axiateam
RDT company

Microservices

Testing Spring Boot

Dependencias Maven

- Spring-boot-starter-test es la dependencia principal que contiene la mayoría de los elementos necesarios para nuestras pruebas.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
    <version>2.5.0</version>
</dependency>
```

- **Junit 4:** A partir de Spring Boot 2.4, el antiguo motor de JUnit se eliminó de spring-boot-starter-test. Si aún queremos usar JUnit 4, debemos agregar la siguiente dependencia.

```
<dependency>
    <groupId>org.junit.vintage</groupId>
    <artifactId>junit-vintage-engine</artifactId>
    <scope>test</scope>
    <exclusions>
        <exclusion>
            <groupId>org.hamcrest</groupId>
            <artifactId>hamcrest-core</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

- Para ejecutar con Maven:

```
mvn clean test
mvn clean -Dtest=Test1, Test2 test
mvn clean -Dtest=com.microcompany.productsservice.persistence.** test
```

Pruebas con @DataJpaTest

- **@ExtendWith(SpringExtension.class)** proporciona un puente entre las funciones de prueba de Spring Boot y JUnit.
  - Siempre que estemos usando cualquier característica de prueba de Spring Boot en nuestras pruebas JUnit, se requerirá esta anotación.
- **@DataJpaTest** proporciona una configuración estándar necesaria para probar la capa de persistencia:
  - configurar H2, una base de datos en memoria
  - configurar Hibernate, Spring Data y DataSource
  - realizando un @EntityScan
  - activa el registro de SQL
- Para llevar a cabo operaciones de base de datos, necesitamos un EntityManager.
  - Para configurar estos datos, podemos usar **TestEntityManager**.
  - **@AutoConfigureTestEntityManager** pre-configurará este entity manager.
  - Spring Boot TestEntityManager es una alternativa al JPA EntityManager estándar que proporciona métodos comúnmente utilizados al escribir pruebas.
- Disponemos, también, de la anotación **@Sql** que nos permite cargar archivos sql para las pruebas.

Pruebas con @DataJpaTest

```java
@ExtendWith(SpringExtension.class)
@DataJpaTest()
@ComponentScan(basePackages = {"com.microcompany.productsservice.persistence"})
@AutoConfigureTestEntityManager
class JPAProductsRepositoryTest {

    @Autowired
    private TestEntityManager entityManager;

    @Autowired
    private IProductsRepository jpaRepo;

    @Test
    void findAll() {
        // given
        Product aProduct = new Product(null, "Fake Product", "123-123-1234");
        entityManager.persist(aProduct);
        entityManager.flush();

        // when
        List<Product> prods = jpaRepo.findAll();

        // then
        assertThat(prods.size()).isGreaterThan(0);

        assertNotNull(prods);
    }

    // ... Other test cases
}
```

Pruebas de integración con @SpringBootTest

- Las pruebas de integración se enfocan en integrar diferentes capas de la aplicación.
- Idealmente, deberíamos mantener las pruebas de integración separadas de las pruebas unitarias y no deberíamos ejecutarlas juntas.
- Podemos hacer esto usando un perfil diferente para ejecutar solo las pruebas de integración.
- Las razones para hacer esto son que:
  - las pruebas de integración consumen mucho tiempo y
  - pueden necesitar una base de datos real para ejecutarse.

MockMvc

- El framework de prueba Spring MVC, también conocido como MockMvc, proporciona soporte para probar aplicaciones Spring MVC.
  - https://docs.spring.io/spring-framework/reference/6.1/testing/spring-mvc-test-framework/server.html
- Realiza el manejo completo de solicitudes Spring MVC, pero a través de objetos simulados de la solicitud y respuesta en lugar de un servidor en ejecución.

Pruebas de integración con @SpringBootTest

- La anotación @SpringBootTest es útil cuando necesitamos **arrancar todo el contenedor**.
  - La anotación funciona creando el ApplicationContext que se utilizará en nuestras pruebas.
  - Lo que significa que podemos hacer @Autowire de cualquier bean que haya sido recogido por el escaneo de componentes en nuestra prueba.
- Podemos usar el atributo **webEnvironment** de @SpringBootTest para configurar nuestro entorno en tiempo de ejecución.
  - Se usa **WebEnvironment.MOCK** para que el contenedor funcione en un entorno de servlet simulado.
- **@AutoConfigureMockMvc** habilita toda la configuración automática relacionada con MockMvc y SÓLO MockMvc.
- La anotación **@TestPropertySource** ayuda a configurar las ubicaciones de los archivos de propiedades específicos de nuestras pruebas.
  - Tener en cuenta que el archivo de propiedades cargado con @TestPropertySource **anulará el archivo application.properties** existente.

```
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.MOCK,
        classes = ProductsServiceApplication.class)
@AutoConfigureMockMvc
@TestPropertySource(locations = "classpath:application-integrationtest.properties")
public class ProductServiceControllerIntegrationTest {

    @Autowired
    private MockMvc mvc;

    @Autowired
    private ProductsRepository repository;

    // test cases here
}
```

Pruebas de integración con @SpringBootTest

Los casos de prueba para las pruebas de integración pueden parecerse a las pruebas unitarias de la capa del controlador:

```
@Test
public void givenProducts_whenGetProducts_thenStatus200() throws Exception {
    Product nuevoProd = new Product();
    nuevoProd.setName("Nuevo Prod");
    repository.save(nuevoProd);

    mvc.perform(get("/products").contentType(MediaType.APPLICATION_JSON))
            .andExpect(status().isOk())
            .andExpect(content().contentTypeCompatibleWith(MediaType.APPLICATION_JSON))
            .andExpect(jsonPath("$[0].name", is("Nuevo Prod")));
}
```

La diferencia con las pruebas unitarias de la capa del controlador es que aquí no se simula nada y se **ejecutan escenarios de extremo a extremo**.

Ejecutando tests de requests:

- https://docs.spring.io/spring-framework/reference/6.1/testing/spring-mvc-test-framework/server-performing-requests.html
- https://docs.spring.io/spring-framework/reference/6.1/testing/spring-mvc-test-framework/server-defining-expectations.html

Configuración de prueba con @TestConfiguration

- Es posible que deseemos evitar el arranque del contexto de la aplicación real y usar una configuración de prueba especial. Podemos lograr esto con la anotación **@TestConfiguration** .
- Hay dos formas de usar la anotación

(1). Una **clase interna estática** en la misma clase de prueba donde queremos @Autowire the bean:

```
@ExtendWith(SpringExtension.class)
public class ProductServiceImplIntegrationTest {

    @TestConfiguration
    static class ProductServiceImplTestContextConfiguration {
        @Bean
        public ProductsService productsService() {
           return new EmployeeService() {
                // implement methods
            };

        }
    }
}
```

Configuración de prueba con @TestConfiguration

(2). Crear una **clase de configuración de prueba** separada:

```
@TestConfiguration
public class ProductServiceImplTestContextConfiguration {
    @Bean
    public ProductsService productsService() {
       return new EmployeeService() {
            // implement methods
        };

    }
}
```

- Las clases de configuración anotadas con @TestConfiguration están excluidas del análisis de componentes, por lo tanto, debemos **importarlas explícitamente** en cada prueba en la que queramos hacer @Autowire .

```
@ExtendWith(SpringExtension.class)
@Import(ProductServiceImplTestContextConfiguration.class)
public class ProductServiceImplIntegrationTest {

    @Autowired
    private ProductsService productsService;

    // remaining class code
}
```

Mocking con @MockBean

- Nuestro código de capa de Servicio usualmente depende de nuestro Repositorio.
- Sin embargo, para probar la capa de servicio , no necesitamos saber cómo se implementa la capa de persistencia ni preocuparnos por ella. Idealmente, deberíamos poder escribir y probar nuestro código de capa de servicio sin cableado con nuestra capa de persistencia completa.
- Para lograr esto, podemos usar el soporte de simulación proporcionado por Spring Boot Test.

```
@ExtendWith(SpringExtension.class)
public class ProductServiceImplIntegrationTest {

    @TestConfiguration
    static class ProductServiceImplTestContextConfiguration {
        @Bean
        public ProductsService productsService() {
            return new ProductsService();
        }
    }

    @Autowired
    private ProductsService productsService;

    @MockBean
    private ProductsRepository productsRepository;

    // write test cases here

}
```

Mocking con @MockBean

- @MockBean crea una simulación para el Bean de repositorio , que se puede usar para omitir la llamada al repositorio real.
- Esto lo logramos usando **Mockito**.
  https://site.mockito.org/

  Tip – Mockito cookbook:
  https://www.baeldung.com/mockito-behavior

```
public void setUp() {
    List<Product> products = Arrays.asList(
        new Product(1L,"Fake product")
    );
    Mockito.when(productsRepository.findByNameContaining("Fake"))
            .thenReturn(products);
}
```

- El caso de prueba será más simple:

```
@Test
public void whenValidText_thenProductsShouldBeFound() {
    String text = "Fake";
    List<Product> found = productsService.
        getProductsByText(text);

    assertThat(found).isNotEmpty();
    assertThat(found.get(0).getName()).contains("Fake");
}
```

Pruebas unitarias con @WebMvcTest

- **@AutoConfigureWebMvc**
  - Permite cargar toda la configuración automática relacionada con la capa web y **SÓLO la capa web**.
  - Este es un subconjunto de la configuración automática general.
  - Se usa si se necesita configurar la capa web para realizar pruebas pero no necesita usar MockMvc.
- Para probar los controladores , podemos usar **@WebMvcTest**.
  - Incluye @AutoConfigureWebMvc y @AutoConfigureMockMvc, entre otras funciones.
  - Configurará automáticamente la infraestructura Spring MVC para nuestras pruebas unitarias.
- En la mayoría de los casos, @WebMvcTest se limitará a arrancar un solo controlador.
- También podemos usarlo junto con @MockBean para proporcionar implementaciones simuladas para cualquier dependencia requerida.
- @WebMvcTest también configura automáticamente MockMvc , que ofrece una forma poderosa de probar fácilmente los controladores MVC sin iniciar un servidor HTTP completo.

Pruebas unitarias con @WebMvcTest

```
@ExtendWith(SpringExtension.class)
@WebMvcTest(ProductServiceController.class)
public class ProductServiceControllerIntegrationTest {

    @Autowired
    private MockMvc mvc;

    @MockBean
    private ProductsRepository repository;

    @Test
    public void givenProducts_whenGetProducts_thenReturnJsonArray() throws Exception {

        Product aProduct = new Product(1L, "Fake product");

        List<Product> allProducts = Arrays.asList(aProduct);

        given(repository.findAll()).willReturn(allProducts);

        mvc.perform(get("/products")
                        .contentType(MediaType.APPLICATION_JSON))
                .andExpect(status().isOk())
                .andExpect(jsonPath("$", hasSize(1)))
                .andExpect(jsonPath("$[0].name", is(aProduct.getName())));
    }

}
```

Pruebas configuradas automáticamente

Además de las anotaciones anteriores, algunas anotaciones ampliamente utilizadas:

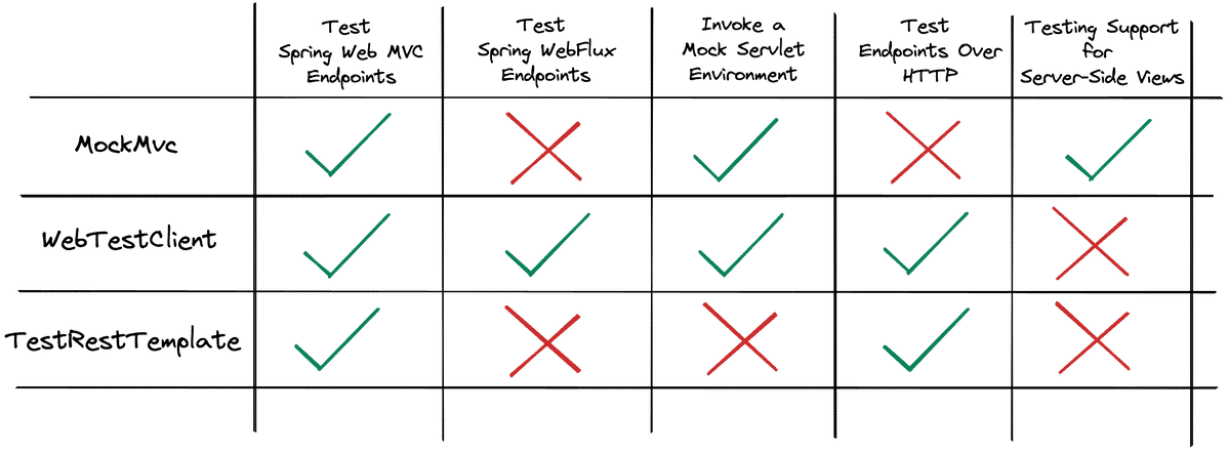
- **@WebFluxTest**: probar los controladores Spring WebFlux. A menudo se usa junto con @MockBean para proporcionar implementaciones simuladas para las dependencias requeridas.
- **@JdbcTest**: probar aplicaciones JPA, pero es para pruebas que solo requieren un DataSource. La anotación configura una base de datos incrustada en memoria y un JdbcTemplate.
- **@JooqTest**: pruebas relacionadas con jOOQ. Configura un DSLContext.
- **@DataMongoTest**: para probar aplicaciones MongoDB. De manera predeterminada, configura un MongoDB incrustado en memoria si el controlador está disponible a través de dependencias, configura un MongoTemplate, busca clases de @Document y configura repositorios de Spring Data MongoDB.
- **@DataRedisTest**: prueba de aplicaciones de Redis. Busca clases de @RedisHash y configura repositorios Spring Data Redis de forma predeterminada.
- **@DataLdapTest**: configura un LDAP incrustado en la memoria (si está disponible), configura un LdapTemplate , busca clases de @Entry y configura repositorios Spring Data LDAP de manera predeterminada.
- **@RestClientTest**: probar clientes REST. Configura automáticamente diferentes dependencias, como la compatibilidad con Jackson, GSON y Jsonb; configura un RestTemplateBuilder; y agrega soporte para MockRestServiceServer de forma predeterminada.
- **@JsonTest**: Inicializa el contexto de la aplicación Spring solo con los beans necesarios para probar la serialización JSON.

  https://docs.spring.io/spring-framework/reference/testing/annotations/integration-spring.html

TestRestTemplate

- Alternativa conveniente de RestTemplate que es adecuada para **pruebas de integración** y **E2E**.
- **TestRestTemplate** es tolerante a fallos.

  Respuestas 4xx y 5xx no generan una excepción y, pueden detectarse a través de la entidad de respuesta y su código de estado.
- Opcionalmente, un TestRestTemplate puede llevar encabezados de autenticación básica.
- En la siguiente tabla se puede ver una comparación entre distintos casos para las opciones de testing en spring:



TestRestTemplate

Con TestRestTemplate se iniciará la aplicación y las peticiones serán reales, como si lo hiciera un cliente de verdad.

```
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class HttpRequestTest {

    @Value(value="${local.server.port}")
    private int port;

    @Autowired
    private TestRestTemplate restTemplate;

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

TestRestTemplate

```java
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class SpringBootDemoApplicationTests{
    @Autowired
    private TestRestTemplate restTemplate;

    @LocalServerPort
    private int port;

    @Test
    public void givenAProduct_whenPostWithHeader_thenSuccess() throws URISyntaxException {
        final String baseUrl = "http://localhost:"+port+"/api/products";
        URI uri = new URI(baseUrl);
        Product product = new Product(null, "New Test Product","111-222-3333");

        HttpHeaders headers = new HttpHeaders();
        headers.set("ACCEPT", "application/xml");

        HttpEntity<Product> request = new HttpEntity<>(product, headers);

        ResponseEntity<String> result = this.restTemplate.postForEntity(uri, request, String.class);

        System.out.println(result);

        //Verify request succeed
        assertThat(result.getStatusCodeValue()).isEqualTo(201);
    }

    // ...

}
```