# Testing JPA entities with Spring Boot

Testing the correct behavior of the persistence layer is an occurrence of an **integration test**.

While unit testing concentrates on single classes in order to prove that they are intrinsically correct and requires object mocking in order to decouple a class from its dependencies (which could, in principle, be the reason of the failure, so they are evicted and replaced by mocks that are simple enough to be reliable), integration testing combines different modules of the application, and tests them as a whole in order to assess whether they work correctly or not.

In the persistence case, the DBMS enters the testing scope: this means that it must be properly configured and populated, before performing each test, and properly restored afterwards, whatever the test results happened to be.

Spring Boot supports testing various parts of an application in (partial) isolation, by adding – in front of the test class – one of the following annotations: `@JsonTest`, `@WebMvcTest`, `@DataMongoTest`, `@JdbcTest`, `@DataJpaTest`, …

The two most important annotations are:

- @WebMvcTest – to test your web-layer with MockMvc
- @DataJpaTest – to test your persistence layer

There are also annotations available for more niche-parts of your application:

- @JsonTest – to verify JSON serialization and deserialization
- @RestClientTest – to test the RestTemplate
- @DataMongoTest – to test MongoDB-related code

Whenever a test is run, a TestContext is created: it is a Spring container populated with a subset of the beans declared in the application as a whole. Each of the above-mentioned annotations select a different subset of beans which will be available during the test.

Each annotation supports a set of attributes that provide fine control over what is included and excluded from the bean set, as well as configure some application details in a different way with respect to the standard configuration.

Since some of the application beans are initialized with values coming from the application.properties file, it is possible to override those values which makes no sense in the test, labelling the test class also with @TestPropertySource(locations=[“classpath:test.properties”]).
Here, “test.properties” will refer to a file that contains key/value pairs that override/add to those indicated in the application.properties file, located in a <project>/src subfolder (typically the resource one).

@DataJpaTest starts a sliced Spring context with only relevant JPA components (entities, repositories, EntityManager, …).

By default, this will try to instantiate an in memory H2 embedded DBMS: in order to succeed, the proper Gradle dependency must be included in the build.gradle.kts file:

```
dependencies {
      //other dependencies…
      testImplementation("org.springframework.boot:spring-boot-starter-test")
      testImplementation("com.h2database:h2")
}
```

Moreover, different properties must be setup to properly configure the data-source. They will be stored in the test.properties file:

```
spring.datasource.url=jdbc:h2:mem:test
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.H2Dialect
spring.jpa.hibernate.ddl-auto = create-drop
spring.jpa.properties.hibernate.show_sql = true
spring.jpa.properties.hibernate.format_sql = true
```

If having the database structure automatically derived from the entities does not look convenient (there might be a difference between the database schema during the test and production), it is possible to add – in the <project>/src/test/resources folder, a “schema.sql” file containing the definition of the database schema. In this case, property spring.jpa.hibernate.ddl-auto should be set to “verify”.
In order to populate the test database with some entries, the “data.sql” file – stored in the same folder as above – can be used.

All tests must be stored in the <project>/src/test/kotlin folders.
Tests consists of methods labelled with @Test located in a class labelled with @DataJpaTest.

- Autowiring can be used to inject relevant components.
- Method names should be descriptive of what is tested.
- Methods contain assertions regarding expected behaviors.

Here an example of a test class

```
@DataJpaTest()
@TestPropertySource(locations      =      ["classpath:application-integration-
test.properties"])
class UserTest() {

    @Autowired
    lateinit var userRepository: UserRepository

    @Test
    fun duplicateNamesAreRejected() {
        val u1a =
           User("u1","p1","u1a@gmail.com",false,Rolename.CUSTOMER.toString())
        userRepository.save(u1a)
        val u1b =
           User("u1","p1","u1b@gmail.com",false,Rolename.CUSTOMER.toString())
```

```
        userRepository.save(u1b)
        assertThrows<Exception> { userRepository.flush() }
    }

    @Test
    fun duplicateEmailsAreRejected() {
        val u1a =
          User("u1a","p1","u1@gmail.com",false,Rolename.CUSTOMER.toString())
        userRepository.save(u1a)
        val u1b =
          User("u1b","p1","u1@gmail.com",false,Rolename.CUSTOMER.toString())
        userRepository.save(u1b)
        assertThrows<Exception> { userRepository.flush() }
    }
}
```

## Further references

- https://reflectoring.io/spring-boot-data-jpa-test/
- https://rieckpil.de/test-your-spring-boot-jpa-persistence-layer-with-datajpatest/
- https://www.baeldung.com/spring-boot-testing