Automation Testing

Extending JUnit

A Day in the Life of an Automation Test Engineer

Marcelo has completed his task to create and implement the concept of Dynamic Tests and Parameterized Tests.

To complete his project, Marcelo had to understand the extensions, which are used to extend the behavior of test classes or methods, and can be reused for multiple tests.

To achieve the above, he will learn a few concepts in his lesson that can help him to find a solution for the scenario.

Learning Objectives

By the end of this lesson, you will be able to:

- Explain what is Extending JUnit
- Identify the use of Extension Point
- Recognize the different Parameter Injections
- Analyze the meta-annotations in JUnit

Introduction to Extending JUnit

Introduction to Extending JUnit

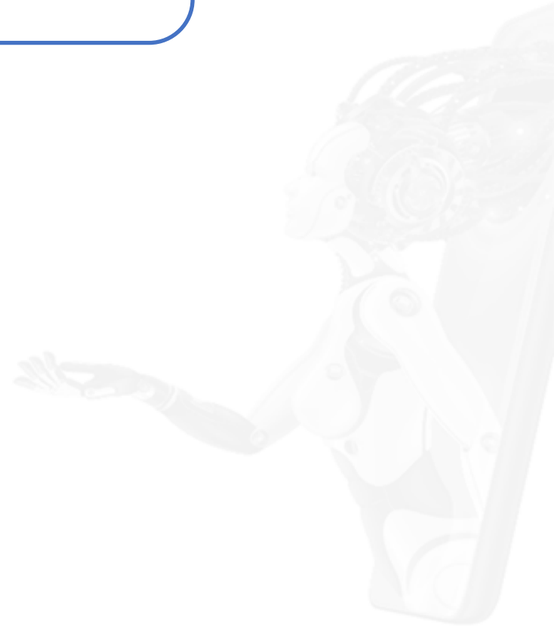
- JUnit 5 extensions are related to a certain event in the execution of a test, referred to as an extension point.
- When a certain life cycle phase is reached, the JUnit engine calls registered extensions.
- JUnit 5 extensions allow for the extension of the behavior of test classes and methods.
- These extensions are typically used for adding additional information to test methods and resolving parameters.

Introduction to Extending JUnit

In JUnit 5, customizing the framework generally meant using a @RunWith annotation to specify a custom runner. Using multiple runners was problematic and usually required chaining or using a @Rule. This has been simplified and improved in JUnit 5 using extensions.

Here is an example for building tests:

```
@RunWIth(SpringJUnit5ClassRunner.class)
 Public class MyControllerTest
 {
  //…
 }
```



Extension Points

Extension Points

Extension points that are available in JUnit 5:

01 Test Instance Post-processing

02 Conditional Test Execution

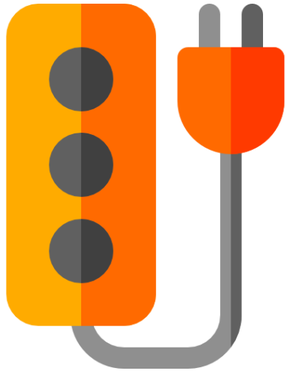
03 Life-Cycle Callbacks

04 Parameter Resolution

05 Exception Handling

Test Instance Post-processing

They are a type of extension which executed after an instance of a test has been created. The interface to implement is TestInstancePostProcessor which has a postProcessTestInstance() method to override.



Example of Test Instance Post-processing

```
public class LoggingExtension implements TestInstancePostProcessor
 {

    @Override
    public void postProcessTestInstance(Object testInstance,
      ExtensionContext context) throws Exception
      {
        Logger logger = LogManager.getLogger(testInstance.getClass());
        testInstance.getClass()
          .getMethod("setLogger", Logger.class)
          .invoke(testInstance, logger);
      }
  }
                                          }
```

Conditional Test Execution

JUnit 5 provides an extension that can control whether or not a test should be run. It is defined by implementing the ExecutionCondition interface.

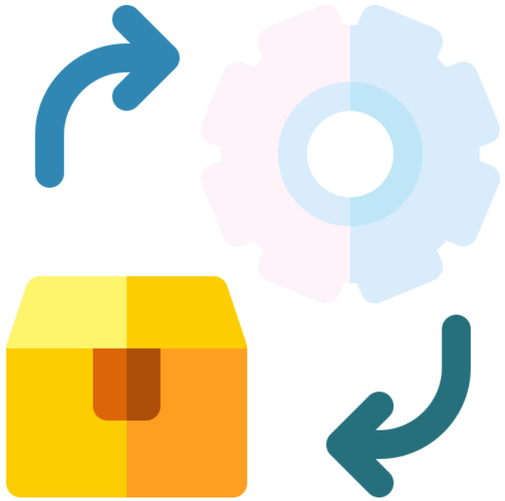
Example of Conditional Test Execution

```
public class EnvironmentExtension implements ExecutionCondition
 {
    @Override
    public ConditionEvaluationResult evaluateExecutionCondition(
      ExtensionContext context)
     {
        Properties props = new Properties();
        props.load(EnvironmentExtension.class
        .getResourceAsStream("application.properties"));
        String env = props.getProperty("env");
        if ("qa".equalsIgnoreCase(env))
         {
            return ConditionEvaluationResult
           .disabled("Test disabled on QA environment");
         }

        return ConditionEvaluationResult.enabled(
          "Test enabled on QA environment");
     }
 }
```

Life-Cycle Callbacks

This set of extensions is related to events in a test's lifecycle and can be defined by implementing the following interfaces:



- **BeforeAllCallback and AfterAllCallback:** Executed before and after all the test methods
- **BeforeEachCallBack and AfterEachCallback:** Executed before and after each test method
- **BeforeTestExecutionCallback and AfterTestExecutionCallback:** Executed immediately before and immediately after a test method

Example of Life-Cycle Callbacks

Here is an example of Life-cycle Callbacks with an Employee entity database using JDBC:

```
public class Employee
 {
   private long id;
    private String firstName;
 }
public class JdbcConnectionUtil
 {
    private static Connection con;
    public static Connection getConnection()
    throws IOException, ClassNotFoundException, SQLException
      {
        if (con == null)
        {
            // create connection
            return con;
        }
        return con;
     }
 }
```

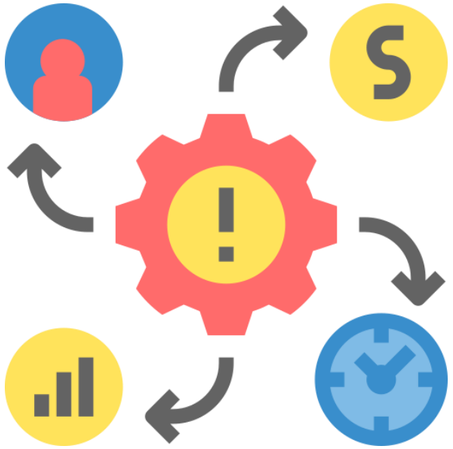
Example of Life-Cycle Callbacks

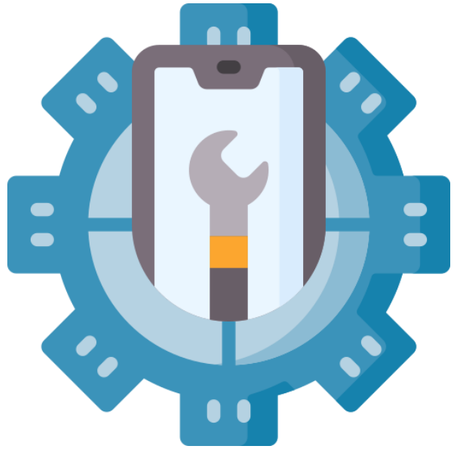
Here is an example of Life-cycle Callbacks with an Employee entity database using JDBC:

```
public class EmployeeJdbcData
 {
    private Connection con;
    public EmployeeJdbcData(Connection con)
    {
        this.con = con;
    }
    public void createTable() throws SQLException
    {
        // create employees table
    }
    public void add(Employee emp) throws SQLException
    {
       // add employee record
    }
    public List<Employee> findAll() throws SQLException
    {
       // query all employee records
    }
}
```

Parameter Resolution

If a test constructor or method receives a parameter, it must be resolved at runtime by a ParameterResolver.





Example of Parameter Resolution

```
public class EmployeeDaoParameterResolver implements ParameterResolver
{
    @Override
    public boolean supportsParameter(ParameterContext parameterContext,
      ExtensionContext extensionContext) throws ParameterResolutionException
      {
        return parameterContext.getParameter().getType()
          .equals(EmployeeJdbcDao.class);
      }
}
```

Exception Handling

The TestExecutionExceptionHandler interface can be used to define the behavior of a test when encountering certain types of exceptions (runtime errors).

Example of Exception Handling

```
public void handleTestExecutionException(ExtensionContext context,
      Throwable throwable) throws Throwable
 {
        if (throwable instanceof FileNotFoundException)
        {
            logger.error("File not found:" + throwable.getMessage());
            return;
        }

        throw throwable;
 }
```

Parameter Injections

Parameter Injections

- Parameter Injections define the API for test extensions that dynamically resolve parameters at runtime.
- If a test class constructor, a test method, or a lifecycle method accepts a parameter, the parameter must be resolved at runtime by a registered ParameterResolver.
- Users can inject as many parameters as they want in any order they want them to be.
- Currently, three built-in resolvers are there in Parameter Injections. Other parameter resolvers must be explicitly enabled by registering appropriate extensions via @ExtendWith.

Parameter Injections

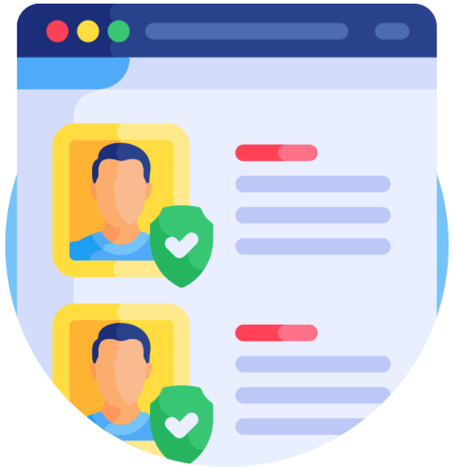
The Parameter Resolvers are:

TestInfoParameterResolver

TestReporterParameterResolver

RepetitionInfoParameterResolver

TestInfoParameterResolver



If a constructor or method parameter is of type TestInfo, the TestInfoParameterResolver supplies an instance of TestInfo. This instance corresponds to the current container or test as the value for the parameter.

Example of TestInfoParameterResolver

```
Class TestInfo Test
{
  TestInfoTest(TestInfo testInfo)
  {
   assertEquals(“TestInfoTest”, TestInfo.getDisplayName());
  }
  @BeforeEach
  Void setup(TestInfo testInfo)
  {
   String displayName = testInfo.getDisplayName();
   assertTrue(displayName.equals)(“display name of the method”) ||
   displayName.equals(“testGetNameOfTheMethod(TestInfo)”));
  }
 }
```

TestReporterParameterResolver

- The constructor or method parameter is of the type TestReporter, the TestReporterParameterResolver supplies an instance of TestReporter.
- TestReporter is a functional interface and can be used as the assignment target for a lambda expression or method reference.
- Parameters of type TestReporter can be injected into methods of test classes annotated with @BeforeEach, @AfterEach, and @Test.
- The TestReporter can be used to publish additional data about the current test run.

Example of TestReporterParameterResolver

```
Class TestReporter Test
{
  @Test
  void testReporterSingleValue(TestReporter testReproter)
  {
   testReporter.publishEntry("Single value");
  }
  @Test
  void testReporterValuePair(TestReporter testReproter)
  {
   testReporter.publishEntry("Key", "value");
  }
  @Test
  void testReportMultipleKeyValuePairs(TestReporter testReproter)
  {
   Map<String, String> values= new HAshMAp<>();
   values.put("user", "Marcelo");
   values.put("password", "RealMCF");
   testReporter.publishEntry(values);
  }
}
```

RepetitionInfoParameterResolver

- RepetitionInfo can be used to retrieve information about the current repetition and the total number of repetitions for the corresponding @RepeatedTest.
- If a method parameter in a @RepeatedTest, @BeforeEach, or @AfterEach method is of type RepetitionInfo, theRepetitionInfoParameterResolver supplies an instance of RepetitionInfo.

Example of RepetitionInfoParameterResolver

```
public class Junit5_RepetitionInfo_Test
{
  @BeforeEach
  void beforeEach(TestInfo testInfo, RepetitionInfo repetitionInfo)
   {
        int currentRepetition = repetitionInfo.getCurrentRepetition();
        int totalRepetitions = repetitionInfo.getTotalRepetitions();
        String methodName = testInfo.getTestMethod().get().getName();
        System.out.println(String.format("About to execute repetition %d of %d for %s", //
                currentRepetition, totalRepetitions, methodName));
    }

  @RepeatedTest(3)
  void test_Add(RepetitionInfo repetitionInfo)
  {
    System.out.println("start test_Add() : "+repetitionInfo.getCurrentRepetition());
    assertEquals(5, MathUtil.add(3, 2));
  }
}
```

Meta Annotations

Meta Annotations

The meta-annotations is an annotations that can be applied to another annotation. That means that the user can now define their custom annotations that are an combinantion of many Spring annotations into one.

Example of Meta Annotations

```
package com.sample.app;

import static org.junit.jupiter.api.Assertions.assertTrue;

public class MetaAnnotationDemo
{
  @SpeedTest
  public void speedTest1()
 {
    assertTrue(true);
  }

}
```

Key Takeaways

- Test Instance Post-processing is a type of extension which executed after an instance of a test has been created.
- JUnit 5 extensions allow for extend of the behavior of test classes and methods.
- The TestReporter can be used to publish additional data about the current test run.
- Meta Annotations is an annotations that can be applied to another annotation.