1- DDT

1- For JSON

To implement **Data-Driven Testing (DDT)** in Java Selenium using a JSON file for login testing, you can use **Page Object Model (POM)** along with **Jackson** or **Gson** libraries for JSON parsing. Here's a step-by-step approach:

Step 1: Set Up Dependencies

Add the following dependencies in your `pom.xml` file:

```
<dependencies>
<!-- Selenium Dependency -->
<dependency>
    <groupId>org.seleniumhq.selenium</groupId>
    <artifactId>selenium-java</artifactId>
    <version>4.1.0</version>
</dependency>

<!-- Jackson JSON Parsing Dependency -->
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.13.0</version>
</dependency>

<!-- TestNG Dependency -->
<dependency>
    <groupId>org.testng</groupId>
    <artifactId>testng</artifactId>
    <version>7.4.0</version>
    <scope>test</scope>
</dependency>
</dependencies>
```

Step 2: Create JSON Data File

Create a JSON file named `loginData.json` in the `resources` directory to hold the test data for different users.

```
[
  [
    "username1",
    "password1"
  ],
  [
    "username2",
    "password2"
  ]
]
```
Or

```
[
  {
    "username": "user1@example.com",
    "password": "password1"
  },
  {
    "username": "user2@example.com",
    "password": "password2"
  }
]
```

Step 3: Define the Login Page Object

In the `pages` package, create a `LoginPage.java` class with methods for interacting with the login page.

```java
public class LoginPage {
    WebDriver driver;

    public LoginPage(WebDriver driver) {
        this.driver = driver;
    }

    private By usernameField = By.id("username");
    private By passwordField = By.id("password");
    private By loginButton = By.id("submit");

    public void setUsername(String username) {
        driver.findElement(usernameField).sendKeys(username);
    }

    public void setPassword(String password) {
        driver.findElement(passwordField).sendKeys(password);
    }

    public void clickLogin() {
        driver.findElement(loginButton).click();
    }

    public void login(String username, String password) {
        setUsername(username);
        setPassword(password);
        clickLogin();
    }
}
```

Step 4: Create a Helper Class to Parse JSON Data

Create a utility class `JsonDataReader.java` in the `utils` package to read JSON data and parse it into a Java object.

```
public class JsonDataReader {
    private static final String FILE_PATH_ARR_TO_ARR =
"src/test/java/Files/Data.json";
    private static final String FILE_PATH_OBJECTS =
"src/test/java/Files/d.json";


    public static Object[][] getLoginDataArrToArr() {
        ObjectMapper objectMapper = new ObjectMapper();
        System.out.println(new File(FILE_PATH_ARR_TO_ARR).exists());
        try {
            return objectMapper.readValue(new File(FILE_PATH_ARR_TO_ARR),
Object[][].class);
        } catch (IOException e) {
            e.printStackTrace();
            return null;
        }
    }



        public static Object[][] getLoginDataObjects() {
            ObjectMapper objectMapper = new ObjectMapper();
            try {
                // Read JSON data as a List of Maps
                List<Map<String, String>> data = objectMapper.readValue(new
File(FILE_PATH_OBJECTS), List.class);
                // Convert the List to Object[][] for TestNG DataProvider
compatibility
                Object[][] loginData = new Object[data.size()][2];
                for (int i = 0; i < data.size(); i++) {
                    loginData[i][0] = data.get(i).get("username");
                    loginData[i][1] = data.get(i).get("password");
                }
                return loginData;
            } catch (IOException e) {
                e.printStackTrace();
                return null;
            }
        }
}
```

Step 5: Create a Test Class with TestNG and DataProvider

```
public class Login extends BaseTest{

        LoginPage login;


    @BeforeMethod
    public void init (){
        login= new LoginPage(driver);
    }
    @DataProvider(name = "loginData")
    public Object[][] loginDataProvider() {
        return JsonDataReader.getLoginDataArrToArr();
        //or return JsonDataReader.getLoginDataOjects();
    }

    @Test(dataProvider = "loginData")
    public void login(String username , String password){
        login.setUsername(username);
        login.setPassword(password);

    }
```

```
URLs :
login form : https://practicetestautomation.com/practice-test-login/
Register form : https://demo.wpeverest.com/user-registration/simple-
registration-form/
```

2- For Excel :

To use **Data-Driven Testing (DDT)** in Selenium with Java using **Excel** as the data source, we need to read data from an Excel file and apply it to a login test. Here's a step-by-step guide using **Apache POI** for reading Excel data, organized with **Page Object Model (POM)** and TestNG's **DataProvider** feature.

Step 1: Add Apache POI Dependency in POM

In your `pom.xml` file, add dependencies for Apache POI to work with Excel files:

```
<dependencies>
<!-- Apache POI for Excel handling -->
<dependency>
    <groupId>org.apache.poi</groupId>
    <artifactId>poi</artifactId>
    <version>5.2.2</version>
</dependency>
<dependency>
    <groupId>org.apache.poi</groupId>
    <artifactId>poi-ooxml</artifactId>
    <version>5.2.2</version>
</dependency>
</dependencies>
```

Step 2: Create a Utility Class to Read Data from Excel

We'll create a utility method to read Excel data and convert it to a format compatible with TestNG's `DataProvider`.

```java
public class ExcelDataReaderLogin {
    private static final String FILE_PATH = "src/test/java/Files/Data.xlsx";


    public static Object[][] getExcelData() {
        try (FileInputStream file = new FileInputStream(FILE_PATH);
             Workbook workbook = new XSSFWorkbook(file)) {

            Sheet sheet = workbook.getSheetAt(0); // Read the first sheet
            int rowCount = sheet.getPhysicalNumberOfRows();
            int colCount = sheet.getRow(0).getPhysicalNumberOfCells();

            Object[][] data = new Object[rowCount - 1][colCount];

            Iterator<Row> rows = sheet.iterator();
            rows.next(); // Skip the header row
            int i = 0;
            while (rows.hasNext()) {
                Row row = rows.next();
                for (int j = 0; j < colCount; j++) {
                    data[i][j] = row.getCell(j).toString();
                }
                i++;
            }
            return data;
        } catch (IOException e) {
            e.printStackTrace();
            return null;
        }
    }
}
```

Step 3: Create a Login Page Class Using Page Object Model (POM)

Define a `LoginPage` class that interacts with the login page elements.

```java
public class LoginPage {
    WebDriver driver;

    public LoginPage(WebDriver driver) {
        this.driver = driver;
    }

    private By usernameField = By.id("username");
    private By passwordField = By.id("password");
    private By loginButton = By.id("submit");

    public void setUsername(String username) {
        driver.findElement(usernameField).sendKeys(username);
    }

    public void setPassword(String password) {
        driver.findElement(passwordField).sendKeys(password);
    }

    public void clickLogin() {
        driver.findElement(loginButton).click();
    }

    public void login(String username, String password) {
        setUsername(username);
        setPassword(password);
        clickLogin();
    }
}
```

Step 4: Implement the Test Case Using TestNG and Apply DataProvider :

```
public class Login extends BaseTest{

        LoginPage login;


    @BeforeMethod
    public void init (){
        login= new LoginPage(driver);
    }
    @DataProvider(name = "loginData")
    public Object[][] loginDataProvider() {
        return ExcelDataReaderLogin.getExcelData();
        //or return JsonDataReader.getLoginDataOjects();
    }

    @Test(dataProvider = "loginData")
    public void login(String username , String password){
        login.setUsername(username);
        login.setPassword(password);

    }
}
```

2- Run group of classes or Methods

```
<!DOCTYPE suite SYSTEM "https://testng.org/testng-1.0.dtd">

<suite name="Suite1" verbose="1">
    <test name="Nopackage">
        <classes>
            <class name="classname"/>
        </classes>
    </test>
```

3- Soft assertions VS hard assertions

Soft Assertions vs Hard Assertions in Selenium Java

In Selenium (and Java testing frameworks like TestNG or JUnit), assertions are used to validate the test outcome. Assertions can be classified into **hard assertions** and **soft assertions**, each with distinct behaviors and use cases.

Hard Assertions

- **Behavior**: If a hard assertion fails, the test execution stops immediately at the failed assertion.
- **Use Case**: Ideal for scenarios where the continuation of the test doesn't make sense if the assertion fails, e.g., verifying a critical precondition.
- **Example**:

```
java
Copy code
import org.testng.Assert;

@Test
public void testHardAssertion() {
    System.out.println("Step 1");
    Assert.assertEquals("actualValue", "expectedValue", "Values do not
match!");
    System.out.println("Step 2"); // This won't execute if the assertion
fails
}
```

- **Common Methods**:
  - `Assert.assertTrue(condition, message)`
  - `Assert.assertFalse(condition, message)`
  - `Assert.assertEquals(actual, expected, message)`
  - `Assert.assertNotEquals(actual, expected, message)`

Soft Assertions

- **Behavior**: If a soft assertion fails, the test execution continues. All failures are collected and reported at the end of the test.
- **Use Case**: Useful when you want to validate multiple checkpoints within the same test and review all failures simultaneously.
- **Example**:

```
java
Copy code
import org.testng.asserts.SoftAssert;

@Test
public void testSoftAssertion() {
    SoftAssert softAssert = new SoftAssert();
    System.out.println("Step 1");
    softAssert.assertEquals("actualValue1", "expectedValue1", "First Check
Failed");
    System.out.println("Step 2");
    softAssert.assertEquals("actualValue2", "expectedValue2", "Second Check
Failed");
    System.out.println("Step 3");
    softAssert.assertAll(); // Consolidates all assertion results
}
```

- **Important Notes**:
    - `softAssert.assertAll()` is critical to mark the test as failed if any assertion fails. Without this, failures will be ignored.
    - Soft assertions require you to manage the `SoftAssert` object explicitly.

Comparison Table

Feature	Hard Assertions	Soft Assertions
Execution Behavior	Stops execution on failure	Continues execution on failure
Failure Reporting	Reports failure immediately	Reports failures at the end
Use Case	Critical checkpoints	Multiple validations in one test
Example Method	`Assert.assertEquals`	`softAssert.assertEquals`

Best Practices

1. **When to Use Hard Assertions**:
   - Validating preconditions (e.g., page navigation).
   - Critical test steps that make subsequent steps irrelevant if they fail.
2. **When to Use Soft Assertions**:
   - Verifying multiple attributes (e.g., field values in a form).
   - Non-critical checks where the test can proceed even if one validation fails.
3. **Mixing Hard and Soft Assertions**:
   - Use hard assertions for critical validations and soft assertions for secondary checks within the same test.