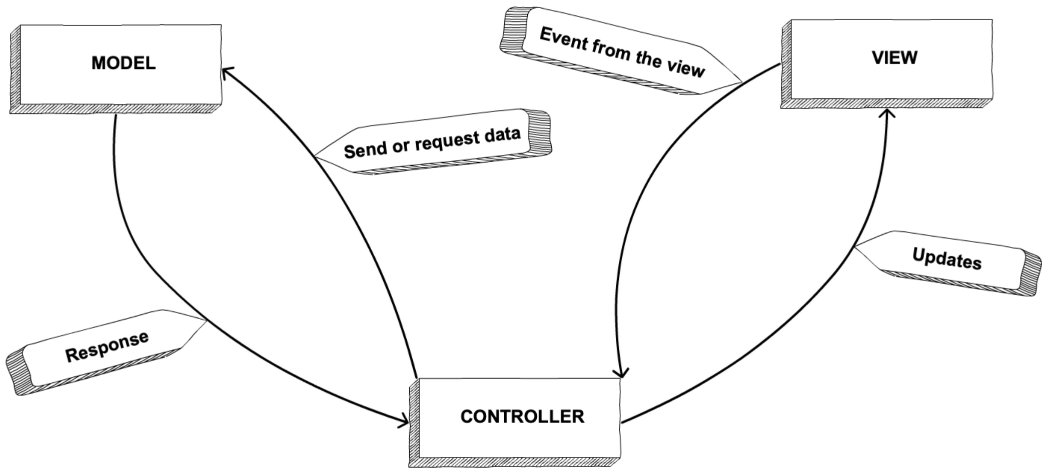
10.1. Simple Controllers

The first of the MVC elements we'll work on implementing are the controllers. Recall that controllers are like the traffic cops of our application. They handle the requests made from users interacting with the application's view and update model data accordingly. Conversely, changes to model data are sent to the view via controller methods.



MVC flow

10.1.1. @Controller

In the Spring Boot context, we'll organize controller code into a controller package. Remember when we mentioned that the framework works by convention over configuration? This is what we mean. It's not required for a controller to be in a controller package, but it's generally a good idea.

To designate a given class as a controller within the Spring framework, we use the annotation @Controller. Recall that Java Annotations are like metadata about your code. They help the framework do its work by adding context to your code.

```
@Controller
public class HelloSpringController {

   // class code here ...

}
```

10.1.2. Controllers Map to Requests

@GetMapping is another critical annotation used on controller methods. @GetMapping designates a controller action with a URL path. For every GET request made to the provided path, the controller method will be called. The path is defined with @GetMapping("pathname"). If the pathname value is null, then the path used is the index path, or /.

```
@Controller
public class HelloSpringController {

   // responds to get requests at "/hello"
   @GetMapping("hello")
   public String hello() {
      // method code here ...
   }

}
```

For every controller method that you want to respond to a request, you will want to use a mapping annotation. Not surprisingly, though, @GetMapping only handles GET requests. If you want to write a controller method that takes care of a POST request, you’ll want to use @PostMapping. Of course, there are other annotations for the other request methods, but these are the two we will use in this class.

```
@Controller
public class HelloSpringController {

   // responds to post requests at "/goodbye"
   @PostMapping("goodbye")
   public String goodbye() {
      // method code here ...
   }

}
```

If we want to write a controller method that will be used for both GET and POST at the same path, we can label the method with @RequestMapping. @RequestMapping can handle more than one method as such:

```
@Controller
public class HelloSpringController {

   // responds to get and post requests at "/hellogoodbye"
   @RequestMapping(value="hellogoodbye", method = {RequestMethod.GET,
RequestMethod.POST})
   public String hellogoodbye() {
      // method code here ...
   }

}
```

The default method of @RequestMapping is GET. Another added capability of @RequestMapping is that it can be applied to a whole class, not just a single method. When applied to a whole class, @RequestMapping essentially designates a base path that all methods in the class start with.

10.1.3. @ResponseBody

@ResponseBody is yet another annotations used in the Spring controller context to return plain text from a controller method. This annotation we will only need to use for a short while, before we start to work with templates. Spring Boot's default action when responding to a controller method is to return a template. Since we aren't doing that yet however, we need to tell the framework to return plain text by adding the @ResponseBody annotation.

Let's put it all together:

```
@Controller
public class HelloSpringController {

   // responds to get requests at "/hello"
   @GetMapping("hello")
   @ResponseBody
   public String hello() {
      return "Hello, Spring!";
   }

}
```

10.1.4. Check Your Understanding

Question

True/False: The @Controller annotation goes above a method to classify it as a controller method.

1. True
2. False

Question

Which of the following is true about controllers?

1. Controllers handle the data storage of an MVC app.
2. Controllers manage what the user of an MVC application sees.
3. Controllers relay the messages between data and views in an MVC application.
4. Controllers determine what information can be submitted in an online form.

10.2. Controllers with Parameters

Now that you kow the basics of a controller method, we can start to add some more variables into the mix. Some controller methods can take in parameters in the form of query strings or sections of the URL path. Passing this URL data to the controller is one step closer to more flexible web applications.

A **route** is the mechanism by which a request path gets assigned to a controller within our application. We'll explore routes and how data is transferred from a webpage with a given route to a specific controller.

10.2.1. Query Strings are URL Data

A brief refresher: query strings are additional bits of information tacked onto the ends of urls. They contain data in key-value pairs.

```
www.galaxyglossary.net?aKey=aValue&anotherKey=anotherValue&thirdKey=thirdValue
```

Note

Do HTTP requests and responses feel unfamiliar? Do you remember what a **query string** is? If you're feeling rusty on these topics, it's a good idea to brush up now, as routing requires a foundational understanding of HTTP data transfer.

Here's our introduction to HTTP for reviewing the concepts.

10.2.2. `@RequestParam`

We can pass `@RequestParam` as an argument of a controller method. This annotation as an argument lets the handler method know about a given query string based on its own argument.

```
// Responds to get requests at /hello?coder=LaunchCoder
@GetMapping("hello")
@ResponseBody
public String hello(@RequestParam String coder) {
   return "Hello, " + coder + "!";
}
```

The controller method looks for the query string in the URL that matches its parameter, `coder`, and puts the paired value of that `coder` key into the response text.

10.2.3. `@PathVariable`

Another way to handle data with a controller is by accessing the data via a segment of the URL. This is done with `@PathVariable`. `@PathVariable` takes an argument that, if matching a portion of the URL, will deliver this data into the handler.

```
// Responds to get requests at /hello/LaunchCode
@GetMapping("hello/{name}")
@ResponseBody
public String helloAgain(@PathVariable String name) {
   return "Hello, " + name + "!";
   }
```

```
}
```

Above, `name` is a placeholder, indicating where in the URL segment to look for the `@PathVariable`. From the comment, we know that that the actual value is `LaunchCode`, but this can easily be changed. If we changed the value of this URL segment to `/hello/Ada`, then this controller would respond with `Hello, Ada` when a `GET` request is made.

Note

Also know that you can redirect a user by removing the `@ResponseBody` annotation from the controller and returning `"redirect:/DESIREDPATH"`.

10.2.4. Check Your Understanding

Question

Your application is served at myfavoriteplanets.net. What is the path that this controller maps to?

```
@GetMapping("venus")
@ResponseBody
public String venusSurface(@RequestParam String terrestrial) {
if (terrestrial == true) {
   return "Venus is rocky."
} else {
   return "Venus is gaseous."
}
```

1. myfavoriteplanets.net/venus?terrestrial=true
2. net.myfavoriteplanets/venus?terrestrial=true
3. myfavoriteplanets/venus?terrestrial=true

3. myfavoriteplanets/venus/terrestrial

Question

Your application is served at myfavoriteplanets.net. What URL do you need to hit so that the response is: `Akatsuki currently orbits Venus.`?

```
@GetMapping("venus/{orbiter}")
@ResponseBody
public String venusOrbiter(@PathVariable String orbiter) {
   return orbiter + " currently orbits Venus."
}
```

1. myfavoriteplanets.net/venus/{Akatsuki}
2. myfavoriteplanets.net/venus/orbiter=Akatsuki
3. myfavoriteplanets.net/venus/Akatsuki
4. myfavoriteplanets.net/venus/name=Akatsuki

10.3. Controllers with Forms

What if we want to send over some form data? To send data via a simple form in Spring Boot, we'll set things up like this:

We have a controller method that generates a form at index.

```
@GetMapping
@ResponseBody
public String helloForm() {
   String html =
      "<html>" +
         "<body>" +
            "<form method = 'get' action = '/hello'>" +
               "<input type = 'text' name = 'coder' />" +
               "<input type = 'submit' value = 'Greet Me!' />" +
            "</form>" +
         "</body>" +
      "</html>";
   return html;
}
```

Remember, without an argument, `@GetMapping` maps to `/`. On form submission, the data is sent to another path, `/hello`. We need a controller for that.

```
@GetMapping("hello")
@ResponseBody
public String hello(@RequestParam String coder) {
   return "Hello, " + coder + "!";
}
```

Now, you have a controller that can handle the form submission. When the form submits, the input entered in the text box will be passed to the URL via a query string. This is why the controller method above passes in `@RequestParam` to handle the `coder` key.

To be able to submit the form via `POST`, we'll need to modify the `hello()` controller to use `@RequestMapping`. Remember, `@RequestMapping` can annotate a method that responds to both `GET` and `POST`.

```
// Responds to get and post requests at /hello?coder=LaunchCoder
@RequestMapping(value = "hello", method = {RequestMethod.GET, RequestMethod.POST})
@ResponseBody
public String hello(@RequestParam String coder) {
   return "Hello, " + coder + "!";
}

@GetMapping
@ResponseBody
public String helloForm() {
   String html =
      "<html>" +
         "<body>" +
            "<form method = 'post' action = '/hello'>" +
               "<input type = 'text' name = 'coder' />" +
               "<input type = 'submit' value = 'Greet Me!' />" +
            "</form>" +
         "</body>" +
      "</html>";
   return html;
}
```

10.3.1. Check Your Understanding

Question

From the list below, which annotations are applied above a controller class.

1. `@Controller`
2. `@GetMapping`
3. `@PostMapping`
4. `@RequestMapping`
5. `@ResponseBody`
6. `@RequestParam`
7. `@PathVariable`

Question

From the list below, which annotations are applied above a controller method.

1. `@Controller`
2. `@GetMapping`
3. `@PostMapping`
4. `@RequestMapping`
5. `@ResponseBody`
6. `@RequestParam`
7. `@PathVariable`

10.4. Class Level Controller Annotations

Once you have written several controller methods within a class, you may notice some similar behavior across handlers. This is an opportunity to DRY your code. Some of the annotations we use at the method level can also be used on whole controller classes.

We mention this earlier. If all of the methods in a controller class begin with the same root path, `@RequestMapping` can be added above the class.

```
@Controller
@RequestMapping(value="hello")
public class HelloController {

   // responds to /hello
   @GetMapping("")
    @ResponseBody
    public String hello() {
     return "Hello";
    }

   // responds to /hello/goodbye
   @GetMapping("goodbye")
    @ResponseBody
    public String helloGoodbye() {
     return "Hello, Goodbye";
    }
}
```

Note that we use `@RequestMapping` on the class. `@GetMapping` and `@PostMapping` cannot be applied at the class level.

In a related fashion, if you find that each of your methods in a controller class use `@ResponseBody` to return plain text, this annotation may be added at the top of the class, rather than at each method declaration.

10.4.1. Check Your Understanding

Question

True/False: No one controller method can handle several request types.

1. True
2. False

Question

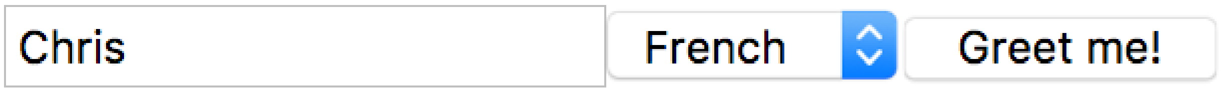
We want the method `hello` to take another parameter, `@RequestParam String friend`, that will add a friend's name to the returned greeting. The use should also be able to enter this name via a text field. What needs to be added to the form?

1. Another `input` tag with a `friend` attribute.
2. Another `input` tag with a `name` attribute.
3. Another `form` tag with a `method` attribute.
4. Another `submit` tag with a `friend` value.

10.5. Exercises: Controllers and Routing

While reading the chapter, you created a basic Hello, World application using Spring Boot called `hello-spring`. Open that project up in IntelliJ, and get ready to add some features!

Modify your `HelloController` class to display a form on a `GET` request that asks the user for both their name and the language they would like to be greeted in. It should look something like this:



Greeting Form

The resulting form submission should return and display the message, “Bonjour Chris”.

Note

The language is presented in a dropdown, more formally known as a `select` element. For more information about the `select` element and how it works, read the MDN documentation.

When the user submits the form (via a `POST` request), they should be greeted in the selected language. Your new feature should:

1. Include at least 5 languages, with English being the default. If you don’t speak 5 languages yourself, ask your friend the Internet.
2. Include a new `public static` method, `createMessage`, in the `HelloController` that takes a name as well as a language string. Based on the language string, you’ll display the proper greeting.

10.5.1. Bonus Mission

1. Instead of returning the greeting as plain text, add a bit of HTML to the response string so that the displayed message looks a bit nicer.

10.6. Studio: Skills Tracker

Wouldn't it be nice to have a small tracker to show us what skills we have built and where we are at in learning them? Let's build something that would let us do just that!

As always, read through the whole studio before starting!

At the end of the studio, your final project should be able to take input from a user via a form and post the information in a way that is easy to read.

10.6.1. Start up Spring

10.6.1.1. Using the Spring Initializr

Head to the Spring initializr to initialize a new Spring project.

1. For Project, select *Gradle Project*, for Language, select *Java*, and for Spring Boot, select *2.2.x* (whichever is the most recent release, that is NOT a snapshot).
2. For Project Metadata, we can use *org.launchcode* as Group and *skills-tracker* as the artifact. Also, remember to expand *Options* so you can make sure that Java 13 is selected!
3. Finally, we want to add *Spring Web* and *Spring DevTools* as our dependencies.

Double check that everything is ready to go and click *Generate*!

10.6.1.2. Launching IntelliJ

Unzip the newly generated skills-tracker.zip and move the folder into the directory where you are keeping your homework for this class!

Open IntelliJ and select *Import Project*. Double click *bootRun* and head to localhost:8080 to make sure there is nothing there.

Now you are ready to start coding!

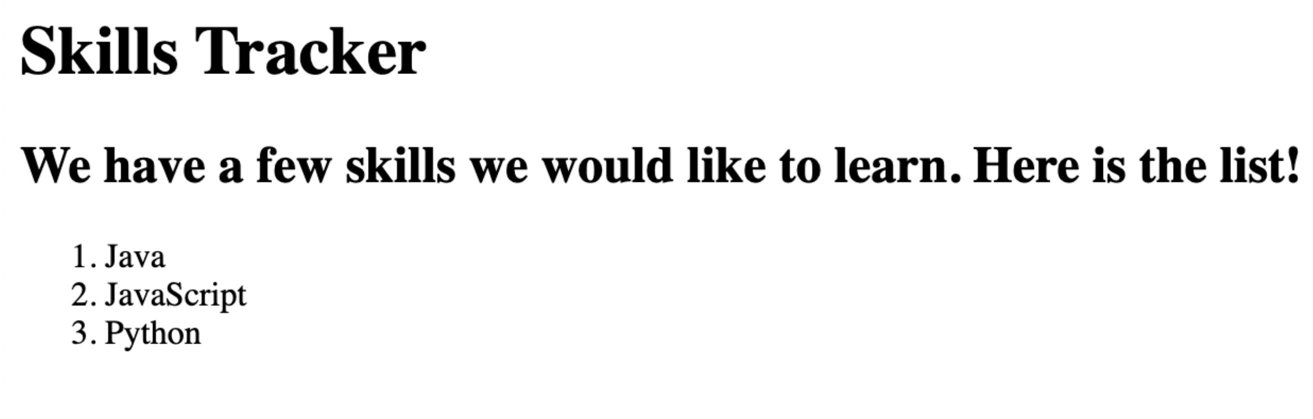
10.6.2. Creating Controllers

In your org.launchcode package, add another package called controllers and add a class called SkillsController. Inside SkillsController, you will add several methods to accomplish the followng:

1. At localhost:8080, add text that states the three possible programming languages someone could be working on. You need to have an h1 with the title "Skills Tracker", an h2, and an ol containing three programming languages of your choosing.
2. At localhost:8080/form, add a form that lets the user enter their name and choose their favorite, second favorite, and third favorite programming languages on your list. Use select elements for each of the rankings. Just as with the exercises, we will use @GetMapping().
3. Also at localhost:8080/form, use @PostMapping and @RequestParam to update the HTML with an h1 stating the user's name and an ol showing the three programming languages in the order they chose.

10.6.3. End Result

At the end of the studio, when you navigate to localhost:8080, you should see the following:

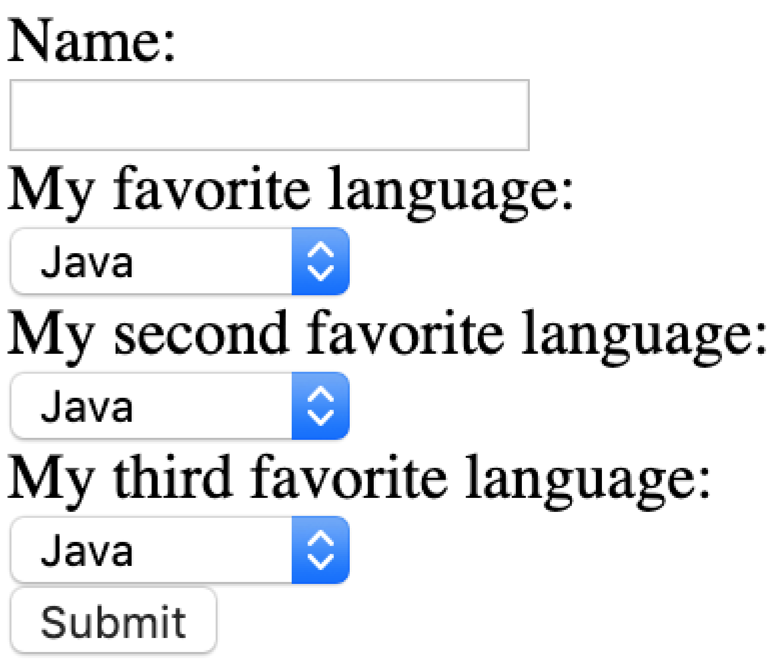


Skills Tracker

We have a few skills we would like to learn. Here is the list!

1. Java
2. JavaScript
3. Python

When you navigate to localhost:8080/form, you should see a blank form that looks something like:



Name:

My favorite language:

Java

My second favorite language:

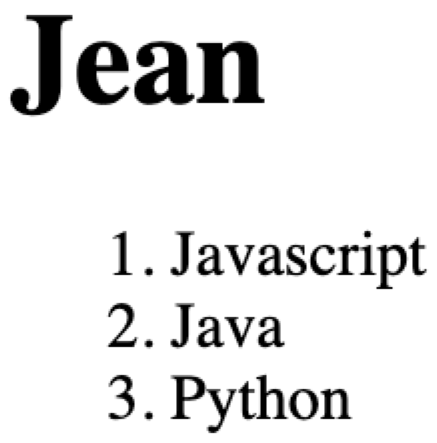
Java

My third favorite language:

Java

Submit

If you fill out the form, your page may render like so:


Jean

1. Javascript
2. Java
3. Python

10.6.4. Bonus Missions

1. Reformat your form page to use a table instead of an ordered list.
2. Add a new path to the site to display the information from the completed form.