Project Walkthrough - Day 2

This day can be used as alternate code for web service day.

Install Bootstrap in a Vue project

Because we want to use Bootstrap for some of the buttons and form fields, we need to add it to our project. Here are the instructions:

Add links from https://getbootstrap.com/ to the src/index.html

See links in Doc Quickstart: Step 2 -
https://getbootstrap.com/docs/5.3/getting-started/introduction/

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Bootstrap demo</title>
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" rel="stylesheet" integrity="sha384-QWTKZyjpPEjISv5WaRU9OFeRpok6YctnYmDr5pNlyT2bRjXh0JMhjY6hW+ALEwIH" crossorigin="anonymous">
  </head>
  <body>
    <h1>Hello, world!</h1>
    <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/js/bootstrap.bundle.min.js" integrity="sha384-YvpcrYf0tY3lHB60NNkmXc5s9fDVZLESaAA55NDzOxhy9GkcIdslK1eN7N6jIeHz" crossorigin="anonymous"></script>
  </body>
</html>
```

```
<script src="https://cdn.jsdelivr.net/npm/@popperjs/core@2.11.8/dist/umd/popper.min.js"
integrity="sha384-I7E8VVD/ismYTF4hNIPjVp/Zjvgyol6VFvRkX/vR+Vc4jQkC+hVqc2pM8ODe
wa9r" crossorigin="anonymous">
</script>

<script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/js/bootstrap.min.js"
integrity="sha384-0pUGZvbkm6XF6gxjEnlmuGrJXVbNuzT9qBBavbLwCsOGabYfZo0T0to5e
qruptLy" crossorigin="anonymous">
</script>
```

Retrieving players using an Axios GET Request

1. FIrst thing we need to do is remove the players[] from the Vuex store. We are going to get that information from our backend Spring Boot server.

2. Now go to the PlayerSearch component and add a players[] to the data section:

```
data() {
  return {
    nameFilter: "",
    players: []
  };
},
```

3. Now we need to create a service client. Under the services folder, create a **PlayerService.js** file

```
import axios from 'axios';

export default {
   fetchPlayers() {
     return axios.get('/players')
    },
```

```
}
```

4. Now we need to import this service. It needs placed inside the <script> section of the PlayerSearch.vue component, but above the export default:

```
import playerService from "../services/PlayerService.js";
```

It should look like the following when done:

```
<script>
import playerService from "../services/PlayerService.js";

export default {
  data() {
    return {
      nameFilter: "",
      players: []
    };
  },
```

5. We are now ready to add the code that makes the web service call. Since we want this data to be fetched when the component loads, we want to add it to the created() hook. We don't have one yet, so we need to create it inside the <script> section. Below the computed is a great place for it.

```
  created() {
    playerService.fetchPlayers().then((response) => {
      this.players = response.data;
    });
  },
```

TODO: We should go back and add exception handling on this web service call to catch

any http errors that return. We will skip that for now.

6. Next, we need to change our filteredPlayers() to NOT use the store, but instead use: this.players. It's the one we added to the data section earlier.

```
    filteredPlayers() {
      const players = this.players;

      return players.filter((player) => {
        return this.nameFilter == "" ? true :
player.fullName.includes(this.nameFilter);
      });
    },
```

7. Now, if we look at the web page, you will notice that the page loads, but there is no data. But we tested the service through Postman previously and the service worked. (If not, I would start there). If the server side works, then maybe the browser console will tell us something.

search:1 Access to XMLHttpRequest at 'http://localhost:9000/players' from origin **'http://127.0.0.1:5173' has been blocked by CORS policy:** No 'Access-Control-Allow-Origin' header is present on the requested resource.

CORS is a server side issue. But it has an easy fix. We need to head over to the controller.

8. Open IntelliJ and go to the PlayerController. Add the following annotation, `@CrossOrigin` just under @RestController, but before the line that has public class PlayerController. Once finished, restart the server and verify it starts correctly.

9. Now refresh the page and you should see all data loading.

Create a form to add a new player

Next up we want to build a form that will submit a new player to the roster. To do this we will build another component, called **AddPlayer.vue**, and place it inside an **AddPlayerView.vue** page.

After submitting the form data, if successful, we will build a dynamic route (automatically forward them back to the search page. - While I would prefer that this form was hidden on the main search page, this is an opportunity to show dynamic routing.) To keep them both on the same page would require us, using the $store.state to trigger rendering of the newly added player to the roster. TODO: come back at some point and update how to do that.

1. Let's start by creating the AddPlayer.vue component. By now, you should be fairly comfortable building the basic structure of one.

```
<template>
  <div>
     <p>Test Me</p>
  </div>
</template>

<script>

export default {
  data() {
    return {
    };
  }
</script>

<style scoped>
</style>
```

While we are at it, let's add this component to the **AddPlayerView.vue** page.

```
<script>

import AddPlayer from '../components/AddPlayer.vue';

export default {
   components: {
       AddPlayer
   }
}
</script>

<style scoped>

</style>
```

Let's set up a route to navigate to this page and add a menu item:

router → index.js

```
  {
    path: '/addplayer',
    name: 'addplayer',
    component: AddPlayerView,
    meta: {
      requiresAuth: false
    }
  },
```

TheHeader.vue

```
<router-link v-bind:to="{ name: 'addplayer' }">Add Player</router-link>
```

2. Now let's add some form fields. Feel free to cut and paste what I created below as it takes time to build by hand, but feel free to try it on your own first.

```
<div id="addplayerform">
  <form class="playerForm">
    <div class="form-group">
      <label for="jerseyNumber">Jersey Number:</label>
      <input
        id="jerseyNumber"
        type="text"
        class="form-control"

      />
    </div>
    <div class="form-group">
      <label for="firstName">First Name:</label>
      <input
        id="firstName"
        type="text"
        class="form-control"

      />
    </div>
    <div class="form-group">
      <label for="lastName">Last Name:</label>
      <input
        id="lastName"
        type="text"
        class="form-control"

      />
    </div>
    <div class="form-group">
      <label for="salary">Salary:</label>
      <input
        id="salary"
```

```
          type="text"
          class="form-control"

        />
      </div>
      <div class="form-group">
        <label for="salary">Position:</label>
        <select>
          <option>
             Pitcher
          </option>
        </select>
      </div>

      <button class="btn btn-submit">Submit</button>
      <button class="btn btn-cancel" type="cancel">Cancel</button>
    </form>
  </div>
```

3. Now let's add in the CSS:

```
#addplayerform {
  margin-left: auto;
  margin-right: auto;
  width: 500px;
}
form input {
  width: 100%;
}
.playerForm {
  padding: 10px;
  margin-bottom: 10px;
}
.form-group {
  margin-bottom: 10px;
  margin-top: 10px;
}
```

```css
.form-control {
  display: flex;
  align-items: flex-start;
  width: 100%;
  height: 30px;
  padding: 0.375rem 0.75rem;
  font-size: 1rem;
  font-weight: 400;
  line-height: 1.5;
  color: #495057;
  border: 1px solid #ced4da;
  border-radius: 0.25rem;
}
textarea.form-control {
  height: 75px;
  font-family: Arial, Helvetica, sans-serif;
}
select.form-control {
  width: 20%;
  display: inline-block;
  margin: 10px 20px 10px 10px;
}
.btn-submit {
  color: #fff;
  padding: 10px 24px;
  background-color: #38b412;
  box-shadow: 0 12px 26px 0 rgba(0, 0, 0, 0.24),
    0 17px 50px 0 rgba(0, 0, 0, 0.19);
}
.btn-cancel {
  padding: 10px 24px;
  box-shadow: 0 12px 26px 0 rgba(0, 0, 0, 0.24),
    0 17px 50px 0 rgba(0, 0, 0, 0.19);
}
.btn-submit:hover {
  color: #fff;
  padding: 10px 24px;
  background-color: #65f307;
  box-shadow: 0 12px 26px 0 rgba(0, 0, 0, 0.24),
```

```
    0 17px 50px 0 rgba(0, 0, 0, 0.19);
}
.btn-cancel:hover {
  padding: 10px 24px;
  background-color: #65f307;
  box-shadow: 0 12px 26px 0 rgba(0, 0, 0, 0.24),
    0 17px 50px 0 rgba(0, 0, 0, 0.19);
}
.status-message {
  display: block;
  border-radius: 5px;
  font-weight: bold;
  font-size: 1rem;
  text-align: center;
  padding: 10px;
  margin-bottom: 10px;
}
.status-message.success {
  background-color: #90ee90;
}
.status-message.error {
  background-color: #f08080;
}
```

4. For the data section, we will need three things:

   a) selectedValue: “” to hold the selection from the drop down player-position box
   b) options: [] to hold the player positions to fill the drop down box
   c) newPlayer: {} to two-way bind (v-model) each form field to.

```
  data() {
    return {
      selectedValue: "",

      options: [
        { value: "", text: "" },
        { value: "Pitcher", text: "Pitcher" },
```

```
        { value: "Catcher", text: "Catcher" },
        { value: "First Baseman", text: "First Baseman" },
        { value: "Second Baseman", text: "Second Baseman" },
        { value: "Third Baseman", text: "Third Baseman" },
        { value: "Shortstop", text: "Shortstop" },
        { value: "Outfielder", text: "Outfielder" },
      ],
      newPlayer: {
        firstName: "",
        lastName: "",
        jerseyNumber: "",
        salary: "",
        positions: [],
        teamId: 1,
        fullName: "",
      },
    };
  },
```

5. Now we need to connect the newPlayer object to the form fields and attach the options array to the select box. Here they are for quick reference.
For the input boxes, use these:

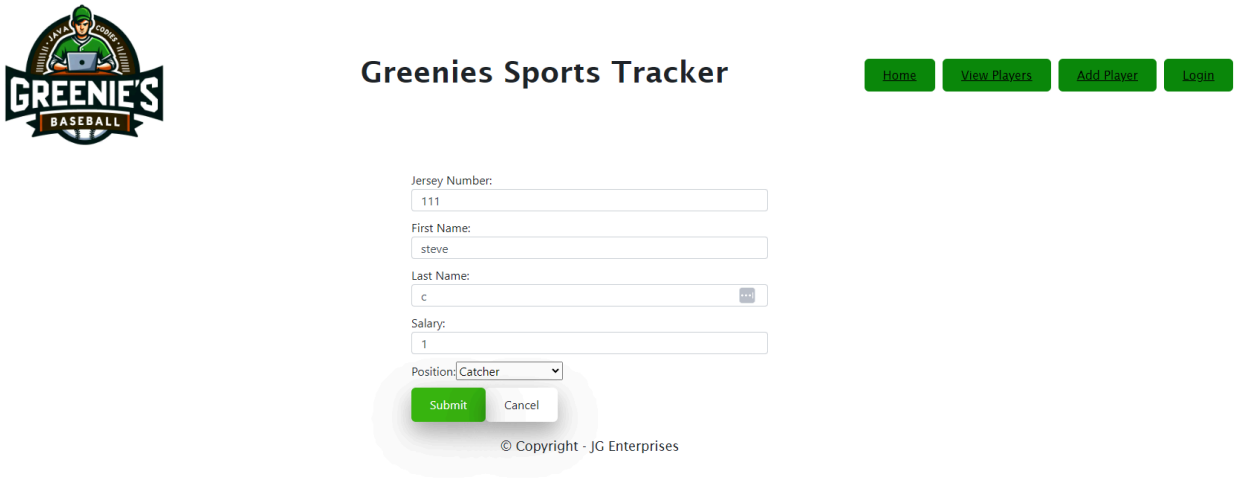
```
v-model.number="newPlayer.jerseyNumber"
v-model.number="newPlayer.firstName"
v-model.number="newPlayer.lastName"
v-model.number.number="newPlayer.salary"
```

For the player-positions select box (it is tricky), do this:

```
          <select v-model="selectedValue">
           <option
            v-for="option in options"
            :value="option.value"
            v-bind:key="option.value"
          >
            {{ option.text }}
          </option>
```

```
        </select>
```

Here is what the form should look like:



Sending form data as an Axios POST Request

1. Finally, we need to hook up the form to the web service. The first thing we need is to add a web service call to our PlayerService.js file. Using axios, we can do this:

```
    addPlayer(player) {
      return axios.post('/players', player)
    }
```

Then we need to import the player service. We do this just below the opening <script> tag, but above the export default. Here is what it should look like:

```
import playerService from '../services/PlayerService.js';
```

After importing the service, we need to use it. Since the addPlayer web service won't be called until a user takes action, we will put the call in a *methods* section. Let's add the methods section just below the data() section.

Here is what goes inside it:

```
methods: {
    sendForm() {
      // set the position selected on the object
      this.newPlayer.positions[0] = this.selectedValue;
      this.newPlayer.fullName = this.newPlayer.firstName + " " +
this.newPlayer.lastName;

      playerService
        .addPlayer(this.newPlayer)
        .then((response) => {
          if (response.status === 201) {
            //add a success message and refetch the data
            console.log(response.data);
            this.newPlayer = {};
          }
        })
        .catch((error) => {
          //do something
          console.log(error);
        });
    },
```

Explanation of above code:

```
// set the position selected on the object
this.newPlayer.positions[0] = this.selectedValue;
```

The select box was v-modeled to this.selected value, a variable we added to data section earlier. The newPlayer object needed an [] for the player positions due to the Server-side API requirements. This simply grabs the selectedValue (containing the player position, such as Pitcher, Catcher, etc) and adds it to the first element in the newPlayer.positions array.

```
this.newPlayer.fullName = this.newPlayer.firstName + " " +
this.newPlayer.lastName;
```

Since we wanted our form as streamlined as possible, we did not include a full name form field. We can derive that by using what they entered for their firstName and lastName. This does that before we send it to the server.

Finally, we can make the web service call

```
    playerService
        .addPlayer(this.newPlayer)
        .then((response) => {
          if (response.status === 201) {
            //add a success message and refetch the data
            console.log(response.data);
            this.newPlayer = {};
          }
        })
        .catch((error) => {
          //do something
          console.log(error);
        });
    },
```

As you can see, we added exception/error handling to the code. For successful http responses, we can look those codes by using response.status (200 series codes). If we received a 201 Created HTTP Status back, per the API server side documentation we created, this was a successful response.

For now, I just log the response.data to the browser console. Later, we will add a dynamic route here, to forward them back to the search page.

Finally, I clear the newPlayer object

At the end of the promise (the then() part), I attached a .catch with a function to catch an error IF we get back a HTTP error code (400, 500 series, etc).

TODO: For now, I log to the browser console, but if we have time at some point, we should add

this as an error message.

Lastly, hook this up to the form, we need to add a v-on tag to the form:

```
<form class="playerForm" v-on:submit.prevent="sendForm">
```

If time, we should hook up the Cancel button to clear out the form. I will leave that for another time. Basically, add another method that clears the newPlayer object and either keeps them on the same page, or dynamically routes them back to the search page. Don't forget to add the v-on tag to the Cancel button

Add an error message to the Add Player Page

You may have noticed that we are not dealing with errors coming back from the server. We are simply logging them in the browser. Let's fix that.

1. The first thing we need is to add a new variable to our data().

```
  errorMsg: ''
```

2. Next use one-way data-binding to bind to the template. A good spot is at the top of the page just above the form and inside the first main div.

```
<div class='status-message error' v-show="errorMsg != ''">{{errorMsg}}
</div>
```

3. Now we need to create a function to handle any errors

```
handleErrorResponse(error) {
        //figure out the type of error
        if (error.response) {
          // set a variable to msg text
```

```
          this.errorMsg = 'Error adding a new player. Error: ' +
error.response.status;
        }
        else if (error.request){
          //set a variable to msg text
          this.errorMsg = 'Error adding a new player. Server could not
be reached.';
        }
        else {
             // catch all - return generic error msg
             this.errorMsg = 'General Error. Call 1-800-Bla-Blah';
        }
    }
```

4. We can test our error handling by shutting down the server. We should get the following message: