Spring Boot Testing

Spring Boot Testing - Georgii Lvov - 21.02.2024

Types of spring-microservice testing

Unit test

- Focus on testing individual classes in isolation
- Mock all dependencies of the class under the test
- Do not load Spring application context

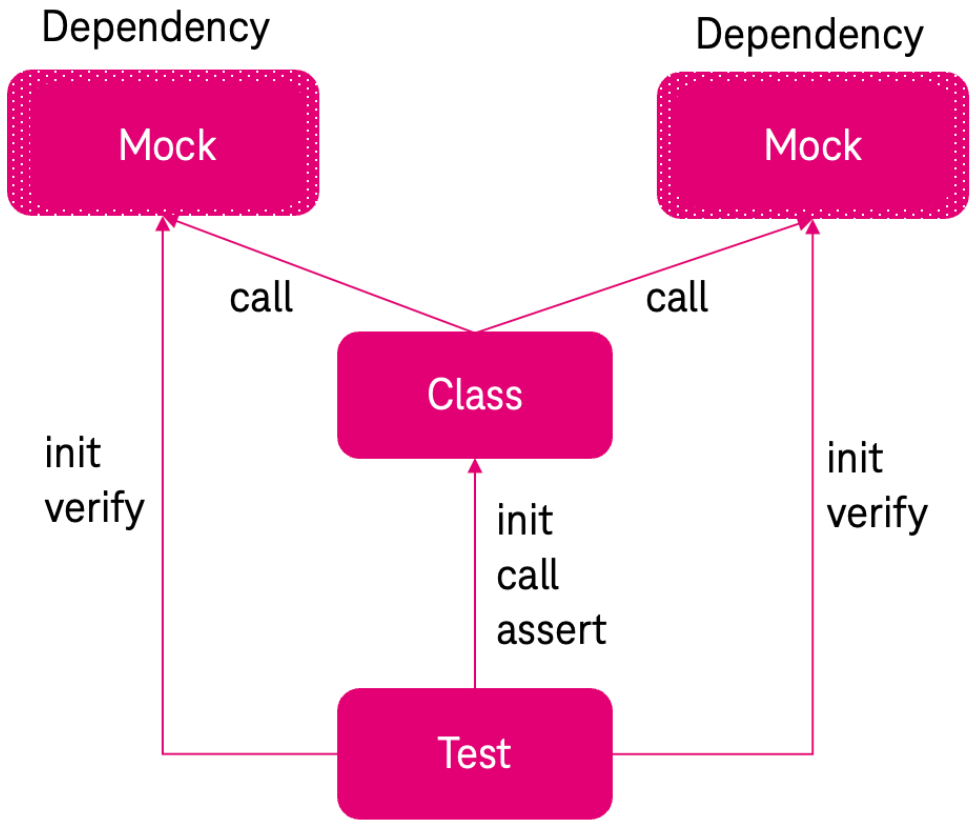
Component test

- Focus on testing a **single spring-component** or **specific layer** such as MVC or DB layer
- Mock some dependencies (beans) if necessary
- Load only beans necessary for testing

Integration test

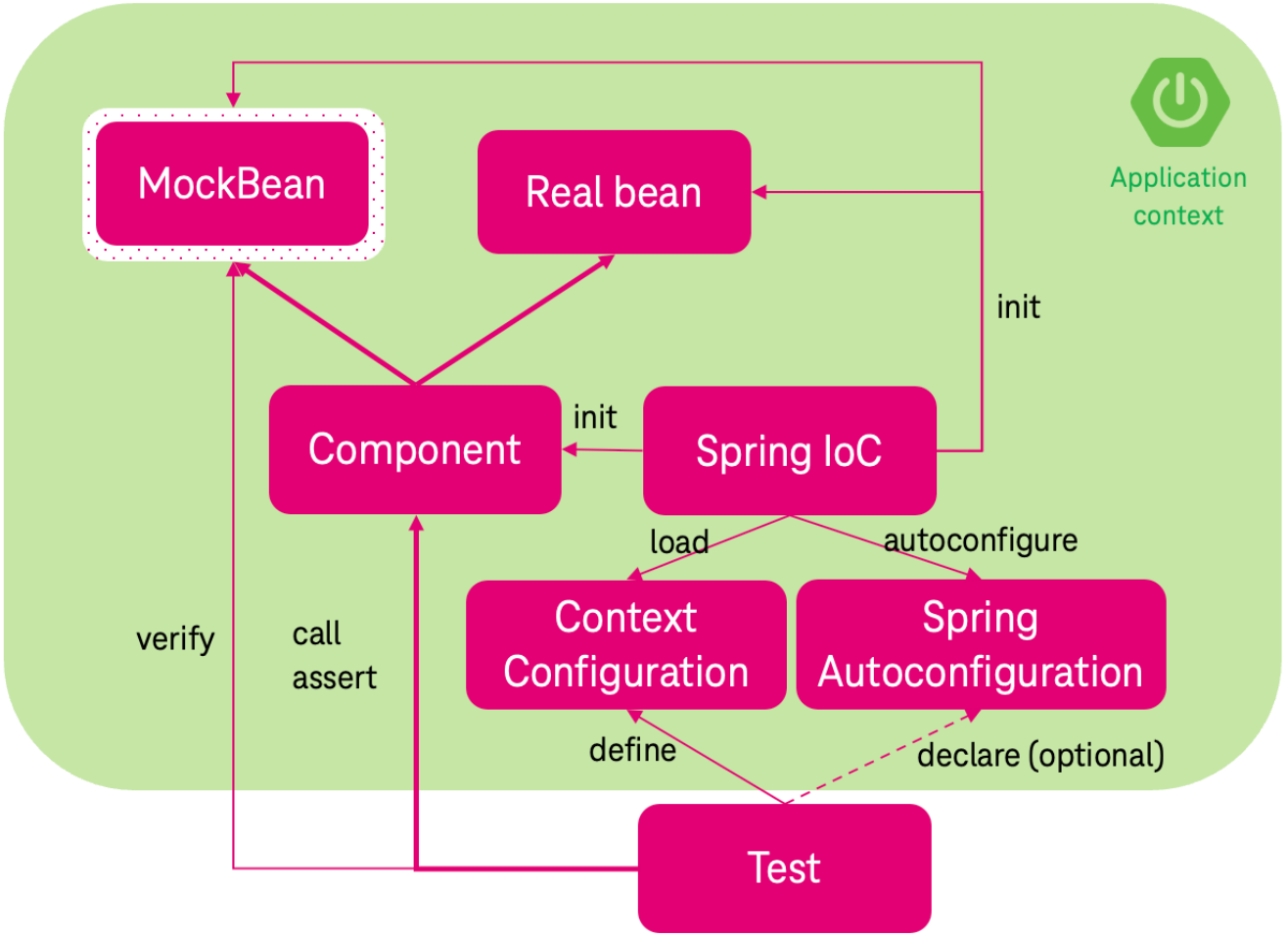
- Focus on testing the **entire microservice**
- Do not mock any beans or classes
- Load the full Spring application context
- Used technologies (e.g. test-database) should be close to production

Unit test



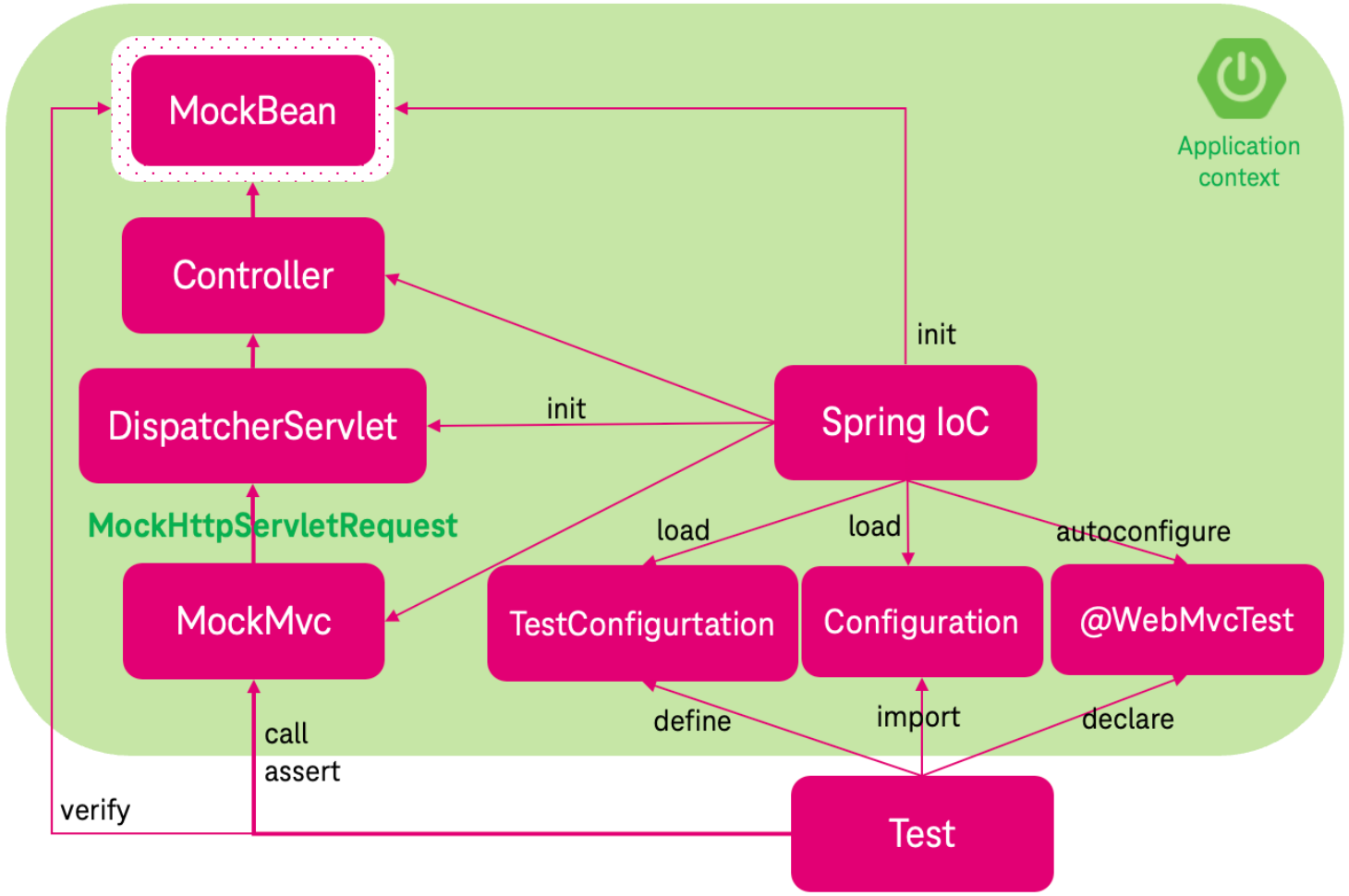
- Unit under test is a class
- Class functionality is tested in isolation
- Dependencies are classes and are mocked
- Test (itself or using the Mockito Framework) controls the lifecycle of all actors

Component tests. Generic.



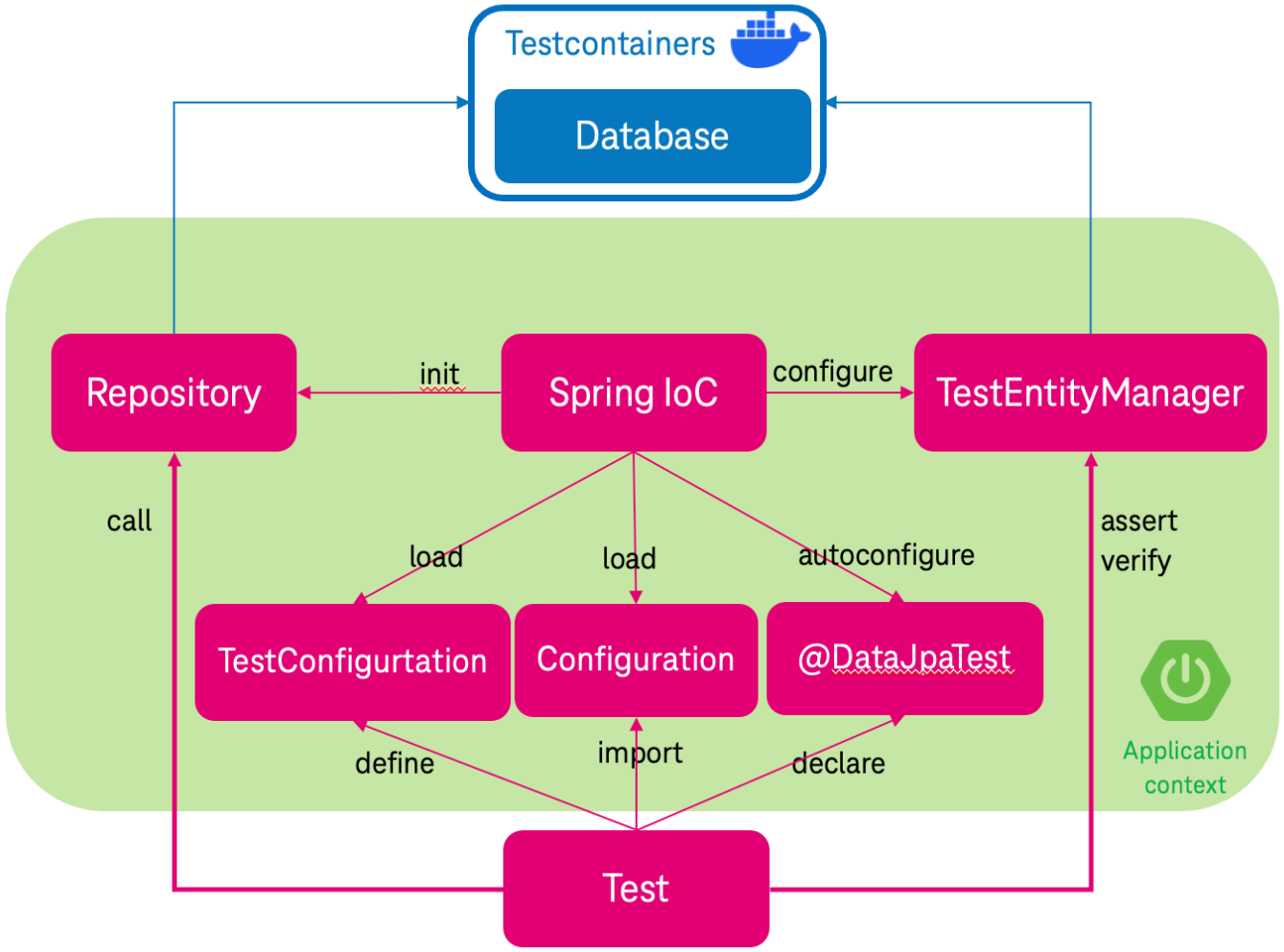
- Unit under test is a spring component (spring-bean)
- Test specifies which components will be present in context
- Dependencies are real spring-beans or mocked-beans
- Spring controls the lifecycle of all actors

Component tests. WebMvcTest.



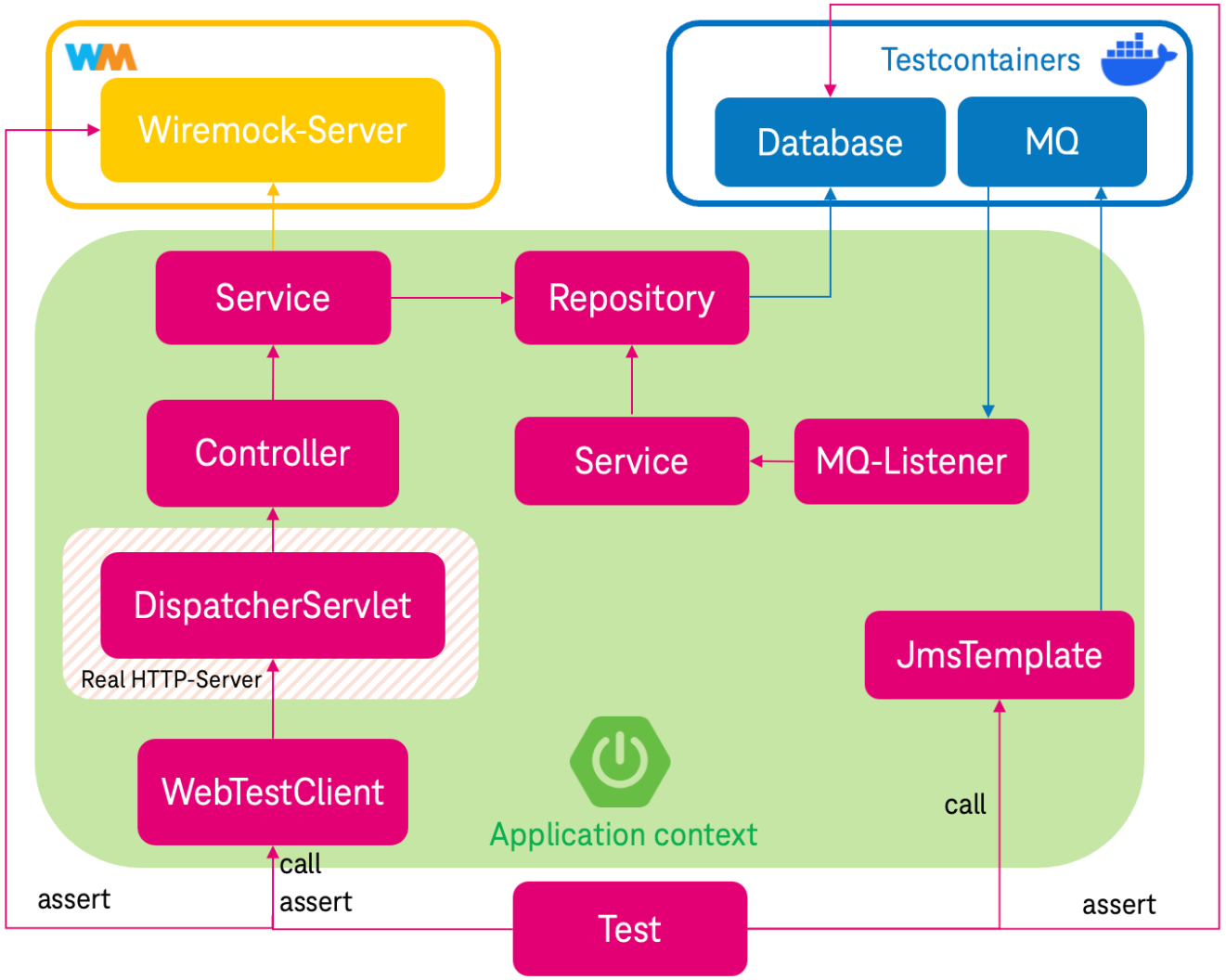
- Unit under test is a class annotated with *@Controller* / *@RestController*
- Dependencies of the controller class are mocked
- Test is annotated with *@WebMvcTest* and defines test-configuration if necessary
- Spring only loads web context
- Servlet environment is mocked
- The test primarily utilizes *MockMvc*
- Spring controls the lifecycle of all actors

Component tests. DataJpaTest.



- Unit under test is a class annotated with *@Repostirory*
- Test is annotated with *@DataJpaTest* and defines test-configuration if necessary
- The database is either embedded or e.g. launched within a *Testcontainers* instance
- Spring only loads the beans needed for JPA testing
- Spring controls the lifecycle of all actors

Integration tests



- Unit under test is a **microservice**
- Test is annotated with *@SpringBootTest*
- Real HTTP-Server (e.g. Tomcat) is running
- Database is close to the production DB
- No mocked or overridden beans in context
- Spring loads full context with spring-boot features
- Spring controls the lifecycle of all actors

Integration test. Tips.

MockMvc vs WebTestClient

MockMvc

- Does not make actual HTTP requests to a real HTTP-server
- Sends *MockHttpServletRequest* to DispatcherServlet
- Code that relies on lower-level servlet container behaviour **cannot be tested**
- Does not require a real HTTP-server
- Testing within a mocked environment is faster

WebTestClient

- Makes actual HTTP requests to a real HTTP-server
- Requires a real HTTP-server (e.g. Tomcat)
- Code that relies on lower-level servlet container behaviour **can be tested**
- Requires *spring-webflux* in classpath

By default, when using `@SpringBootTest`, the web environment is mocked.
To start a real HTTP server, it must be explicitly specified:

```
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT
```

Adding both *spring-boot-starter-web* and *spring-boot-starter-webflux* modules in your application results in Spring Boot auto-configuring Spring MVC, not WebFlux.
This behaviour has been chosen because many Spring developers add *spring-boot-starter-webflux* to their Spring MVC application to use the reactive WebClient.
But we can still enforce our choice by setting the application type to `WebApplicationType.REACTIVE`.

@Configuration vs @TestConfiguration

@Configuration 

- Is scanned by `@SpringBootTest`, it means by each SpringBootTest class
- It deactivates the scanning process of spring boot test

  if used as follows:

```
@SpringBootTest(classes = YourConfiguration.class)
```

@TestConfiguration 

- Is not scanned by `@SpringBootTest`
- It does not deactivate the scanning process of spring boot test

  if used as follows:

```
@SpringBootTest(classes = YourTestConfiguration.class)
```

When no `@Configuration` class is specified in the *classes* attribute of the `@SpringBootTest` annotation, Spring Boot test starts searching for the `@SpringBootConfiguration` annotation.
This search occurs in both the *test-classpath* and *the main-classpath* (as the *test-classpath* extends the *main-classpath*).
The `@SpringBootConfiguration` annotation is typically found inside the `@SpringBootApplication` annotation in the application's main class.
Once found, Spring Boot scans all configurations within the same package of main class and its sub-packages.
Consequently, any `@Configuration` classes located within test packages are also scanned.

@Transactional

Avoid relying on `@Transactional` to clear integration test data, opt for SQL scripts with `@Sql` annotation.

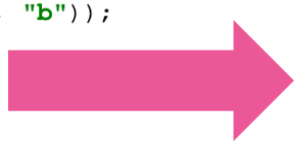
If test is `@Transactional`, it will rollback the transaction at the end of each test method by default.
But, if this arrangement is used in combination with either RANDOM_PORT or DEFINED_PORT, any transaction initiated on the server won't rollback as the test is running in a **different** transaction, than the server processing.

It also causes the test data we prepared for the database **not be committed** until the end of the test.
Hence the other transaction that handles our http-request to the server **does not see the uncommitted data** from test transaction.

In integration tests, **avoid** depending on your code to create test data, like using `jpaRepository.save(testData)`, as bugs in your repository methods may lead to failures during test data preparation. Instead, consider using SQL scripts with the `@Sql` annotation.

Arrange-Act-Assert (AAA) Pattern

```java
@Test
void test() {
    // arrange
    OrderEntity orderEntity = getOrderEntity();

    when(orderRepository.findById(1L))
            .thenReturn(Optional.of(orderEntity));

    // action
    orderProcessor.updateOrder(1L, new OrderRequest("a", "b"));

    // assert
    verify(wmstkNotificationSender)
            .sendNotification(orderDtoCaptor.capture());

    // arrange
    OrderDto expcetedOrderDto = new OrderDto(1L, "a", "b");

    // assert
    assertThat(orderDtoCaptor.getValue())
            .usingRecursiveComparison()
            .isEqualTo(expcetedOrderDto);
}
```



```java
@Test
void test() {
    // arrange
    OrderEntity orderEntity = getOrderEntity();
    OrderDto expcetedOrderDto = new OrderDto(1L, "a", "b");

    when(orderRepository.findById(1L))
            .thenReturn(Optional.of(orderEntity));

    // action
    orderProcessor.updateOrder(1L, new OrderRequest("a", "b"));

    // assert
    assertAll(
            () -> verify(wmstkNotificationSender)
                    .sendNotification(orderDtoCaptor.capture()),
            () -> assertThat(orderDtoCaptor.getValue())
                    .usingRecursiveComparison()
                    .isEqualTo(expcetedOrderDto)
    );
}
```

Links

- Project on GitHub with all types of tests
- Spring Boot Testing documentation
- Component test auto-configuration annotations
- Improved Testcontainers support in Spring Boot 3
- Another approach to start DB using Testcontainers
- Don-t-use-transactional-in-tests-article