Using View Templates in a Spring Boot web application (Part 2)

Overview

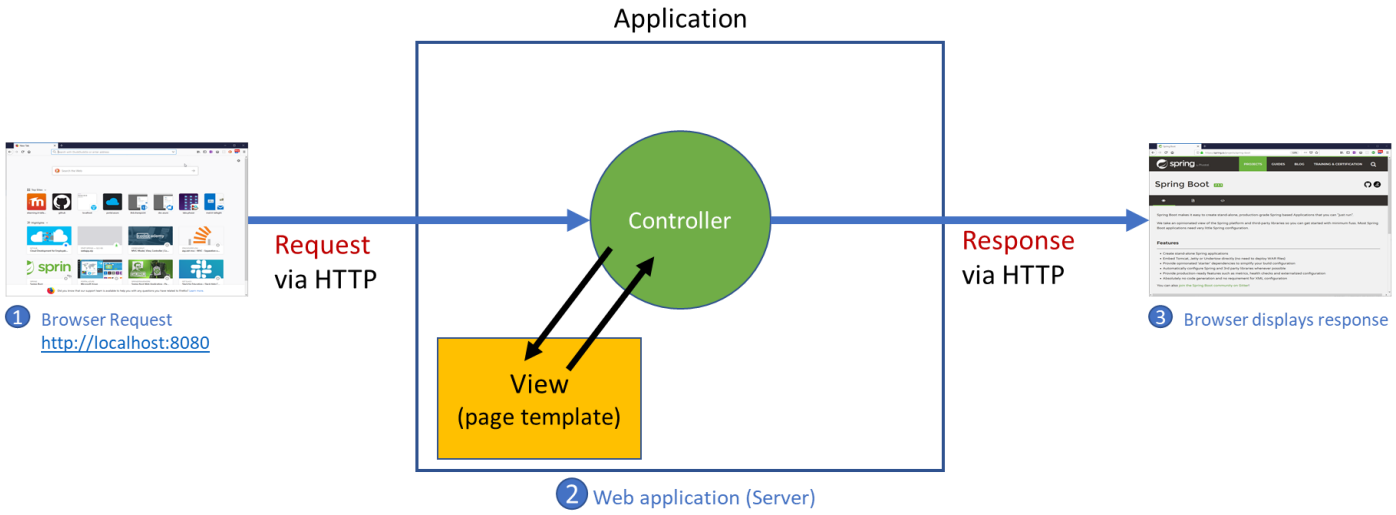
At the end of part 1 you should have had a working Spring Boot application which generated a message string in a controller method and returned it, as part of an HTTP response, to the browser.

While this works fine for sending a small amount of to the browser, it is not particularly scalable. If a complex web page was to be displayed, all the content, layout, etc. would need to be coded in the controller method. That would make page design complex and make the web application difficult to maintain. Imagine how a team made up of Java developers and web page designers might approach this task.

As a solution, the **MVC** (Model – View – Controller) **design pattern** requires separation of concerns in the application:

- **Models** – Represents the application data structure
  - For example Java classes representing data objects and the operations (methods) which can be performed on that data.
- **Views** – deal with presentation, i.e. the user interface and data formatting.
  - Page templates in a web application.
- **Controllers** - control application flow and makes decisions about data.
  - In a web application deal with requests and generate an appropriate response.
  - A controller can make use of Views and Models to generate output.

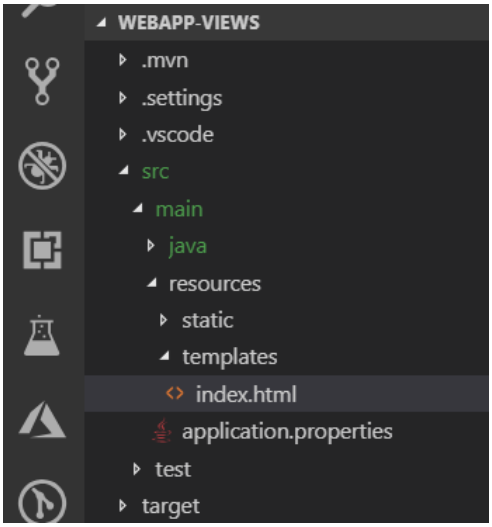
This section will focus on **Views**, we will define **view templates** for pages in the application and use the **controller** to fill these templates to generate an output.



This tutorial continues from where Part 1 ended. Start with a copy of your completed part 1. We will be using the **Thymeleaf** templating engine, https://www.thymeleaf.org/, for views (you may remember choosing the add on when creating the application in part 1). There are many alternatives but Tymeleaf's advantage is that is based on standard HTML 5 which can be easily previewed in any browser during development.

1. Adding a view template

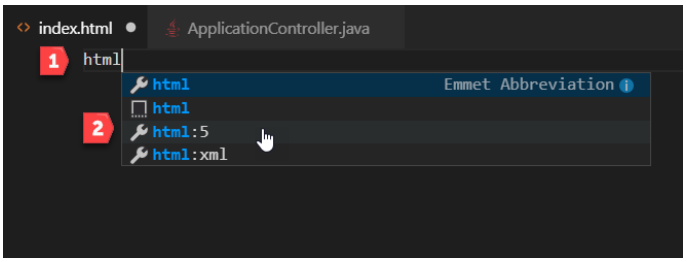
Find the templates folder in your application (under **src/main/resources**)



Then add a new file and name it **index.html**. This will be the view template associated with the index() method in **ApplicationController.java**, the hello world home page.

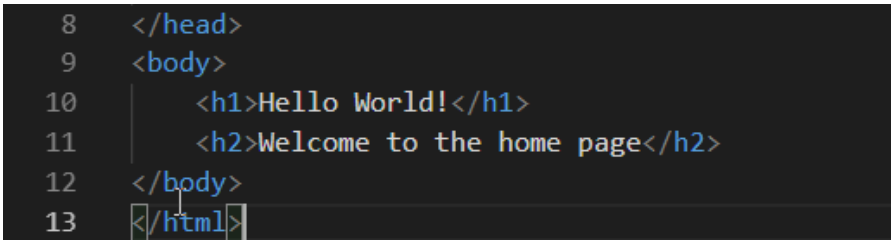
Open **index.html** and add some content. VS Code includes Emmet abbreviations which provide timesaving html and css shortcuts (see https://www.sitepoint.com/faster-workflow-mastering-emmet-part-1/ for a demo).

To create add a standard HTML 5 structure in the new view, type **html**. A pop-up with options will appear, use the mouse or tab key to choose the **html:5** option.



A skeleton HTML page will be created

Using the same method, type **h** to generate **<h1>** and **<h2>** elements in the page body. Then add text as follows:



Save the page when finished – you have now created a view template for the home page.

2. Using the View

To use the view, the **index()** method in **ApplicationController.java** must be modified. Remember this method is called when no page or the root directory of the site ("**/**") is requested. In other words, it loads the default page.

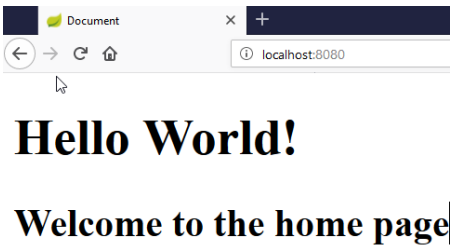
First remove the **@ResponseBody** line, just before the **index()** method declaration, as that causes the controller to send a direct response without first loading a view template.

Then modify the **return** statement in the method, replacing the message with the name of the view template which should be used - "**index**" in this example.

```
12  // The @ annotation identifies this as a Controller class
13  @Controller
14  public class ApplicationController {
15
16      // This method, index(), serves as the site index - the default page
17      // Requests for the root '/' will be handled by this method
18      @RequestMapping(value = "/", method = RequestMethod.GET)
19      public String index() {
20          // Load and return the index view
21          return "index";
22      }
23  }
```

Save the controller and then run the application using by entering **mvn spring-boot:run** in the VS Code terminal.

Open the page in a browser, you should see:



3. Adding some Dynamic Content

At this point the application does nothing that couldn't be achieved using static HTML. The benefit of using dynamic web applications is the ability to handle dynamic content, read from user input, database queries, etc.

This section will read a parameter value (**name**) from the URL and display it in the page. Instead of Hello World! You will greet the name value, e.g. Hello Bob!

The request URL will look something like this http://localhost:8080/?name=Bob note the **?** indicates the start of the **Querystring** which contains parameter name=value pairs (separated by & if more than one). This example has a single parameter and value, **name=Bob**

3.1 Read the parameter in the controller

The **index()** controller method must be modified so that it can accept a parameter.

1. The following imports are required, if not already present, to give access to the required Spring Framework classes. The first, **org.springframework.ui.Model**, is required to pass data to the view. The second enables parameters to be read.

```
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestParam;
```

2. Add the parameters required to the **index()** method. The **name** value will be accepted from the HTTP request. The **model** object will be passed to the method and then passed to the view template.

```
// index accepts a request parameter, named name,  from the address URL
// the parameter is optional and has a default value if not provided
// String name will be assigned the param value
// Model model is used to pass data to the view
public String index(@RequestParam(name = "name", required = false, defaultValue = "") String name, Model model) {
```

3. First modify the controller again, this time add **name** and its value to the view **model** object.

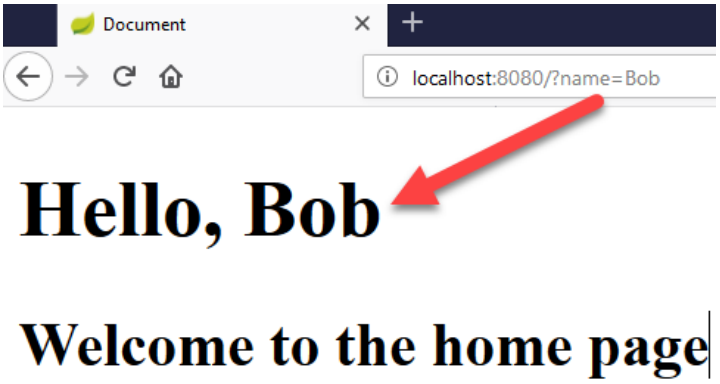
```
24    public String index(@RequestParam(name = "name", required = false, defaultValue = "") String name, Model model) {
25
26        // add name and its value to the view model object
27        model.addAttribute("name", name);
28
29        //Load and return the index view
30        return "index";
31    }
32 }
```

Then modify the **index.html** view template so that it displays the greeting "Hello <name>" where name is is the name parameter value. The thymeleaf template command **th:text= "'Hello, '+ ${name}"** inserts the message. Note that the **name** variable passed from the controller, via **model**, is expressed as **${name}**

index.html × ApplicationController.java

```
1   <!DOCTYPE html>
2   <html lang="en">
3   <head>
4       <meta charset="UTF-8">
5       <meta name="viewport" content="width=device-width, initial-scale=1.0">
6       <meta http-equiv="X-UA-Compatible" content="ie=edge">
7       <title>Document</title>
8   </head>
9   <body>
10      <h1 th:text="'Hello, ' + ${name}"></h1>
11      <h2>Welcome to the home page</h2>
12  </body>
13  </html>
```

4. Save everything and test in your browser, the result should look like this:



4. Add more pages and navigation

4.1 Add the About page, with its own view template.

Start by adding a new method **`about()`** to the controller. Note **`@RequestMapping`** which defines **`/about`** as the route to this method.

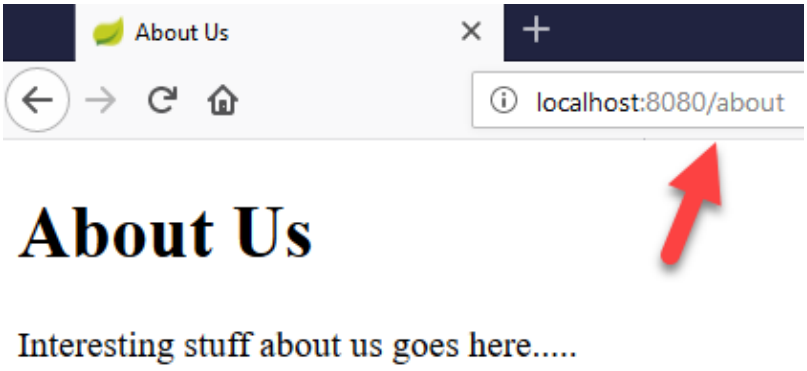
```
// The about page will be accessed using /about from the browser
@RequestMapping(value = "/about", method = RequestMethod.GET)
public String about() {
    //Load and return the about view
    return "about";
}
```

Next add the view template, **`about.html`**, something like this (don't forget to use emmet abbreviations):

about.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>About Us</title>
</head>
<body>
    <h1>About Us</h1>
    <p>Interesting stuff about us goes here.....</p>
</body>
</html>
```

Test the page



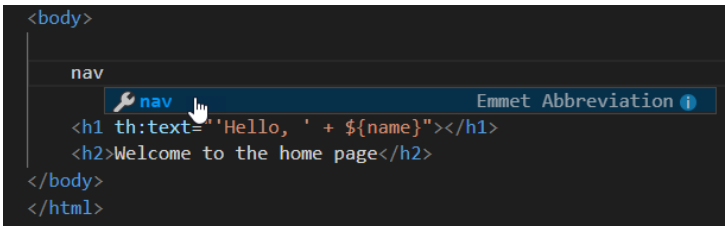
About Us

Interesting stuff about us goes here.....

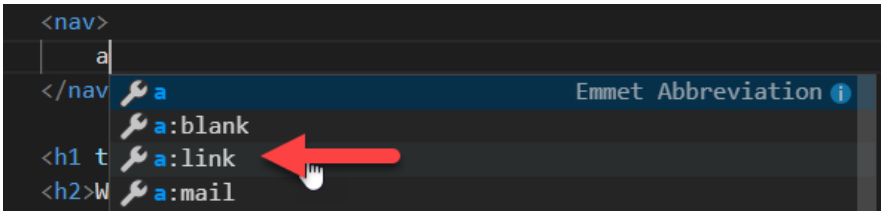
4.2 Navigation

Navigation links will allow easy access to all (two) pages in the site.

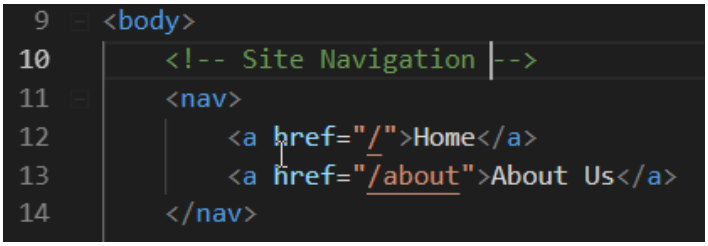
Open **index.html** and at the top of the body section, type **nav**. Use **Emmet** to complete the element



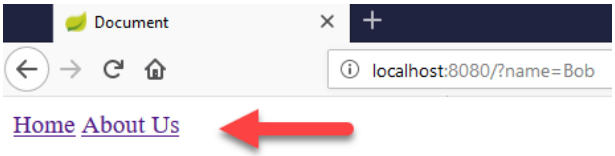
Then add 2 **anchor** (hyperlink) elements to the new **<nav>** element, again use **Emmet**:



Fill in the elements to create the links to each page



Save **index.html** and reload the application in your browser, you should see two links at the top



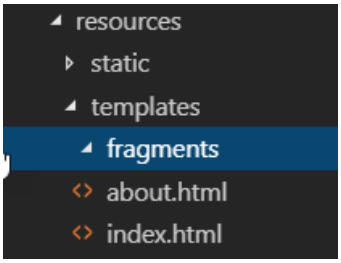
Hello, Bob

Welcome to the home page

Clicking About Us will load the About page, but that page will not have the links. There are two options

1. Replicate the navigation links on every page.
    a. Easy on a simple site but difficult to maintain across many pages.
2. Share the navigation menu html across all pages.
    a. The advantage of this method is that, in case of changes, the navigation menu will require just one edit.

*4.2.1. To use option 2, create a new folder, named **fragments**, inside the templates folder*

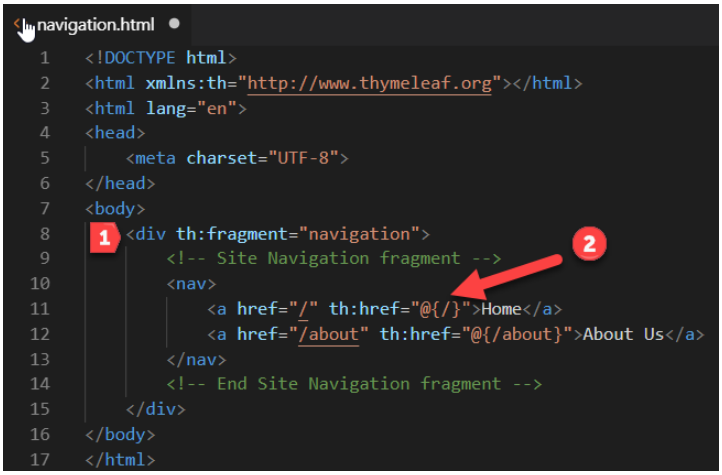


Next add a file, named **navigation.html** to the **fragments** folder and add the following content:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org"></html>
<html lang="en">
<head>
    <meta charset="UTF-8">
</head>
<body>
    <div th:fragment="navigation">
        <!-- Site Navigation fragment -->
        <nav>
            <a href="/" th:href="@{/}">Home</a>
            <a href="/about" th:href="@{/about}">About Us</a>
        </nav>
        <!-- End Site Navigation fragment -->
    </div>
</body>
</html>
```

This looks like a normal html page, but note the highlighted content:

1. The element is defined as a `th:fragment=navigation`
    a. That allows the enclosed html to be inserted into other pages
2. The links are defined using `th:href` so that the links can be validated when this fragment is inserted



One the fragment is created and saved it can be inserted into other pages such as **about.html**. Note the green `th:include` element below which inserts the fragment before the page is sent to the browser.

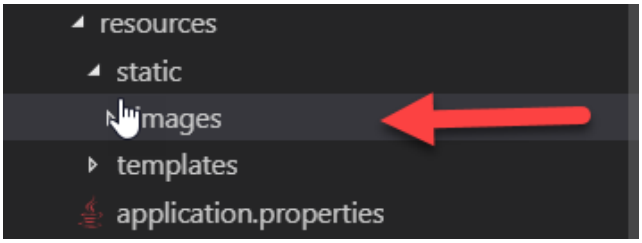
```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>About Us</title>
</head>
<body>
    <!--/*/ <th:block th:include="fragments/navigation :: navigation"></th:block> /*/-->
    <h1>About Us</h1>
    <p>Interesting stuff about us goes here.....</p>
</body>
</html
```

The same `th:include` element can be used anywhere the navigation fragment is required. Try it in **index.html**

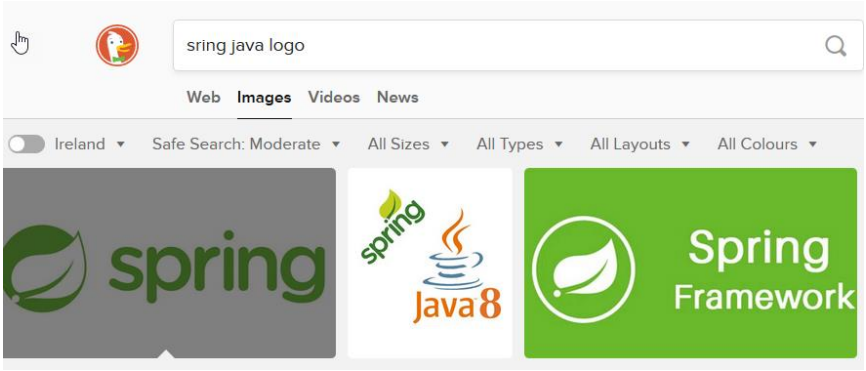
5. Adding images (and other 'static' content)

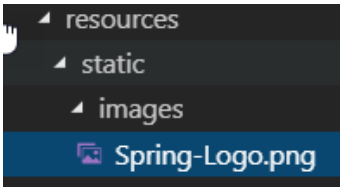
Content which is not generated programatically (i.e. in Java code) is refered to as static. This might include images, css files, douments, or any other content not changed by the application code.

Static conent should be placed in the **`resources/static`**. Add a new sub folder named **`images`** to static:



Now find a jpeg or png image, for display on the home page, and save it to the images folder.For example:





5.1 Display the image

Open the index view template and type **`img`** at the location where the image should appear (use Emmet to complete the element). Add the image source path, relative to the location of the template **`../`** goes up a directory. Setting width is optional but it may be useful if the image is large.

```
    <img src="../images/Spring-Logo.png" width="400"/>

</body>
```

The above will work but is also a good idea to include a th: version of the path – especially when placing images in a fragment.

```
<img src="../images/Spring-Logo.png" width="400" th:src="@{/images/Spring-Logo.png}" />
```

The result: