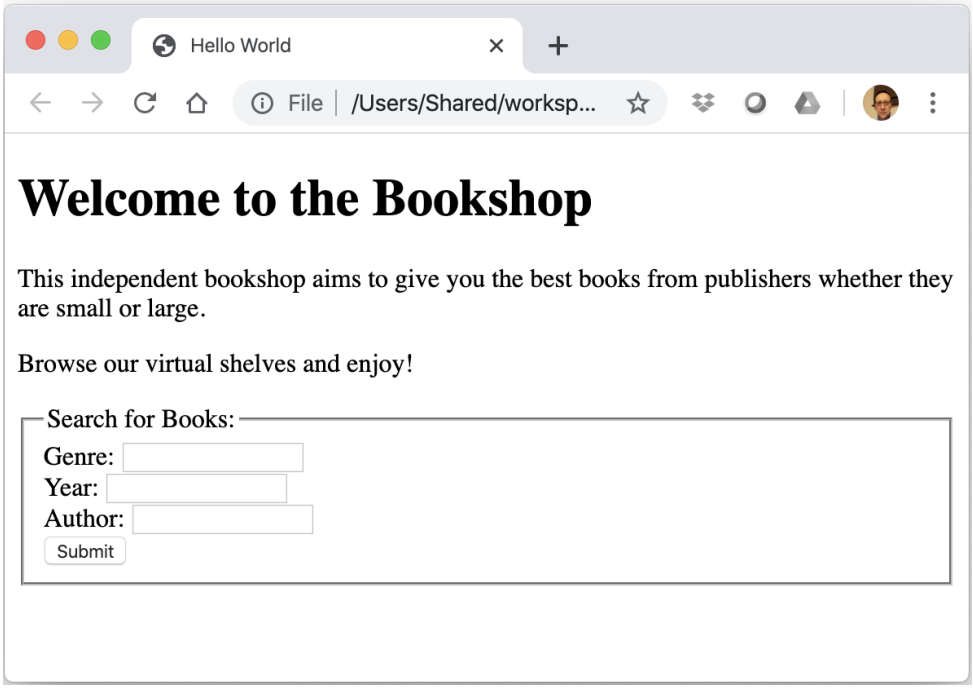
Lab: Bookshop Web Page

The aim of this lab is to create a web page using HTML.

In this lab we will not worry about styling of the content; we are just using HTML to provide the structure of the web page.

The page you should create is illustrated below:



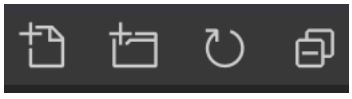
This page contains

- A heading
- A couple of paragraphs
- A from with three field in it.

Create a file called index.html within your IDE and store it within the Vagrant shared directory under the HTML directory.

You might wish to create a folder to hold each of the labs we work on so that you can see your own progression or reference back to earlier versions of labs.

To do this use the new folder icon next to the project directory within Visual Studio code. This is the second icon along:



You can then use the first icon to create a file within the new folder.

Within the file you will need to

1. Create the top level body element
2. Create the header
3. Create the body
4. Add the content to the body

Lab: Styled Bookshop Web Page

The aim of this lab is to style the web page created in the last lab.

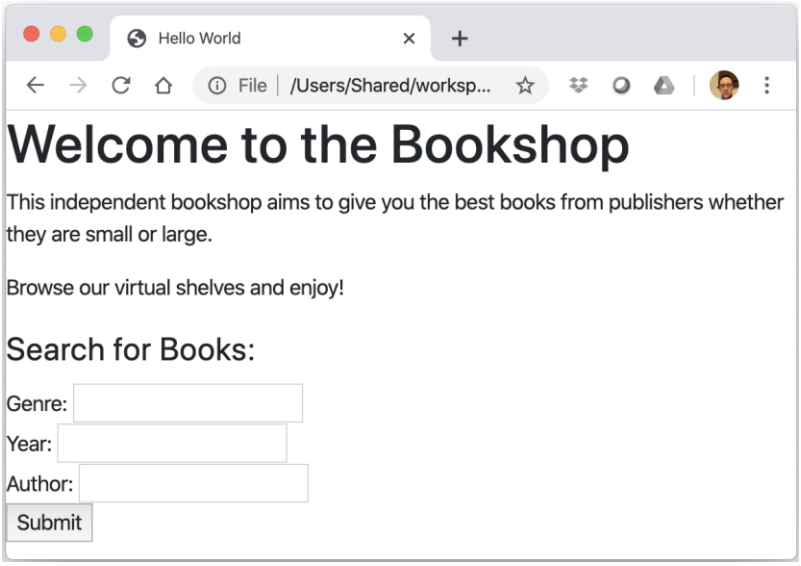
In this lab we will be using CSS to provide style information for the web page we previously created.

If required, you can change the HTML to add div (block) elements etc. as required.

The lab is into two parts.

Part 1: Add the Bootstrap style information
In this part you should add a link to the Bootstrap CSS style library to the head of your web page.

The layout of the page should now alter subtly as shown below:

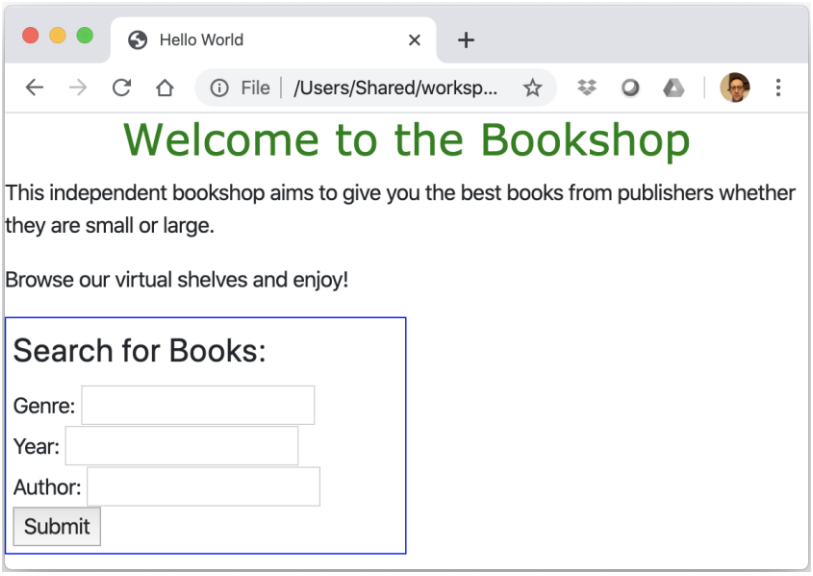


This is the result of adding bootstrap. Notice the way in which the submit button is shaded. Also notice that the border around the search from has disappeared. Are there any other changes?

Part 2: Create your own style sheet
Create your own style sheet that will build in the default in Bootstrap. Create a file within the HTML directory of the shared Vagrant directory.

In this style sheet

1. Change the colour of the heading of the page to green
2. Centre the heading
3. Add a border back into the from and make it Blue
4. Make the from border only 50% of the width of the page


Hello World
File | /Users/Shared/worksp...
Welcome to the Bookshop
This independent bookshop aims to give you the best books from publishers whether they are small or large.
Browse our virtual shelves and enjoy!
Search for Books:
Genre:
Year:
Author:
Submit

Lab: Set up Vagrant

The aim of this lab is to set up the Vagrant environment on your own machine.

To do this follow the steps in the Vagrant slides.

Lab: Selenium

The aim of this lab is to set up a set of Selenium tests.

Step 1: Create a new module in IntelliJ

The new module should again be a Maven managed module.

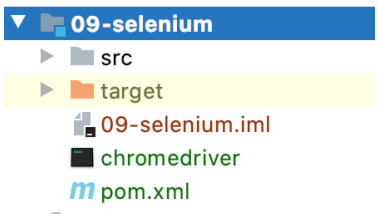
The contents of the POM file should this time be:

```
<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.jjh</groupId>
  <artifactId>sample</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>war</packaging>

  <name>sample Maven Webapp</name>
  <url>http://www.example.com</url>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>11</maven.compiler.source>
    <maven.compiler.target>11</maven.compiler.target>
    <junit-version>5.7.0</junit-version>
    <junit-platform-runner-version>1.7.0</junit-platform-runner-version>
     <selenium-java-version>3.141.59</selenium-java-version>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter-api</artifactId>
      <version>${junit-version}</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.junit.platform</groupId>
      <artifactId>junit-platform-runner</artifactId>
      <version>${junit-platform-runner-version}</version>
      <scope>test</scope>
```

```
    </dependency>
    <dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter-engine</artifactId>
      <version>${junit-version}</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter-params</artifactId>
      <version>${junit-version}</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.seleniumhq.selenium</groupId>
      <artifactId>selenium-java</artifactId>
      <version>${selenium-java-version}</version>
      <scope>test</scope>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <artifactId>maven-surefire-plugin</artifactId>
        <version>2.22.2</version>
      </plugin>
      <plugin>
        <artifactId>maven-failsafe-plugin</artifactId>
        <version>2.22.2</version>
      </plugin>
    </plugins>
  </build>
</project>
```

Step 2: Add the Chrome Driver to your project

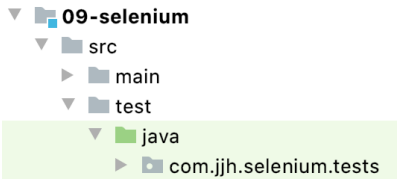
The Chrome Driver has already been downloaded for you to your local machine. However the easiest thing to do is to add it to your local project. Copy the chromedriver.exe to the root of your module, so that you have:



Step 3: Add a Selenium Configuration Class

We will add a class to represent configuration information for Selenium that can be used by any test class. This code could be placed anywhere however we will place it in a separate class called SeleniumConfig so that it can easily be reused.

However, this code should not be part of the production code base and will this be placed under the src/test/java path. To do this open the src/test/java nodes within the project view and add a new package (for example called com.jjh.selenium.tests). For example:



Note that the green indicates that this package is part of the test package and not part of the prouction code.

In this package add the SeleniumConfig class. The contents of this class is given below:

```
package com.jjh.selenium.tests;

import java.util.concurrent.TimeUnit;

import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;

public class SeleniumConfig {
  private WebDriver driver;

  public SeleniumConfig() {
    driver = new ChromeDriver();
    driver.manage().deleteAllCookies();
    driver.manage().timeouts().pageLoadTimeout(40, TimeUnit.SECONDS);
    driver.manage().timeouts().implicitlyWait(30, TimeUnit.SECONDS);
  }

  static {
    // Setting system properties of ChromeDriver
    System.setProperty("webdriver.chrome.driver", "chromedriver.exe");
  }

  public void close() {
    driver.close();
  }

  public WebDriver getDriver() {
    return driver;
  }
}
```

You can adjust the timeouts as you see fit.

Step 4: Writing the Test

We will write a simple Selenium test that will validate that the title of a specific web page is as expected. The web page will be the web page you create dint he previous lab and run through Apache web server running within the Linux VM managed by Vagrant.

The test class should be called Test<something> or <something>Test – in my case I have called it WelcomeWebPageTest.

The test is a JUnit test that will set up the Selenium configuration and then use the driver supplied by the SeleniumConfig to access the web page being displayed.

The definition of the JUnit test is given below:

```
package com.jjh.selenium.tests;

import static org.junit.jupiter.api.Assertions.assertEquals;

import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.Test;

import org.openqa.selenium.By;

/**
 * This is a JUnit Selenium test.
 */
public class WelcomeWebPageTest {

   private static SeleniumConfig config;

   @BeforeAll
   public static void setUp() {
      config = new SeleniumConfig();
      config.getDriver().get("http://localhost:8080/index.html");
   }

   @AfterAll
   public static void tearDown() {
      config.close();
   }

   @Test
   public void check_page_title_is_Hello_World() {
      String actualTitle = config.getDriver().getTitle();
      assertEquals("Hello World", actualTitle);
   }

   @Test
   public void check_heading_is_Welcome() {
      String heading =
            config.getDriver()
                  .findElement(By.className("heading"))
                  .getText();
```

```
        assertEquals("Welcome", heading);
    }

}
```

Step 5: Running the Web Site

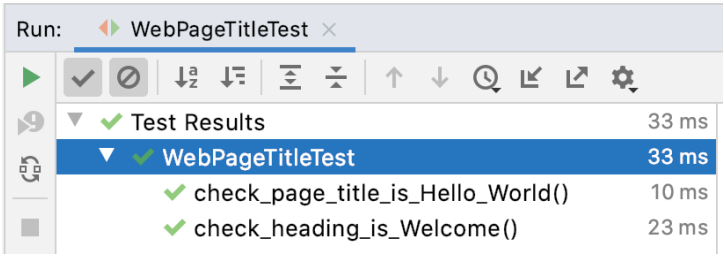
Make sure Vagrant is up and running. You can start up Vagrant using

> vagrant up

Step 6: Now run the Selenium Test

Select the text class in the Project view within the test directory structure and from the right mouse menu select to run the test.

If your tests passed, you should now see



If one or more of your tests failed; check that the information you are testing is correct; fix either the web page or the test as appropriate.

Lab: JavaScript

The aim of this lab is to allow you to explore working with JavaScript, iteration, conditional statements and functions.

Calculate the Factorial of a number

You should write a program that can find the factorial of any given number.

For example, find the factorial of the number 5 (often written as 5!) which is 1 * 2 * 3 * 4 *5 and equals 120.

The factorial is not defined for negative numbers and the factorial of Zero is 1; that is 0! = 1.

Your program should define a number at the start of the program for which the factorial will be calculated, for example:

```
console.log("Starting factorial calculation program")
let number = 5;
console.log('Calculating the factorial for', number)
```

You should determine

1. If the number is less than Zero return with an appropriate message.
2. Check to see if the number is Zero – if it is then the answer is 1 – print this out.
3. Otherwise use a loop to generate the result and print it out.

The conditional aspect of the above program can be handled using an if statement (ideally with else if elements).

For example

```
if (number < 0) {
  console.log('Factorial is not defined for negative numbers')
} else if (number == 0) {
  console.log("0! factorial is 1");
} else {
    // write your code here
}
```

The for loop will need to loop an appropriate number of times. For example:

```
let factorial = 1;
  for (let i = 1; i <= number; i++) {
    factorial = factorial * i;
  }
  console.log(number + "! factorial is", factorial);
```

Notes: Make sure you create a function for the factorial

This should allow the factorial function to be called with different values so that the user of the function can ask for the factorial of say -1, 0, 1, 3, 5, 7 etc to be calculated, for example:

```
factorial(-1)
factorial(0)
factorial(1)
factorial(3)
factorial(5)
factorial(7)
```

The output for this from the sample solution is:

```
Starting factorial calculation program
Calculating factorial  -1
Factorial is not defined for negative numbers
Calculating factorial  0
0! factorial is 1
Calculating factorial  1
1! factorial is 1
Calculating factorial  3
3! factorial is 6
Calculating factorial  5
5! factorial is 120
Calculating factorial  7
7! factorial is 5040
```

Lab: JavaScript and the DOM

The aim of this lab is to use JavaScript to manipulate the DOM within a web page.

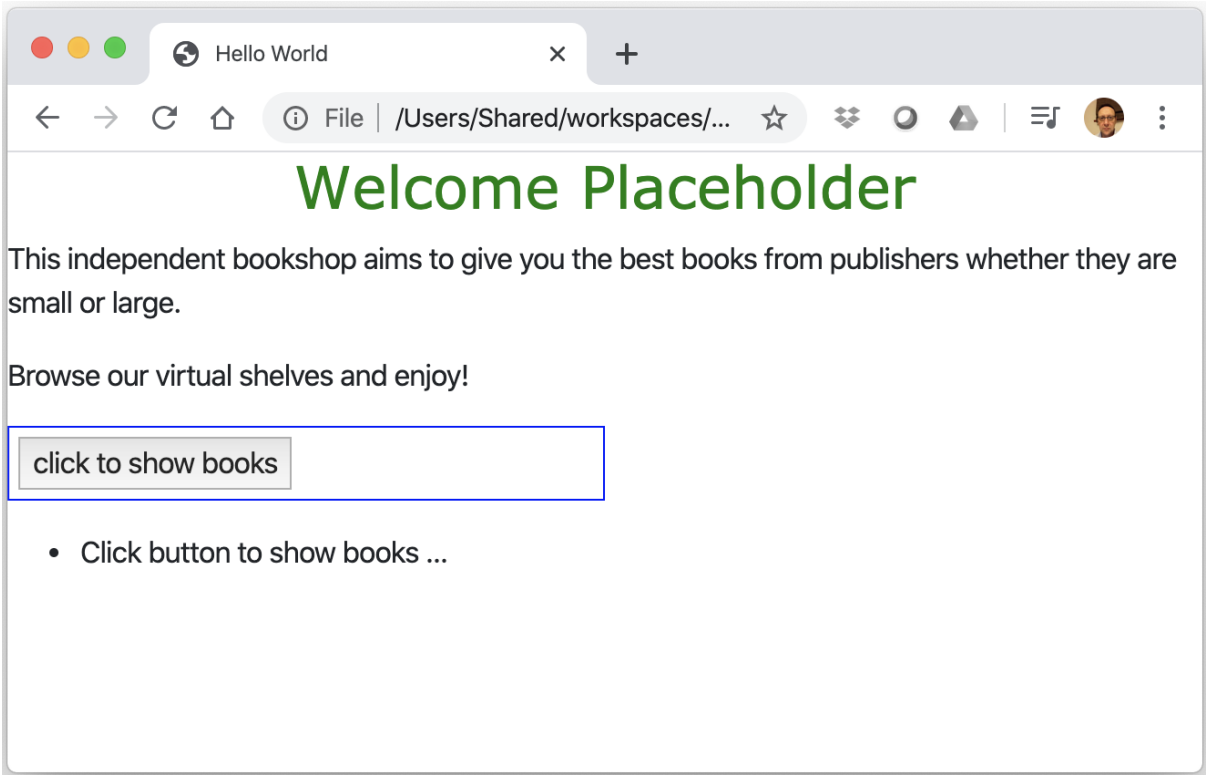
We will follow the approach of defining our JavaScript in a separate file from our HTML page (and any CSS style files).

In fact, we will build on the JavaScript we have already been writing and add functions to work with the Book classes and the array of Books you have already created.

The page will have a basic structure defined in HTML. However, we will use JavaScript to populate the title of the page and then when the user requests it we will use JavaScript to generate a list of books to be displayed within the page.

Step 1:

Create the basic HTML page to resemble this:



Note it uses Bootstrap and the CSS style file from earlier labs.

Step 2:

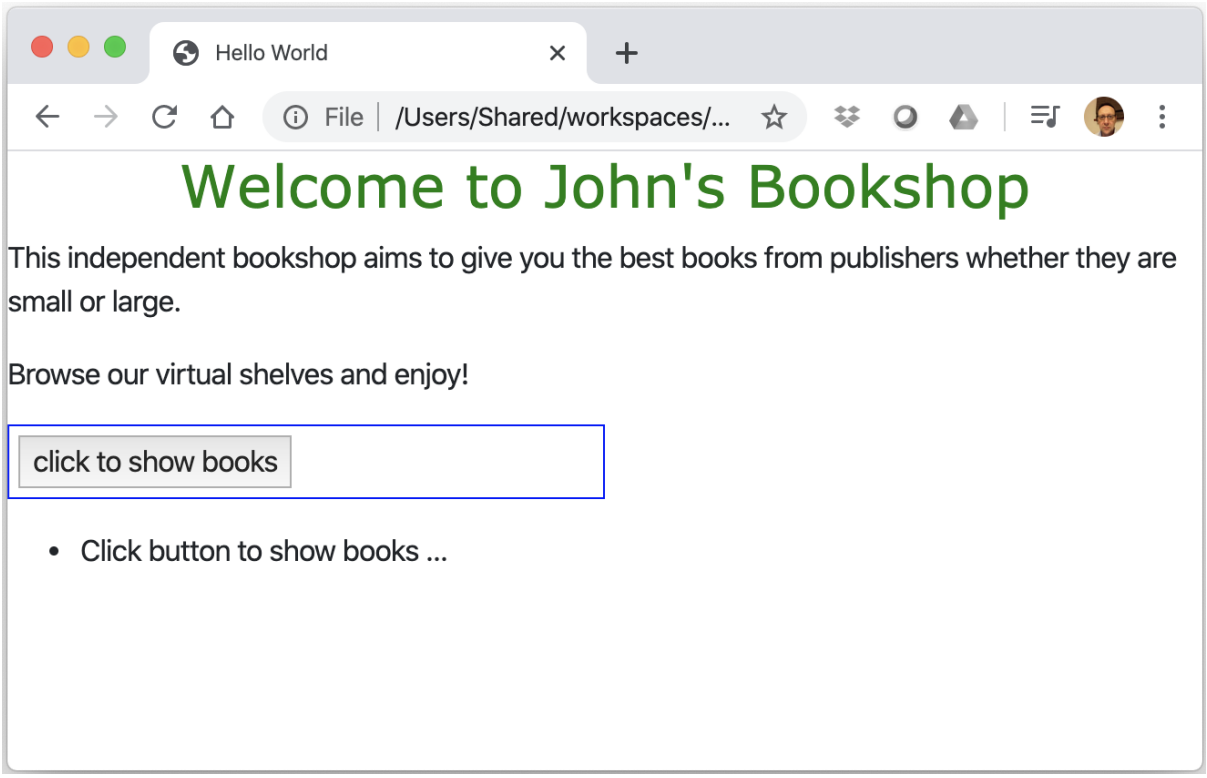
Now modify the body element such that it is defined as follows:

```
<body onload="setup()">
```

This indicates that the setup() function should be run when the body is loaded.

This function should replace the 'Welcome Placeholder' text with a Welcome message generated by the function.

So that the page now looks like this:



We will break this process down into two functions based on the Single Responsibility Principle. That is one function will provide the string to display (this means it could be loaded from a database or property file) and one function will handle updating the DOM within the web page.

We will therefore have:

```
function getWelcomeMessage() {
  return "Welcome to John's Bookshop";
}

function setup() {
  let heading = document.getElementsByTagName("h1")[0];
  heading.innerHTML = getWelcomeMessage();
}
```

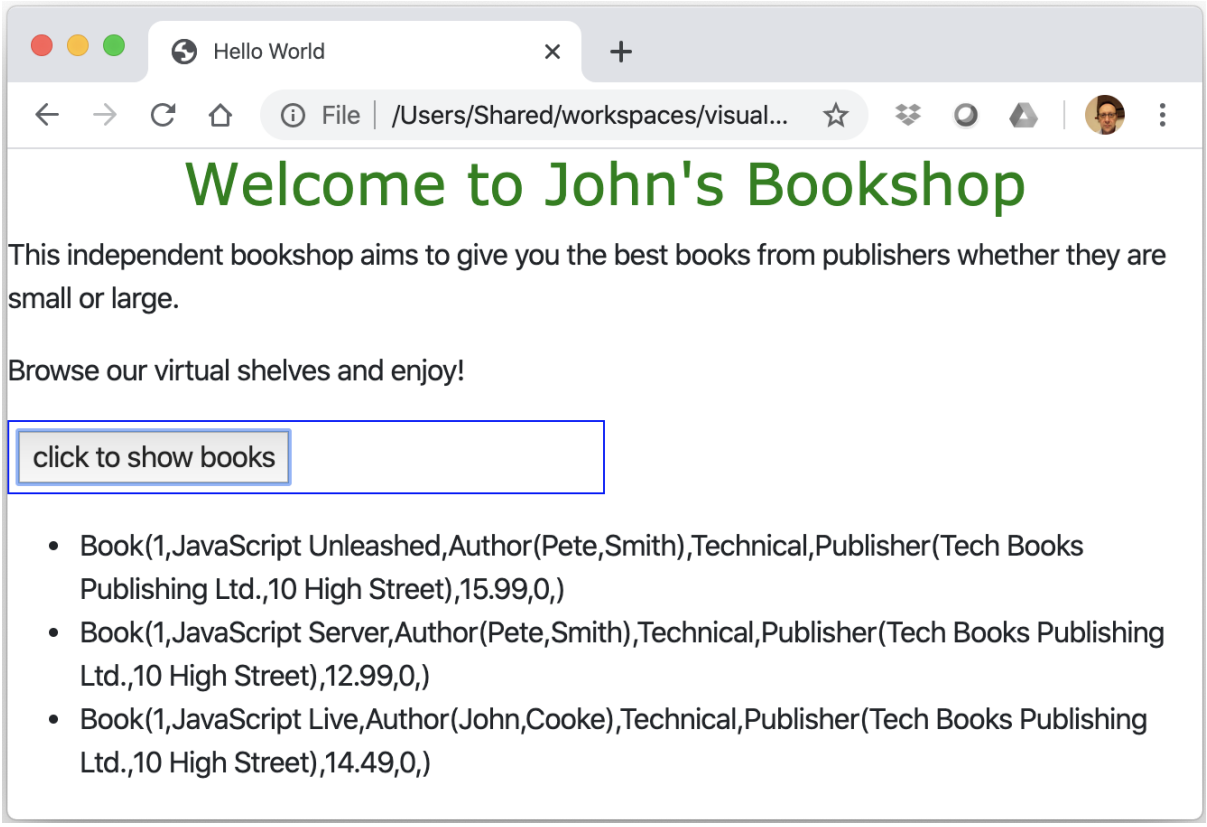
What this does

1. You will need to obtain a reference to the h1 DOM element, for example
   `let heading = document.getElementsByTagName("h1")[0];`
2. You can replace the initial string with the one from the getWelcomeMessage() function using the innerHTML property of the node.

Step 3:

Make the button work

Our final step is to make it so that when the user clicks on the button., the books within the bookstore are listed, for example:



To do this we will need specify that a function should be called when the button is clicked, for example:

```
<form>
    <input type="button" value="click to show books"
onclick="showBooks()" />
</form>
```

In this case the function is called showBooks() and will be define din the bookshop.js file along with all the other bookshop elements.

The function needs to:

1. Get hold of the existing unordered list element (ul)

```
let listItem = document.querySelector("ul");
```

2. Remove the current content

```
let range = document.createRange();
range.selectNodeContents(listItem);
range.deleteContents();
```

3. Add each book in the array of books as a new list element (li element), for example

```
for (let book of books) {
    listValue = document.createElement("li");
    listValue.textContent = book;
    listItem.appendChild(listValue);
 }
```

This assumes that you have created an array of books in your JavaScript file.

You can do this via

```
Let books = [
[1, "JavaScript Unleashed", "Pete Smith", "Technical", "Tech
Books", 15.99]
[2, "JavaScript Server", "Pete Smith", "Technical", "Tech
Books", 12.99]
[3, "JavaScript Live", "John Cooke", "Technical", "Tech
Books", 14.49]
];
```

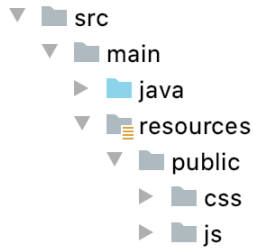
Lab: JQuery and Services

The aim of this lab is to use jQuery to obtain data from the bookshop service your created in the previous course.

You will be using IntelliJ for this Lab.

Step 1: Set up the project
To do this you first need to get the bookshop service project loaded back into IntelliJ.

Next you need to create the correct project structure under resources for the web site part of your application. This is shown below:



That is you need to:

1. Create a public directory under resources
2. Create a css directory under public
3. Create a js directory under public

Step 2: Create the web page
You should now create a HTML file under public that will be used to trigger the wite application to call the Java service.

This page can be as simple as you like. All it needs is a button that can be clicked to tigger a function that will access your back end service.

For example, you can call this file books.html or index.html etc.

The following is an example of what you might write:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">

    <script src="https://code.jquery.com/jquery-3.5.1.js"></script>
    <script
src="https://cdn.jsdelivr.net/npm/popper.js@1.16.0/dist/umd/popper.min.js"
integrity="sha384-
Q6E9RHvbIyZFJoft+2mJbHaEWldlvI9IOYy5n3zV9zzTtmI3UksdQRVvoxMfooAo"
crossorigin="anonymous"></script>
    <script
src="https://stackpath.bootstrapcdn.com/bootstrap/4.5.0/js/bootstrap.min.js
" integrity="sha384-
```

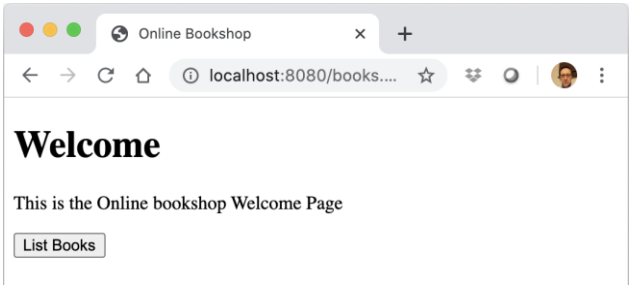
```
OgVRvuATP1z7JjHLkuOU7Xw704+h835Lr+6QL9UvYjZE3Ipu6Tp75j7Bh/kR0JKI"
crossorigin="anonymous"></script>
    <link rel="stylesheet" href="css/style.css" type="text/css"
media="screen"></link>

    <script src="js/books-script.js"></script>

    <title>Online Bookshop</title>
</head>
<body>
<h1>Welcome</h1>
<p>This is the Online bookshop Welcome Page</p>
<form>
    <input type="submit" id="show" value="List Books"/>
</form>

<div id="booklist"></div>
</body>
</html>
```

Points to note

1. Uses Bootstrap
2. Has own style sheet style.cs in css directory
3. Has JavaScript file book-script.js in js directory

You should modify this to whatever files, styles etc. you are using.

The result is shown below:



Step 3: Add the JavaScript file

We now need to add some JavaScript to handle the user clicking on the button.

To do this create a file under the js directory. In the above example, this file is called books-script.js.

Inside this file you need to run some JavaSc ript to register a function with the #show button. This function will use jQuery get function to call out to your service and then define a function to run as a call back when the data is supplied.

An example of what you might create is shown below:

```
$(document).ready(function() {
  $("#show").click(function() {
```

```
    console.log("running click on show");
    event.preventDefault();
    $.get("http://localhost:8080/bookshop/list",
function(books) {
          console.log(books);
          let html = "<div class='book'>";
          html += JSON.stringify(books);

          html += "</div>";
          $("#booklist").append($(html));
    });
  });
});
```

Note that the call back function updates the DOM in the web page by appending data to the element with the id £booklist which is itself a div.

Step 4: Run the Web Application
You now need to run the web application by executing the Spring boot application as you did in the last course.

Step 5: Access the web site
You should now open your Chrome browser and enter the appropriate URL.

In the example being presented here it is

http://localhost:8080/books.html

If you used a different name for your HTML file then you should replace books.html with that.

Next click on the button in the web page; this should cause the backend service tobe invoked and the data displayed back to the user, for example: