TestNG

* **What is TestNG and what is it used for**TestNG (Test Next Generation) is a powerful testing framework in Java, inspired by JUnit but designed to offer more advanced features for test automation. It is used to define, group, prioritize, and run test cases in a flexible and organized way. TestNG supports annotations like @Test, @BeforeMethod, @AfterClass, and more to manage test execution flow efficiently. One of its major advantages is the ability to execute tests in parallel, define test dependencies, run tests in groups, and generate detailed HTML and XML reports. It integrates well with Selenium for automating web application testing, making it easier to manage test suites, assertions, and test configurations through testng.xml. In real-time projects, I’ve used TestNG to structure automation frameworks, implement retry logic, parameterize tests using @DataProvider, and integrate with CI tools like Jenkins.
* **Key Features of TestNG**

TestNG offers a wide range of features that make it highly suitable for automation testing. One of its core features is the use of **annotations** like @Test, @BeforeMethod, @AfterClass, which help manage test execution flow cleanly. It supports **parallel test execution**, which speeds up test runs and improves efficiency. TestNG also allows **test grouping**, so you can run specific sets of tests (like smoke, regression, or sanity) using groups in testng.xml. Another key feature is **dependency management**, allowing one test to run only after another has passed. TestNG provides built-in support for **parameterization** using @Parameters and @DataProvider, which is useful for data-driven testing. It generates **detailed HTML and XML reports** automatically after test execution. Additionally, it integrates well with build tools like Maven and CI tools like Jenkins, making it a preferred framework in real-time Selenium automation projects.

* Difference between TestNG and NUnit  
  TestNG and NUnit are both powerful unit testing frameworks used in automation testing—TestNG is for Java, while NUnit is for C#. Functionally, they are quite similar and support features like annotations/attributes (@Test in TestNG vs [Test] in NUnit), grouping, assertions, data-driven testing, and parallel execution. TestNG uses annotations like @BeforeMethod, @AfterClass, while NUnit uses [SetUp], [TearDown], etc., to manage test lifecycle events. Both support XML-based configuration for defining test suites and allow integration with CI/CD tools like Jenkins. However, NUnit is native to the .NET ecosystem and integrates well with tools like Visual Studio, while TestNG is more suited for Java-based frameworks and works seamlessly with Maven and Eclipse/IntelliJ. In summary, the choice between the two depends mainly on the programming language and tech stack being used.
* How to Install and Set Up TestNG in IntelliJ IDEA  
  To install and set up TestNG in IntelliJ IDEA, we can either add it manually as a library or manage it through Maven or Gradle. The easiest and most common way is using Maven. First, we create or open a Maven project in IntelliJ, then add the TestNG dependency to the pom.xml file:  
  A computer screen with white text

  AI-generated content may be incorrect.  
  Once saved, IntelliJ automatically downloads the dependency. If you're not using Maven, you can manually add the TestNG JAR file by going to File > Project Structure > Modules > Dependencies > + > JARs or directories, and select the downloaded TestNG JAR. After adding it, you can create test classes and annotate methods with @Test. To run tests, right-click the test class or method and select "Run". IntelliJ also supports generating testng.xml files for suite configuration and provides a built-in TestNG runner for easy execution and report viewing.
* How to Write a Basic TestNG Test Case  
  Writing a basic TestNG test case in Java involves creating a class, writing a method, and annotating it with @Test. First, make sure the TestNG library is added to your project—either via Maven or by adding the JAR. Then, create a Java class and import the necessary TestNG annotations. Here's a simple example:  
  A computer screen shot of a program code

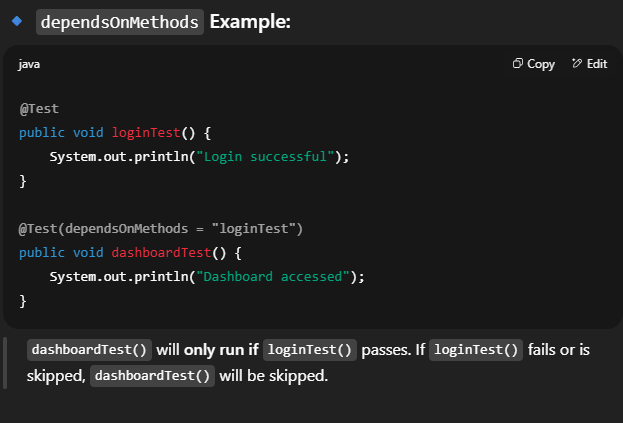
  AI-generated content may be incorrect.  
  **You can run this test by right-clicking on the method or class in IntelliJ and selecting "Run", or you can configure it in a testng.xml suite file. TestNG will automatically recognize the @Test annotation and execute the method. You can also add assertions, setup methods (@BeforeMethod, @AfterMethod), and parameters to build more advanced test cases.**
* **What is the significance of @Test annotation in TestNG**The @Test annotation in TestNG is used to mark a method as a test case so that TestNG can recognize and execute it during test runs. It eliminates the need to follow any specific naming convention for test methods—as long as a method is annotated with @Test, TestNG treats it as a valid test. This annotation also provides several attributes like priority, enabled, groups, invocationCount, and dependsOnMethods, which offer powerful control over how and when tests are executed. For example, you can set @Test(priority = 1) to define the order of execution, or use @Test(enabled = false) to skip a test. In short, @Test is the core of TestNG's test execution model and allows for flexible, structured, and maintainable test case development.
* What are the different annotations in TestNG  
  TestNG provides a rich set of annotations to control the execution flow of test cases, similar to JUnit but with more flexibility. The most commonly used annotation is @Test, which marks a method as a test case. Lifecycle annotations include @BeforeSuite and @AfterSuite, which run before and after the entire test suite. @BeforeTest and @AfterTest are used before and after <test> tags in the testng.xml file. @BeforeClass and @AfterClass run once before and after all the test methods in a class. @BeforeMethod and @AfterMethod run before and after each test method. For data-driven testing, TestNG provides @DataProvider, and @Parameters is used for parameterization through XML. There are also annotations like @BeforeGroups and @AfterGroups that run code before and after specific groups of tests. These annotations give fine-grained control over test execution and setup, making it easier to manage complex test scenarios.
* Execution Order of TestNG Annotations  
  TestNG annotations follow a specific execution order that defines how test setup and teardown operations are handled around the test methods. The typical flow is:  
  **@BeforeSuite** – Runs once before the entire test suite.  
  **@BeforeTest** – Runs before the <test> tag in the testng.xml file.  
  **@BeforeClass** – Runs once before any method in the test class.  
  **@BeforeMethod** – Runs before each @Test method.  
  **@Test** – The actual test method runs.  
  **@AfterMethod** – Runs after each @Test method.  
  **@AfterClass** – Runs once after all the test methods in the class.  
  **@AfterTest** – Runs after all the tests in the <test> tag.  
  **@AfterSuite** – Runs once after the entire suite is done.  
  This structured execution flow allows you to perform setup (like launching browsers, connecting to databases) and cleanup (like closing connections or browsers) in an organized way. In real-time projects, @BeforeMethod and @AfterMethod are most commonly used for setting up and tearing down test-specific configurations.
* Difference between @BeforeTest and @BeforeMethod in TestNG  
  In TestNG, both @BeforeTest and @BeforeMethod are used for setup tasks, but they differ in scope and frequency of execution. @BeforeTest is executed **once** before any test methods run that are included in a specific <test> tag in the testng.xml file. It's useful for setting up configurations or data that are shared across multiple test classes, such as launching a browser or setting global properties. On the other hand, @BeforeMethod runs **before each individual @Test method**, meaning it is called multiple times—once for every test method in a class. It’s typically used for method-level setup like navigating to a URL, logging in, or resetting test data. So, @BeforeTest is suite-wide or test-wide setup, whereas @BeforeMethod is per-test-method setup.
* Can we use multiple annotations in one method in TestNG?  
  **Yes**, in TestNG, you **can use multiple annotations on a single method**, but it’s generally **not recommended** because it can lead to **confusing or unpredictable behavior** depending on the context and execution flow.  
  A computer screen with text

  AI-generated content may be incorrect.  
  Technically, this will compile and run, but TestNG may **not behave as expected**, because it's unclear whether the method should execute once per class (@BeforeClass) or once before each test method (@BeforeMethod).  
    
  Best Practice:  
  Instead of combining multiple annotations on a single method, it’s better to **separate them into different clearly named methods** for maintainability and clarity:  
  A screen shot of a computer program

  AI-generated content may be incorrect.
* How do you prioritize test cases in TestNG  
  In TestNG, test case execution order can be controlled using the priority attribute of the @Test annotation. Lower numbers have higher priority, so a test with priority = 1 will run before a test with priority = 2. If no priority is specified, TestNG will execute tests in alphabetical order by default. For example:  
  A computer screen with text on it

  AI-generated content may be incorrect.  
  You can also use **negative values** for even higher priority (e.g., priority = -1). If two test methods have the same priority, TestNG falls back to method name sorting. Prioritization is helpful when you have dependent flows or want to control the sequence of test execution, especially in end-to-end scenarios.
* How do you group test cases in TestNG  
  In TestNG, you can group test cases using the groups attribute of the @Test annotation. This allows you to organize related test methods—like smoke, regression, sanity, or API tests—into logical collections. You assign a test method to one or more groups like this:  
  A screen shot of a computer program

  AI-generated content may be incorrect.  
  Then, in the testng.xml file, you can include or exclude specific groups during execution:  
  A computer screen with text on it

  AI-generated content may be incorrect.  
  Grouping is especially useful when you want to selectively run subsets of tests without changing the code. You can also combine groups in the same method or use @BeforeGroups and @AfterGroups to define setup and teardown for specific groups.
* What is dependsOnMethods and dependsOnGroups in TestNG & How to Handle Test Case Dependencies  
  In TestNG, dependsOnMethods and dependsOnGroups are used to handle test case dependencies, meaning you can configure a test to run **only if another test or group of tests has passed**. This is especially useful in scenarios where one test relies on the successful execution of another, such as logging in before accessing a user dashboard.  
   A screenshot of a computer program

  AI-generated content may be incorrect.  
  Notes:  
  Tests marked as dependent are **skipped** if their dependencies fail or are skipped.  
  Avoid circular dependencies—it will cause TestNG to throw an exception.  
  alwaysRun = true can be used to **force execution** of a test, even if its dependency fails:  
  @Test(dependsOnMethods = "loginTest", alwaysRun = true)  
  public void cleanup() {}
* What is enabled = false in the @Test annotation in TestNG  
  In TestNG, enabled = false is an attribute of the @Test annotation that is used to **disable or skip a test method** from being executed without deleting or commenting out the code. When a test method is marked with @Test(enabled = false), TestNG completely ignores that test during execution. This is useful when a test is under development, temporarily broken, or not relevant for a specific test run.  
  A screen shot of a computer

  AI-generated content may be incorrect.  
  It acts as a quick switch to turn tests on or off while maintaining them in the codebase. This is better than commenting out code because the test still exists in the reports and can be re-enabled easily.
* What is Parameterization in TestNG and How Is It Done  
  Parameterization in TestNG is the process of passing different sets of input data to the same test method, allowing it to run multiple times with different values. This is useful for data-driven testing, such as testing login functionality with different username-password combinations. TestNG supports parameterization in two main ways: using @Parameters (with testng.xml) and using @DataProvider.  
  A screenshot of a computer program

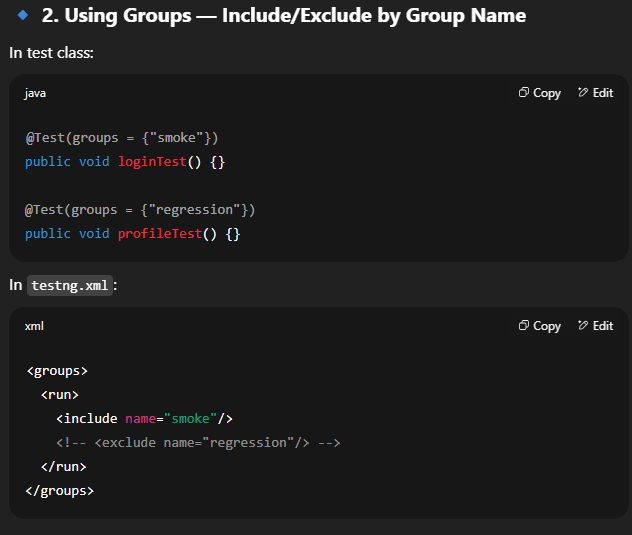
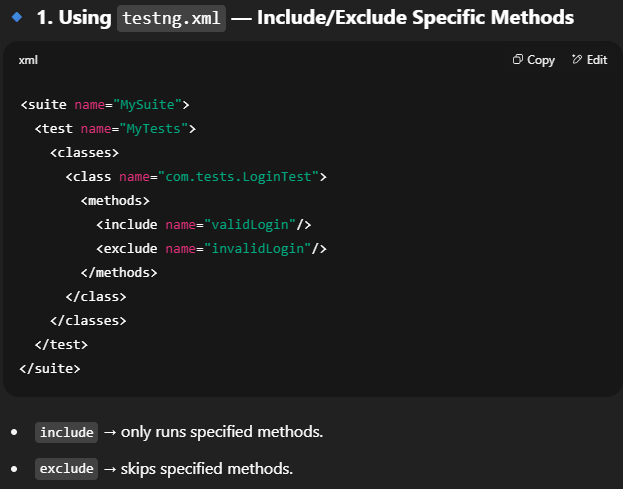
  AI-generated content may be incorrect.

A screen shot of a computer

AI-generated content may be incorrect.  
**✅ Summary:**- Use @Parameters for static data from XML.- Use @DataProvider for multiple dynamic datasets.- Parameterization makes tests reusable, scalable, and cleaner for different input combinations.

### **What are Soft and Hard Assertions in TestNG** In TestNG, assertions are used to validate expected vs actual results. There are two types: Hard Assertions and Soft Assertions. **When to Use:**

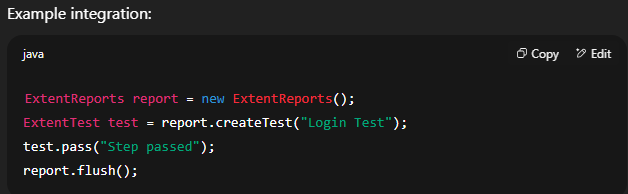
* Use **Hard Assertions** when a failure should block further steps (e.g., login must pass before going to dashboard).
* Use **Soft Assertions** when you want to **validate multiple things in one test** and report all failures together (e.g., verifying multiple UI fields).
* How to Use ITestResult to Capture Test Status in TestNG   
  In TestNG, ITestResult is an interface provided in the org.testng package that allows you to access the result of a test method during runtime. It is commonly used in listener methods like @AfterMethod to capture whether a test has passed, failed, or been skipped. This is useful for logging, taking screenshots on failure, or generating custom reports  
  A screenshot of a computer program

  AI-generated content may be incorrect.  
  **🔹 Commonly Used ITestResult Methods:**getName() – returns the test method name.  
  getStatus() – returns status: SUCCESS, FAILURE, or SKIP.  
  getThrowable() – returns the exception/error if the test failed.  
  getMethod() – gets the method reference for further metadata.
* How Do You Perform Parallel Execution in TestNG  
  In TestNG, parallel execution is achieved by configuring the testng.xml file or using annotations to instruct TestNG to run tests, classes, or methods concurrently. This improves execution speed and is especially useful for running tests across multiple browsers or datasets  
    
  parallel="methods" → Runs test methods in parallel within the same class.  
  parallel="classes" → Runs different test classes in parallel.  
  parallel="tests" → Runs entire <test> blocks in parallel.
* How to Include or Exclude Tests from Execution in TestNG  
  In TestNG, you can include or exclude tests from execution using the testng.xml file or annotations. This feature is useful when you want to run only specific tests like smoke, regression, or to temporarily skip test cases  
  
* **What Default Reports Are Generated by TestNG**TestNG automatically generates a set of HTML and XML reports after test execution, even without any additional configuration. These reports are stored in the test-output directory inside your project root. The most commonly used default reports are  
  **🔹 1. index.html**This is the main summary report.  
  It provides an overview of test suite execution: passed, failed, and skipped tests.  
  You can click through to see detailed results for each class and method. **🔹 2. emailable-report.html**A clean and simple HTML report that is suitable for emailing to stakeholders.  
  Shows test name, method name, status (PASS/FAIL/SKIP), and execution time.  
  **🔹 3. testng-results.xml**Contains all test execution details in XML format.

Useful for CI/CD integrations, parsing by custom tools, or reporting plugins  
**🔹 4. testng-failed.xml**Lists only the failed test cases.

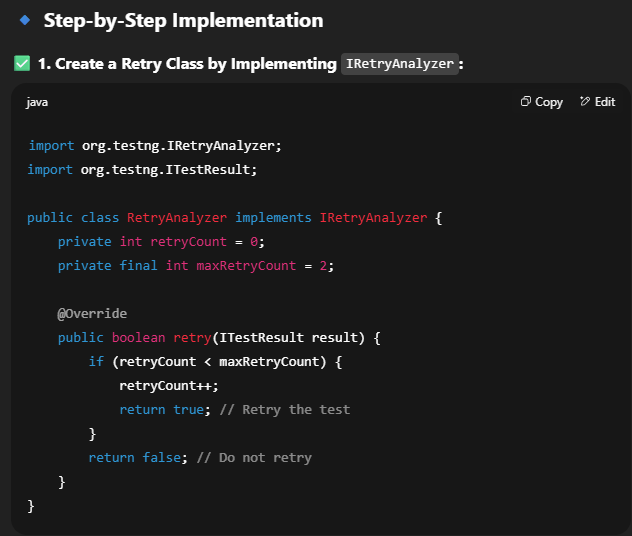
Helpful when you want to rerun only failed tests without executing the full suite.  
  
By default, TestNG generates comprehensive **HTML and XML reports** that give insights into test performance, result status, and error details—all without writing any custom reporting code.

* How Do You Customize TestNG Reports  
  While TestNG provides default reports, they are basic and may not meet all real-time project needs. To create more professional, detailed, and visually rich reports, we customize TestNG reports in the following ways.  
    
  🔹 **1. Using ITestListener Interface (Custom Listener)**You can implement the ITestListener interface to hook into the test lifecycle and customize reporting logic. For example, log test start, pass, fail, or skip events, and even capture screenshots.  
  

🔹 **2. Using Third-Party Reporting Tools  
✅ ExtentReports (Most popular for UI automation):**  
- Beautiful HTML reports  
- Supports screenshots, logs, categories, exceptions, and charts  
  
**✅ Allure Reports:  
-** Advanced visual reporting too  
- Supports annotations and integrates with Jenkins/Maven  
- Requires setup with Maven dependencies and a plugin

* How to Integrate TestNG with Maven and Jenkins  
  Integrating **TestNG with Maven and Jenkins** allows you to automate test execution in a CI/CD pipeline. This setup ensures your Selenium or API tests can run automatically on every code push or schedule  
  🔹 **1. Integrate TestNG with Maven**✅ Step 1: Add TestNG dependency in pom.xml  
  ✅ Step 2: Add Surefire Plugin to run TestNG  
  ✅ Step 3: Create a testng.xml  
  ✅ Run from terminal: mvn clean test  
    
  🔹 **2. Integrate Maven + TestNG Project with Jenkins**✅ Step 1: Install Jenkins & Maven Integration Plugin  
  ✅ Step 2: Create a Maven Job✅ Step 3: Configure Maven Job✅ Step 4: Add Post-Build Actions (Optional)

✅ Step 5: Build the Job

* What Are Listeners in TestNG and Their Types  
  In TestNG, listeners are interfaces that allow you to hook into the test execution lifecycle and perform custom actions like logging, taking screenshots, generating custom reports, or retrying failed tests. They "listen" to test events such as test start, pass, failure, and completion.  
  ✅ **Types of Listeners in TestNG**🔹 **1. ITestListener  
  -** Most commonly used listener.  
  - Listens to individual test method events  
  - Methods include: onTestStart(), onTestSuccess(), onTestFailure(), onTestSkipped(), onFinish() and onStart()  
  👉 **Use Case:** Capture screenshots on failure, log status, attach to reports  
    
  🔹 **2. ISuiteListener**- Listens to suite-level events  
  - Methods: onStart(ISuite suite), onFinish(ISuite suite)  
  👉 **Use Case:** Start/stop external services or generate summary reports before/after the test suite.  
    
  🔹 **3. IInvokedMethodListener**- Listens before and after **every method invocation**, including @Before/@After methods.  
  - Methods: beforeInvocation(), afterInvocation()  
  👉 **Use Case:** Logging or measuring method execution time.  
    
  🔹 **4. IReporter**- Used to create **custom TestNG reports**.  
  - Method: generateReport(...)  
  👉 **Use Case:** Build your own HTML/XML reports, export data to Excel/DB, etc.  
    
  🔹 **5. IAnnotationTransformer**👉 **Use Case:** Implement retry logic dynamically or set groups/priority from code.  
    
  ✅ **How to Register Listeners**- Via @Listeners annotation  
  - In testng.xml:
* How Do You Retry Failed Test Cases in TestNG  
  In TestNG, you can retry failed test cases using the IRetryAnalyzer interface. This is useful for handling flaky tests that may fail due to temporary issues like browser sync, network latency, or environment hiccups  
    
  A screenshot of a computer program

  AI-generated content may be incorrect.