Junit

What is JUnit ?

JUnit is a unit testing framework for the Java programming language. **JUnit** has been important in the development of test-driven development, and is one of a family of unit testing frameworks collectively known as xUnit that originated with **JUnit**.

Testing is the process of checking the functionality of the application whether it is working as per requirements and to ensure that at developer level, unit testing comes into picture. Unit testing is the testing of single entity (class or method). Unit testing is very essential to every software company to give a quality product to their customers.

Unit testing can be done in two ways
1.Manual testing
2.Automated testing

1.Manual testing:
Executing the test cases manually without any tool support is known as manual testing.

2.Automated testing:
aking tool support and executing the test cases by using automation tool is known as automation testing.

Features:

- JUnit is an open source framework which is used for writing & running tests.
- Provides Annotation to identify the test methods.
- Provides Assertions for testing expected results.
- Provides Test runners for running tests.
- JUnit tests allow you to write code faster which increasing quality
- JUnit is elegantly simple. It is less complex & takes less time.
- JUnit tests can be run automatically and they check their own results and provide immediate feedback. There's no need to manually comb through a report of test results.

What is a Unit Test Case ?

A Unit Test Case is a part of code which ensures that the another part of code (method) works as expected. To achieve those desired results quickly, test framework is required .JUnit is perfect unit test framework for java programming language.

A formal written unit test case is characterized by a known input and by an expected output, which is worked out before the test is executed. The known input should test a precondition and the expected output should test a postcondition.

Example1:

```
import org.junit.Test;
import junit.framework.TestCase;

public class FirstClass extends TestCase  {

    protected int v1,v2;


    protected void setUp(){
        v1=2;
        v2=3;

    }
    @Test
    public void testAdd(){
          double result= v1 + v2;
          //assertTrue(result == 6);

          assertEquals(5, 5);
       }
```

JUnit test framework provides following important features:

Fixtures:

Fixtures is a fixed state of a set of objects used as a baseline for running tests. The purpose of a test fixture is to ensure that there is a well known and fixed environment in which tests are run so that results are repeatable. It includes

- setUp() method which runs before every test invocation.
- tearDown() method which runs after every test method.

```
public class JavaTest extends TestCase {
   protected int value1, value2;

   // assigning the values
   protected void setUp(){
      value1=3;
      value2=3;
   }

   // test method to add two values
   public void testAdd(){
      double result= value1 + value2;
      assertTrue(result == 6);
   }
}
```

Test suite:

Test suite means bundle a few unit test cases and run it together. In JUnit, both @RunWith and @Suite annotation are used to run the suite test. Here is an example which uses TestJunit1 & TestJunit2 test classes.

Test runner:

Test runner is used for executing the test cases and if it is success or fails.

Example1:

```
public class TestRunner {
   public static void main(String[] args) {
```

```
      Result result =
JUnitCore.runClasses(TestJunit.class);
      for (Failure failure : result.getFailures()) {
         System.out.println(failure.toString());
      }
      System.out.println(result.wasSuccessful());
   }
}
```

JUnit classes:

JUnit classes are important classes which is used in writing and testing JUnits. Some of the important classes are

- Assert which contain a set of assert methods.
- TestCase which contain a test case defines the fixture to run multiple tests.
- TestResult which contain methods to collect the results of executing a test case

Assert Class:

Following is the declaration for **org.junit.Assert** class:

```
public class Assert extends java.lang.Object
```

This class provides a set of assertion methods useful for writing tests. Only failed assertions are recorded. Some of the important methods of **Assert** class are:

Methods & Description:

1.void assertEquals(boolean expected, boolean actual)

Check that two primitives/Objects are equal

2.void assertFalse(boolean condition)

Check that a condition is false

3.void assertNotNull(Object object)

Check that an object isn't null.

4.Void fail()

Fails a test with no message.

Example:

```
public class TestJunit1 {

   @Test
   public void testAdd() {
      //test data
      int num= 5;
      String temp= null;
      String str= "Junit is working fine";

      //check for equality
      assertEquals("Junit is working fine", str);

      //check for false condition
      assertFalse(num > 6);

      //check for not null value
      assertNotNull(str);
   }
}
```

Next, let's create a java class file name TestRunner1.java

```
public class TestRunner1 {

   public static void main(String[] args) {
      Result result =
JUnitCore.runClasses(TestJunit1.class);
      for (Failure failure : result.getFailures()) {
         System.out.println(failure.toString());
      }
      System.out.println(result.wasSuccessful());
   }
}
```

Annotation:

Annotations are like meta-tags that you can add to you code and apply them to methods or in class. These annotation in JUnit gives us information about test methods , which methods are going to run before & after test methods, which methods run before & after all the methods, which methods or class will be ignore during execution.

List of annotations and their meaning in JUnit :

1.@Test

The Test annotation tells JUnit that the public void method to which it is attached can be run as a test case.

2.@**Before**

Several tests need similar objects created before they can run. Annotating a public void method with @Before causes that method to be run before each Test method.

3.**@After**

If you allocate external resources in a Before method you need to release them after the test runs. Annotating a public void method with @After causes that method to be run after the Test method.

4.@**Ignore**

The Ignore annotation is used to ignore the test and that test will not be executed.

5.@BeforeClass annotation specifies that method will be invoked only once, before starting all the tests.

6.@AfterClass annotation specifies that method will be invoked only once, after finishing all the tests.

Example:

```
public class JunitAnnotation {

   //execute only once, in the starting

   @BeforeClass
   public static void beforeClass() {
      System.out.println("in before class");
   }

   //execute only once, in the end

   @AfterClass
   public static void  afterClass() {
      System.out.println("in after class");
   }

  //execute for each test, before executing test

   @Before
   public void before() {
      System.out.println("in before");
   }

   //execute after test
   @After
   public void after() {
      System.out.println("in after");
   }

   //test case
   @Test
```

```
   public void test() {
      System.out.println("in test");
   }

   //test case ignore and will not execute
   @Ignore
   public void ignoreTest() {
      System.out.println("in ignore test");
   }
```

First of all beforeClass() method execute only once

- Lastly, the afterClass() method executes only once.
- before() method executes for each test case but before executing the test case.
- after() method executes for each test case but after the execution of test case
- In between before() and after() each test case executes.

A test method annotated with @Ignore will not be executed.

- If a test class is annotated with @Ignore then none of its test methods will be executed.

Junit provides a handy option of Timeout. If a test case takes more time than specified number of milliseconds then Junit will automatically mark it as failed. The **timeout** parameter is used along with @Test annotation. Now let's see @Test(timeout) in action.

Create Test Case Class

- Create a java test class say TestJunit.java.
- Add timeout of 1000 to testPrintMessage() test case.

Example:

```
public class TestJunit {

   String message = "Robert";
   MessageUtil messageUtil = new MessageUtil(message);

   @Test(timeout=1000)
   public void testPrintMessage() {
      System.out.println("Inside testPrintMessage()");
      messageUtil.printMessage();
   }

   @Test
   public void testSalutationMessage() {
      System.out.println("Inside
testSalutationMessage()");
      message = "Hi!" + "Robert";

assertEquals(message,messageUtil.salutationMessage());
   }
```

In hybris we write all TestCases in testsrc folder or web/testsrc.

DefaultTestCase in hybris:

Example:

```
public class YcommercewebservicestestTest extends HybrisJUnit4TransactionalTest

{
	/** Edit the local|project.properties to change logging behaviour (properties
log4j.*). */
	@SuppressWarnings("unused")
```

```
	private static final Logger LOG =
Logger.getLogger(YcommercewebservicestestTest.class.getName());

	@Before
	public void setUp()
	{
		// implement here code executed before each test
	}

	@After
	public void tearDown()
	{
		// implement here code executed after each test
	}

	/**
	 * This is a sample test method.
	 */
	@Test
	public void testYcommercewebservicestest()
	{
		final boolean testTrue = true;
		assertTrue("true is not true", testTrue);
	}
```

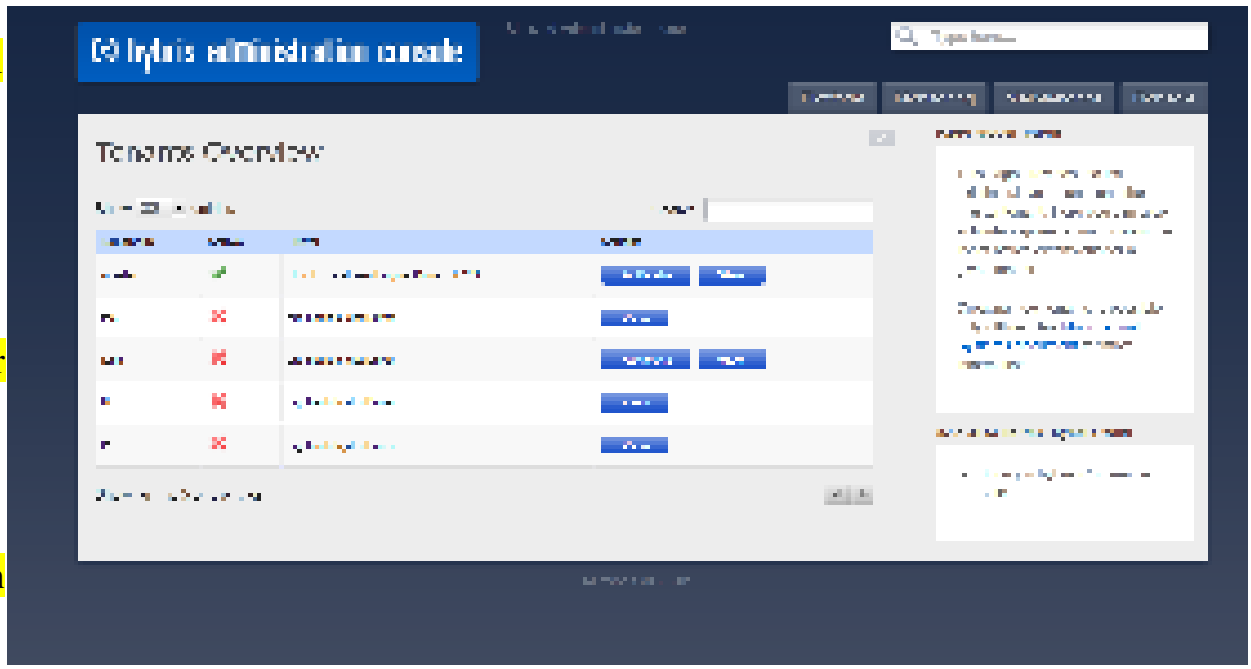
Initialize the junit tenant in hybris:

In this trail you will create an integration test, which unlike a unit test requires access both to the database and the hybris Platform environment.
This can easily be done by using the preconfigured junit tenant in which you can execute your JUnit tests isolated from other tenants. This enables you to test business functionality using real item instances.

Initialize the junit tenant as follows:

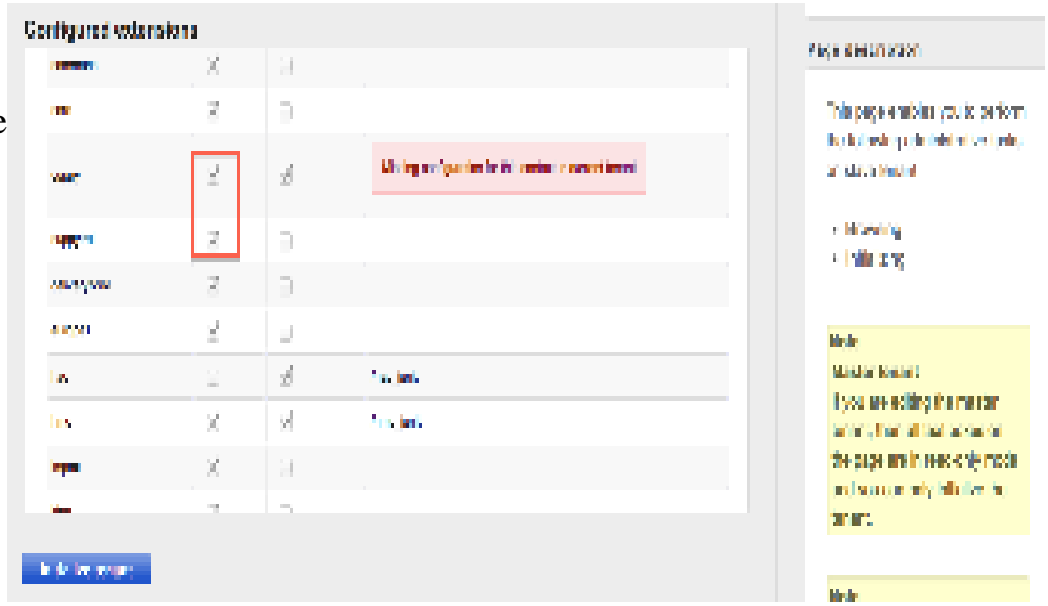
- Log in into http://localhost:9001 as admin
- Click on the tenant-link (Platform/Tenants)

Click on "View" under TenantID junit



- Make sure, that you check cuppy and cuppytrail and then click "Initialize Tenant"

Click on "initialize tenant", deselect Project data settings (we don't need them for junit tenant) and start the initialization process



- As you can see in the upper left corner, you're now in the junit tenant. Wait until the initialization has finished and switch back to master tenant. Make sure that both tenants are active.

Another way of initializing the junit tenant is to run the **ant yunitinit** target.

DAOIntegrationTest.java

```
public class DefaultStadiumDAOIntegrationTest extends ServicelayerTransactionalTest
{
   /** As this is an integration test, the class (object) being tested gets injected here. */
   @Resource
   private StadiumDAO stadiumDAO;

   /** Platform's ModelService used for creation of test data. */
   @Resource
   private ModelService modelService;

   /** Name of test stadium. */
   private static final String STADIUM_NAME = "wembley";

   /** Capacity of test stadium. */
   private static final Integer STADIUM_CAPACITY = Integer.valueOf(12345);

   @Test
   public void stadiumDAOTest()
   {
      List<StadiumModel> stadiumsByCode =
stadiumDAO.findStadiumsByCode(STADIUM_NAME);
      assertTrue("No Stadium should be returned", stadiumsByCode.isEmpty());

    List<StadiumModel> allStadiums = stadiumDAO.findStadiums();
         final int size = allStadiums.size();

         final StadiumModel stadiumModel = new StadiumModel();
         stadiumModel.setCode(STADIUM_NAME);
         stadiumModel.setCapacity(STADIUM_CAPACITY);
         modelService.save(stadiumModel);

         allStadiums = stadiumDAO.findStadiums();
         assertEquals(size + 1, allStadiums.size());
         assertEquals("Unexpected stadium found", stadiumModel,
allStadiums.get(allStadiums.size() - 1));

         stadiumsByCode =
```

```
stadiumDAO.findStadiumsByCode(STADIUM_NAME);
        assertEquals("Did not find the Stadium we just saved", 1,
stadiumsByCode.size());
        assertEquals("Retrieved Stadium's name attribute incorrect",
                        STADIUM_NAME,
stadiumsByCode.get(0).getCode());
        assertEquals("Retrieved Stadium's capacity attribute
incorrect",
                        STADIUM_CAPACITY,
stadiumsByCode.get(0).getCapacity());
    }
```

Run the test

1. Right-click **cuppytrail/testsrc/de/hybris/platform/cuppytrail/daos/impl/DefaultStadiumDAOIntegrationTest.java** in Eclipse's Package Explorer and select RunAs|JunitTest