Adobe, 345 Park Avenue, San Jose, California 95110, USA.

04/20/2021

Contents

Writing JUnit Tests

Introduction

In Adobe Experience Manager (AEM), the testing frameworks, such as Maven, Mockito, and Sling JUnit help automate the testing process. Unit tests in AEM can be set up and run very quickly outside any container. The idea of writing tests in AEM is to detect defects as early as possible, ultimately reducing cost.

Unit testing is the process where the application is divided into units, and each unit is individually tested for its successful operation.

Objectives

After completing this course, you will be able to:

- Describe testing framework
- Explain the different types of testing
- Run unit tests and functional tests on your project
- Create unit tests using Mockito
- Create unit tests using Sling Mocks
- Create unit tests using AEM Mocks

Understanding Testing Frameworks

Software testing is done to ensure that the system performs as expected. The results of a test are compared with the expected outcomes to validate the functionalities of the system. Some of these tests can be extensive and repetitive. In such cases, you can use testing frameworks to automate the testing process.

A testing framework is a system with a set of rules for the automation of software testing. This system makes use of function libraries, test data sources, object details, and various reusable modules.

The Mockito Framework

Mockito is an open source testing framework that allows the creation of mock objects in automated unit tests. Mock testing frameworks effectively replicate some external dependencies so that the object being tested has a consistent interaction with its outside dependencies. Mockito intends to streamline the delivery of these external dependencies that are not subjects of the test.

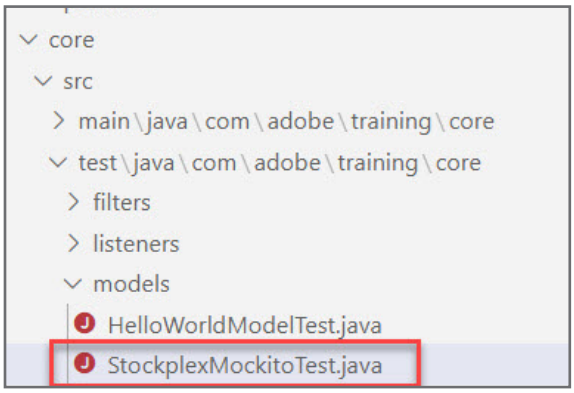
Features of Mockito:

- Mocks concrete classes as well as interfaces
- Verification errors are clean—click on stack trace to see failed verification in test. Click on an exception's cause to navigate to actual interaction in code. Stack trace is always clean.
- Allows flexible verification in order
- Supports exact-number-of-times and at-least-once verification
- Flexible verification or stubbing using argument matchers
- Allows for the creation of custom argument matchers or using existing hamcrest matchers

Exercise 1: Create unit tests using Mockito

In this lab exercise, you will create a unit test and test the Stockplex java class you created earlier. This is a basic unit Test outside the server. If you do not have the Stockplex.java class implemented, then you will need to go back and perform those exercises first.

1. In your IDE, navigate to **core** > **src** > **test/java/com/adobe/training/core/models**.
2. Right-click **models** and select **New File**.
3. Name the file as **StockplexMockitoTest.java**.



4. Copy the contents from **Exercise_Files-EC** under **/core/src/test/java/com/adobe/training/core/models/StockplexMockitoTest.java** to **StockplexMockitoTest.java** in your IDE. The **StockplexMockitoTest.java** class is created.
5. Verify the code, as shown:

```
package com.adobe.training.core.models;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.junit.jupiter.MockitoExtension;
import java.util.Random;
import static org.junit.jupiter.api.Assertions.assertNotNull;
import static org.junit.jupiter.api.Assertions.assertTrue;
import static org.junit.jupiter.api.Assertions.assertFalse;
import static org.mockito.Mockito.*;

/**
 * Tests the Stockplex model using the Mockito testing framework.
 * This is good for simple test cases, where the stubbing is fairly simple like in this example.for
more complex mocking, use the Sling Mock API or the AEM Mocks when AEM objects need to be mocked.
 *
 * Note that the testable class is under /src/main/java:
```

```
 * com.adobe.training.core.models.Stockplex.java
 *
 *  To correctly use this testing class:
 *  -put this file under training.core/src/test/java in the package com.adobe.training.core.models
 *
 */
@ExtendWith(MockitoExtension.class)
public class StockplexMockitoTest {

    private Stockplex stock;

    @BeforeEach
    public void setup() throws Exception {

        //Adapt the Resource if needed
        Stockplex STOCKMODEL_MOCK = mock(Stockplex.class);

        stock = STOCKMODEL_MOCK;

        //Setup stock symbol
        final String alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
        final int N = alphabet.length();
        Random r = new Random();
        String str = "";
        for (int i = 0; i < 4; i++) {
            str = str + alphabet.charAt(r.nextInt(N));
        }
        when(STOCKMODEL_MOCK.getSymbol()).thenReturn(str);

        //Setup current trade price
        Random rand = new Random();
        double n = Math.round(100*(rand.nextInt(150) + 100)+rand.nextDouble())/100; //random value between 100.00 and 150.00
        when(STOCKMODEL_MOCK.getCurrentPrice()).thenReturn(n);

    }

    @Test
    void testGetLastTradeValue() throws Exception{
        assertNotNull(stock, "lastTradeModel is null»);
        assertTrue(stock.getCurrentPrice() > 100, "current value is incorrect");
        assertFalse(stock.getSymbol().isEmpty(), "stock symbol is incorrect");

        // Verify that only those two methods were called on the mocked object
        verify(stock).getCurrentPrice();
        verify(stock).getSymbol();

        // Verify that no other interaction occurred on the mocked object
        verifyNoMoreInteractions(stock);
//The one below is when we want to make sure that a specific method was never invoked
        verify(stock, never()).getSummary();
    }

}
```

6. Examine the code.

Note: This class will test the Java class Stockplex.java.

7. Open a command prompt to the location of your Maven project. For example: **C:/adobe/<myproject>**

Note: If you are using an IDE with an integrated terminal such as Visual Studio Code, you can run your Maven commands there instead since it is already open to your project.

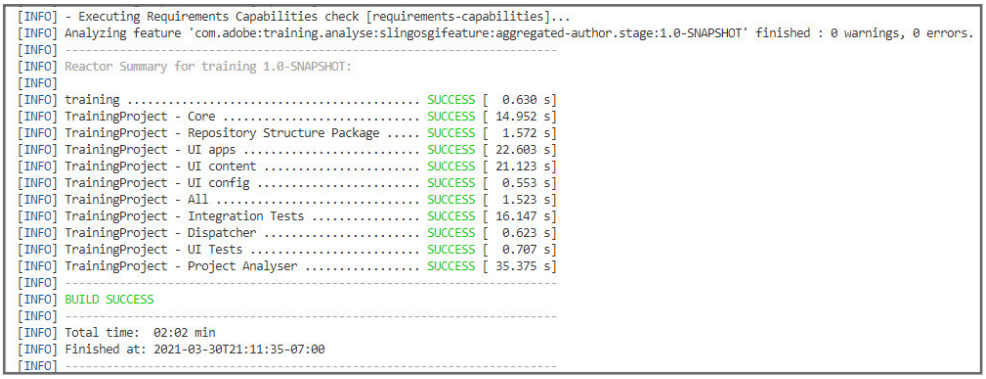
8. In the command prompt navigate to the core module:

```
cd core
```

9. Run the unit tests by using Maven:

```
mvn clean test
```

10. Verify the tests ran successfully:



Sling Mocks

Sling Mocks are a part of the AEM archetype.

The Maven dependency for Sling Mocks are as follows:

Junit 5:

```
<dependency>
  <groupId>org.apache.sling</groupId>
  <artifactId>org.apache.sling.testing.sling-mock.junit5</artifactId>
</dependency>
```

The mock implementation supports:

- ResourceResolver implementation for reading and writing resource data using the Sling Resource API Backed by a mocked or real Jackrabbit JCR implementation
  - Uses the productive Sling JCR resource provider implementation internally to do the Resource-JCR mapping.
  - Alternatively, the non-JCR mock implementation provided by the Sling resourceresolver-mock implementation can be used.
- AdapterManager implementation for registering adapter factories and resolving adaptions
  - The implementation is thread-safe, so it can be used in parallel running unit tests.
- SlingScriptHelper implementation providing access to mocked request/response objects and supports getting OSGi services from the mocked OSGi environment.
- Implementations of the servlet-related Sling API classes like SlingHttpServletRequest and SlingHttpServletResponse
  - It is possible to set request data to simulate a certain Sling HTTP request.
- Support for Sling Models (Sling Models API 1.1 and Impl 1.1 or higher required), all relevant Sling Models services are registered by default.
- Additional services: MimeTypeService
- Context Plugins

Exercise 2: Create unit tests using Sling Mocks

In this lab exercise, you will create unit tests using Sling mocking framework.
This exercise includes three tasks:

1. Add the sling-mock dependency
2. Create a unit test with sample data
3. Run the test

Task 1: Add the Sling-mock dependency

Note: This dependency allows you to use the Sling Mocks framework. Sling Mocks allow you to create mock resources to test different classes. Mock resources can be autogenerated using JSON files.
The code is available as part of the parent-pom.xml under **Exercise_Files-EC/training-files/Writing-Tests/**.

1. In the **Exercise_Files-EC**, navigate to **/training-files/Writing-Tests/.** Open **core-pom.xml**.
2. Copy the **junit5** dependency at the bottom:

```
<!-- Dependency for StockplexSlingMockTest.java -->
<dependency>
    <groupId>org.apache.sling</groupId>
    <artifactId>org.apache.sling.testing.sling-mock.junit5</artifactId>
    <version>2.3.4</version>
    <scope>test</scope>
</dependency>
```

3. In your IDE, navigate to **training** > **core** > **pom.xml**.
4. Double-click **pom.xml** to open it in the editor. The **pom.xml** file opens.
5. Add the **org.apache.sling.testing.sling-mock.junit5** dependency, as shown:

```
138         <!-- Dependency for StockplexSlingMockTest.java -->
139         <dependency>
140             <groupId>org.apache.sling</groupId>
141             <artifactId>org.apache.sling.testing.sling-mock.junit5</artifactId>
142             <version>2.3.4</version>
143             <scope>test</scope>
144         </dependency>
```

Note: The code is available as part of the parent-pom.xml under **Exercise_Files-EC/training-files/Writing-Tests/**.

6. Save the changes.

Task 2: Create a unit test with sample data

1. In your IDE, navigate to **core** > **src** > **test/java/com/adobe/training/core/models**.
2. Right-click **models** and select **New File**.
3. Name the file as **StockplexSlingMockTest.java**.
4. Copy the contents from **Exercise_Files-EC** under **/core/src/test/java/com/adobe/training/core/models/StockplexSlingMockTest.java** to **StockplexSlingMockTest.java** in your IDE. The **StockplexSlingMockTest.java** class is created.
5. Verify the code, as shown:

```
package com.adobe.training.core.models;
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertNotEquals;
import org.apache.sling.api.resource.Resource;
import org.apache.sling.models.factory.ModelFactory;
import org.apache.sling.testing.mock.sling.junit5.SlingContext;
import org.apache.sling.testing.mock.sling.junit5.SlingContextExtension;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import com.adobe.training.core.StockDataWriterJob;
import com.google.common.collect.ImmutableMap;
/**
        Tests the Stockplex model using the Sling Mock implementation for Sling APIs
*
        A Sling context (the mock environment) needs to be created to test the classes.
        The implementation used in that example is the ResourceResolver mock,
        to allow in-memory reading and writing resource data using the Sling Resource API.
*
        It also provides support for Sling Models (Sling Models API 1.1 and Impl 1.1 or higher required)
*
        Note that the testable class is under /src/main/java:
        com.adobe.training.core.models.Stockplex.java
*
        To correctly use this testing class:
        -put this file under training.core/src/test/java in the package com.adobe.training.core.models
*
        In order test with SlingContext, in parent/pom.xml add the dependency:
                <dependency>
                <groupId>org.apache.sling</groupId>
                <artifactId>org.apache.sling.testing.sling-mock.junit5</artifactId>
                <version>2.3.4</version>
                <scope>test</scope>
                </dependency>
*
        And in the core/pom.xml add:
        <dependency>
                <groupId>org.apache.sling</groupId>
                <artifactId>org.apache.sling.testing.sling-mock.junit5</artifactId>
        </dependency>
*
*/
@ExtendWith(SlingContextExtension.class)
```

```
public class StockplexSlingMockTest {
private SlingContext context = new SlingContext();
@BeforeEach
void setup() throws Exception {
        //it may be required to register the models of this project manually depending on your project/IDE setup
        context.addModelsForClasses(Stockplex.class);
        //Load dummy data for imported stock data
        context.load().json("/imported-stock-data.json",StockDataWriterJob.STOCK_IMPORT_FOLDER);
}

/**
        Loads a page into the context via a JSON file and then tests the stock model can successfully return values
        */
@Test
void testLoadedContent() throws Exception{
                //Load a resource from JSON
                Resource pageResource = context.load().json("/test-page.json", "/content/training/us/en");
                Resource stockplexResource = pageResource.getChild("jcr:content/root/container/stockplex");
                context.request().setResource(stockplexResource);

                Stockplex model = context.getService(ModelFactory.class).createModel(context.request(), Stockplex.
                class);

                assertEquals("ADBE", model.getSymbol());
                assertEquals("Adobe's stock price today", model.getSummary());
                assertNotEquals("MSFT", model.getSymbol());
}
/**
        Load a resource into the context with the create().resource method.
        */
@Test
void testCreatedContent() {
                //Create a resource programmatically

                Resource stockplexResource = context.create().resource("/content/mypage/jcr:content/root/
                container/stockplex", ImmutableMap.<String,Object>builder()
                        .put("jcr:primaryType", "nt:unstructured")
                        .put("jcr:createdBy", "admin")
                        .put("summary", "Adobe's stock price today")
                        .put("jcr:lastModifiedBy", "admin")
                        .put("symbol", "ADBE")
                        .put("jcr:created", "Thu Mar 21 2019 12:23:17 GMT-0400")
                        .put("showStockDetails", "true")
                        .put("jcr:lastModified", "Thu Mar 21 2019 12:23:34 GMT-0400")
                        .put("sling:resourceType", "training/components/content/stockplex")
                        .build());
                context.request().setResource(stockplexResource);


                Stockplex model = context.getService(ModelFactory.class).createModel(context.request(), Stockplex.
                class);

                assertEquals("ADBE", model.getSymbol());
                assertNotEquals("", model.getSummary());
                assertNotEquals("MSFT",model.getSymbol());
}

/**
        In this test we create a resource that uses the stockplex model. This model then looks for imported stock data
        under /content/stocks/ADBE. Since ADBE stock was loaded, the expected outcome is the price being returned.
        */
@Test
void testImportedStockData() {
                String stockSymbol = "ADBE";
```

```
                Resource stockplexResource = context.create().resource("/content/mypage/jcr:content/root/
                container/stockplex", ImmutableMap.<String,Object>builder()
                        .put("jcr:primaryType", "nt:unstructured")
                        .put("symbol", stockSymbol)
                        .put("sling:resourceType", "training/components/stockplex")
                        .build());
                context.request().setResource(stockplexResource);
                Stockplex model = context.getService(ModelFactory.class).createModel(context.request(), Stockplex.
                class);
                assertNotEquals(0, model.getCurrentPrice()); //Validate there is stockprice
                //validate the data returned has a size greater than 1. If it does, then data was successfully imported
                assertNotEquals(model.getData().size(), 1);
}

/**
        In this test we create a resource that uses the stockplex model. This model then looks for imported stock data
        under /content/stocks/APPL. Since APPL stock was not loaded, the expected outcome is 0 and data size to be 1
        containing the noDataString message
        */
@Test
void testNoneExistantStockData() {
                String stockSymbol = "APPL";
                Resource stockplexResource = context.create().resource("/content/mypage/jcr:content/root/
                container/stockplex", ImmutableMap.<String,Object>builder()
                        .put("jcr:primaryType", "nt:unstructured")
                        .put("symbol", stockSymbol)
                        .put("sling:resourceType", "training/components/stockplex")
                        .build());
                context.request().setResource(stockplexResource);
                Stockplex model = context.getService(ModelFactory.class).createModel(context.request(), Stockplex.
                class);
                assertEquals(Double.valueOf("0"), model.getCurrentPrice());
                //if data only returns 1 value in the set, then there was no data and it just contains the noDataString
                assertEquals(model.getData().size(), 1);
}
}
```

6. Examine the code. Observe how the mock test loads a json file named imported-stock-data.json. This code is used to simulate stock data already imported to the JCR.

The **StockplexSlingMock.java** essentially has two types of page setups:

1. The page resource is imported via json (test-page.json): a.testLoadedContent()

2. The page resource is manually created using the following methods:

a. testCreatedContent()

b. testImportedStockData() - This method tests that the stockplex component is pulling the actual stock

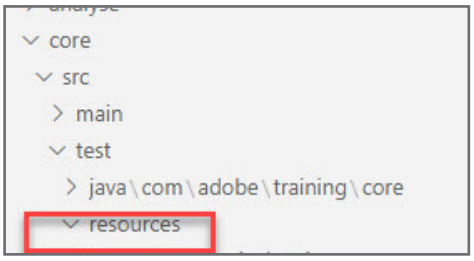
data loaded

c. testNoneExistantStockData() - This method tests that the stockplex component is displaying a 'no stock data' message appropriately

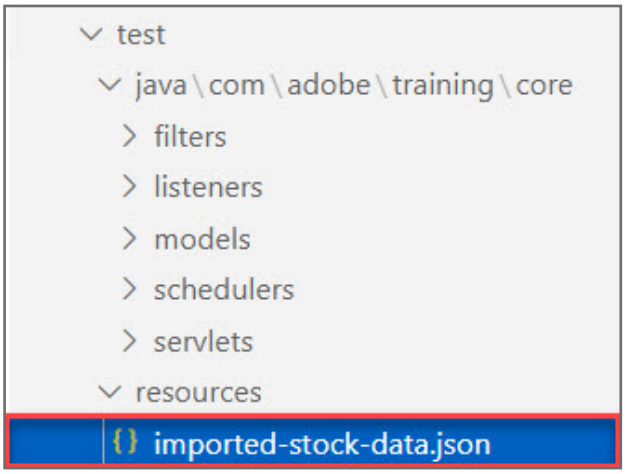
7. In your IDE **Explorer**, navigate to **training/core** > **src/test.**
8. Right-click **src/test** and select **New Folder**. The **New Folder** screen opens.

Note: If you are using Eclipse, create a **New Source Folder** named resources.

9. Provide the value to create the folder, as shown:
   - Folder name: **resources**

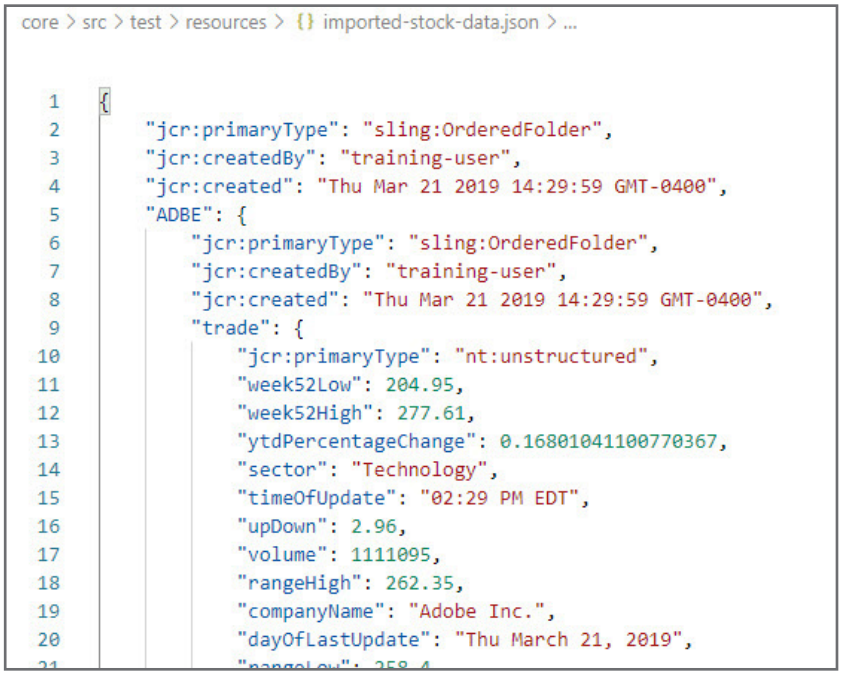


10. Click **Finish**. The folder is created.
11. Create a New File named **imported-stock-data.json** in the **resources** folder**, as shown:**



12. Copy the content of the file **imported-stock-data.json** from **/Exercise_Files-EC/core/src/test/resources/ imported-stock-data.json.**

13. The **imported-stock-data.json** is created, as shown:



14. Similarly, create another file named **test-page.json** in the **resources** folder.

15. Copy the content of the file **test-page.json** from **/Exercise_Files-EC/core/src/test/resources/test-page.json.**

16. The **test-page.json** is created. Examine the json file for the data.

Task 3: Run the test

1. To run all the tests open a command prompt to the location of your Maven project. For example: **C:/adobe/< myproject >**

Note: If you are using an IDE with an integrated terminal such as Visual Studio Code, you can run your Maven commands there instead since it is already open to your project.

---

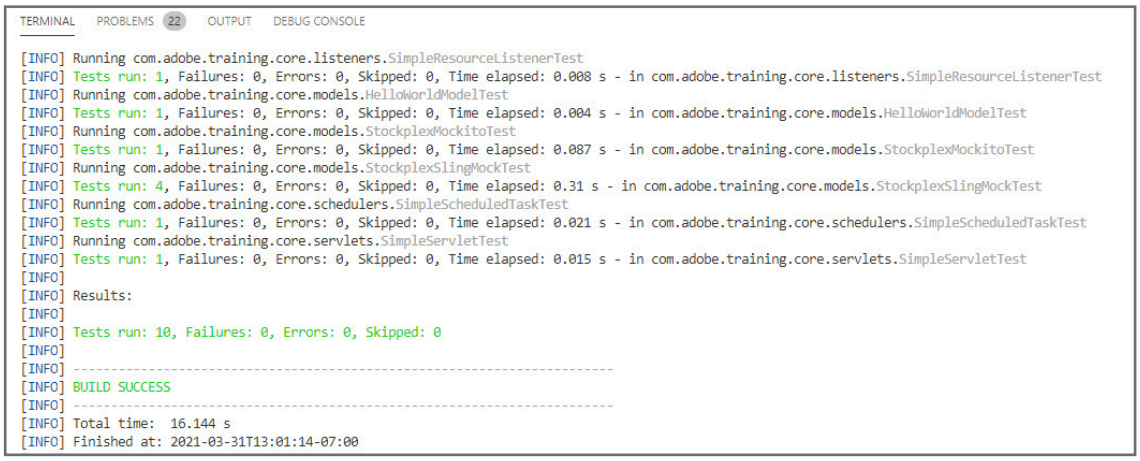
2. In the command prompt navigate to the core module:

```
cd core
```

3. Run the unit tests by using Maven:

```
mvn clean test
```

4. Verify the tests ran successfully, as shown:



AEM Mocks

AEM Mocks are inclusive of JCR, OSGi, and Sling Mocks. It also adds extra functionality on top for pages, assets, and other AEM objects.
Usage:
The AemContext object provides access to mock implementations of:

- OSGi Component Context
- OSGi Bundle Context
- Sling Resource Resolver
- Sling Request
- Sling Response
- Sling Script Helper

Additionally, it supports:

- Registering OSGi services
- Registering adapter factories
- Accessing JSON Importer

The AEM mock context can be injected into a JUnit test using a custom JUnit rule named AemContext. This rule takes care of all initialization and cleanup tasks required to make sure all unit tests can run independently. Example:

```
public class ExampleTest {

  @Rule
  public final AemContext context = new AemContext();

  @Test
  public void testSomething() {
    Resource resource = context.resourceResolver().getResource("/content/sample/en");
    Page page = resource.adaptTo(Page.class);
    // further testing
  }

}
```

It is possible to combine such a unit test with a @RunWith annotation. The AEM mock context supports different resource resolver types (provided by the Sling Mocks implementation).

Exercise 3: Create unit tests using AEM Mocks

In this lab exercise, you will create unit tests using AEM mocking framework.
This exercise includes three tasks:

1. Add the AEM-mock dependency
2. Create a unit test with sample data
3. Run the test

Task 1: Add the AEM-mock dependency

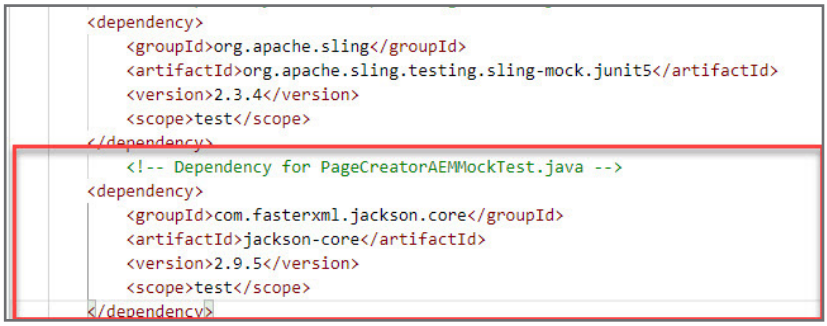
Note: The AEM-mock dependency allows you to use the Sling Mocks framework. Sling Mocks allow you to create mock resources to test our different classes. Mock resources can be autogenerated using JSON files.
The code is available as part of the parent-pom.xml in **Exercise_Files-EC** under **/training-files/Writing-Tests/**.

1. In the **Exercise_Files-EC**, navigate to **/training-files/Writing-Tests/.** Open **core-pom.xml**.
2. Copy the **jackson-core** dependency at the bottom:

```
<!-- Dependency for PageCreatorAEMMockTest.java -->
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-core</artifactId>
    <version>2.9.5</version>
    <scope>test</scope>
</dependency>
```

3. In your IDE, navigate to **training** > **core** > **pom.xml**.
4. Double-click **pom.xml** to open it in the editor. The **pom.xml** file opens.

5. Add the **com.fasterxml.jackson.core** dependency, as shown:



```
<dependency>
    <groupId>org.apache.sling</groupId>
    <artifactId>org.apache.sling.testing.sling-mock.junit5</artifactId>
    <version>2.3.4</version>
    <scope>test</scope>
</dependency>
    <!-- Dependency for PageCreatorAEMMockTest.java -->
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-core</artifactId>
    <version>2.9.5</version>
    <scope>test</scope>
</dependency>
```

6. Save the changes.

Task 2: Create a unit test with sample data

1. In your IDE, navigate to **core** > **src** > **test/java/com/adobe/training/core/models**.
2. Right-click **models** and select **New File**.
3. Name the file as **PageCreatorAEMMockTest.java**.
4. Copy the contents from **Exercise_Files-EC** under **/core/src/test/java/com/adobe/training/core/models/PageCreatorAEMMockTest.java** to **PageCreatorAEMMockTest.java** in your IDE. The **PageCreatorAEMMockTest.java** class is created.
5. Verify the code, as shown:

```
1.  package com.adobe.training.core.servlets;
2.  import static org.junit.jupiter.api.Assertions.assertEquals;
3.  import static org.junit.jupiter.api.Assertions.assertNotNull;
4.  import static org.mockito.ArgumentMatchers.eq;
5.  import static org.mockito.Mockito.mock;
6.  import static org.mockito.Mockito.when;
7.  import javax.servlet.http.HttpServletResponse;
8.  import com.day.cq.tagging.TagManager;
9.  import com.day.cq.wcm.api.PageManagerFactory;
10. import com.fasterxml.jackson.databind.JsonNode;
11. import com.fasterxml.jackson.databind.ObjectMapper;
12. import com.google.common.collect.ImmutableMap;
13. import org.apache.sling.api.resource.ResourceResolver;
14. import org.junit.jupiter.api.BeforeEach;
15. import org.junit.jupiter.api.Test;
16. import org.junit.jupiter.api.extension.ExtendWith;
17. import io.wcm.testing.mock.aem.junit5.AemContext;
18. import io.wcm.testing.mock.aem.junit5.AemContextExtension;
19. /**
        • This class uses the AemContext object from AEM mocks to test the PageCreator Model.
20. *
        • In order to test with JSON objects, in parent/pom.xml add the dependency:
21. <dependency>
22. <groupId>com.fasterxml.jackson.core</groupId>
23. <artifactId>jackson-core</artifactId>
24. <version>2.9.5</version>
25. <scope>test</scope>
26. </dependency>
27. *
        • And in the core/pom.xml add:
28. <dependency>
29. <groupId>com.fasterxml.jackson.core</groupId>
30. <artifactId>jackson-core</artifactId>
```

```
31.  </dependency>
32.  *
33.  */
34.  @ExtendWith(AemContextExtension.class)
35.  public class PageCreatorAEMMockTest {
36.  private AemContext context = new AemContext();
37.  private PageCreator pageCreator;
38.  private static String TEMPLATE_PATH = "/conf/training/settings/wcm/templates/page-content";
39.  private static String PAGE_PATH = "/content/training/us/en";
40.  /**
         • The setUp method loads the AemContext with services and resources to be used in the tests below.
41.  */
42.  @BeforeEach
43.  void setUp() throws Exception {
44.  //Register PageManagerFactory and mock the return for PageMannager from AEMContext
45.  PageManagerFactory pmf = context.registerService(PageManagerFactory.class, mock(PageManagerFactory.class));
46.  when(pmf.getPageManager(eq(context.request().getResourceResolver()))).thenReturn(context.pageManager());
47.  //Create PageCreator with Active Services
48.  pageCreator = context.registerInjectActivateService(new PageCreator());
49.  //Create Mocked Content
50.  context.load().json("/test-page-template.json", TEMPLATE_PATH);
51.  context.load().json("/test-page.json", PAGE_PATH);


52.  try (ResourceResolver rr = context.resourceResolver()) {
53.  TagManager tm = rr.adaptTo(TagManager.class);
54.  tm.createTag("/content/cq:tags/training/biking", "Biking", "");
55.  } catch (Exception e) {
56.  e.printStackTrace();
57.  }
58.  }
59.  /**
         • This test creates a POST request for the PageCreator class to validate
         • a correct input for page creation
60.  */
61.  @Test
62.  void testSuccessfulInput() throws Exception{
63.  //Simulate a POST request
64.  context.request().setMethod("POST");
65.  context.request().setParameterMap(ImmutableMap.of(
66.  "importer",PAGE_PATH+"/community,Our Community,"+TEMPLATE_PATH+",/content/cq:tags/training/biking")
67.  );
68.  //Call the Servlet
69.  pageCreator.service(context.request(), context.response());
70.  assertEquals(HttpServletResponse.SC_OK, context.response().getStatus());
71.  String jsonString = context.response().getOutputAsString();
72.  ObjectMapper mapper = new ObjectMapper();
73.  JsonNode actualObj = mapper.readTree(jsonString);
74.  assertEquals("Successful", actualObj.findValue("Status").asText());
75.  }
76.  /**
         • This test creates a POST request for the PageCreator class to validate
         • if an error message is returned if there is no parent path given
77.  */
78.  @Test
79.  void testNoParentPath() throws Exception{
80.  //Simulate a POST request
81.  context.request().setMethod("POST");
82.  context.request().setParameterMap(ImmutableMap.of(
83.  "importer",",Our Community,"+TEMPLATE_PATH+",/content/cq:tags/training/biking")
84.  );
85.  //Call the Servlet
86.  pageCreator.service(context.request(), context.response());
87.  assertEquals(HttpServletResponse.SC_OK, context.response().getStatus());
88.  String jsonString = context.response().getOutputAsString();
89.  ObjectMapper mapper = new ObjectMapper();
90.  JsonNode actualObj = mapper.readTree(jsonString);
91.  assertEquals("Error", actualObj.findValue("Status").asText());
92.  }


93.  /**
         • This test creates a POST request for the PageCreator class to validate
```

```
        • if the default template is used when none is given
94.  */
95.  @Test
96.  void testNoTemplate() throws Exception{
97.  //Simulate a POST request
98.  context.request().setMethod("POST");
99.  context.request().setParameterMap(ImmutableMap.of(
100. "importer",PAGE_PATH+"/community,Our Community,,/content/cq:tags/training/biking")
101. );
102. //Call the Servlet
103. pageCreator.service(context.request(), context.response());
104. assertEquals(HttpServletResponse.SC_OK, context.response().getStatus());
105. String jsonString = context.response().getOutputAsString();
106. ObjectMapper mapper = new ObjectMapper();
107. JsonNode actualObj = mapper.readTree(jsonString);
108. //Assert the default template is used
109. assertEquals(TEMPLATE_PATH, actualObj.findValue("Template Used").asText());
110. }
111. /**
        • This test creates a POST request for the PageCreator class to validate
        • if an error message is returned because the template is not a valid template in the project.
112. */
113. @Test
114. void testDNETemplate() throws Exception{
115. //Simulate a POST request
116. context.request().setMethod("POST");
117. context.request().setParameterMap(ImmutableMap.of(
118. "importer",PAGE_PATH+"/community,Our Community,/conf/training/settings/wcm/templates/mytemplate,/content/
    cq:tags/training/biking,FALSE")
119. );
120. //Call the Servlet
121. pageCreator.service(context.request(), context.response());
122. assertEquals(HttpServletResponse.SC_OK, context.response().getStatus());
123. String jsonString = context.response().getOutputAsString();
124. ObjectMapper mapper = new ObjectMapper();
125. JsonNode actualObj = mapper.readTree(jsonString);
126. //Assert that the status is Error and a Template message is given
127. assertEquals("Error", actualObj.findValue("Status").asText());
128. assertNotNull(actualObj.findValue("Template"));
129. }
130. }
```

Note: The PageCreatorAEMMockTest.java class has a few tests. The testSuccessfulInput() tests a valid input for PageCreator. The other three methods testNoParentPath(), testNoTemplate() and testDNETemplate() test error-prone inputs.

6. Examine the code. Observe how the mock test loads two json files for the page and template to use for the tests.
7. In your IDE **Explorer**, navigate to **training/core** > **src/test.**
8. Right-click the **resources** folder and create a New File named **test-page-template.json.**
9. Copy the content of the file **test-page-template.json** from **/Exercise_Files-EC/core/src/test/resources/test-page-template.json.**
10. The **test-page-template.json** is created.

Task 3: Run the test

1. To run all the tests open a command prompt to the location of your Maven project. For example: **C:/adobe/< myproject >**

Note: If you are using an IDE with an integrated terminal such as Visual Studio Code, you can run your Maven commands there instead since it is already open to your project.

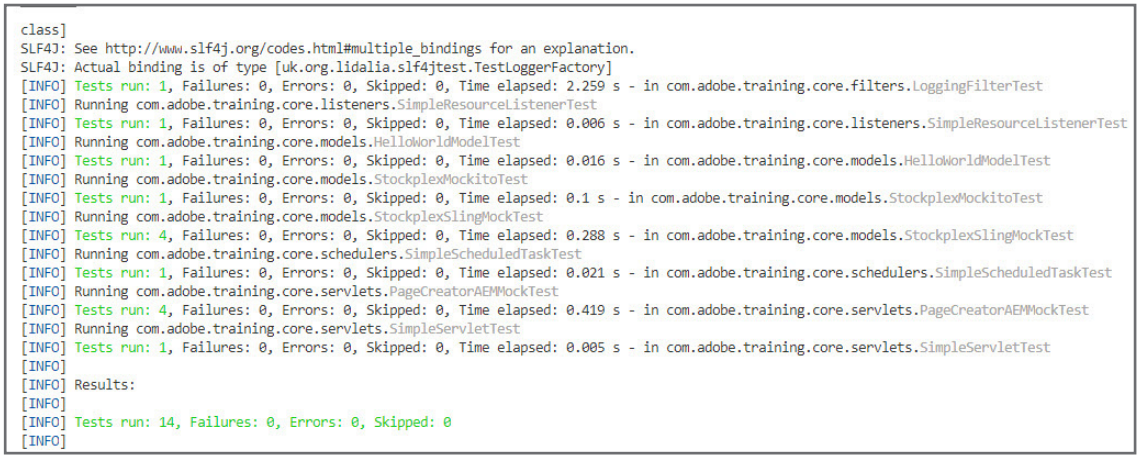
2. In the command prompt navigate to the core module:

```
cd core
```

3. Run the unit tests by using Maven:

```
mvn clean test
```

4. Verify the tests ran successfully, as shown:



References

Sling Mocks: https://sling.apache.org/documentation/development/sling-mock.html

AEM Mocks: http://wcm.io/testing/aem-mock/usage.html

AEM Mocks in AEM project Archetype: https://wcm-io.atlassian.net/wiki/spaces/WCMIO/pages/1046609921/How+to+use+AEM+Mocks+in+Adobe+AEM+Project+Archetype

wcm.io - Troubleshooting:
https://wcm-io.atlassian.net/wiki/spaces/WCMIO/pages/4816898/Troubleshooting

Maven dependency: https://mvnrepository.com/artifact/io.wcm/io.wcm.testing.aem-mock

Maven dependency: http://wcm.io/testing/aem-mock/usage-content-loader-builder.html