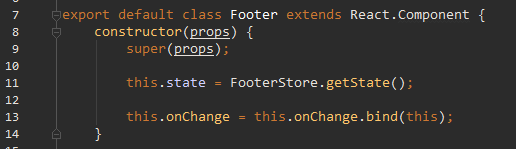
/source/client/actions/FooterActions.js

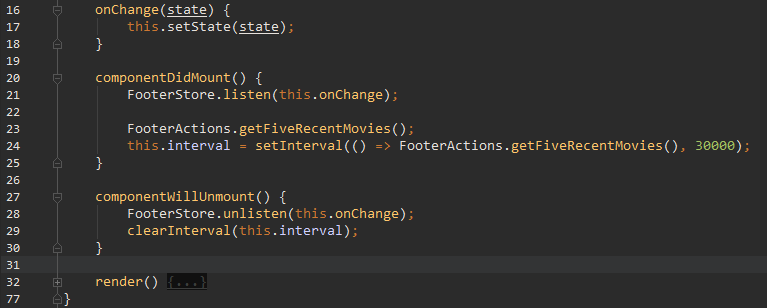


/source/client/stores/FooterStore.js



And finally /source/client/components/Footer.js





Do you notice how in **action** methods we return true;? This is to prevent an annoying deprecated warning that floods the console.

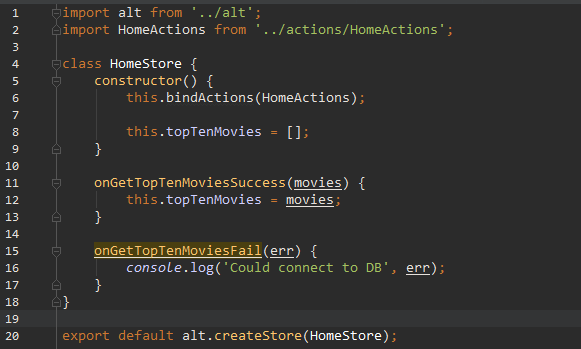
### Refactoring Home

We’re done with the **layout** components. Now lets taggle **Home.** It’s our first **view component**. In our case we need it to send an **ajax** and retrieve our movie data. Go to /source/client/actions and create /HomeActions.js:

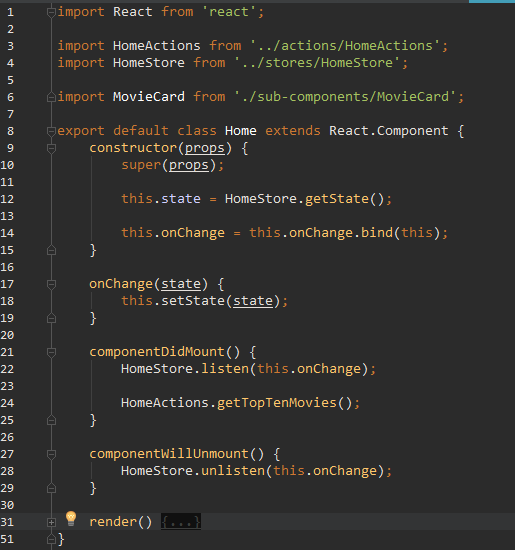


Not a lot of action – we are getting the data, then updating the **store.**

Now head to /source/client/stores and add HomeStore.js:

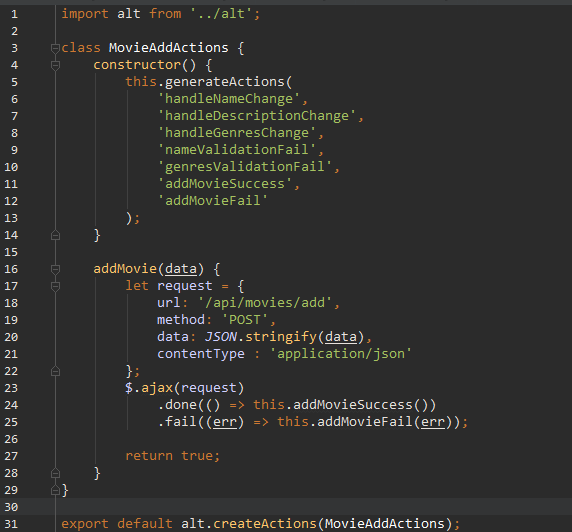


Now let’s see how our **view component** should look like. Open Home.js and chage it like this:



### Refactoring MovieAdd

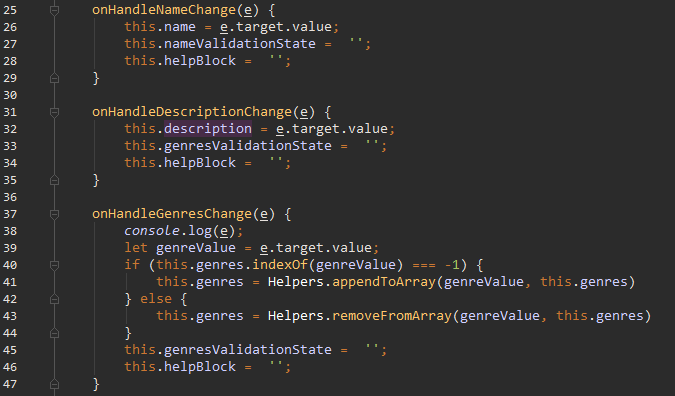
This one is more action packed. Create /source/client/actions/MovieAddActions.js:

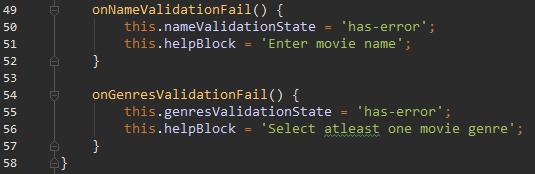


Nothing special here. **Store** is bigger this time: /source/client/stores/MovieAddStores.js

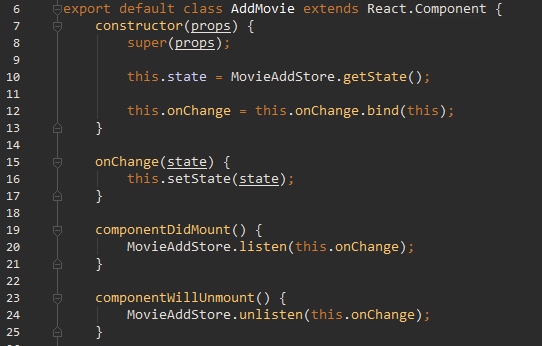


In our **state** we will hold movie information and validation classes. These will be explained more in **Part 3 – events&forms**. Then we handle addMovie **action** with onAddMovieSuccess and onAddMovieFail.

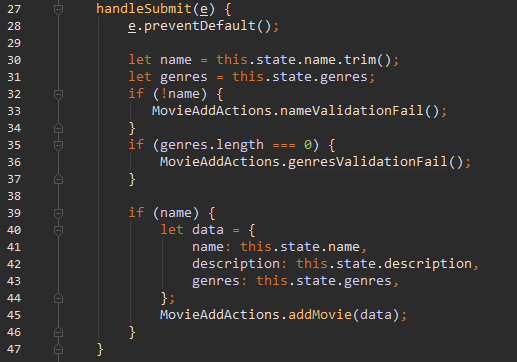


And here we set our **validation fields** in **state**.

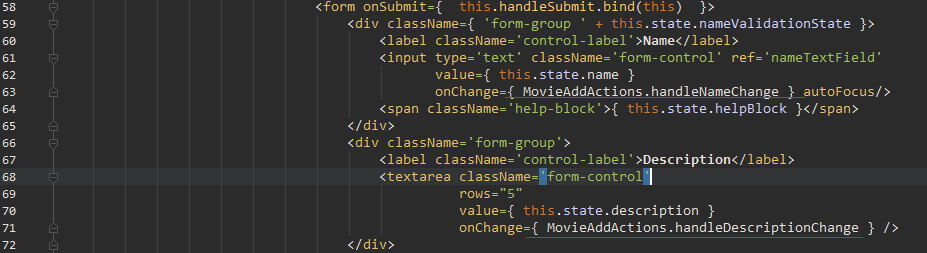
MovieAdd.js is till pretty lengthly, that’s because it’s a form, and it is responsible for submitting data:



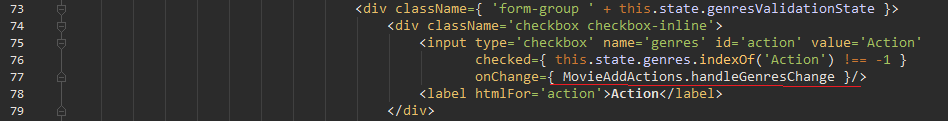
And now handleSubmit function:



We also need to update our render method to use our MovieAddActions instead:



And in **ALL FORM GROUPS** for the checkbox genres, like so:



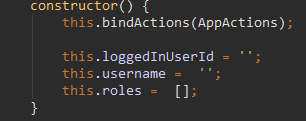
Now the same for **Horror**, **Sci**-**fi**, **Fantasy**, **Romance**, **Thriller**, **Adventure.** You may have also noticed that we removed history prop from data. Also in MovieAddStore.js inside onMovieAddSuccess we no longer redirect. Iit’s not very useful to redirect there, because it’s annoying if we want to add multiple movies.

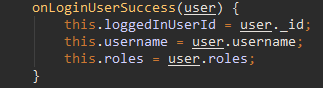
### Refactoring UserProfile

Let’s see about the **UserProfile** now. We already created UserActions.js and UserStore.js. Let’s add more data to them now. Open UserActions.js and change userId in the **ajax done** callback to data:



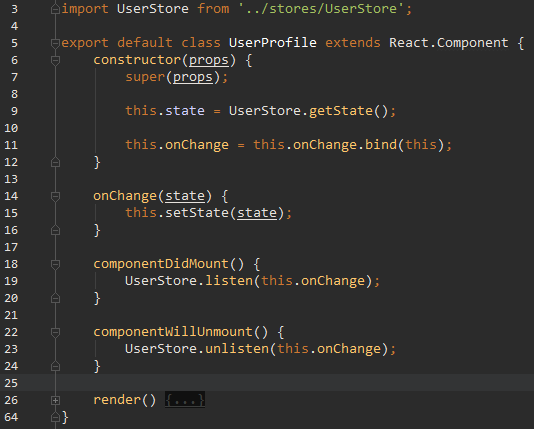
Now let’s head to UserStore.js and add the information we need in UserProfile.js





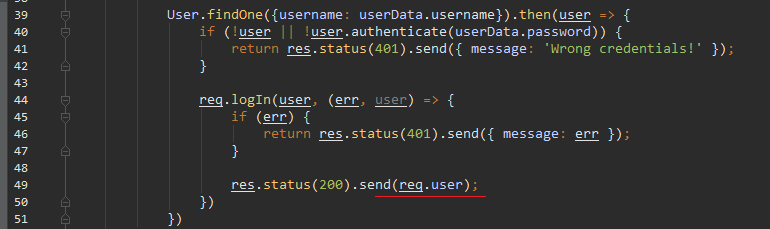
Don’t forget the empty the **store** in onLogoutUserSuccess.

Now to go UserProfile.js:



This is a good example of a significant **Flux** benefit. App.js is responsible for **User store**, because it is the proper place to put **login** / **logout**. Without **Flux** we had to either send **ajax** to the backend to get our user data. Another solution would be to think of a way to pass the data from App.js to UserProfile.js, but that would be complicated, because of the routes. Istead with **Flux**, we just “subscribe” UserProfile.js to UserStore.js and we update UserStore.js from App.js. Does it start to make sense?

**!We need a small change in the back-end. This is not a goal of this course, I know, but it really isn’t a big deal.** Go to /source/server/controllers/user.js and change login.post’s reponse on **line 49:**

****

### Refactoring MovieCard sub-component

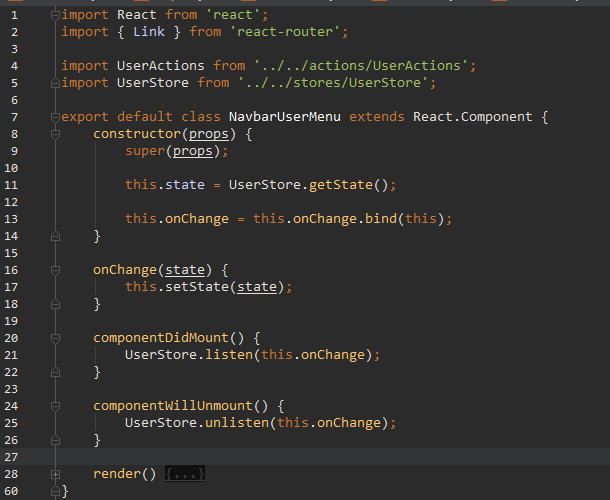
In light of recent events, let’s change MovieCard.js to:



The **JSX** should be the same, but check on it, just to be sure.

### Refactoring NavbarUserMenu

Go to NavbarUserMenu.js:



And we have done it. This was the last **component** needing **refactoring.** Now let’s do something more fun:

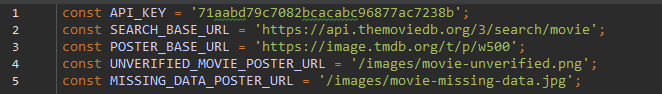
## Implement TMDB api requests

What’s [TMDB](https://www.themoviedb.org/)? Basically as the title says: A movie database. It’s primarily user-driven, which means they have a large comuniti of active users, adding new movies, rating, reviewing, commenting and so on. It’s the inspiration of our **MDB** App. But what’s even greater the devs at **TMDB** implemented very rich **API**, which we can query for movie data. More on that [here](https://www.themoviedb.org/documentation/api).

### Add poster to our movies

Our MovieCard **components** right now are quite plain. And what is a movie card, without a poster? Let’s get one. We are going to query **TMDB api** with our **movie name** and get the result.

Go to /source/client/utilities and create RequesterTMDB.js:



API\_KEY is provided by **TMDB** upon registering you application for access to their **API.** SEARCH\_BASE\_URL and POSTER\_BASE\_URL are taken from the **API docs**. UNVERIFIED\_MOVIE\_POSTER\_URL and MISSING\_DATA\_POSTER\_URL are local images, stored under /public/images.



We form our **URL** as per **API docs**. In fail callback we reject a message and error. This handles **ajax request errors** and shouldn’t happen normally. In done callback we have 3 outcomes:

1. If movie with the provided name was not found in **TMDB database** (tmdbResponse.total\_results === 0) we set posterUrl to our local image for unverified movies. These movies probably do not exists and should not be taken seriously bt the user.
2. If movie with provided name was found, but it’s poster\_path is null – We set missing data image, as **URL**
3. If move was found and poster\_path exists - set the posterUrl to the absolute path to the image, on **TMDB**’s servers.

Now we have to implement this requester in our components. Where would you put it? It may come intuitively that you put it inside MovieCard.js. But that would be **Flux antipattern**. Why so? Well, as per documentation, we should design our app in such a way that **only** the **top-most** component of every “page” handles **actions** and **stores.** If you try to implement the request in such a way that every MovieCard fires action when it moutns you are going to face **simoultaneous dispatches error.** In our case the correct approach would be to get the posters from HomeActions.js and just pass down the data. Let’s do that:

Go to /source/client/actions/HomeActions.js and change getTopTenMovies to:



This is very messy, but unfortunately **TMDB** devs did not implement multiple search query, so We have to send 10 different requests and wait for all responses, before we update our **store**. Another thing we could do is To call TMDB.getTopTenMovies for each movie and within then callback we could push each movie individually to our state. In that case we **must** check if movie exists in **store** and if so, we must update it, instead of just appending. The above is my method of choise, but you could do as you prefer. Preformance wise – it really doesn’t matter for 10 movies ☺

### Get creative

I have left out a **log** of tmdbResponse inside RequesterTMDB.js. Take a look at all that data. It’s a shame that we use only the poster. Incorporate some of that data in our app. For example: you could show the overview instead of our own description. Or you could use both.

**Fair** **warning**: you could toy arroud with votes and rating, but in **Part 3** we are going to be implementing our **own** **voting** system. Combineing both of them will take some extra work, which **will not** be described in the lab document. Procced with caution.

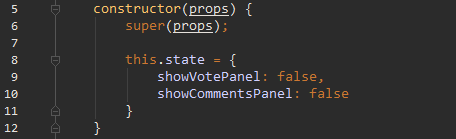
## Prepare for Pat 3 of this Lab

Since the topic of **stateful** vs **stateless**, actions and stores are on the plate – let’s prepare some of our component’s for next part’s **forms**. Primarely we will focus MovieCard.js and UserProfile.js. Let’s start by clarifying what we want to achieve. If you take a look at the pictures in part one, you will see we have buttons “Vote” and “Comments” on each **MovieCard**. When we click those buttons we want to display a panel, one for voting on a movie rating, one for adding a comment. Let’s implement those panels. For now, we won’t add the forms, just the panels themselves.

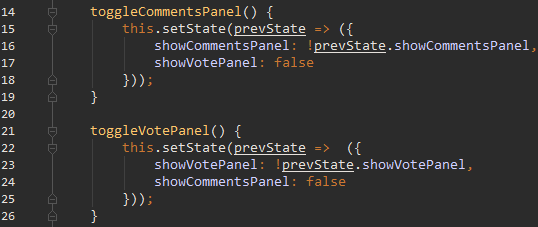
**Disclaimer**: There are people who will argue agains this approach, but I find that in this case, containing the component’s **state** within itself is not a problem, because the state does not involve any data interaction. It only holds a **Boolean**, which shows or hides a panel.

### MovieCard

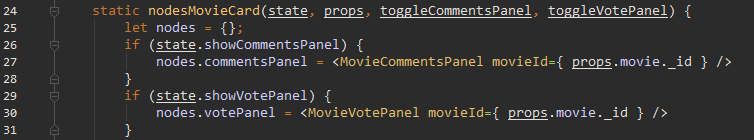
Let’s implement that state. Open MovieCard.js and change the constructor to:



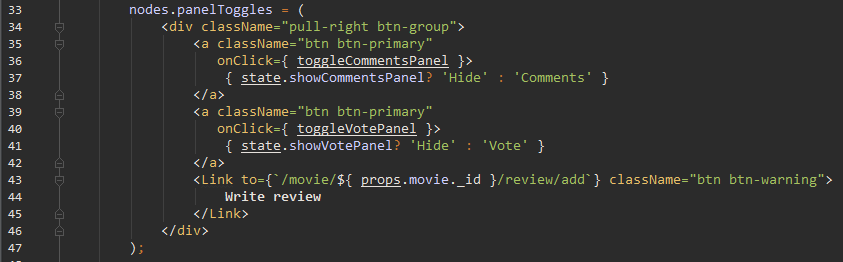
Now we need to add function which change this state:



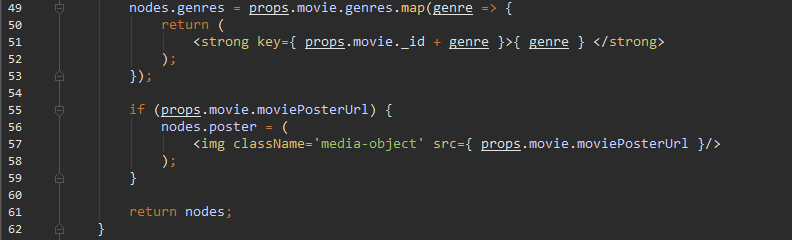
Now we need to create those buttons, and if they are clicked to display the panel. We do that in MovieCard.js’s render method, but that’s going to get ugly. So let’s create a **Helper** function to do it for us:



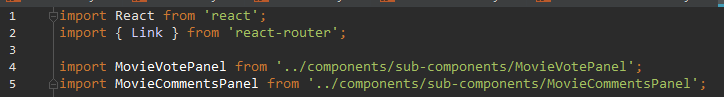
Those are our panel components. We will implement those shortly.



Those are our panel toggle buttons. They execute the **toggle** functions we created in MovieCard.js when clicked.



Since we already have a helper, lets put the **genres** and **poster** inside as well. Also don’t forget to import at the top of Helpers.js:

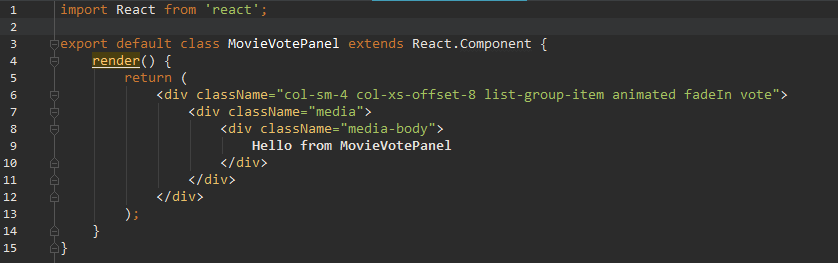


Now, lets get those nodes inside our **component:**

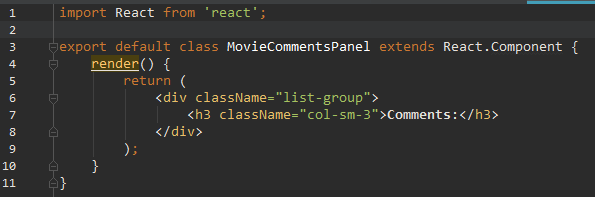
****

Define nodeson top and uncomment the sections at the end of render. And that is our MovieCard.js. Let’s now implement those panel components. Also don’t forget to **import** Helpers.js.

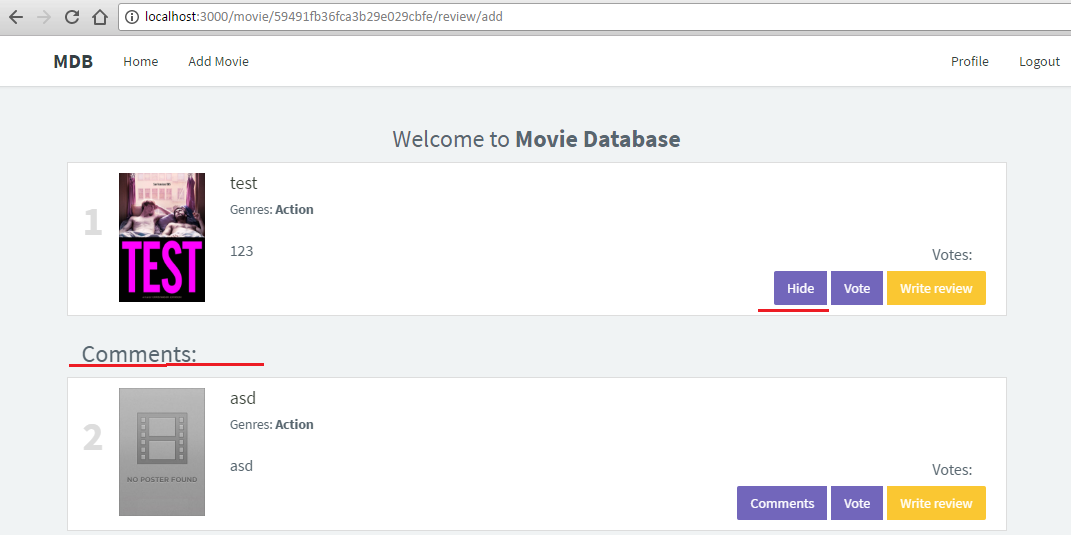
Go to /source/client/components/sub-components and create MovieVotePanel.js:



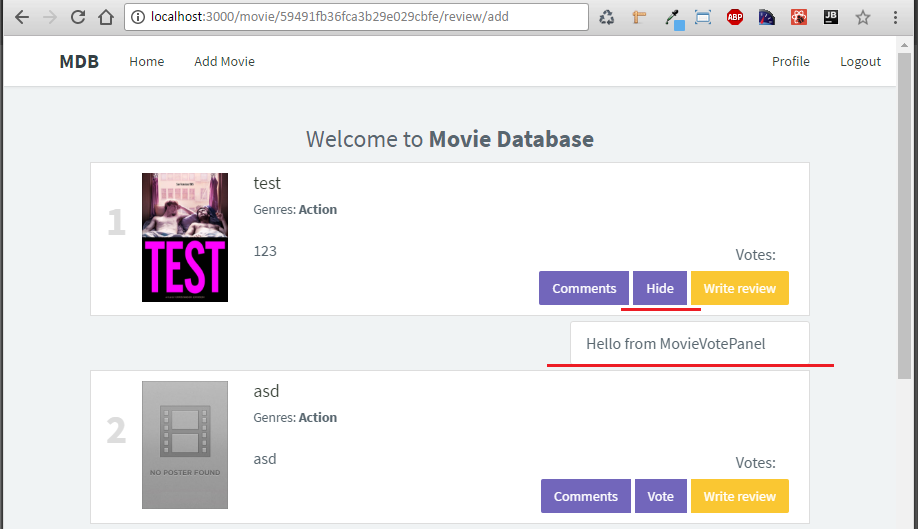
For now, were going to leave it like this. Now create MovieCommentsPanel.js:



Test the buttons. You should see:



And

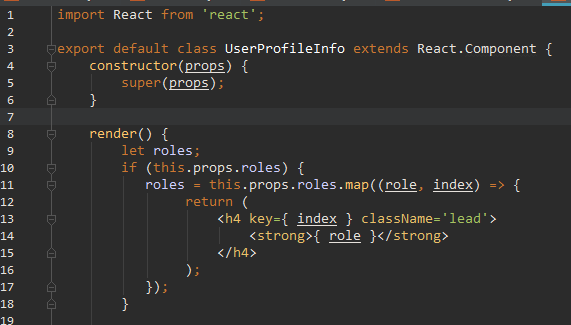


**Write Review** will not work, naturally. We’ll leave that for **Part 3**.

**Note:** Using Helpers like I did is certainly not the best thing you can do. The best practice would be all those **nodes** that we defined in Helpers.js to be separate **stateless** react components. This is sort of “stepping stone” or “training wheels”. Let’s drop the “training wheels”

### UserProfile

Since our UserProfile.js is the single component on our page, we can’t really change it the same way we changed MovieCard.js. Why? UserProfile gets its **state** from UserStore, thus it’s state is externalized. So what shall we do? We will separate UserProfie as **container** component (with similar role to Home.js) then we will create 3 **view components** as its children. These components will hold **information**, **rated by user movies** and finally **user reviews.** Go to /source/client/component/sub-components and create UserInfo.js:

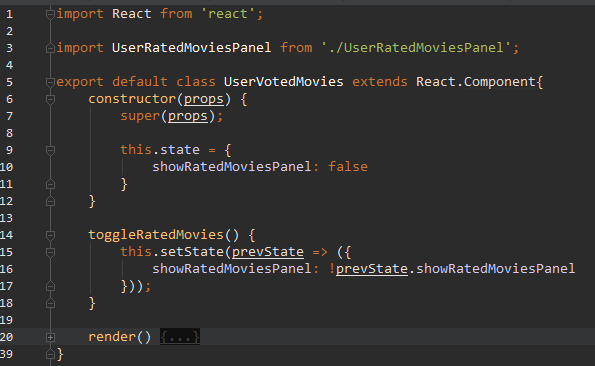


Notice how we moved **roles** and the **if** check from Helpers.js inside this **stateless** component. This is the proper way to do it. Now return:

return (  
 <div className='container profile-container'>  
 <div className='profile-img'>  
 <img src='/images/user-default.png'/>  
 </div>  
 <div className='profile-info clearfix'>  
 <h2><strong>{ this.props.name }</strong></h2>  
 <h4 className='lead'>**Roles:**</h4>  
 { roles }  
 <p>{ this.props.information }</p>  
 </div>  
 </div>  
);

Woot! No more **HTML by hand.** You are free to copy-paste!

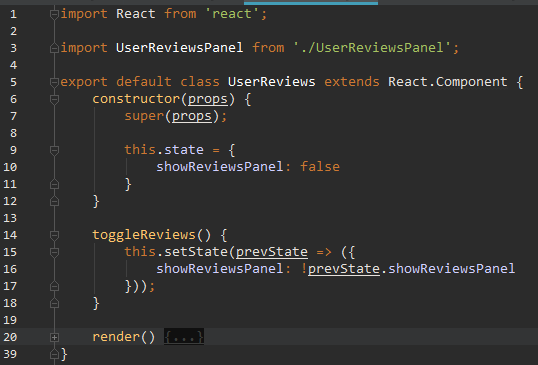
Now create UserRatedMovies.js:



And its render

render() {  
 return (  
 <div className="container profile-container">  
 <div className='profile-stats clearfix'>  
 <ul>  
 <li>  
 <span className='stats-number'>{this.props.votes? this.props.votes.length : 0 }</span>**Votes** </li>  
 </ul>  
 </div>  
 <div className="pull-right btn-group">  
 <a className="btn btn-primary" onClick={ this.toggleRatedMovies.bind(this) }>  
 { this.state.showRatedMoviesPanel? 'Hide': 'Rated Movies' }  
 </a>  
 </div>  
 { this.state.showRatedMoviesPanel? <UserRatedMoviesPanel movies={ this.props.votes } /> : null }  
 </div>  
 );  
}

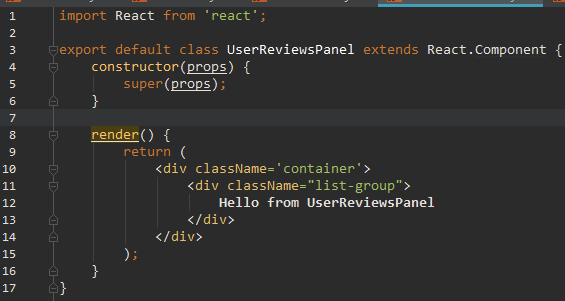
Now create UserReviews:



And it’s render:

And lastly: create UserReviewsPanel.js:

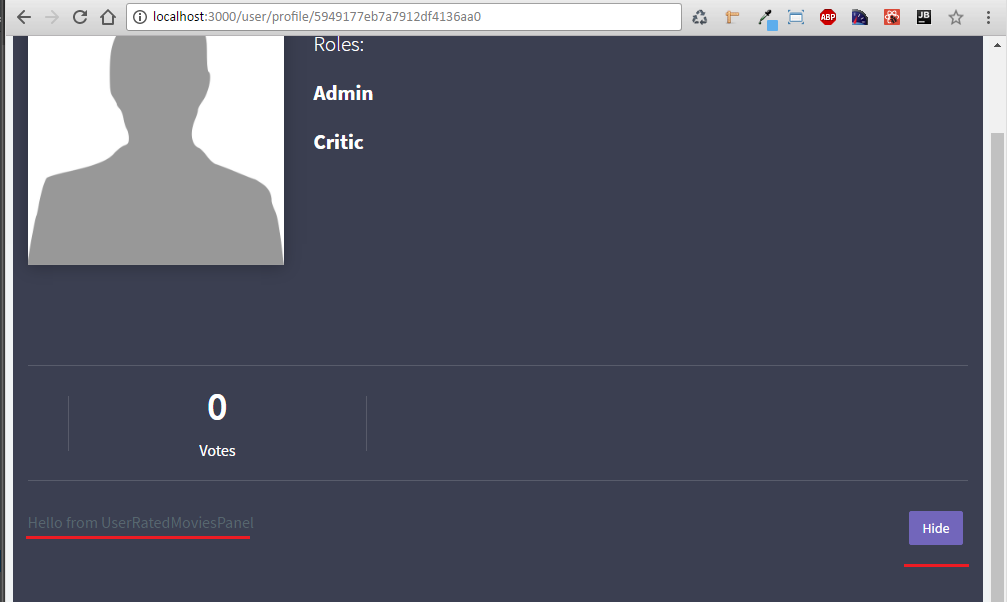
render() {  
 return(  
 <div className="container profile-container">  
 <div className='profile-stats clearfix'>  
 <ul>  
 <li>  
 <span className='stats-number'>{this.props.reviews? this.props.reviews.length : 0}</span>**Reviews** </li>  
 </ul>  
 </div>  
 <div className="pull-right btn-group">  
 <a className="btn btn-primary" onClick={ this.toggleReviews.bind(this) }>  
 { this.state.showReviewsPanel? 'Hide' : 'Reviews' }  
 </a>  
 </div>  
 { this.state.showReviewsPanel? <UserReviewsPanel reviews={ this.props.reviews } /> : null }  
 </div>  
 );  
}



And that was the last one. Refresh and click on “Profile”. You should see our panel and immediately spott a bug in our css. Lets fix that large image – to to /source/client/styles/main.less and add:

.profile-img img {  
 width: 100%;  
 height: 100%;  
}

When you click on the toggle buttons, you should see the text show and hide on the left:



Now go ahead and refactor the Helper.js for MovieCard as well. That’s it for today.