A good **folder structure for Java automation** makes your test framework **organized, maintainable, and scalable**. Since most Java automation frameworks (Selenium, Cypress-Java, Rest-Assured, Appium, etc.) are built using **Maven/Gradle** and follow **Page Object Model (POM)** or **Hybrid** patterns, I’ll give you a **standard Maven-based folder structure** that you can adapt.

**📂 Standard Java Automation Project Structure (Maven)**

AutomationFramework/

│── pom.xml # Maven configuration (dependencies, plugins, build)

│── testng.xml # TestNG suite file (if using TestNG)

│── junit-platform.properties # If using JUnit 5 configs

│

├── src

│ ├── main

│ │ ├── java

│ │ │ └── com.company.projectname

│ │ │ ├── base # Base setup classes (DriverManager, TestBase)

│ │ │ ├── config # Configuration utils (PropertyReader, Constants)

│ │ │ ├── pages # Page Object Model classes (LoginPage, HomePage)

│ │ │ ├── utils # Common utilities (ExcelUtils, WaitUtils, Logger)

│ │ │ ├── api # API automation layer (if using Rest-Assured)

│ │ │ └── enums # Enum classes (e.g., BrowserType, Environment)

│ │ │

│ │ └── resources

│ │ ├── config.properties # Environment variables, credentials

│ │ ├── log4j2.xml # Logging configuration

│ │ ├── testdata # Test data files (Excel, CSV, JSON)

│ │ └── drivers # WebDriver executables (if needed)

│ │

│ ├── test

│ │ ├── java

│ │ │ └── com.company.projectname

│ │ │ ├── tests # Test classes (LoginTest, CheckoutTest)

│ │ │ └── runners # TestNG/JUnit runners (if using Cucumber, hooks, tags)

│ │ │

│ │ └── resources

│ │ └── testng.xml # Alternative test suites (optional)

│ │

│── reports # Test reports (ExtentReports, Allure, surefire-reports)

│── screenshots # Screenshots of failed test cases

**📌 Explanation of Key Folders**

* **base/**  
  Holds BaseTest or DriverFactory classes → sets up WebDriver, API clients, etc.
* **config/**  
  Manages property files and constants → Environment URLs, browser type, timeouts.
* **pages/**  
  Implements **Page Object Model (POM)** → One class per page, containing locators and methods.  
  Example: LoginPage.java, DashboardPage.java.
* **utils/**  
  Common helper functions:
  + WaitUtils.java → Explicit waits
  + ExcelUtils.java → Read/write test data
  + ScreenshotUtils.java → Capture failures
* **api/**  
  (Optional) If you also automate APIs, keep **API request/response classes** using **Rest-Assured** here.
* **tests/**  
  Actual test cases (grouped by feature/module).  
  Example: LoginTest.java, CartTest.java.
* **runners/**  
  Used if running Cucumber/BDD tests with tags.  
  Example: TestRunner.java.
* **reports/ & screenshots/**  
  Store test execution reports & screenshots.

**📂 Example for Cucumber BDD Projects**

If you’re using **Cucumber + TestNG/JUnit**, add:

│── src/test/resources

│ ├── features # Cucumber feature files (.feature)

│ ├── stepdefinitions # Step definition classes

│ └── hooks # Hooks for before/after scenarios

⚡ **Best Practices**

* Keep **test logic separate from business logic** (tests should only call page methods).
* Use **Page Factory** or simple POM for maintainability.
* Store credentials/test data in **config.properties** or external test data files (never hardcoded).
* Integrate with **Extent Reports / Allure** for rich reporting.
* Use **CI/CD (Jenkins/GitHub Actions)** with headless execution.

Great question 👍. Let’s break down **how a Java-based test automation framework actually works** step by step. I’ll explain this in a **generic way** (applicable to Selenium/TestNG/JUnit frameworks, and even BDD with Cucumber).

**🔹 How a Test Automation Framework Works**

Think of a test framework as a **machine with multiple layers**. Each layer has a role, and when you run tests, these layers work together.

**1. Test Runner (Entry Point)**

* You trigger tests from:
  + testng.xml (for TestNG)
  + @Test annotations (JUnit/TestNG)
  + @CucumberOptions runner (Cucumber BDD)
* The runner controls **which tests execute** (smoke, regression, API, UI, etc.).
* It also integrates with **build tools** like Maven/Gradle, and CI/CD pipelines.

**Example:**

<suite name="Regression Suite">

<test name="Login Tests">

<classes>

<class name="com.company.project.tests.LoginTest"/>

</classes>

</test>

</suite>

**2. Base Layer (Setup & Teardown)**

* Before any test runs:
  + Framework reads config (e.g., browser type, URL, timeout).
  + Initializes **WebDriver** (for UI tests) or **API client** (for API tests).
  + Opens the browser / sets API headers.

**Example flow:**

* BaseTest loads properties → DriverManager launches Chrome → Test starts.
* After tests finish → @AfterSuite or @AfterMethod closes resources.

**3. Page Object Model (Business Layer)**

* Each **Page Class** represents a web page / module.
* Contains locators + reusable methods.
* Keeps tests **readable** and **maintainable**.

**Example (LoginPage.java):**

public class LoginPage {

@FindBy(id = "username") WebElement username;

@FindBy(id = "password") WebElement password;

@FindBy(id = "loginBtn") WebElement loginButton;

public void login(String user, String pass) {

username.sendKeys(user);

password.sendKeys(pass);

loginButton.click();

}

}

**4. Test Layer**

* Actual **test cases** live here.
* They **call Page methods**, not direct WebDriver code.
* Assertions are placed here (Expected vs Actual).

**Example (LoginTest.java):**

@Test

public void verifyLogin() {

LoginPage loginPage = new LoginPage(driver);

loginPage.login("admin", "admin123");

Assert.assertTrue(new HomePage(driver).isDisplayed(), "Login Failed!");

}

**5. Utilities & Helpers**

* Handle **common logic** used across tests:
  + Waits (WaitUtils)
  + Excel/CSV readers (ExcelUtils)
  + Screenshots (ScreenshotUtils)
  + API clients (RestAssuredUtils)

**6. Test Data Layer**

* Test data comes from:
  + config.properties (environment configs)
  + Excel/CSV/JSON (data-driven testing)
  + Hardcoded constants (not recommended)

**Example:**

url=https://demoapp.com

browser=chrome

username=admin

password=admin123

**7. Assertions & Reporting**

* Test validations happen with assertions (Assert in TestNG/JUnit).
* Reports are generated (Extent, Allure, Surefire, etc.).
* Screenshots are captured on failures.

**8. Execution & CI/CD**

* You run tests via:
  + mvn test (Maven command)
  + Jenkins pipeline / GitHub Actions for CI/CD.
* Tests execute in **parallel** or **sequentially**, based on configuration.
* Results are published in reports and dashboards.

**🔹 Putting It All Together (Flow)**

1. **Run suite** (testng.xml / Maven / Jenkins).
2. **Framework setup** loads → reads configs → launches driver/API client.
3. **Tests run** → Call Page Objects → Perform actions.
4. **Assertions** check expected vs actual results.
5. **On failure** → Logs + Screenshot + Report entry.
6. **Teardown** → Driver/API closes → Reports generated.

👉 In short:

* **Runner** controls execution.
* **Base Layer** manages setup/teardown.
* **Pages** hold business logic.
* **Tests** hold validation logic.
* **Utils + Data + Reports** support the execution.

⚡ Harish, do you want me to make a **diagram flow (visual)** showing how these layers interact, so you can easily explain it in interviews?

Nice catch 👍 This is the part that often confuses people when they first see a **Java automation framework**. Let me explain clearly **where tests actually live and how they get invoked after the driver launches**.

**🔹 Where the Tests Live**

* All **test classes** (e.g. LoginTest.java, CheckoutTest.java) are kept under:

src/test/java/com.company.project.tests

* These are the **entry points for actual test execution**.
* Inside these classes you have @Test (TestNG/JUnit) or @Given/@When/@Then (Cucumber) annotated methods.

**🔹 How Tests Get Invoked**

Let’s break the flow:

1. **Runner is triggered**
   * If using TestNG → testng.xml specifies which test classes to run.
   * If using JUnit → @Test annotation itself is the entry.
   * If using Cucumber → TestRunner.java picks up feature files.
2. **Before running a test method**
   * The test class (like LoginTest.java) usually extends BaseTest.java.
   * In BaseTest, you have @BeforeMethod or @BeforeClass that **launches the driver**.
   * Example:
   * public class BaseTest {
   * protected WebDriver driver;
   * @BeforeMethod
   * public void setUp() {
   * driver = DriverManager.getDriver("chrome");
   * driver.get(ConfigReader.get("url"));
   * }
   * @AfterMethod
   * public void tearDown() {
   * if (driver != null) {
   * driver.quit();
   * }
   * }
   * }
3. **Test class picks it up**
   * When TestNG runs LoginTest, it first calls BaseTest.setUp().
   * That launches the browser.
   * Then your actual test code runs (calling Page Object methods).
4. **After the test**
   * tearDown() runs → browser closes.

**🔹 Example Structure in Practice**

src/test/java

│── com.company.project.tests

│ ├── LoginTest.java

│ ├── CheckoutTest.java

│ └── SearchTest.java

│

src/main/java

│── com.company.project.base

│ └── BaseTest.java

│── com.company.project.pages

│ └── LoginPage.java

│ └── HomePage.java

│── com.company.project.utils

│ └── ConfigReader.java

│ └── DriverManager.java

**LoginTest.java**

public class LoginTest extends BaseTest {

@Test

public void verifyLogin() {

LoginPage login = new LoginPage(driver);

login.login("admin", "admin123");

HomePage home = new HomePage(driver);

Assert.assertTrue(home.isDisplayed(), "Login failed!");

}

}

* Here, verifyLogin() is invoked by TestNG/JUnit.
* But before this test runs, BaseTest.setUp() is automatically invoked.
* That’s where the **driver launch happens**, so when the test body executes, driver is ready.

✅ **So the short answer to your doubt:**

* Tests are always picked from the **tests package under src/test/java**.
* They are invoked by the **Test Runner (TestNG/JUnit/Cucumber)**.
* The driver is already launched by the **Base class (BaseTest)**, thanks to @BeforeMethod or @BeforeClass.
* The test class just **reuses the driver** to run steps via Page Objects.

Harish, do you want me to show you a **minimal working Maven project skeleton** (with pom.xml, BaseTest, one Page, and one Test) that you can literally run with mvn test to see this in action?

Perfect 👌 Let me give you a **minimal working Java Selenium TestNG Maven project skeleton** that you can directly run with mvn test.

**📂 Project Structure**

automation-framework/

│── pom.xml

│── testng.xml

│

├── src

│ ├── main

│ │ ├── java

│ │ │ └── com.demo.framework

│ │ │ ├── base

│ │ │ │ └── BaseTest.java

│ │ │ ├── pages

│ │ │ │ ├── LoginPage.java

│ │ │ │ └── HomePage.java

│ │ │ └── utils

│ │ │ ├── DriverManager.java

│ │ │ └── ConfigReader.java

│ │ └── resources

│ │ └── config.properties

│ │

│ ├── test

│ │ └── java

│ │ └── com.demo.framework.tests

│ │ └── LoginTest.java

**⚙️ pom.xml**

Dependencies for Selenium + TestNG + WebDriverManager:

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0

http://maven.apache.org/maven-v4\_0\_0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.demo</groupId>

<artifactId>automation-framework</artifactId>

<version>1.0-SNAPSHOT</version>

<packaging>jar</packaging>

<dependencies>

<!-- Selenium -->

<dependency>

<groupId>org.seleniumhq.selenium</groupId>

<artifactId>selenium-java</artifactId>

<version>4.21.0</version>

</dependency>

<!-- TestNG -->

<dependency>

<groupId>org.testng</groupId>

<artifactId>testng</artifactId>

<version>7.9.0</version>

<scope>test</scope>

</dependency>

<!-- WebDriverManager (auto driver setup) -->

<dependency>

<groupId>io.github.bonigarcia</groupId>

<artifactId>webdrivermanager</artifactId>

<version>5.8.0</version>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-surefire-plugin</artifactId>

<version>3.2.5</version>

<configuration>

<suiteXmlFiles>

<suiteXmlFile>testng.xml</suiteXmlFile>

</suiteXmlFiles>

</configuration>

</plugin>

</plugins>

</build>

</project>

**⚙️ testng.xml**

Defines which tests to run:

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

<suite name="Automation Suite">

<test name="Login Tests">

<classes>

<class name="com.demo.framework.tests.LoginTest"/>

</classes>

</test>

</suite>

**⚙️ config.properties**

url=https://opensource-demo.orangehrmlive.com/

browser=chrome

**⚙️ DriverManager.java**

package com.demo.framework.utils;

import io.github.bonigarcia.wdm.WebDriverManager;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.chrome.ChromeDriver;

public class DriverManager {

public static WebDriver initDriver(String browser) {

WebDriver driver = null;

if (browser.equalsIgnoreCase("chrome")) {

WebDriverManager.chromedriver().setup();

driver = new ChromeDriver();

}

driver.manage().window().maximize();

return driver;

}

}

**⚙️ ConfigReader.java**

package com.demo.framework.utils;

import java.io.FileInputStream;

import java.io.IOException;

import java.util.Properties;

public class ConfigReader {

private static Properties props;

static {

try {

FileInputStream fis = new FileInputStream("src/main/resources/config.properties");

props = new Properties();

props.load(fis);

} catch (IOException e) {

throw new RuntimeException("Config file not found!");

}

}

public static String get(String key) {

return props.getProperty(key);

}

}

**⚙️ BaseTest.java**

package com.demo.framework.base;

import com.demo.framework.utils.ConfigReader;

import com.demo.framework.utils.DriverManager;

import org.openqa.selenium.WebDriver;

import org.testng.annotations.AfterMethod;

import org.testng.annotations.BeforeMethod;

public class BaseTest {

protected WebDriver driver;

@BeforeMethod

public void setUp() {

driver = DriverManager.initDriver(ConfigReader.get("browser"));

driver.get(ConfigReader.get("url"));

}

@AfterMethod

public void tearDown() {

if (driver != null) {

driver.quit();

}

}

}

**⚙️ LoginPage.java**

package com.demo.framework.pages;

import org.openqa.selenium.By;

import org.openqa.selenium.WebDriver;

public class LoginPage {

private WebDriver driver;

private By username = By.id("txtUsername");

private By password = By.id("txtPassword");

private By loginBtn = By.id("btnLogin");

public LoginPage(WebDriver driver) {

this.driver = driver;

}

public void login(String user, String pass) {

driver.findElement(username).sendKeys(user);

driver.findElement(password).sendKeys(pass);

driver.findElement(loginBtn).click();

}

}

**⚙️ HomePage.java**

package com.demo.framework.pages;

import org.openqa.selenium.By;

import org.openqa.selenium.WebDriver;

public class HomePage {

private WebDriver driver;

private By dashboard = By.id("menu\_dashboard\_index");

public HomePage(WebDriver driver) {

this.driver = driver;

}

public boolean isDisplayed() {

return driver.findElement(dashboard).isDisplayed();

}

}

**⚙️ LoginTest.java**

package com.demo.framework.tests;

import com.demo.framework.base.BaseTest;

import com.demo.framework.pages.HomePage;

import com.demo.framework.pages.LoginPage;

import org.testng.Assert;

import org.testng.annotations.Test;

public class LoginTest extends BaseTest {

@Test

public void verifyLogin() {

LoginPage loginPage = new LoginPage(driver);

loginPage.login("Admin", "admin123"); // default creds for OrangeHRM demo site

HomePage homePage = new HomePage(driver);

Assert.assertTrue(homePage.isDisplayed(), "Login Failed!");

}

}

**🚀 How to Run**

1. Save all files in the structure above.
2. Open terminal in project root.
3. Run:
4. mvn clean test
5. Chrome browser will launch → open OrangeHRM demo → perform login → verify Dashboard.

Harish, do you want me to also **add Extent Reports integration** so that after test execution you get a beautiful HTML report instead of plain console logs?

Perfect 👍 Harish! Let’s add **Extent Reports** to our framework and then I’ll explain the **step-by-step flow of how everything works** so you can confidently explain it in interviews.

**🔹 1. Add Extent Reports Dependency**

Update your pom.xml:

<!-- Extent Reports -->

<dependency>

<groupId>com.aventstack</groupId>

<artifactId>extentreports</artifactId>

<version>5.1.1</version>

</dependency>

**🔹 2. Create ExtentManager.java**

This manages report creation.

package com.demo.framework.utils;

import com.aventstack.extentreports.ExtentReports;

import com.aventstack.extentreports.reporter.ExtentHtmlReporter;

public class ExtentManager {

private static ExtentReports extent;

public static ExtentReports getInstance() {

if (extent == null) {

ExtentHtmlReporter reporter = new ExtentHtmlReporter("reports/extent-report.html");

reporter.config().setReportName("Automation Test Report");

reporter.config().setDocumentTitle("Automation Results");

extent = new ExtentReports();

extent.attachReporter(reporter);

}

return extent;

}

}

**🔹 3. Create ExtentTestListener.java**

Implements TestNG listener to log events.

package com.demo.framework.utils;

import com.aventstack.extentreports.ExtentReports;

import com.aventstack.extentreports.ExtentTest;

import org.testng.ITestContext;

import org.testng.ITestListener;

import org.testng.ITestResult;

public class ExtentTestListener implements ITestListener {

private static ExtentReports extent = ExtentManager.getInstance();

private static ThreadLocal<ExtentTest> test = new ThreadLocal<>();

@Override

public void onTestStart(ITestResult result) {

ExtentTest extentTest = extent.createTest(result.getMethod().getMethodName());

test.set(extentTest);

}

@Override

public void onTestSuccess(ITestResult result) {

test.get().pass("Test Passed");

}

@Override

public void onTestFailure(ITestResult result) {

test.get().fail("Test Failed: " + result.getThrowable());

}

@Override

public void onTestSkipped(ITestResult result) {

test.get().skip("Test Skipped: " + result.getThrowable());

}

@Override

public void onFinish(ITestContext context) {

extent.flush();

}

}

**🔹 4. Update testng.xml to Register Listener**

Add the listener so TestNG knows to use it:

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

<suite name="Automation Suite">

<listeners>

<listener class-name="com.demo.framework.utils.ExtentTestListener"/>

</listeners>

<test name="Login Tests">

<classes>

<class name="com.demo.framework.tests.LoginTest"/>

</classes>

</test>

</suite>

**🔹 5. Run Tests and View Report**

* Run:
* mvn clean test
* After execution → open
* /reports/extent-report.html
* You’ll see a beautiful HTML report with **pass/fail status**.

**🔹 6. Flow of How This Framework Works**

Let’s break it down like a **story**:

**(A) Test Execution Trigger**

* You start tests by running mvn test.
* Maven reads pom.xml → invokes **Surefire Plugin** → reads testng.xml.

**(B) TestNG Suite**

* testng.xml tells TestNG:
  + Which classes (LoginTest) to run.
  + Which listeners (ExtentTestListener) to attach.

**(C) Setup (BaseTest)**

* Before each test:
  + @BeforeMethod runs.
  + DriverManager launches browser.
  + ConfigReader loads URL.
  + Browser navigates to the application.

**(D) Test Execution**

* LoginTest runs:
  + Calls LoginPage.login().
  + That enters username, password, clicks Login.
  + Then HomePage.isDisplayed() checks Dashboard.

**(E) Assertions**

* If Dashboard is visible → Assert.assertTrue(true) → TestNG marks it **PASS**.
* If not visible → Assertion fails → TestNG marks it **FAIL**.

**(F) Reporting**

* Each test start/success/failure/skip is logged into **Extent Report** by ExtentTestListener.
* On finish, extent.flush() writes everything to HTML.

**(G) Teardown**

* After each test, @AfterMethod runs.
* Driver quits, cleaning up resources.

**(H) Reports**

* End result:
  + Console log (TestNG default).
  + HTML Report (extent-report.html).
  + Screenshots (optional – can be added on failure).

**🔹 Visual Flow Diagram (Simplified)**

mvn test

│

└── testng.xml

│

├── ExtentTestListener (registers report)

└── LoginTest

│

└── BaseTest.setUp()

│

└── DriverManager → Launch Browser

│

└── Navigate to URL

│

└── Test Execution (calls PageObjects)

│

└── Assertion → PASS/FAIL

│

└── ExtentReport logs result

│

└── BaseTest.tearDown() → quit browser

│

└── extent-report.html (final report generated)

✅ So now you have:

* A **working Selenium + TestNG + Extent Report** framework.
* Full **execution flow** to explain in interviews.
* Extendable structure (add APIs, multiple pages, data-driven tests).

👉 Do you want me to also add **screenshot capture on failure**, so the Extent Report shows screenshots when a test fails?

Got it Harish 👍 — usually teams capture only **failed test screenshots**, but if you want **screenshots for both Passed and Failed cases (authentic evidence)**, we can extend our framework easily.

**🔹 1. Add Screenshot Utility**

📄 ScreenshotUtils.java

package com.demo.framework.utils;

import org.openqa.selenium.OutputType;

import org.openqa.selenium.TakesScreenshot;

import org.openqa.selenium.WebDriver;

import java.io.File;

import java.io.IOException;

import java.nio.file.Files;

import java.text.SimpleDateFormat;

import java.util.Date;

public class ScreenshotUtils {

public static String captureScreenshot(WebDriver driver, String testName) {

String timestamp = new SimpleDateFormat("yyyyMMdd\_HHmmss").format(new Date());

String screenshotPath = "reports/screenshots/" + testName + "\_" + timestamp + ".png";

File srcFile = ((TakesScreenshot) driver).getScreenshotAs(OutputType.FILE);

File destFile = new File(screenshotPath);

try {

Files.createDirectories(destFile.getParentFile().toPath());

Files.copy(srcFile.toPath(), destFile.toPath());

} catch (IOException e) {

throw new RuntimeException("Failed to save screenshot: " + e.getMessage());

}

return screenshotPath;

}

}

**🔹 2. Modify ExtentTestListener.java to Attach Screenshots for Both Passed and Failed Tests**

📄 ExtentTestListener.java

package com.demo.framework.utils;

import com.aventstack.extentreports.ExtentReports;

import com.aventstack.extentreports.ExtentTest;

import com.aventstack.extentreports.MediaEntityBuilder;

import org.openqa.selenium.WebDriver;

import org.testng.\*;

public class ExtentTestListener implements ITestListener {

private static ExtentReports extent = ExtentManager.getInstance();

private static ThreadLocal<ExtentTest> test = new ThreadLocal<>();

@Override

public void onTestStart(ITestResult result) {

ExtentTest extentTest = extent.createTest(result.getMethod().getMethodName());

test.set(extentTest);

}

@Override

public void onTestSuccess(ITestResult result) {

WebDriver driver = ((com.demo.framework.base.BaseTest) result.getInstance()).driver;

String screenshotPath = ScreenshotUtils.captureScreenshot(driver, result.getMethod().getMethodName());

test.get().pass("Test Passed",

MediaEntityBuilder.createScreenCaptureFromPath(screenshotPath).build());

}

@Override

public void onTestFailure(ITestResult result) {

WebDriver driver = ((com.demo.framework.base.BaseTest) result.getInstance()).driver;

String screenshotPath = ScreenshotUtils.captureScreenshot(driver, result.getMethod().getMethodName());

test.get().fail("Test Failed: " + result.getThrowable(),

MediaEntityBuilder.createScreenCaptureFromPath(screenshotPath).build());

}

@Override

public void onTestSkipped(ITestResult result) {

test.get().skip("Test Skipped: " + result.getThrowable());

}

@Override

public void onFinish(ITestContext context) {

extent.flush();

}

}

**🔹 3. Run Flow (Now with Screenshots for All Tests)**

1. Run mvn clean test.
2. For each test case:
   * On **Pass** → Screenshot is captured & attached in Extent Report.
   * On **Fail** → Screenshot is captured & attached with error log.
3. Open report at:
4. reports/extent-report.html

→ You’ll see thumbnails/screenshots for **every test case**.

**🔹 Example Report Entry**

✅ **Passed Test:**

LoginTest.verifyLogin → PASS

[Screenshot thumbnail attached in report]

❌ **Failed Test:**

CheckoutTest.verifyCheckout → FAIL

java.lang.AssertionError: Expected title mismatch

[Screenshot thumbnail attached in report]

**🔹 Flow of Framework with Screenshot Evidence**

mvn test

│

└── testng.xml

│

└── LoginTest

│

└── BaseTest.setUp() → Launch browser

│

└── Execute Test → Call PageObjects

│

└── Assert PASS/FAIL

│

└── ScreenshotUtils.captureScreenshot()

│

└── ExtentReport logs screenshot

│

└── BaseTest.tearDown() → quit browser

│

└── extent-report.html (with screenshot evidence for all cases)

✅ With this, every test has **visual proof** → useful for audits, client reviews, and debugging.

👉 Harish, do you also want me to add **timestamped report names** (like extent-report-20250822.html) so that every run generates a new report instead of overwriting?

Great question 👍  
This is **one of the most common real-time interview points** about frameworks:  
👉 How do you tag/classify your tests (Smoke, Regression, Sanity, etc.) and run them selectively?

Let’s go step by step:

**1. Using TestNG Groups (most common in Java Selenium)**

In TestNG, you can assign groups to your test methods using the @Test(groups=...) annotation.

**Example:**

import org.testng.annotations.Test;

public class LoginTests {

@Test(groups = {"smoke"})

public void validLoginTest() {

// Steps for login with valid data

}

@Test(groups = {"regression"})

public void invalidLoginTest() {

// Steps for login with invalid data

}

@Test(groups = {"smoke", "regression"})

public void forgotPasswordTest() {

// Common test needed in both smoke and regression

}

}

**2. Running Specific Groups from testng.xml**

You can create multiple testng.xml files for different suites (smoke, regression, sanity).

**Example testng-smoke.xml:**

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

<suite name="SmokeSuite">

<test name="SmokeTests">

<groups>

<run>

<include name="smoke"/>

</run>

</groups>

<classes>

<class name="tests.LoginTests"/>

<class name="tests.DashboardTests"/>

</classes>

</test>

</suite>

**Example testng-regression.xml:**

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

<suite name="RegressionSuite">

<test name="RegressionTests">

<groups>

<run>

<include name="regression"/>

</run>

</groups>

<classes>

<class name="tests.LoginTests"/>

<class name="tests.CheckoutTests"/>

</classes>

</test>

</suite>

**3. Execution Flow**

1. **Driver Initialization** → from BaseTest (@BeforeSuite or @BeforeClass)
2. **Tests Invoked** → based on what testng.xml is triggered
   * If you run testng-smoke.xml, only methods with @Test(groups="smoke") execute.
   * If you run testng-regression.xml, only regression methods run.
3. **Listeners** → capture results, logs, and screenshots for both pass/fail.
4. **Reports** → HTML, Extent, or Allure will show suite type (Smoke/Regression) and test outcomes.

**4. Alternative: Using Maven Profiles**

You can bind suite files to Maven profiles in pom.xml.

**Example:**

<profiles>

<profile>

<id>smoke</id>

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-surefire-plugin</artifactId>

<version>3.0.0</version>

<configuration>

<suiteXmlFiles>

<suiteXmlFile>testng-smoke.xml</suiteXmlFile>

</suiteXmlFiles>

</configuration>

</plugin>

</plugins>

</build>

</profile>

<profile>

<id>regression</id>

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-surefire-plugin</artifactId>

<version>3.0.0</version>

<configuration>

<suiteXmlFiles>

<suiteXmlFile>testng-regression.xml</suiteXmlFile>

</suiteXmlFiles>

</configuration>

</plugin>

</plugins>

</build>

</profile>

</profiles>

Then you can run:

mvn test -Psmoke

mvn test -Pregression

✅ **Summary**:

* **Groups in TestNG** let you tag tests as smoke/regression.
* **testng.xml** decides which tagged group to run.
* **Maven profiles** allow CI/CD pipelines to pick test suites dynamically.
* Reports (Extent/Allure) will show which suite ran + pass/fail evidence.

👉 Harish, do you want me to also **show you how this looks in Extent Reports** (like Smoke vs Regression in the report sidebar), so you can present it confidently in interviews?