Integration Testing in Spring Boot

In this article, we are going to see how integration tests can be run for a Spring Boot application.

by Rajesh Bhojwani, Jan. 17, 19 (https://dzone.com/articles/integration-testing-in-spring-boot-1)

Overview

When we talk about integration testing for a spring boot application, it is all about running an application in *ApplicationContext* and run tests. Spring Framework does have a dedicated test module for integration testing. It is known as *spring-test*. If we are using spring-boot, then we need to use *spring-boot-starter-test* which will internally use *spring-test* and other dependent libraries.

In this article, we are going to see how integration tests can be run for a Spring Boot application.

@SpringBootTest

Spring-Boot provides an `@SpringBootTest` annotation which provides spring-boot features over and above of the *spring-test* module. This annotation works by creating the *ApplicationContext* used in our tests through *SpringApplication*. It starts the embedded server, creates a web environment and then enables *@Test* methods to do integration testing.

By default, *@SpringBootTest* does not start a server. We need to add attribute *webEnvironment* to further refine how your tests run. It has several options:

- *MOCK*(Default): Loads a web ApplicationContext and provides a mock web environment
- *RANDOM_PORT*: Loads a WebServerApplicationContext and provides a real web environment. The embedded server is started and listen on a random port. This is the one should be used for the integration test
- *DEFINED_PORT*: Loads a WebServerApplicationContext and provides a real web environment.
- *NONE*: Loads an *ApplicationContext* by using *SpringApplication* but does not provide any web environment

In the Spring Test Framework, we used to have *@ContextConfiguration* annotation in order to specify which spring *@Configuration* to load. However, It is not required in spring-boot as it automatically searches for the primary configuration when not defined.

If we want to customize the primary configuration, we can use a nested *@TestConfiguration* class in addition to the application's primary configuration.

Application Setup

Let's set up a simple REST API application and see how to write integration tests on it. We will be creating a Student API to create and retrieve student record and write the integration tests for the controller.

Maven Dependency

We will be using spring-boot, spring-data, h2 in-memory DB and spring-boot test for this example:

```
<dependencies>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-test</artifactId>
<scope>test</scope>
</dependency>
<dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
    <dependency>
            <groupId>com.h2database</groupId>
            <artifactId>h2</artifactId>
        </dependency>
</dependencies>
```

Controller

The controller will expose *createStudent()* and *retrieveStudent()* method as REST API:

```
@RestController
public class StudentController {
    @Autowired
    private StudentService studentService;
    @PostMapping("/students")
    public ResponseEntity<Void> createStudent() {
        List<Student>students  =  studentService.createStudent();
        URI location = ServletUriComponentsBuilder.fromCurrentRequest().path(
            "/{id}").buildAndExpand(students.get(0).getId()).toUri();
        return ResponseEntity.created(location).build();
    }
    @GetMapping("/students/{studentId}")
    public Student retrieveStudent(@PathVariable Integer studentId) {
        return studentService.retrieveStudent(studentId);
    }
}
```

Service

This will implement the logic to call the repository to create and retrieve the *student* record:

```
@Component
public class StudentService {
    @Autowired
    private StudentRepository repository;
    public List<Student> createStudent() {
        List<Student> students = new ArrayList<Student>();
        List<Student> savedStudents = new ArrayList<Student>();
        students.add(new Student("Rajesh Bhojwani", "Class 10"));
        students.add(new Student("Sumit Sharma", "Class 9"));
        students.add(new Student("Rohit Chauhan", "Class 10"));
        Iterable<Student> itrStudents=repository.saveAll(students);
        itrStudents.forEach(savedStudents::add);
        return savedStudents;
    }
    public Student retrieveStudent(Integer studentId) {
       return repository.findById(studentId).orElse(new Student());
    }
}
```

Repository

Create a spring data repository *StudentRepository,* which implements all the CRUD operations:

```
@Repository
public interface StudentRepository extends CrudRepository<Student, Integer>{
}
```

A continuación, se presentan cuatro alternativas para realizar pruebas de integración en Spring boot. Estas son:

- **TestRestTemplate**. En este caso se usa una clase para comportarse como un cliente que consume el servicio que se desea probar en el controlador bajo prueba. Lo anterior implica que se levanta la aplicación Rest tal como se hace cuando esta se ejecuta y se prueban todas las capas desde los controladores hasta la persistencia (BD).
- **@MockBean**. Esta anotacón permite crear mocks de clases colaboradoras, de manera que no se utilice el código de producción de esas clases, ya sea porque aún no se han implementado o porque no se desea incluir el código existente.
- **MockMvc**. Esta alternativa se utiliza cuando no se necesita iniciar el servidor, puesto que se requiere probar sólo desde la capa de controladores. En este enfoque, Spring maneja la solicitud HTTP entrante y la entrega al controlador que corresponda (la que se le indica a MockMvc).
- **WebTestClient**. Se utiliza cuando se construyen servicios mediante programación reactiva.

TestRestTemplate

As explained above, for integrating testing of a spring-boot application, we need to use *@SpringBootTest.* spring-boot also does provide other classes like *TestRestTemplate* to test the REST APIs. Like *RestTemplate* class, it also does have methods *getForObject(), postForObject(), exchange(),* etc.. Let's implement *@Test* methods to test create and retrieve both.

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class StudentControllerTests {
    @LocalServerPort
    private int port;
    TestRestTemplate restTemplate = new TestRestTemplate();
    HttpHeaders headers = new HttpHeaders();
    @Test
    public void testCreateStudent() throws Exception {
        HttpEntity<String> entity = new HttpEntity<String>(null, headers);
        ResponseEntity<String> response = restTemplate.exchange(
          createURLWithPort("/students"), HttpMethod.POST, entity, String.class);
        String actual = response.getHeaders().get(HttpHeaders.LOCATION).get(0);
        assertTrue(actual.contains("/students"));
    }
    @Test
    public void testRetrieveStudent() throws Exception {
        HttpEntity<String> entity = new HttpEntity<String>(null, headers);
        ResponseEntity<String> response = restTemplate.exchange(
          createURLWithPort("/students/1"), HttpMethod.GET, entity, String.class);
        String expected = "{\"id\":1,\"name\":\"Rajesh Bhojwani\",\"description\":\"Class 10\"}";
        JSONAssert.assertEquals(expected, response.getBody(), false);
    }
    private String createURLWithPort(String uri) {
        return "http://localhost:" + port + uri;
    }
}
```

In the above code, we have used *WebEnvironment.RANDOM_PORT* to spin up the application on random port temporarily. *@LocalServerPort* helps to read the current port and build the URI to hit by template class. We have used *exchange()* method as we want to get *ResponseEntity* as return type.

@MockBean

With *TestRestTemplate*, we have tested all the layers from Controller to DB layer end to end. However, sometimes DB setup or third-party services won't be ready and you still want to test all the application layers in your scope. That would require mocking all the external systems and services. *@MockBean* helps to enable the mocking of a certain layer. In this example, we will be mocking the *StudentRepository:*

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class StudentControllerMockTests {
    @Autowired
    private StudentService studentService;
    @MockBean
    private StudentRepository studentRepository;
    @Test
    public void testRetrieveStudentWithMockRepository() throws Exception {
        Optional<Student> optStudent = Optional.of(new Student("Rajesh","Bhojwani"));
        when(studentRepository.findById(1)).thenReturn(optStudent);

        assertTrue(studentService.retrieveStudent(1).getName().contains("Rajesh"));
    }
}
```

We have used *Mockito* methods along with *@MockBean* to set the expected response.

MockMvc

There is one more approach in which server start is not required at all but test only the layer below that. In this approach, Spring handles the incoming HTTP request and hands it off to the controller. That way, almost the full stack is used, and the code will be called the same way as if it was processing a real HTTP request, but without the cost of starting the server. To do that we will use Spring's *MockMvc*, and to inject that we will be using another annotation called *@AutoConfigureMockMvc*. Let's implement it for both create and retrieve use case:

```
@RunWith(SpringRunner.class)
@SpringBootTest
@AutoConfigureMockMvc
public class StudentControllerMockmvcTests {
    @Autowired
    private MockMvc mockMvc;
    @Test
    public void testCreateRetrieveWithMockMVC() throws Exception {
        this.mockMvc.perform(post("/students")).andExpect(status().is2xxSuccessful());
        this.mockMvc.perform(get("/students/1")).andDo(print()).andExpect(status().isOk())
          .andExpect(content().string(containsString("Rajesh")));
    }
}
```

WebTestClient

With Spring 5, *Webflux* has been introduced to provide support for reactive streams. In that scenario, we would need to use *WebTestClient* class to test the REST API.

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
public class WebTestClientExampleTests {
    @Autowired
    private WebTestClient webClient;
    @Test
    public void exampleTest() {
        this.webClient.get().uri("/students/1").exchange().expectStatus().isOk()
           .expectBody(String.class).isEqualTo("Rajesh Bhojwani");
    }
}
```

Conclusion

In this article, we have seen how we can do integration testing using Spring Boot test framework with the support of several different annotations.

As always, you can find the source code related to this article on GitHub.