Video #1

Description:

If you know the basics of the spring framework, the next thing to do is to keep on practicing… and building apps with real life features is a great next step in your journey to becoming a developer. The main benefits of this is that it sharpens and grows your skills and helps you build your portfolio for prospective employers. In this course, we will build a prototype of a Kanban tool for personal projects using Spring boot 2.0 in the backend, ReactJS and Redux on the front end.

These are some of the cool things we will work on:

We will build our REST APIs with Spring boot for CRUD operations

We will create our front end using ReactJS and Boostrap

And will use Redux and Thunk to manage the state of our application in the front-end

We will secure our application using JWT tokens

Last but not least we will deploy our application to Heroku’s free tier. This is a great opportunity to get practical experience with two amazing technologies that are highly desirable by prospective employers.

Video #2 Tool Demo:

Features to be implemented:

1. The user will have the ability to register to create a new account with a unique username/email
2. As a user I can login to my account with an existing username and valid password and view the projects in my account, if any
3. As a user I can CRUD one or More Projects
4. As a user I can view the Project Backlog, the project backlog will contain 3 buckets with 3 possible statuses: TO\_DO, IN\_PROGRESS, and DONE
5. As a user I can CRUD Project Backlog Items (PBI)
6. As a user I can update the PBI status of a PBI item on the board

Git

1. Set up repo in folder
2. When repo is set up in folder git branch <branch>
3. Switch to that branch: git checkout <branch>
4. Do your work
5. Then: git add .
6. Then: git commit -m “message”
7. Then: git push --set-upstream origin <branch>
8. Then: git checkout master
9. Then: git merge <Branch>
10. Then: git push

Video #3 Expectations:

The purpose of this project is to provide you with practical knowledge in the technologies we are using. This project is intended as a basic starting point for you add more features on your own or as a core blue print for all sorts of CRUD applications.

Please note the requirements regarding the knowledge you need to have beforehand. Another clarification is the IDE and text editor I will be using. I’ll use IntelliJ for the Java/Spring portion and VSCode for the react/redux piece. Intellij is a great IDE but it is not free beyond 30 days. Since you already have experience with basic CRUD operations using the spring framework, you are surely familiar with an IDE. You can use whatever ide you want and if you have the minimum level required for this course, me using intellij while you use something else shouldn’t be a problem for you. If it is, this course is too advanced for you. I have seen great courses getting low ratings just because students didn’t take the time to understand the requirements, signed up for the course and then weren’t successful because they lacked the required basic knowledge. This is not in your best interest as a student because you are likely to get frustrated and give up and it is not fair to us instructors to get a bad rating because the student ignored the requirements.

#1: CRUD Projects:

Feature to be implemented: *“As a user I can CRUD one or More Projects”*

Desired Functionality: As a user I can Create, View, Update and Delete Projects

Step #1: Define the “Project” Entity

We are going to define the “Project” object by creating the following fields and Getters and setters for these fields. Please note that this is just the initial structure and will change as we define other objects and relationships:

|  |  |  |  |
| --- | --- | --- | --- |
| Field | Type | Validation | ORM |
| id | Long | @Id | @GeneratedValue strategy AUTO which means JPA selects the best strategy depending on the database |
| projectName | String | @NotBlank(message = **"Project name is required"**) |  |
| projectId | String | @NotBlank(message = **"Project nickname is required"**) @Size(min = 4, max = 5, message = **"Please use 5 characters"**) @Column(unique = **true**) |  |
| description | String | @Lob @NotBlank(message = **"Project description is required"**) |  |
| start\_date | Date |  |  |
| end\_date | Date |  |  |
| created\_at | Date | Create onCreate() method with @PrePersist |  |
| updated\_at | Date | Create onUpdate() method with @PreUpdate |  |

* We create the class an use the Annotation @Entity from javax.persistance
* Implement the fields above using the validation
* Enable the logging of SQL statements by going to the application.properties file and adding the following line: spring.jpa.show-sql=true
* Navigate to <http://localhost:8080/h2-console> to make sure the table was created

Step #2: create a CRUD repository for the entity

JPA provides a useful interface called CrudRepository for … yes you guessed it, CRUD Operations

* For this we will create a package called “repository”
* Then we need to create an interface, in this example, we will call it “ProjectRepository”
* Next, we need to annotate this class as @Repository and then extend the Spring Boot JPA CrudRepository
* Once we extend we need to specify the object that we are creating the repository for and the ID type for the object

Some of the methods that you get by extending the CrudRepository are:

We will have a lot of fun with these as the project progresses

|  |  |
| --- | --- |
| **Method** | **Description** |
| long count() | Returns the number of entities |
| Iterable<T> findAll() | Returns all items of given type |
| Optional<T> findById(ID Id) | Returns one item by id |
| void delete(T entity) | Deletes an entity |
| void deleteAll() | Deletes all entities of the repository |
| <S extends T> save(S entity) | Saves an entity |

Step# 3: Create a RESTful end point for project CRUD Operations

REST stands for Representational State Transfer which is an architectural style that defines a set of constraints to be used for creating web services. At it’s core it allows for applications to communicate over the internet using the HTTP protocol. As we will explore throughout the process of building our prototype, this allows for services to expose operations such as GET, POST, PUT/PATCH, DELETE. There are important guiding principles to create RESTful web services that are very important for us developers to understand. I encourage you to research these and get familiar. As a Java/Spring developer, it is very to have a good understanding of this. For now, the scope of our course will be to define the aforementioned operations (GET, POST, PUT/PATCH, DELETE)

* Our 1st step will be to create a package called “web”. I’ve seen other developers call this package “resource” or “resources” and a few other variations. Don’t get too hung up on naming conventions, just make sure that you follow your organization’s standards
* Now in the “web” package we are going to create a class called “ProjectController”. As you remember, the Controller’s main role is to route accordingly and in some cases return a response depending on the operation the client is requesting via a request path.
* Since we are building a REST API, we need to add the @RestController annotation to this this class

Step #3.1

* Before we get into enabling the CRUD operations, there is one more step that we need to take, and this is to create a Service Layer to interact with the repository. There are many examples and courses out there that teach people to expose the repository to the controller. The downfall of this is that if you have logic that you need to implement, for example logic to react to a user creating an account with a username that already exists, this logic would have to be implemented in the controller. This is acceptable if it is just a very simple application. The problem is that as your application grows, you will end up having spaghetti code at the controller layer, this will make it extremely hard to troubleshoot and maintain, not to mentioned that it will ultimately lead to you repeating yourself more than once. It is a good practice to keep the controller layer nice and neat.
* For this we will create a package called “service” and then we are going to create a class within that package called “ProjectService”
* After creating the Project Service, the next step is to inject the ProjectRepository in the Project service
* Now we are going to inject our service into the ProjectController

Step #3.2 POST a new project

* First we are going to create a base path for all the end points that we will be creating. For this, we need to go to the application.properties file and add the following line “**spring.data.rest.basePath**=**/api”**
* Next step is to add the @RequestMapping annotation to the ProjectController class to determine the paths for project operations.
* Next we are going create to controller route that returns a ResponseEntity of type project (we will change this in the future). A response entity allows you to control what goes into the Http response including the body and the response code. Now that we have created this method we need to make sure that we specify that it takes a @ResponseBody of type Project.the next thing to do is to add the @PostMapping annotation to it. (Explain the requestmapping option)
* Now we are going specify what this controller route should return, for this we are going to invoke the save operation from the service (which we need to create next) and return the new project with a status code of CREATED
* Now All we need to do at this stage is go to the ProjectService class, and create a “saveProject” method that returns the newly created Project object
* For this we will leverage the save method that comes out of the box with the CrudRepository Interface that we are extending with our project repository

Test Happy Path

* Open Postman and create a “Kanban Tool” collection \*\*\*Note: if you are on a network that requires Proxy settings, Postman might not work\*\*\*
* Then we are going to specify that we are sending a POST request and the path we are sending it to
* Go to “Body”, select “raw” and “JSON(application/json)
* Then we are going to fill out the responsebody of what we are sending to the server and for this, we will review the object and input the project fields needed to create a project object Check the db one more time, make sure is empty and post

Validation

* First we are going to test trying to send a POST request without any of the required fields, the objective of this test is see how the server responds. We will send them blank and then with a broken body, as you can see we are sending garbage to our database
* Now in the “ProjectController” in the “createNewProject” method signature, we are going to add the @Valid right in front of the @ResponseBody. This will enforce the constraints that we have in our object and should prevent an invalid object to be persisted to the db
* Now we are going to try to send invalid data again (Blanks and Nulls), and now two things have happened 1) we did not persist invalid data to the db 2)we got 400 bad request with details on what went wrong. Now we are getting somewhere.
* Now if we take a look at the response from the server, it is not very meaningful and somewhat difficult to extract anything useful for front end app to use and display when if the user makes mistakes like these.

Return better error messages:

* Another tool that we can use is the BindingResult API. Simply by including it in our method signature (at ProjectController => createNewProject) we can determine if there are errors in the response body and add conditionals to handle them differently
* So first we add the BindingResult API to the method signature
* Then we can use the hasErrors() function to determine custom actions if we get an invalid object from the client and return more meaningful errors.
* Now we are going to run into a little setback here, our method only returns ResponseEntities of type project so a custom error message will not compile as you can see here.
* In order to resolve this, we can use Generics. As mentioned before, it is expected that you have a good grasp on the fundamentals of Java, if you are unfamiliar with Java Generics, please read up on it and come back to this video.
* So in the return type of the createNewProject we are going to replace the project by a generic type
* Now we can try and test the api returning a custom error message. For this quick test we are going to use this simple code, restart the server, try the validation tests

**if**(result.hasErrors()){  
 **return new** ResponseEntity<String>(**"Invalid Project info"**, HttpStatus.***BAD\_REQUEST***);   
}

* Restart server, test that the db is empty, then try happy path, and then validation tests
* Now we want to take this one step further, we want to be able to tell the user exactly what fields we are missing or passing wrong data to
* Lets comment out our conditional statement, and go back to review the response we get from the server.
* For our application we are only interested in displaying the field and the defaultMessage, so how can we go about that?
* A good way to go about this is extracting the field error list from the bindingresult api, let’s test that out
* Let’s uncomment some of the code and explore it together
* Since the server clearly returns an object that contains information on the errors, it is likely that the api has a method to access this data. A good hint is that result.hasErrors() is obviously a Boolean that determines if there are errors or not. By that logic, we could explore if there is a method that provides a list of such fields. Lets check in the IDE
* As you can see, it has a method.getFieldErrors, with two implementations, lets grab the one with no parameters. We are also getting the method returning a list due to the fact that we have more than 1 field that can be invalid at a given time. So now let’s change our condition to return the list of field errors
* **if**(result.hasErrors()){  
    
   **return new** ResponseEntity<List<FieldError>>(result.getFieldErrors(), HttpStatus.***BAD\_REQUEST***);  
  }

and as you can see, we get the same thing as before, but it is not a 500, it is a more meaningful server response.

* Now in this scenario, let’s say we have a quick conversation with our front-end developer and she tells us: I would like a response that is much simpler, basically {field: message} , rather than having to deal with that massive response. An example would be:

{

"projectName": "Project name is required"

}

* So if we look at the response, we would have to map those two values in order to return such custom response. So you guessed it, HashMap data structure to the rescue!
* First lets create our error map
* Then let us traverse the .getFieldErrors list and extract what we need, but lets review it first. Basically we need the field and the defaultMessage.
* Lets now loop through the getFieldErrors list, remember, everything in this list is of type Field Error.
* In the loop, we can add the errors found to our errorMap, based on the requirement we got, we will use the field as the Key and the message as the value.
* Now that we have properly built our map, lets update the return statement in our condition

**return new** ResponseEntity<Map<String,String>>(errorMap, HttpStatus.***BAD\_REQUEST***);

* Lets restart the server and check it out
* Before we move forward, let’s do a bit of thinking about the future of our tool. In the future we will have more objects that will require validation, and we will probably want to return errors the same way. If we leave that logic in our controller, not only are will unnecessarily cluttering it with code but we will end up, REPEATING OURSELVES or engaging in copy/paste programing. There are many ways to go about this, and the one we will use is turning that chunk of code into a validation service
* Lets go to our services package and create a MapValidationErrorsService class
* Annotate it as @Service
* Then create a method called MapValidationErrorsService that returns a generic ResponseEntity and that takes a BindingResult as a parameter
* Then cut the error condition from the controller and paste it in the MapValidationErrorsService method, make sure it returns null if the condition is not met.
* Now let’s inject this service into our controller
* Now lets pass the results to our service, we will do this by declaring an error map ResponseEntity in the controller method.
* Then we will create a short conditional **if**(errorMap != **null**) **return** errorMap; remember since this is a very simple conditional, we don’t need the brackets and we can have this functionality in 1 line of code.
* Now, not only is this much nicer code, but also we will be able to reuse for other entities in need of validation.
* Lets test it, everything should work as before
* And there you have it, he have completed the create project operation with validation.

Handle Exceptions (duplicate

* As you remember, in our project object, we have a projectID string, that can only be 4 to 5 characters long and that has to be Unique. In this design, the purpose is for us to have a more meaningful alternative to interact with a specific project object other than the id in the db.
* To enforce this, we used the @Column(unique = **true**) annotation on the projectId field. And as you can see it works, The downside is that although it works, doesn’t allow for a more meaningful server response. Since the @Column annotation is not part of the validation.constraints, we don’t get this out of our MapValidationService. It is nevertheless an exception that could be handled better!
* So we are going to create a custom exception for this. This will be a good practice to create custom exceptions.
* The 1st thing we are going to do is create an “exceptions” package in our project.
* Given the requirement we got from the front end developer in regard to the server response in case of exceptions, we will create a “ProjectIdExceptionResponse” object
* We are going to give this class 1 attribute “projectId”, create a constructor and getters and setters. Remember, the response from the server is a json object, so this allows for us to structure it that way.
* Next step is for us to create the exception itself, so in the “exceptions” package create a class called “ProjectIdAlreadyExistException” class
* This class extends RuntimeException, we will also determine the response status if this exception is thrown which in this case would be a 400 bad request. For this we will use the @ResponseStatus annotation
* Final step is to create a constructor for the custom message we expect this exception to throw
* Next step now is to setup a global exception handling for this we are going to create a class in the exception package called “CustomizedResponseEntityExceptionHandler”
* The objective of this class is to apply our custom exception across the whole application, not just to an individual controller. For this we need to add 2 annotations, we also need this class to extend “ResponseEntityExceptionHandler” the reason we are extending this class is that it provides an @ExceptionHandler method for handling internal Spring MVC exceptions.
* Now we are going to implement a public final method that returns a ResponseEntity of type object that takes two parameters: ProjectIdAlreadyExistException, and the WebRequest
* This method instantiates a new ProjectIdExceptionResponse that takes the message and returns the ResponseEntity containing the message in the format we want to see.
* Now the final step is to use a try an catch block to handle this exception better. Since this is an exception that happens at persistence, we will update the saveProject Method in ProjectService to handle this in a more elegant way than a Internal Server error
* So. In the try block, we are going to isolate the operation of setting the ProjectID which is the moment in which the exception will be thrown as it is expected to be unique
* And in the catch block we will throw our custom exception.
* Lots of wiring up so time to reboot the server and try again
* Lets create a project, and let’s try to create it again with the same project id. Good stuff, it works!

Now we are done the the Create Operation, including validation

Step #3.3 GET a project by projectId

Happy Path

* This one is going to be fun, we are going to now retrieve projects by their unique project id, this will help us in the future to get project information and of course update the project
* First we will happy path this, 1st step I want to take is to enable this type of search in the repository.
* 1st, let’s review what we have out of the box, we have all these methods BUT we want to be able to find by the attribute productID
* JPA as a nice way of helping us with this, let me show you: first you specify what it is that you are returning, in this case it is a Project object, then you specify how it is that you need to find such object, so you type “findBy” and this will give you a list of options including, the fields of the object, so if we start typing projectId, you’ll notice that it is an option. Then we specify the parameter which in this case is a String.
* Now that we have this on the repository, let’s enable work on our service to expose this method. Remember we are just coding the happy path right now ☺
* Now we are ready to update our end point to return a project by ID, in the ProjectController class, we are going to create a getProjectById method which we are going to add the @GetMapping annotation to
* Now we need to specify that the id will be passed in the path, which is the parameter we need for our service to retrieve it from the db.
* For this, we will specify it in the @GetMapping annotation path
* Then we need to include the @PathVariable in the method signature, remember, we are passing a string.
* Now lets use are newly created method to retrieve the project
* Last but not least, make sure that we return the project that exists.

Project that doesn’t exist

* Now let’s see what happens when a project doesn’t exist. Nothing now let’s think for a second: we are only using this api to return hardcoded code in the UI, we are not doing a fuzzy search HOWEVER, we still want the API to be more gracious and tell us that no project was found. Since it is not an exception perse, we’ll take the easy route, put a condition in the controller and return a descriptive message and a 400 bad request status.
* Now the downside is that our front end prefers a json response instead of text, so one way to tackle this is to create another custom exception response just like we did the 1st one, actually this is a nice challenge for you tackle on your own, what we are looking for here is a clean json response { "projectNotFound": "Project not found" }
* Once we create the exception, then we will go to the ProjectService and update the findByProjectId method to throw the response if the project object returns null

Step #3.3 GET all projects:

* The next step is to have an API that retrieve all the projects created in the system. There are two approaches: 1) all projects, period. 2) all projects by account. Since we do not have security just yet, we will start with find all.
* The first thing we want to do is to access a list of all project from the database, So we will go to our service and explore what is out of the box with the CrudRepository Interface
* Let’s create a method that returns a list of projects, this method takes no parameters, HOWEVER, in the future, we parameter might be the username so we can only retrieve projects associated with said user.
* Lets explore what is available out of the box. As you can see, I made the mistake of making the method of type List, however, what it returns out of the box is an Iterable data structure
* Now lets update the controller, We are going to use the @GetMapping annotation, and we will give it the path “/all”
* Lets create the public method getProjects() that returns the list we defined in the service.
* Now let’s test it, let us create two projects and then see what we get back.
* Sweet, I am not setting this up with a message if no projects are found, we can totally do this at the front end.

Step #3.4 PATH existing project:

* We are done with the CR of the CRUD so it is time for us to update
* If you pass the db id when using the post operation, if the item exists, JPA will update it, now what if it doesn’t, then it will just take it as a new item, run it by the validations and ignore the bogus id you give it, and that is it!!. On the front end, when we update, we just need to make sure we pass the id in the request we are sending form the request form.

Step #3.5 DELETE existing project:

* This one should be a PIECE OF CAKE FOR Y’ALL, why don’t you give this one a shot and come back and watch my solution hint, make sure that you delete by projected
* \*\*\*Clean up\*\*\* you don’t have to do this, BUT I really want to remain consistent, since the projectId is the main identifier in our application and not the id in the db, I like to make that one not updatable. All we have to do is go to the @Column and add , updatable = false
* Now that this attribute is not updatable, then we are going to work on the service layer. Make sure we use the project not found exception
* Now time to go to the controller layer, we will use the @DeleteMapping that takes a projectId, please note that this is very similar to getting the project by ID
* Now let’s create a few projects
* Then let’s try deleting a project that doesn’t exist
* And finally let’s get rid of a few projects

Step 4 Basic REACT APP.

* We know why we are here so let’s start having some react fun!!!!
* What is react?: React is a client side library which means that it react apps run on the users machine browser.
* This is different from say JSP or Thymeleaf pages which run on the server. And if you fulfill the minimum knowledge required for this course, I’m sure you are familiar with those java technologies and have perhaps created apps with those before
* React was developed by facebook and it’s main purpose is to allow us to build user interfaces by leveraging the concept of “Components” which gives us a nice layer of organization which in the end enables reusability and increases your productivity
* Building apps with react is much simpler than building with vanilla js and in my personal experience, the learning curve to react is not as steep as Angular’s .. given, of course, that you have a decent grasp on the javascript language, more specifically ES6 syntax. Remember Just as Spring is to Java, React, Angular, Vue, etc are to Javascript, if you are going to work with a framework or library, you need to know the language syntax.

Step #4.1 set up environment

* Text editor: if you have a preferred text editor may continue to use that, however I strongly recommend, specially if this is your 1st rodeo with react, that you stick to Visual Studio code. VScode is a very fast and intuitive editor and it has wonderful extensions that greatly support react development. We are not going to go through the installation, it is very simple and they have great documentation the site. Once again, Although this course is novice friendly, it is not meant for absolute beginners, so there will be things, like setting up your environment, that I expect you to be able to figure out on your own
* You also need to install nodejs in your environment which is a javascript runtime and it installs npm which stands for node package manager. We will use this one heavily when we are running are app and installing packages that we will need while developing our react app.
* In terms of browser, I recommend that you use Chrome and that you install both the React Developer tools and ReduxDevTools, this one will be extremely useful when we get into redux as it shows you your application state and actions.
* Setting up vscode, (set it up as per the MERN course)
  + Text editor format on save
  + Emmet
  + Emmet: include languages
* Can use filter to filter the state of a PBI
* Next thing we are going to use npm to install the create-react-app packagnpe, so for this we will run the npm install -g create-react-app command which installs it globaly, please note that you might need to sudo if you are on mac or linux, I have it installed so I will not go through that process
* Last step is to create the app, which we are going to call KanbanTool. I’m going to create a folder on my desktop for this, but you can create it anywhere you’d like create-react-app KanbanTool
* As you can see, this generates a react app with some out of the box packages, so if everything went well, we should be able to run it by typing npm start
* As you can see we have it running at port 3000 with some basic styling which we will discard as we build our app
* You can stop the server by pressing ctrl-c at the terminal (this is for both mac or windows)
* Another important command is npm build, npm build created a folder that contains static assets that you would want to deploy as it compiles your entire app in format that the browser can read it. If you open the js file, you’ll see that although it is code, it is mostly unreadable. Remember you are not supposed to do anything in this build folder, this is just what you are going to deploy.
* At all times you can delete this file and regenerated with the npm run build command we just used

Step #4.2 Review the project generated by create-react-app

* Package.json: is a file that holds various metadata relevant to the project. Some of the main uses for this file is to declare dependencies and scripts required to the project. As we progress we will be installing other libraries, to add additional features. These libraries will be listed as dependencies. Although it is not an apples to apples comparison, in a sense you could think of this file as your pom.xml file in spring and npm being like maven. Again, this is just a very very high-level comparison based on something that is familiar to us Java enthusiasts
* Now let’s look at the public folder, in here you will find the index.html file which is the main entry point to the application. The most important part of this file is the div with the id of root. This is where the main app component is rendered. As we create components, they will all be a part of the main app component which in turn is a part of the root component. Let’s do a quick change and call this Kanban Tool
* Now let’s explore how the main app is rendered. If we go to src / index.js you can see that we are importing App from App.js. Here we are using the render function to load the <App /> component into the div with id=root
* App.js file: is the main component which is what’s being rendered in that div with id of root. As you can see, this is a class component that extends Component from react. There are other components that are merely functional, we will discuss those as we move along. This is a very common structure for a component, it has the render function that returns jsx which is all that is rendered in our browser. Btw now that the server is running, you can make changes to what’s being rendered and it will automatically refresh
* Now let’s do a bit of clean up, basically get rid of some stuff that we don’t need: let’s start with the content in the App component, we are just leaving the div = “App”, also let’s get rid of the logo, as cool as it is, we don’t really need it, so make sure you clean the import so your app doesn’t break and then delete the svg file. Let me show you what happens if you don’t clean the import first. Then the app.css, we can also clear that out as we are going to use bootstrap classes for our styling. Please remember this is not a css or design course. I’ll make this app responsive on a best effort but I’m mostly focussed on functionality . last but not least let’s get rid of the index.css file and as always, make sure you get rid of the import on the index.js file. Now we are ready to start building our react app

Step #4.3 Identifying components for basic Project CRUD ui

* First let’s discuss what a component is: in react, the ui is comformed by components, some which will have props (properties) and state, and some that are more functional and are not influenced by the aforementioned
* All these components get loaded in the App Component which is displayed at the root div, also these components will have sub-components in them.
* To illustrate this better, I’m going to run you by the components we will be creating to power the basic CRUD ops in react. Note: we are not yet working with Redux yet, once we introduce it, we will refactor accordingly.

Step #4.4 JSX

* JSX is a javascript extension that allow us to write html in our js code.
* This is very helpful and makes coding very easy, but like anything good, you must abide by certain rules. Since we want to get coding quickly, I will be sure to highlight those for you, a few examples would be only one main parent element div, classname, label htmlFor. Also stuff without ending tags needs to be needs to have the slash at the end Input />
* JSX expressions: first, in the render method, you can write javascript code to create var, lets, constants as well as functions.
* The 1st example is a const with a project name. when putting it on the jsx you need {projectname} to put it in curly braces, curly braces you can pretty much put javascript expressions such as ternary expressions. When needed, this is just a quick example to get you comfortable, remember this is not an introduction to React it is more of a refresher.

Step 4.4 – Create the Dashboard component

* Let’s create our first component which is going to be the Dashboard component. This component shows all the projects in an account. We don’t have security just yet, so for now we will assume that we are getting projects from 1 account
* In src create a folder called components and create the Dashboard.js file
* At the beginning of the course we installed a pluging called ES7 react redux. With this component, we have access to keyboard shortcuts that give us shortcuts to create class based and functional component, lets review the documentation, then we can type rcc ->tab and as you can see it generates the boilerplate for your component.
* Although this is valid syntax, I learned to always have the export default and the class name at the bottom. I have also seen many react instructors and experts do it that way. Once we start adding functionality, it will become clear that this style comes in handy while wiring up the component
* Now since this component is expected to display Projects in the account, we are going to hardcode some mock projects to it
* <div>
* <ul>
* <li>Project 1 - React Project</li>
* <li>Project 2 - Java Project</li>
* <li>Project 3 - Python Project</li>
* </ul>
* </div>

Nothing out of the ordinary, just quick example

Now that we have this we want to render our new component. As previously mentioned, the App component is the meeting place for all components that we wish to display so we will go to the App.js file and import the Dashboard component for this we import it like this: import Dashboard from "./components/Dashboard";

The reason why we do not need the {} curly braces is because we are using de export default statement. In the import, the {} stands for destructuring, as you can see, we are importing Component from react and we can use Component directly, if we were to get rid of {Component} then we would need to type React.Component which is a bit too verbose.

* Now we see the projects displayed. Note you can include the same component more than once so for example if I put dashboard twice, we get the same list twice in the UI

Step #4.5 Create the navbar functional component

* The next thing we want to do is start creating components for our layout. We are going to use the keyboard shortcut “rfc” to create a functional component, in this case the Header.
* Let’s refactor it a bit
* function Header() {
* return (
* <div>
* <h1>Kanban Tool</h1>
* </div>
* );
* }
* export default Header;
* Now let’s bring it to App js

Step #4.6 Working with Props in the component:

* What are props: props are input data that can be used for rendering a component. A prop is a js object. Usually, the default behavior is that the component is rerendered when props change, another important topic about props is that The props are immutable, so the component cannot change its props. The props are received from the parent component. The component can access the props through the this.props object. For example, take a look at the following component:
* Lets do an example, in the header component in the App.js file, let’s include a prop called toolType which will contain a string called “Kanban Prop”
* Now this prop is coming from the parent as it is where we declared it, in order to use this property, it needs to be one of the parameters of the arrow function that creates the Header. So we go to Heather.js and pass the keyword “props” as a parameter
* Now using jsx expressions syntax, we can open curly braces and include the props we want to get from the parent, in this case it is {props.toolType} an if we save, you can see that it took the header that we defined in the parent element
* A good way to check props associated with our component is in the react tool in chrome. If you highlight it, you can see Props read only and the one prop we are passing.
* Now, to keep practicing passing props from parent to child, we will do the same with the Dashboard element. This is also good practice to see the this in action for a class based component.
* <Dashboard project="Project 1 " projectType="React Project" />
* Now we are going to save this and go into Dashboard.js and consume the props we just passed to it at the parent component.
* So we can get rid of the list content we hardcoded
* Now we are going to include another dashboard component with different data so we can really bring this concept home and how it shows the reusability of a component in react
* A good way to go about props that is cleaner and less verbose is to use destructuring. Since props is a js object, we can extract the attributes and use them in JSX, let me show ya. In header:

const Header = props => {

const { toolType } = props;

return (

<div>

<h1>{toolType}</h1>

</div>

);

};

* Here we don’t need the “this” as this is not a class based component,
* Now let’s do this at the Dashboard, right in the render method, here we do need this.props because it is a class based component.

class Dashboard extends Component {

render() {

const { project, projectType } = this.props;

return (

<div>

<ul>

<li>

{project} - {projectType}

</li>

</ul>

</div>

And as you can see, everything still works great

Step #4.7 default props and PropTypes:

* Default Props is very simple, it merely allows you to set, like the word states, default properties for your component. In the case of a functional component suchs as the Header we can set the following

Header.defaultProps = {

toolType ="Please define a tool type"

}

When we save, no thing changes, but look at what happens if we remove the prop at the parent level

* Now lets introduce PropTypes, PT is a form of validation, you can use this to enforce prop data types, and whether or not a prop is required for the component, lets now work on the Dashboard component to illustrate this.
* First thing we need to do is import PropTypes in the Dashboard.js file. The snippets plugin that we installed has a set of nice shortcuts that will save us some typing ☺ so let’s check it out, as you can see here, we can use the command impt + Tab to create the import. Now that we have the import we will get a warning that is not being used, you can disregard for now but keep an eye on those, make sure you don’t have unnecessary unused variables in your projects
* Next step is to define the propTypes we want to “enforce” in your component. The syntax is as follow Component.propTypes => now keep an eye on the casing. While learning react I WASTED HOURS trying to figure out why my app didn’t work. A lot of times it was because I had typed Component.PropType instead of propTypes
* Then we define the prop, the type, and whether or not isRequired. We will see all kinds of types suchs as: object, func, string etc. now that we have defined this lets check what happens if we don’t pass props. And if we pass the incorrect props. We get the errors in the console. Just to be thorough, let me show you what happens if you misspell propTypes with “lower p”

Step #4.8 Install bootstrap

* In this course we are going to use bootstrap so we can have a decent looking prototype without diving too much into design. If you are not familiar with bootstrap simply got to getb…., they have good documentation and I promise you, it is very easy to learn.
* Now let’s open up the terminal in the project folder. Lets run the command npm install bootstrap / then let’s go to package.json to check the version. If you are watching in the future and there is a new version of bootstrap and classes get changed or deprecated, you will want to: force install this version or just research and use the new class name
* Next step is to import bootstrap into the App.js file. For this we need to 1st make find where it is installed, for this you can go to node\_modules / boostrap /dist/ css and import the min.css file, as you can see, vscode is really good at giving you a hand in creating the path

Step #4.9 lets create the initial front end and talk about State

* At this point in our project, we want to define 3 things: Layout, Dashboard and project items to be displayed on the dashboard, form to create and edit a project. Since this is a course mostly focused on React and Spring boot, we will avoid html / css coding. In this lesson you will find the html we are going to use. Please download it and follow along
* So let’s clean up our dashboard component so we can start bringing some of the desing to life.
* First I’m going to copy paste the Navbar design and Dashboard Design
* Now I’m going to copy and paste the design for the dashboard, please remember to change all class to className now let’s run it and we should see our app.

Step #4.10 State, display components within components

* Before we start calling our SpringBoot API, I do want to keep reviewing some react fundamentals, in this case we are going to start working on the state. In preparation to covering this topic let’s start by creating some sample data.
* First, in the component’s folder, lets create a Folder called “Project” and inside this folder a file called ProjectItem.js
* Let’s use the rcc – tab command to create a class based component as this will have state
* Now the fundamental idea of state is, like props, is the input data specified to render, or MOUNT, the component. In react, the initial value is given in the component’s constructor. So in Dashboard.js, we are going to create a constructor right outside of the render() method. Now that we have a constructor, please remember that we are extending a class, in this case, Component class. So we need to call the super() method inside of the constructor. We are java folks, this makes sense to us.
* Now that we have this, lets give the dashboard component a hardcoded state. The state in a component is an object. And in this object we are going to create an array of projects which is somewhat similar to what you’d get from the API when calling the get all projects endpoint. Let’s make sure we are following the attributes we defined in our POJO.
* We will start by including this.state = {}. Inside of this object we are going to create the project’s array and inside this array a few project objects. Just to drive this concept home why don’t we extract these objects right out of our server response. Will save us some typing and will make the idea clearer
* Run your server, open postman, go to your collections (If for some reason you haven’t created your collections just yet, please go back and do so! It will save you a TON of time)
* Let’s create a few projects, and then let’s find them all and grab the array of project json objects and paste it in our dashboard state. I want to rehash that in our design, we want the users to see all of their projects when they open the dashboard. So when the component is rendered, it need to load this info in the component’s state.
* Since this is a list, we need to iterate through it in order to extract the individual project object. For this we are going to use Map. 1st we are going to extract the project list from the component state by using destructuring, then we are going to set up the map function

{projects.map(project => (

<h1>{project.projectName}</h1>

))}

And then save and we should see our project names in the front end.

* Now that we have covered this basic step let’s work in the ProjectItem Component. The PI is basically a card that has info on the project object as you can see in our design project. So, lets copy that html code (always remember the JSX clean up ☺ )
* Now that we have that let’s import our project item comp to our dashboard, looks good, font awesome is not working, but let’s take care of that when we are doing wiring this thing up
* Lets include a few PI’s just to get a taste of the look and feel.
* The next thing we want is for each card to have the actual project data in the state, so for this, the 1st step is to pass the project info as a prop to the PIC. Let’s reuse the map code. Lets place the PI inside than pass each project as a prop

{projects.map(project => (

<ProjectItem key={project.id} project={project} />

))}

As you can see we have 3 projects which matches the amount that we have in the state.

* Now, next thing we want to do is extract the props we are passing on to the PIC and display exactly the date that gets mounted in the state. For this we are going to go to the PIC and in the render method we will destructure the project info from this.props
* Now let’s use the curly braces to update the components and boom! It displays the projects that we have in our state. Lets just do a quick check to prove the point

Step #4.11 Create the project form

* First we are going to create a functional component for the create button. We don’t necessarily need to do this, but I just want to show you a use case. Remember, this button has no state
* Now let’s import Link so we can declare the path to the form
* Now let’s create a class based component called addProject. This is the component that will contain the form
* Let’s add a quick h1 for now
* Now lets go to the App.js file and declare the route for our form
* Let’s test real quick
* Now let’s bring the project form from the design documents and update the class to className

Step 4.13 React Router:

* We want to continue to wire up our React App, and the next thing that we are going to bring in is React Router. This will help us display different paths and of course render different components
* In your project folder, we want to run the following command npm install react-router-dom to bring the dependency to our project.
* Now let’s go to our App.js file to enable the routing in the app. First we are going to import BrowserRouter from react-router-dom, which is the parent component that stores all the routes in the application. We could use it directly BUT it is a good practice to give it an alias, you can do this by adding the keyword as followed by the alias, in this case Router. We are going to add Route which is the tag we will use for our individual routes and Switch for static routing logic such as defining a 404 page for invalid urls
* After importing this, we want to wrap the main app with the <Router> tag.
* Now let’s create a route, first, right below header, we are going to get rid of the Component and we are going to bring in the Switch tab.
* Inside our switch, we will create our first route to our dashboard. After typing the tag, we want to specify the path. For this we will use the keywords exact path. Why exact? Because without it, it basically interprets the paths very ambiguously which will result in an awful user exp. Though it has its uses, we will not get into it in this course. Now we want to declare the component we want to mount when that path is called. In this case Dashboard. Lets test it

Step #4.14 Redux and HTTP requests CRUD with Redux

* Explain Redux: The main purpose of redux is to allow for the components to share data by storing the state
* The 1st step to npm i redux react-redux redux-thunk. Redux as explained in the state manager, react-redux is the library that allows for our react app to work with redux and redux-thunk is a middleware that …
* Next we are going to start configuring our app for redux for this we are going to 1st create a store.js file at the src directory. This file is what controls and holds our application state. As we dive into reducers, this will become clearer
* In the store.js file we are going to:
  + Import {createStore, applyMiddleware, compose} from ‘redux’ ->compose allows us to apply the middleware AND configure our store at the same time This is a functional programming utility, and is included in Redux as a convenience. You might want to use it to apply several [store enhancers](https://redux.js.org/glossary#store-enhancer) in a row. Inthis app we are applying the middleware and we also want to enable the Redux Dev tools in chorme.
  + Import thunk from ‘redux-thunk’
  + Import rootReducer from ‘./reducers’
  + Then create const initial state
  + Then const middleware so we can pass more than one
  + Then const store using the createStore
  + Then open the redux dev tool, get the instructions and paste what’s needed in the compose parameters
* Lets now create the reducers folder in the src folder and create an index.js which will be our root reducer. Since the store only takes one reducer, the strategy here is to combineReducers in this file, basically bundle all of your reducer actions in this file.
* Then we are going to import {combineReducers} from ‘redux’; and export a list of all reducers we will use

import { combineReducers } from "redux";

export default combineReducers({

//Add reducers we need

});

* Let’s now go to the App js to wire up redux and the store. We are going to import {Provider} from ‘react-redux’ library which is the main component that provides our application with the state coming form the store
* Now we want to wrap the entire application in the provider and pass the store to the provider
* Now that we have wired this up, lets create the first reducer, we are going to feed to our rootReducer: the projectReducer, in the reducers folder create a projectReducer.js
* Steps to set up the reducer:
  + Declare initial state, all reducers will have an initial state
  + Then export default function that takes a state and action
  + Then switch statement for the diff actions and we will start with the default one
  + Then lets include the project list in the state
  + Then wire it to index js (project:projectReducer)
  + Then check it in the state, then pass it a dummy project object to the initial state and check it again

Step #5 Hook up the back end with the react/redux front end

Step #5.1 GET ALL PROJECTS

* As you remember, we hard coded the initial state for the account dashboard right in there and also hard coded a project in the initial global state. What we want to do next is to actually get that data from the spring boot API and slowly enable our back end and front end to work together
* The 1st step is going to be to load the projects in our API to our dashboard, so let’s clean some stuff up, let’s start with the initial state in the project reducer and then with the state we hardcoded in the dashboard
* Now, in the src folder, we are going to create a folder called Actions. Here we will define the actions for all the components we will have in our app.
* Inside this folder we are going to create 2 files types.js and projectActions.js
* If you remember, when we created the reducer, we set up a switch statement that has different cases depending on the action type. in the types.js file, we configure the types of actions available for our components
* Now let’s import the 1st action we are going to enable which is to get all projects in both our projectActions and projectReducers files
* So in projectActions, import axios (npm install it) and set up the getProjects() method

import { GET\_PROJECTS } from "./types";

import axios from "axios";

export const getProjects = () => async dispatch => {

const res = await axios.get("http://localhost:8080/api/project/all");

dispatch({

type: GET\_PROJECTS,

payload: res.data

});

};

* Now in the reducer lets create the case for GET PROJECTS in the reducer

case GET\_PROJECTS:

return {

...state,

projects: action.payload

};

* Now lets let’s go to dashboard and use the lifecycle hook “ComponentDidMount” this lchook pretty much dictates: “everytime you render or load this component, you need to mount this data” in our case it is the list of projects. There is another option to this called willmount, the reason we are using didmount: <https://daveceddia.com/where-fetch-data-componentwillmount-vs-componentdidmount/>
* We don’t need the constructor anymore, so we will replace it with componentdidmount
* Now to wire this thing up we need to import the action getProjects
* Now let’s set up the proptypes we need so far

Dashboard.propTypes={

project: PropTypes.object.isRequired,

getProjects: PropTypes.func.isRequired

}

* Now we need to map the state to the properties so we can extract the array, remember the projects prop contains in the array the state.project, is what is in the root reducer, and projects is what we have in the initial state. <https://www.sohamkamani.com/blog/2017/03/31/react-redux-connect-explained/>
* Now that that’s been taken care of, we need to connect our component to the redux store so that our component has access to the state. For this we need to import connect from react-redux
* Now we need to finally connect our component with the store, for this we bring the func, wrap the component name in parenthesis, and then pass two parameters to the connect function: the props we are mapping to the state and the action we imported
* Now that we have all of that, we need to update the component did mount with by specifying the getContacts function required for it to render and in the render function now we know that “projects’ comes from the props.project that we mapped from the state.
* Even though we don’t have security, we might still need to enable crossorigin so our back end can talk to our front end. When we implement security we will cover this for then entire app, for now, all you need is to update the controller with @CrossOrigin

Step #5.2 DELETE Actions

* Having fun? God knows I am! Ok so now we are pretty much going to hit our API with a POSTS and make sure it is working
* So let’s start with the delete action, import the deleteproject type

export const deleteContact = id => async dispatch => {

await axios.delete(`http://localhost:8080/api/project/${id}`);

dispatch({

type: DELETE\_PROJECT,

payload: id

});

};

* Now let’s update our reducer with the delete action, import the type as well (the only reason I’m doing it so repetitive is so you get the hang of this)

case DELETE\_PROJECT:

return {

...state

};

* Now lets update the component to handled the delete action. 1st lets import the action to the ProjectItem component which is the one holdin the delete button
* The delete project is an a tag, but react uses a different tag called link so let’s import and implement it. Then let’s update all the a tags with Link and change all the hrefs for “to” which is how Link rolls. Now for the delete we won't use link, just the role=”button”
* Then let’s take care of the proptypes required for this component, we have to import the PropTypes btw
* Then we need to import connect for react-redux, and the mapstatetoprops is going to be null for now
* Now lets enable to button to be clicked, there are several ways of doing this, but for now let us start with the basics. Lets create the function for ondelete when clicked. This function takes the project id and calls the delete action
* Now we need to enable the button to call this function to by using the onClick function, we need to bind the function to the specific object being clicked and of course pass the id of the project that we want to delete.
* Let’s test it, it is gone from the ui, let’s check the db, gone.

Step #5.3 CREATE project Actions

* Let’s start by creating the action, please not that this is only the initial state, we are going to have a lot of fun on this part once we get to the validation piece, for now, let’s happy path it

export const createProject = (project, history) => async dispatch => {

const res = await axios.post("http://localhost:8080/api/project", project);

history.push("/dashboard");

dispatch({

type: ADD\_PROJECT,

payload: res.data

});

};

* Now let’s update our reducer with the add project just returning the state for now. This will most likely force us to refresh the dashboard to see the new project, don’t worry about that for now.

Step #5.4 CREATE project action, Turn the Create form into a controlled component

* The values that will compose a valid new project will come from the state of each field at the moment the form is submitted. So, whatever was written by the user goes into the db as long as it is valid
* So, first, lets create a constructor for the component, in this constructor we need to set the state of the attributes we are passing. As with any basic form, we start off with empty fields. So we need to set the initial state by creating an empty project object in the state. We can basically grab the fields from our json just so we are super duper sure, there are no typos
* Now we need to set the value of the inputs to what’s on the state, make sure the name matches the attribute that we set in the state.

name="end\_date"

value={this.state.end\_date}

* Now, let’s try the form. As you can see, you cannot type anything… why? Because state is immutable. The only way to change state is to set it by using the setState function.
* I’ve seen many ways to go about this, but a pattern that I’ve come to like is and that I personally recommend, is to just create a function that sets the updated state for the fields, we will call this onChange. This function takes an event parameter
* Now with this we can start enabling the input to change the state. There is a long way and a short way of doing this. Let me show you the long way with one field, project Name

onChange(e) {

this.setState({ projectName: e.target.value });

}

* Then we need to bind this to the input field with the name projectName

<input

type="text"

className="form-control form-control-lg "

name="projectName"

value={this.state.projectName}

placeholder="Project Name"

onChange={this.onChange.bind(this)}

/>

If I do this, it allows me to type, but only on that input field which means that I’d have to repeat myself in order to enable each field.

* There is a MUCH better way to go about this which is to have only 1 onChange method for all the fields. So to do this, first let’s include this to all our input fields without the bind
* Since we defined the name attribute on each input to match what we declared in the initial state (WARNING, THEY HAVE TO MATCH PERFECTLY FOR THIS TO WORK) we can just pass it in brackets.
* Now the last step is to bind this, we can do .bind.this to every field like this:

onChange={this.onChange.bind(this)}

* Or we can DRY the hell out of it by adding it to the constructor instead

this.onChange = this.onChange.bind(this);

Step #5.5 Submit the project form – Happy Path

* Now we are just going to submit the form to our back end by passing a valid project object.
* First we need to import the projectAction we need to talk to our back end
* Then we create the function onSubmit that the form will invoke to hit the backend. This function takes an event parameter
* Then this form gathers the input data and packages it in a new project object that contains the fields we have defined. To avoid typos, lets copy and paste what we defined in our state and make sure that the object contains whatever data is in the state of this component.
* Let’s wire this function up so it is the form’s action,

<form onSubmit={this.onSubmit}>

* then let’s bind it in the constructor just like we did for on change and let’s perform a field test by login the object on submit.

this.onSubmit = this.onSubmit.bind(this);

* As you can see, displayed it for a split second but went ahead and reloaded the page. Since we passed it an event, lets use the js function preventDefault() on the onSubmit() function so it doesn’t reload
* So now we can see our object, so it is time to write this to the backend, let’s start by defining the proptypes we are working with right now: lets first import PropTypes

import PropTypes from "prop-types";

AddProject

.propTypes = {

createProject: PropTypes.func.isRequired

};

* Now we need the connect from react redux

import { connect } from "react-redux";

export default connect(

null,

{ createProject }

)( AddProject);

* Now we are able to pass the create project function to our onSubmit in order to persist our new project.

this.props.createProject(newProject);

* Please note that it is not going to look great just now. We just want to make sure we are hitting the end point
* And we just did, let us try this again AND IT WORKS

Step #5.5 Implement error validation

* Our form works great when we pass it a valid project object, but what happens when we don’t. For starters, our backend has validation that we implemented and tested in previous videos. But what would happen now that we have a front end? basically the form won’t submit and the user will have NO IDEA of what the heck went wrong
* In order to wire up our react app to display the server errors we need to start by creating the GET\_ERRORS type in actions/types.js

export const GET\_ERRORS = "GET\_ERRORS";

* Now, I want to create an errors reducer file in the errors folder
* We are going to start by importing the GET\_ERRORS type
* Then we are going to declare the initial state for errors which is an empty object
* Lets create the reducer

import { GET\_ERRORS } from "../actions/types";

const initialState = {};

export default function(state = initialState, action) {

switch (action.type) {

case GET\_ERRORS:

return action.payload;

default:

return state;

}

}

* Now let’s bring the reducer to the root reducer
* Now let’s update our createProject Action and try it, because at this specific point, we should at least see the errors from the service in our state. We can simply use a try catch and update our action as follows:

export const createProject = (project, history) => async dispatch => {

try {

const res = await axios.post("http://localhost:8080/api/project", project);

history.push("/dashboard");

dispatch({

type: ADD\_PROJECT,

payload: res.data

});

} catch (err) {

dispatch({

type: GET\_ERRORS,

payload: err.response.data

});

}

};

* Now that we see the errors in our state, we need to update our component to display these errors. First we are going to update the initial state of the component with an empty errors object
* Then we are going to use the componentWillReceiveProps lifecycle hook so that we can get the errors from our state. Since an error or a group of errors represent a new stream of props, componentWillReceiveProps will help us grab those new props from the state. This lchook takes nextProps as an argument
* Then we verify if the new props coming in have errors and if it does, then we update our errors object
* Next we are going to update the proptypes to specify the error object and last but not least, we are going to map the state to props since we are in fact grabbing this from the state
* Now lets pass it on to our connect function
* Now in the render function, lets use destructuring to grab the errors form the state

render() {

const { errors } = this.state;

* Now everything should be wired up for us to display errors on the UI,
* In order for us to have a more dynamic experience, we are going to use the classnames package. This will allow us to implement logic in the jsx that we will use to make our UI display custom error messages, go to terminal and npm install classnames
* Now that we have it installed, let’s import it

import classnames from "classnames";

* Now we are going to use the ‘is-invalid’ bootstrap class to highlight fields that are invalid. However we want this to be more dynamic so we are going to use classnames to add logic
* In className lets add curly braces and classname, first parameter is the classnames that we want to stay by default which is the class we want for the form in it’s normal state. Then, the second parameter is the class we want to add IF the condition is met.

className={classnames("form-control form-control-lg", {

"is-invalid": errors.projectName

})}

* Now, it’d be really nice to get a message specifying what the problem is, so right after the input tag we can add the following logic to add a message:

{errors.projectName && (

<div className="invalid-feedback">{errors.projectName}</div>

)}

* This should be extremely easy for you to implement the rest on your own. Go check the POJO and implement it for all objects that have validation. If you cannot implement this on your onw, rewatch the lecture until you can, seriously, no point in moving forward if this is not clear.

Step #5.6 Update Project – Action and State

* First thing we are going to do is create a new component for updating a project. There are options to add logic and reuse the AddProject Component BUT I rather give you the extra practice of creating a new component with similar features. This one is going to be fairly similar to the AddProject component, it will even use the same action, the main difference is that it needs to load the current project to component’s state so we can in fact update it
* We can Literally copy past the AddProject just to save time, I do recommend that you do take the time to pause the video here, write it down, let it stick!
* Next step is make the projectId disabled, remember that the projectId is inmutable.
* Now let’s try it, lets use the Link to in the update project button and add the “/updateProjet” path
* Then let’s go to the appjs file and add the route to the new component. That works
* Now lets create new types, we need 2: GET\_PROJECT, and UPDATE\_PROJECT
* Then we need to go to your projectActions file in order to fetch the 1 project we want to update (we need to import the new two types)

export const getProject = id => async dispatch => {

const res = await axios.get(`http://localhost:8080/api/project/${id}`);

dispatch({

type: GET\_PROJECT,

payload: res.data

});

};

* Now we need to update our projectReducer, first we need to account that our initial state can also contain a single contact object. Right now we just have an array.

const initialState = {

projects: [],

project: {}

};

* Now let’s create the GET\_CONTACT case, (we are using the auto import in vscode, if not always make sure you import the type)
* Now that we have this, what we want to do is that when we call “GET\_CONTACT” this fills out the form so we can just change the existing attributes
* So in the updateContact form, we have some of the imports we need, however we still need to bring in the getProject action
* We also need to update our mapStateToProps, to map our single contact, if we look at the state, you can see that we have both projects and single project, this is the reason we want to be specific here.
* We also need to update our PropTypes now that both the getContact func and the contact object are needed for this component to function
* Finally we are going to update our connect parameters to pass the get project function

UpdateProject.propTypes = {

createProject: PropTypes.func.isRequired,

getProject: PropTypes.func.isRequired,

errors: PropTypes.object.isRequired,

project: PropTypes.object.isRequired

};

const mapStateToProps = state => ({

errors: state.errors,

project: state.project.project

});

export default connect(

mapStateToProps,

{ createProject, getProject }

)(UpdateProject);

* We are done wiring up the action so we can use the lifecycle hook componentDidMount to load up the project onto the form. If you remember, our API takes the project id, for both getting a single project and updating one.
* So we are going to start by extracting the project id from the path parameter, we are going to use destructuring for this
* Then we are going to pass that id to our action that gets a project by ID
* Before we load the form, let’s make sure we update the update route to pass the id as a parameter
* First we will go to the projectItem component and update the link to on the update button to contain the project id. Since this this going to be an expression that takes the id parameter we need to wrap it in curly braces

<Link to={`/updateProject/${project.projectId}`}>

* Now we need to update the ROUTE in App.js so it is clear that the route needs to pass a param prop to the component. The syntax is as follow

<Route exact path="/updateProject/:id" component={UpdateProject} />

* Now we should be able to click update project and load the project on to our state

Step #5.6 Update Project – Load edit form with existing data

* Alright so we have accomplished getting the project on to our state not it is time to grab this from our state so we are going to update our componentWillReceiveProps to load our form
* We are going to use destructuring here to set the state with the attributes we are getting

componentWillReceiveProps(nextProps) {

if (nextProps.errors) {

this.setState({ errors: nextProps.errors });

}

const {

id,

projectName,

projectId,

description,

start\_date,

end\_date

} = nextProps.project;

this.setState({

id,

projectName,

projectId,

description,

start\_date,

end\_date

});

}

* Now, as you remember, our api saves and updates, and in order for us to differentiate one from the other we have to pass an existing db id, if you don’t pass an existing db id it will think that it is creating a new one so the submit a project with an existing projectId will throu an exception. All we need to do now is just pass the id as an additional parameter in the onSubmit Function

onSubmit(e) {

e.preventDefault();

const newProject = {

id: this.state.id,

projectName: this.state.projectName,

projectId: this.state.projectId,

description: this.state.description,

start\_date: this.state.start\_date,

end\_date: this.state.end\_date

};

BOOM it works!!! Lets make sure our validation still stands, yes it does. Now because of this, we don’t really need the UPDATE\_ in type or reducer, if you want to be thorough you might want to create an action for update but again, not needed

Step 6. Secure our backend

This portion of our project is going to have a considerable amount of moving parts, so I will move a bit slower here, When I first set it up I ran into a lot of bugs just because I was swing it, I don’t want you to have to jump through hurdles, so I rather teach it to you nugget by nugget so you can easily go back to whatever it is that might’ve broken your app

* Before we move forward with more features, we want to secure our app.
* We want the user to be able to register using a valid /unique email
* We want the user to log in and see his/her projects and only his/her projects
* We want to make sure that our backend is stateless and that we use a JWT token to secure it.
* So let’s begin:

Step #6.1 initial set up

* bring spring security dependency into our POM file, this will immediately lock down our application, let’s see, as you can see we can no longer use the API unless with are authenticated and authorized to check this resource
* now If you checkout your console, you’ll notice that spring security provides a generated password, if we use it with the generic admin username, then we will be able to get all projects
* At this point we want to start configuring Spring security, for this we are going to create a package called “Security” and in this package we will create a SecurityConfig class
* We need to give this class 2 annotations @Configuartion and @EnableWebSecurity
* Now our class needs to extend the WebSecurityConfigurerAdapter. By extending this class we can override the access rules and make it our own
* Now that we have extended this, we have a list of methods that we can override
* Since we are going to be dealing with http requests we are going to override the configure(HttpSecurity http) method

Step #6.2 User domain

* In the domain package, we are going to create user class and annotate it as @Entity
* We are going to add the following attributes to the user object

|  |  |  |  |
| --- | --- | --- | --- |
| Field | Type | Validation | ORM |
| id | Long | @Id | @GeneratedValue(strategy = GenerationType.***IDENTITY***) |
| username | String | @Email(message = **"Username is Invalid, it needs to be an email"**) @NotBlank(message = **"username field is required**  @Column(unique = **true**)  **and it needs to be a valid email"**) |  |
| Fullname | String | @Size(min = 2, max = 30, message = **"From Java Server - Name must be between 2 and 30 characters"**) |  |
| password | String | @NotBlank(message = **"Password field is required"**) |  |
| confirmPassword | String | @Transient |  |
| role | String | @Lob @NotBlank(message = **"Project description is required"**) |  |
| created\_at | Date | Create onCreate() method with @PrePersist |  |
| updated\_at | Date | Create onUpdate() method with @PreUpdate |  |

* Let’s now create an empty constructor
* Next thing is to create a user repository interface, extends crudrepo, @Repository annotation

Step #6.3 Enable user registration, happy path

* Next component we need to create is a UserController. This is an @RestController and has a path of “api/users”
* Next order of business is to create the @PostMapping with the path “/register”. We are going to follow a very similar pattern as we used when creating a project object. Feel free to challenge yourself, right now we are not looking at the user => project relationship, all we want is the happy path of creating a new user
* Let’s put the controller work on hold for a bit and go and work on our service layer. In the services package, create a UserService class and annotate it with @Service
* Next thing we want to do is to to autowire the User repository
* Now let’s create our SaveOrUpdateUser method. Remember we are just happy pathing at this point

@Autowired  
**private** UserRepository **userRepository**;  
  
  
**public** User saveOrUpdateUser(User newUser){  
 **return userRepository**.save(newUser);  
}

* Cool so now lets Autowire our new service in our User controller and so we can save a new user
* Now let’s update our controller to save the new valid user and return it

@PostMapping(**"/register"**)  
**public** ResponseEntity<?> registerUser(@Valid @RequestBody User user, BindingResult result){  
  
 User newUser = **userService**.saveOrUpdateUser(user);  
  
 **return new** ResponseEntity<User>(newUser, HttpStatus.***CREATED***);  
  
}

* Now that we have our end point, the next thing we are going to do for now is to open the registration end point so we can test the happy path. For this we need to go to the SecurityConfig file and give it an initial http configuration to permit access to this path, first we are going to disable csrf
* **public class** SecurityConfig **extends** WebSecurityConfigurerAdapter {  
    
   @Override  
   **protected void** configure(HttpSecurity http) **throws** Exception {  
   http.csrf().disable()  
   .authorizeRequests()  
   .antMatchers(**"/api/users/\*\*"**)  
   .permitAll();  
   }
* Now that we have updated our security, let’s go to postman, create a folder for the User’s collection and then let’s attempt to create a new user object by passing the required attributes in the object. I’m going to do some silly copy pasting just because I don’t like leaving anything to chance. For now let’s comment out the confirmPassword
* Let’s add a json object in the body at post man and send it and great, it works!!

{

"username":"test@test.com",

"fullName":"John Doe",

"password":"password"

}

* Let’s now save the postman request and check our database, uh oh, we can’t check our h2 db!! The reason for this is that our application is secured now, so we need to add the “h2-console” path to permit it, we also need to additional configurations which allows us to open the h2 console
* http.csrf().disable().headers().frameOptions().sameOrigin().and()  
   .authorizeRequests()  
   .antMatchers(**"/api/users/\*\*"**)  
   .permitAll().antMatchers(**"/h2-console/\*\*"**).permitAll();
* ok so now we can see that we can access the db AND that the record is created, now what about validation? Good thing we created a maperror service, now we can just reuse it for us to display errors properly, so let’s autowire the MapValidationErrorsService and implemented just like we did for the create project endpoint and try it, works great
* Now let’s check another validation which is that an email needs to be unique. As you can see we get a nasty 500, BUT we know how to handle this don’t wee, remember we did this for the project ID to be unique. It is literally the same impl. You might even reuse that exception, but I rather give you the practice of creating another custom one

Step #6.4 User registration custom exception

* Create two java classes UsernameAlreadyExistException and UsernameAlreadyExistResponse in the exceptions package
* Let’s wire up the the exception, 1st we need the response status annotation so we can define the status we want it to return, then we are going to extend the RuntimeException class, then we are going to give it a constructor that takes the message we want to display
* @ResponseStatus(HttpStatus.***BAD\_REQUEST***)  
  **public class** UsernameAlreadyExistException **extends** RuntimeException {  
   **public** UsernameAlreadyExistException(String message) {  
   **super**(message);  
   }
* Now let’s work on the Response, this response is specific to the field “username” now all we need is a constructor and getters and setters,
* **public class** UsernameAlreadyExistResponse {  
    
   **private** String **username**;  
    
   **public** UsernameAlreadyExistResponse(String username) {  
   **this**.**username** = username;  
   }  
    
   **public** String getUsername() {  
   **return username**;  
   }  
    
   **public void** setUsername(String username) {  
   **this**.**username** = username;  
   }  
  }
* Now we need to update the CustomizedResponseEntityExceptionHandler to include this exception

@ExceptionHandler  
**public final** ResponseEntity<Object>  
handleUsernameAlreadyExist(UsernameAlreadyExistException ex, WebRequest request){  
 UsernameAlreadyExistResponse exceptionResponse = **new** UsernameAlreadyExistResponse(ex.getMessage());  
 **return new** ResponseEntity(exceptionResponse, HttpStatus.***BAD\_REQUEST***);  
}

* And now all we need is to update the Userservice to throw the exception if the username already exists, we can use an if statement or a try / catch. Let’s use the try catch. Remember the exception will happen when we try to set a username that already exists
* **public** User saveOrUpdateUser(User newUser){  
    
   **try** {  
   newUser.setUsername(newUser.getUsername());  
   **return userRepository**.save(newUser);  
   }**catch** (Exception e){  
   **throw new** UsernameAlreadyExistException(**"Username "**+newUser.getUsername()+**" already exists"**);  
    
   }  
    
  }

Sweet, this is looking really good!

Step #6.5 User registration password confirmation

* I’m sure you remember that we also want to be able to use a confirm password field. Though we are not going to persist this password, we do want to compare notes before we persist the actual password
* For this we are going to use the trascient annotation to indicate that we are not persisting this to the db
* Now we are going to create a custom validator in order for us to make sure the passwords match BEFORE persisting, let’s create a validator package and a class called UserValidator we are going to give this class the @Component annotation so it is picked up at the component scan at runtime and then we are going to implement the Validator interface, it is important that we implement this one **import** org.springframework.validation.Validator; and not the javax one
* Now let’s implement the methods of the interface
* First thing we need to do is specify the class that this validator supports the way to implement this is as follows
* @Override  
  **public boolean** supports(Class<?> aClass) {  
   **return** User.**class**.equals(aClass);  
  }
* Now let’s specify what it is that we need to validate, let’s start by instantiating the new user object we are going to validate, we are going to cast the object parameter it takes as a User object
* Now here we are going to add two validations rules, length of the password and whether or not it matches the password confirmation
* @Component  
  **public class** UserValidator **implements** Validator {  
    
   @Override  
   **public boolean** supports(Class<?> aClass) {  
   **return** User.**class**.equals(aClass);  
   }  
    
   @Override  
   **public void** validate(Object object, Errors errors) {  
    
   User user = (User)object;  
     
   **if**(user.getPassword().length() <6){  
   errors.rejectValue(**"password"**, **"Length"**, **"Password must be at least 6 characters"**);  
   }  
     
   **if**(!user.getPassword().equals(user.getConfirmPassword())){  
   errors.rejectValue(**"confirmPassword"**,**"Match"**, **"Passwords must match"**);  
   }  
    
    
   }  
  }
* Next thing we need to do is to Autowire the uservalidator to the user controller and run the user and the result by the validate function we just updated, this is our new controller now
* @PostMapping(**"/register"**)  
  **public** ResponseEntity<?> registerUser(@Valid @RequestBody User user, BindingResult result){  
   **userValidator**.validate(user,result);  
   ResponseEntity<?> errorMap = **mapValidationErrorsService**.MapValidationErrorsService(result);  
   **if**(errorMap != **null**) **return** errorMap;  
    
   User newUser = **userService**.saveOrUpdateUser(user);  
    
   **return new** ResponseEntity<User>(newUser, HttpStatus.***CREATED***);  
    
  }
* Let’s restart the server and try this puppy, works great, we are done and we have a clean impl, including validation and exception handling for our registration API

Step #6.6 Encrypt password AND ignore the confirm password.

* Ok, 1 last step is to encrypt the password, if we look at our data base, we have a string there and this is not very secure, so we are going to use an API that comes with spring security called bcrypt to encrypt the password persisted.
* If you remember you basic java, the class with the psvm method is the entry way into the application, for this reason we are going to wire the password encoder in the KanbanToolApplication. 1st we are going to use the @Configuration annotation, in order to enable what we are going to do next.
* we are going to use the @Bean annotation on a default method that return a bcryptencoder. This will make this available to our application so we can Autowire it and use it to encrypt our password
* @SpringBootApplication  
  @Configuration  
  **public class** KanbantoolApplication {  
    
   @Bean  
   BCryptPasswordEncoder bCryptPasswordEncoder(){  
   **return new** BCryptPasswordEncoder();  
   }  
    
   **public static void** main(String[] args) {  
   SpringApplication.*run*(KanbantoolApplication.**class**, args);  
   }  
  }
* Now we are going to autowire this to our UserService and use the encode method in bcrypt to encode the password we are trying to persist. Let’s try it again. Great you can si in the json response that we get our password and after checking the db we can also see we have your encrypted password there
* I don’t like, however, that the json response still displays the string version of our password in the confirmPassword field, there is a very easy way to go around this, let’s just encode both ☺
* Hmm didn’t like how slow that was, took 400ms, what if we leave it blank instead? As you can see it was a bit faster 319 ms. Since we don’t really do anything with that, let’s not waste cycles in encrypting it, let’s just throw it away. Alright, we are well on our way to start configuring our security, this is going to be fun

Step #6.7 Configure our entry point.

* Next step we want to take is to be explicit to those trying to access our endpoint with out the right credentials that they in fact need to be authenticated. But first we need to be explicit on making sure that all requests other than the ones we have identified are authenticated. So we are going to go to the SecurityConfig.java file and add the following config:
* http.csrf().disable().headers().frameOptions().sameOrigin().and()  
   .authorizeRequests()  
   .antMatchers(**"/api/users/\*\*"**)  
   .permitAll().antMatchers(**"/h2-console/\*\*"**).permitAll()  
   .anyRequest().authenticated();
* Now if we try any of the resources in our api we should get a forbidden error
* The first thing we are going to customize is our authentication entry point. The implementation of this springsecurity interface provide the implementation of the commence() method. This method is called whenever an exception is thrown due to an unauthenticated user trying to access a resource that requires authentication. Which is what happened when we tried to create a new project
* Since we are going to secure our end point with a JWTToken we will start wiring up things accordingly
* In the security package create a class called JwtAuthenticationEntryPoint. This class needs an annotation of @Component so that the component scanning mechanisim can pick it up and pull it into the application context. This class also implements the AuthenticationEntryPoint interface. This, again, will allows to override the commence method which is the one that deals with handling unauthorized requests
* Let’s now do a quick test of having a custom error message by configuring the the response. We will use the httpservletresponse that this method takes as a parameter and we are going to call its sendError method. This method takes two parameters, the type of response and a custom message.
* Next stop is our SecurityConfig file, we need to perform a few changes so that spring security uses our custom unauthorized message
* First we need to autowire the JwtAuthenticationEntryPoint into our securityconfig file
* Then in our configure method we are going to to tell spring security that for exceptionhandling at the authentication entry point please use our custom implementation of the commence method

**protected void** configure(HttpSecurity http) **throws** Exception {  
 http.csrf().disable()  
 .exceptionHandling().authenticationEntryPoint(**unauthorizedHandler**).and()  
 .headers().frameOptions().sameOrigin()  
 .and()

* Now let’s restart and try this, now as you can see, we have more control over what we display as an error message, I’d still want to customize it a little more rather than having such a generic error message. We will do that in the next video

Step #6.8 Configure our entry point message

* We just got a request from our front end dev team, they want the message to come back as follows from the api:

{

"username": "Invalid Username",

"password": "Invalid Password"

}

Their reasoning is that they don’t want to give a potential malicious user a clue of what it is that they are missing, because, so we are just telling them in the UI, your credentials are bad. In order to accomplish this, we are going to create a custom response

* A custom response like this is an object, so we are going to create an InvalidLoginResponse object, we will do so in the exceptions package, it takes two parameters: username, password
* Then we are going to create a constructor that takes no parameters but that hardcodes the message that our front end team requested, since we are not implementing any logic that specifies whether it is the username or password that is wrong, this should be enough, lastly getters and setters.
* Now back to our JwtAuthenticationEntryPoint lets implement this response, first we need to instantiate the new Invalid response object, then we need to declare the content type we need in our response which in this case is json, after that we want to set the status which is 401 and then we want to print the custom response by using the getWriter().print() method.
* So we tried it, doesn’t look good! Seems like for some reason it is not passing json. One way around it is to use Gson. Gson is an open source Java library to serialize and deserialize Java objects to JSON which is what we are going to do right now
* So let’s google gson’s maven dependency so we can bring it into our project’s POM file,
* Now lets convert our response to json, try one more time, voila
* **public void** commence(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse, AuthenticationException e)  
   **throws** IOException, ServletException {  
   InvalidLoginResponse loginResponse = **new** InvalidLoginResponse();  
   String jsonLoginResponse = **new** Gson().toJson(loginResponse);  
    
   httpServletResponse.setContentType(**"application/json"**);  
   httpServletResponse.setStatus(401);  
   httpServletResponse.getWriter().print(jsonLoginResponse);  
    
  }

Step #6.9 Custom Spring Security User Details

* In order to start working on Authentication and Authorization we need to define our custom UserDetails. Spring Security will use the information stored in this user to perform authentication and authorization.
* A common pattern is to implement a separate UserDetails class from the User POJO. However, I’m going to do it a bit differently and implement the UserDetails interface right on the User POJO. There is a case to be made in terms of not doing it this way, and one is that it clutters the POJO too much and I agree. The thing is that we are going to keep changing the user POJO when we create relationships with the projects and backlog items and for now I want to avoid the complexity of updating two files
* We are going to go to our User.java file and implement the UserDetails interface
* Then we are going to implement the interface methods at the very bottom of our java file
* We are not going to implement anything related to authorithies in this phase of the project so we can leave that null
* Then we are going to set up all the other methods to true. You can add logic for these conditions, but we don’t have those requirements now.
* Now, we have changed our pojo, lets run the server and try to create a new user… now we get the new methods being returned, and I don’t think we want to pass so much stuff to our front end, so we are going to use the @JsonIgnore annotation so we leave the out of the json response, now everything is back to the way it was

Step #6.10 Implementing the UserDetailsService

* Now that we have our userDetails, we need to implement a service that loads the user by username if the user exists. We need this service to load the user and then verify that the password is valid. Since we are using a jwt token, we are going to implement this in a rather different way than you might’ve when implementing spring security for templating engines
* In the security package, we are going to create a CustomUserDetailsService. This class needs the @Service annotation and it implements the UserDetailsService interface. It implements its methods
* We need to implement two methods, let’s 1st override loadbyusername, for this we need to @Autowire our UserRepository. Please note, this will not work with a service, it requires the repo
* Then we need to enable the userrepo so a user can be found by the username attribute. We are going to use JPA for this findByUsername
* now let’s wire up this method
* @Override  
  **public** UserDetails loadUserByUsername(String username) **throws** UsernameNotFoundException {  
   User user = **userRepository**.findByUsername(username);  
   **if**(user==**null**) **new** UsernameNotFoundException(**"Username not found"**);  
   **return** user;  
  }
* Next thing we need to do is to create a method that loads a user by Id, we will need this in the future when we define the JWTAuthenticationfilter
* We can use crudrepository findbyid to get the user by the long id, however, you need to know that the new version of the CrudRepo, returns an Option type. nothing wrong with this, one of the main purposes of Optional is to avoid null pointer exceptions. I do preffer to stick with the basics so I’m going to force this a little bit. Crud repository has a way around Optionals which are the gets
* In the user repository, lets create a method that returns a user object instead of an optional one (getById) again nothing wrong with using the Optional, BUT it is always good to know a way around this
* Now we can finish wiring up our loadbyid method. Last thing we want to add to our method is the @Transactional annotation, this provides transaction management from Jpa, please make sure you use the one from the springframework. This is quite broad topic.

Step #6.11 Generate our JWT token

* We are ready to generate a valid token for authenticate users.
* First we are going to do some ground work, to abstract the jwt configuration away from the token provider. For this we are going to create constants.
* In the security package, create a java class called SecurityConstants.
* Here we are going to define a few constants to set up our our jwt token provider and even some config that we are throwing into are SecurityConfig java file. It is always a good idea to abastract things this way.
* First constant is the secret keyworkd needed by the JWT
* **public static final** String ***SECRET*** = **"SecretKeyToGenJWTs"**;  
  **public static final long *EXPIRATION\_TIME*** = 3000000; *// 10 days***public static final** String ***TOKEN\_PREFIX*** = **"Bearer "**;  
  **public static final** String ***HEADER\_STRING*** = **"Authorization"**;  
  **public static final** String ***SIGN\_UP\_URL*** = **"/api/users/register"**;  
  **public static final** String ***H2\_URL*** = **"/h2-console/\*\*"**;
* not going to explain what this means right this second because if you are new to jwt it will make more sense as we start implementing this
* Now that we have this, in the same security package let’s create a JwtTokenProvider class. Before we do anything, we need to bring the jsonwebtoken dependency, we need this to generate and validate our jwt
* Now we are going to annotate our JwtTokenProvider class with the @Component
* The first method we are going to create in this class is the one that generates the token, this method returns a string (token) and takes in the Authetication OBJECT from the spring security core API
* 1st step is to get the authentication principal, basically, once the username are password are correct, we get this authenticated user to provide it with a jwt
* When we use the getPrinicipal, we need to cast it to type user
* Then we want to grab the current system date in milliseconds
* Then we want to set up an expiry date which is the current system date + the expiry time you want to allow for your token which we abstracted in the SecurityConstrains class
* Now let’s start building our jwt token, for this we are going to return Jwts object, this object will contain the following attributes
* In the setSubject we are going to pass the application user id, specifically the one we get from the principal, this takes a string so we need to extract the value and parse it as a string when passing it to the set subject
* Next thing we are going to do is to set up claims. Claims are attributes from the entity, in this case our User object that we want to pass on to the token. This way, when the react tool gets the token, it can extract important information from the user
* Before we can pass claims individually as you can see in this method or a map containing several claims from, in this case, the user. So what we are going to do is create a map with the claims we wish to pass on to the builder. For the id, since it is a Long, we can use the toString method as we need to pass this as a string. Now that we have set up the map, we can just pass the claims to setClaims
* We now need to define the jwt token life span, we are goint to start by setting the time it was issued ad and the expiry date
* Last requirement is the algorithm, this is what makes the jwt token so trusted, it is signed using the secret key in the signature file. This verifies the integrity of the claims contained in the token. The JWT spec also defines a number of asymmetric signing algorithms (based on RSA and ECDSA). This is a very broad topic and I encourage you to go deeper into it. Just know that this algorithm along with the key is used to secure our token. So in in the signWith method we pass the algorithm we want to user which in this case is HS512 along with the SECRET constant. JWT supports several algorithms for the signature, I know this is a bit blurry right now if you are new to JWT’s.
* Last but not least, we need to use the compact() method. Tokens are designed to be compact
* @Component  
  **public class** JwtTokenProvider {  
    
   **public** String generateToken(Authentication authentication){  
   User user = (User)authentication.getPrincipal();  
   Date now = **new** Date(System.*currentTimeMillis*());  
    
   Date expiryDate = **new** Date(now.getTime() +***EXPIRATION\_TIME***);  
   String userId = Long.*toString*(user.getId());  
    
   Map<String, Object> claims = **new** HashMap<>();  
   claims.put(**"id"**,(Long.*toString*(user.getId())));  
   claims.put(**"username"**, user.getUsername());  
   claims.put(**"fullNme"**, user.getFullName());  
    
    
    
   **return** Jwts.*builder*()  
   .setSubject(userId)  
   .setClaims(claims)  
   .setIssuedAt(now)  
   .setExpiration(expiryDate)  
   .signWith(SignatureAlgorithm.***HS512***,***SECRET***)  
   .compact();  
    
   }

Step #6.12 Generate our JWT token part #2

* We are marching towards our server generating a token if a valid username and password are provided. We still have ways to go to use the token to perform locked operations such as creating a project
* In order to generate a token, we need to wire up that our server can in fact validate a username and a password, 1st step is to go to the SecurityConfig class, we are going to override the configure method from the WebSecurityConfigurerAdapter class that takes the authentication manager builder as a parameter,
* Then we need to pass on to the authenticationmanagerbuilder the userdetailsservice and the passwordencoder we are using. For this we need to 1st bring in our custom detail service and our bcrypt encoder
* @Override  
  **protected void** configure(AuthenticationManagerBuilder authenticationManagerBuilder) **throws** Exception {  
   authenticationManagerBuilder.userDetailsService(**customUserDetailsService**).passwordEncoder(**bCryptPasswordEncoder**);  
  }
* Before we go on to creating our login API we also need to configure the authentication manager which is the instance created by the builder with the credentials. This is the main spring security interface for authenticating a user
* Still in our SecurityConfig file, we are going to override the authentication manager method from the WebSecurityConfigurerAdapter class. We are going to annotate this as a bean and give it a BeanIds AuthManager. BeanId contains globally defined IDs so we can inject it into our controller in order to set up our login endpoint
* @Override  
  @Bean(BeanIds.***AUTHENTICATION\_MANAGER***)  
  **protected** AuthenticationManager authenticationManager() **throws** Exception {  
   **return super**.authenticationManager();  
  }
* Next step before working on the end point is to define a login request object. This is basically to provide structure that we are expecting an object containing a username and a password. Let’s create a package called payload and then create then create the LoginRequest class. Here we only need the getters.
* @NotBlank(message = **"Username cannot be blank"**)  
  **private** String **username**;  
  @NotBlank (message = **"Password cannot be blank"**)  
  **private** String **password**;  
    
  **public** String getUsername() {  
   **return username**;  
  }  
    
  **public** String getPassword() {  
   **return password**;  
  }
* Now let’s go to the UserController and create our login api, first steps we are going to take is to bring in the dependencies we need which are the jwttokenprovider and the authenticcationmanager
* Now we have our dependencies… time to create the api, let’s define the path “/api/users/login” this is a post mapping that returns a generic response entity and takes a valid login request object
* Let’s first make sure that our fields are not blank, we will re use our mapvalidationerror service, exact same impl we have for our register
* Next, we are going to pass our login request to our authentication manager in order to a new UsernamePasswordAuthenticationToken
* So we are going to instantiate an authentication object, make sure we are using org.springframework.security.core.Authentication; anytime you have more than one option, make sure you stick to the one I’m using. If for some reason I forget to give you the heads up in the video and you run into issues, make sure you verify the imports first
* Authentication authentication = **authenticationManager**.authenticate(  
   **new** UsernamePasswordAuthenticationToken(  
   loginRequest.getUsername(),  
   loginRequest.getPassword()  
   )  
  );
* Then we are going to set the authentication in the SecurityContextHolder
* Now we are ready to generate the string containing the jwt token
* SecurityContextHolder.*getContext*().setAuthentication(authentication);  
    
  String jwt = ***TOKEN\_PREFIX*** + **tokenProvider**.generateToken(authentication);  
    
  **return new** ResponseEntity<String>(jwt, HttpStatus.***OK***);
* Now let’s test three things: we cannot pass blank parammeters, we cannot pass invalid parameters, we get a jwt token when we pass valid ones
* Before we test though, we need to go to our security config and make sure our endpoint is stateless, since we are using a token, we do not need a session to be created.
* http.csrf().disable()  
   .exceptionHandling().authenticationEntryPoint(**unauthorizedHandler**).and()  
   .headers().frameOptions().sameOrigin()  
   .and()  
   .sessionManagement()  
   .sessionCreationPolicy(SessionCreationPolicy.***STATELESS***)  
   .and()  
   .authorizeRequests()  
   .antMatchers(**"/api/users/\*\*"**)  
   .permitAll().antMatchers(**"/h2-console/\*\*"**).permitAll()  
   .anyRequest().authenticated();
* Now we run the server and test, and there we have it, our jwt token… we cannot use this yet !!! to trigger secured endpoints suchs as create project because we still need to handle an incoming jwt token

Step #6.13 Custom JWT token json response.

* So, we talked to our react developer and showed him the response from our backend during our demo. He said “I would prefer this in json NOT text and in the following format”

{

"success": true,

"token": "Bearer eyJhbGciOiJIUzUxMiJ9.eyJuYW1lIjoiY2FybG9zIiwiaWQiOiIxIiwiYXZhdGFyIjoiLy93d3cuZ3JhdmF0YXIuY29tL2F2YXRhci9iZWFhNzdlNDBiYTg4N2ZiMjhkMzEwNjQwNjU3ZjY0NT9zPTIwMCZyPXBnJmQ9bW0iLCJleHAiOjE1MzYwOTQ2ODcsImlhdCI6MTUzNjA5MTY4NywiZW1haWwiOiJjZWFnNjhAaG90bWFpbC5jb20ifQ.B97v1MW\_Zc1C5bWgARQfw1QeZZfQ529oWUcjh7IXEcBGoyFYSrC-Yd9oEsWH0BhstscI3LnWm5mgJTuu5eM09w"

}

* This shouldn’t be a problem for you at this stage, please pause the video and try to generate a response like this yourself.
* On the payload package, please create a JWTLoginSuccessResponse class
* We are going to set this up with two attributes Boolean success and a String token
* Now we are going to create constructor where we pass both attributes
* Set getters and setters
* Override the toString method
* Now let’s go to our UserControllr and update the return statement as follows
* **return** ResponseEntity.*ok*(**new** JWTLoginSuccessResponse(**true**,jwt));
* let’s try it, success

Step #6.14 JWT Authentication filter - Prework

* We are almost over the hump, now we need to set up our filter in order for us to be able to user the token we are generating to access secured resources but first, we are going to do some work
* If you remember in the Project Controller we added the annotation @CrossOrigin. We are going to do some work for the future and actually Enable CrossOrigin for our entire backend. This will give us flexibility when we create other secured paths. Erase this from the Project controller, then in the “security” package, create a class called WebMvcConfig
* We are going to annotate this class as @Configuration and we are also going to implement the WebMvcConfigurer interface
* Now all we need to do is but override 1 method: addCorsMappings
* @Override  
   **public void** addCorsMappings(CorsRegistry registry) {  
   registry.addMapping(**"/\*\*"**);  
   }  
  }
* Piece of cake – unfortunately we cannot test this right this second but take my word for it, it will work
* In our road to use the jwt we need to enable the provider to validate the token’s format and extract the user id from it. So we need to implement two important changes in our token provider
* In the jwttokenprovider class we are goint to start by creating a method called validateToken. This this method is a try catch block that parses the token string using the secret key and throws specific exceptions depending on that is wrong with the token if it cannot be parsed properly. What out that the parseClaimsJws method you are using is the jws not jwt. The main difference is that jwt returns the header and the claims, and jws only returns the claims which is what we need right now
* Then we are going to bring in custom exceptions from jsonwebtoken
* **public boolean** validateToken(String authToken){  
   **try**{  
   Jwts.*parser*().setSigningKey(***SECRET***).parseClaimsJws(authToken);

**return true**;  
 }**catch** (SignatureException ex) {  
 System.***out***.println(**"Invalid JWT signature"**);  
 } **catch** (MalformedJwtException ex) {  
 System.***out***.println(**"Invalid JWT token"**);  
 } **catch** (ExpiredJwtException ex) {  
 System.***out***.println(**"Expired JWT token"**);  
 } **catch** (UnsupportedJwtException ex) {  
 System.***out***.println(**"Unsupported JWT token"**);  
 } **catch** (IllegalArgumentException ex) {  
 System.***out***.println(**"JWT claims string is empty."**);  
 }  
 **return false**;  
}

* Last step we are going to take in this video before starting to work on the authorization filter is to extract the userid from the token, once again, this is required for our jwt auth filter to identify the user, kind of like when the details service verifies that it is an actual user in the db, the jwtfilter also needs to know that the token is from a valid user
* For this we are going to create a method called getUserIdFromJWT. Now we are going to grab the claims from the token, as you can see there, we are passing it as a string
* Then let’s put the string id in a variable id and return it using the parseLong method in the Long wrapper
* **public** Long getUserIdFromJWT(String token){  
    
   Claims claims = Jwts.*parser*().setSigningKey(***SECRET***).parseClaimsJws(token).getBody();  
   String id = (String)claims.get(**"id"**);  
     
   **return** Long.*parseLong*(id);
* Now that we are done with this prework, let’s start working on the Filter

Step #6.15 JWT Authentication filter

* In the security package, we are going to create a JwtAuthenticationFilter
* What is a filter: we need to create a filter to inspect requests that we want authenticated. The filter’s job is to confirm that the request has a valid token, and set the corresponding Authentication. Since we only want this to run once per request we need to extend the [OncePerRequestFilter](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/filter/OncePerRequestFilter.html) class to set this up. OncePerRequestFilter. This is a filter base class that guarantees a single execution per request dispatch on any servlet container
* Now we need to implement method the do filter internal method
* Before we start customizing our filter, we need to be able to extract our jwt from the servletrequest, for this we are going to create a getJwtFromRequest method that returns a string and takes httpservelet request as a parameter. And extracts the header containing the jwt token
* **private** String getJwtFromRequest(HttpServletRequest request){  
   String bearerToken = request.getHeader(***HEADER\_STRING***);  
    
   **if**(StringUtils.*hasText*(bearerToken) && bearerToken.startsWith(***TOKEN\_PREFIX***)){  
   **return** bearerToken.substring(7, bearerToken.length()); //This is basic java folks, if your are unfamiliar with substring, go back to your basics and revie this  
     
   }  
     
   **return null**;  
   }  
  }
* now let’s inject our token provider and our customuserdetailsservice
* So back to the internal filter now, we are going to set this up as a try catch

Step #6.16 JWT Authentication filter Test

* Now that we have our filter, it is time to use it in our security config so that requests can be authenticated via jwt token
* Let’s do some clean up
* Lets bring the authentication filter as a bean
* @Bean  
  **public** JwtAuthenticationFilter jwtAuthenticationFilter(){  
   **return new** JwtAuthenticationFilter();  
  }
* So let’s go to the Security Config file at the end of the configure http we are going to add the following line

http.addFilterBefore(jwtAuthenticationFilter(),UsernamePasswordAuthenticationFilter.**class**);

* This basically enables spring security to use the jwt filter rather than expecting credentials
* Bug:
* **public boolean** validateToken(String authToken){  
   **try**{  
   Jwts.*parser*().setSigningKey(***SECRET***).parseClaimsJws(authToken);  
   **return true**; forgot return true so the if in the filter failed
* Everything works

Step #7 Register and Login with React/Redux

Step #7.1 Create Register and Landing components

* In the layout folder we are going to create a register and login component in the User folder (we already have the landing component.
* Then we are going to copy paste templates for both and of course, update it to jsx (class => classnames
* Now that we have done this, let’s go to our App.js file and create routes for registering and login in and test our views
* Oops let’s update the landing page, too, sweet, no let’s update the route flow between landing and registration and landing and login, in the Landing component, let’s make sure that those two buttons point to where the need to
* First we need to import Link in order to replace the a tags

import { Link } from "react-router-dom";

* Replace a tags for link and hrefs by “to”
* Lets now do the same in the Header Component
* Now that it is all updated let’s try it
* Ok so it all works!

Step #7.2 Let’s enable registering

* Let’s go to our Register component, and set up the component state. Again, this is going to be a controlled component. For this we need to add a constructor in the Register component
* In this constructor we are going to add the initial state of our fields which in this case will be blank, as always, a simple way to make sure that our front end and back end are aligned is to literally grab these attributes from the server response. This might seem mighty tedious to you, I apologize but sometimes this additional upfront time investment pays out when you don’t have to invest twice the time debugging potential bugs

class Register extends Component {

constructor() {

super();

this.state = {

username: "",

fullName: "",

password: "",

confirmPassword: "",

errors: {}

};

}

* Now, we need the name attribute in our input to match these fields. Let’s just take care of this now
* In the constructor, just like we did when we worked on the addProject component, we need an errors object
* Now let’s wire the state of our input fields to the value, which as you might remember, will lock down the input, let’s try again and now we can’t type

name="fullName"

value={this.state.fullName}

* Now, just like we did in our previous forms, we are going to create the onChange function… I sincerely hope that at this point, you are confident enough to stop the video and wire up the on change on your own. Anyway on each fill we will add the onChange={this.onChange} line
* Now, right above the render method, lets create the onchage function, this one takes an event parameter. As you remember what we want to do is that everytime the user types, it calls this function that sets a new state in the state variables. This on change will be identical as the one we wired up before
* Last step is to bind this in the constructor just like we did before

this.onChange = this.onChange.bind(this);

* I’m sure you realized that we are going to start moving a bit faster as we are pretty much repeating steps here.
* Now let’s check the console, make sure we have no errors and also let’s see again how it changes state in the react tool when we are on the Register component

Step #7.3 registration form Happy Path

* First we need to create an onSubmit function that will fire off when the form is submitted
* Just like before we are going to create the function and bind it in the constructor. The only reason I keep reminding you like a broken record is because I really want to jog your memory and give you the confidence to knock repetitive stuff on your own, get comfortable with react

this.onSubmit = this.onSubmit.bind(this);

* Our onSubmit function takes an event parameter, and like before, we are going to prevent the default submit behaviour.
* Our onSubmit builds and object that we are going to pass to the backend. This object gets filled out by the current state of the input fields
* As always let’s console log it first before submitting it to our API – works!
* Guys, you should be confident enough to do this on your own for the Login form, at least get it to this state where you can console log the username and password

Step #7.4 registration Form Happy Path API

* 1st thing we are going to do is start wiring redux, so we are going to start by creating a reducer for all things related to registration and authentication. We are going to call this reducer authReducer.
* As with previous reducers, we are going to set up the initial state

const initialState = {

isAuthenticated: false,

user: {}

};

export default function(state = initialState, action) {

switch (action.type) {

default:

return state;

}

}

* Then we need to bring this into our root reducer

import { combineReducers } from "redux";

import projectReducer from "./projectReducer";

import errorReducer from "./errorReducer";

import authReducer from "./authReducer";

export default combineReducers({

project: projectReducer,

errors: errorReducer,

auth: authReducer

});

* Now, we need to create an actions file for auth related operations, we will do this in the actions folder and will call it “authActions.js”
* To start wiring our registration operation we are going to create the registerUser action, for this we need to first import axios

import axios from "axios";

* Once we import this, we have enough to code our happy path. This is also similar to the create project action. But let’s got step by step so it is nice and clear

export const createNewUser = (newUser, history) => async dispatch => {

const res = await axios.post(

"http://localhost:8080/api/users/register",

newUser

);

history.push("/login");

};

* Now in our register component, we need 3 imports 4 imports:

import { createNewUser } from "../../actions/authActions";

import PropTypes from "prop-types";

import { connect } from "react-redux";

import classnames from "classnames";

* Next thing we are going to do in our Register component is to map our auth and error state to our props, reason being is because there is going to be logic directly associated to whether or not the user is authenticated. In this case, in the future we will not allow a user to register if he or she is logged in. so let’s mapour state to props. Also, we want to display validation errors in the form just like we did for the project creation

const mapStateToProps = state =>({

auth: state.auth,

errors: state.errors

})

* Next, we want to connect our component to the store and pass the action that we are using

export default connect(mapStateToProps, {createNewUser})(Register);

* Last but not least, let’s set up our propTypes

Register.propTypes = {

createNewUser: PropTypes.func.isRequired,

auth: PropTypes.object.isRequired

};

* Now that we have this, let’s update our on submit function to call the action that calls the API
* Lets test this now, and it works, all good with the happy path.

Step #7.5 registration Form handling errors

* Now we want to handle errors coming from the server when we fail to pass a valid user object. Let’s first see what is happening right now when we pass an invalid object. Funny, we still have the html validation, let’s get rid of it. As you can see, we get the 400 from the server but we do not get a meaningful error.
* First thing we want to do is get the errors from the state. As you remember we have an errors reducer that we can reuse for this purpose. So the first step is to update our createuser action in the authactions file and surround it by a try catch

export const createNewUser = (newUser, history) => async dispatch => {

try {

const res = await axios.post(

"http://localhost:8080/api/users/register",

newUser

);

history.push("/login");

} catch (err) {

dispatch({

type: GET\_ERRORS,

payload: err.response.data

});

}

};

* So now that we have updated our action, we can see it in our state.
* Next thing is to get these errors into the component. In order to pass these props into the Register component, we will use the componentWillReceiveProps lifecycle hook. As you remember, it takes next props as a parameter
* Then we are going to check if these props contain errors and if there are errors present, we are going to set the state of our errors object

componentWillReceiveProps(nextProps) {

if (nextProps.errors) {

this.setState({ errors: nextProps.errors });

}

}

* Next we need to get errors from the state right in the render function
* Now we are going to use classnames to set up our error display, let’s do the 1st attribute together and then you’ll knock down the rest on your own.

<div className="form-group">

<input

type="text"

className={classnames("form-control form-control-lg", {

"is-invalid": errors.fullName

})}

placeholder="Full Name"

name="fullName"

value={this.state.fullName}

onChange={this.onChange}

/>

{errors.fullName && (

<div className="invalid-feedback">{errors.fullName}</div>

)}

</div>

Step #7.6 Login Action

* We are going to now start working to enable the workflow that we have in our server. We get a token with valid credentials and pass the valid credential with our requests whenever we are trying to access a protected route. For this we are going to keep our tokens in local storage and send them along with the action request to the server. We are also going to set up a logout to clear such token from the store
* We are going to start by creating a login action in the authActions file

export const login = LoginRequest => async dispatch => {

const res = await axios.post(

"http://localhost:8080/api/users/login",

LoginRequest

);

const { token } = res.data;

localStorage.setItem("jwtToken", token);

setAuthToken(token);

};

* Now we are going write a function to grab the token and set it to the header of every request whenever we are logged in. This is something that is fairly straight forward with Axios. This function will attach the token to the authorization header of our requests

import axios from "axios";

const setAuthToken = token => {

if (token) {

axios.defaults.headers.common["Authorization"] = token;

} else {

delete axios.defaults.headers.common["Authorization"];

}

};

export default setAuthToken;

* Now the next thing we want to do is to decode the token we are getting. Remember,the token has claims which is important information from the user that we need to set user. For us to enable this, we need a package called jwt decode: npm install jwt-decode
* Now let’s import it in our authActions file

import jwt\_decode from "jwt-decode";

* Then we need to create a function called setCurrentUser. In preparation for this we are going to create a type SET\_CURRENT\_USER and import it to our authActions file

import { GET\_ERRORS, SET\_CURRENT\_USER } from "./types";

export const setCurrentUser = decoded => {

return {

type: SET\_CURRENT\_USER,

payload: decoded

};

};

* Now that we have our function, we need to update our login function (happy path edition) as such

export const login = LoginRequest => async dispatch => {

const res = await axios.post(

"http://localhost:8080/api/users/login",

LoginRequest

);

const { token } = res.data;

localStorage.setItem("jwtToken", token);

setAuthToken(token);

const decoded = jwt\_decode(token);

dispatch(setCurrentUser(decoded));

};

* Now we need to update our authReducer to include this case
* Another requirement that we have is to determine if the payload coming from the SET\_CURRENT\_USER type is empty or not. If it is not empty, that means we have a valid token. What we are going to do before we start wiring the reducer is to create a function that determines if an object is empty. I learn this trick from one of the many amazing React tutorials I took when I was learning the technology

const isEmpty = value =>

value === undefined ||

value === null ||

(typeof value === "object" && Object.keys(value).length === 0) ||

(typeof value === "string" && value.trim().length === 0);

export default isEmpty;

* Now let’s update the reducer

export default function(state = initialState, action) {

switch (action.type) {

case SET\_CURRENT\_USER:

return {

...state,

isAuthenticated: !isEmpty(action.payload),

user: action.payload

};

default:

return state;

}

}

Step #7.7 Login Form – Controlled Component

* We need to turn our login form into a controlled component. I really want you to do this, just like we did for register, make it a controlled form and console log the login request
* 1st we need a constructor that sets an initial state for our input fields

constructor() {

super();

this.state = {

username: "",

password: ""

};

}

* Then we need to set the value in the input fields to what’s on the state. We should again notice that we cannot input anything on to the fields

<input

type="email"

className="form-control form-control-lg"

placeholder="Username / Email Address"

name="username"

value={this.state.username}

/>

* Now we need to create the onChange function

onChange(e) {

this.setState({ [e.target.name]: e.target.value });

}

* Then let’s bind this at the constructor level

this.onChange = this.onChange.bind(this);

* Then let’s update our input fields for the onChange event to be fired up everytime the user types a new value. Now we are able to type again
* Next step is to console log the login request when we submit the form. For this we are going to create an onSubmit function that too takes an event parameter. As we did before we want to prevent the default submit action. Then we want to create the login request object. The object parameters take the current state of the fields

onSubmit(e) {

e.preventDefault();

const LoginRequest = {

username: this.state.username,

password: this.state.password

};

console.log(LoginRequest);

}

* Then we need to bind this at the constructor level

this.onSubmit = this.onSubmit.bind(this);

* Then replace the form action with

<form onSubmit={this.onSubmit}>

* That’s all she wrote

Step #7.8 Login Form functionality – happy path

* Now we are going to bring in a few imports so we can wire up our component

import PropTypes from "prop-types";

import { connect } from "react-redux";

import classnames from "classnames";

import { login } from "../../actions/authActions";

* Now that we have these imports, lets connect our component to the redux store. For now we are going to pass a null parameter in lieu of mapstatetoprops and the login action
* export default connect(null, {login})(Login);
* let’s now set up our mapStateToProps

const mapStateToProps = state => ({

auth: state.auth,

errors: state.errors

});

export default connect(

mapStateToProps,

{ login }

)(Login);

* And then let’s set up the PropTypes for our component

Login.propTypes = {

login: PropTypes.func.isRequired,

auth: PropTypes.object.isRequired,

errors: PropTypes.object.isRequired

};

* Component will receive props: we will do it here because we want

if (nextProps.auth.isAuthenticated) {

this.props.history.push("/dashboard");

}

* Ok, now we are ready to test our happy path, so we need to pass the loginrequest onto our action right in the onSubmit function. As you can see we get the authorization in the state as well as the token in our local storage. Now that we are logged in, any request we make will have the auth header.

Step #7.9 Handle page reload

* Now that we have a semi-functional login action, we need to handle what happens when the page reload. When it does, the information that we have in the state goes away, however, our token remains in the localstorage. In order to avoid this behaviour, we are going to go to App.js and set up a condition to help with this
* First let’s import jwt-decode and the setAuthToken utility in app.js

import jwt\_decode from "jwt-decode";

import setAuthToken from "./utility/setAuthToken";

* Let’s also bring in “setCurrentUser”

import { setCurrentUser } from "./actions/authActions";

* Now let’s create a conditional that on any request that goes through our main routing app, it checks whether or not there’s a token and if there is, it sets the user in our state

const jwtToken = localStorage.jwtToken;

if (jwtToken) {

setAuthToken(jwtToken);

const decoded\_jwtToken = jwt\_decode(jwtToken);

store.dispatch(setCurrentUser(decoded\_jwtToken));

}

class App extends Component {

* And as you can see, we maintain our user in the state.

Step #7.10 Handle exceptions.

* This should be piece of cake for y’all, please try to do this on your own and comeback to compare notes
* First let’s update our login action to handle exceptions with a nice try/catch block

export const login = LoginRequest => async dispatch => {

try {

const res = await axios.post(

"http://localhost:8080/api/users/login",

LoginRequest

);

const { token } = res.data;

localStorage.setItem("jwtToken", token);

setAuthToken(token);

const decoded = jwt\_decode(token);

dispatch(setCurrentUser(decoded));

} catch (err) {

dispatch({

type: GET\_ERRORS,

payload: err.response.data

});

}

};

Now our errors are in our state

* Next thing is to get the errors in our error object. For this, you guessed it, we will use “Componentwillreceiveprops”

componentWillReceiveProps(nextProps) {

if (nextProps.errors) {

this.setState({ errors: nextProps.errors });

}

if (nextProps.auth.isAuthenticated) {

this.props.history.push("/dashboard");

}

}

* Now that we set this up, lets then destructure it out of our state in case we have errors
* Now we want to display the errors from the server. At this point you should be more than comfortable doing this on your own. Check the code from the register component and force yourself to get comfortable with this

className={classnames("form-control form-control-lg", {

"is-invalid": errors.password

})}

* Right after input

{errors.password && (

<div className="invalid-feedback">{errors.password}</div>

)}

Step #7.11 Handle Logout.

* To handle logout, the first thing that we are going to wire in is our logout action. Our action will remove the token from storage, set authentication to false and empty the current user object

export const logout = history => dispatch => {

localStorage.removeItem("jwtToken");

setAuthToken(false);

dispatch(setCurrentUser({}));

};

* Now we are going to change the header to make it more dynamic. For this we are going over to our Header component and import

import PropTypes from "prop-types";

import { connect } from "react-redux";

import { logout } from "../../actions/authActions";

* We want to have a different set of links depending on our auth state, for this we need to map the state to props so we can bring in the auth state

const mapStateToProps = state => ({

auth: state.auth

});

* Then let’s connect our component to the store

export default connect(mapStateToProps, {logout}) (Header);

* Then let’s define the proptypes

Header.propTypes = {

logout: PropTypes.func.isRequired,

auth: PropTypes.object.isRequired

};

* Now, what we want is to change the header options based on whether or not the user is logged in. So, using destructuring, we are going to extract “isAuthenticated” and “user” from this.props.auth
* Then we are going to create two jsx components, one that contains links for authenticated users and one that contains links for guests

const userIsAuthenticated = (

<div className="collapse navbar-collapse" id="mobile-nav">

<ul className="navbar-nav mr-auto">

<li className="nav-item">

<Link className="nav-link" to="/dashboard">

Dashboard

</Link>

</li>

</ul>

<ul className="navbar-nav ml-auto">

<li className="nav-item">

<Link className="nav-link " to="/register">

{user.fullName} Carlos Arosemena

</Link>

</li>

<li className="nav-item">

<Link className="nav-link" to="/logout">

Logout

</Link>

</li>

</ul>

</div>

);

const guestUser = (

<div className="collapse navbar-collapse" id="mobile-nav">

<ul className="navbar-nav ml-auto">

<li className="nav-item">

<Link className="nav-link " to="/register">

Sign Up

</Link>

</li>

<li className="nav-item">

<Link className="nav-link" to="/logout">

Login

</Link>

</li>

</ul>

</div>

);

* Now let’s try it real quick
* Now that we have tried it we are going to create a let headerLinks and then will pass links there depending on whether or not the user is authenticated

let headerLinks;

if (isAuthenticated && user) {

headerLinks = userIsAuthenticated;

} else {

headerLinks = guestUser;

}

* Good, so now let’s enable our logout button. For this we are going to create a function, logout (1st, we need to import the logout action, ) and then we need to create the click function and bind it to the logout element. Here I personally want the default refresh behaviour. I’m also going to force the redirect to the landing page. The reason we cannot use history here as we have for components such as register, is because the prop is passed by router when the component is rendered by the router. In this case it wasn’t and this is the reason we are going to use some good ol’ JS to redirect to the landing page

logout() {

this.props.logout();

window.location.href = "/";

}

* Update the logout link as suchs

<li className="nav-item">

<Link

className="nav-link"

to="/logout"

onClick={this.logout.bind(this)}

>

Logout

</Link>

</li>

* Let’s also update the user name with a font awesome icon
* <li className="nav-item">

<Link className="nav-link " to="/register">

<i className="fas fa-user-circle mr-1" />

{user.fullName}

</Link>

</li>

Step #7.12 Handle token expiration

* Moving on, we want to handle what to do when the token expires, first let’s check our server and review how long will our token be valid for. As you can see, our token is valid for 8 hours right now. You can make this value whatever you need it to be, just make sure you are aligned with your security team when doing this for production apps. For now in our server, we are going to make it 30 seconds
* So back to our App.js file we are going to update our token logic to account for expired tokens. This is another little trick I learned in one of the awesome learning materials I used to teach myself react. Ain’t no shame in admitting stuff like this. So in the if statement we are going to add an additional condition, first we need to define the currentTime and then we need to check if the token’s exp is less than current time. (remember time keeps moving forward so the past will always be less.) then we want to dispatch the logout action and it works

Step #7.13 Logic when logged In

* We are going to start implementing logic for when we are logged in. First thing we are going to do is prevent the user from: Registering and Login in while authenticated. First we are going to to the login component and use the lifecycle method componentDidmount
* Within cdm we are going to set a condition where if the user is authenticated, we will just redirect to the main dashboard view, works, let’s do it for register

componentDidMount() {

if (this.props.auth.isAuthenticated) {

this.props.history.push("/dashboard");

}

}

* You can do this for the landing page if you’d like, the same behaviour.

Step #7.14 Private Routes

* At this very moment, we have secured our application, however, users can still go to routes that supposedly are only available for users that logged in. Let’s inventor these routes.
* A way to do this IF you only have a hand full of routes is at mount. Let’s try it with the Dashboard component. First lets update our mapstatetoprops and our prop types so we can get the auth state

Dashboard.propTypes = {

project: PropTypes.object.isRequired,

getProjects: PropTypes.func.isRequired,

auth: PropTypes.object.isRequired

};

const mapStateToProps = state => ({

project: state.project,

auth: state.auth

});

* Now that we have this, we can use the lifecycle hook “component did mount” and check whether or not the user is authenticated, if it’s not, redirect the user to the login page

componentDidMount() {

if (!this.props.auth.isAuthenticated) {

this.props.history.push("/login");

}

this.props.getProjects();

}

We can do it this way, but this will become very very tedious and potentially buggy for larger projects. Another way to do it is to create a private route. Big shout out to Brad Traversy, it was by checking out his course that I learned about this

* Let’s go to our utility folder and create a file called SecuredRoute.js
* This is going to be a functional based component so we can use the rfc shortcut
* Let’s start by importing everything we need to make this happen

import React from "react";

import { Route, Redirect } from "react-router-dom";

import { connect } from "react-redux";

import PropTypes from "prop-types";

const SecuredRoute = ({ component: Component, auth, ...otherProps }) => (

<Route

{...otherProps}

render={props =>

auth.isAuthenticated === true ? (

<Component {...props} />

) : (

<Redirect to="/login" />

)

}

/>

);

SecuredRoute.propTypes = {

auth: PropTypes.object.isRequired

};

const mapStateToProps = state => ({

auth: state.auth

});

export default connect(mapStateToProps)(SecuredRoute);

* Now let’s import this on App.js
* Now let’s replace Route with SecuredRoute and clean up Dashboard
* So now that we are going to try it, we will notice some weird behaviour… After doing some digging, what solved it is to Wrap all private route with Switch. This prevents these strange redirect behaviours. And all good!!

Step #8 set User / Project relationship

* So we are done with the most difficult part of our project, no2 it is time to add some useful logic to our backend. If we log in with two different accounts, you’ll notice that both users see the same project list. This is not WRONG, but our use case is that this is a tool for personal projects. So right now we are working on the premise that one user can own several projects, but one project can only have 1 user. Please note that in a future version of this app, We will update it so a project can have one owner and several users so it can serve small teams, say 5 of your best friends and you are creating an app. For now the scope is for personal projects.
* First thing we are going to do is to set up a one to may relationship between a user and a project
* The owning side of the relationship will obviously be the user so we are going to update the user object. Basically, each user will have a list of one or more projects. Tip, everytime you have a collection as an attribute to an object, make sure you initialize it!! I have ran into strange issues for not doing so.

**private** List<Project> **projects** = **new** ArrayList<>();

* Next we are going to create getters and setters
* And finally we are going to set the OneToMany relationship
* @OneToMany(  
   cascade = CascadeType.***ALL***, *//* fetch = FetchType.***EAGER***,  
   mappedBy = **"user"**)
* This basically means that if we delete the user, we delete all projects associated with the user AND whenever we load the user, we load all the projects. This is a small app so for now we can get away with this. There are better ways to do this but remember this is a prototype!
* Now that we have this, we want to enable a route where we find projects depending on the user. Let’s start with our project repository
* Iterable<Project>findAllByUserUsername(String user);
* We need to update the save operation controller and service layer in order to set the user for a given project, this is one way of doing it:

Service controller method takes the Principal parameter and passes the username to the service

* **public** ResponseEntity<?> createNewProject(@Valid @RequestBody Project project, BindingResult result, Principal principal){  
    
   ResponseEntity<?> errorMap = **mapValidationErrorsService**.MapValidationErrorsService(result);  
   **if**(errorMap != **null**) **return** errorMap;  
    
   Project newProject = **projectService**.saveProject(project, principal.getName());  
    
   **return new** ResponseEntity<Project>(newProject,HttpStatus.***CREATED***);  
  }
* Then we autowire the user repo in the project service and set the user for the project
* **public** Project saveProject(Project project, String username) {  
   **try**{  
   User user = **userRepository**.findByUsername(username);  
   project.setUser(user);  
   project.setProjectId(project.getProjectId().toUpperCase());  
   **return projectRepository**.save(project);  
   }**catch** (Exception e){  
   **throw new** ProjectIdAlreadyExistException(**"Project ID already exists"**);  
    
   }  
    
  }
* We need to @JsonIgnore the user attribute in the project class
* Then we need to create a new method in the projectservice layer
* **public** Iterable<Project> findAllProjectsByUser(String username){  
    
   **return projectRepository**.findAllByUserUsername(username);  
  }

and now let’s create the end point

* @GetMapping(**"/myProjects"**)  
  **public** Iterable<Project> getMyProjects(Principal principal){  
   **return projectService**.findAllProjectsByUser(principal.getName());  
  }
* Now let’s test this thing in postman
* Now let’s update the projectAction in react

export const getProjects = () => async dispatch => {

const res = await axios.get("http://localhost:8080/api/project/myProjects");

dispatch({

type: GET\_PROJECTS,

payload: res.data

});

};

* All is great!! Now let’s try accessing a project from another user account

Step #8.1 Improve User / Project relationship

* Right now, there are a few things that don’t look good. 1st and it is the fact that although each user can only see their own projects, the can view alter and delete other people’s projects. What we are going to do is change the api to prevent this behaviour
* We will change both the controller and the service layer and the project object. So we are going to add an additional parameter to the project object called projectOwner + getters and setters
* Then we are going to update the service layer, changes are: the method takes a username parameter, finds the user by username, sets the project user, sets the projectOwner string
* **public** Project saveProject(Project project, String username) {  
    
   ProjectOwnerOnly(project, username);  
    
   **try**{  
   User user = **userRepository**.findByUsername(username);  
   project.setUser(user);  
   project.setProjectLeader(user.getUsername());  
   project.setProjectId(project.getProjectId().toUpperCase());  
   **return projectRepository**.save(project);  
    
   }**catch** (Exception e){  
   **throw new** ProjectIdAlreadyExistException(**"Project ID already exists"**);  
    
   }  
    
  }
* we are also creating an algorithm that checks if the payload has a id. Remember, jpa allows us to pass the id of an existing object and updates it if it exists, if not it creates a new one. We are going to change that behaviour to only update if the id exists
* **private void** ProjectOwnerOnly(Project project, String username) {  
   **if**(project.getId()!=**null**){  
   Project existingProject = **projectRepository**.findByProjectId(project.getProjectId());  
    
   **if**(existingProject !=**null** && (existingProject.getUser().getUsername()!=username)){  
   **throw new** ProjectNotFoundException(**"Project not found in your account"**);  
   }**else if**(existingProject == **null**){  
   **throw new** ProjectNotFoundException(**"No project found to update, please create a project"**);  
    
   }  
    
   }  
  }
* now that we have this in place, our createNewProject method remains the same

Step #8.2 Improve User / Project relationship

* Now that we know that a user cannot update a project that doesn’t below to him, let’s make sure the user cannot get by id or delete unless he/she is the leader
* Let’s start with get by id, lets update the controller layer
* *//Get an existing Project by Id*@GetMapping(**"/{projectId}"**)  
  **public** ResponseEntity<?> getProjectById(@PathVariable String projectId, Principal principal){  
   Project project = **projectService**.findByProjectId(projectId, principal.getName());  
    
   **return new** ResponseEntity<Project>(project, HttpStatus.***OK***);  
    
  }
* Now let’s update the service layer
* **public** Project findByProjectId(String projectId, String username){  
    
   Project project = **projectRepository**.findByProjectId(projectId);  
    
   **if**(project == **null**){  
   **throw new** ProjectNotFoundException(**"Project not found"**);  
   }  
     
   **if**(project.getProjectLeader()!= username){  
   **throw new** ProjectNotFoundException(**"Project not found in our account"**);  
   }  
    
   **return** project;  
  }
* let’s test it, good
* So now let’s do the same thing for delete. I am repeating myself on purpose, it will be your challenge to abstract this in a service or a private method that handles these cases without having to be so redundant … SO THAT WORKED

Step #8.3 Improve User / Project relationship REACT

* Let’s handle the Update UI, so if I try to hit the url with the code, good news is that it won’t load the project for an unauthorized user BUT still renders the form, we want to prevent that.
* 1st, our action get project, we need to wrap it in a try catch block. Now we get the error from the server in the state.
* An easy way to handle this is that if there are errors from the server when getting the id, just redirect back to the dashboard. For this we just need to update our getproject action to take the history parameter and then push to dashboard

export const getProject = (id, history) => async dispatch => {

try {

const res = await axios.get(`http://localhost:8080/api/project/${id}`);

dispatch({

type: GET\_PROJECT,

payload: res.data

});

} catch (err) {

history.push("/dashboard");

// dispatch({

// type: GET\_ERRORS,

// payload: err.response.data

// });

}

};

* Changes in UpdateProject
* componentDidMount() {
* const { id } = this.props.match.params;
* this.props.getProject(id, this.props.history);
* }
* We do not need to do this we delete

Step #9 Create the backlog

* Each project is going to have its own backlog and each backlog will have a set of product backlog items. 1st thing we are going to do is create a backlog object. We will annotate this as an entity. A Backlog needs the following attributes

|  |  |  |  |
| --- | --- | --- | --- |
| Field | Type | Validation | ORM |
| id | Long | @Id | @GeneratedValue(strategy = GenerationType.***IDENTITY***) |
| Integer | PBISequence | Starts = 0 |  |
| projectId | String |  |  |
| project | Project | OneToOne |  |
|  |  |  |  |

* Empty constructor
* Getters and setters
* Now we need to establish the relationship between these two classes, our use case will be as follow: One project has 1 backlog, 1 backlog belongs to one project. I want to clarify something, we could have the items directly into our project and each project would have a set of items. What I don’t like about that design is that every time we load a project, we’d be loading all the tickets or PBIs associated witht the project. Say we have thousands of entries, why would you want to load that when you load the project objects… this way we can have a separate call to load the backlog only when we want to see the project board
* So let’s set the relationship in the backlog object
* @OneToOne(fetch = FetchType.***LAZY***)  
  @JoinColumn(name=**"project\_id"**, nullable = **false**)  
  @JsonIgnore *// As you might remember from when we set the relationship between User and Project***private** Project **project**;
* Now let’s set the relationship in the project object
* @OneToOne(fetch = FetchType.***LAZY***, cascade = CascadeType.***ALL***, mappedBy = **"project"**)  
  **private** Backlog **backlog**;
* Alright, now that we have the object and the relationship, we need instantiate the backlog. In this design, since it is a OneToOne, we will instantiate upon project creation. For this, we will update the projectservice.saveorupdate method as follows:
* **public** Project saveProject(Project project, String username) {  
    
   ProjectOwnerOnly(project, username);  
    
   **try**{  
   User user = **userRepository**.findByUsername(username);  
   project.setUser(user);  
   project.setProjectLeader(user.getUsername());  
   project.setProjectId(project.getProjectId().toUpperCase());  
   Backlog backlog = **new** Backlog();  
   project.setBacklog(backlog);  
   backlog.setProject(project);  
   backlog.setProjectIdentifier(project.getProjectId().toUpperCase());  
   **return projectRepository**.save(project);  
    
   }**catch** (Exception e){  
   **throw new** ProjectIdAlreadyExistException(**"Project ID already exists"**);  
    
   }  
    
  }
* Bug, projectId conflicts with the project\_id column needed for the onetoone relationship, updated it to **private** String **projectIdentifier**;
* Looks good, now if we delete our project, the backlog should go away
* Beautiful

Step #9.1 Create the backlog end point

* Now we want to create an end point that loads our backlog, including backlog items, allows us to CRUD a backlog item, so in the “web” package we are going to create the BacklogController class. We are going to give it the “/api/backlog” mapping and annotate it as a RestController and Enable CrossOrigin
* Before we create the 1st method in the controller, let’s enable our backend first. In the backlogrepository, we are going to enable a backlog to be found by project identifier

Backlog findByProjectIdentifier(String Identifier);

* Then we are going to create a backlog service class and annotate it as service
* Then we are going to autowire our backlog repository
* Then in the backlogservice let’s create our method
* @Autowired  
  **private** BacklogRepository **backlogRepository**;  
    
  **public** Backlog findBacklogByProjectId(String identifier){  
   **return backlogRepository**.findByProjectIdentifier(identifier);  
  }
* Now let’s autowire the service to our backlog controller, then let’s create our method
* @GetMapping(**"/{projectId}"**)  
  **public** ResponseEntity<Backlog> getProjectBacklog(@PathVariable String projectId){  
   **return new** ResponseEntity<Backlog>(**backlogService**.findBacklogByProjectId(projectId), HttpStatus.***OK***);  
  }
* So it works BUT if we try to access the backlog from a different account we can, this is of course, unacceptable. We will solve this in the next video

Step #9.1 Check user service

* So we are about to repeat ourselves again. I trust that you took on the challenge of abstracting the repeated code. Let me show you my solution. 1st let’s clarify where we needed. We need it on all except the save or update.. why because in all instances, we are working out of our custom project id except for the save where we are passing the actual project object, although we could refactor that, Im going to leave that one alone. Because it is a bit more complex. So we are going to create a generice service for this called CheckProjectOwnerService. We will annotate this as a service and will inject the projectrepository and Backlog repository
* @Service  
  **public class** CheckProjectOwnerService {  
    
   @Autowired  
   **private** ProjectRepository **projectRepository**;  
    
   **public** Project PROJECT\_checkProjectOwner(String projectId, String username){  
    
   Project project = **projectRepository**.findByProjectId(projectId);  
    
   **if**(project.getProjectLeader()!= username){  
   **throw new** ProjectNotFoundException(**"Project not found in our account"**);  
   }  
   **if**(project == **null**){  
   **throw new** ProjectNotFoundException(**"Project not found"**);  
   }  
    
   **return** project;  
    
   }  
    
    
   **public void** ProjectOwnerOnly(Project project, String username) {  
   **if**(project.getId()!=**null**){  
   Project existingProject = **projectRepository**.findByProjectId(project.getProjectId());  
    
   **if**(existingProject !=**null** && (existingProject.getUser().getUsername()!=username)){  
   **throw new** ProjectNotFoundException(**"Project not found in your account"**);  
   }**else if**(existingProject == **null**){  
   **throw new** ProjectNotFoundException(**"No project found to update, please create a project"**);  
    
   }  
    
   }  
   }
* New Project Service
* @Autowired  
   **private** CheckProjectOwnerService **checkProjectOwnerService**;  
    
   **public** Project saveProject(Project project, String username) {  
    
   **checkProjectOwnerService**.ProjectOwnerOnly(project, username);  
    
   **try**{  
   User user = **userRepository**.findByUsername(username);  
   project.setUser(user);  
   project.setProjectLeader(user.getUsername());  
   project.setProjectId(project.getProjectId().toUpperCase());  
   Backlog backlog = **new** Backlog();  
   project.setBacklog(backlog);  
   backlog.setProject(project);  
   backlog.setProjectIdentifier(project.getProjectId().toUpperCase());  
   **return projectRepository**.save(project);  
    
   }**catch** (Exception e){  
   **throw new** ProjectIdAlreadyExistException(**"Project ID already exists"**);  
    
   }  
    
   }  
    
    
   **public** Project findByProjectId(String projectId, String username){  
    
   **return checkProjectOwnerService**.PROJECT\_checkProjectOwner(projectId,username);  
   }  
    
   **public** Iterable<Project> findAllProjects(){  
   **return projectRepository**.findAll();  
   }  
    
   **public** Iterable<Project> findAllProjectsByUser(String username){  
    
   **return projectRepository**.findAllByUserUsername(username);  
   }  
    
   **public void** deleteProjectByProjectId(String projectId, String username){  
    
    
   **projectRepository**.delete(**checkProjectOwnerService**.PROJECT\_checkProjectOwner(projectId,username));  
   }  
    
    
    
  }
* New Backlog service
* @Autowired  
  **private** BacklogRepository **backlogRepository**;  
    
  @Autowired  
  **private** CheckProjectOwnerService **checkProjectOwnerService**;  
    
  **public** Backlog findBacklogByProjectId(String identifier, String username){  
    
   **checkProjectOwnerService**.PROJECT\_checkProjectOwner(identifier, username);  
   **return backlogRepository**.findByProjectIdentifier(identifier);  
  }
* It all works and you have, in my opinion, much cleaner code,

Step #9.1 Create a BacklogItem Object

* So we have our project, our project has a backlog and the backlog holds backlog items or project tasks if you will. For the sake of argument, let’s call it ProjectTask, we are going to annotate it as an entity and give it the following attributes

|  |  |  |  |
| --- | --- | --- | --- |
| Field | Type | Validation | ORM |
| id | Long | @Id | @GeneratedValue(strategy = GenerationType.***IDENTITY***) |
| projectSequence | String |  |  |
| Summary | String |  |  |
| acceptanceCriteria | String |  |  |
| Status | String |  |  |
| Priority | String |  |  |
| Created\_At | Date |  |  |
| Updated\_At | Date |  |  |

* Make sure we add the @prepresist for on update and on create
* @PrePersist  
  **protected void** onCreate(){  
   **this**.**created\_At** = **new** Date();  
  }  
    
  @PreUpdate  
  **protected void** onUpdate(){  
   **this**.**updated\_At** = **new** Date();  
  }
* Also empty constructor and getters and setters
* Let’s run it and check the tables
* Lets now assign constraints, some times, all you need to do is literally get steps our of your head but not much time to set up other details such as acceptance criteria and or priority, so the only constraint we are adding is that the summary cannot be blank. You can add as many constrains as you want!! It all depends on how you want your data.
* Now that we have the object, we need to set up the relationship with the backlog. A backlog has one or more project tasks, but a project task, in this usecase, can only belong to 1 backlog. Could this be many to many in real life as in, you have 1 task, that affects two or more projects, sure can. Since this is for personal projects, I’ll keep it one to many
* So let’s add the backlog attribute to our ProjectTask object, including getters and setters
* Now let’s set the ManyToOne relationship in our projecttask object. Remember, with any relationship, in the owned element, always json ignore the parent element so it prevents the infinite loop
* @ManyToOne(fetch = FetchType.***LAZY***, cascade = CascadeType.***REFRESH***)  
  @JoinColumn(name = **"backlog\_id"**, updatable = **false**, nullable = **false**)  
  @JsonIgnore  
  **private** Backlog **backlog**;
* Last step in this video is to set the relationship on the Backlog side, remember the backlog is the owning side of this relationship. One more small step, let’s create the project task repo interface, annotate it as a repository and extend the CrudRepo. Let’s restart our app and make sure it boots up properly

Step #9.2 Create a BacklogItem End point.

* Lets set up this up so we can create a projecttask. Wiring this up needs a fair amount of considerations as we want to make sure that
  + The user creating the pt owns the project
  + We create a pt in an existing backlog
  + That we set the backlog for the pt
  + That we set an initial status
  + That we add the pt to the list of pts
  + And that we save the backlog
* We are going to do this basic impl because I really want to show you something that could be an inconvenience for your app, in the backlog service, we will add an addProjectTask method that returns the updated backlog object. This method takes 3 parameters from the controller: backlog\_id, valid projectTask object, username
* Now that we have the service, let’s create the end point, let’s start by injecting our map validation service
* Then we are going to create a PostMapping that takes a backlog id as a parameter
* Then let’s create a method called addPTToBacklog that returns a generic response entity AND takes the following parameters (@Valid @RequestBody ProjectTask projectTask,  
   BindingResult result, @PathVariable String backlog\_id, Principal principal)
* The let’s implement our errorMap
* Let’s now invoke our service
* @PostMapping(**"/{backlog\_id}"**)  
  **public** ResponseEntity<?> addPTToBacklog(@Valid @RequestBody ProjectTask projectTask,  
   BindingResult result, @PathVariable String backlog\_id, Principal principal){  
   ResponseEntity<?> errorMap = **mapValidationErrorsService**.MapValidationErrorsService(result);  
   **if** (errorMap != **null**) **return** errorMap;  
     
   Backlog backlog = **backlogService**.addProjectTask(backlog\_id,projectTask,principal.getName());  
     
   **return new** ResponseEntity<Backlog>(backlog, HttpStatus.***CREATED***);  
    
  }
* Alright, let’s fire up our server, let’s regression test All good

Step #9.3 Project Tasks sequence.

* So let’s create a second project on our secondary account. So as you can see, we created TWO new pt in our brand new project BUT we get id 4 and 5? This might not seem like a big deal, BUT I promise you, as your have more projects with more tasks, this is going to become annoying. So what we are going to do is make sure that each project has it’s own sequence. Here I’m going to replicate what JIRA (PM tool from atlassian) does. What we want is the PROJECT Identifier + the sequence number *We also want it in a way that if PROJ-33 is erased before PROJ-34 is created, the next id given out by the server  
  is PROJ-34*
* Please note that this is the reason why we set the PTSequence to 0, so everytime we instantiate a new backlog with a new project the count starts at 0
* First let’s extract the current count from the object
* Let’s increase it by 1
* Let’s set the sequence to the project task we are creating
* Last but not least, lets update the backlog object sequence. Since this is incremental an not tied to the account of PTs in the database, we can achieve that once a pt is deleted, that id will not be given away.
* Alright, let us restart and regression test, so our sequence works

Step #9.4 Project Tasks sequence.

* Now that we know that the relationships work great, I think it is a good time to tap into a real backend. As you know, everytime we refresh our server, the h2 db gets flushed. So at this point in our project we are going to hook it up to a MySQL db. Two requirements on your end are: have MySQL installed on your environment, and have the SQL workbench. Once again, this is part of the course requirements for the student, if you don’t have this set up, go do it and come back to this lecture. Please not that we are going to update are json responses in later videos, so this way you don’t have to keep creating dummy data
* Alright so make sure MySQL server is running in our environment and then open MYSQL workbench and create a schema, you can give it whatever name you’d like, I’ll name it pptool, now that we have this
* Let’s go to our application.properties file to configure the access to the db
* Bug, not letting us in our account, ran debug mode, and it seems that it is unpacking the project object in the if statement which obviously doesn’t match our username, lets add an additional step here.
* String projectLeader = project.getProjectLeader();  
  **if**(projectLeader!= username){
* Rookie mistake, we were comparing two strings BUT we are not comparing the same object in memory, remember one comes for project.getProjectLeader and the other comes from principal.getName() if you === them, they are not the same. Why it worked in h2? It scapes me, I have my theory BUT I rather research on it rather than giving you bad feedback. Regardless, the proper way of comparing strings in java is equals anyway. This is a humbling moment. Let’s completely refactor this service to avoid issues
* Bug, you update the project, it still triggers the creation of a new backlog and duplicates backlog with the same identifier. We have two options. Separate the update function or just update the service.
* This is how I solved it
* **public** Project saveProject(Project project, String username) {  
    
    
    
   **checkProjectOwnerService**.ProjectOwnerOnly(project, username);  
    
   **try**{  
   User user = **userRepository**.findByUsername(username);  
   project.setUser(user);  
   project.setProjectLeader(user.getUsername());  
   *//The project id will only be null when it is a new project, so we only instantiate a new backlog  
   //for a new project* project.setProjectId(project.getProjectId().toUpperCase());  
   */\*\*  
   \* Here is the logic behind this, when we update the project we do not want to update the backlog  
   \* so when the project has a null id, we know that we are creating a project from scratch so we need a new backlog  
   \*/* **if**(project.getId()==**null**){  
   Backlog backlog = **new** Backlog();  
   project.setBacklog(backlog);  
   backlog.setProject(project);  
   backlog.setProjectIdentifier(project.getProjectId().toUpperCase());  
   }  
   */\*\*  
   \* here, if we pass a project id, that means that we are updating the project information  
   \* however, we do not want to update the backlog and we don't want to duplicate it either  
   \* for this reason, we need to reset the backlog on update, this way we  
   \* don't pass an null backlog for not including the backlog attribute when sending the json update  
   \* we protect the backlog from changes related to project info  
   \*/* **if**(project.getId()!= **null**){  
   project.setBacklog(**backlogRepository**.findByProjectIdentifier(project.getProjectId()));  
   }  
   **return projectRepository**.save(project);  
    
   }**catch** (Exception e){  
   **throw new** ProjectIdAlreadyExistException(**"Project ID already exists"**);  
   }  
  }

Step #9.5 Update expensive API responses.

* Now we are going to review our api, the 1st thing that I want to look at is that the performance in certain operations. The one that I’m mostly concerned about is when we load a project by ID. In our use cases we load project objects when we want to get the list of our projects and when we want to get a single project so lets check out loading a project with an empty backlog
* So as you can see when we get the newly created project is about 639Bytes
* As you can see the project loads with the backlog which means that it will too log its contents, so let’s create a few backlog items and see the load . so now that I added a few almost empty project tasks it went from 639 to 1.82 kb. And as we keep adding the bulkier our project object is. This might be fine if we are only planning to have 1 user, but as we move forward to having a decent amount of people using this, this will slow down and slow down the performance
* The 1st thing that we need to keep in mind is that when we load the project object, is only to see the project details and to update it, we do not need the backlog at all!! So in order to improve this, let’s use @JsonIgnore to remove the backlog from the response body. Now let’s request the project object again, right back down from 8.kb to just 560B. we do not need a huge, ever increasing load, lets try this now by adding more tasks. Now we have 15.kb worth of tasks, lets request our project object again 560, nice and steady.
* The other improvement is our projecttask post method, why do we need to bring the entire backlog back when all we need is to bring back the object created. Since we will likely do a redirect to the component calling the backlog, no need for such an expensive response so let,s update the backlogservice to return a project task and the controller for the same thing. As you can see we have optimized heavy responses where we really did not needed all that load

Step #9.6 Find projectTask in backlog

* In preparation for our update and delete operations, we first need to
  + Find existing PT ONLY if we are the projectLeader
  + Display an appropriate message when the PBI doesn’t exist
* We are going to start by creating a ProjectTaskService and inject the repository. We are going to annotate this as @Service
* We are going to create the following method that takes the following parameters and returns a project task
* **public** ProjectTask findByProjectIdentifier(String ProjectId, String backlogId, String username){  
    
  }
* then we are going to inject the CheckProductOwnerService
* now we are going to enable our repository to find a pt by the the projectSequence attribute
* @Repository  
  **public interface** ProjectTaskRepository **extends** CrudRepository<ProjectTask, Long> {  
    
   ProjectTask findByProjectSequence(String sequence);
* Then we are going to wire up our service
* **public** ProjectTask findByProjectIdentifier(String ProjectId, String backlogId, String username){  
    
   **checkProjectOwnerService**.PROJECT\_checkProjectOwner(backlogId,username);  
     
   ProjectTask projectTask = **projectTaskRepository**.findByProjectSequence(ProjectId);  
     
   **if**(projectTask == **null**){  
   **throw new** ProjectNotFoundException(**"Project Task: "**+ProjectId+**" not found"**);  
   }  
     
     
   **return** projectTask;  
     
  }
* now it is time to set up our backlogcontroller
* let’s autowire the projecttaskservice to our backlogcontroller
* **public** ProjectTask findByProjectIdentifier(String ProjectId, String backlogId, String username){  
    
   **if**(**backlogRepository**.findByProjectIdentifier(backlogId) == **null**){  
   **throw new** ProjectNotFoundException(**"Project with code: "**+backlogId+**" does no exist"**);  
   }  
    
    
   **checkProjectOwnerService**.PROJECT\_checkProjectOwner(backlogId,username);  
    
   ProjectTask projectTask = **projectTaskRepository**.findByProjectSequence(ProjectId);  
    
   **if**(projectTask == **null**){  
   **throw new** ProjectNotFoundException(**"Project Task: "**+ProjectId+**" not found"**);  
   }  
    
    
   **return** projectTask;  
    
  }

All test passed except for when I put a project from another user with the valid projectIdentifier. For this we need to

Add this to our project task object, and set it up on creation

**public** ProjectTask findByProjectIdentifier(String ProjectId, String backlogId, String username){  
  
 Backlog backlog =**backlogRepository**.findByProjectIdentifier(backlogId);  
 **if**( backlog== **null**){  
 **throw new** ProjectNotFoundException(**"Project with code: "**+backlogId+**" does not exist"**);  
 }  
  
 **checkProjectOwnerService**.PROJECT\_checkProjectOwner(backlogId,username);  
  
 ProjectTask projectTask = **projectTaskRepository**.findByProjectSequence(ProjectId);  
  
 **if**(projectTask == **null**){  
 **throw new** ProjectNotFoundException(**"Project Task: "**+ProjectId+**" not found"**);  
 }  
  
 **if**(!projectTask.getBacklog().getProjectIdentifier().equals(backlogId)){  
 **throw new** ProjectNotFoundException(**"Project Task does not exist in project: "**+backlogId+**" not found"**);  
 }  
  
 **return** projectTask;  
  
}

Step #9.7 Delete projectTask in backlog

* The first thing we are going wire up is the ProjectTaskService

**public void** deleteByProjectIdentifier(String ProjectId, String backlogId, String username){  
  
 Backlog backlog =**backlogRepository**.findByProjectIdentifier(backlogId);  
 **if**( backlog== **null**){  
 **throw new** ProjectNotFoundException(**"Project with code: "**+backlogId+**" does not exist"**);  
 }  
  
 **checkProjectOwnerService**.PROJECT\_checkProjectOwner(backlogId,username);  
  
 ProjectTask projectTask1 = **projectTaskRepository**.findByProjectSequence(ProjectId);  
  
 **if**(projectTask1 == **null**){  
 **throw new** ProjectNotFoundException(**"Project Task: "**+ProjectId+**" not found"**);  
 }  
  
 **if**(!projectTask1.getBacklog().getProjectIdentifier().equals(backlogId)){  
 **throw new** ProjectNotFoundException(**"Project Task does not exist in project: "**+backlogId+**" not found"**);  
 }  
  
  
 List<ProjectTask> pts = backlog.getProjectTasks();  
 pts.remove(projectTask1);  
 **backlogRepository**.save(backlog);  
  
 **projectTaskRepository**.delete(projectTask1);  
  
}

* After that we are going to wire up the backlogcontroller

@DeleteMapping(**"/{backlog\_id}/{pt\_id}"**)  
**public** ResponseEntity<?> deleteProjectTask(@PathVariable String backlog\_id, @PathVariable String pt\_id, Principal principal){  
  
 **projectTaskService**.deleteByProjectIdentifier(pt\_id,backlog\_id,principal.getName());  
  
 **return new** ResponseEntity<String>(**"Project Task "**+pt\_id+**" was deleted successfully"**, HttpStatus.***OK***);  
}

Step #10 Project Board elements

* 1st step we are going to take to display our backlog items is to create a project board class based component called ProjectBoard, so in the Project folder let’s create ProjectBoard.js
* Then we are going to use the rcc-> shortcut to generate the boiler plate code
* Then we are going to change the export default
* Then let’s set up a quick h1 message
* Then we are going to take care of the routing just so that we can test our component rendering in App.js
* Then we need to go to our ProjectItem component and set up the Link to point to /projectBoard
* Now we get Greetings from project Board
* Next component that we are going to create is the backlog component. It is an rcc and it will contain all the project tasks
* Now, we are going to replace our h1 in the projectboard with the backlog component
* Now we are going to create our projecttaskcomponent with sample data
* Then we are going to replace the h1 in the backlog component for a few ProjectTasks
* So this is looking really good, so now we are going to stop here and in the next video let’s start wiring this up to display project related data

Step #10.1 Hookup our backend

* The 1st thing we are going to do is set up our reducer, so in the reducers folder, create a backlogReducer.js file
* Before we do anything here, let’s add a few additional types in types.js in the actions folder

export const GET\_BACKLOG = "GET\_BACKLOG";

export const GET\_PROJECT\_TASK = "GET\_PROJECT\_TASK";

* Now that we have those two new action types, lets import them to our backlog reducer, actually let’s just import get backlog for now
* Now let’s set up our initial state, when it comes to the backlog we either have a collection of project tasks or a single project task
* import { GET\_BACKLOG } from "../actions/types";
* const initialState = {
* project\_tasks: null,
* project\_task: null
* }
* export default function(state = initialState, action) {
* switch (action.type) {
* case GET\_BACKLOG:
* return {
* ...state,
* project\_tasks: action.payload
* };
* default:
* return state;
* }
* }
* Now that we have created our initial reducer lets it up in index.js
* Cool, the next thing we need is to set up our backlog actions, so in the actions folder we are going to create a file called backlogActions.js
* Let’s start by bring in axios and the GET\_BACKLOG TYPE

import axios from "axios";

import { GET\_BACKLOG } from "./types";

* Create the action

export const getBacklog = backlog\_id => async dispatch => {

try {

const res = await axios.get(

`http://localhost:8080/api/backlog/${backlog\_id}`

);

dispatch({

type: GET\_BACKLOG,

payload: res.data

});

} catch (err) {

dispatch({

type: GET\_ERRORS,

payload: err.response.data

});

}

};

* Awesome, now that we have our action we need to then start wiring our components, we are going to cascade down from where the request is originated all the way down to the project task. So first, we need to go to the projectitem component and update the link just like we have for Update project
* Now that we are passing a parameter to this route, we need to account for this in app.js so we can actually pass this to the project board component component. Again we are going to replicate what we did for update
* If you remember previous lessons, whenever a component is rendered by a route, one of the props it gets is params, so now we are going to go to our project board component and start with some important imports

import { Link } from "react-router-dom";

import PropTypes from "prop-types";

import { getBacklog } from "../../actions/backlogActions";

* Now that we have our imports let’s connect the component to our application’s state
* Then let’s map our current state to props and pass the object as a param to connect
* Now let’s set our PropTypes

ProjectBoard.propTypes = {

backlog: PropTypes.object.isRequired,

errors: PropTypes.object.isRequired,

getBacklog: PropTypes.func.isRequired

};

const mapStateToProps = state => ({

backlog: state.backlog,

errors: state.errors

});

export default connect(

mapStateToProps,

{ getBacklog }

)(ProjectBoard);

* Now that we have this, we are going to use the componentDidMount lifecycle hook to load the backlog by extracting the parameter we are passing on from the projectitem component

componentDidMount() {

const { id } = this.props.match.params;

this.props.getBacklog(id);

}

* So now we have our project tasks loaded in our state

Step #10.2 display project tasks

* Now that we have our project tasks in our state, the next thing we want to do is display them in our backlog, but first let’s handle errors, so we are going to use a constructor and the lchook component will receive props

this.state = {

errors: {}

};

}

componentDidMount() {

const { id } = this.props.match.params;

this.props.getBacklog(id);

}

componentWillReceiveProps(nextProps) {

if (nextProps.errors) {

this.setState({ errors: nextProps.errors });

}

}

* Now, in the ProjectBoard component, we are goint to implement an algorithm to handle the load of the project tasks

const boardAlgorithm = (errors, project\_tasks) => {

if (project\_tasks === null) {

if (errors.projectNotFound) {

return (

<div className="alert alert-danger text-center" role="alert">

{errors.projectNotFound}

</div>

);

} else {

return <h1>Loading...</h1>;

}

} else {

if (project\_tasks.length < 1) {

return (

<div className="alert alert-info text-center" role="alert">

No Project Tasks on this board

</div>

);

} else {

return <Backlog project\_tasks={project\_tasks} />;

}

}

};

BoardContent = boardAlgorithm(errors, project\_tasks);

return <div className="container">{BoardContent}</div>;

* Now we just found one more issue and it is when we pass a project that doesn’t exists it throws a 500 error, chance the order of the logic @ CheckProjectOwnerService
* **public** Project PROJECT\_checkProjectOwner(String projectId, String username){  
    
   Project project = **projectRepository**.findByProjectId(projectId);  
    
   **if**(project == **null**){  
   **throw new** ProjectNotFoundException(**"Project not found"**);  
   }  
    
   **if**(!project.getProjectLeader().equals(username)){  
   **throw new** ProjectNotFoundException(**"Project not found in our account"**);  
   }  
    
   **return** project;  
    
  }
* With that out of the way, now we need to actually display project tasks
* We need to go back to our backend because right now we are not retrieving the backlog list as an object. So for this we are going to update the backlog controller as follows
* @GetMapping(**"/{projectId}"**)  
  **public** ResponseEntity<List<ProjectTask>> getProjectBacklog(@PathVariable String projectId, Principal principal){  
   **return new** ResponseEntity<List<ProjectTask>>(**backlogService**.findBacklogByProjectId(projectId, principal.getName()).getProjectTasks(), HttpStatus.***OK***);  
  }
* Now let’s update our backlog component

class Backlog extends Component {

render() {

const { project\_tasks } = this.props;

console.log("from bl", project\_tasks);

//let BacklogContent;

const tasks = project\_tasks.map(project\_task => (

<ProjectTask key={project\_task.id} project\_task={project\_task} />

));

return <div className="container">{tasks}</div>;

}

}

* And finally our project task component

class ProjectTask extends Component {

render() {

const { project\_task } = this.props;

return (

<div className="card mb-1 bg-light">

<div className="card-header text-primary">

{project\_task.projectSequence}

</div>

<div className="card-body bg-light">

<h5 className="card-title">{project\_task.summary}</h5>

<p className="card-text text-truncate ">

{project\_task.acceptanceCriteria}

</p>

<a href="#" className="btn btn-primary">

View / Update

</a>

</div>

</div>

);

}

}

* In preparation for the update portion I want to make two quick backend changes so that the project task also holds the projectIdentifier, so at the POJO
* *//Update***private** String **projectIdentifer**;
* Then at the backlogservice, at the addProjectTask method, right after setting the sequence, we’ll set the identifier
* projectTask.setProjectSequence(backlog.getProjectIdentifier()+**"-"**+BacklogSequence);  
  projectTask.setProjectIdentifer(projectIdentifier);
* Let’s wipe out our data and create a few tasks, in doing this I just realized that when creating a tasks, it gets duplicated, we need to remove this line of code
* *//let's add this new task to the backlog list  
  //projectTasks.add(projectTask*); also remove the list
* We do not need to save the backlog either
* *// backlogRepository.save(backlog);*

Step #10.3 create project tasks step 1 – Create Component

* Created type
* Created reducer
* Created action
* Created AddProjectComponent
* Added HTML and updated to JSX
* Added button to ProjectBoard + BR +HR and path “/addProjectTask”
* Added the component to App.js and tested the route
* Now it is time to update the route in projectboard to pass a parameter to the link and App.js to take the parameter
* We are going to extract the parameter again from props but this time in the render method
* Now let’s update the back to project board button to take us back to our backlog in the addprojecttask component. We need to import Link

Step #10.4 create project tasks step 2

* Let’s first bring all the imports we need
* Then let’s make this a controlled component. Actually, I want you pause the video and go do this on your own,
* 1st let’s add the constructor in order to set the field’s state
* Again, I love being super careful so lets bring the json response from postman
* Just noticed that the projecttask object doesn’t contain the due date so I’m adding it right now
* Now that we have set up an initial state, time to bind the value to the state
* So now again we won’t be able to do anything to the form anymore
* Alright so now on change and bind it in the constructor
* Now let’s hook in up in our input fields
* As always, before sending stuff back to the server, let’s do the onsubmit to console to make sure our pt json object looks good – bind the form
* Additional test, grabbed json object and used in postman, so now it is matter of testing our action and reducer

Step #10.5 create project tasks step 3

* Now we are going to wire up our happy path,
* 1st step is to connect our app to the store
* 2ndo step is to mapstateprops as we could get errors if the object is invalid
* Proptypes too
* Now that we have this, we need a lchook and at this point, you need to know which one, I’m really not telling!!!! And add an errors object to the constructor
* Then we need to update the on submit
* Updated the backend (BACKLOG SERVICE FOR SAVING TASKS)
* **if**(projectTask.getPriority()==**null**){  
   projectTask.setPriority(**"LOW"**);  
  }
* **if**(projectTask.getStatus()==**""**|| projectTask.getStatus()==**null**){  
   projectTask.setStatus(**"TO\_DO"**);  
  }

Step #10.6 let’s deal with errors now

* Let’s send an invalid one
* Errors in state
* Let’s use classnames to display errors

Step #10.7 lets update:

* Create the component
* Set up the routing in appjs
* Set up and import the link in project task
* Set up types GET\_PROJECT\_TASK,
* Import GET\_PROJECT\_TASK in backlogActions and backlogReducer
* Update backlog reducer
* Now create the get action
* Now,let’s make sure we pass the required parameters from the project task to the updateprojecttask component – we also need to update appjs btw
* Now that we have the action, let’s connect the component and load it in the state so let’s start by bringing
* 1st thing we are going to use is component did mount – remember, at this stage we just want our tool loaded in our state
* In componentdidmount we want to extract the params
* Before we move any further let’s take care of the imports
* Now lets finish component didmount with the getProjectTask
* And now let’s connect our component to the state
* So now that we have this project task in our state, lets load our form