POJO / JavaBeans testing

The first reaction to the question “Should I unit test a data class?” the answer is usually no because there is no big meaning to check, for example, the getter/setter behaviour, especially if they have been generated by the IDE or by libraries like Lombok and Immutables.

Real world projects are, unfortunately, not so perfect and there are cases where unit testing a POJO or JavaBean could make sense. Let me list some situations I have encountered.

1. Security sensitive fields

If the POJO contains a security sensitive field, a password, an API token,…, this field should NOT be logged.

See [https://cheatsheetseries.owasp.org/cheatsheets/Logging\_Cheat\_Sheet.html#data-to-exclude](https://cheatsheetseries.owasp.org/cheatsheets/Logging_Cheat_Sheet.html" \l "data-to-exclude)

For this reason, the class should not include sensitive fields in the toString method; a very pratical way to assure this condition is via an unit test.

2. Logging unfriendly fields

The POJO can contain fields which are very long string or binary data. In this case, we should exclude these fields from the toString method to prevent them to bloat our logging files. Again, a good way to assure we are not logging them is via an unit test.

3. Non standard equals and hashcode

The standard way to implement an equals method is to compare, one by one, all fields in the class. Sometimes one field contains a “unique” identifier that makes other fields comparison useless or even dangerous.

For example, the field can contain the primary key column of the database record from which the POJO data has been extracted. In this case, the POJO equality could depend only on its primary key fielda and the equals (and hashCode) implementation should use just this field.

4. No args constructor (JavaBeans)

The JavaBean standard requires the presence of a no-args constructor, a constructor without parameters.

The compiler usually creates the no args constructor unless there are already other (non no-args ) constructors defined in the class.

At runtime, the library (for example, the JPA libraries) which rely on the no-args constructor presence will fail at runtime due to the missing noargs constructor.

One important remark is the following: checking all above conditions using an unit test is way to guarantee the correct behaviour of the data class along all project life. The class can be correctly implemented when created but, later, one developer, while adding a new field in the class, can wrongly regenerate the toString method, restoring the log of a password field, or forget to regenerate the equals/hashCode methods in the proper way.

Checking data class quality is also useful in legacy projects to detect improper classes definitions because unit class can be added aside the “main” classes, without disturbing the legacy code.

The Bean-matchers library let us fix in a test the safety conditions to guarantee that all future changes in the class will not break them.

<https://github.com/orien/bean-matchers>

Let’s assume we have the following data class:

class BasicBean {

private int id;

private String string;

private char[] password;

private Long[] longArray;

private String veryLongString;

Conditions we want to assure:

a. the id field should be used to check equality and to generate the class hash

b. the password field should not be logged because security sensible

c. the veryLongString field should not be logged to avoid logging files bloating

d. the class should have a no-args constructor to be used with JPA.

Using the Bean-matchers library, the test class can be:

@Test

public void testTheClassIsGoodJavaBean() {

MatcherAssert.assertThat(BasicBean.class,

CoreMatchers.allOf(

// This is Java Bean so we want an empty constructor

BeanMatchers.hasValidBeanConstructor(),

// All fields should have getter and setter

BeanMatchers.hasValidGettersAndSetters(),

// Only the 'id' field in hashcode and equals

BeanMatchers.hasValidBeanHashCodeFor("id"),

BeanMatchers.hasValidBeanEqualsFor("id"),

// Password and veryLongString fields should not be included in the toString method

BeanMatchers.hasValidBeanToStringExcluding("password", "veryLongString")

));

}

The Beans-matchers library create an instance of our BasicBean class and checks, by reflection, if all conditions we have defined on

\* the constructor

\* the getters and setters,

\* the equals method

\* the hashCode method

\* the toString method

are valid otherwise the test fails.

A test failure will alert us whenever a change in the BasicBean class breaks the above conditions. It is a kind of safety net.

For Lombok users

Using Lombok, we can implement our bean conditions using Lombok annotations. The following code is the equivalent version of the plain Java version:

@Data

@NoArgsConstructor

@EqualsAndHashCode(of = "id")

@ToString(exclude = {"password", "veryLongString"})

public class LombokBean {

private int id;

private String string;

private char[] password;

private Long[] longArray;

private String veryLongString;

public LombokBean(final int id) {

this.id = id;

}

}

When using Lombok, I believe there is no need to have a unit test to check our bean conditions because, in general, we should not test third party libraries (Lombok in this case), but having it will not damage your project ;-)

Final remarks

Another possible benefit of testing data classes, especially in legacy projects, is that their code coverage is easily 100%. This should not be the main purpose (remember, there is no meaning to test getter and setter) but a side effect which I’ve found useful: having the data classes at 100%, the low test coverage is due to the logic classes, the classes which contains the code which implement the application logic. They are the main target of any test !