11.1. Templates

Take a look at the homepage for WebElements. The content includes text, images, a navigation bar, a search box, linked menu options at the bottom of the page, and 118 carefully colored boxes with links—one for each element on the periodic table. All of this content is very deliberately arranged and styled.

Imagine your boss tasks you with creating this website. Setting up the HTML tags for the navigation bar would be straightforward, but what about the element boxes? You would need to make 118 similar structures, but with different text, links, and colors. Trying to make the table structure work would be tedious at best, and excruciatingly difficult at worst.

Also, what if a new element gets discovered, or some of the data for the elements changes? Updating the text, colors, layout, etc. means adjusting those items in the HTML. Also, if that information appears in other areas of the website, then you need to modify that code as well.

Whew! Changing the website rapidly becomes problematic, especially since it contains lots of data and consists of multiple pages. This is where templates come in to play. They help automate the tasks required to build and maintain a website.

11.1.1. Templates are Frameworks

A **template** provides the general structure for a web page. Templates outline for us and Spring where different elements get placed on the page. Any page made with a template includes its elements and follows its rules. If we add content to the template or modify it in some way, all pages made from that template will reflect the changes.

Let’s see how using a template makes our lives easier.

11.1.1.1. No Template

The code below displays a simple list. It defines the location for the heading and each <li> element, in addition to a couple of fun links. The CSS file (not shown) specifies the font, text size, colors, etc.

```
<body>
   <h1>Java Types</h1>
   <div class="coffeeList">
      <ol>
         <li>French Roast</li>
         <li>Espresso</li>
         <li>Kopi Luwak</li>
         <li>Instant</li>
      </ol>
   </div>
   <hr>
   <div class="links">
      <h2>Links</h2>
      <a href="https://www.launchcode.org/">LaunchCode</a> <br>
      <a href="https://en.wikipedia.org/wiki/Coffee">Coffee</a>
   </div>
</body>
```

We could drastically improve the appearance and content of the page by playing around with the tags, classes, styles and text. However, any change we want to make needs to be coded directly into the HTML and CSS files, and this quickly becomes inefficient.

11.1.1.2. A Better Way

Recall that a template represents a *view* in the MVC world. It sets up a structure to display the data delivered by the controller, and the template guides where that information goes. This provides much more flexibility than hard-coding, since data can change based on a user's actions.

```
<body>
   <h1 {templateInstructions}></h1>
   <div class="coffeeList">
      <ul>
         <li {templateInstructions}></li>
      </ul>
   </div>
   <hr>
   <div class="links">
      <h2>Links</h2>
      <a {templateInstructions}></a>
   </div>
</body>
```

This HTML looks similar to the previous example, but it replaces some of the code with *instructions*.

{templateInstructions} refers to instructions and data passed into the template by the controller. These will automatically create HTML tags in order to display or update the information.

By using a template to build the website, changing the list involves altering something as simple as an ArrayList or object. After changing that data, the template does the tedious work of modifying the HTML.

11.1.1.3. Templates Support Dynamic Content

Besides making it easier to organize and display content, templates also allow us to create a *dynamic* page. This means that its appearance changes to fit new information. For example, we can define a grid for displaying photos in rows of 4 across the page. Whether the images are of giraffes, tractors, or balloons does not matter. The template sets the layout, and the code feeds in the data. If more photos are found, extra rows are produced on the page, but each row shows 4 images.

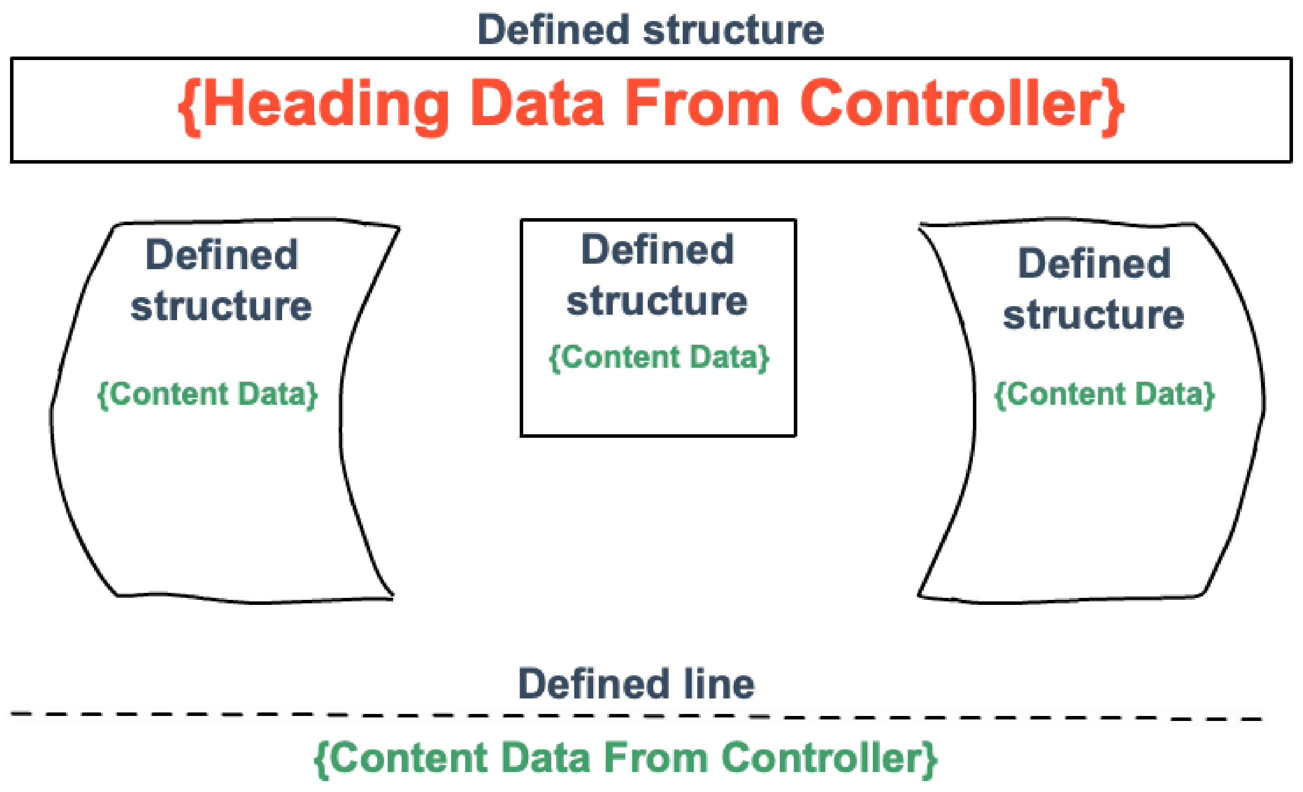
In the last lesson, you built a simple website that displayed a welcome message and responded to changing values for a user's name. You did NOT apply a template for this page, and it is possible to create an interactive site without one. However, as your projects grow in size, templates make it MUCH easier to maintain your work.

Tip

Use templates when building a web-based project.

11.1.2. Templates Provide Structure, Not Content

Templates allow us to decide how to display data in the view, even if we do not know exactly what that data will be. Information pulled from forms, APIs, or user input will be formatted to fit within our design.



In the figure, the black outlines represent different areas defined by the template—spaces for lists, images, links, etc. As the controller feeds data into the template, the appearance of the page changes.

Note

If the template expects data for a list, but the controller does not provide the information, that part of the screen remains empty.

11.1.3. Check Your Understanding

Question

Why should we use a template to design a web page rather than just coding the entire site with HTML and CSS?

11.2. Creating a Template

Using templates is a useful way to reduce the effort required to create and maintain a web-based project. Before you can dive into using templates, however, you need to take care of a little groundwork first.

11.2.1. Thymeleaf

In combination with Java and Spring Boot, we will use a library called **Thymeleaf**. This set of tools helps simplify the creation of flexible, reusable templates for standalone projects and web-based applications.

More information can be found on the introduction page of the Thymeleaf website.

11.2.1.1. Thymeleaf, Naturally

The developers behind Thymeleaf emphasize the idea of *natural templates*. This means that templates constructed with Thymeleaf look and operate just like regular HTML code. You can open any template in a browser and view it just like a static HTML file.

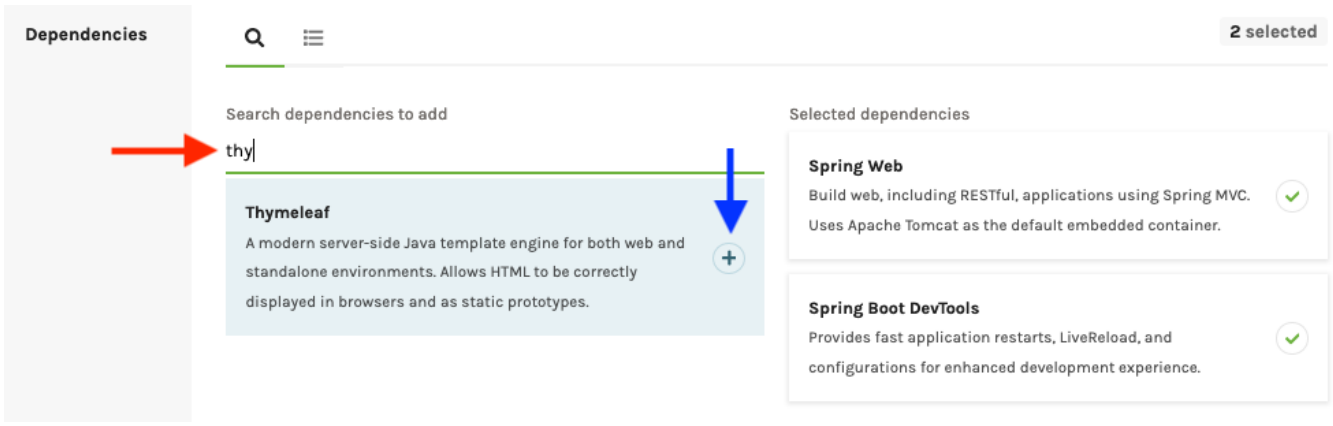
Any logic we add to a template occurs inside the tags, which preserves the ability to open the file and display it correctly. This helps us as we consider the best layout for presenting data on the screen.

11.2.2. Thymeleaf Dependency

In this chapter, you will construct some small practice projects to help you learn how to implement Thymeleaf templates. To use the library, however, you need to provide IntelliJ with some information to make the IDE recognize Thymeleaf syntax and commands. You need to include in the proper *dependencies*, and there are two common ways to accomplish this.

11.2.2.1. During Setup for a New Project

When you create a new Gradle project on the start.spring.io website, search for and select the *Thymeleaf* dependency. For the practice projects you build in this class, be sure to add the *Spring Web* and *Spring Boot DevTools* dependencies as well.



Thymeleaf
A modern server-side Java template engine for both web and standalone environments. Allows HTML to be correctly displayed in browsers and as static prototypes.

Spring Web
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Spring Boot DevTools
Provides fast application restarts, LiveReload, and configurations for enhanced development experience.

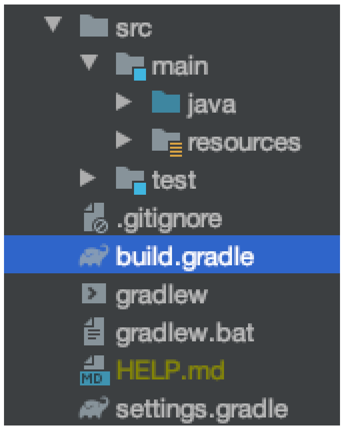
In IntelliJ, opening up the `build.gradle` file of the new project shows the dependencies you selected:

```
   dependencies {
22    implementation 'org.springframework.boot:spring-boot-starter-thymeleaf'
23    implementation 'org.springframework.boot:spring-boot-starter-web'
24    developmentOnly 'org.springframework.boot:spring-boot-devtools'
25    testImplementation('org.springframework.boot:spring-boot-starter-test') {
26       exclude group: 'org.junit.vintage', module: 'junit-vintage-engine'
27    }
28 }
29
```

Lines 23 - 25 establish links to the `thymeleaf`, `spring-boot-starter-web`, and `spring-boot-devtools` libraries, respectively.

11.2.2.2. Add to an Existing Project

If you have an existing project that does not currently use Thymeleaf, you can add the functionality by updating the `build.gradle` file.



In the `dependencies` block, just paste in the `implementation` statement seen in line 23 above. Also, be sure to include the Spring Boot libraries if the old project is missing those as well.

11.2.3. Start a Practice Project

Open up your `hello-spring` project in IntelliJ and code along with the following video.

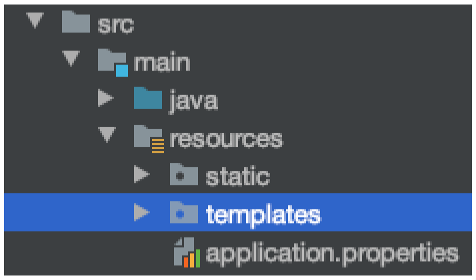
Warning

This videos in this chapter walk you through building small web-based projects. Do NOT skip this practice, because the end of chapter exercises pick up where the tutorials end.

The sections below outline the main ideas presented in the video. However, the text is NOT a substitute for completing the work described in the clip.

11.2.3.1. Template Location

In IntelliJ, create a `templates` folder inside the `resources` directory. All of your templates should be put in the `templates` folder. As you saw in the video, Thymeleaf streamlines finding required files by using `resources/templates` as the default location.



11.2.3.2. Add a Template

To create a new template, right-click on the `templates` folder and select *New –> HTML file*. Give your template a descriptive name, and note that IntelliJ provides some boilerplate code:

```
   <!DOCTYPE html>
 1 <html lang="en">
 2 <head>
 3    <meta charset="UTF-8">
 4    <title>Title</title>
 5 </head>
 6 <body>
 7
 8 </body>
 9 </html>
10
```

To tie in the Thymeleaf information, you need to add a URL inside the `html` tag on line 2:

```
<html lang="en" xmlns:th="https://www.thymeleaf.org/">
```

The `xmlns:th` attribute pulls in information from `thymeleaf.org` about the keywords and methods that we will use with Thymeleaf. This allows IntelliJ to properly apply syntax highlighting and error reporting.

A side effect of the `xmlns:th` attribute is that it implements stricter requirements for closing HTML tags. In HTML5, we can get away with leaving out the `/` character in standalone tags like `<input>`. However, with the xml format in Thymeleaf, we must include the character.

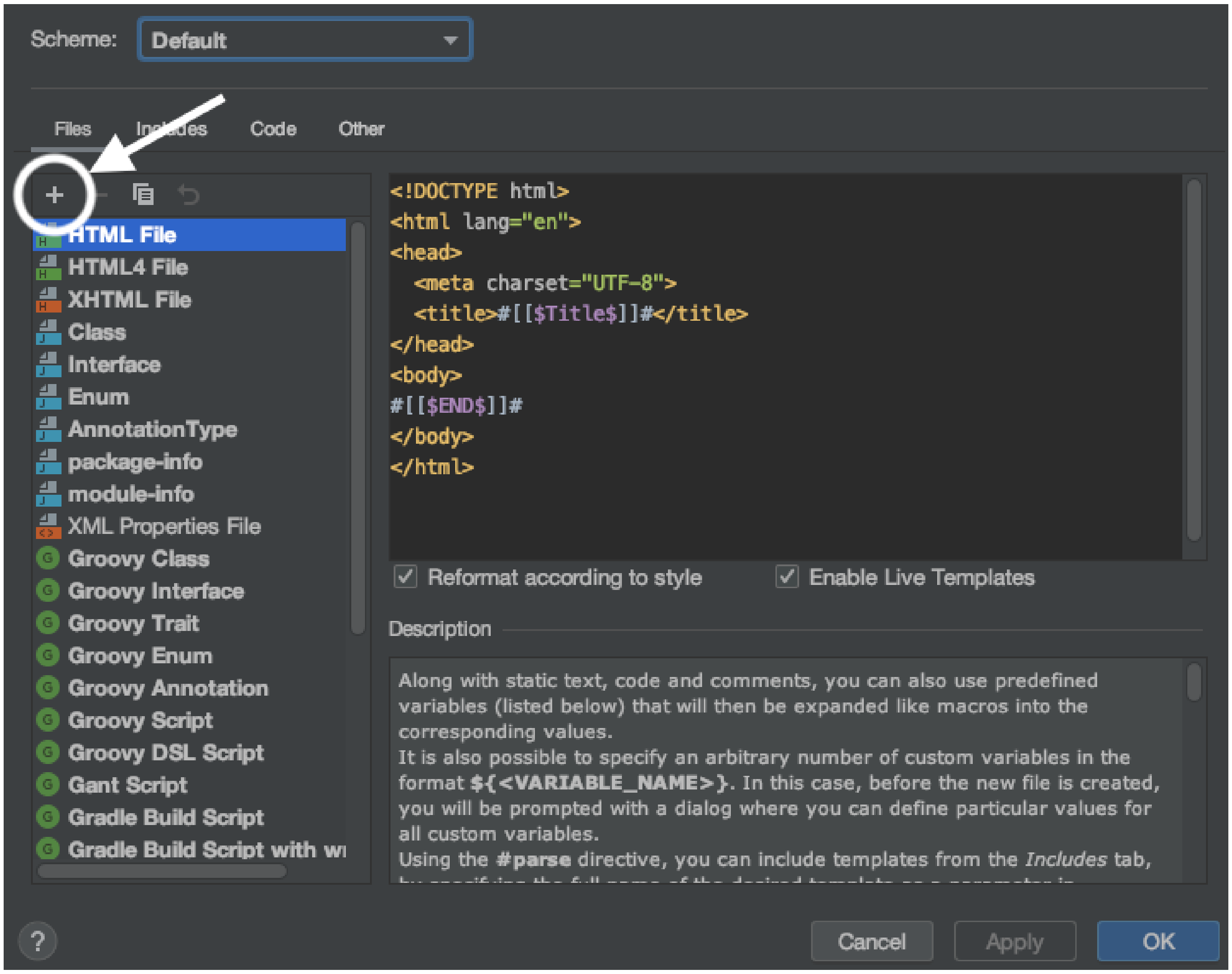
For example, we need to update line 4 in the boilerplate HTML code to close the `meta` tag:

```
<meta charset="UTF-8" /> <!-- Note the closing '/' character -->
```

11.2.4. Thymeleaf Template

As described in the video, you can save yourself some time by creating your own boilerplate code for a Thymeleaf template. This will save you from having to make the edits described above every time you add a new base html file.

1. Right-click on the `templates` folder (or any other directory), and select *New –> Edit File Templates*.
2. In the window that pops up, click the "+" icon to add a new file.

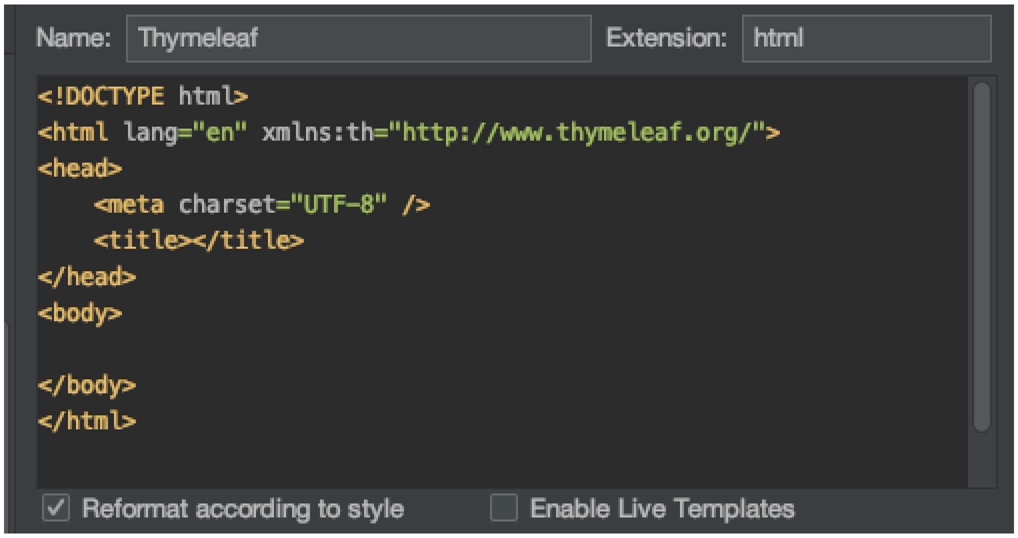


```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>#[[$Title$]]#</title>
</head>
<body>
#[[$END$]]#
</body>
</html>
```

☑ Reformat according to style ☑ Enable Live Templates

Along with static text, code and comments, you can also use predefined variables (listed below) that will then be expanded like macros into the corresponding values.
It is also possible to specify an arbitrary number of custom variables in the format **${<VARIABLE_NAME>}**. In this case, before the new file is created, you will be prompted with a dialog where you can define particular values for all custom variables.
Using the **#parse** directive, you can include templates from the *Includes* tab,

3. Name your template, set the extension as `html`, then edit the starter code. This will be the boilerplate HTML that appears anytime you select your custom template. For Thymeleaf, the code should at least close the `meta` tag and include the `xmlns` attribute.

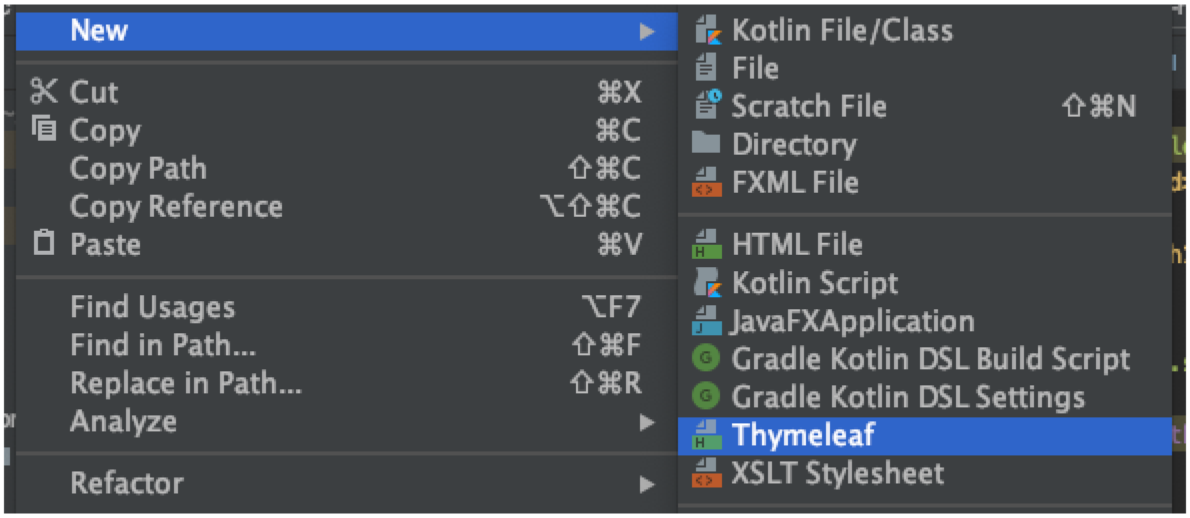


```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org/">
<head>
    <meta charset="UTF-8" />
    <title></title>
</head>
<body>

</body>
</html>
```

If you find yourself routinely using other code in your Thymeleaf files, you can return to this window and edit the HTML as needed. Don’t forget to save your changes.

4. To use your custom Thymeleaf template, right-click on the `templates` folder and select *New –> TemplateName*.



11.3. Using a Template

In the video at the bottom of this page, you will extend your `hello-spring` project and practice using Thymeleaf templates as you code along with the clip.

Before you get there, however, let’s review one core skill and two best-practices.

11.3.1. Passing Data to a Template

The controller class contains methods that send data to different templates. These methods have a structure similar to:

```
1 public String methodName(Model model) {
2
3    // method code here
4
5    model.addAttribute("variableName", variableValue);
6
7    return "pathToTemplate";
8 }
```

Each method that sends data to a template requires:

1. A `Model` parameter (line 1). This object stores the variable names and values that get passed into a template.
2. One or more `addAttribute` statements that add data to the `model` object (line 5). `variableName` must be a string. `variableValue` can be a variable of any type, a number, a ‘character’, or a “string”.
3. A `return` statement of type `String`.
4. The `return` string contains the path and file name for the desired template (line 7). For example, if our `templates` folder contains a subfolder called `animals` that holds a template called `dogs.html`, then the return statement would be `return "animals/dogs";`.

11.3.1.1. Accessing Data in a Template

Thymeleaf commands appear (in most cases) as *attributes* within a standard HTML tag. Each command starts with the prefix `th`, followed by a keyword.

```
<tag th:keyword = "..."></tag>
```

The code inside the quotation marks will vary depending on the keyword. Usually, this will involve the data passed in from the controller. The data available to a template includes any variables and values stored in the `model` object, and these will be accessed with the syntax `${variableName}`.

For example, if the controller stores data in the `name` variable, then the template can display that value like so:

```
<p th:text = "${name}"></p>
```

In essence, `th:text` says, *Take the value of the variable inside the ${} and display it as the text for this element*. If `name` stores the string “Rutabaga”, then when the program runs, the code interprets `<p th:text = "${name}"></p>` in the same way as:

```
<p>Rutabaga</p>
```

By using `th:text = "${name}"`, we make our webpage *dynamically* display data within the `p` element. Changing the value of `name` leads to a corresponding change in the text in the view after refreshing.

11.3.2. Setting Default Text

What if we want to view our template without running the program? As mentioned earlier in the chapter, Thymeleaf files can be opened in a browser and displayed as a webpage. With no data, however, opening a template shows blank spaces wherever information from the controller is expected. Fortunately, we can add some default text within the framework to give us an idea of what the page will look like when we start the program.

To include default text in our template, simply include some *placeholder* information inside the elements that contain a Thymeleaf attribute.

```
<h2 th:text = "${title}">Default Title</h2>
<div>
   <p th:text = "${bookQuote}">Don't Panic</p>
   <a href = "someURL" th:text = "${linkText}">LaunchCode</a>
</div>
```

The text `Default Title`, `Don't Panic`, and `LaunchCode` appear if we open the template file in a browser. When the program runs, however, these text samples will be replaced by the values stored in the `title`, `bookQuote`, and `linkText` variables.

In most cases, you will never see the default text in your live webpage. Including it helps, however, if you need to visualize your planned layout for the webpage before your project is completely finished.

Tip

Best-practice encourages us to include default text in our templates. This improves the readability of the code, and it gives an outside observer a better idea of the structure of the webpage. Default text also indicates what data will appear in different sections.

11.3.3. Organizing Templates

As any project grows, the number of templates required to build the website will increase. Instead of just throwing all of the files into the `templates` folder, best-practice mandates that we place related items into subfolders.

For example, if we build a website for a zoo, we can help ourselves immensely if we avoid a `templates` folder with a single level of files for every animal or feature of the site. A better approach would be to divide the templates into related categories like `feedingSchedules`, `concessions`, `donations`, `pachyderms`, etc. Each subfolder can also hold finer categories as needed.

The goal is to consolidate your files into related groups. That way, you only need to use a single file path in a given controller. This improves the efficiency of your code, saves you from getting a headache trying to find and fix a specific file, and streamlines updates by reducing the lines of code that need to be modified.

11.3.4. Check Your Understanding

Question

Given the code:

```
<p th:text = "${name}">Name: Default</p>
```

What will be displayed on the screen if the controller sends in a `name` variable with a value of “Blake”?

1. Name: Default
2. Name: Blake
3. Blake: Default
4. Blake

Question

We want a list element to read, “Item name: ______, Price = ______”, where the blanks need to be filled in with `name` and `price` values sent from the controller.

Which of the following will produce the desired result?

1. `<li th:text = "${'Item name: ' + name + ', Price = ' + price}"></li>`
2. `<li th:text = "Item name: ${name}, Price = ${price}"></li>`
3. `<li th:text = "${name}, ${price}">Item name: , Price = </li>`
4. `<li>Item name: ${name}, Price = ${price}</li>`

11.4. Iterating in a Template

Let’s revisit part of the non-efficient HTML from the introduction, where we hard-coded coffee options in a list.

```
<ol>
   <li>French Roast</li>
   <li>Espresso</li>
   <li>Kopi Luwak</li>
   <li>Instant</li>
</ol>
```

If we want to add, remove, or edit the list items, we need to go in and change the individual tags, which is a poor use of our time. Fortunately, there is a way to streamline the process.

In Java, we use `for/each` loops to iterate through the items in a data collection.

```
for (Type item : collectionName) {
   // Code to repeat...
}
```

Thymeleaf gives us a similar functionality to use in our templates, but we need to learn a different syntax.

11.4.1. `th:each`

The general syntax for iterating through a collection in Thymeleaf is:

```
th:each = "variableName : ${collectionName}"
```

1. The `th:each` statement gets placed INSIDE an HTML tag.
2. `collectionName` represents an ArrayList, HashMap, or other collection.
3. `variableName` represents an individual item or element within the collection.
4. The `${}` specifies the data provided by the controller.

We can think of a `th:each` statement as saying, *“For each item within ‘collectionName’, repeat this HTML tag, but use the current value of ‘variableName’.”*

Let’s apply `th:each` to the HTML example above. Assume that we store each of the coffee names as a string in an ArrayList called `coffeeOptions`.

```
<ol>
   <li th:each="item : ${coffeeOptions}" th:text="${item}"></li>
</ol>
```

Some points to note:

1. `coffeeOptions` is passed into the template by the controller.
2. In the first pass through the loop, `item` takes the value of the first coffee name in the ArrayList.
3. `th:text="${item}"` displays the value of `item` in the view, so the `li` element will show the first coffee name.
4. Each successive iteration, `item` takes the next value in the list, and Thymeleaf adds a new `<li></li>` element to display that data.

11.4.1.1. Location Matters

`th:each` creates one HTML tag for each item in a collection, and only one `th:each` statement can be placed in a given tag. BE CAREFUL where you place it.

Consider the following code blocks:

Examples

Option #1:

```
<div>
   <h3>Java Types</h3>
   <ol>
      <li th:each="item : ${coffeeOptions}" th:text="${item}"></li>
   </ol>
</div>
```

Option #2:

```
<div>
   <h3>Java Types</h3>
   <ol th:each="item : ${coffeeOptions}">
      <li th:text="${item}"></li>
   </ol>
</div>
```

Option #3:

```
<div th:each="item : ${coffeeOptions}">
   <h3>Java Types</h3>
   <ol>
      <li th:text="${item}"></li>
   </ol>
</div>
```

Which option produces:

1. One heading, 4 ordered lists, and 4 coffee names (each name labeled as “1”)?
2. One heading, one ordered list, and 4 coffee names (with the names labeled 1, 2, 3…)?
3. 4 headings, 4 ordered lists, and 4 coffee names (each name labeled as “1”)?

11.4.2. `th:block`

`th:block` is a Thymeleaf *element* rather than an attribute, and it has the general syntax:

```
<th:block>
   <!-- Your HTML code -->
</th:block>
```

`th:block` allows you to apply the same Thymeleaf attribute, like `th:if` or `th:each` to a block of code. For example, assume you have an ArrayList of coffee objects, each of which contains a `name` and `description` field. If you want to display this information as name/description pairs, you might try:

```
<h3 th:each = "coffee : ${coffeeOptions}" th:text = "${coffee.name}">Coffee Name</h3>
<p th:each = "coffee : ${coffeeOptions}" th:text = "${coffee.description}">Info</p>
```

However, `th:each` operates on and repeats a single HTML element. Instead of alternating name/description pairs, the view in this case displays `h3` headings for all of the coffee names *before* rendering the `p` elements for the descriptions.

To replicate the `h3` and `p` elements as pairs, you need to wrap them inside `th:block`:

```
<th:block th:each = "coffee : ${coffeeOptions}">
   <h3 th:text = "${coffee.name}">Coffee Name</h3>
   <p th:text = "${coffee.description}">Info</p>
</th:block>
```

Set up this way, the code gets interpreted as, *"Repeat the following HTML elements once for each item in 'coffeeOptions'."* Each iteration of the loop renders a heading/paragraph pair using a new `coffee` object from `coffeeOptions`.

Note

Unlike a standard HTML tag (like `div`, `ul`, `p`, etc.), the `th:block` element does NOT get rendered in the view.

Tip

To see an example for how to use `th:block` and iteration to produce a table, check the Thymeleaf docs.

11.4.2.1. Readability and Nested Loops

`th:block` helps clarify your HTML code by placing related Thymeleaf attributes into separate tags.

Examples

This code generates multiple `li` elements and text by putting two attributes inside of the tag:

```
<ol>
   <li th:each="item : ${coffeeOptions}" th:text="${item}"></li>
</ol>
```

The code below accomplishes the same thing, but it separates the loop command from the `th:text` command:

```
<ol>
   <th:block th:each="item : ${coffeeOptions}">
      <li th:text="${item}"></li>
   </th:block>
</ol>
```

The second format more closely mimics the appearance of a Java `for/each` loop, which improves the clarity of your code.

```
for (Coffee item : coffeeOptions) {
   // Code using the "item" variable...
}
```

`th:block` also allows you to set up nested loops:

Example

Assume you have a collection of different coffee shop objects, and each object stores a `name` and a list of `coffeeOptions`. To display a list of lists:

```
<th:block th:each = "shop : ${shops}">  <!-- Iterate through shops -->
   <p th:text = "${shop.name}">Shop name</p>
   <ul>
      <!-- Iterate through coffeeOptions -->
      <th:block th:each = "flavor : shop.coffeeOptions">
         <li th:text="${flavor}"></li>
      </th:block>
   </ul>
</th:block>
```

11.4.3. Check Your Understanding

Use the three code samples listed in the Location Matters section to answer the following questions:

Question

Which option produces one heading, 4 ordered lists, and 4 coffee names (each name labeled as “1”)?

1. Option 1
2. Option 2
3. Option 3

Question

Which option produces one heading, one ordered list, and 4 coffee names (with the names labeled 1, 2, 3…)?

1. Option 1
2. Option 2
3. Option 3

Question

Which option produces 4 headings, 4 ordered lists, and 4 coffee names (each name labeled as “1”)?

1. Option 1
2. Option 2
3. Option 3

11.5. Conditionals in a Template

In addition to iteration, Thymeleaf can also add or remove content on a webpage based on certain conditions. Going back to the coffee example, we could generate the ordered list ONLY IF `coffeeOptions` contains data. If the ArrayList is empty, then there is no need to include the `<ol>` element. Instead, the template could include a `<p>` element with text stating that there are no options to select.

Just like the `for/each` syntax differs between Java and Thymeleaf, we need to examine how to include *conditionals* in our templates. The logic remains the same, but the implementation requires practice.

11.5.1. Display Content `if`

The general syntax for including a conditional in Thymeleaf is:

```
th:if = "${condition}"
```

1. The `th:if` statement gets placed inside an HTML tag.
2. `condition` represents a boolean variable provided by the controller.
3. Alternatively, `condition` can be a statement that evaluates to `true` or `false`.

If `condition` evaluates to `true`, then Thymeleaf adds the HTML element to the webpage, and the content gets displayed in the view. If `condition` is `false`, then Thymeleaf does NOT generate the element, and the content stays off the page.

Example

Assume that `coffeeOptions` and `userSelection` represent an ArrayList and String, respectively, that are passed in by the controller.

```
1  <ol th:if = "${coffeeOptions.size() > 1}">
2     <li th:each="item : ${coffeeOptions}" th:text="${item}"></li>
3  </ol>
4
5  <h2 th:if = '${userSelection.equals("instant")}'>You can do better!</h2>
```

The conditional in line 1 checks that `coffeeOptions` contains more than one item. If `true`, then the ordered list is rendered in the view. The `th:if` statement in line 5 compares a user’s flavor choice to the string `"instant"`. If they match, then Thymeleaf adds the heading element to the view.

11.5.1.1. Adding vs. Displaying/Hiding

`th:if` determines if content is *added or not added* to a page. This is different from deciding if content should be *displayed or hidden*.

Hidden content still occupies space on a page and requires some amount of memory. When `th:if` evaluates to `false`, content remains absent from the page—requiring neither space on the page nor memory. This is an important consideration when including items like images or videos on your website.

11.5.1.2. What is `true`?

The `th:if = "${condition}"` attribute can evaluate more than simple boolean variables and statements. It will also return `true` according to these rules:

1. If `condition` is a boolean or statement and `true`.
2. If `condition` is a non-zero number or character.
3. If `condition` is a string that is NOT `"false"`, `"off"`, or `"no"`.
4. If `condition` is a data type other than a boolean, number, character, or String.

`th:if` will evaluate to `false` whenever `condition` is `null`.

11.5.1.3. `unless` Instead of `else`

In Java, we use an `if/else` structure to have our code execute certain steps when a variable or statement evaluates to `true` but a different set of steps for a `false` result. Thymeleaf provides a similar option with `th:unless`:

```
th:unless = "${condition}"
```

Just like `th:if`, the `th:unless` attribute gets placed inside an HTML tag. In this case, however, Thymeleaf adds the HTML element to the webpage when `condition` evaluates to `false`.

We could update our coffee code with an `unless`:

Example

```
<h2 th:unless = '${userSelection.equals("instant")}'>Excellent choice!</h2>
```

As long as `userSelection` is NOT `"instant"`, the condition evaluates to `false`, and the `h2` element gets added to the view.

If we want to set up a situation like *if true, do this thing. Otherwise, do this other thing*, we need to pair a `th:if` with a `th:unless`.

Example

```
<ol th:if = "${coffeeOptions.size()}">
   <li th:each="item : ${coffeeOptions}" th:text="${item}"></li>
</ol>

<p th:unless = "${coffeeOptions.size()}">No coffee brewed!</p>
```

If `coffeeOptions.size()` evaluates to 0, then Thymeleaf considers it a `false` result. In that case, it ignores the `ol` element and generates the `p` element.

11.5.1.4. Logical Operators

We can use logical operators with `th:if` and `th:unless`. The Thymeleaf syntax for these is as follows:

1. Logical AND = `and`,
2. `th:if = "${condition1 and condition2 and...}"`
3. `// Evaluates to true if ALL conditions are true`
4. Logical OR = `or`,
5. `th:if = "${condition1 or condition2 or...}"`
6. `// Evaluates to true if ANY condition is true`
7. NOT = `!`, `not`.
8. `th:if = "${!condition}"`
9. `// Evaluates to true when condition is false`

Note

Since `th:unless` looks for a `false` result, we can accomplish the same thing by adding a `not` operator to a `th:if` statement.

The code:

```
<p th:unless = "${variableName == 5}">Value is NOT equal to 5.</p>
```

does the same thing as:

```
<p th:if = "${variableName != 5}">Value is NOT equal to 5.</p>
```

11.5.2. Try It!

The video below provides you some live-coding practice with Thymeleaf templates. Return to your `hello-spring` project and code along as you watch the clip.

The text on this page and the previous two provides details for some of the concepts presented in the clip. Note that these summaries are NOT intended as a replacement for the walkthrough. To get better at coding, you need to actually CODE instead of just reading about how to do it.

11.5.3. Check Your Understanding

Assume you have an ArrayList of integers called `numbers`, and you display the values in an unordered list.

```
<ul>
   <th:block th:each = "number : ${numbers}">
      <li th:text = "${number}"></li>
   </th:block>
</ul>
```

Question

You want to display the list only if `numbers` contains data. Which of the following attributes should you add to the `ul` tag?

1. `th:if = "${numbers.size()}"`
2. `th:unless = "${numbers.size()}"`

Question

Now you want to display ONLY the positive values in the list. Which of the following attributes could you add to the `li` tag? Select ALL that work.

1. `th:if = "${number}"`
2. `th:if = "${number < 0}"`
3. `th:if = "${number > 0}"`
4. `th:unless = "${number}"`
5. `th:unless = "${number >= 0}"`
6. `th:unless = "${number <= 0}"`

Question

Now you want to display ONLY the positive, even values in the list. Which of the following should you add to the `li` tag?

1. `th:if = "${number > 0 and number%2 == 0}"`
2. `th:if = "${number > 0 or number%2 == 0}"`
3. `th:unless = "${number < 0 and number%2 == 0}"`
4. `th:unless = "${number < 0 or number%2 == 0}"`

11.6. Thymeleaf Forms

Templates allow you to build generic forms. This lets you reuse the structure by rendering the same template with different labels and data. Thus, a single form can serve different purposes, saving you extra effort.

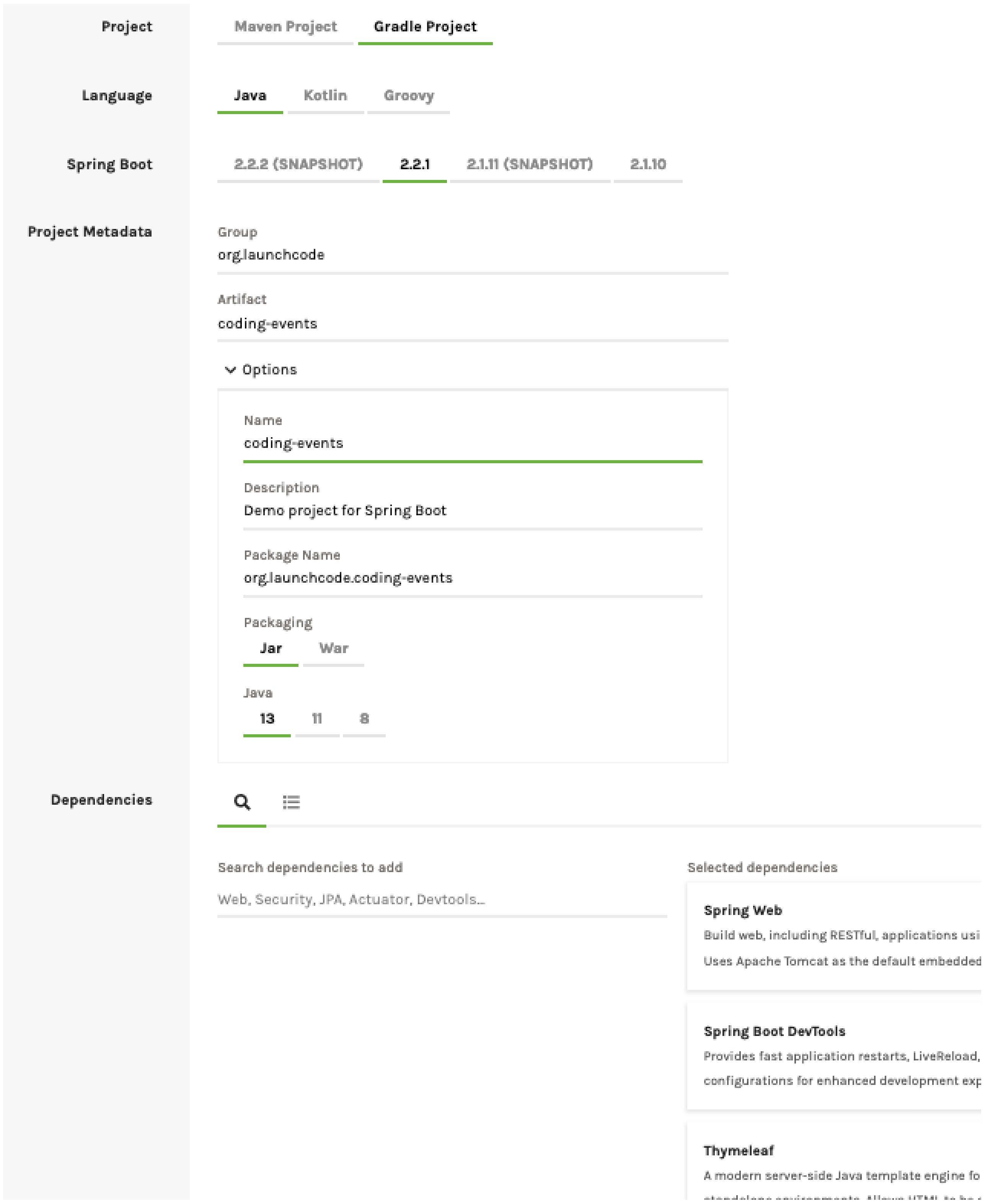
Whenever possible, reuse existing templates!

11.6.1. Start a New Project

You will build a new project so you can practice with templates and forms. If you have not done so, commit and push any unsaved work from your `hello-spring` project.

Your new project will keep track of some fictional coding events.

1. Use the start.spring.io website to initialize your new project.
2. Follow the steps you used to setup hello-spring, but call the new project `coding-events`.
3. Be sure to add the *Thymeleaf* dependency in addition to *Spring Web*, and *Spring Boot DevTools*.



Project	Maven Project [✓ Gradle Project]
Language	[✓ Java] Kotlin Groovy
Spring Boot	2.2.2 (SNAPSHOT) [✓ 2.2.1] 2.1.11 (SNAPSHOT) 2.1.10
Project Metadata	Group: org.launchcode
	Artifact: coding-events
	Options
	Name: coding-events
	Description: Demo project for Spring Boot
	Package Name: org.launchcode.coding-events
	Packaging: [✓ Jar] War
	Java: [✓ 13] 11 8
Dependencies	Search dependencies to add: Web, Security, JPA, Actuator, Devtools...

Selected dependencies

Spring Web
Build web, including RESTful, applications usi
Uses Apache Tomcat as the default embedded

Spring Boot DevTools
Provides fast application restarts, LiveReload,
configurations for enhanced development exp

Thymeleaf
A modern server-side Java template engine fo

4. *Generate* the `.zip` file and then import it into IntelliJ.

11.6.2. Try It!

Now that you have `coding-events` up and running, add features to it by coding along with the videos below:

Before moving on, be sure to commit and push your changes. Do this after each video to create a fallback position just in case disaster strikes your project in the future.

A summary of Thymeleaf forms is given below, but remember that the text supports the videos and is NOT intended as a replacement.

11.6.3. Create and Render a Form

A Thymeleaf form is simply a template that includes a `<form>` element inside the `body` of the HTML. The method for the form should be of type `post`.

```
<body>

   <!-- Other HTML -->

   <form method="post">
      <input type="text" name="inputName">
      <input type="submit" value="submitButtonText">
   </form>

   <!-- Other HTML -->

</body>
```

You can include as many inputs as you need in the form, and these can be of different types (e.g. text, email, checkbox, etc.). However, each different piece of data you want to collect needs to have a unique `name` attribute.

To *render* the form in the view, add a method to the controller using the `@GetMapping` annotation:

```
1 @GetMapping("formTemplateName")
2 public String renderFormMethodName(Model model) {
3
4    // Method code...
5
6    return "pathToTemplate";
7 }
```

Some points to note:

1. Line 1: The string parameter for `GetMapping` must be the name of the form template you want to use.
2. Line 2: Declare a `Model` object to hold data that needs to be passed to the template.
3. The method code performs any data manipulation required before rendering the form. The `model.addAttribute` statements would be included here.
4. The `return` string specifies the path to the template. Recall that Spring automatically adds MOST of the file path—up through `.../templates`. You need to add any path details that follow.
   1. For example, if our `templates` folder contains a subfolder called `events` that holds a template called `create.html`, then line 6 would be `return "events/create";`.

11.6.4. Add a Form Handler Method

Now that you have created and rendered a form in your `coding-events` project, you need to add a method to the controller to *handle* its submission. Code along with the video below to add this functionality.

As usual, the following summary outlines the ideas from the clip.

11.6.4.1. Handle Form Submission

To *process* a form after the user clicks the *Submit* button, you need to add a method to the controller using the `@PostMapping` annotation:

```
  @PostMapping("formTemplateName")
1 public String processFormMethodName(@RequestParam Type parameter1, Type parameter2,
2 ...) {
3
4    // Method code...
5
6    return "redirect:templateName";
7 }
```

Some points to note:

1. Line 1: The string parameter for `PostMapping` must be the name of the form template.
2. Line 2: For each piece of data that needs to be retrieved from the form, declare a parameter of the appropriate type.

   Note

   `@RequestParam` matches the parameters to the submitted data. For this to work, the parameter names MUST match the `name` attributes used in each of the `input` elements.

3. The method code performs any data manipulation required after the information gets submitted.
4. Line 6: Generally, we want to send the user to a different page after they successfully submit a form. Instead of re-rendering the form, the `return` string *redirects* the user to a method that handles a different template.

11.6.5. Resources

1. Coding events starter code.

11.7. Template Fragments

Just like methods in Java provide us with the ability to reuse useful code, Thymeleaf allows us to do something similar with HTML.

Fragments are blocks of HTML elements that we want to use across multiple templates. The fragments are stored in a separate file, and they can be accessed by using the keywords fragment and replace.

A summary of these two keywords is given below, followed by another video walkthrough to give you some more live-coding practice.

11.7.1. Fragments DRY Your Code

Anythyme you find yourself typing identical code into different templates, you should consider ways to streamline your work.

Never fear, coder. Fragments are the tool you need. The general syntax is:

```
th:fragment = "fragmentName"
th:replace = "fileName :: fragmentName"
```

11.7.1.1. th:fragment

Let's assume that you want to add the same styled header and a set of links to multiple pages within your web project.

Examples

The common header styled with CSS:

```
<h1 class="fancyTitle" th:text="${title}></h1>
```

The link list, which appears below a dividing line:

```
<hr>
<a href = "https://www.launchcode.org">LaunchCode</a> <br/>
<a href = "https://www.lego.com">Play Well</a> <br/>
<a href = "https://www.webelements.com">Other Building Blocks</a>
```

Instead of pasting this code into every template, we will store the HTML in a separate file.

1. Create a new html file inside the templates folder and give it a clear name (e.g. fragments.html is a common practice).
2. Inside this file, use the th:fragment attribute on each block of code that you want to reuse. Note that you must provide a different name for each sample.

   Examples

For the h1 element:

```
<h1 th:fragment = "styledHeader" class="fancyTitle" th:text="${title}></h1>
```

For multiple elements, we need to wrap them in another tag:

```
<div th:fragment = "linkList">
   <hr>
   <a href = "https://www.launchcode.org">LaunchCode</a> <br/>
   <a href = "https://www.lego.com">Play Well</a> <br/>
   <a href = "https://www.webelements.com">Other Building Blocks</a>
</div>
```

We can now pull either of the fragments—styledHeader or linkList–into any template in our project.

Tip

What if we do not want to keep the link list inside its own div element? One option is to use th:block:

```
<th:block th:fragment = "linkList">
   <hr>
   <a href = "https://www.launchcode.org">LaunchCode</a> <br/>
   <a href = "https://www.lego.com">Play Well</a> <br/>
   <a href = "https://www.webelements.com">Other Building Blocks</a>
</th:block>
```

Another option is to use the attribute th:remove, which allows us to selectively discard the wrapper tag, but not any of its children.

```
<div th:fragment = "linkList" th:remove = "tag">
```

For a more detailed discussion of the different th:remove options, consult the Thymeleaf documentation.

11.7.1.2. th:replace

This attribute does just what the name implies—it *replaces* the tag that contains it with the selected fragment. Thus, if the fragment is a <p> element, and the template contains <div th:replace = "...">, then the div in the template will be replaced with a p. Similarly, if the fragment contains multiple elements, the single template tag will be replaced with the entire code block.

Take home lesson: The template tag that contains th:replace does NOT have to match the HTML tags in the fragment.

Now let’s see how to pull fragments into a template:

Examples

```
 1 <!DOCTYPE html>
 2 <html lang="en" xmlns:th="http://www.thymeleaf.org/">
 3 <head th:fragment="head">
 4    <meta charset="UTF-8"/>
 5    <title th:text="${pageTitle}"></title>
 6 </head>
 7 <body>
 8
 9    <h1 th:replace = "fragments :: styledHeader"></h1>
10
11    <!-- Specific template code here... -->
12
13    <p th:replace = "fragments :: linkList"></p>
14
15 </body>
```

When the code runs, the h1 element in line 9 will be replaced by the styledHeader fragment stored in the fragments.html file. Also, the p element in line 13 will be replaced by the <hr> and three <a> elements defined in the linkList fragment.

11.7.2. Try It!

Code along with the following video to practice using fragments in your templates:

Remember that the summary text for the fragment and replace keywords supports the video and is NOT intended as a replacement.

11.7.3. Check Your Understanding

Question

Given our code fragment in fragments.html:

```
<th:block th:fragment = "linkList">
   <hr>
   <a href = "https://www.launchcode.org">LaunchCode</a> <br/>
   <a href = "https://www.lego.com">Play Well</a> <br/>
   <a href = "https://www.webelements.com">Other Building Blocks</a>
</th:block>
```

Which of the following would place the linkList fragment inside a <div> element in the template?

1. <div th:replace = "fragments :: linkList"></div>
2. <div>${th:replace = "fragments :: linkList"}</div>
3. <div><p th:replace = "fragments :: linkList"></p></div>
4. <p><div th:replace = "fragments :: linkList"></div></p>

Bonus Question

Research th:remove to answer this question. Which of the following does NOT remove the wrapper tag but does eliminate all of its children.

1. th:remove = "all"
2. th:remove = "body"
3. th:remove = "tag"
4. th:remove = "all-but-first"
5. th:remove = "none"

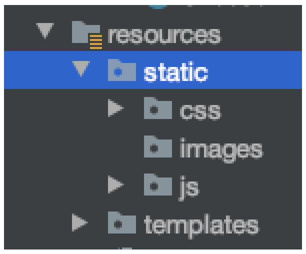
11.8. Static Resources

Up to now, we used templates to display controller data as text in the view. If we need to display an image or video, or if we want to create a link to a different file, then we need to move beyond a text output.

With Thymeleaf, we can set values for the HTML `src` and `href` attributes. Instead of hard-coding a file path or external URL inside a tag, `th:src` and `th:href` take advantage of a simpler syntax. You did this near the end of the last video when you referenced information stored in files other than the controller or the template.

11.8.1. Accessing Resources

Inside the `resources` folder, there is another directory called `static`. By convention, this is the location where we store files that our project needs to access—like images, CSS stylesheets, JavaScript code, etc.



11.8.1.1. `th:src`

To access a file with the standard HTML `src` attribute, we need to provide either a detailed or relative file path in order to establish a link. `th:src` shortens this process by filling in most of the file path automatically (`.../project-name/src/main/resources/static`). All we need to do is fill in the last portion of the file path—everything after `/static`.

The general syntax is:

```
th:src = "@{/filePath}"
```

Examples

If we have an image file called `familyPhoto1.jpg` stored in `static`, then we can display it in the view with an `img` tag as follows:

```
<img th:src = "@{/familyPhoto1.jpg}" />
```

If the image is contained in an `images` folder inside `static`, then the syntax is:

```
<img th:src = "@{/images/familyPhoto1.jpg}" />
```

Think of the `@` symbol as representing everything in the file path up to the `static` folder.

11.8.1.2. `th:href`

The syntax for `th:href` is the same as that for `th:src`. For example, to link to a CSS stylesheet:

```
<link rel = "stylesheet" th:href="@{/css/styles.css}" />
```

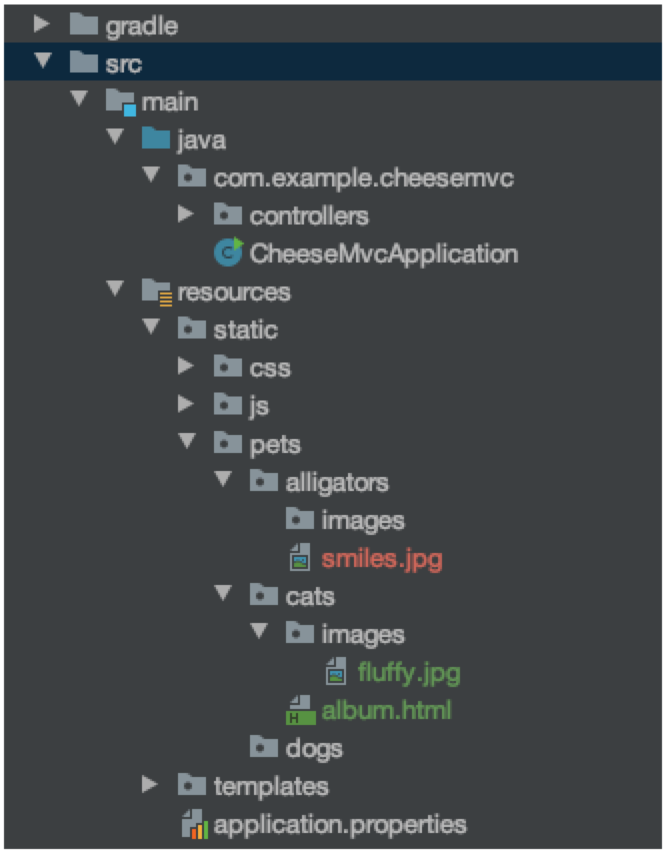
11.8.2. Try It!

Code along with this final video to practice adding static resources to your `coding-events` project:

11.8.3. Check Your Understanding

Question

Given the file tree shown below, which option displays the correct syntax for finding the image `fluffy.jpg`?



```
▶ gradle
▼ src
  ▼ main
    ▼ java
      ▼ com.example.cheesemvc
        ▶ controllers
          CheeseMvcApplication
    ▼ resources
      ▼ static
        ▶ css
        ▶ js
        ▼ pets
          ▼ alligators
              images
              smiles.jpg
          ▼ cats
            ▼ images
                fluffy.jpg
              album.html
            dogs
      ▶ templates
        application.properties
```

1. `th:src = "@{/pets/cats/fluffy.jpg}"`
2. `th:src = "@{/static/pets/cats/fluffy.jpg}"`
3. `th:src = "@{/static/pets/cats/images/fluffy.jpg}"`
4. `th:src = "@{/pets/cats/images/fluffy.jpg}"`

11.9. Exercises: Thymeleaf

In the chapter, we started working on an application for tracking various coding events around town.

Open up your `coding-events` project in IntelliJ.

11.9.1. Getting Started

Checkout a new branch off of `master` called `my-exercises-solution`.

Now for some Git magic! We are going to go back in time to when our templates were still using data from a static event list.

1. Use the `git log` command. Go through the logs until you find the commit where you finished creating a static event list.

   Note

   If you were not making commits after each video, that is fine. You can fork and clone this repo to get started.

2. Use `git reset --hard <commit>` to go back to that moment.

Warning

Before you reset to an older commit, make sure you are on a separate branch! This will reset your repo to a previous state!

Now, let’s add descriptions to our events!

11.9.2. Expanding our Events Schedule

1. Comment out your previous code in the `displayAllEvents` method.
2. In the videos, we learned how to use templates to display the elements in a static list called `events`. Let’s make our `events` list a HashMap! This enables us to add descriptions to our events.
3. Fill your `events` HashMap with the names and descriptions of 3 coding events around town.
4. Using `th:block` and `th:each`, put together the events and their descriptions in a table as opposed to an unordered list.
5. Use fragments to store the address of the new tech hub where all of the programmers are hanging out. Use `th:replace` in your main template to bring in the address as a third column in your table. You may need to create a new `fragments.html`.
6. Add some CSS to style your table to make it easier to read and center it on the page. You may need to create a new `styles.css` as well. Make sure to connect `styles.css` to the appropriate template with `th:href`.

11.9.3. Bonus Mission

Try to add one more column to the table with pictures for each coding event. You need to use `th:href` to pull in pictures in the appropriate template.

11.10. Studio: Spa Day!

After all of the hard work we have put into learning about Thymeleaf, it is time for a spa day! First, we need to put our knowledge of Thymeleaf to the test. Instead of heading out to our favorite spa, let's make an application for the owners!

Our application needs to do the following:

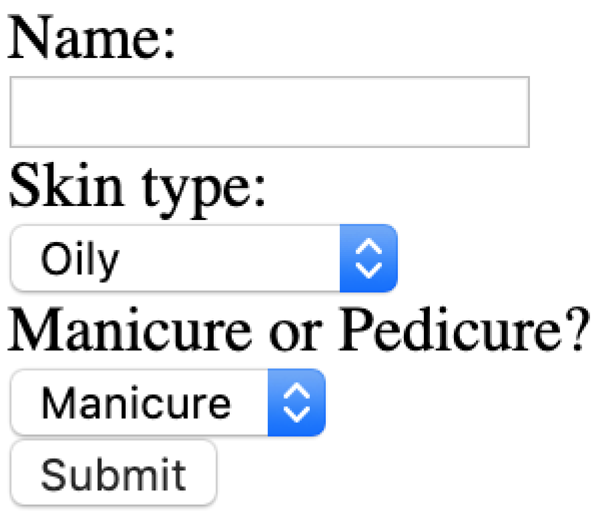
1. Display the user's name and skin type under their customer profile.
2. Display the appropriate facial treatments for their skin type.
3. Display the description of the spa's manicures or pedicures depending on the user's interest.

As always, read through the whole page before starting to code.

11.10.1. Setup

Fork and clone the appropriate repository. Check out the project via Version Control in IntelliJ.

When you open the project and run it using `bootRun`, go to `localhost:8080` and you should see a form!


Name:

Skin type:

Oily

Manicure or Pedicure?

Manicure

Submit

11.10.2. The Customer Profile

In `controllers`, we have one controller called `SpaDayController`. Inside `SpaDayController`, we have three methods.

1. The `checkSkinType()` method. The owners gave us this method to help us figure out which facial treatments are appropriate for which skin type.
2. The `customerForm()` method, which looks very similar to what we did in the last lesson.
3. The `spaMenu()` method, which we will use to bring in our Thymeleaf template, `menu.html`.

In `templates`, we have a Thymeleaf template called `menu.html`. Inside `menu.html`, there are two main divs in the body. Let's focus on the div with the id, `clientProfile`.

1. Add an `h2` that says “My profile”.
2. Add a `p` tag and use `th:text` to bring in the value of `name`.
3. Add a `p` tag and use `th:text` to bring in the value of `skintype`.
4. Add a `p` tag and use `th:text` to bring in the value of `manipedi`.

Run the application and head to `localhost:8080` to see the results! When we fill out the form, we should see a new page with the client profile at the top!

11.10.3. List All Appropriate Facial Treatments

Luckily for us, the spa owners gave us the `checkSkinType()` method in our `SpaDayController`. Also, our teammates already set up code in our `spaMenu()` method to fill an `ArrayList<String>` with facial treatments that would work for the user’s skin type. Now, we just need to use Thymeleaf to display the appropriate facial treatments (stored in the ArrayList, `appropriateFacials`)!

Let’s head back to `menu.html` and checkout the empty div with the id, `servicesMenu`.

Add a table and iteratively (using our `th:block` and `th:each` combo) add rows for the values in `appropriateFacials`. If you need a quick reminder of the syntax, review the th:block section .

11.10.4. Mani or Pedi?

One other thing the spa owners want to be cautious of is their treatment descriptions. Because the descriptions rarely change and are going to be used in multiple places on the site, the owners have written up the descriptions as Thymeleaf fragments.

Checkout the file, `fragments.html`, in the `templates` directory. The owners have already written up the descriptions for their manicure and pedicure in separate `p` tags.

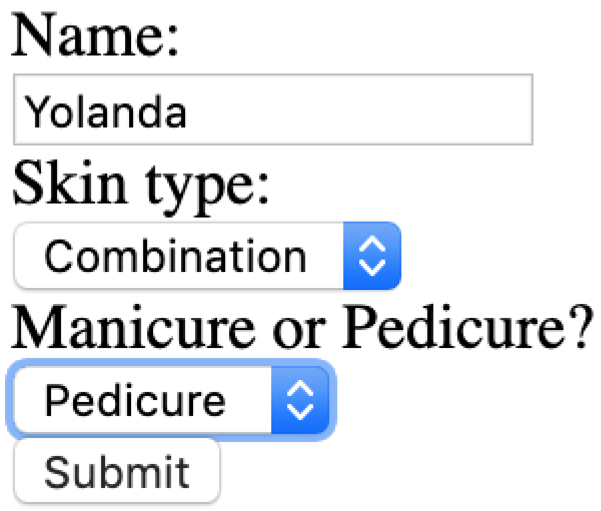
We want to put the description in a `div` along with an `h3` stating that it is either a manicure or pedicure. This new `div` should be inside the `servicesMenu` div.

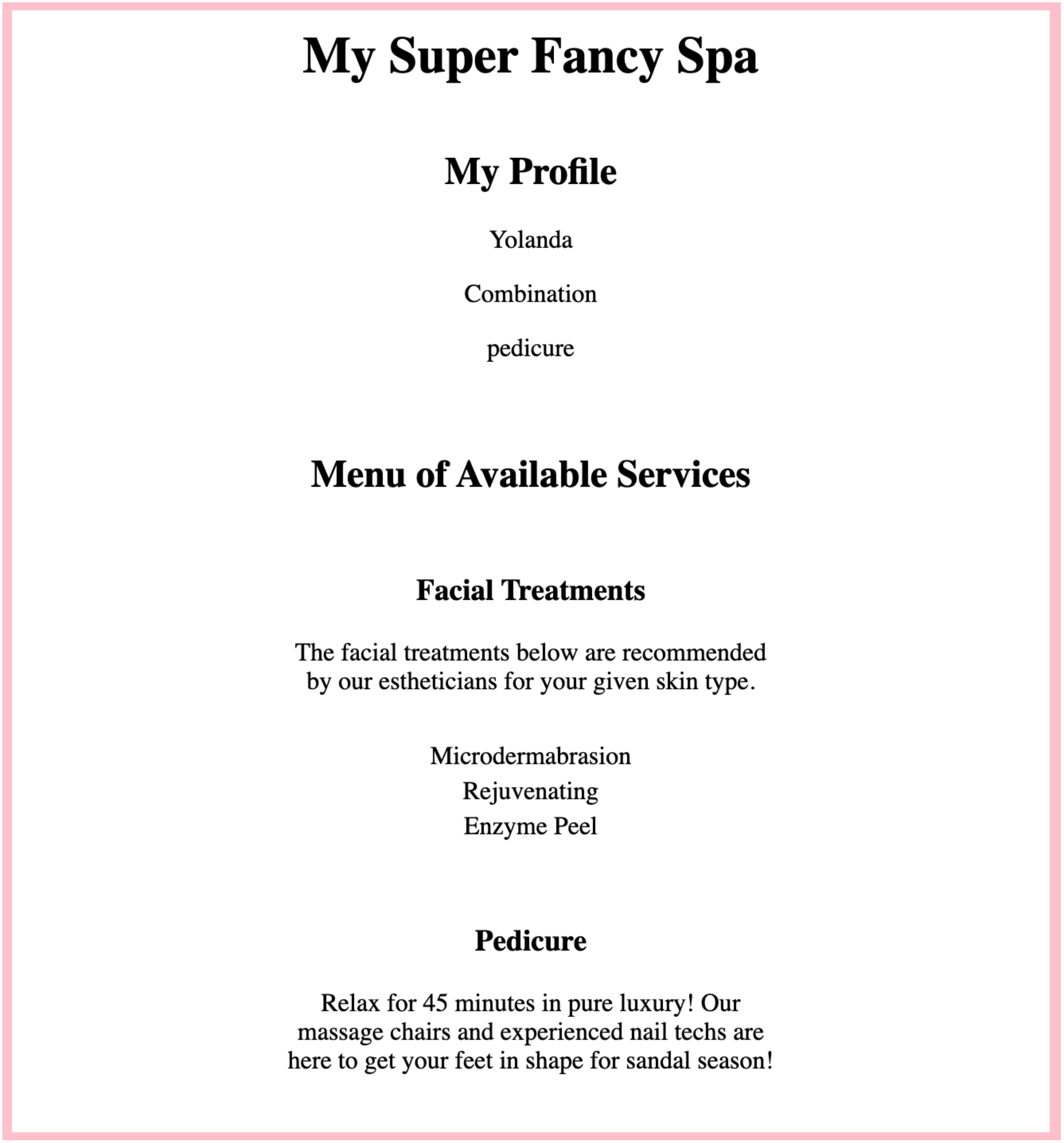
Use `th:if` to determine if the value of `manipedi` is a `"manicure"` or `"pedicure"`.

1. If the value of `manipedi` is `"manicure"`, the `div` element containing the `h3` that says `"Manicure"` needs to be on the page and the `p` tag needs to be *replaced* with the fragment of the manicure description from `fragments.html`.
2. If the value of `manipedi` is `"pedicure"`, the `div` element containing the `h3` that says `"Pedicure"` needs to be on the page and the `p` tag needs to be *replaced* with the fragment of the pedicure description from `fragments.html`.

11.10.5. End Result

After you are done with the studio, you should be able to fill out the form, click “Submit”, and see your profile page.


Name:
Yolanda
Skin type:
Combination
Manicure or Pedicure?
Pedicure



My Super Fancy Spa

My Profile

Yolanda

Combination

pedicure

Menu of Available Services

Facial Treatments

The facial treatments below are recommended by our estheticians for your given skin type.

Microdermabrasion
Rejuvenating
Enzyme Peel

Pedicure

Relax for 45 minutes in pure luxury! Our massage chairs and experienced nail techs are here to get your feet in shape for sandal season!

11.10.6. Bonus Mission

1. Modify `styles.css` to get some CSS practice! Try add a footer with square shaped `div` elements. Each square should be a different color for different available nail polishes.
2. Modify the form to allow the user to select either a manicure or pedicure or *both*. If the user selects both, display both the manicure and pedicure descriptions on the `Spa Menu` page.
3. Work with routes and paths to display the spa menu page on a separate route from the form.