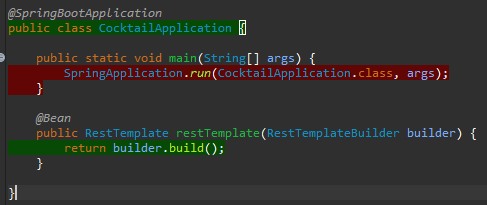
Spring Boot Testing

# Tools we will need

**EclEmma** – this tool we enable us to see the test coverage that our tests provide. We will need to download this tool using the Eclipse Marketplace.

**JUnit** & **Mockito**– These libraries will already be packaged within your Spring Boot Dependencies, specifically in the spring-boot-starter-test dependency

# Drivers and Stubs

In Unit Testing the Software under test will be a Unit in the system, usually this will be a method that we are testing the functionality of.

A **Stub** represents a class that our Unit relies on to complete some of its function.

A **Driver** represents a class that provides information to the Software under test.

# Unit Testing

Below is the default Test Case that Spring Boot will create for you when you first Create a Spring Boot Project.

It uses 2 Annotations from JUnit – which you have likely seen and worked with before, as well as 2 new ones.

* @**SpringBootTest**
  + Specifies that this class will be running Spring Boot tests – provides features that are abstracted from us right now
* **SpringRunner**.**class**
  + An alias for SpringJUnit4ClassRunner – supports using the JUnit 4 annotations @Test, @Before, @Ignore etc.

package com.qa;

import org.junit.Test;

import org.junit.runner.RunWith;

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

import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)

@SpringBootTest

public class CocktailApplicationTests {

    @Test

    public void contextLoads() {

    }

}

# Smoke Test

The following test is quite basic and we will call it a **Smoke** test. Here we are testing that we have set our configuration up correctly, however it doesn’t really test the **functionality** of the system.

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

import org.junit.Test;

import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired;

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

import org.springframework.test.context.junit4.SpringRunner;

import com.qa.rest.CocktailController;

@RunWith(SpringRunner.class)

@SpringBootTest

public class SmokeTest {

@Autowired

private CocktailController controller;

@Test

public void contextLoads() {

assertThat(controller).isNotNull();

}

}

# Looking at Mocking

* **@InjectMocks** – the Class under test
* **@Mock –** The dependencies the class under test requires, its *dependencies*.

@InjectMocks

    CocktailController controller;

    @Mock

    CocktailService service;

    private static final Cocktail MOCK\_COCKTAIL\_1 = new Cocktail(1L, "Vodka somthing");

    private static final Cocktail MOCK\_COCKTAIL\_2 = new Cocktail(2L, "Whiskey Doo-dah");

    @Test

    public void getCocktailsTest() {

        List<Cocktail> MOCK\_LIST = new ArrayList<>();

        MOCK\_LIST.add(MOCK\_COCKTAIL\_1);

        MOCK\_LIST.add(MOCK\_COCKTAIL\_2);

        Mockito.when(service.findAll()).thenReturn(MOCK\_LIST);

        assertEquals(MOCK\_LIST, controller.getAllCocktails());

        Mockito.verify(service).findAll();

    }

The concept goes like this, we want each of our Test Classes to test a class of our actual code base, in this example the **CocktailController** class is **the Class under Test.** Anything that the **CocktailController** class requires in order to operate, for example a **CocktailService** object is **Mocked** using the **@Mock.**

Each of our tests will test a **Unit** in our system, in the example we are testing the **getAllCocktails()** method, the first 4 lines are us **mocking** the dummy response for the **findAll()** method of the **CocktailService** object we have mocked.

We will then call the **getAllCocktails()** of our Class, which in turn will use our mocked object and its dummy response to ensure that we get the functionality we expect from the **Class we are testing.**

# Mocking RestTemplate

A point needs to be made about how we should be sending our HTTP Requests from one MicroService to another. We have chosen to use a RestTemplate Bean that we create ourselves, (hint: Check the Screenshot at the beginning of this handout), we will therefore have to mock our RestTemplate Bean as well as the method that it uses (*exchange()*).

    @Test

    public void microTest() {

        Mockito.when(restTemp.exchange(MOCK\_URL, HttpMethod.GET, null, String.class)).thenReturn(MOCK\_MICRO\_BODY\_RESPONSE);

        assertEquals(MOCK\_MICRO\_BODY\_RESPONSE.getBody(), controller.getMicro());

        Mockito.verify(restTemp).exchange(MOCK\_URL, HttpMethod.GET, null, String.class);

    }

As you have researched how to use RestTemplate you may have come across **RestTemplateBuilder,** it is not recommended that you build your project using the **RestTemplateBuilder** as it makes it quite harder to test.

# Integration Testing

@RunWith(SpringRunner.class)

@WebMvcTest(CocktailController.class)

@AutoConfigureMockMvc

public class WebMockTest {

    @Autowired

    private MockMvc mockMvc;

    @MockBean

    private CocktailService service;

    @MockBean

    private RestTemplate restTemplate;

    private static final Cocktail MOCK\_COCKTAIL\_1 = new Cocktail(1L, "Vodka somthing");

    private static final Cocktail MOCK\_COCKTAIL\_2 = new Cocktail(2L, "Whiskey Doo-dah");

    @Test

    public void getAllTest() throws Exception {

        List<Cocktail> MOCK\_LIST = new ArrayList<>();

        MOCK\_LIST.add(MOCK\_COCKTAIL\_1);

        MOCK\_LIST.add(MOCK\_COCKTAIL\_2);

        when(service.findAll()).thenReturn(MOCK\_LIST);

        mockMvc.perform(get("/getAllCocktails")).andExpect(content().string(containsString("Vodka somthing")));

    }

Here we are not starting the server at all – saving the cost of starting the server. Spring will emulate the HTTP request we are trying to make and passes it to the controller, meaning that the code will be called exactly the same way as if it was processing a real HTTP request.

This is integration testing as it makes use of some of the servlets that Spring Boot abstracts from us. For example, the **Dispatcher** Servlet

# Annotations

* **MockMvc** - this object in part supports testing an application without having to run an application server.
* **@WebMvcTest** - we use this annotation to auto configure the **MockMvc** object.
  + The **parameter** that we pass to this annotation is the class under test.
* **@MockBean** is what this class requires, its *dependencies*.
* **@InjectMocks** – the Class under test
* **@Mock –** The dependencies the class under test requires, its *dependencies*.