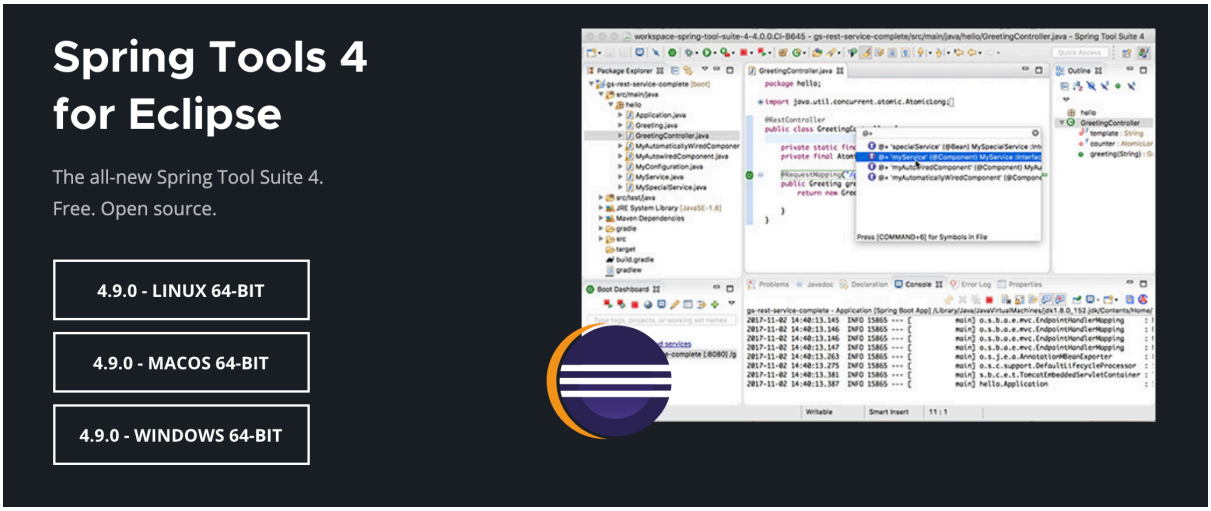
Practical 03

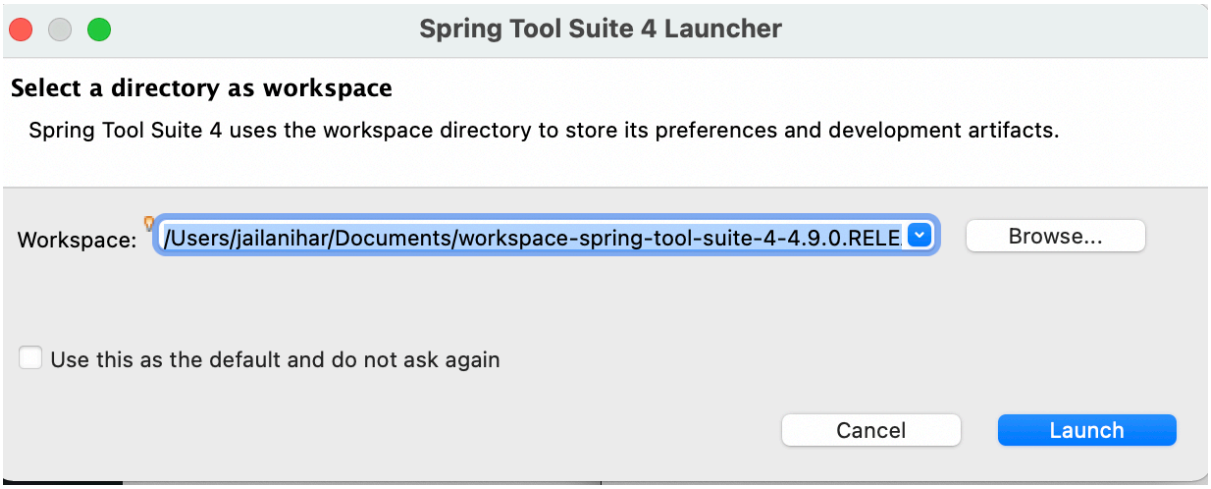
Project 01 - Spring Web Application

Lesson 1.1 - Preparing Spring Web Application Development Environment

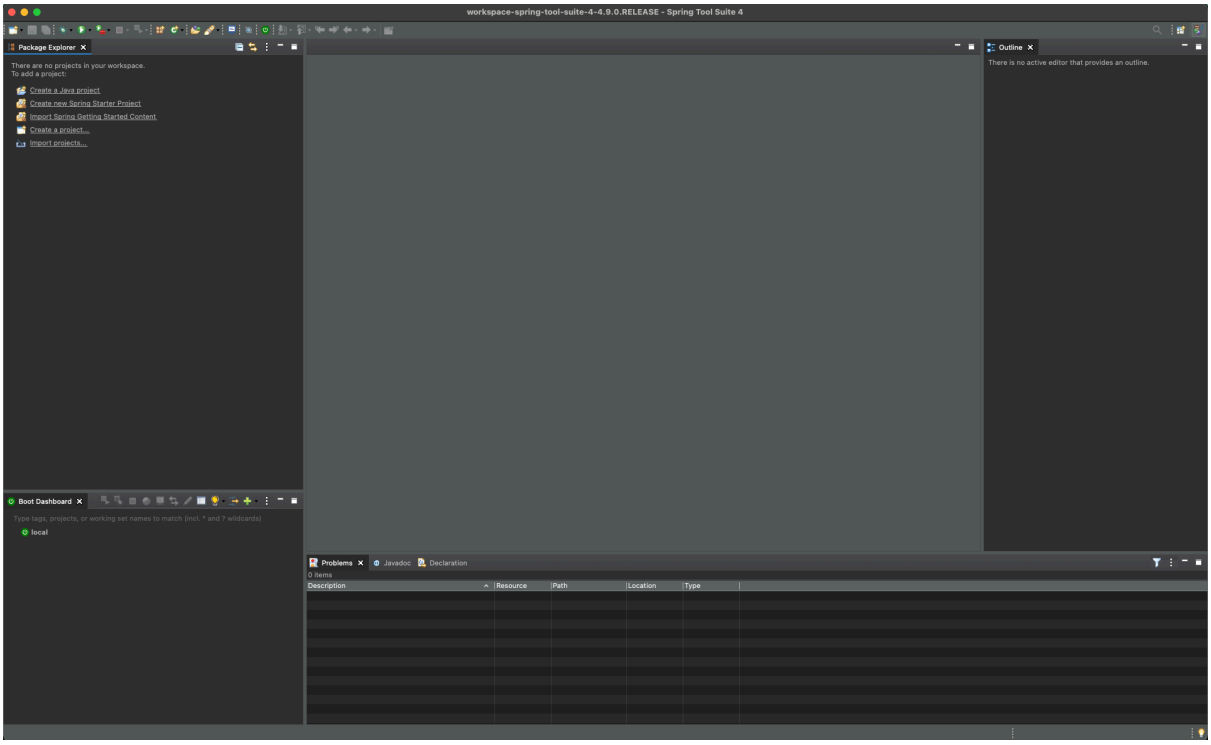
1. Since most are familiar with Eclipse Integrated Development Environment (IDE). To streamline your Spring Web Application development. It is recommended to download and install **Spring Tools for Eclipse** from https://spring.io/tools



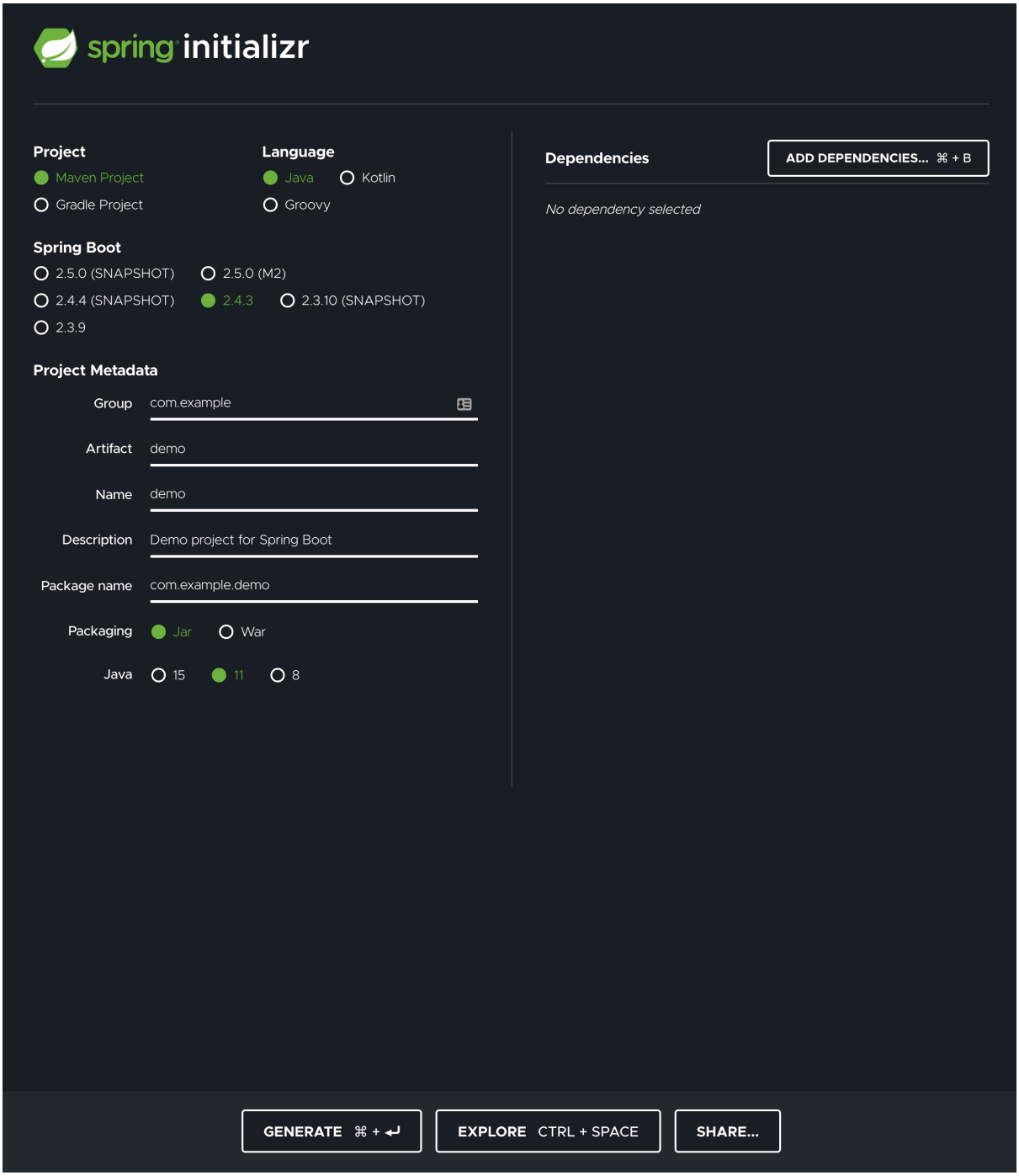
2. Once installed, run the spring tools suite application and you will be asked to choose workspace location. If the folder does not exists, spring tools suite will create the workspace folder for you. All of your projects should be inside this workspace.



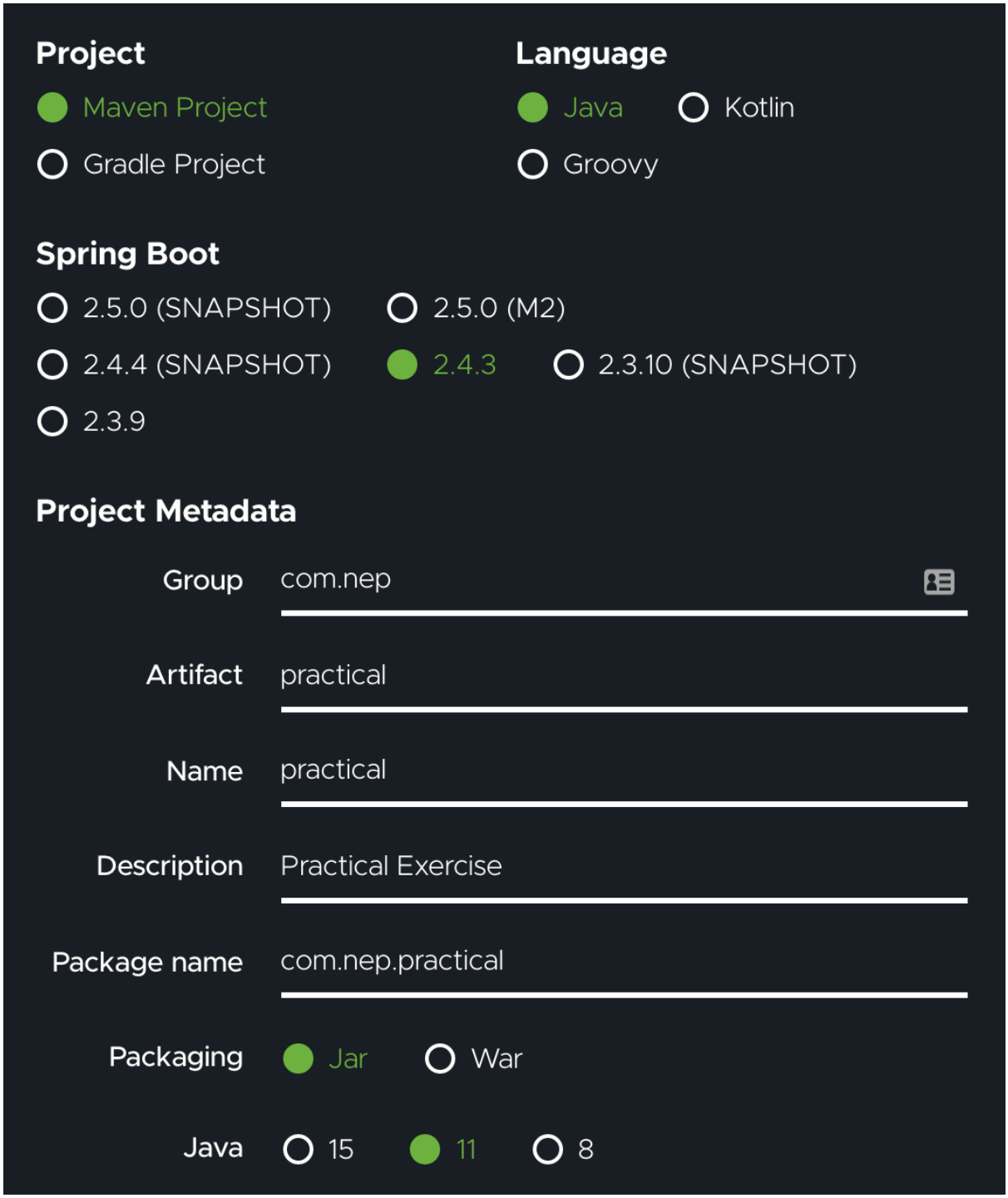
3. Once launched, you'll notice that the layout is the same as eclipse. Actually it is eclipse IDE with additional Spring Framework addons that will help you develop Spring Web Application.



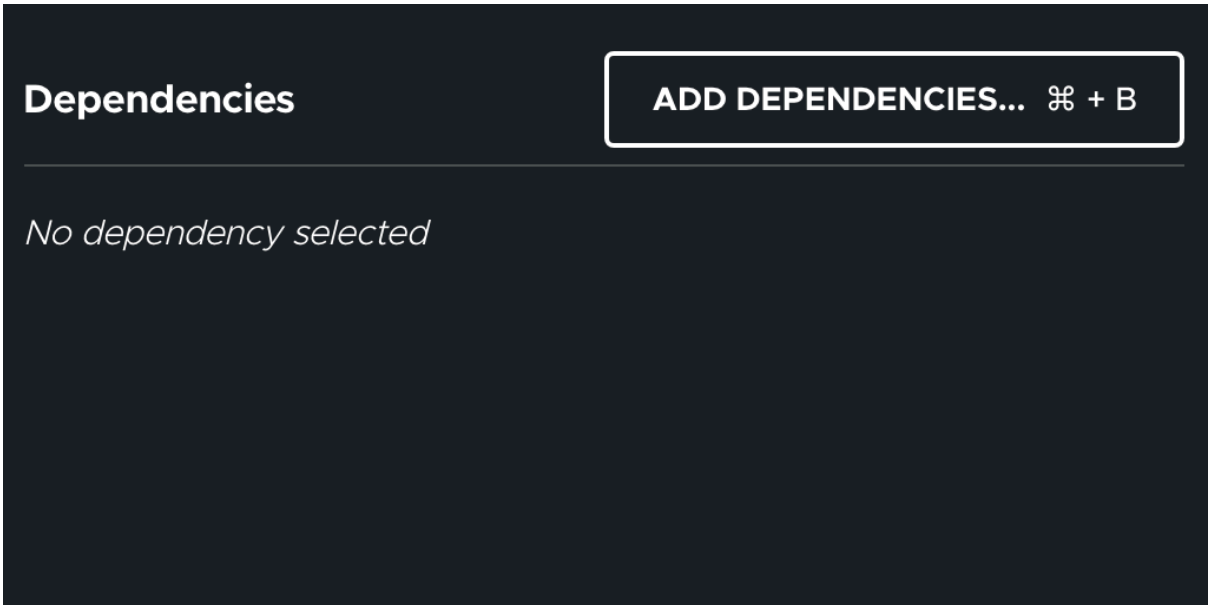
4. Next, instead of creating and configuring your own spring project. It would be better to generate the spring project first then we can start developing our spring web application without worrying about configuration from scratch. To generate, go to https://start.spring.io/



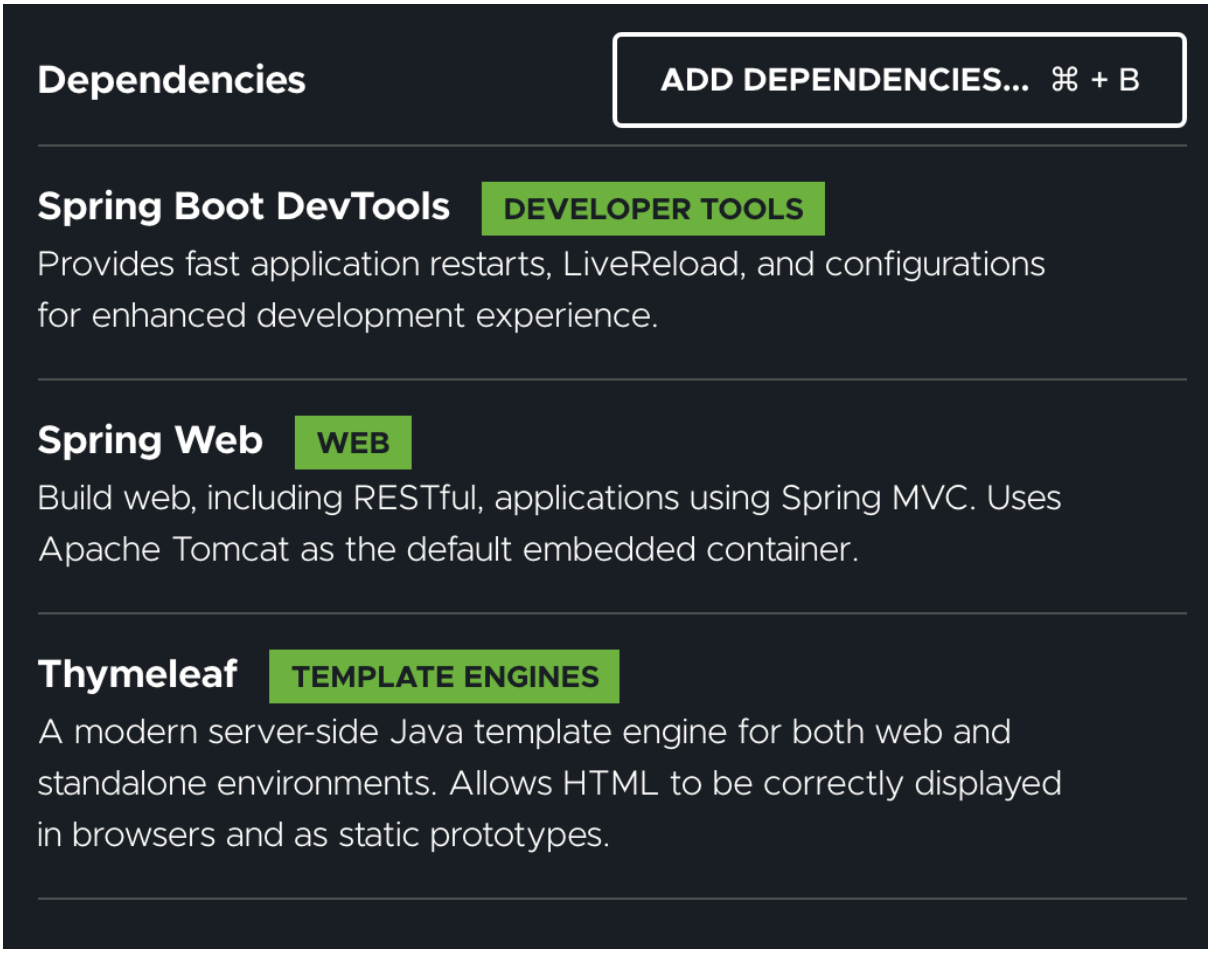
5. Depending on your Java installation, you choose which one is appropriate. Java 15, 11 or 8. Make sure you follow the other settings. (Note: Spring Boot version might be different. Just leave it by default.)



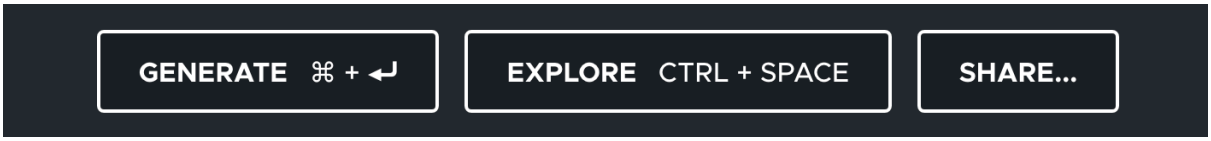
6. Lets add the required libraries needed for our Spring Web Application. Click the Add Dependencies...



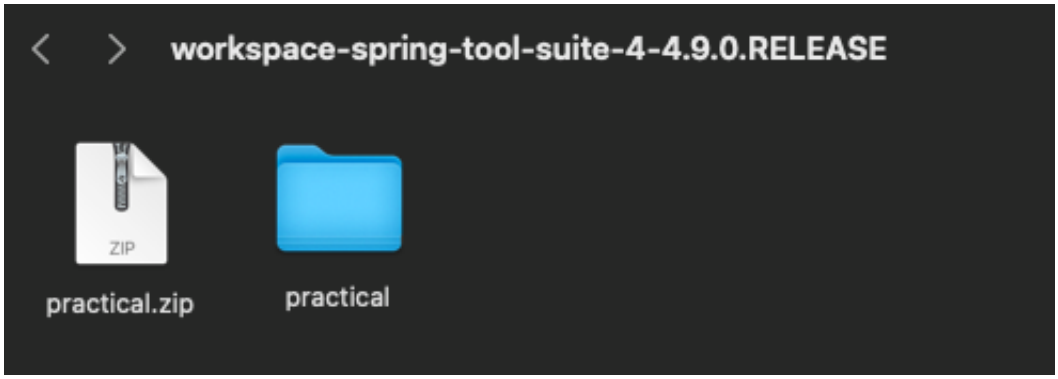
7. We need Spring Boot DevTools, Spring Web and Thymeleaf. You may read the description to understand what the purpose of each dependency.



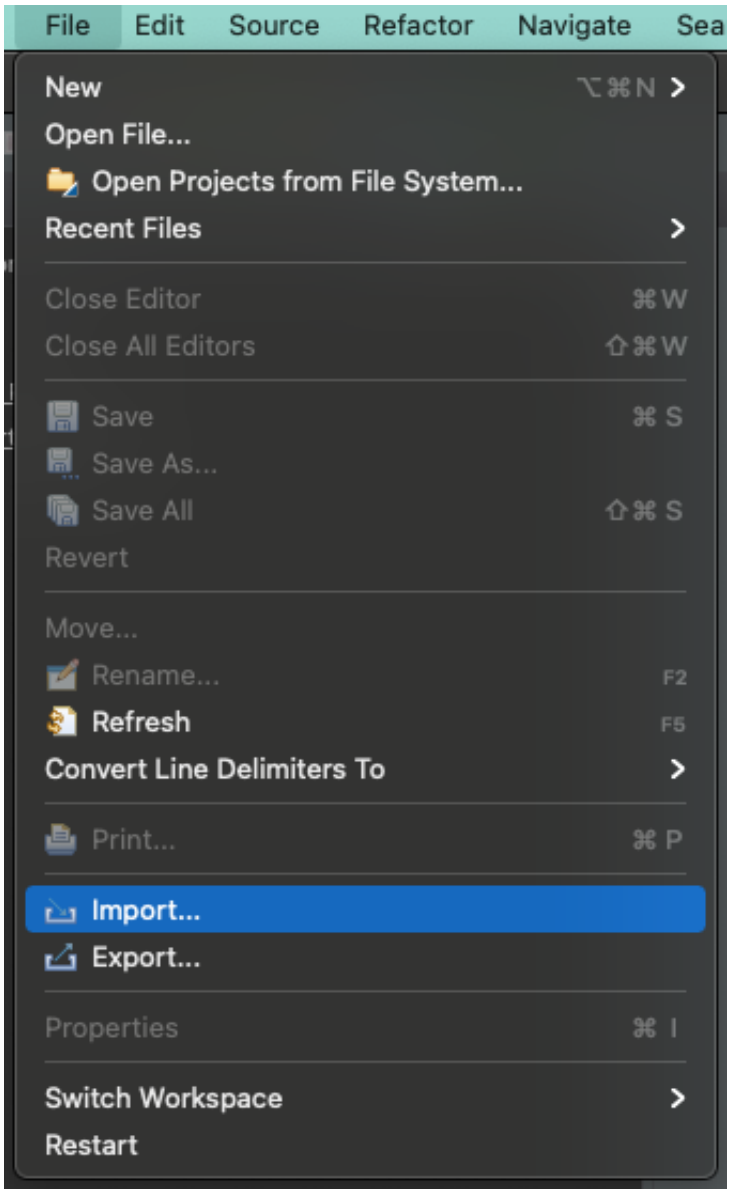
8. Once you're done, click Generate. It should download the project folder. Depending on your Artifact name, it should be the same. In this case, practical.zip.



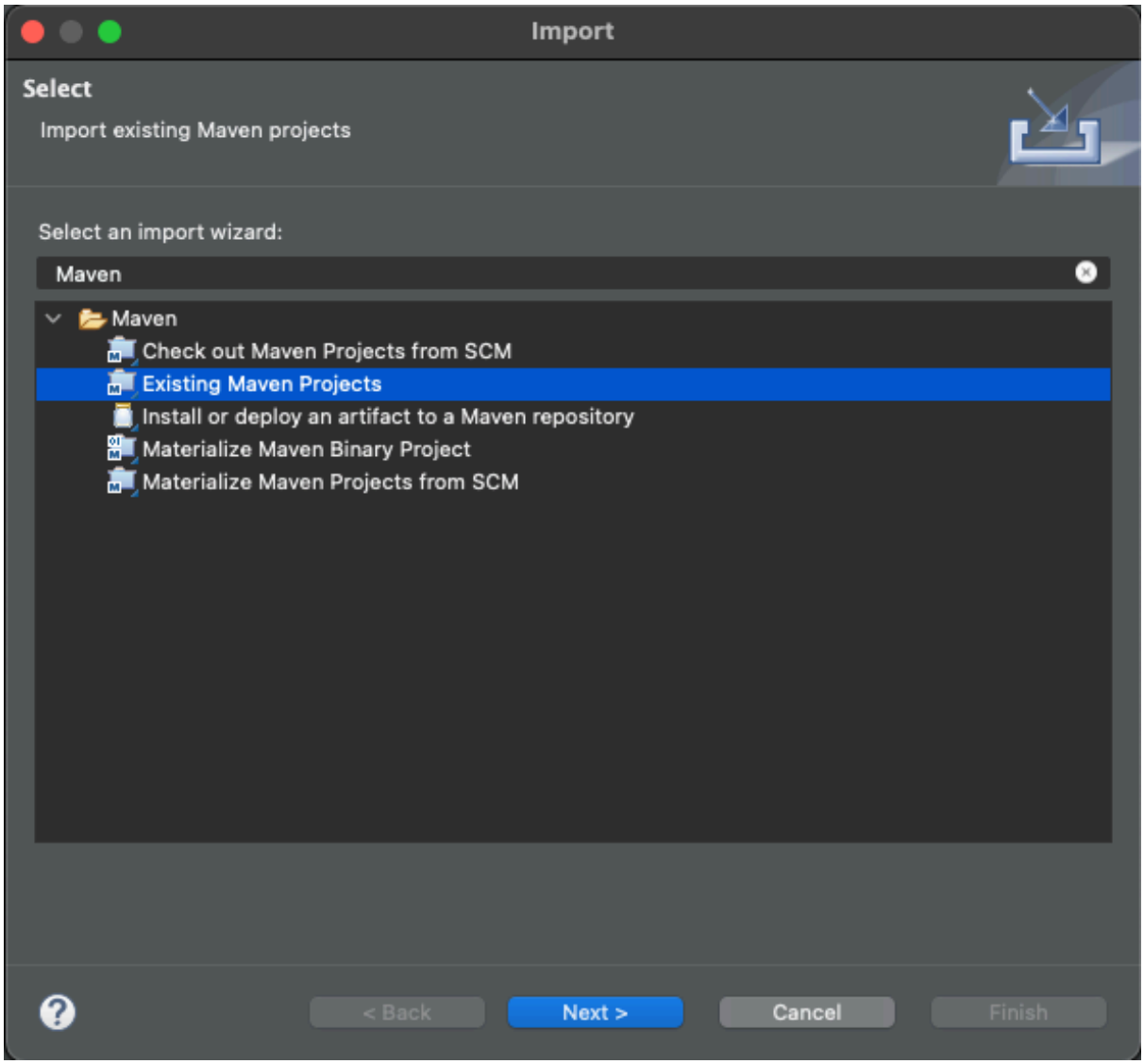
9. Move the practical.zip to your spring tools suite workspace and extract it.



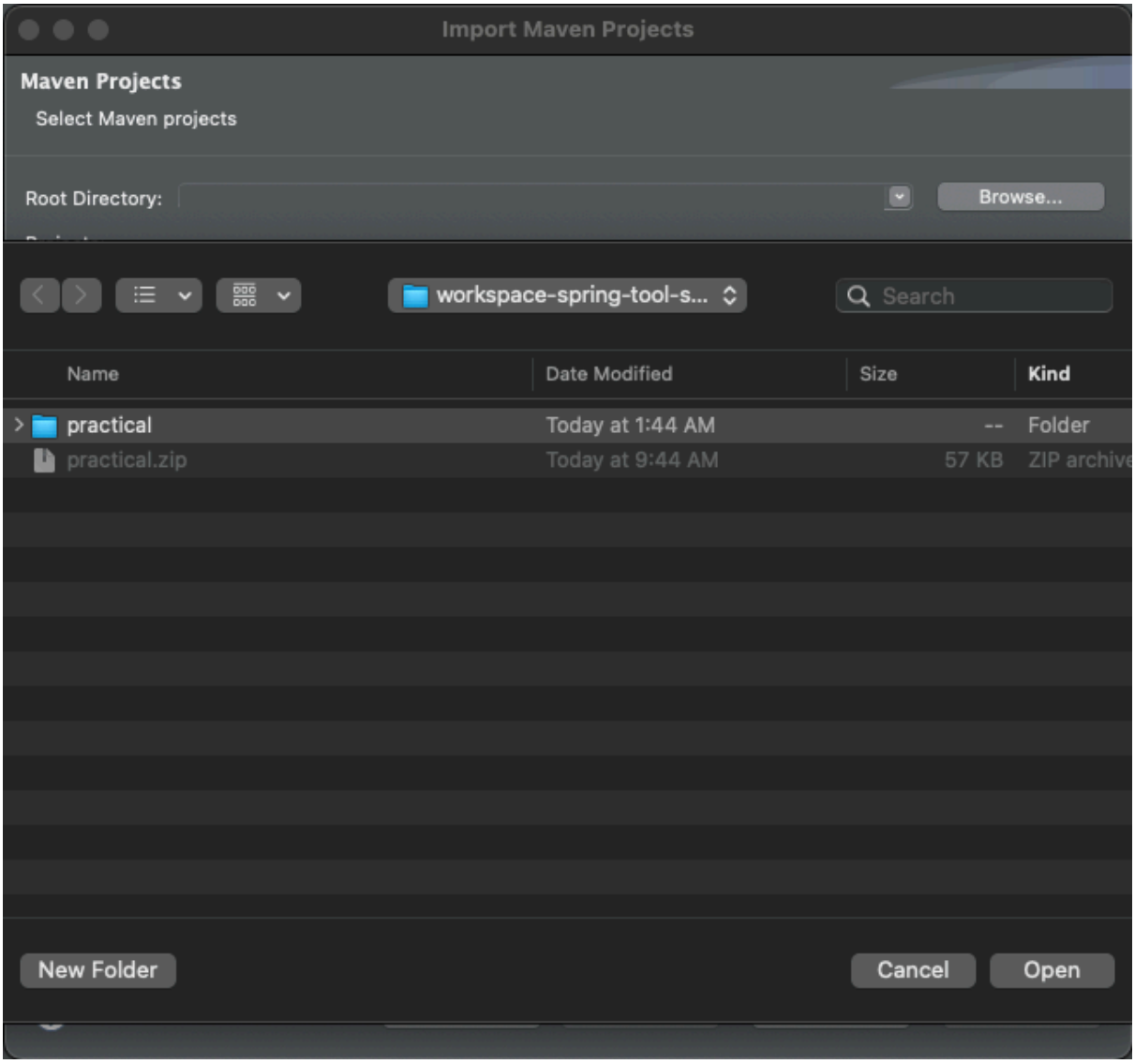
10. Go back to your Spring Tool Suite. Go to File > Import



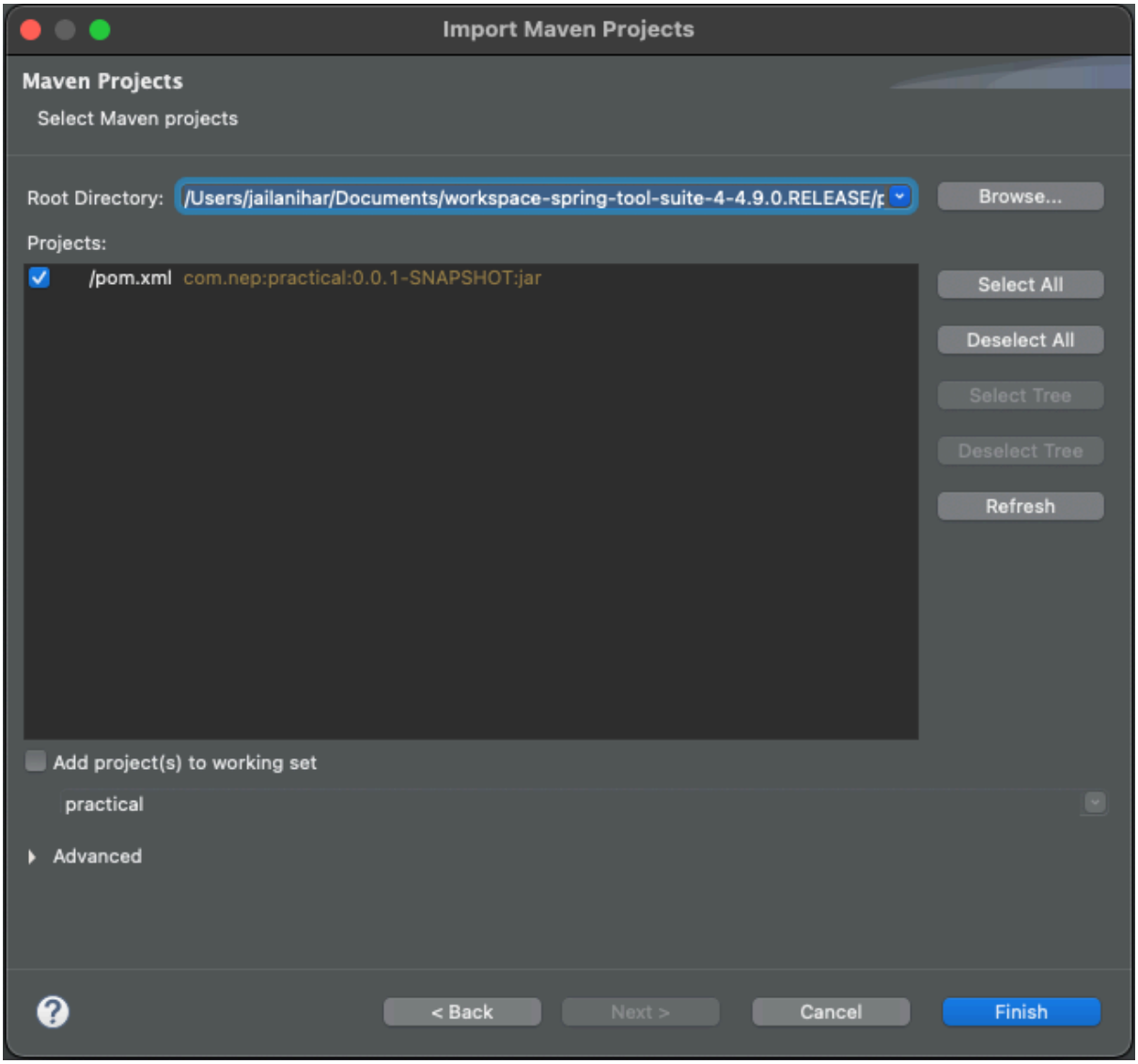
11. Find Maven > Existing Maven Projects. Then click next



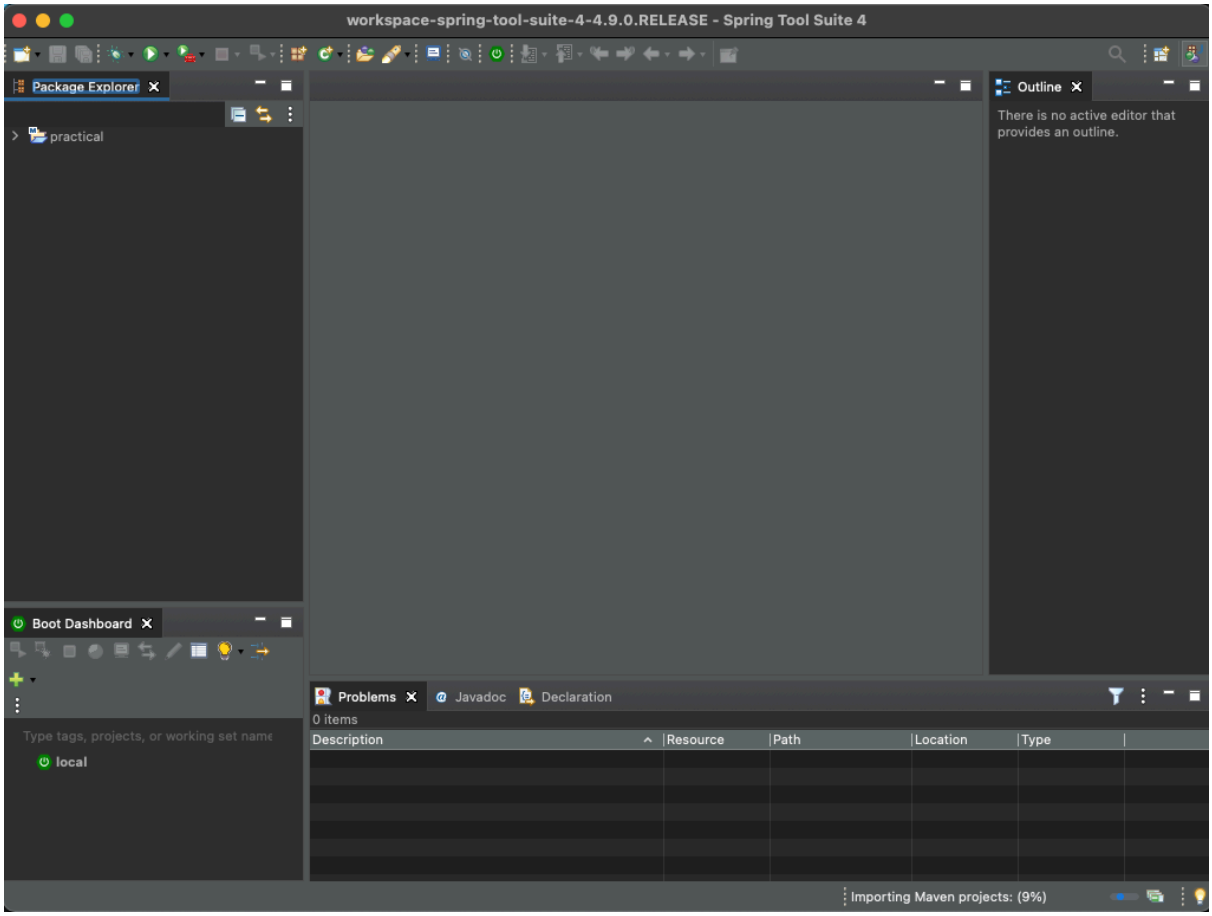
12. On Root Directory, click Browse and go to your spring tool suite workspace. Select the extracted practical project we did previously.



13. There should be pom.xml under Projects. Just click Finish after that.

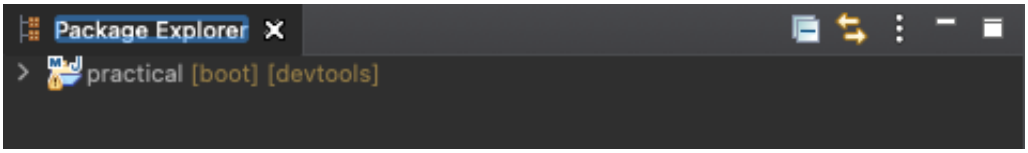


14. Depending if you have your depencies downloaded to your `<home folder>/.m2/repository`. Spring tool suite might need to download any depencies that is not available in the repository. If it requires different version from what you have in your repository. It needs to download the new versions as well. In my case, I don't have anything in my .m2 folder. That means it has to download all the dependencies. You can see it at the bottom right of spring tool suite. **Make sure you have internet access**

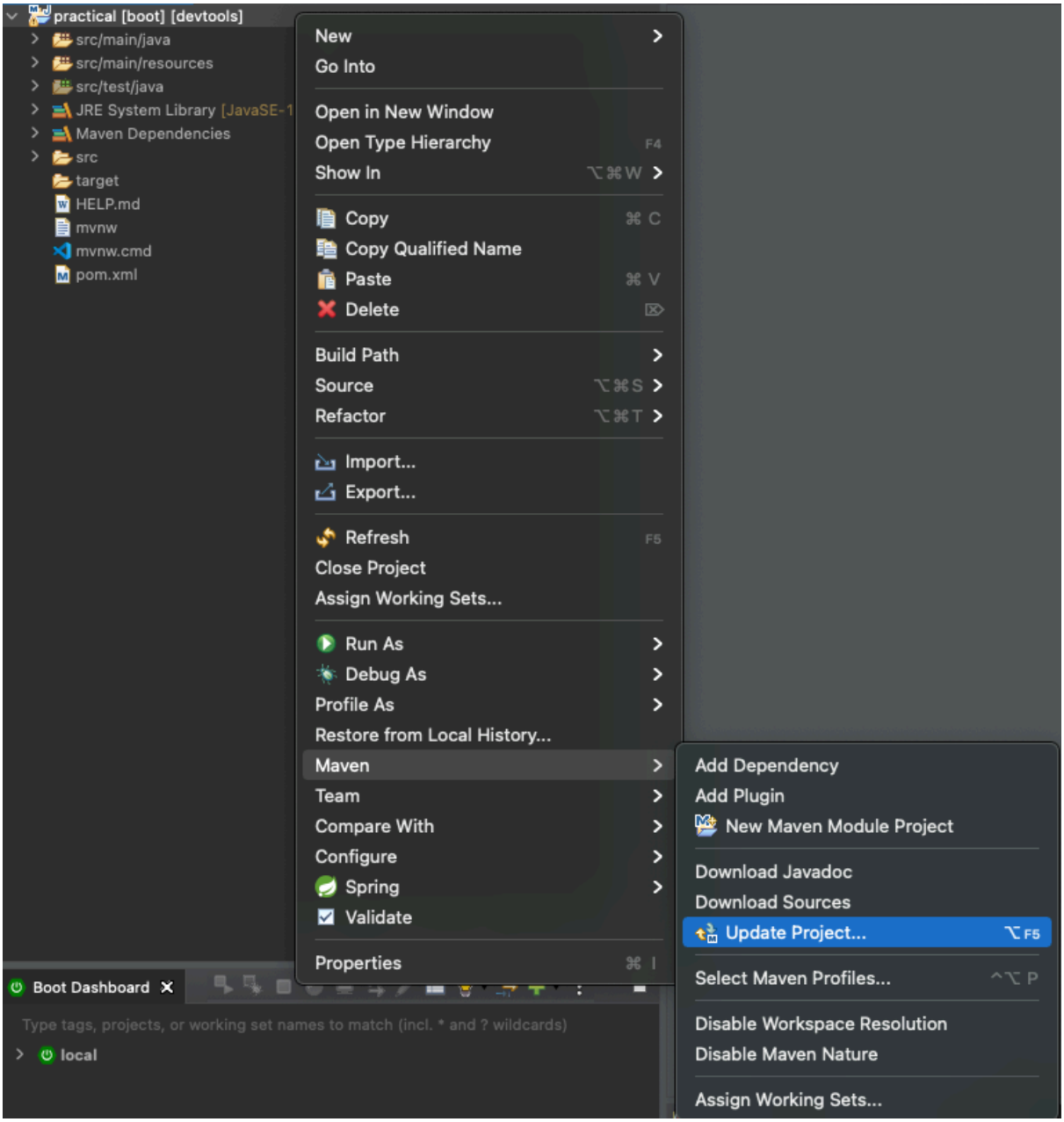


15. If you click  you will see the download progress. **Leave it to download everything until it finish. It might take a while**.

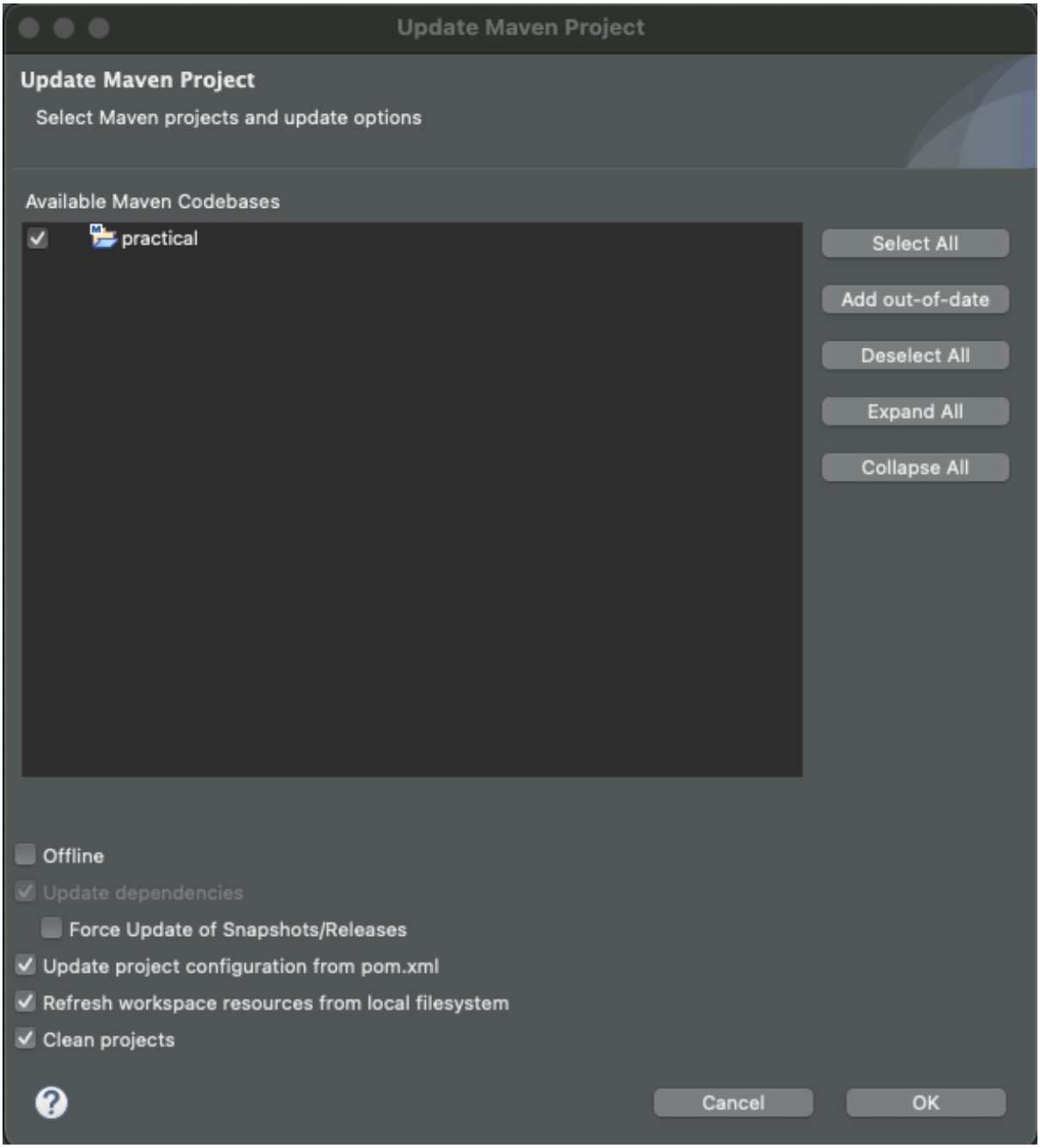
16. If it finished download or if there are no dependencies it needs to download further. Your project folder should have [boot] and [devtools].



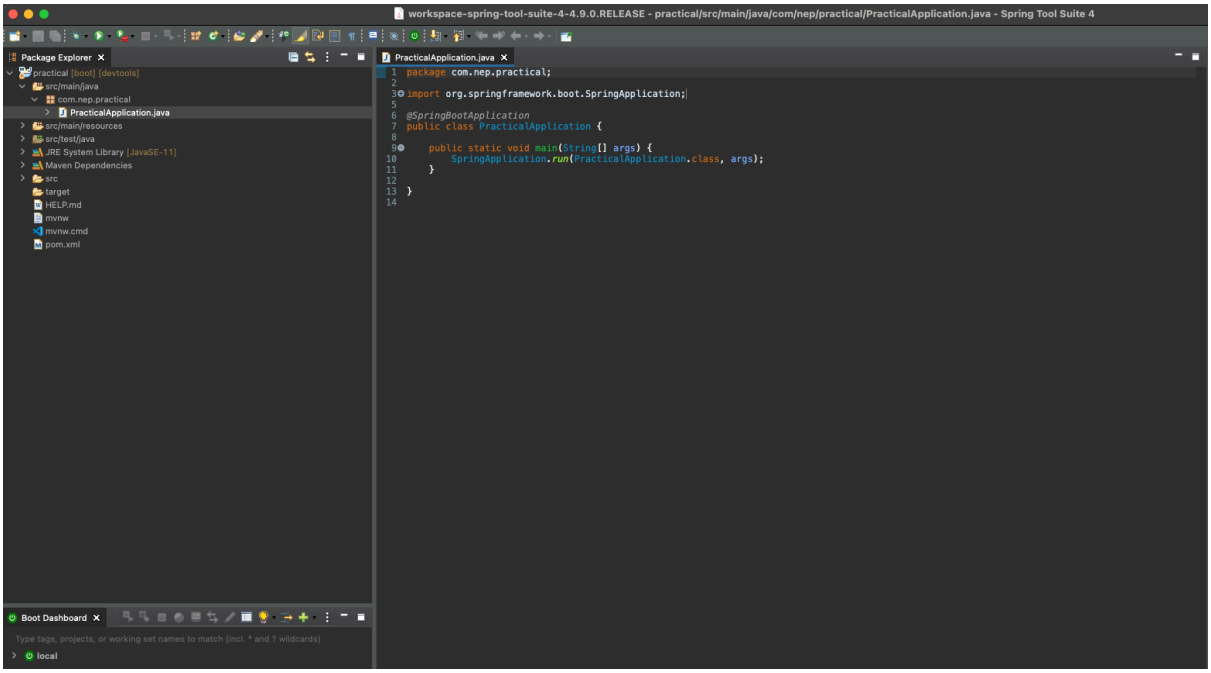
17. **If you encountered any problems** (otherwise, ignore these two steps), you might need to delete the content of .m2 folder and in Spring tool suite, Right click the project > Maven > Update Project.



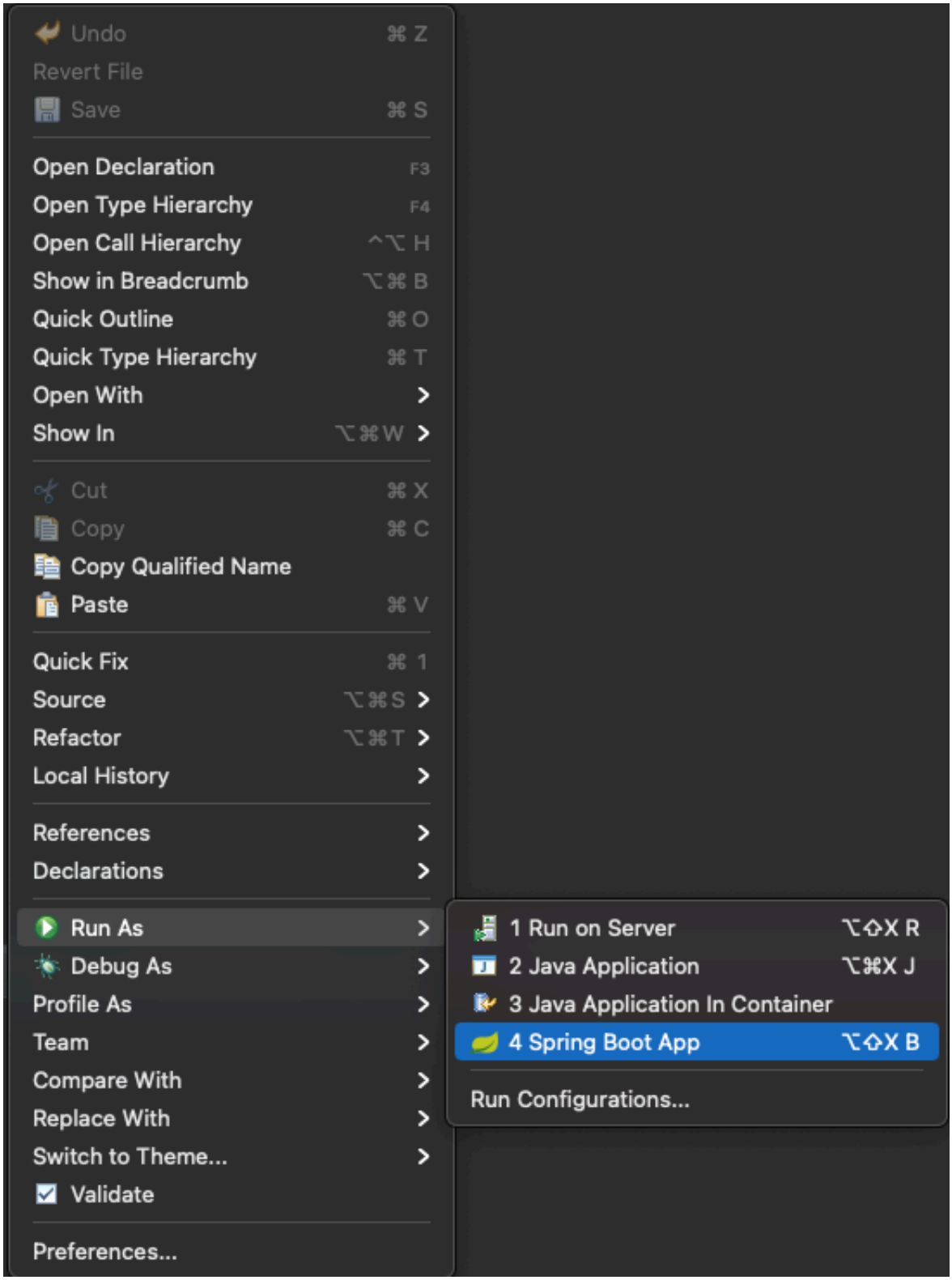
Just click ok. It should try to redownload the dependencies.



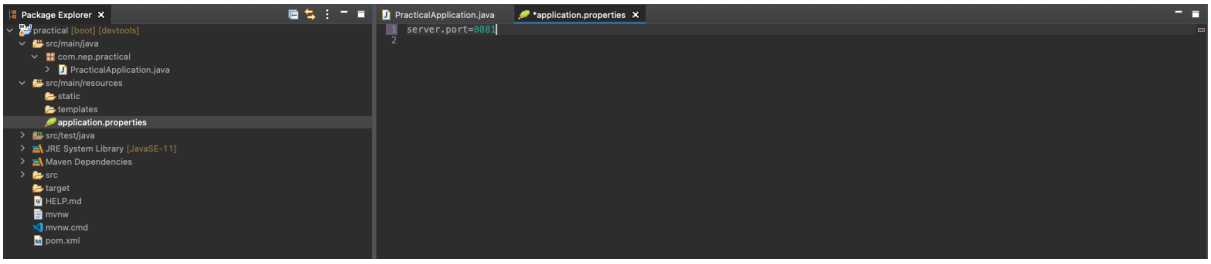
18. Open PracicalApplication.java.



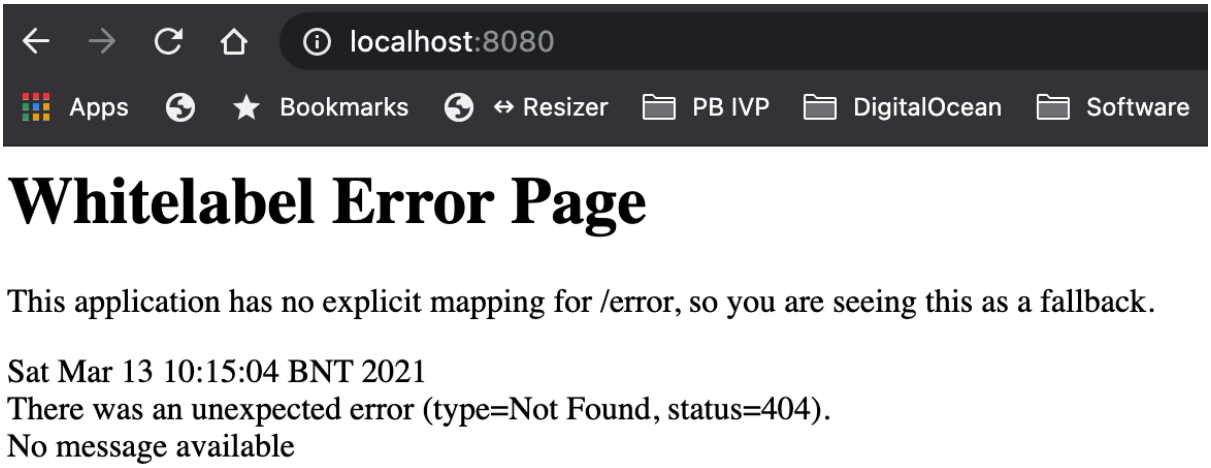
19. Right click the class itself > Run As > Spring Boot App



20. This will run the spring boot application which includes Apache Tomcat to host the spring web application. By default, Apache Tomcat will try to start on port number 8080. If your computer port number 8080 is currently used by other application. It will cause a BindException. You might need to stop the other application or change the port number Apache Tomcat to start. To change the port number. Open up application.properties. **You dont need to do this if your spring application run properly**

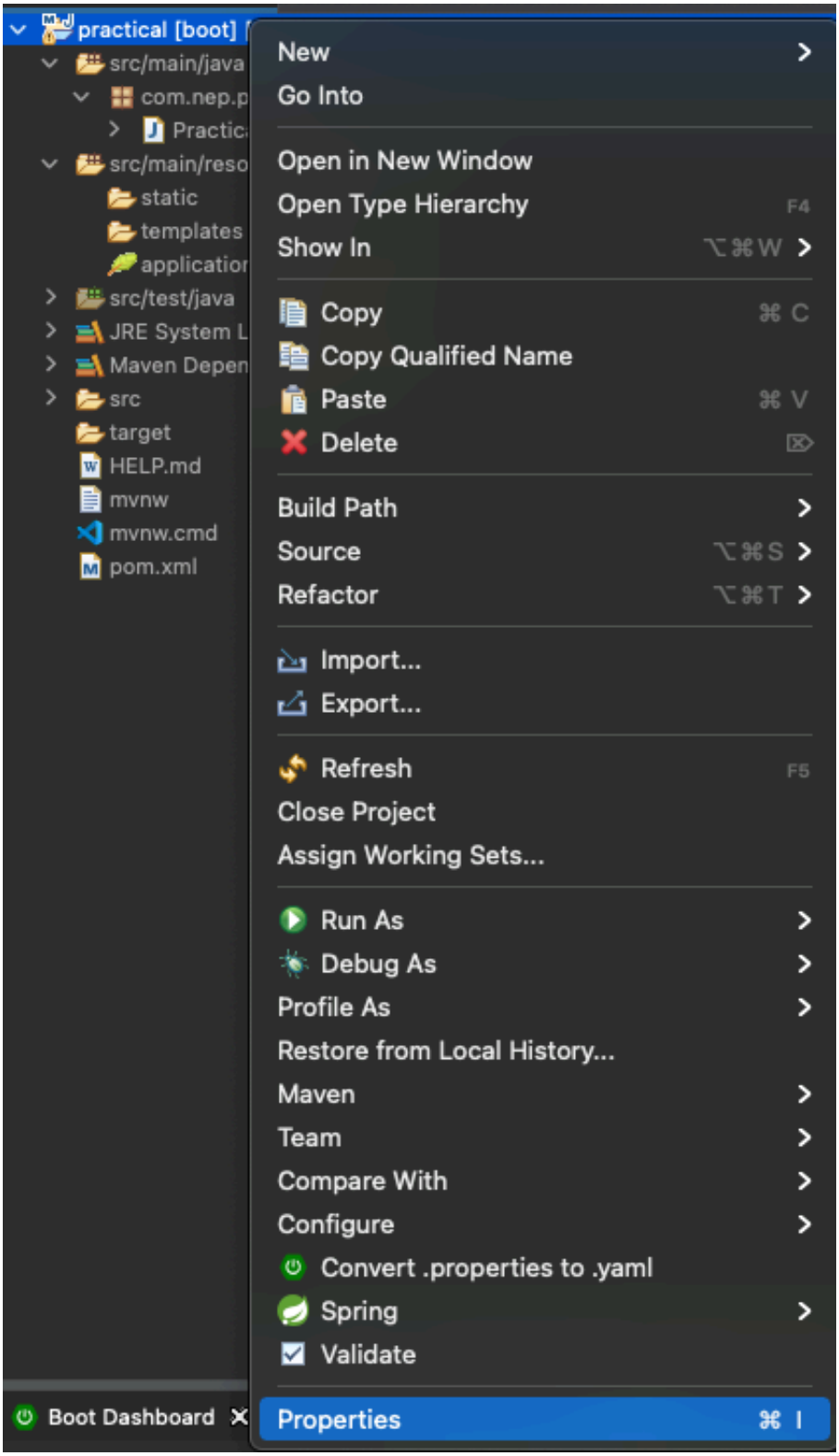


21. Once your application is running, you need to open up your web browser and go to `localhost:8080`. You should encountered Whitelable Error Page. This should be fine for now.

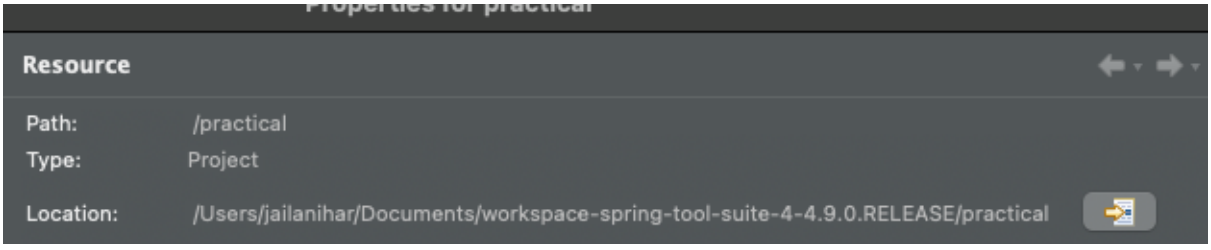


Lesson 1.2 - Setting up Git to your Spring Project

1. Right click your spring project > Properties



2. There should be Location details. Copy the location.



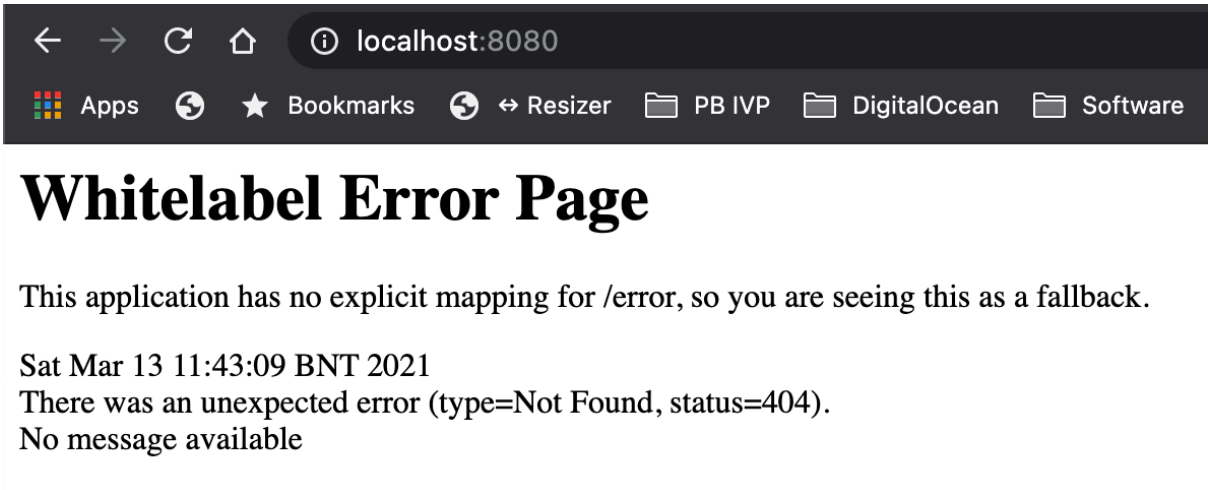
3. Open your `Terminal` / `Git Bash` and `cd <to the location>`. Example:

4. Then run `git init`.

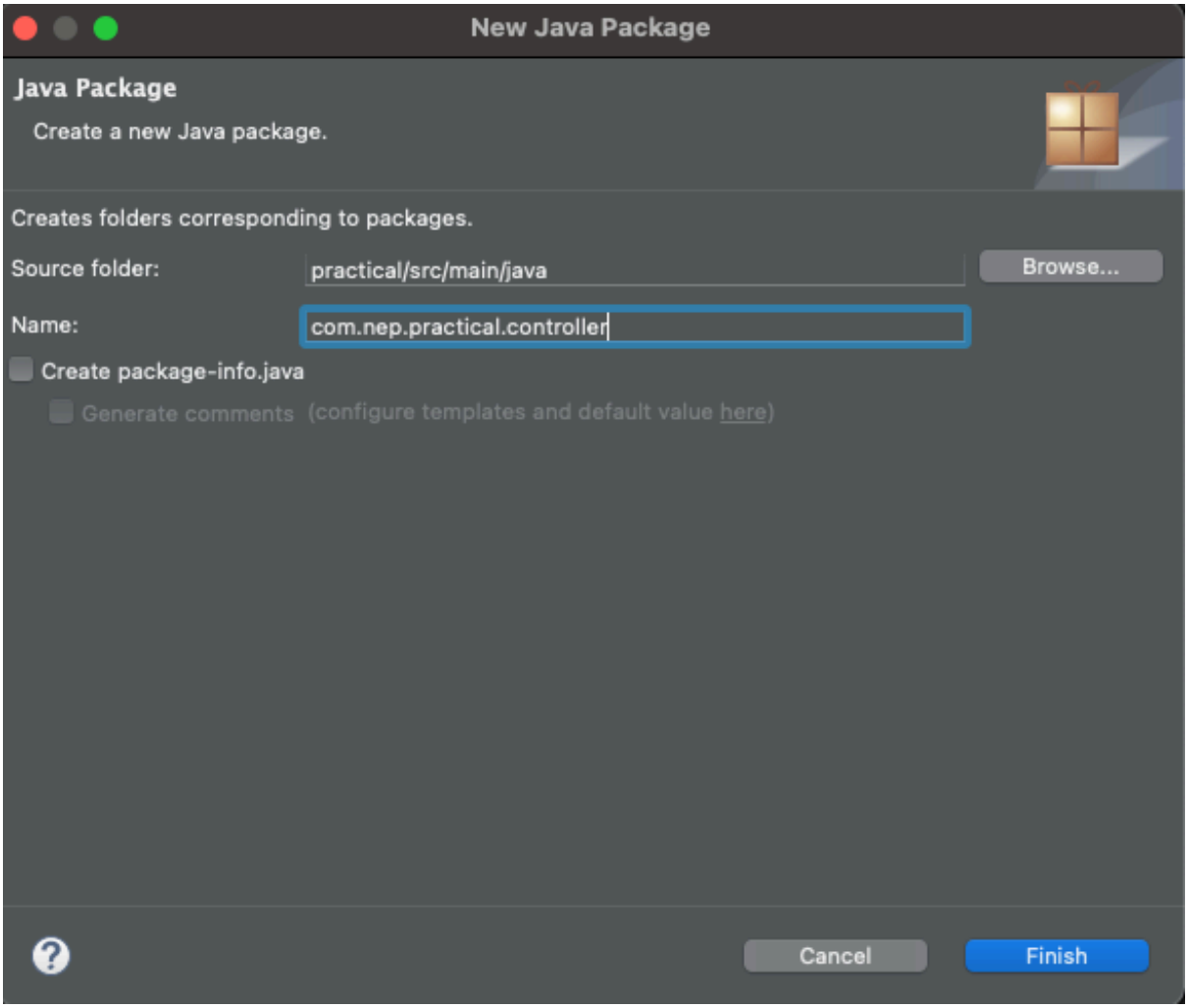
> ## ⚠️ Milestone:
>
> Commit the changes made to the `src` folder with message "Lesson 1.2 - Setting up Git to your Spring Project".
>
> Hint: Add the files to the staging area first, `git add <files>` and then proceed to commit, `git commit -m "Lesson 1.2 - Setting up Git to your Spring Project"`.

Lesson 1.3 - Spring Controller

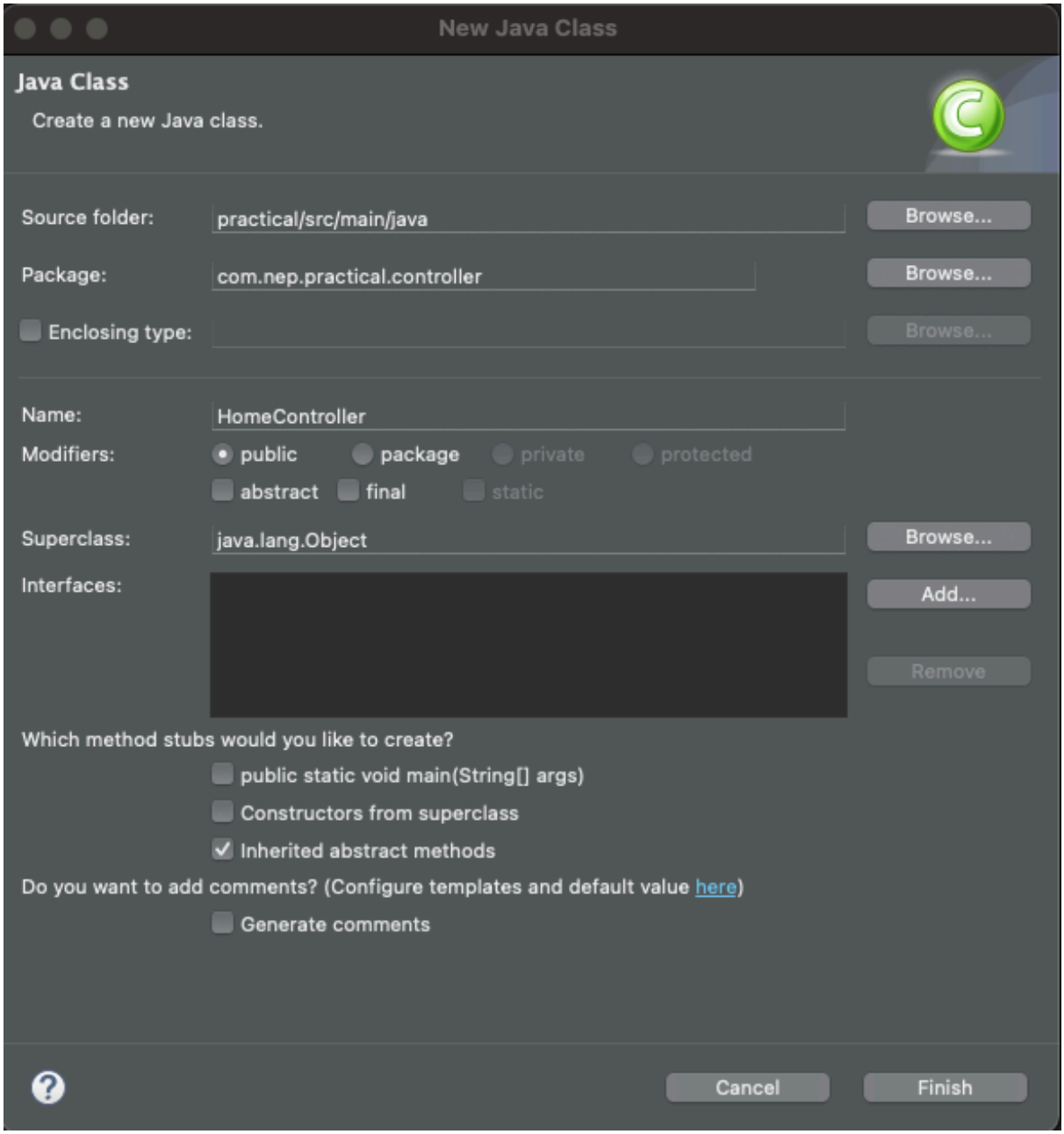
1. Previously, when we access the Uniform Resource Locator (URL), `localhost:8080` using our browser, it produce a Whitelabel Error Page. This means that the Uniform Resource Identifier (URI) for `/` is not being handler in the application. When we types localhost:8080 . By default, it would request URI `/` to the web application. Its the same as when you type `localhost:8080/` in your web browser.



2. Create a new package called `controller` in `src/main/java`, at `com.nep.practical` package.



3. Create a class called `HomeController` in the package.



4. To indicate that the Java class is a spring controller, we add an annotation on the class itself called `@Controller`, on top of the class.

```
package com.nep.practical.controller;

import org.springframework.stereotype.Controller;

@Controller
public class HomeController {

}
```

5. Then define a method called `home` that return String data type to response to URIs request.

```
package com.nep.practical.controller;

import org.springframework.stereotype.Controller;

@Controller
public class HomeController {

  public String home() {

  }

}
```

6. In this case we want to handle `/` URI request. To handle that URI request, since we want the `home` method to be called when `/` URI request is received. We assign annotation `@RequestMapping` with the URI value on top of that method.

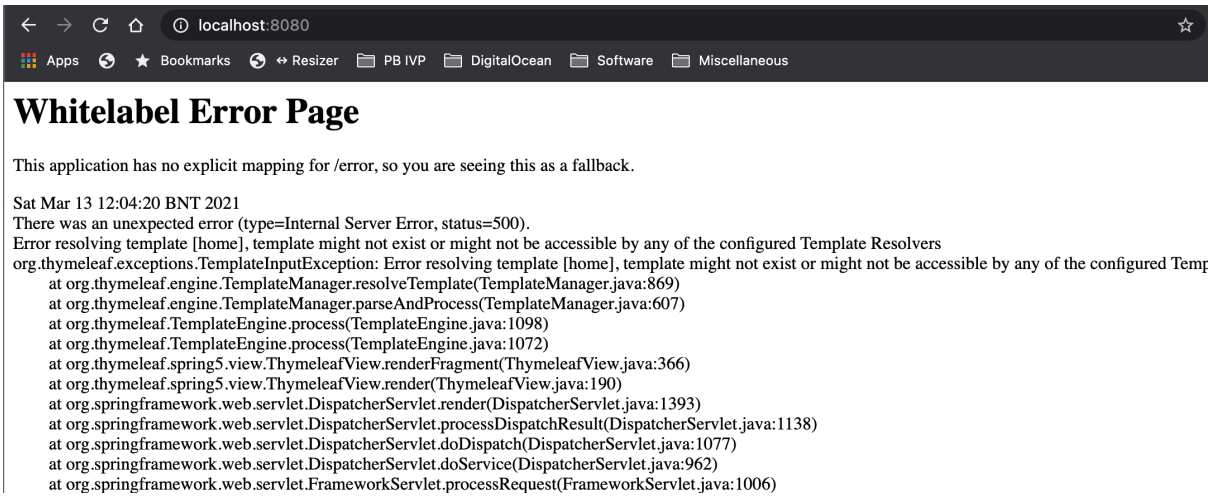
```
package com.nep.practical.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class HomeController {

  @RequestMapping(value="/")
  public String home() {

  }

}
```

7. Since we still have yet implemented any Hypertext Markup Language (HTML) files. We cannot just state the method to return a String. Lets say `return "home";`. Since we are using Thymeleaf, it will try to find the file `home.html` in `src/main/resources/templates/`.

```
package com.nep.practical.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class HomeController {
```

```
  @RequestMapping(value="/")
  public String home() {
    return "home";
  }

}
```

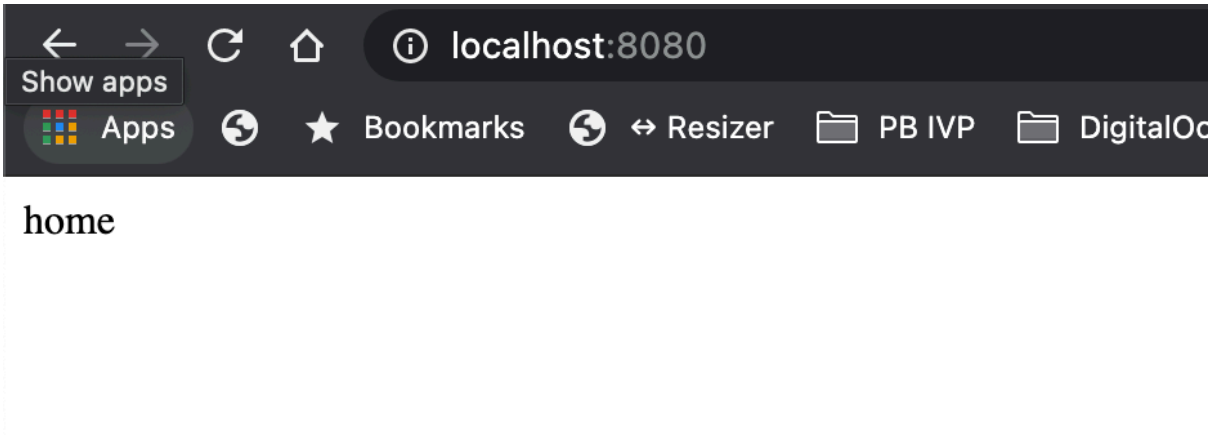
8. If you open your web browser, it should print the error stating that it could not find home.html.



9. For now we want to show the literal String "home" in the Browser. Lets add another annotation on top of the method called `@ResponseBody`.

```
package com.nep.practical.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class HomeController {

  @RequestMapping(value="/")
  @ResponseBody
  public String home() {
    return "home";
  }

}
```

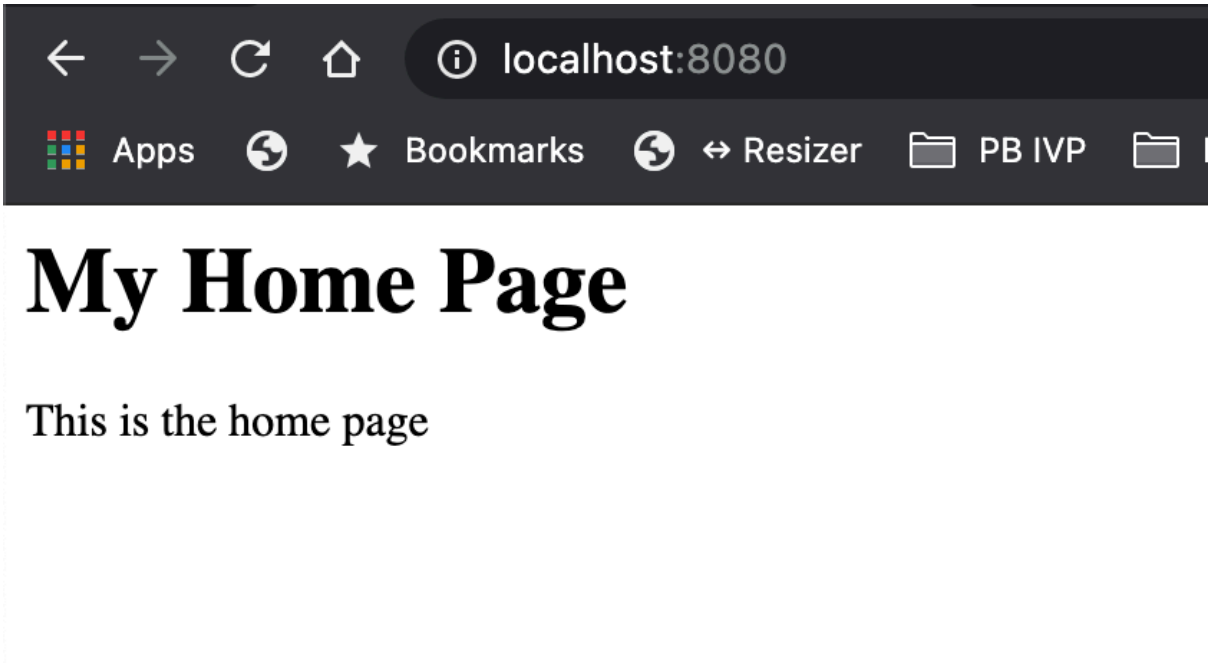
10. The literal String "home" will be displayed.



11. You could write HTML syntax in the return String.

```
package com.nep.practical.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class HomeController {

  @RequestMapping(value="/")
  @ResponseBody
  public String home() {
    return "<h1>My Home Page</h1><p>This is the home page</p>";
  }

}
```

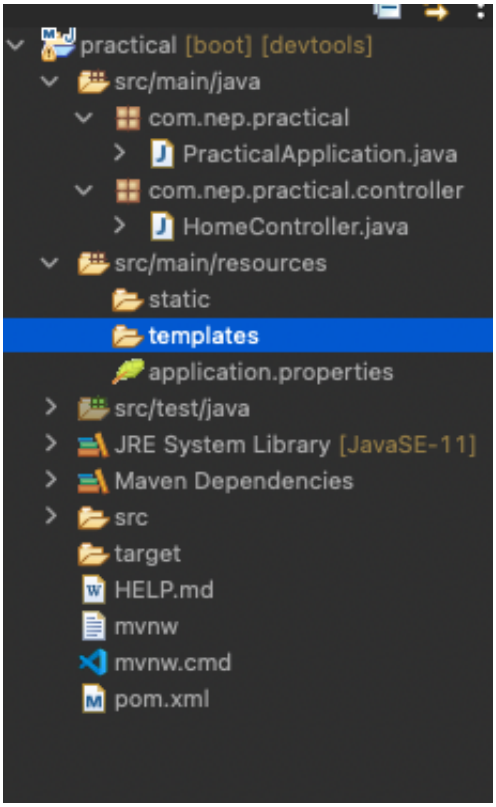
12. This will show the HTML syntax accordingly.



> ## ⚠️ Milestone:
>
> Commit the changes made to the `src` folder with message "Lesson 1.3 - Spring Controller".
>
> Hint: Add the files to the staging area first, `git add <files>` and then proceed to commit, `git commit -m "Lesson 1.3 - Spring Controller"`.

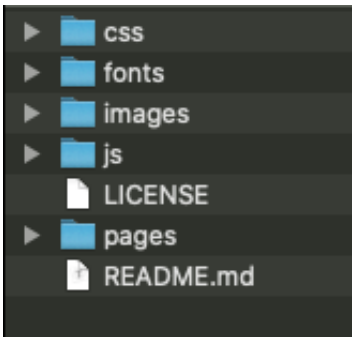
Lesson 1.4 - Spring View

1. In this practical, we are going to use Thymeleaf templating engine to allow us to write HTML file while including place holders for data that will come from Java objects. By default, Thymeleaf library wil expect the templates to be found in `src/main/resources/templates` directory of the project.

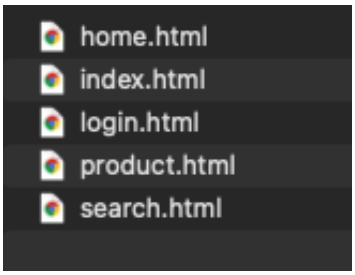


2. Download the `template_for_notes.zip` file at PBLMS. https://lms.pb.edu.bn/vle/sict/pluginfile.php/129327/mod_folder/content/0/template_for_notes.zip?forcedownload=1

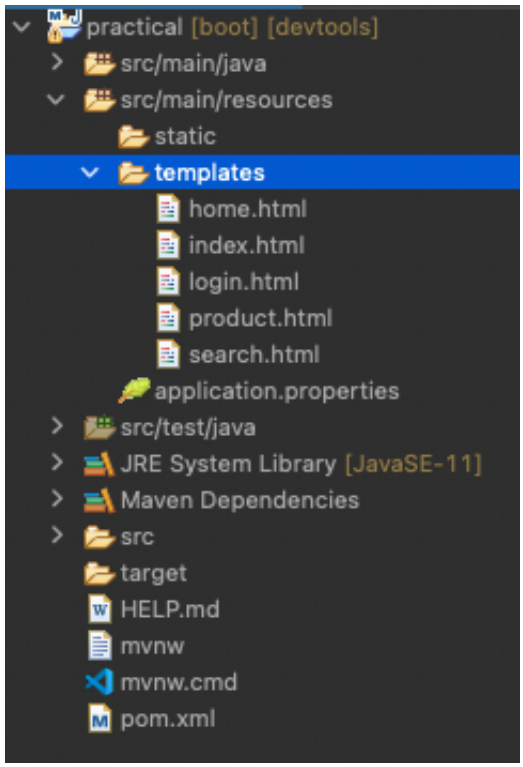
3. Extract the zip file and inside the zip file will be `template.zip`. Extract that zip file as well. You will find the contents as follows:



4. In the folder `pages` you will find html files. We will use this as our pages for our web application.



5. Copy all the html files and paste them in your project folder `src/main/resources/templates` directory.

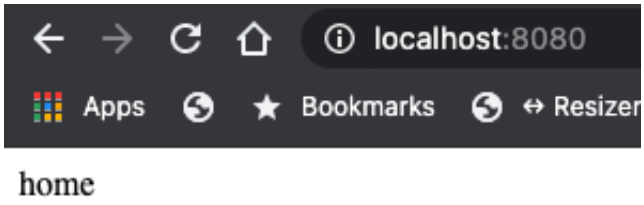


6. Now lets go back to our `HomeController.java` class and modify the `home` method. The idea here is for URI `/`, we would like to response with `home.html` file. So in the method, we would change the return value to the name of the file.

```
package com.nep.practical.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class HomeController {

  @RequestMapping(value="/")
  @ResponseBody
  public String home() {
    return "home";
  }
```

```
}
```

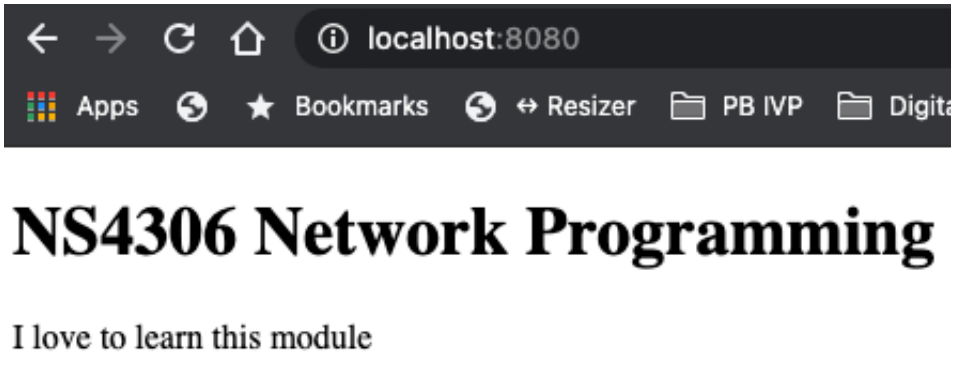
7. If you access URI `/`, it will look like this.



8. The reason why this happens is because of the annotation `@ResponseBody`. This will literally response with the String value. Lets remove the annotation.

```
package com.nep.practical.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class HomeController {

  @RequestMapping(value="/")
  public String home() {
    return "home";
  }

}
```

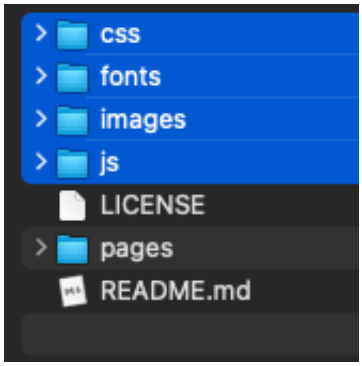
9. Now if you access URI `/`, it should show the content of `home.html`.



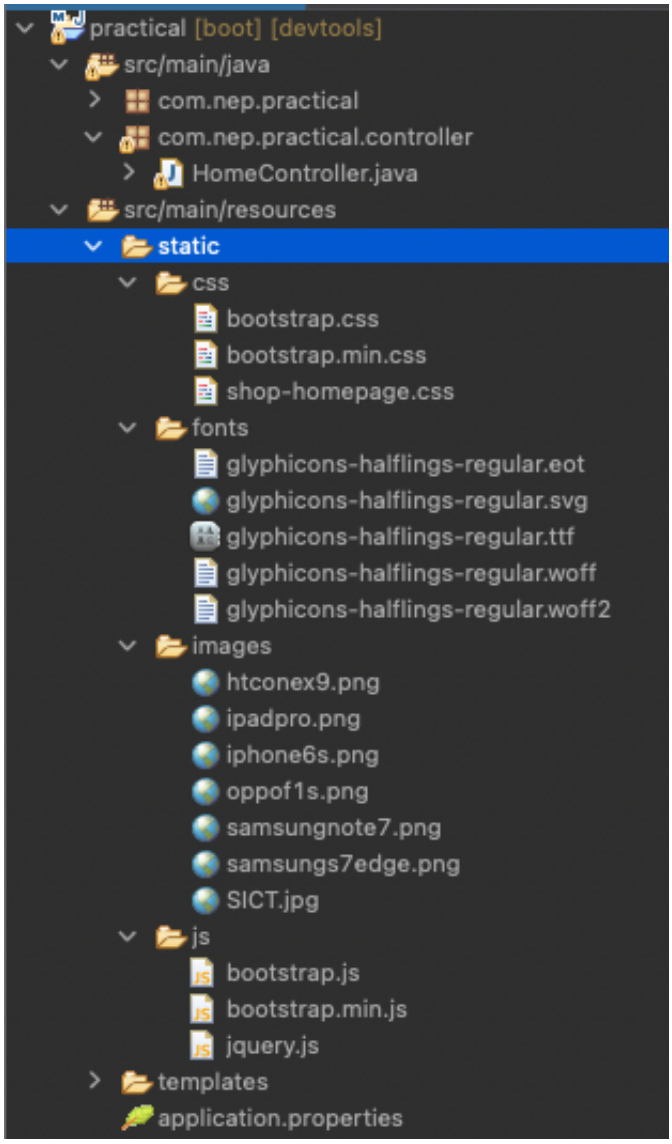
> ## ⚠️ Milestone:
>
> Commit the changes made to the `src` folder with message "Lesson 1.4 - Spring View".
>
> Hint: Add the files to the staging area first, `git add <files>` and then proceed to commit, `git commit -m "Lesson 1.4 - Spring View"`.

Lesson 1.5 - Static File

1. Static files are files that its content does not change that your template will use. For example, images, css and javascript files. In `template_for_notes.zip`, there are some static files that you need to add to your web application. Copy `css`, `fonts`, `images` and `js` folder.



2. Paste it to your project folder `src/main/resources/static`.



3. Lets add an image file into your current web application, `home.html`. Instead of entering an absolute path for the static files, we will let Thymeleaf figure out the path by using url expression. This will leverage the templating engine to generate a URL for us. For this case, we will use the `img` html tag. Add the tag under the `<!-- Add the img tag -->` comment.

```
<!DOCTYPE html>
<!-- TODO -->
<!-- Slide #19 Example Thymeleaf URL Expression -->
<!-- Add xml namespace for th to http://www.thymeleaf.org -->
<html lang="en">

<head>
    <title>I like Network Programming</title>
</head>

<body>
    <h1>NS4306 Network Programming</h1>
    <p>I love to learn this module</p>
    <!-- TODO -->
    <!-- Slide #19 Example Thymeleaf URL Expression -->
```

```
    <!-- Add the img tag -->
    <img src="" />
</body>

</html>
```

4. Then add a prefix `th:` to the `img` tag attribute `src`. This will notify Thymeleaf to process this part of html file.

```
<!DOCTYPE html>
<!-- TODO -->
<!-- Slide #19 Example Thymeleaf URL Expression -->
<!-- Add xml namespace for th to http://www.thymeleaf.org -->
<html lang="en">

<head>
    <title>I like Network Programming</title>
</head>

<body>
    <h1>NS4306 Network Programming</h1>
    <p>I love to learn this module</p>
    <!-- TODO -->
    <!-- Slide #19 Example Thymeleaf URL Expression -->
    <!-- Add the img tag -->
    <img th:src="" />
</body>

</html>
```

5. The value for `th:src` attribute will be `@{ /images/SICT.jpg }`. `@` sign indicates to the Thymeleaf engine that a URL should be generated here and `{ }` sign indicates the content inside the curly braces is the URL or URI that needs to be modified. Please refer to Java Web Programming II lecture notes to learn more on the syntax for Thymeleaf URL expression.

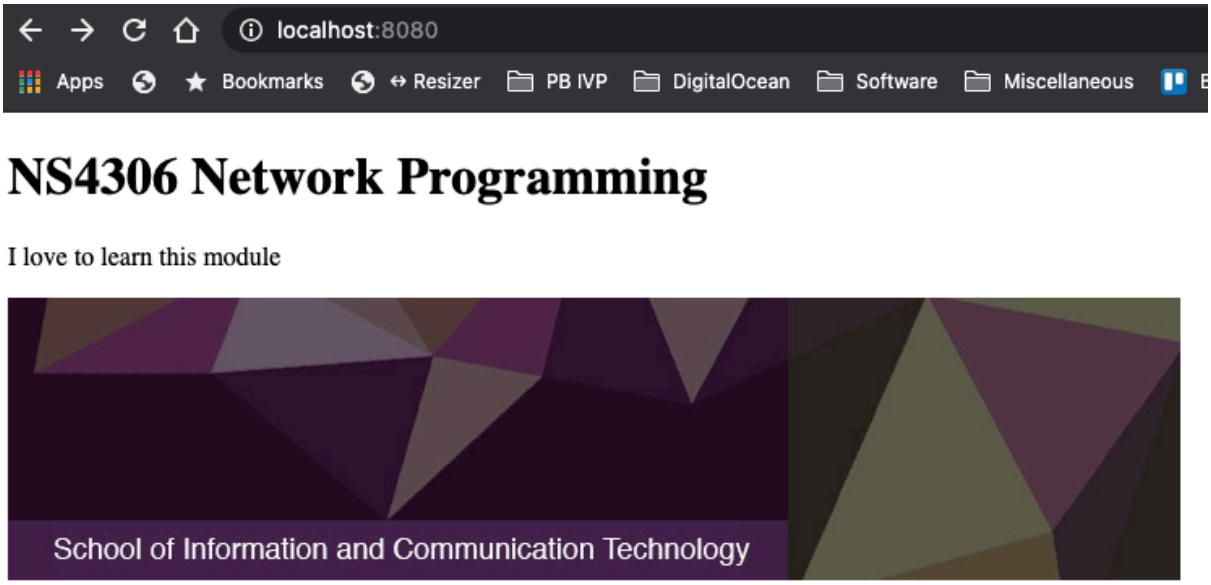
```
<!DOCTYPE html>
<!-- TODO -->
<!-- Slide #19 Example Thymeleaf URL Expression -->
<!-- Add xml namespace for th to http://www.thymeleaf.org -->
<html lang="en">

<head>
    <title>I like Network Programming</title>
</head>
```

```
<body>
    <h1>NS4306 Network Programming</h1>
    <p>I love to learn this module</p>
    <!-- TODO -->
    <!-- Slide #19 Example Thymeleaf URL Expression -->
    <!-- Add the img tag -->
    <img th:src="@{ /images/SICT.jpg }" />
</body>

</html>
```

6. If you have added `th:` to some attributes, some IDE might consider this as an error. You

```
<!DOCTYPE html>
<!-- TODO -->
<!-- Slide #19 Example Thymeleaf URL Expression -->
<!-- Add xml namespace for th to http://www.thymeleaf.org -->
<html lang="en" xmlns:th="http://www.thymeleaf.org">

<head>
    <title>I like Network Programming</title>
</head>

<body>
    <h1>NS4306 Network Programming</h1>
    <p>I love to learn this module</p>
    <!-- TODO -->
    <!-- Slide #19 Example Thymeleaf URL Expression -->
    <!-- Add the img tag -->
    <img th:src="@{ /images/SICT.jpg }" />
</body>

</html>
```

7. Now if you access URI `/`, it should show the content of `home.html`.



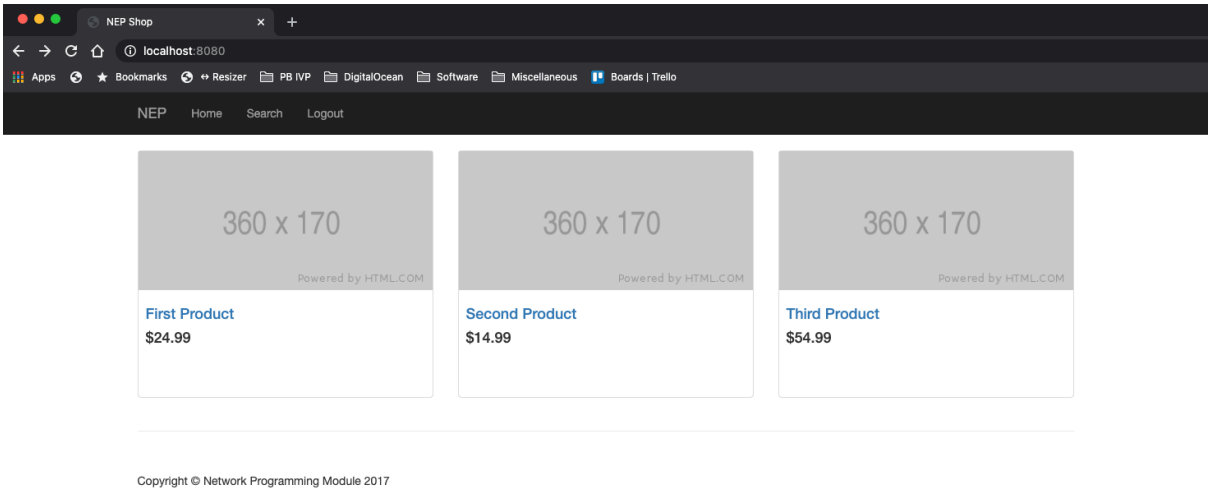
> ## ⚠️ Milestone:
>
> Commit the changes made to the `src` folder with message "Lesson 1.5 - Static File".
>
> Hint: Add the files to the staging area first, `git add <files>` and then proceed to commit, `git commit -m "Lesson 1.5 - Static File"`.

Lesson 1.6 - Spring Model

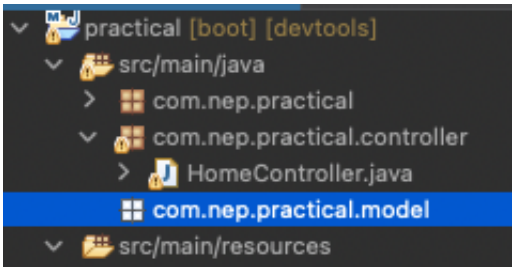
1. Currently, we managed to serve web application with static web content only. We have yet to implement a web application that have data that are dynamically generated. For now we going to change the html file that we serve for URI `/`. Lets change from `return "home"` to `return "index"`.

```
package com.nep.practical.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class HomeController {

  @RequestMapping(value="/")
  public String home() {
    return "index";
  }

}
```

2. This will display the content of `index.html`.



3. We are going to create Plain Old Java Object (POJO) classes to model the data of the web application. Lets create a new package called `com.nep.practical.model` in `src/main/java`.



4. Create a new class called `Product` in the package.

```
package com.nep.practical.model;

public class Product {

}
```

5. A POJO class, will only have fields, constructors, getters and setters, and methods that are useful to the object. For now we need the name of the product, price, file and is the product in stock or not. Lets implement those fields to this POJO class.

```
package com.nep.practical.model;

public class Product {

  private String name;
  private double price;
  private String file;
  private boolean inStock;

}
```

6. We need to add constructor, getters and setters methods. It is important you follow the standard naming scheme for getters and setters methods.

```
package com.nep.practical.model;

public class Product {

  private String name;
  private double price;
  private String file;
  private boolean inStock;

  public Product(String name, double price, String file, boolean inStock) {
    this.name = name;
    this.price = price;
    this.file = file;
    this.inStock = inStock;
  }

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }

  public double getPrice() {
    return price;
  }

  public void setPrice(double price) {
    this.price = price;
  }

  public String getFile() {
```

```
    return file;
  }

  public void setFile(String file) {
    this.file = file;
  }

  public boolean isInStock() {
    return inStock;
  }

  public void setInStock(boolean inStock) {
    this.inStock = inStock;
  }

}
```

7. Making an object in a controller is the same as making an object in any other Java classes. For now, lets create the object inside the `home` method. The file `/images/htconex9.png` is already included when you copy them `images` folder in the template to the `static` folder in the project.

```
package com.nep.practical.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

import com.nep.practical.model.Product;

@Controller
public class HomeController {

  @RequestMapping(value="/")
  public String home() {
    Product product = new Product("HTC One X9", 900.00,
"/images/htconex9.png", false);
    return "index";
  }

}
```

8. For Thymeleaf to access the object in the controller, you need to pass a `ModelMap` object as the parameter.

```
package com.nep.practical.controller;
```

```
import org.springframework.stereotype.Controller;
import org.springframework.ui.ModelMap;
import org.springframework.web.bind.annotation.RequestMapping;

import com.nep.practical.model.Product;

@Controller
public class HomeController {

  ...
  public String home(ModelMap modelMap) {
    ...
  }

}
```

9. Then use the ModelMap `put` method to insert objects into it. You need to assign a key to the object. This key will be use to access the object in `index.html`.

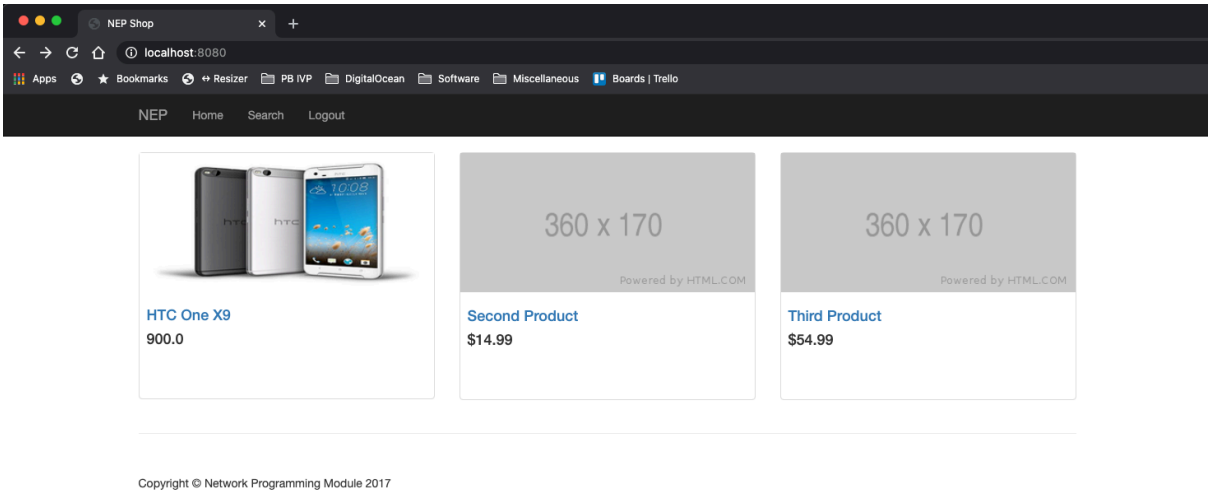
```
package com.nep.practical.controller;

import org.springframework.stereotype.Controller;
import org.springframework.ui.ModelMap;
import org.springframework.web.bind.annotation.RequestMapping;

import com.nep.practical.model.Product;

@Controller
public class HomeController {

  @RequestMapping(value="/")
  public String home(ModelMap modelMap) {
    Product product = new Product("HTC One X9", 900.00,
"/images/htconex9.png", false);
    modelMap.put("product", product);
    return "index";
  }

}
```

10. The key `"product"` mentioned in the ModelMap can be used by Thymeleaf to access the data and generate the data into its template, HTML file. Lets modify `index.html` to access the object. In this case, you need to add the following into the attribute inside certain HTML tags.

- Add `th:` prefix to the attribute.
- In the value of the attribute, use dollar sign `$` symbol and then curly braces `{}`
- Inside the curly braces, put in the key of the object data that you want to retrieve followed by the dot operator `.` and the field you want to retrieve.
- For adding text to the page, you need `th:text` attribute. **Dont forget to remove the static text from the tag**

```
...

                            <!-- TODO -->
                            <!-- Slide #32 Example Data to Thymeleaf
Templates -->
                            <img th:src="${ product.file }" alt="">
                            <div class="caption">
                                <!-- TODO -->
                                <!-- Slide #32 Example Data to Thymeleaf
Templates -->
                                <!-- Slide #51 Example Dynamic Page (cont.)
-->
                                <a href="product.html"><h4 th:text="${
product.name }"></h4></a>
                                <!-- TODO -->
                                <!-- Slide #32 Example Data to Thymeleaf
Templates -->
                                <h4 th:text="${ product.price }"></h4>
                            </div>

...
```

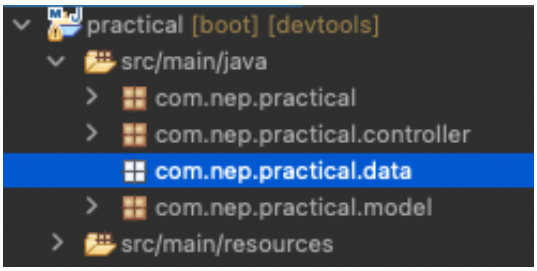
11. Access `/` URI. It should display the details from the `Product` object passed through the controller.



> # ⚠️ Milestone:
>
> Commit the changes made to the `src` folder with message "Lesson 1.6 - Spring Model".
>
> Hint: Add the files to the staging area first, `git add <files>` and then proceed to commit, `git commit -m "Lesson 1.6 - Spring Model"`.

Lesson 1.7 - Data Repository

1. At the moment it is only able to display the details of one hard coded object. A repository is essentially a collection that we have stored somewhere in memory or in a database. Lets create a new package called `com.nep.practical.data` in `src/main/java` directory.



2. Create a new class called `ProductRepository` in the package. This class will act as both the storage for objects and a methods for interacting with those objects.

```
package com.nep.practical.data;

public class ProductRepository {

}
```

3. To reduce our coding, we are going to allow Spring to initialise fields for us if it find a spring component of the same class for the field. Add `@Component` annotation on top of the class definition.

```
package com.nep.practical.data;

import org.springframework.stereotype.Component;

@Component
public class ProductRepository {

}
```

4. For now we are going to store it within the application using static Java list but in the future we will be interacting with a database.

```
package com.nep.practical.data;

import java.util.Arrays;
import java.util.List;

import org.springframework.stereotype.Component;

import com.nep.practical.model.Product;

@Component
public class ProductRepository {

  private static final List<Product> ALL_PRODUCTS = Arrays.asList(
      new Product("Apple iPhone 6s", 1210.50, "/images/iphone6s.png",
true),
      new Product("Apple iPad Pro", 1310.99, "/images/ipadpro.png", true),
      new Product("Samsung Galaxy S7 Edge", 835.92,
"/images/samsungs7edge.png", false),
      new Product("Samsung Galaxy Note 7", 1035.92,
"/images/samsungnote7.png", true),
      new Product("HTC One X9", 850.20, "/images/htconex9.png", false),
      new Product("Oppo F1s", 400.28, "/images/oppof1s.png", true)
      );

}
```

5. Add a getter method to allow retrieval of all of the products.

```
package com.nep.practical.data;

import java.util.Arrays;
import java.util.List;

import org.springframework.stereotype.Component;

import com.nep.practical.model.Product;

@Component
public class ProductRepository {

  private static final List<Product> ALL_PRODUCTS = Arrays.asList(
```

```
      new Product("Apple iPhone 6s", 1210.50, "/images/iphone6s.png",
true),
      new Product("Apple iPad Pro", 1310.99, "/images/ipadpro.png", true),
      new Product("Samsung Galaxy S7 Edge", 835.92,
"/images/samsungs7edge.png", false),
      new Product("Samsung Galaxy Note 7", 1035.92,
"/images/samsungnote7.png", true),
      new Product("HTC One X9", 850.20, "/images/htconex9.png", false),
      new Product("Oppo F1s", 400.28, "/images/oppof1s.png", true)
      );

  public static List<Product> getAllProducts() {
    return ALL_PRODUCTS;
  }

}
```

6. Now its time to retrieve the data from the repository in the controller. Since we set the repository class as Spring Component (Using `@Component` annotation). We need to declare the repository as the controller field and annotate it with `@Autowired` for Spring to initialise the field.

```
package com.nep.practical.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.ModelMap;
import org.springframework.web.bind.annotation.RequestMapping;

import com.nep.practical.data.ProductRepository;
import com.nep.practical.model.Product;

@Controller
public class HomeController {

  @Autowired
  ProductRepository productRepository;

  @RequestMapping(value="/")
  public String home(ModelMap modelMap) {
    Product product = new Product("HTC One X9", 900.00,
"/images/htconex9.png", false);
    modelMap.put("product", product);
    return "index";
  }

}
```

7. Lets remove the Product object in the `home` method and pass the list of products to the `ModelMap`. There will be a warning for `productRepository.getAllProducts()` since it is a static method. For now we will ignore it. The main thing is that to understand `@Component` and `@Autowired` annotation.

```
package com.nep.practical.controller;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.ModelMap;
import org.springframework.web.bind.annotation.RequestMapping;

import com.nep.practical.data.ProductRepository;
import com.nep.practical.model.Product;

@Controller
public class HomeController {

  @Autowired
  ProductRepository productRepository;

  @RequestMapping(value="/")
  public String home(ModelMap modelMap) {
    List<Product> products = productRepository.getAllProducts();
    modelMap.put("products", products);
    return "index";
  }

}
```

8. Then we need to update our `index.html` file to handle a list of products. First remove the following codes.

```
...
                ...
                    <!-- TODO -->
                    <!-- Remove this div because in Slide #44 Example List
of Data to Thymeleaf Templates -->
                    <!-- The foreach will generate this automatically -->
                    <div class="col-sm-4 col-lg-4 col-md-4">
                        <div class="thumbnail">
                            <img src="http://placehold.it/360x170" alt="">
```

```
                            <div class="caption">
                                <a href="product.html"><h4>Second
Product</h4></a>
                                <h4>$14.99</h4>
                            </div>
                        </div>
                    </div>
                    <!-- TODO -->
                    <!-- Remove this div because in Slide #44 Example List
of Data to Thymeleaf Templates -->
                    <!-- The foreach will generate this automatically -->
                    <div class="col-sm-4 col-lg-4 col-md-4">
                        <div class="thumbnail">
                            <img src="http://placehold.it/360x170" alt="">
                            <div class="caption">
                                <a href="product.html"><h4>Third
Product</h4></a>
                                <h4>$54.99</h4>
                            </div>
                        </div>
                    </div>
                ...
...
```

9. In this `<div>` tag, you should have these left.

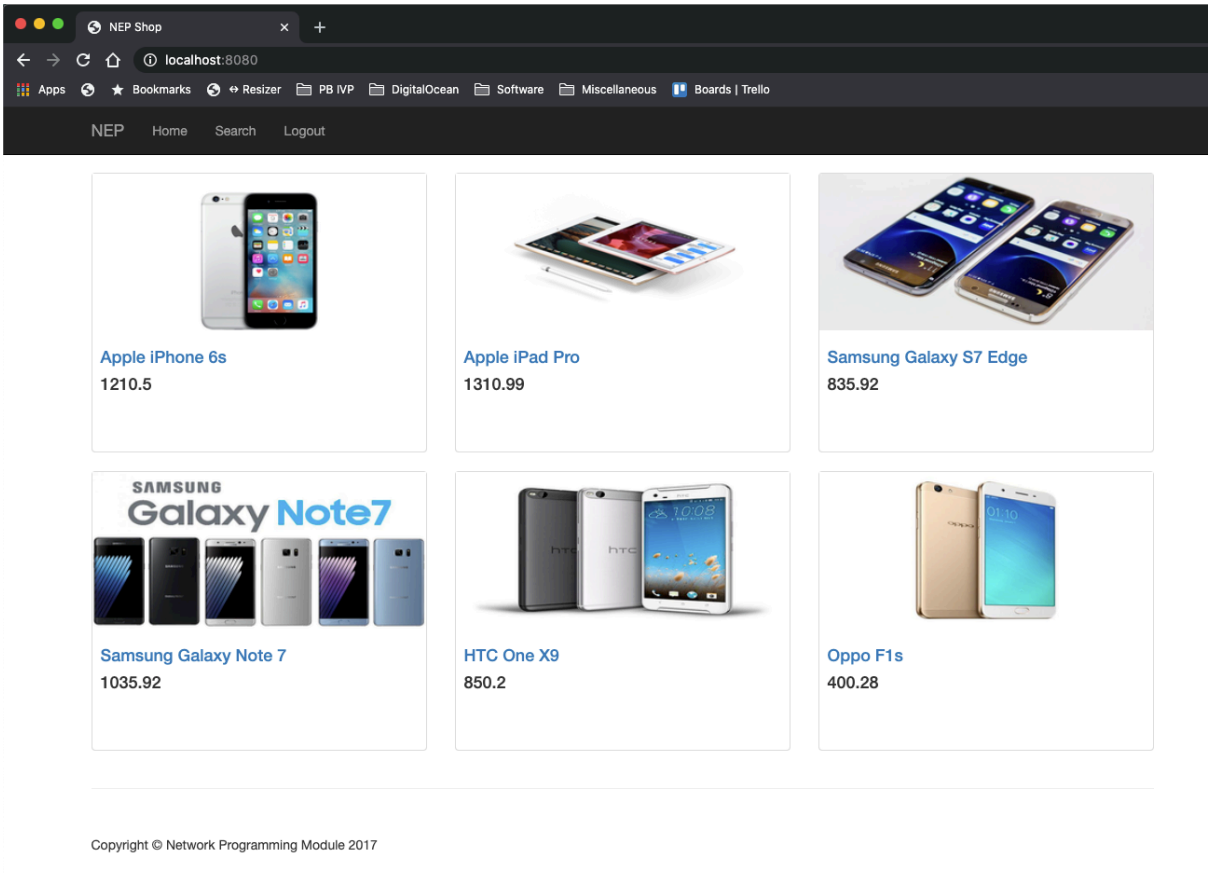
```
...
                <div class="row">
                    <!-- TODO -->
                    <!-- Slide #44 Example List of Data to Thymeleaf
Templates -->
                    <div class="col-sm-4 col-lg-4 col-md-4">
                        <div class="thumbnail">
                            <!-- TODO -->
                            <!-- Slide #32 Example Data to Thymeleaf
Templates -->
                            <img th:src="${ product.file }" alt="">
                            <div class="caption">
                                <!-- TODO -->
                                <!-- Slide #32 Example Data to Thymeleaf
Templates -->
                                <!-- Slide #51 Example Dynamic Page (cont.)
-->
                                <a href="product.html"><h4 th:text="${
product.name }"></h4></a>
                                <!-- TODO -->
```

```
                                <!-- Slide #32 Example Data to Thymeleaf
Templates -->
                                <h4 th:text="${ product.price }"></h4>
                            </div>
                        </div>
                    </div>
                </div>
...
```

10. Now lets add a for-each loop Thymeleaf syntax for it to got through the list of products and generate it in the page. We are going to add it below the comment `<!-- Slide #44 Example List of Data to Thymeleaf Templates -->`. For `th:each="product : ${ products }"`, it is the same as Java for each loop `for(Product product : products){}`

```
...
                <div class="row">
                    <!-- TODO -->
                    <!-- Slide #44 Example List of Data to Thymeleaf
Templates -->
                    <div class="col-sm-4 col-lg-4 col-md-4"
th:each="product : ${ products }">
                        <div class="thumbnail">
                            <!-- TODO -->
                            <!-- Slide #32 Example Data to Thymeleaf
Templates -->
                            <img th:src="${ product.file }" alt="">
                            <div class="caption">
                                <!-- TODO -->
                                <!-- Slide #32 Example Data to Thymeleaf
Templates -->
                                <!-- Slide #51 Example Dynamic Page (cont.)
-->
                                <a href="product.html"><h4 th:text="${
product.name }"></h4></a>
                                <!-- TODO -->
                                <!-- Slide #32 Example Data to Thymeleaf
Templates -->
                                <h4 th:text="${ product.price }"></h4>
                            </div>
                        </div>
                    </div>
                </div>
...
```

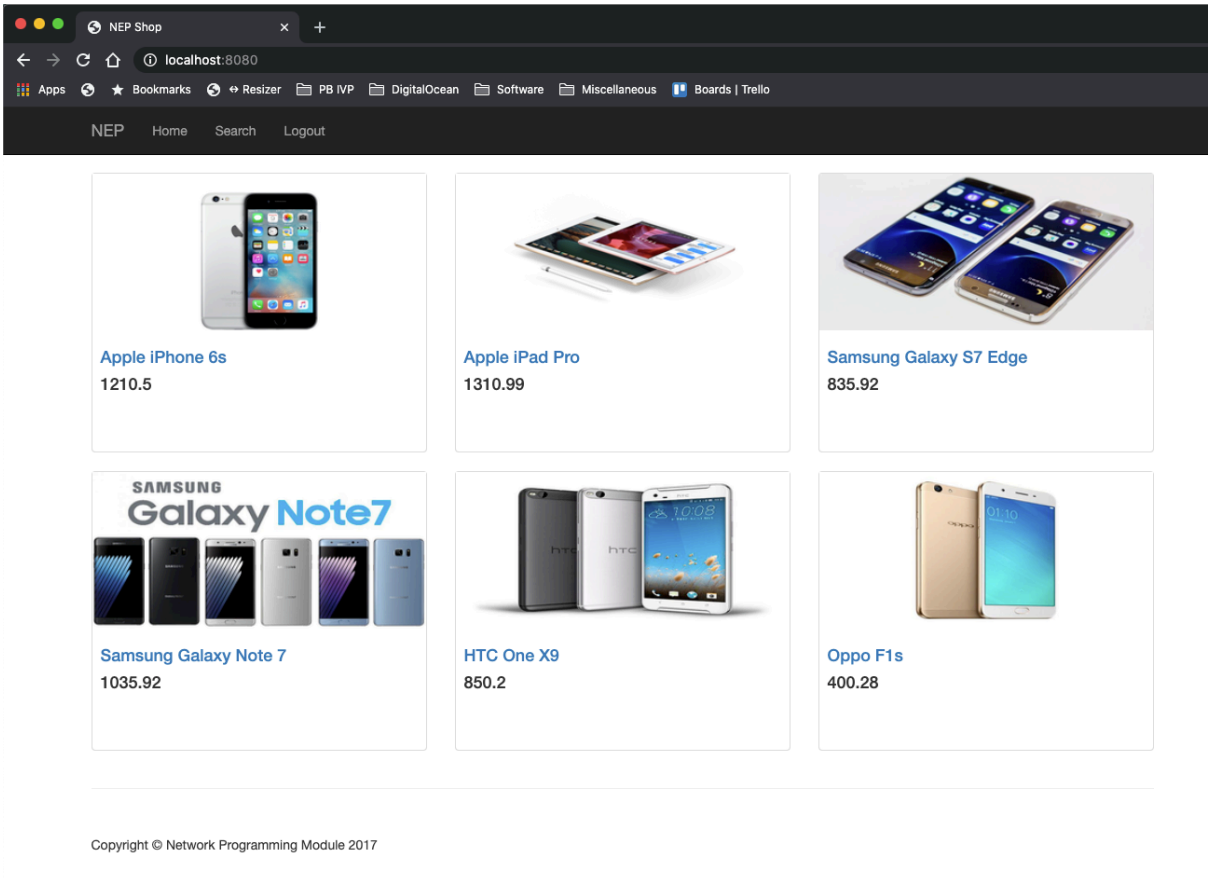
11. Go to URI `/`. It should display all the products in the list.



> # ⚠ Milestone:
>
> Commit the changes made to the `src` folder with message "Lesson 1.7 - Data Repository".
>
> Hint: Add the files to the staging area first, `git add <files>` and then proceed to commit, `git commit -m "Lesson 1.7 - Data Repository"`.

Lesson 1.8 - Dynamic Page

1. Now that you have implemented a list of data repository to your page. But what if you want to allow user to click one of the data and go to a page where it shows the clicked data details only. One way to do it is, to have a dynamic Uniform Resource Identifier (URI) depending on the data clicked. Example:
    - A home page `/` where it has a list of products.
    - When the user clicked on lets say Oppo F1s, it will go to URI `/product/Oppo F1s`.



2. Before we create and handle dynamic URI, create a method called `findByName` to retrieve a single data from the data repository based on what field. Retrieve a Product object from Project repository based on the name of the product.

```
...
public class ProductRepository {
  ...

  public static List<Product> getAllProducts() {
    return ALL_PRODUCTS;
  }

  public Product findByName(String name) {
    for(Product product : ALL_PRODUCTS) {
      if(product.getName().equals(name)) {
        return product;
      }
    }
    return null;
  }

}
```

3. Lets create a dynamic URI. In `index.html`, under comment `<!-- Slide #51 Example Dynamic Page (cont.) -->`. Lets modify the hyperlink tag `<a>` attribute `href`. Concatenated URI `/product/` with the name of the product. It will generate a hyperlink for each product using the product name.

```
...
                <div class="row">
                    <!-- TODO -->
                    <!-- Slide #44 Example List of Data to Thymeleaf
Templates -->
                    <div class="col-sm-4 col-lg-4 col-md-4"
th:each="product : ${ products }">
                        <div class="thumbnail">
                            <!-- TODO -->
                            <!-- Slide #32 Example Data to Thymeleaf
Templates -->
                            <img th:src="${ product.file }" alt="">
                            <div class="caption">
                                <!-- TODO -->
                                <!-- Slide #32 Example Data to Thymeleaf
Templates -->
                                <!-- Slide #51 Example Dynamic Page (cont.)
-->
                                <a th:href="@{'/product/' +
${product.name}}"><h4 th:text="${ product.name }"></h4></a>
                                <!-- TODO -->
                                <!-- Slide #32 Example Data to Thymeleaf
Templates -->
                                <h4 th:text="${ product.price }"></h4>
                            </div>
                        </div>
                    </div>
                </div>
...
```

4. Let go to URI `/`. The link for each product should include the name of the respective product.
    - If I hover my cursor on the name of the product, in this case product Apple iPhone 6s, the URI should be `/product/Apple iPhone 6s`.
    - If I hover my cursor on Oppo F1s, the URI should be `/product/Oppo F1s`.
    - I'm using Google Chrome browser, the link is shown at the bottom left.

NEP Shop

localhost:8080

Apps Bookmarks Resizer PB IVP DigitalOcean Software Miscellaneous Boards | Trello

NEP Home Search Logout

Apple iPhone 6s

1210.5

Apple iPad Pro

1310.99

Samsung Galaxy S7 Edge

835.92

Samsung Galaxy Note 7

1035.92

HTC One X9

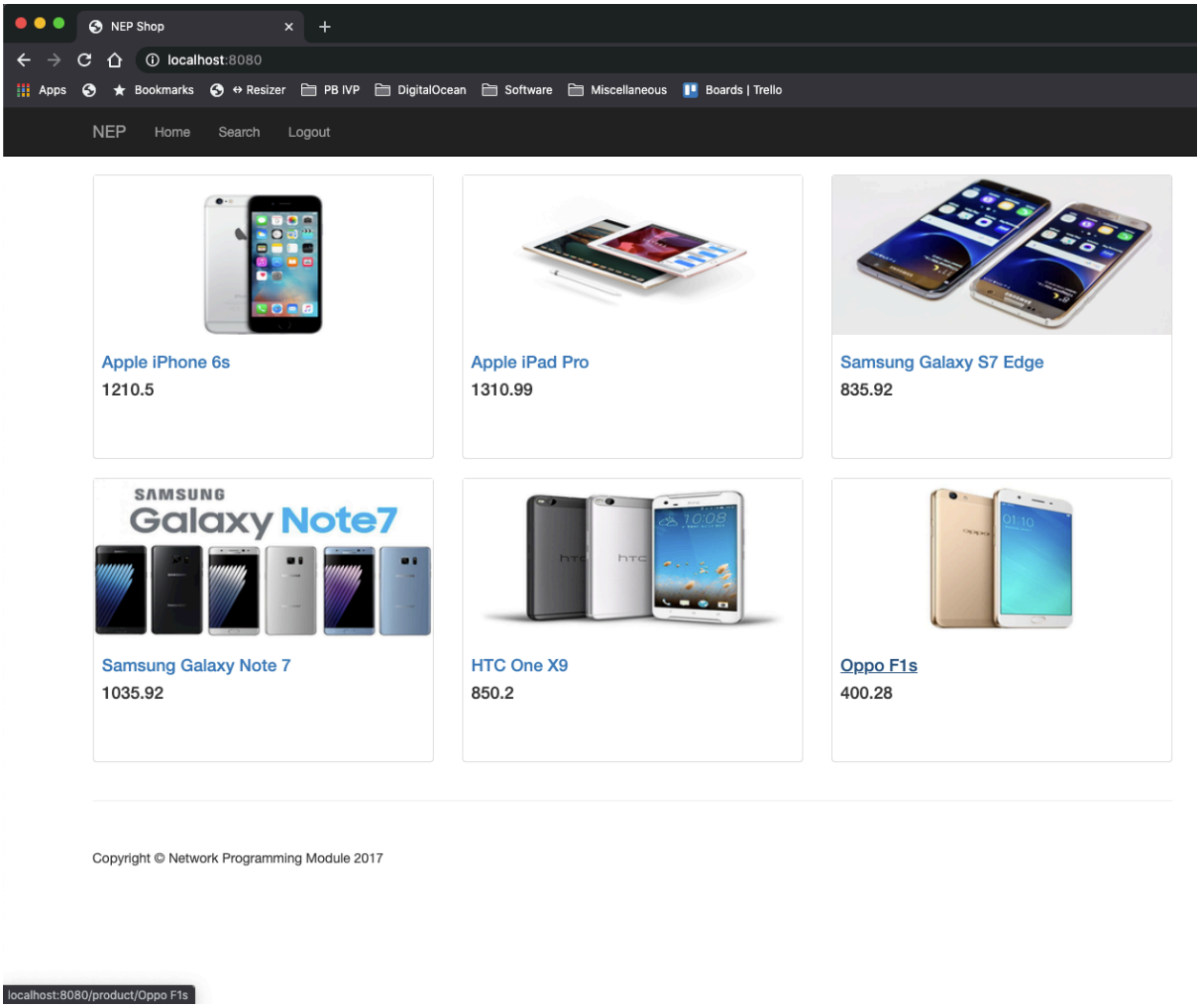
850.2

Oppo F1s

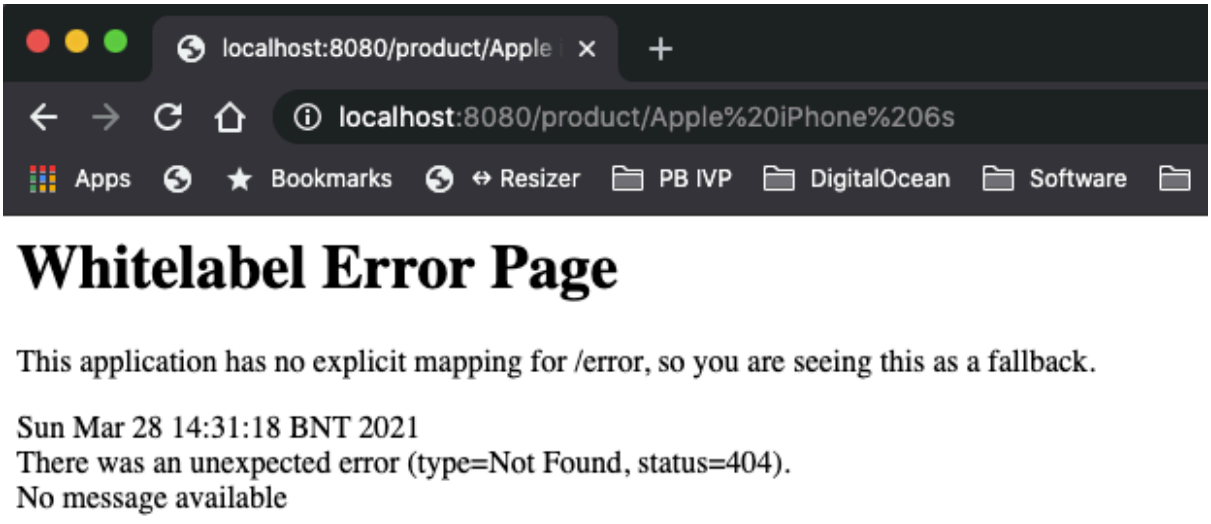
400.28

Copyright © Network Programming Module 2017

localhost:8080/product/Apple iPhone 6s



5. If we click the hyperlink on one of the product. It will produce an error. This is because we not yet handle the URI.



6. For now add another `@RequestMapping` for URI `/product/` in `HomeController` class. When the URI is requested, the application should response with `product.html`.

```
...
public class HomeController {
  ...

  @RequestMapping(value="/product/")
  public String product() {
    return "product";
  }

}
```

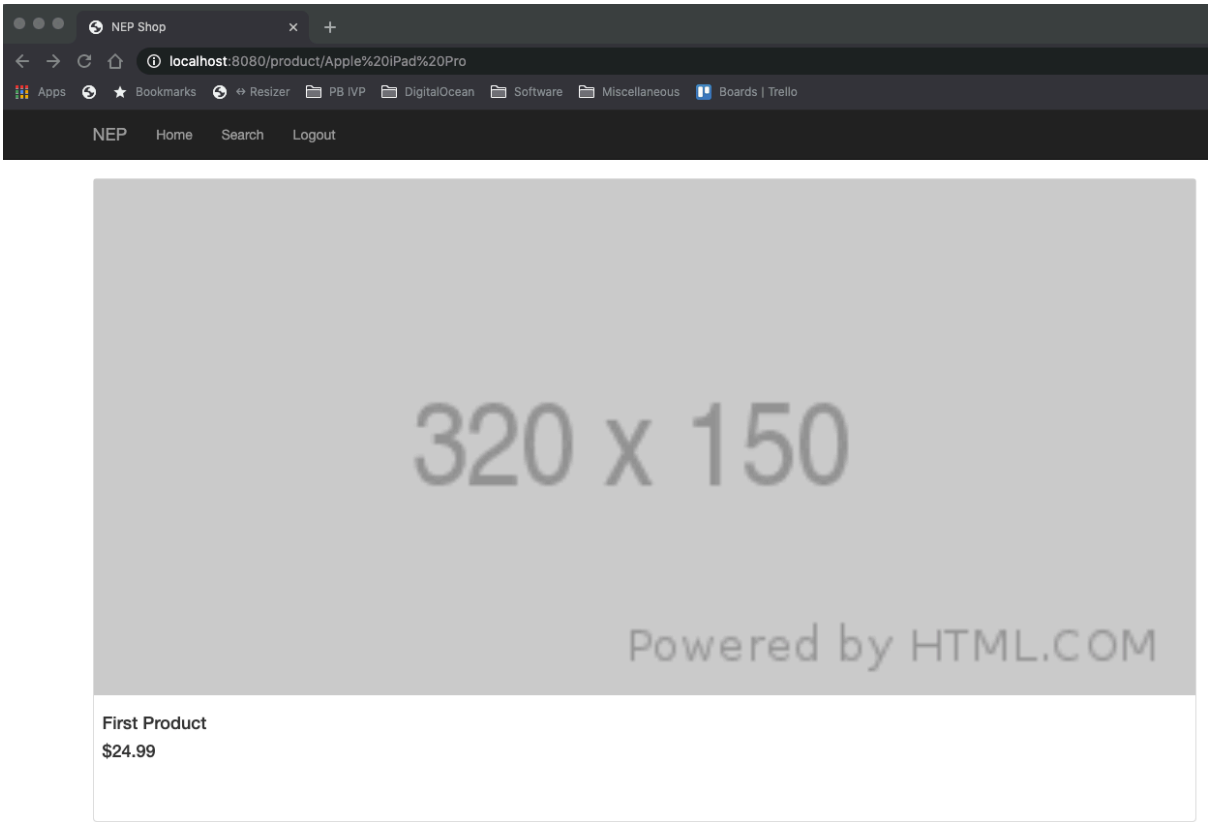
7. Its not practical to implement one `@RequestMapping` for each product. So instead we are going to use a `PathVariable`. In `@RequestMapping` annotation, add a path variable in the URI. In this case, it will be `@RequestMapping(value="/product/{name}")`

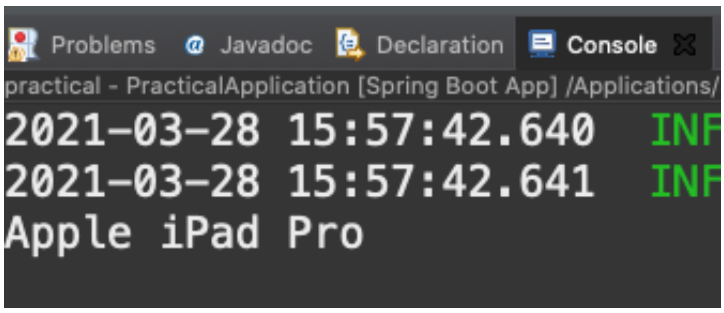
```
...
public class HomeController {
  ...

  @RequestMapping(value="/product/{name}")
  public String product() {
    return "product";
  }

}
```

8. To capture the value passed to the path variable. We need a parameter variable with the same name in method with annotation `@PathVariable`. This value can be use in the method. For now lets add `System.out.println(name);` statement.

```
...
import org.springframework.web.bind.annotation.PathVariable;
...
public class HomeController {
  ...

  @RequestMapping(value="/product/{name}")
  public String product(@PathVariable String name) {
    System.out.println(name);
    return "product";
  }

}
```

9. Go to URI `/product/Apple iPad Pro`. It will show `product.html` and print into console `Apple iPad Pro`. So if the URI is `/product/HelloWorld`, this will print into console `HelloWorld`.





10. Now we can use that path variable to search for the product. Lets replace the `System.out.println(name);` statement to `Product product = productRepository.findByName(name);`.

```
...
public class HomeController {
  ...

  @RequestMapping(value="/product/{name}")
  public String product(@PathVariable String name) {
    Product product = productRepository.findByName(name);
    return "product";
  }

}
```

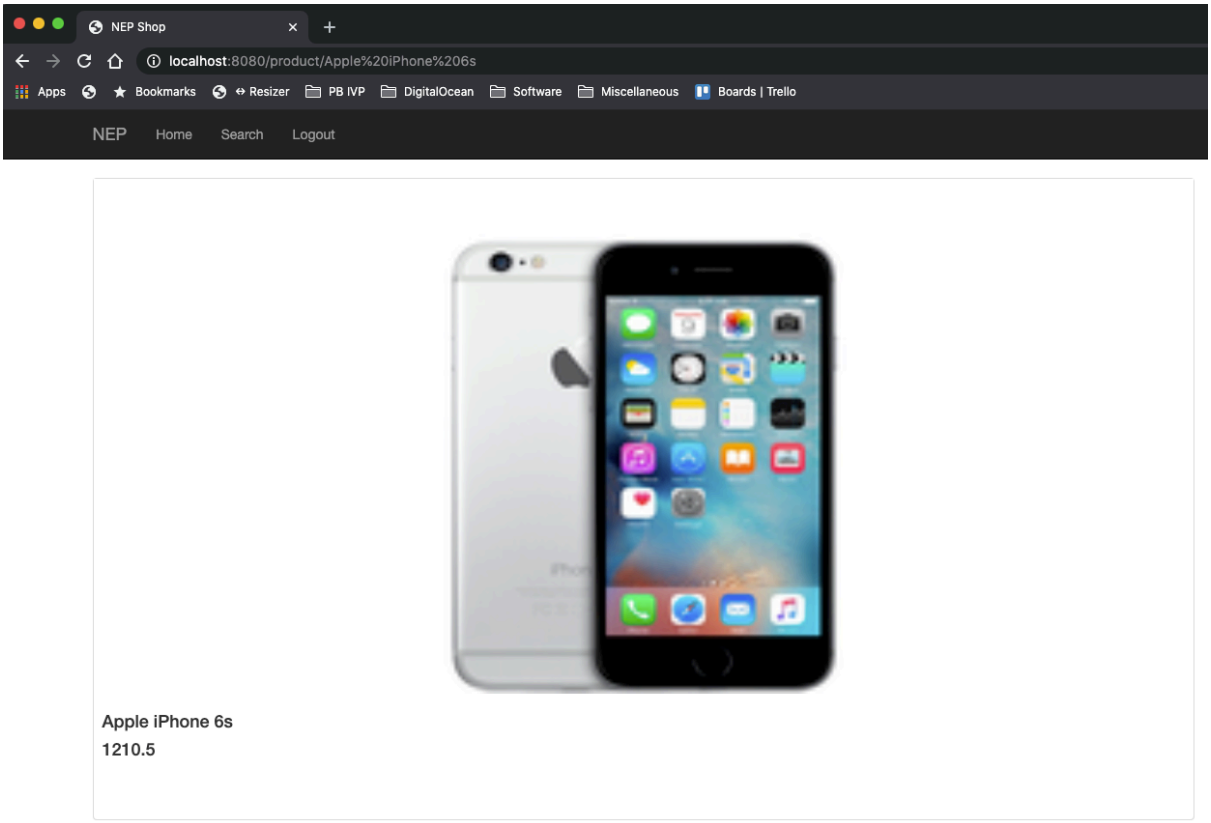
11. Then we need to pass the product to `product.html`.

```
...
public class HomeController {
  ...

  @RequestMapping(value="/product/{name}")
  public String product(@PathVariable String name, ModelMap modelMap) {
    Product product = productRepository.findByName(name);
    modelMap.put("product", product);
    return "product";
  }

}
```

12. Finally lets modify `product.html` to show the product passed from controller. We will do the same like what we did in **Lesson 1.6 - Spring Model #10**

```
...
                        <div class="thumbnail">
                            <img th:src="${ product.file }" alt="">
                            <div class="caption">
                                <h4 th:text="${ product.name }"></h4>
                                <h4 th:text="${ product.price }"></h4>
                            </div>
                        </div>
...
```

13. Try to go to URI `/product/Apple iPhone 6s`. It should show the details of the product.



> ## ⚠️ Milestone:
>
> Commit the changes made to the `src` folder with message "Lesson 1.8 - Dynamic Page".
>
> Hint: Add the files to the staging area first, `git add <files>` and then proceed to commit, `git commit -m "Lesson 1.8 - Dynamic Page"`.

Lesson 1.9 - HTTP Request

1. In some websites, they provide some forms that user can input data into it. So far, we've simply requests to view certain resources where no additional data is supplied on the request. There are two methods to supply additional data on the request:
    - **GET** Method: Retrieve whatever information (in the form of an entity) is identified by the Request-URI.
    - **POST** Method: Used to request that the origin server accept the entity enclosed in the request as a new subordinate of the resource identified by the Request-URI in the Request-Line.

    You can find in `search.html` a `form` tag that has a method GET.

```
...
                    <form class="form-horizontal" action="search.html"
method="get">
                        <div class="col-sm-2 col-lg-2 col-md-2">
                            <button class="btn btn-default btn-block"
type="submit">Search</button>
                        </div>
                        <div class="col-sm-10 col-lg-10 col-md-10">
                            <div class="hidden-sm hidden-lg hidden-
md"> </div>
                            <input class="form-control" type="text"
placeholder="Search For Product" name="productName" />
                        </div>
                    </form>
...
```

2. In this case, this form will be use to search for products that contain a name. Lets add a method in the ProductRepository called `findContainName()`.

```
...
public class ProductRepository {

  ...

  public List<Product> findContainName(String name) {

  }

}
```

3. In the `findContainName()`, we are going to create a new List for storing products that contain the name. Then go through each element from `ALL_PRODUCTS`, if the product contain the name, add to the new List. Once done, return the List.

```
...
public class ProductRepository {

  ...

  public List<Product> findContainName(String name) {
    List<Product> products = new ArrayList<Product>();
    for(Product product : ALL_PRODUCTS) {
      if(product.getName().toLowerCase().contains(name.toLowerCase())) {
        products.add(product);
```

```
      }
    }
    return products;
  }

}
```

4. In `search.html`, set the `action` for the `form` tag to `/search` URI. That means when the button `Search` is clicked, it will trigger HTTP request to the action URI `/search`.

```
...
                    <form class="form-horizontal" action="/search"
method="get">
                        <div class="col-sm-2 col-lg-2 col-md-2">
                            <button class="btn btn-default btn-block"
type="submit">Search</button>
                        </div>
                        <div class="col-sm-10 col-lg-10 col-md-10">
                            <div class="hidden-sm hidden-lg hidden-
md"> </div>
                            <input class="form-control" type="text"
placeholder="Search For Product" name="productName" />
                        </div>
                    </form>
...
```

5. We need to handle this request in our `HomeController` and return `search.html`.

```
...
public class HomeController {

  ...

  @RequestMapping(value="/search")
  public String search() {
    return "search";
  }

}
```

6. Then we need to capture the data sent through the HTTP request to the method `search`. That means any form elements that data will be send through the HTTP request, example, `input`, `select` tag. In this case, it contains one `input` tag with name `productName`.

- `<input class="form-control" type="text" placeholder="Search For Product" name="productName" />`, keep note the name attribute.
- In the method, we declare a parameter variable called `productName` which should be exactly the same as the `input` tag name. Assign annotation `@RequestParam(required=false)` to the parameter variable. The `required` means that when the URI `/search` is requested. Is the data `productName` must be included with the request. `true` means it must be included, `false` means it is optional.

```
...
import org.springframework.web.bind.annotation.RequestParam;
...
public class HomeController {

  ...

  @RequestMapping(value="/search")
  public String search(@RequestParam(required=false) String productName) {
    return "search";
  }

}
```

7. Since we set `required=false`, that means what happens if `productName` is not included. We need to handle this as well. That means if `productName` is not included, it is `null`.

```
...
public class HomeController {

  ...

  @RequestMapping(value="/search")
  public String search(@RequestParam(required=false) String productName) {
    if(productName == null) {

    } else {

    }
    return "search";
  }

}
```

8. Lets decide that if `productName` is `null`. We will send all products to `ModelMap`. Else we will send the list products that contains the `productName`.

```
...
public class HomeController {

  ...

  @RequestMapping(value="/search")
  public String search(@RequestParam(required=false) String productName,
ModelMap modelMap) {
    List<Product> products = null;
    if(productName == null) {
      products = productRepository.getAllProducts();
    } else {
      products = productRepository.findContainName(productName);
    }
    modelMap.put("products", products);
    return "search";
  }

}
```

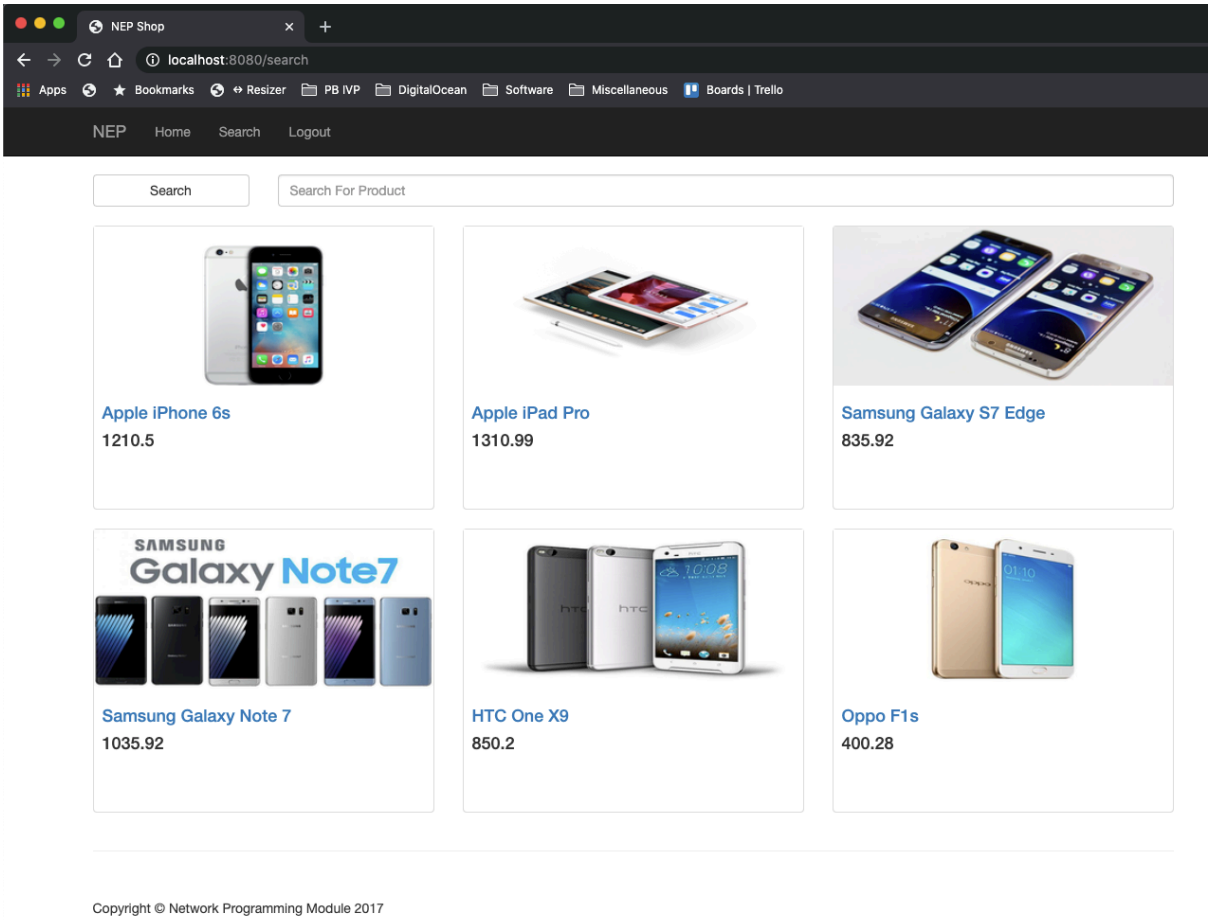
9. Now lets make sure `search.html` can handle to show the list of products. **Remove** these first.

```
...
                    <div class="col-sm-4 col-lg-4 col-md-4">
                        <div class="thumbnail">
                            <img src="http://placehold.it/360x170" alt="">
                            <div class="caption">
                                <a href="product.html"><h4>Second
Product</h4></a>
                                <h4>$14.99</h4>
                            </div>
                        </div>
                    </div>
                    <div class="col-sm-4 col-lg-4 col-md-4">
                        <div class="thumbnail">
                            <img src="http://placehold.it/360x170" alt="">
                            <div class="caption">
                                <a href="product.html"><h4>Third
Product</h4></a>
                                <h4>$54.99</h4>
                            </div>
                        </div>
                    </div>
...
```

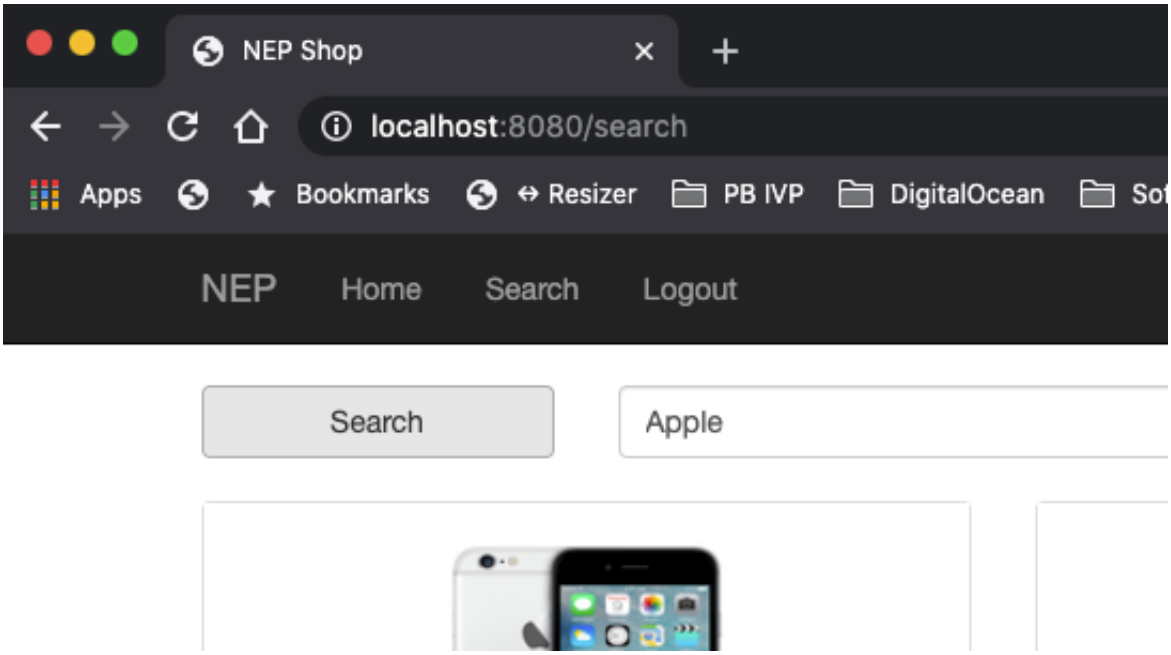
10. Then we will do the same as what we did previously but in `search.html`.

```
...
                    </form>
                </div>
                <div class="row">
                    <div class="col-sm-4 col-lg-4 col-md-4"
th:each="product : ${ products }">
                        <div class="thumbnail">
                            <img th:src="${ product.file }" alt="">
                            <div class="caption">
                                <a th:href="@{'/product/' +
${product.name}}"><h4 th:text="${ product.name }"></h4></a>
                                <h4 th:text="${ product.price }"></h4>
                            </div>
                        </div>
                    </div>
                </div>
...
```

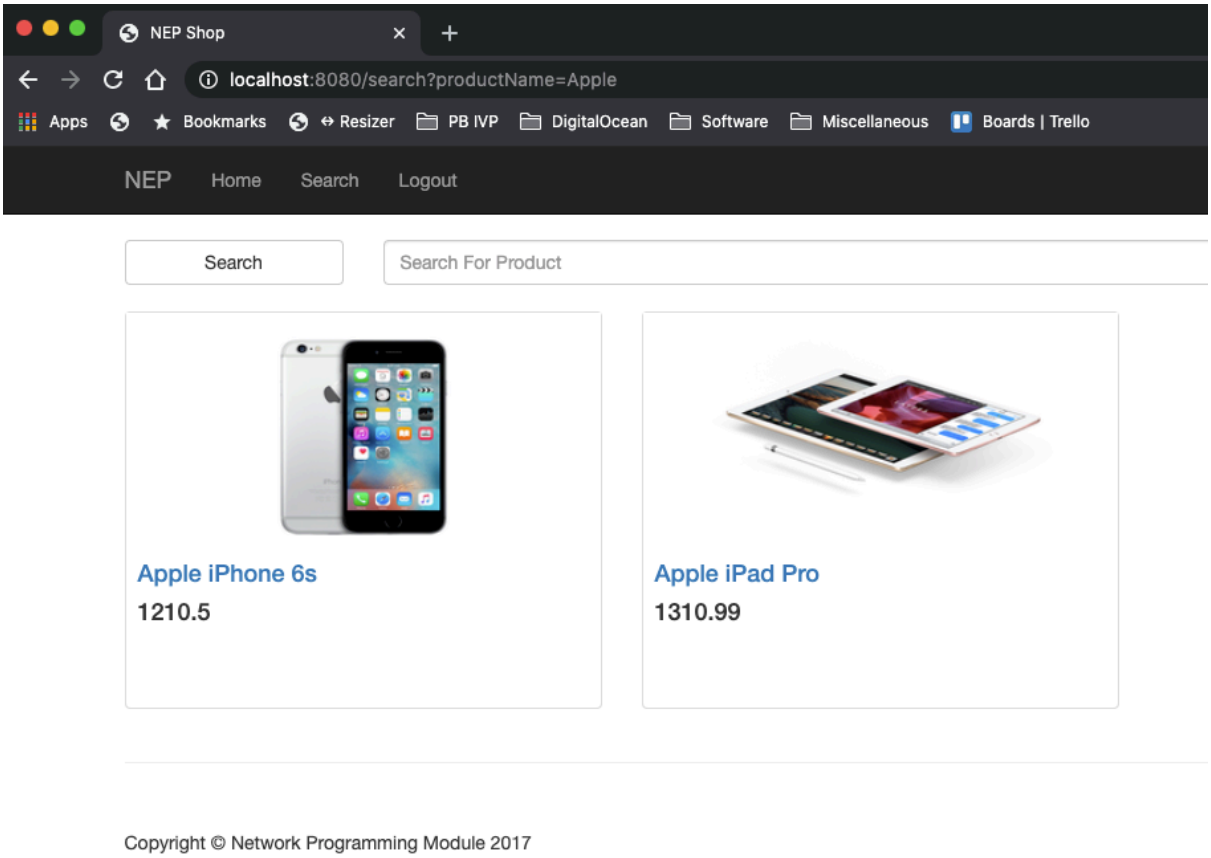
11. Go to URI `/search`. It should display all products since `productName` is not included.



12. Search for word `Apple` and click the `Search` button.



13. It should show only products with name `Apple`.



NOTE: notice that it added `?productName=Apple` to the URI `/search`. This happens if you choose the method `GET`. If you want the data to be included in HTTP header instead of URI. Set the form method to `POST`.

> # ⚠️ Milestone:
>
> Commit the changes made to the `src` folder with message "Lesson 1.9 - HTTP Request".
>
> Hint: Add the files to the staging area first, `git add <files>` and then proceed to commit, `git commit -m "Lesson 1.9 - HTTP Request"`.