Building the code and testing
Module 4.3

Manual testing

Even if test automation has larger potential benefits for DevOps than manual testing, manual testing will always be an important part of software development.

If nothing else, we will need to perform our tests manually at least once in order to automate them.

Pros and cons with test automation

Imagine all the benefits that await us with it:

- Higher software quality
- Higher confidence that the software releases we make will work as intended
- Less of the monotonous tedium of laborious manual testing.

If you just compile programs and deploy them once they pass compilation, you will likely be in for a bad experience. Software testing is completely necessary for a program to work reliably in the real world. Manual testing is too slow to achieve Continuous Delivery. So, we need test automation to succeed with Continuous Delivery. Therefore, let's further investigate the problem areas surrounding test automation and see if we can figure out what to do to improve the situation:

Pros and cons with test automation

- Cheap tests have lower value.
- It is difficult to create test cradles that are relevant to automated integration testing.
- The functionality of programs vary over time and tests must be adjusted accordingly, which takes time and effort.
- It is difficult to write robust tests that work reliably in many different build scenarios.
- It is just hard to write good automated tests, period.

Unit testing

Unit testing is the sort of testing that is normally close at heart for developers. The primary reason is that, by definition, unit testing tests well-defined parts of the system in isolation from other parts.

Many build systems have built-in support for unit tests, which can be leveraged without undue difficulty.

With Maven, for example, there is a convention that describes how to write tests such that the build system can find them, execute them, and finally prepare a report of the outcome.

JUnit

JUnit is a framework that lets you define unit tests in your Java code and run them.

- Test runner:

A test runner runs tests that are defined by an xUnit framework. JUnit has a way to run unit tests from the command line, and Maven employs a test runner called Surefire. A test runner also collects and reports test results. In the case of Surefire, the reports are in XML format, and these reports can be further processed by other tools, particularly for visualization.

• Test case:

A test case is the most fundamental type of test definition. How you create test cases differs a little bit among JUnit versions. In earlier versions, you inherited from a JUnit base class; in recent versions, you just need to annotate the test methods. By convention, Surefire also locates test classes that have the Test suffix in the class name.

RUNNING YOUR TESTS?

https://automationintesting.com/java/junit/lessons/runningtests.html

JUnit

- Test fixtures:

A test fixture is a known state that the test cases can rely on so that the tests can have well-defined behavior. It is the responsibility of the developer to create these. A test fixture is also sometimes known as a test context. With JUnit, you usually use the @Before and @After annotations to define test fixtures. @Before is, unsurprisingly, run before a test case and is used to bring up the environment. @After likewise restores the state if there is a need to.

- Test suites:

You can group test cases together in test suites. A test suite is usually a set of test cases that share the same test fixture.

• Test execution:

A test execution runs the tests suites and test cases. Here, all the previous aspects are combined. The test suites and test cases are located, the appropriate test fixtures are created, and the test cases run. Lastly, the test results are collected and collated.

JUnit

- Test result formatter:

A test result formatter formats test result output for human consumption. The format employed by JUnit is versatile enough to be used by other testing frameworks and formatters not directly associated with JUnit. So, if you have some tests that don't really use any of the xUnit frameworks, you can still benefit by presenting the test results in Jenkins by providing a test result XML file. Since the file format is XML, you can produce it from your own tool, if need be.

• Assertions:

An assertion is a construct in the xUnit framework that makes sure that a condition is met. If it is not met, it is considered an error, and a test error is reported. The test case is also usually terminated when the assertion fails.

JUnit

JUnit has a number of assertion methods available. Here is a sample of the available assertion methods:

• To check whether two objects are equal:

assertEquals(str1, str2);

• To check whether a condition is true:

assertTrue (val1 < val2);

• To check whether a condition is false:

AssertFalse(val1 > val2);

A JUnit example JUnit is well supported by Java build tools. It will serve well as an example of JUnit testing frameworks in general. If we use Maven, by convention, it will expect to find test cases in the following directory:

/src/test/java

Mocking

Mocking refers to the practice of writing simulated resources to enable unit testing. Sometimes, the words "fake" or "stub" are used. For example, a middleware system that responds with JSON structures from a database would "mock" the database backend for its unit tests. Otherwise, the unit tests would require the database backend to be online, probably also requiring exclusive access. This wouldn't be convenient.

Mockito is a mocking framework for Java that has also been ported to Python.

Test Coverage

Test coverage is the percentage of the application code base that is actually executed by the test cases.

- In order to measure unit test code coverage, you need to execute the tests and keep track of the code that has or hasn't been executed. Cobertura is a test coverage measurement utility for Java that does this. Other such utilities include jcoverage and Clover.
- Cobertura works by instrumenting the Java bytecode, inserting code fragments of its own into already compiled code. These code fragments are executed while measuring code coverage during execution of test cases
- Its usually assumed that a hundred percent test coverage is the ideal.

A complete test automation scenario

Assembling the pieces into a cohesive whole can be daunting. In this section, we will have a look at a complete test automation example, continuing from the user database web application for an organization, Matangle.

The application consists of the following layers:

- **A web frontend:**
- **A JSON/REST service interface**
- **An application backend layer**
- **A database layer**

The test code will work through the following phases during execution:

- **Unit testing of the backend code**
- **Functional testing of the web frontend, performed with the Selenium web testing framework**
- **Functional testing of the JSON/REST interface, executed with soapUI**

All the tests are run in sequence, and when all of them succeed, the result can be used as the basis for a decision to see whether the application stack is deemed healthy enough to deploy.

SoapUI: https://www.guru99.com/soapui-tutorial.html