

Testing the Web Layer with JUnit

Description: Spring Boot is an open source, micro service-based Java web framework. The Spring Boot framework creates a fully production-ready environment that is completely configurable using its prebuilt code within its own codebase.

Project: Build a simple web application and use JUnit for testing. Spring Test and Spring Boot features are used to test the interactions between Spring and the code. This project is based on “[Testing the Web Layer](#)” and just expands on the screenshots and step by step instructions. This project might also sequences building the application in attempt to reduce dependency errors.

Technology: This project uses the following technology:

Integrated Development Environment (IDE):

[Spring Tool Suite 4](#) (Version: 4.11.0.RELEASE)

Java Development Kit (JDK):

[Oracle's JDK 8](#) (1.8)

Table of Contents

Glossary of Terminology	3
Generate Spring Boot Download	3
Import the Spring Boot Download	3
Import the project: “testing-web”	3
Project Discussion	4
Create the Project from the Import	4
Main Application Class: TestingWebApplication	4
Create Controller class: “HomeController”	4
Run the Application.....	4
Discuss Main Test class: “TestingWebApplicationTests”	5
Create Test class: “SmokeTest”	6
Create Test class: “HttpRequestTest”	7
Update Test class shell: “HttpRequestTest”	7
Update Main Test class: “TestingWebApplicationTest”	9
Create Test class: “WebLayerTest”	11
Project Testing Discussion to this Point	13
Add Service Layer to Project	13

Create Service class: "GreetingService"	13
Create Controller class: "GreetingController"	13
Test Project Changes.....	14
Test Controller Class: "GreetingController"	14
Create Test class: "WebMockTest"	14
Project Conclusion	16

Glossary of Terminology

For a list of key terms and definitions used throughout this and various Spring Boot demo documents see the document titled “Appendix 01 Glossary”.

Generate Spring Boot Download

Follow the instructions in the document title “Appendix 02 Spring Initializr” to generate a spring boot download for this project.

When the document talks about adding dependencies add only these:

Spring Web – Default in the reference document

When the document talks about the items in the “**Project Metadata**” use the values shown below:

“**Group**” use “**com.example**”

“**Artifact**” use “**testing-web**”

“**Name**” use “**testing-web**”

“**Package name**” use “**com.example.testing-web**”

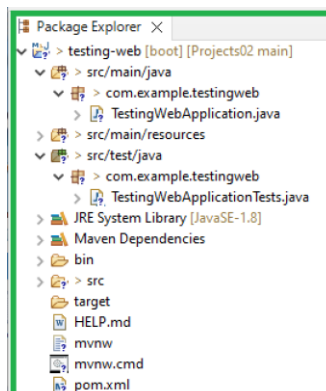
For other items in the “**Project Metadata**” use the defaults. Follow the instructions to extract the files from the zip file into the Spring Tool Suite (STS) 4 workspace.

Import the Spring Boot Download

Open the Spring Tool Suite 4 IDE.

Import the project: “testing-web”

Follow the instructions in the document title “Appendix 03 Import Spring Tool Suite Project” and import the “**testing-web**” project that was created with Spring Initializr. After importing the project, it should look like the following using the IDE “Package Explorer”.



Project Discussion

This project will create a simple web application and use JUnit, Spring Test, and Spring Boot for testing. The project will be tested within the IDE using the Spring Boot built in Tomcat server.

Create the Project from the Import

Use the STS 4 IDE to modify the code after importing the project.

Main Application Class: TestingWebApplication

The class file “TestingWebApplication” containing the startup method generated by [Spring Initializr](#) tool is used without modification.

Create Controller class: “HomeController”

This is a simple class just copy and replace the class shell created.

```
package com.example.testingweb;

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

@Controller
public class HomeController {

    @RequestMapping("/")
    public @ResponseBody String greeting() {
        return "Hello, World";
    }

}
```

Note:

The preceding example does not specify `GET` versus `PUT`, `POST`, and so forth. By default `@RequestMapping` maps all HTTP operations. You can use `@GetMapping` or `@RequestMapping(method=GET)` to narrow this mapping. -- [“Testing the Web Layer”](#)

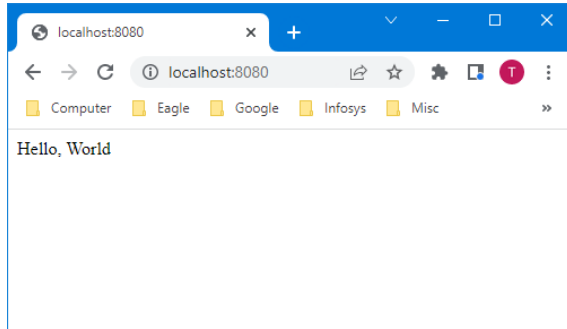
Run the Application

Start the application inside the STS 4 IDE as a spring boot application.

Right click inside “TestingWebApplication” class → “Run As” → “Spring Boot App”

Enter the URL in a browser window:

http://localhost:8080

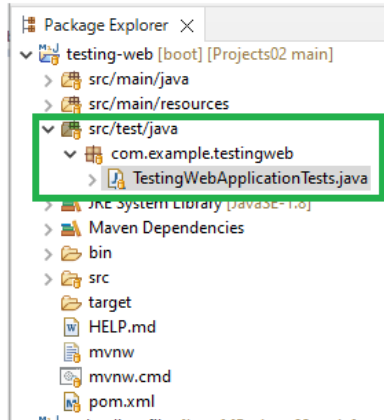


Discuss Main Test class: “TestingWebApplicationTests”

By default Spring Boot adds a testing dependency to the pom.xml.

```
25 <dependency>
26   <groupId>org.springframework.boot</groupId>
27   <artifactId>spring-boot-starter-test</artifactId>
28   <scope>test</scope>
29 </dependency>
```

Also Spring Boot creates a testing class in the “src/test/java” folder under the “com.example.testingweb” package.



Class shell: TestingWebApplicationTests

```
package com.example.testingweb;

import org.junit.jupiter.api.Test;
import org.springframework.boot.test.context.SpringBootTest;

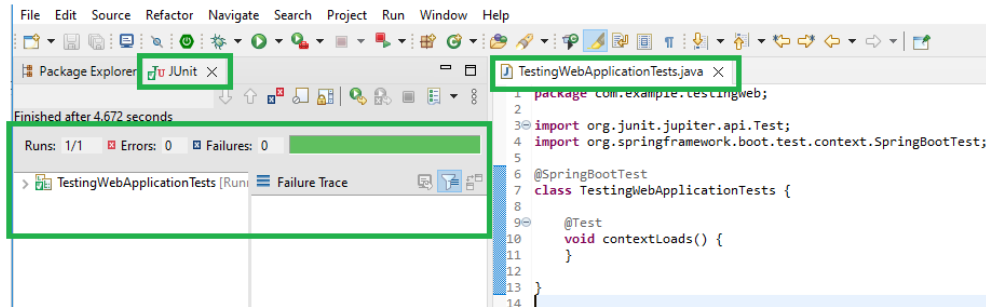
@SpringBootTest
class TestingWebApplicationTests {

    @Test
    void contextLoads() {
    }

}
```

The basic shell code will run and show the test status.
Right click in the test class → “Run As” → “JUnit”

Running as JUnit will automatically start and then stop the Tomcat server inside the IDE for the duration of the testing.



Create Test class: “SmokeTest”

Smoke testing is initial shake out testing to reveal simple failures severe enough to prevent system testing. For example if the program requires a user login prior to doing anything in the application, if the login functionality is not working, then no further actions can be taken.

Under the “src/test/java” folder in the “com.example.testingweb” package create class “SmokeTest”.

Replace the class shell with the following code:

```
package com.example.testingweb;

import static org.assertj.core.api.Assertions.assertThat;

import org.junit.jupiter.api.Test;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

@SpringBootTest
public class SmokeTest {

    @Autowired
    private HomeController controller;

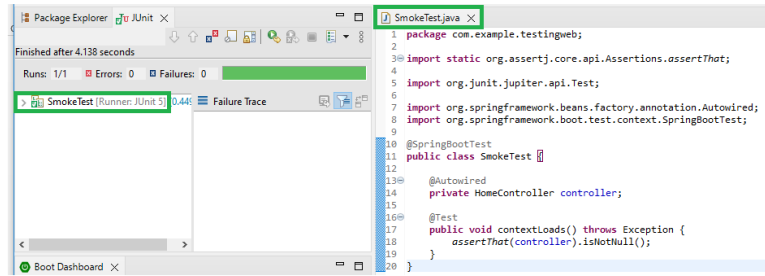
    @Test
    public void contextLoads() throws Exception {
        assertThat(controller).isNotNull();
    }
}
```

The `@SpringBootTest` annotation tells Spring Boot to look for a main configuration class (one with `@SpringBootApplication`, for instance) and use that to start a Spring application context.

Spring interprets the `@Autowired` annotation, and the controller is injected before the test methods are run. We use `AssertJ` (which provides `assertThat()` and other methods) to express the test assertions.

-- “Testing the Web Layer”

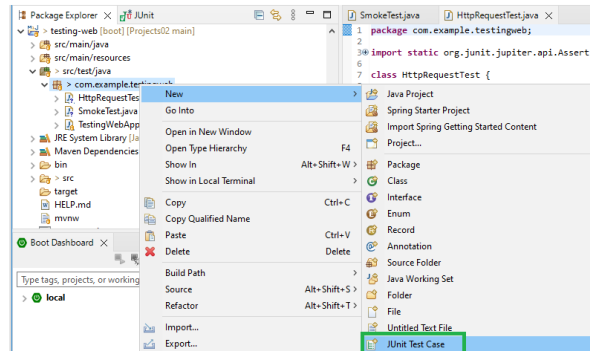
Run this class as a JUnit test and view the results.



Create Test class: “HttpRequestTest”

It is nice to have a sanity check, but you should also write some tests that assert the behavior of your application. To do that, you could start the application and listen for a connection (as it would do in production) and then send an HTTP request and assert the response. -- “[Testing the Web Layer](#)”

To create a JUnit test class from scratch use the IDE Wizard and select JUnit Test Case.



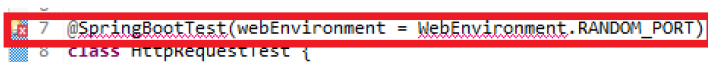
The Wizard will create a basic shell for JUnit testing with the required import statements.

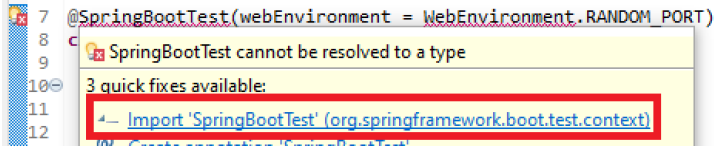


Update Test class shell: “HttpRequestTest”

Add a class annotation and resolve import errors

`@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)`





Add the following important manually, the Wizard did not resolve:

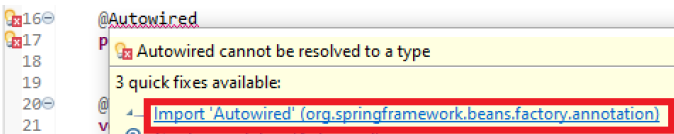
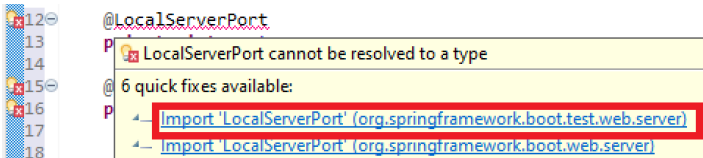
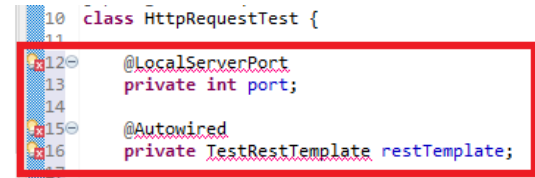
```
import org.springframework.boot.test.context.SpringBootTest.WebEnvironment;
```

```
5 import org.junit.jupiter.api.Test;
6 import org.springframework.boot.test.context.SpringBootTest;
7 import org.springframework.boot.test.context.SpringBootTest.WebEnvironment;
8
```

Add class variables and resolve import errors:

```
@LocalServerPort
private int port;

@Autowired
private TestRestTemplate restTemplate;
```



Replace the existing @Test method with:

```
@Test
public void greetingShouldReturnDefaultMessage() throws Exception {
    assertThat(this.restTemplate.getForObject("http://localhost:" + port + "/",
        String.class)).contains("Hello, World");
}
```

Add the following important manually:

```
import static org.assertj.core.api.Assertions.assertThat;
```



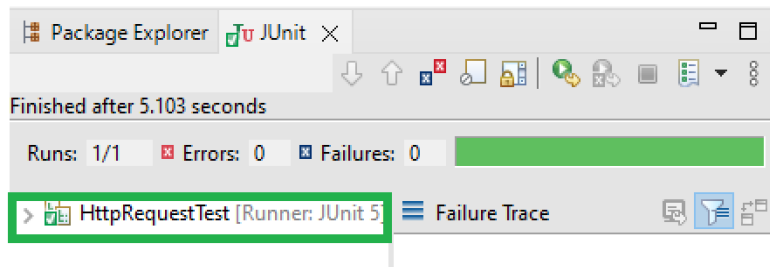
```

10 import org.springframework.boot.test.web.server.LocalServerPort;
11
12 import static org.assertj.core.api.Assertions.assertThat;
13

```

Note the use of `webEnvironment=RANDOM_PORT` to start the server with a random port (useful to avoid conflicts in test environments) and the injection of the port with `@LocalServerPort`. Also, note that Spring Boot has automatically provided a `TestRestTemplate` for you. All you have to do is add `@Autowired` to it. -- ["Testing the Web Layer"](#)

Run the class as JUnit and examine the results:

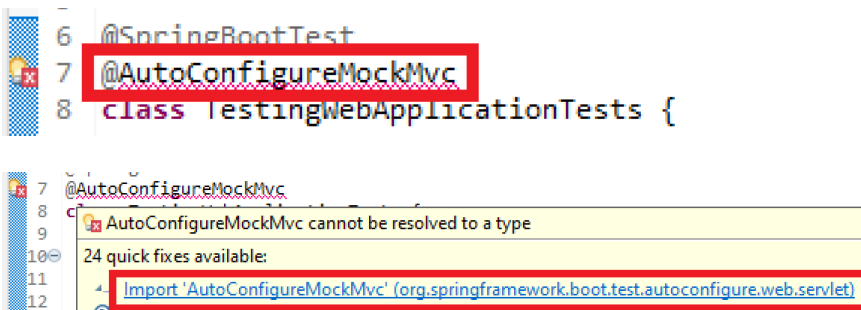


Update Main Test class: ["TestingWebApplicationTest"](#)

Another useful approach is to not start the server at all but to test only the layer below that, where Spring handles the incoming HTTP request and hands it off to your controller. That way, almost of the full stack is used, and your code will be called in exactly the same way as if it were processing a real HTTP request but without the cost of starting the server. To do that, use Spring's `MockMvc` and ask for that to be injected for you by using the `@AutoConfigureMockMvc` annotation on the test case. -- ["Testing the Web Layer"](#)

Add the MockMvc annotation to the class and resolve import error:

```
@AutoConfigureMockMvc
```

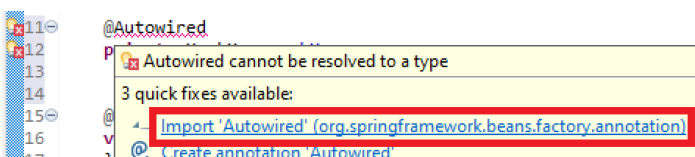


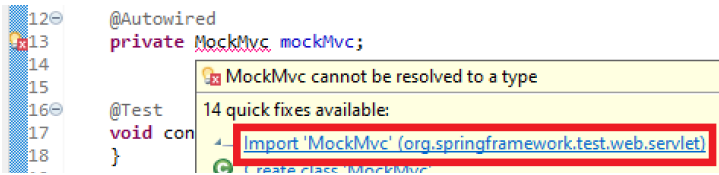
Add autore MockMvc class variable and resolve errors:

```

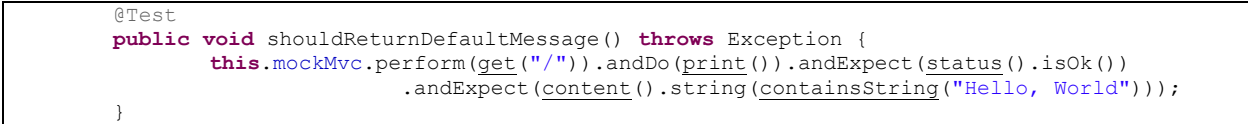
@Autowired
private MockMvc mockMvc;

```

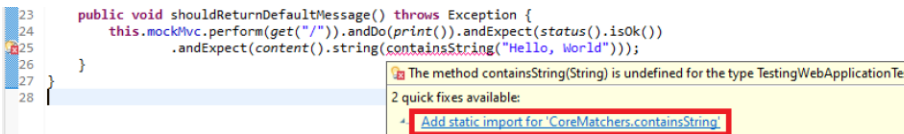
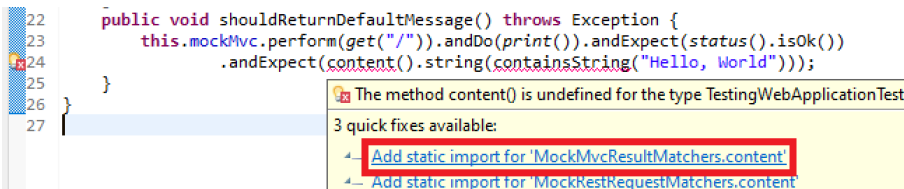
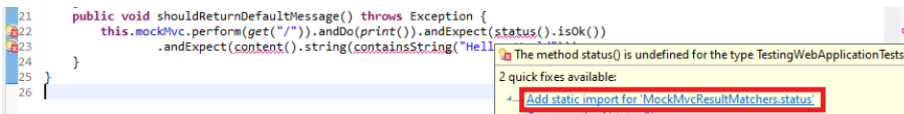
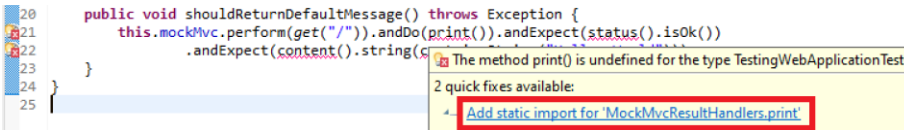
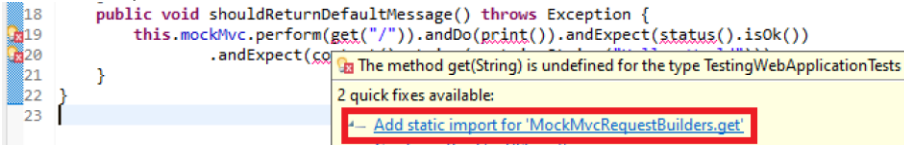




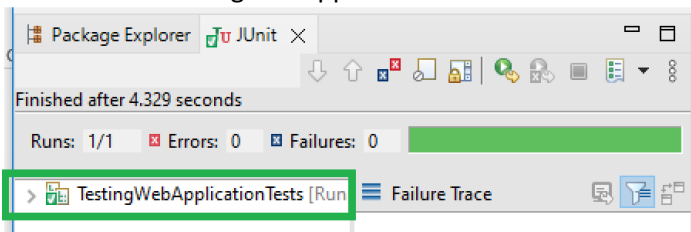
Replace the @Test contextLoads() method with:



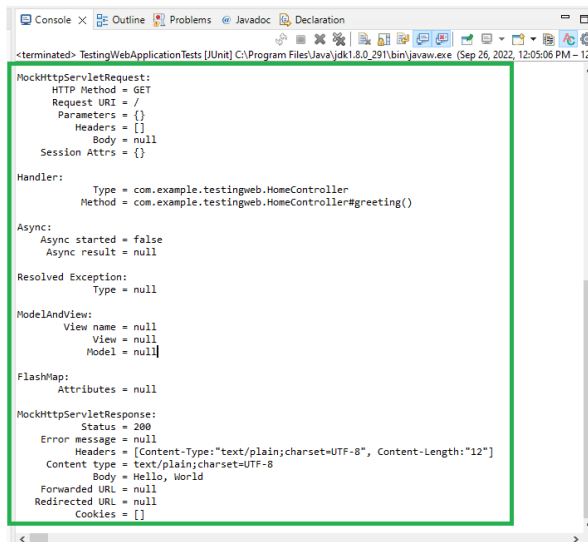
Resolve import errors:



Now run the TestingWebApplicationTest as JUnit we see the results:



There is also output to the console with the modification to the class.



```
<terminated> TestingWebApplicationTests [JUnit] C:\Program Files\Java\jdk1.8.0_291\bin\javaw.exe (Sep 26, 2022, 12:05:06 PM - 12:05:06 PM)

MockHttpServletRequest:
  HTTP Method = GET
  Request URI = /
  Parameters = {}
  Headers = []
  Body = null
  Session Attrs = {}

Handler:
  Type = com.example.testingweb.HomeController
  Method = com.example.testingweb.HomeController#greeting()

Async:
  Async started = false
  Async result = null

Resolved Exception:
  Type = null

ModelAndView:
  View name = null
  View = null
  Model = null

FlashMap:
  Attributes = null

MockHttpServletResponse:
  Status = 200
  Error message = null
  Headers = [Content-Type: text/plain; charset=UTF-8, Content-Length: 12]
  Content type = text/plain; charset=UTF-8
  Body = Hello, World
  Forwarded URL = null
  Redirected URL = null
  Cookies = []
```

Create Test class: “WebLayerTest”

In this test <the test from previous section>, the full Spring application context is started but without the server. We can narrow the tests to only the web layer by using `@WebMvcTest`, as the following listing. -- [“Testing the Web Layer”](#)

For this class copy and replace the class shell:

```
package com.example.testingweb;

import static org.hamcrest.Matchers.containsString;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.result.MockMvcResultHandlers.print;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.content;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

import org.junit.jupiter.api.Test;

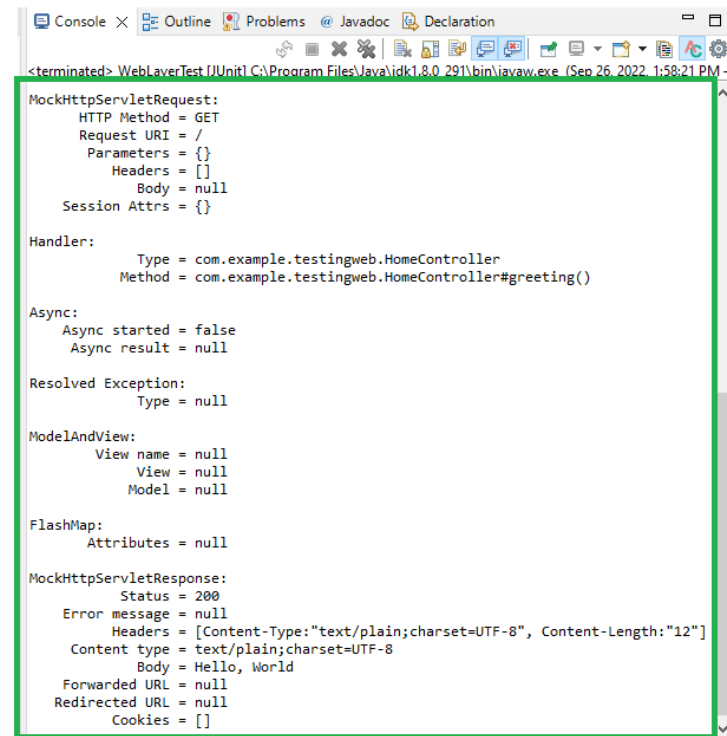
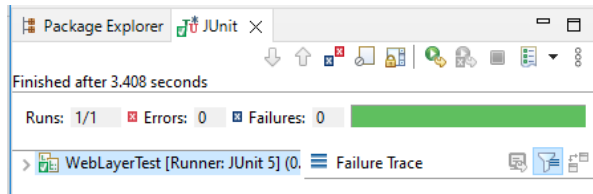
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.test.web.servlet.MockMvc;

@WebMvcTest(HomeController.class)
//tag::test[]
public class WebLayerTest {

    @Autowired
    private MockMvc mockMvc;

    @Test
    public void shouldReturnDefaultMessage() throws Exception {
        this.mockMvc.perform(get("/")).andDo(print()).andExpect(status().isOk())
            .andExpect(content().string(containsString("Hello, World")));
    }
}
//end::test[]
```

Running this class as JUnit we see the same results as with the main class:



Project Testing Discussion to this Point

With the testing performed so far we have only test one layer of the code, the controller layer (HomeController). The project created is simple with only the controller layer. Typical web projects are more complex. In addition to the controller layer it is common to have service layer, data transfer object (DTO) layer, data access object (DAO) layer, and other layers as required by the specific project. The service layer generally has the business rules to apply to the request and data received. DTO layer is just a transport to pass data between the layers. DAO layer interacts with the storing of the data.

Add Service Layer to Project

To demonstrate slightly more complex mock testing, update the project to introduce a service layer. To keep the testing created so far the same add a new controller class.

Create Service class: "GreetingService"

Create service class under folder "src/main/java" in package "com.example.testingweb".
Replace the code shell with:

```
package com.example.testingweb;

import org.springframework.stereotype.Service;

@Service
public class GreetingService {
    public String greet() {
        return "Hello, World";
    }
}
```

This is a very simple service that just returns a greeting string.

Create Controller class: "GreetingController"

Create controller class under folder "src/main/java" in package "com.example.testingweb".
Replace the class shell with:

```
package com.example.testingweb;

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

@Controller
public class GreetingController {

    private final GreetingService service;

    public GreetingController(GreetingService service) {
        this.service = service;
    }

    @RequestMapping("/greeting")
    public @ResponseBody String greeting() {
        return service.greet();
    }
}
```

```
}  
  
}
```

Here the “GreetingController” relies on the service layer to return the message for the greeting.

Test Project Changes

Test Controller Class: “GreetingController”

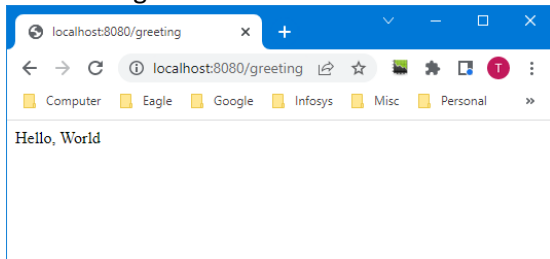
Run the Main Application Class

Right click inside “TestingWebApplication” class → “Run As” → “Spring Boot App”

Enter the Greeting URL in a browser window:

```
http://localhost:8080/greeting
```

When using the IDE Tomcat server and the local host URL for greeting we get the following results:



Create Test class: “WebMockTest”

This test class will test the new controller class “GreetingController”. It will demonstrate the test by injecting a mock service object with a different mock greeting than if using the greeting URL.

Under the “src/test/java” folder in the “com.example.testingweb” package create class “WebMockTest”.

Replace the code shell with the following:

```
package com.example.testingweb;  
  
import static org.hamcrest.Matchers.containsString;  
import static org.mockito.Mockito.when;  
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;  
import static org.springframework.test.web.servlet.result.MockMvcResultHandlers.print;  
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.content;  
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;  
  
import org.junit.jupiter.api.Test;  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;  
import org.springframework.boot.test.mock.mockito.MockBean;  
import org.springframework.test.web.servlet.MockMvc;  
  
@WebMvcTest(GreetingController.class)  
public class WebMockTest {  
  
    @Autowired
```

```

private MockMvc mockMvc;

@MockBean
private GreetingService service;

@Test
public void greetingShouldReturnMessageFromService() throws Exception {
    when(service.greet()).thenReturn("Hello, Mock");
    this.mockMvc.perform(get("/greeting")).andDo(print()).andExpect(status().isOk())
        .andExpect(content().string(containsString("Hello, Mock")));
}
}

```

As mentioned this test class creates a mock service:

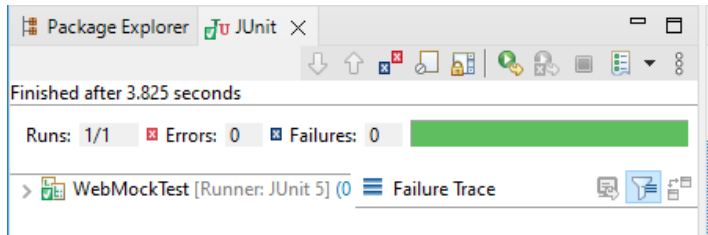
```

@MockBean
private GreetingService service;

```

Inside the @Test method, the method simulates / mock the greeting service with @MockBean. This way a test message "Hello, Mock" can be used in the comparison.

The JUnit test is successful:



In the console output we see the mock greeting.

```

MockHttpServletRequest:
  HTTP Method = GET
  Request URI = /greeting
  Parameters = {}
  Headers = []
  Body = null
  Session Attrs = {}

Handler:
  Type = com.example.testingweb.GreetingController
  Method = com.example.testingweb.GreetingController#greeting()

Async:
  Async started = false
  Async result = null

Resolved Exception:
  Type = null

ModelAndView:
  View name = null
  View = null
  Model = null

FlashMap:
  Attributes = null

MockHttpServletResponse:
  Status = 200
  Error message = null
  Headers = [Content-Type: text/plain; charset=UTF-8, Content-Length: 11]
  Content type = text/plain; charset=UTF-8
  Body = Hello, Mock
  Forwarded URL = null
  Redirected URL = null
  Cookies = []

```

Project Conclusion

This concludes building a simple web application and introducing JUnit and Mockito for testing. Spring Test and Spring Boot built a shell for testing and included test dependency in the pom.xml. This project is very basic and only contains a controller and service layer. Typical web projects will be more complex and JUnit testing with Mockito will be complex and more involved.