# Maven Tests

## Maven Test

When you execute `mvn test` it will compile and test the source code that's underneath the source code folder!

Maven Test allows integration with JUnit, TestNG, and POJO. We will go over JUnit because that's what we use most of the time.

## JUnit

The popular testing framework for Java. Most of the test are written in JUnit 4 but the latest version is JUnit 5.

Writing them are still the same, however, integrating it with behavioral driven testing framework like Cucumber will be different I will go over the differences for using cucumber with JUnit 4 and JUnit 5 (Mainly in the annotation that is used to specify the glue and feature files)

When you write your unit tests using JUnit you want to structure your file directory like the following:

```
src
├── main
│   ├── java
│   │   └── test
│   │       └── App.java
│   └── resources
│       └── application.properties
└── test
    ├── java
    │   └── test
    │       └── AppTest.java
    └── resources
        └── test.json
```

Your main source code directory will contain two different directories.

- The `main` directory contain the Java source code for your application as well as any resources that your application uses
- The `test` directory contain the Java source code used for testing. This is where you will be writing your unit test for your applications

When you run `mvn test` it will compile the source code from the `src` directory and execute the proper unit test using the JUnit framework.

### Dependency needed for JUnit 5

```
<dependencies>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.9.1</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

# Test Class

A test class is a class that will be used by Maven to carry out the testing.

A test class contains methods that will be tested by Maven. Maven use a pattern to check whether a class is a test class or name, the list of pattern used by Maven to determine if a class is a test class is of the following:

```
**/Test*.java
**/*Test.java
**/*Tests.java
**/*TestCase.java
```

# Test Runner Class

> This applies to JUnit 4 test. JUnit 5 has a new API and @RunWith is no longer needed.

A test runner in JUnit is just a simple class that implements the `Describable` interface. The interface just have two abstract methods to implement:

```
public abstract class Runner implements Describable {
  public abstract Description getDescription();
```

```
  public abstract void run(RunNotifier notifier);
}
```

1. getDescription: Is used to return a description containing the information about the test, this is used by different tools such as those from IDE
2. run: This is used to actually run the test class or test suite

You can implement your own test runner to do custom logic for running tests, for example you can print out silly messages just before each test is run.

A simple test runner class implementation can be found below: All it does is before it runs the test class is it prints the message "MyRunner <TestClass>"

**Eample Test Runner**

```
//src/main/java/MinimalRunner
package com.codingninjas.testrunner;



import org.junit.runner.Description;
import org.junit.runner.Runner;
import org.junit.runner.notification.RunNotifier;
import org.junit.Test;
import java.lang.reflect.Method;




public class MinimalRunner extends Runner {



    private Class testClass;
    public MinimalRunner(Class testClass) {
        super();
        this.testClass = testClass;
    }

    //getDescription method inherited from Describable
    @Override
    public Description getDescription() {
        return Description
```

```
            .createTestDescription(testClass, "My runner description");
    }

    //our run implementation
    @Override
    public void run(RunNotifier notifier) {
        System.out.println("running the tests from MyRunner: " + testClass);
        try {
            Object testObject = testClass.newInstance();
            for (Method method : testClass.getMethods()) {
                if (method.isAnnotationPresent(Test.class)) {
                    notifier.fireTestStarted(Description
                      .createTestDescription(testClass, method.getName()));
                    method.invoke(testObject);
                    notifier.fireTestFinished(Description
                      .createTestDescription(testClass, method.getName()));
                }
            }
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

To actually use this example Test Runner you would then just simply annotate your test class with `@RunWith(MinimalRunner.class)` which will delegate the responsibility of running this particular test class to that custom test runner.

```
//src/test/java/CalculatorTest.java
package com.codingninjas.testrunner;


import static org.junit.Assert.assertEquals;
import org.junit.Test;
import org.junit.runner.RunWith;


@RunWith(MinimalRunner.class)
public class CalculatorTest {
```

```
    Calculator calculator = new Calculator();



    @Test
    public void testAddition() {
        System.out.println("in testAddition");
        assertEquals("addition", 8, calculator.add(5, 3));
    }
}
```

> Note that this runner mechanism is no longer present in JUnit 5 and is replaced by Jupiter extension. https://stackoverflow.com/a/59360733/7238625

### Default Test Runner class

For Junit 4 at least, if you don't specify a test runner, JUnit will default to the default `BlockJUnit4ClassRunner` which as you know it will just simply executes the test class.

### Specify one test class to execute

If you only want to run one of the test class then you can run maven test with the following option

```
 mvn test -Dtest="TestFoo"
```

This will only execute the `TestFoo` class.

### Specify one method to execute from a test class

If you only want to execute one of the test then you can run maven test with the following option

```
mvn test -Dtest="TestFoo#testFoo"
```

The left of # denotes the test class, and the right of # denotes the method that you want to execute.

# Cucumber + JUnit 4

> Note that the following code only works with JUnit 4

Cucumber is a test automation tool that follows BDD. You describe the things you want to test in something called feature files which uses Gherkin language specification. This is so that stakeholders who doesn't understand code can also look at the feature file and understand the

behavior of your application without needing to learn the programming language.

Underneath cucumber will then translate the feature files into actual code.

Cucumber framework in Java works by implementing it's own test runner class with JUnit, this is how it makes it work, and why the test class using cucumber test doesn't have anything in them. In the CucumberOptions annotation you will be specifying the feature files you will be looking at, the glue it is using, the specific tags to run, and plugin options.

**The list of sample dependency to use Cucumber with JUnit will be the following:**

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.13.2</version>
  <scope>test</scope>
</dependency>

<dependency>
  <groupId>io.cucumber</groupId>
  <artifactId>cucumber-java</artifactId>
  <version>7.14.0</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>io.cucumber</groupId>
  <artifactId>cucumber-junit</artifactId>
  <version>7.14.0</version>
  <scope>test</scope>
</dependency>
```

## Terminology

1. Glue: A glue is used to associate the Gherkin steps (i.e. what's written in feature files) to a step definition method (i.e. actual code that gets executed). The Gherkin step is "glued" to the step definition. If this is used in the context of the test class, then it is referring to the package that your step definition class lives under.
2. Features: A feature file contains the description of the test in Gherkin. This is what cucumbers look at to execute the actual test. It is meant to be easy to read for stakeholders in plain English so even if they don't understand programming language they can interpret the test cases as well.
3. Plugin: Reporting plugins that can be used to output the result of the cucumber test into different formats. Json and html output, you can also use third party plugins as well!

## Writing cucumber test

Your file structure should look like this:

```
src
└── test
    ├── java
    │   └── com
    │       ├── lol
    │       │   └── StepDefinition.java
    │       ├── OneMoreStep.java
    │       └── TestingFoo.java
    └── resources
        └── features
            └── foo.feature
```

Your feature files will be living under the `resources` folder. Your step definition will be living under the Java folder. Honestly, whether or not it is in another package just depend on you, you will be specifying where the step definitions are as part of glue configuration anyway. If you don't specify glue it will look for it under the test class.

Your test class will need to be annotated with @RunWith to tell JUnit that it will be the responsibility of the Cucumber test runner to run the test.

```
package com;


import org.junit.runner.RunWith;


import io.cucumber.junit.Cucumber;
import io.cucumber.junit.CucumberOptions;


@RunWith(Cucumber.class)
@CucumberOptions(
   features = "src/test/resources/features",
   glue = {"com"},
   plugin = {"html:target/out.html"}
)
public class TestingFoo {
}
```

# Cucumber + JUnit 5

Now because JUnit 5 has changed a lot of stuff, what worked for JUnit 4 will no longer work with JUnit 5 if you want to integrate cucumber with it. Here are the steps on how to do it.

Include the following dependency:

```
<dependencies>
  <dependency>
    <groupId>org.junit.platform</groupId>
    <artifactId>junit-platform-suite-engine</artifactId>
    <version>1.10.0</version>
  </dependency>
  <dependency>
    <groupId>io.cucumber</groupId>
    <artifactId>cucumber-java</artifactId>
    <version>7.14.0</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>io.cucumber</groupId>
    <artifactId>cucumber-junit-platform-engine</artifactId>
    <version>7.14.0</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

The test class will then have to have these following configurations

```
package com;

import org.junit.platform.suite.api.ConfigurationParameter;
import org.junit.platform.suite.api.IncludeEngines;
import org.junit.platform.suite.api.SelectClasspathResource;
import org.junit.platform.suite.api.Suite;



import static io.cucumber.junit.platform.engine.Constants.GLUE_PROPERTY_NAME;

@Suite
@IncludeEngines("cucumber")
@SelectClasspathResource("features")
@ConfigurationParameter(
```

```
    key = GLUE_PROPERTY_NAME,
    value = "com"
)
public class TestingFoo {


}
```

As you can see although the annotation differs a bit, it is still relatively the same.

## Multiple Runners

If for some reason your project is configured to have multiple test classes, remember JUnit will scan all of the test classes underneath the test directory, and you only want to run say one of the test classes. Then you can use the option flag that was reference above for the JUnit section :smile:

---

Revision #7
Created 2023-10-07 15:46:36 UTC by Tamarine
Updated 2023-10-07 22:23:51 UTC by Tamarine