Assignment: TechJobs (MVC Edition)

Introduction

Your first task as an apprentice went well! You and Kathy built the TechJobs console prototype, and after demoing it to the Company Team at LaunchCode, the project has been green-lit to be fully built out as a web application.

The first step in this process will be to quickly develop a **minimum viable product (https://en.wikipedia.org/wiki/Minimum_viable_product)**, or MVP. The goal is to get a functioning web app up and running with as little work as possible, so additional feedback and testing can be done early on in the development process. Then additional behind-the-scenes work will be carried out to fully develop the model and data side of the application.

After working with Kathy on the console demo, you'll be working with Eliot on this project. Here's a picture of Eliot:

Eliot was once a LaunchCode apprentice as well, so he knows just what it's like to be in your shoes. He's done some initial work on the project and left you some TODOs that he knows you can handle.

Learning Objectives

In this project, you'll show that you can:

- Read and understand code written by others
- Work within the controller and view portions of a Spring Boot application
- Create new handler methods to process form submission
- Use Thymeleaf syntax to display data within a view

TechJobs (MVC Edition)

You'll start with some code that Eliot has written to get you started. The idea behind your current assignment is to quickly deliver a functioning Spring Boot application, so you'll focus on the controllers and views.

In order to do this, you'll be reusing the `JobData` class and `job_data.csv` file from the console app. You know you'll have to go back and rewrite the data portion of the application in the future, to make a true, database-backed model, but using the existing `JobData` class to provide some basic data functionality will let you focus on the views and controllers for now.

Your Assignment

Eliot has created a Spring Boot application and filled in some features. It has controllers and views for a home page, along with functionality to display lists of data values for each column/field of the data (employer, location, etc). Eliot also started working on the search functionality, but only got as far as writing the code to display the search form.

He's handed it off to you to finish the rest. You'll add code to controllers and views to process and display search results, along with allowing users to see all jobs in the system via the List page.

Getting Started

Set up a local copy of the project:

- Visit the **repository page (https://github.com/LaunchCodeEducation/techjobs-mvc)** for this project and fork the repository to create a copy under your own GitHub account.
- Back in IntelliJ, select *File > New > Project from Version Control > GitHub*.
- Choose your fork from the repository dropdown, select the parent directory where you'd like to store your project, and hit *Clone*.
- In the screens that follow:
    - Choose *Create Project From Existing Sources* on the first pane
    - Select *Auto Import* in the Gradle configuration pane
    - Select defaults on all other panes

Go ahead and start up the application (via the Gradle pane, *Tasks > Application > bootRun*), so you can refer to both the code as well as the running app while we look at what's in place already.

✶ Note

You'll spend a lot of time reading and understanding the code that's provided for you. That's okay, and in fact should be considered part of the assignment! It's an essential skill for a programmer to be able to read and understand code that others have written.

THE MODEL

✶ Note

When referring to Java class locations, we'll usually only mention the portion below `src/main/java/org/launchcode/`.

The "model" is contained in the `models` package, in the `JobData` class. We put "model" in quotes, since this class isn't a model in the typical, object-oriented sense that we usually mean (maybe a better name for this assignment would be TechJobs VC!).

The `JobData` class is the exact same class that you used in the console app. The only modification is that Eliot changed the path to the `job_data.csv` file so that it could be stored in the `src/main/resources` directory.

You'll use some of the static methods provided by `JobData` in your controller code. Since you're already familiar with these, we'll leave it to you to review their functionality as you go.

THE CONTROLLERS

Expand the `controllers` package, and you'll see that you have three controllers already in place. Let's look at these one at a time.

HomeController

This class has only one handler method, `index`, which displays the home page for the app. As you can see, this controller renders the `index.html` template (in `src/main/resources/templates`).

If you haven't already, go to the app's home page to see what this looks like.

ListController

This controller provides functionality for users to see lists of all values of a given data column: employer, location, skill, and position type. If you look at the corresponding page at `/list` you'll see there's also an "All" option presented. That one doesn't work yet; you'll fully implement that view in your work.

At the top of `ListController` is a special method known as a **constructor**. We'll soon explore constructors in detail, but all you need to understand for the purpose of this assignment is that it allows for data to be initialized. We use it to populate `columnChoices` with values. This HashMap plays the same role as it did in the console app, which is to provide a centralized collection of the different list and search options presented throughout the user interface.

`ListController` also has `index`, `listColumnValues`, and `listJobsByColumnAndValue` handler methods, with routes as annotated above the method definitions. The first of these simply displays the different types of lists that the user can view. The latter two display actual data obtained from `JobData`.

In the `listColumnValues` method, the controller uses the query parameter passed in as `column` to determine which values to fetch from `JobData`. In the case of `"all"` it will fetch all job data, and then render the `list-jobs.html` view template. In all other cases, it fetches only the values for the given column and passes them to the `list-column.html` view template. We'll explore these templates in a moment.

In the `listJobsByColumnAndValue` method, we take in two query parameters: `column` and `value`. This has the net result of working similarly to the search functionality, in that we are "searching" for a particular value with in a particular column and display jobs that match. However, this is slightly different from search in that the user will arrive at this handler method as a result of clicking on a link within one of our views, rather than via submitting a form. We'll see where these links originate when we look at the views. Also note that the `listJobsByColumnAndValue` method doesn't deal with an "all" scenario; it only displays jobs matching a specific value in a specific column.

THE VIEWS

Let's turn our attention to the views.

fragments.html

Open up the `src/main/resources/templates/index.html` file in IntelliJ, and in your browser navigate to the site's home page. You'll notice that there is a fair amount of markup visible on the page that isn't contained in `index.html`. This is because we're using two fragments from `fragments.html`: `head` and `page-header`. These allow for some basic page structure and navigation to be shared across all of our views. Have a look at `fragments.html`, but know you won't have to do any work within this file for this assignment.

🎻 Pro Tip

We use **Twitter's Bootstrap (http://getbootstrap.com/)** CSS, HTML, and JS framework to provide some styling and functionality to our views. The appropriate files are included at the top of `fragments.html` and thus are included on every page of our app.

You won't have to explicitly use Bootstrap at all in this assignment, but it's a great way to make your sites look good with minimal work. Consider using it on your own projects!

List Views

Turn your attention to `list.html`. This template displays the list options, using data from the `columnChoices` dictionary passed in via the model as `columns`. The only remarkable thing in this template is how we generate the links:

```
<a th:href="@{/list/values(column=${column.key})}"
    th:text="${column.value}"></a>
```

We've seen the syntax `@{/list/values}` to generate a link within a Thymleaf template, but we haven't seen the other portion of the link: `(column=${column.key})`. This syntax will cause Thymeleaf to generate query parameters for our URL based on the key/value pairs specified.

In `list.html`, we specify a query parameter named `column` by using `column=`. The value of the query parameter is determined dynamically based on the value of `${column.key}`. Since these values come from `columnChoices` in the controller, they will be employer, location, etc. When the user clicks on these links, they will be routed to the `listColumnValues` handler in the `ListController` controller, which looks for this parameters.

In your browser, click on the Location link. This sends a request as we just outlined, resulting in a list of all of the locations in the data set. The page you're seeing at `/list/values?column=location` is generated by the `list-column.html` template. It has a similar structure as `list.html`, with the exception that the various links are presented in a table, and their URLs have not one, but two query parameter attributes: one for the column and one for the value. In the case of the locations list, these will result in URL paths like:

```
/list/jobs?column=location&value=Kansas%20City
```

(note that Thymeleaf inserts `%20` for us, to represent a space, and may actually be hidden in your browser's address bar).

Clicking on these links will display a list of jobs in the given location, via the `listJobsByColumnAndValue` handler method. However, that display isn't working yet. While the handler method is fully implemented, as we noted above, the view template needs some additional work.

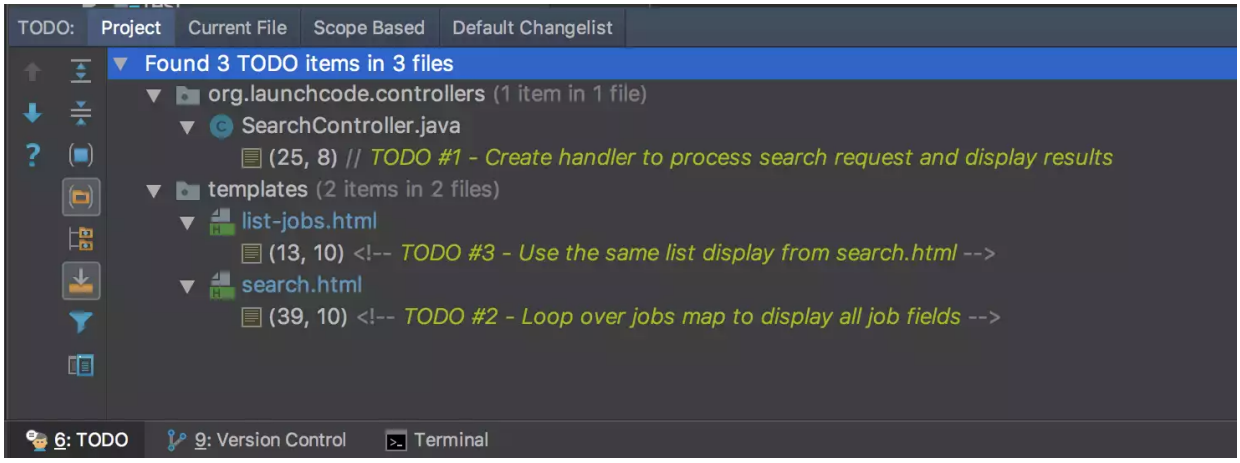
Search View

Finally, click on Search from the home page, or the navigation bar, and open up `search.html`. You'll see a search form (in both the browser and template file) that gives the user the option of searching by a given column, or across all columns. This is an exact visual analog of our console application.

This template will be used to display search results, in addition to displaying the form. This will give the nice user experience of easily searching multiple times in a row.

Your Tasks

Once you understand the controllers and views that are already in place, you're ready to begin your work. In IntelliJ, select *View > Tool Windows > TODO* to pop open a small pane at the bottom of the window. This list is populated by any code comments that start with `TODO`. In particular, you'll see your 3 tasks listed.



ADD SEARCH RESULTS HANDLER

Add another `results` handler method to `SearchController`, overloading the existing method. The method should take in two parameters, specifying the type of search and the search term. In order for the parameters to be properly passed in by Spring Boot, you'll need to name them appropriately, based on the corresponding form field names. You'll also need to use the correct annotations for the method and parameters. To configure the correct mapping route, refer to the form action in `search.html`.

After looking up the search results via the `JobData` class, you'll need to pass them into the `search.html` view via the model. You'll also need to pass `ListController.columnChoices` to the view, as the existing `search` handler does.

DISPLAY SEARCH RESULTS

After you have your `search` handler above passing data to the view, you need to display the data. Open up `search.html` and create a loop to display each job passed in from the controller. You should put each job in its own table, with job field per row.

Add the class `"job-listing"` to each of the tables to get some nice styling, courtesy of Eliot's work!

DISPLAY LIST OF ALL JOBS

Recall that the page at the path `/list/values?column=all` doesn't display any results. This page needs to display full job listings, just like the search results page. In fact, you can reuse the code you just wrote in `search.html` by defining a new fragment in that file, and then including the fragment in `list.jobs.html`.

> **⚠ Warning**
>
> For the fragment to work properly in a different file, the data passed into the view via `model.addAttribute()` must have the same key in both places.

Sanity Check

Before submitting, make sure that your application:

- Allows a user at `/search` to search for jobs matching a specific search term, both within a specific column and across all columns.
- Displays search results at the same URL as the search form.
- Displays jobs with alternately white and gray backgrounds (this is provided by the `"job-listing"` class).
- Displays a listing of all 98 jobs in the system, when the user goes to the List page and selects "All".

How to Submit

To turn in your assignment and get credit, follow the **submission instructions (../)**.

🚀 Bonus Missions

Here are some additional challenges, for those willing to take them on:

- When searching, if we select a given field to search within and submit, our choice is forgotten. Modify the view template to keep the previous search field selected when displaying results.
- The field names in the tables displaying full job data are not capitalized. Fix this. (*Hint:* We capitalize the title string in `fragments.html`, so have a look there.)
- In the search results listing and the listing of all jobs, make each value (except name) hyperlinked to the listing of all jobs with that value, as is done on the `/list/values` page.

- > **⚠ Warning**
>
> This task requires material that is covered in class 7, so only take this on if you're willing to work ahead a bit, or have some previous object-oriented programming experience.

Notice that we went to the trouble of passing in the `actionChoices` list to the view in `HomeController.index` method. This puts the responsibility of which actions should be presented on the controller, and not the view. However, we didn't go to

such lengths for the navigation links displayed on every page of the site. In order to make the navigation links similarly detached, we'd need to pass `actionChoices` in to *every* view, since the nav links are generated by `fragments.html`. We'd have to do something like the line below in every handler method, which would be a pain, not to mention error-prone and difficult to update.

```Java
model.addAttribute("actions", actionChoices);
```

Let's fix this.

1. Make a new controller, `TechJobsController`. This new controller should have a static list, `actionChoices`. The list should be populated via a no-argument/default constructor, just like `columnChoices` is populated in `ListController`.
2. Write a method `view` in `TechJobsController` with the signature

   ```Java
   public String view(Model model, String template);
   ```

3. Add code to this method to add `actionChoices` to the `model` and then return the result of calling the overridden view method via `base.View()`.
4. Modify every one of your other controllers to extend `TechJobsController`, and instead of returning the template in each handler, return the result of calling `view(model, template)`:

   ```Java
   return view(model, "template");
   ```

5. Modify `fragments.html` to use the passed-in action choices to generate the navigation links.