**How to Implement AJAX With Spring Boot==**

**Our Use Case**

For this tutorial, we’ll create an app that allows an HR rep to retrieve employee info.

Here’s what we’re going to do: we’re going to set up a page that displays a drop-down. The drop-down will include various employee IDs.

When the user selects one of the IDs, the app will display the employee’s record on the same page, *without a reload*. The app will also display the user’s profile pic.

Seems easy enough, right? Let’s go!

**Let’s Start at the Basement**

We’ll start this effort by creating a couple of Java beans that represent our data model.

The first bean will contain info about the employee. That info includes the employee’s ID, first name, last name, department, and hours per week.

Since we’re showing the employee’s pic as well, the bean will also hold a reference to the URL of the image file.

Here are the fields that we’ll define in our [Employee](https://github.com/careydevelopment/ajaxdemo/blob/master/src/main/java/com/careydevelopmnet/ajaxdemo/model/Employee.java) class:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | private String employeeId;  private String firstName;  private String lastName;  private String department;  private Integer hoursPerWeek;  private String imageFile; |

Obviously, we’ll follow the standard Java bean convention of defining accessors and mutators (getters and setters) for each of those fields.

The next Java bean we’ll define will act as a wrapper for the employee info. It’s going to include a status field in addition to the employee info.

Why do we do this? So that clients can determine if the REST call was successful or not.

**Wait… What’s REST?**

“This is an AJAX tutorial! Why are you talking about REST?!?”

That’s a good question. Fortunately, I have a good answer.

REST, which stands for REpresentational State Transfer, is often used with AJAX. It’s how we retrieve the data that we’ll display on the front end.

Our REST call uses the following path: /employee/{employeeId}

where {employeeId} corresponds to the ID of the employee that we’re searching for.

So, for example: the path /employee/A552 is searching for an employee with ID A552.

But what if somebody searches for an employee with ID A553 and no such employee exists? That’s where our status field comes into play. The status field will inform the client that it didn’t find the record.

Here’s the code for the second bean, called [RestResponse](https://github.com/careydevelopment/ajaxdemo/blob/master/src/main/java/com/careydevelopmnet/ajaxdemo/model/RestResponse.java):

|  |  |
| --- | --- |
| 1  2  3  4  5 | public static final String NOT\_FOUND = "Not found";  public static final String OK = "Ok";    private String responseStatus;  private Object response; |

The static fields just define possible values for **responseStatus**. The response field is an Object because we might want to use this class for objects other than **Employee**.

We’ll use both the **RestResponse** and **Employee** classes to serve up REST responses in JSON format.

**JSON?**

Yes, it’s time for yet *another* acronym

JSON stands for JavaScript Object Notation. It’s often used with AJAX because, as the name suggest, it’s JavaScript-friendly.

Here’s what our entire JSON response looks like when we submit a REST call for an employee with ID A552:

{"responseStatus":"Ok","response":{"employeeId":"A552","firstName":"Danny","lastName":"DeVito","department":"Accounting","hoursPerWeek":40,"imageFile":"dannydevito.jpg"}}

As you can see, the JSON response consists of two parts: the response status and the response itself.

The response status tells us whether or not the call was successful. A status of “Ok” means we’re good.

The second part of the response is just the employee info.

Now, let’s code up a controller that serves up the JSON response.

**Responding to a REST Call**

We need a controller to handle the REST path. Fortunately, that’s easy to do with Spring Boot.

We’ll create a controller class that returns a **RestResponse** object. That’s the class we created just above.

Spring Boot is smart enough to translate that object into JSON format if you annotate the class with [@RestController](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/bind/annotation/RestController.html). We really don’t have to do any extra coding.

Here’s the request mapping method in [EmployeeRestController](https://careydevelopment.us/blog/):

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | @RequestMapping("/employee/{employeeId}")      public RestResponse singleEmployee(@PathVariable String employeeId) {          //instantiate the response object          RestResponse response = new RestResponse();            //set the employee to null          Employee returnedEmployee = null;            //grab all employees          List<Employee> allEmployees = getAllEmployees();            //look for a match          for (Employee employee : allEmployees) {              if (employee.getEmployeeId().equals(employeeId)) {                  returnedEmployee = employee;                  break;              }          }            if (returnedEmployee == null) {              //the URL contains an unknown employee id - we'll return an empty response              response.setResponseStatus(RestResponse.NOT\_FOUND);              response.setResponse("");          } else {              //good response if we get here              response.setResponseStatus(RestResponse.OK);              response.setResponse(returnedEmployee);          }            return response;      }  } |

There’s quite a bit going on there. Let’s look at each of the pieces.

First, notice the **@RequestMapping** annotation. We’ve specified the **employeeId** in curly braces.

That’s how we tell Spring Boot that we’re looking for a value in that space. It corresponds to the **@PathVariable** annotation in the method signature.

It’s easy to tell that the two are related because they have the same name (**employeeId**).

At the beginning of the method, we instantiate an empty **RestResponse** object and set the **returnedEmployee** variable to null.

If we have a successful call, **returnedEmployee** will contain the **Employee** object that corresponds to the employee ID specified in the call.

The **getAllEmployees()** method does nothing more than return a hardcoded set  of **Employee** objects. Normally, we’d get employee info from a database, but we’re just keeping it simple for this tutorial.

In the for loop, we’re just stepping through each of the **Employee** objects and looking for one with an ID that matches what was provided in the path. If we find a match, we set **returnedEmployee** accordingly and get out of the loop.

Savvy Java developers might notice that there’s a good refactoring possibility here: we should use a **Map** instead of a **List** to store the employees and we could avoid the for loop all together.

Following the loop, we’re going to check the value of **returnedEmployee**. If it’s still null, then we didn’t find a match.

In that case, we’re getting out of the method and returning a status of “Not found” to the unfortunate client.

On the other hand, it **returnedEmployee** isn’t null, then we found a match. We’ll return an “Ok” status to the client with the employee info.

At this point, it might seem like we’ve accomplished quite a bit, but all we’ve done is coded the back end to handle REST requests. Now, we need to develop the front end.

**A Single Page**

As you can imagine, we’re just going to develop a single HTML page for this app. That’s because all the action happens on the same page.

That’s usually *the whole point* of AJAX, after all.

So the first thing we’ll do, after establishing the structure of the page, is add the drop-down. Here’s what that code looks like:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | <select class="bs-select form-control" id="employeeIdSelect" onchange="employeeSelect()">     <option value="default" selected="true">- SELECT -</option>     <option value="A552">A552</option>     <option value="B112">B112</option>     <option value="C997">C997</option>     <option value="A703">A703</option>  </select> |

So we’ve got 4 employee IDs in the drop-down. To keep our demo simple, we’ve hardcoded them.

The most important thing to note here is the **onchange** attribute. As you can see, it points to a JavaScript function called **employeeSelect**.

You shouldn’t be surprised that we’re using JavaScript because that’s the “J” in AJAX.

Basically, that attribute is telling the app to execute the **employeeSelect** function whenever a user selects an employee ID.

But what does that function do? That’s what we’ll go over in the next step.

**Responding to a User Selection**

When the user selects an employee ID, we want to display the employee’s profile just below the drop-down. We’ll handle all of that in a JavaScript function called **employeeSelect**.

Here’s what that function looks like:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | function employeeSelect() {     //display the spinner     $('#ajaxLoader').show();       //first, let's get rid of the default "SELECT" option if it exists     var defaultOption = $("#employeeIdSelect option[value='default']");     if (defaultOption) defaultOption.remove();       //get the selected id     var employeeId = $('#employeeIdSelect').val();       //get the url for the ajax call     var url = "./employee/" + employeeId;       //do the ajax call     $.get(url, populateEmployeeInfo);  } |

As you can see, the method uses JQuery. That library makes life easier for JavaScript developers.

The first thing the method does is show an element with the **ajaxLoader** ID. All we’re doing there is showing a spinner while the app retrieves the data from the back end.

It’s good design to show users a spinner during actions that take a while. That way, the user knows that something is going on.

Truth be told, though, this REST call will go so fast that nobody will see the spinner. Still, it’s a good habit.

The next couple of lines get rid of the default “-SELECT-” option. We do that so that user can’t select it as an option after already selecting another option.

The next line gets the value from the select drop-down. That’s pretty straightforward.

Next, we construct our REST path from with the value we just retrieved. As you might (or might not) recall, the REST path looks like this: /employee/{employeeId}

That {employeeId} bit will be substituted with the value from the drop-down.

In the last line, we perform the actual AJAX call (that’s what **$.get** does in JQuery). Our callback function is called **populateEmployeeInfo**. That’s the function that will be executed when the REST call is completed.

**UI Work**

Once the REST call is complete, it’s time to update the UI.

As we’ve seen, the JavaScript function **populateEmployeeInfo** will be executed after the REST call. That’s the function that will update the UI with employee info.

Here’s what that function looks like:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | function populateEmployeeInfo(data) {     var status = data.responseStatus;       //check the response to make sure it's ok     if (status == "Ok") {        var response = data.response;          //get the JSON data        var department = response.department;        var employeeId = response.employeeId;        var firstName = response.firstName;        var lastName = response.lastName;        var hoursPerWeek = response.hoursPerWeek;        var imageFile = './images/' + response.imageFile;          //set the profile badge values        $('#employeePic').attr("src", imageFile);        $('#employeeName').html(firstName + ' ' + lastName);        $('#employeeDepartment').html(department);          //set the input field values        $('#employeeFirstName').val(firstName);        $('#employeeLastName').val(lastName);        $('#employeeDept').val(department);        $('#employeeId').val(employeeId);        $('#employeeHours').val(hoursPerWeek);          //show the hidden elements        $('#profileRow').css('visibility','visible');     }       //hide the spinner again     $('#ajaxLoader').hide();  } |

Although there are quite a few lines of code in that block, it’s really not that complicated. All we’re doing is setting the contents of various HTML elements with data that we parse from the JSON response.

The first thing we check is the response status. As you can see, we only populate data if the status is “Ok”.

If the status is “Ok”, then we parse out the employee data. The JSON field names correspond to the field names from the Java beans we set up at the very beginning of this tutorial. That’s not a coincidence, because Spring Boot constructs the JSON response from the Java bean fields.

For each piece of employee info (first name, last name, etc.), we populate an HTML element with that data. So, for example, the HTML element with the ID **employeeFirstName** gets the employee’s first name.

The only slight variation to that rule is the employee’s photo. For that, we set the **src** attribute of the **<img>** tag accordingly.

Now, let’s see how this whole thing works.

## Before and After-