**TestNG**

[Introduction 1](#_Toc202695088)

[#How to install? 3](#_Toc202695089)

[@Test(groups = {"smoke", "login-tests"}) 6](#_Toc202695090)

[DependsOn 7](#_Toc202695091)

[@Test(timeOut = 3000) 10](#_Toc202695092)

[@Parameters 10](#_Toc202695093)

[TestNG.xml file 12](#_Toc202695094)

[@Test(invocationCount = 5) 16](#_Toc202695095)

Introduction

It is a Java-based testing framework, designed to be more flexible than Junit.

**Supports:**

* Unit Test
* Functional Test
* Integration Test
* End to End Test

**Annotations**

* @Test, @BeforeMethod, @AfterMethod, @BeforeClass, @AfterClass, @BeforeTest, @AfterTest, @BeforeSuite, @AfterSuite, @BeforeGroups, @AfterGroups

**Data-Driven Testing**

* @DataProvider: Pass multiple sets of data to a single test method.
* Supports complex data types and external data sources.

**Test Configuration & Dependency Management**

* dependsOnMethods, dependsOnGroups: Define method/test execution order.
* Useful for login → then dashboard-type flows.

**Test Grouping**

* Use groups = {"smoke"} or {"regression"} in @Test annotations.
* Run specific groups using testng.xml or command-line.

**Parallel Execution**

* Run tests, classes, or methods in parallel using testng.xml or command-line.
* Improves test suite execution time.

**testng.xml File -** Central config to control: Suites, Groups, Classes, Method inclusions/exclusions, Parallelism, Parameters

**Multiple Invocations**

* @Test(invocationCount = 5) to run a test multiple time.

**Listeners & Reporters:**

* ITestListener, ISuiteListener, etc.
* Capture test lifecycle events like start, pass, fail, skip.
* Create custom logs and integrate ExtentReports.

**Retry Mechanism –**

* IRetryAnalyzer interface for re-running failed tests.
* Helps with flaky test handling.

**Assertions**

* Works with org.testng.Assert
* Assert.assertEquals(), assertTrue(), assertFalse(), etc.
* Supports Soft Assertions (SoftAssert) to continue execution after failures.

**Parameterization**

* @Parameters from testng.xml for external config injection.

**Test Prioritization**

* @Test(priority = 1) to control method order.
* Lower number runs first.

**Expected Exceptions**

* @Test(expectedExceptions = Exception.class) for negative test cases.

**Timeouts**

* @Test(timeOut = 2000) to fail tests that exceed execution time.

**Cross-Browser Testing Support**

* Combine TestNG with Selenium WebDriver to run tests across multiple browsers via parameters and testng.xml.

**Built-in Reporting**

* Generates HTML and XML reports out-of-the-box.
* Easy integration with ExtentReports, Allure, etc.

**Custom Suite Execution**

* Easily create different test suites (e.g., Smoke.xml, Regression.xml) and run via command-line or CI/CD pipelines.

**Method Interceptors and Transformers**

* Fine control over method execution and behavior modification at runtime.

**Junit Not available**: Test Grouping, Dependency Between Tests, Parallel Execution Support (3rd party), Soft Assertions , Suite-level Config Methods (Have @BeforeClass), Dynamic Test Prioritization

#How to install?

Install maven dependency:

<!-- https://mvnrepository.com/artifact/org.testng/testng -->

<dependency>

<groupId>org.testng</groupId>

<artifactId>testng</artifactId>

<version>7.11.0</version>

<scope>test</scope>

</dependency>

**✅** TestNG Annotation Execution Order **(with Explanation)**

| **🔢 Order** | **Annotation** | **When It Runs** | **How Often** | **Scope** |
| --- | --- | --- | --- | --- |
| **1️⃣** | **@BeforeSuite** | **Before any tests in the entire suite** | **Once** | **Global** |
| **2️⃣** | **@BeforeTest** | **Before any test classes in a specific <test> block in testng.xml** | **Once** | **Per <test> tag** |
| **3️⃣** | **@BeforeClass** | **Before any test method in the current class** | **Once** | **Per class** |
| **4️⃣** | **@BeforeMethod** | **Before each test method** | **Repeated** | **Per method** |
| **✅** | **@Test** | **The actual test case** | **Repeated** | **Per method** |
| **5️⃣** | **@AfterMethod** | **After each test method** | **Repeated** | **Per method** |
| **6️⃣** | **@AfterClass** | **After all test methods in the current class** | **Once** | **Per class** |
| **7️⃣** | **@AfterTest** | **After all classes inside a <test> block** | **Once** | **Per <test> tag** |
| **8️⃣** | **@AfterSuite** | **After everything — the entire suite** | **Once** | **Global** |

Visual Flow Example:

@BeforeSuite // Global setup (e.g., connect to DB)

@BeforeTest // Module setup (e.g., launch browser)

@BeforeClass // Setup for this test class

@BeforeMethod

@Test (testMethod1)

@AfterMethod

@BeforeMethod

@Test (testMethod2)

@AfterMethod

@AfterClass // Cleanup for this test class

@AfterTest // Module cleanup (e.g., logout, close browser)

@AfterSuite // Global teardown (e.g., disconnect DB, generate report)

🔧 Optional Annotations

Annotation When It Runs Scope

@BeforeGroups Before the first method in a group runs Per group

@AfterGroups After the last method in a group runs Per group

You must use groups for these to work (i.e., @Test(groups = "regression")).

With Groups:  
**These annotations run only once per group, regardless of how many tests are inside**  
@BeforeSuite -- Prepare environment

@BeforeTest -- Login to Application

@BeforeClass -- Launch Browser

@beforeMethod -- Login to Application

@Test - Execute TestCase

@AfterMethod -- Quit Driver

@BeforeGroups -- Setup Smoke Tests

@beforeMethod -- Login to Application

@Test - Execute TestCase 2

@AfterMethod -- Quit Driver

@AfterGroups -- Clean Smoke Tests

@AfterClass -- clean test Data

@AfterTest - Logout from Application

@AfterSuite -- Clean environment

*@Test(priority = 1)*

@Test(priority = 0)

In TestNG, the priority attribute of the @Test annotation is used to control the **order in which test methods run**.

@Test(priority = 1)

public void testA() { ... }

@Test(priority = 2)

public void testB() { ... }

* Test methods are executed in ascending order of their priority value.
* Lower priority runs before higher priority.
* If no priority is defined, the default is 0.
* If multiple methods have the same priority, the order is not guaranteed (depends on method name or declaration order).
* Don't overuse priorities. Prefer dependsOnMethods if there's a clear dependency.
* It's good practice to write independent tests that don't rely on execution order unless truly required.

@Test(groups = {"smoke", "login-tests"})

Using **groups in TestNG** is essential for organizing and managing your test suite effectively, especially as your test base grows. Grouping allows you to categorize and selectively run tests based on tags like **smoke**, **regression**, **sanity**, **API**, etc.

* Use Meaningful and Consistent Group Names
  + Best Practice: Use group names that clearly indicate the test’s purpose or category. Common examples: "smoke", "sanity", "regression", "api", "ui", "db"
* Group at Method Level, Not Class Level (Unless All Methods Belong to Group)
* Maintain a list or table in your README or test documentation that maps: Group names, Purpose, Test examples
* Use Group Filters in Build Tools
  + mvn test -Dgroups=smoke
  + ✅ Helpful for pipeline automation and parallel test strategies.

*@Test(groups = {"smoke", "login-tests"})*

*public void testLoginSuccess() {*

*// test logic*

*}*

Define and Run Specific Groups in testng.xml

<groups>

<run>

<include name="regression"/>

<exclude name="broken-tests"/>

</run>

</groups>

This helps when you want to run only a subset of tests during CI/CD pipelines.

*DependsOn*

Using dependsOnMethods and dependsOnGroups in TestNG is a great way to control test execution order and set up test dependencies, especially when tests rely on the outcome of others. But if used improperly, they can make test suites fragile and hard to maintain.

* Use dependsOnMethods for Logical Sequences, Use this when one test logically depends on the success of another.

@Test

public void login() {

// login code

}

@Test(dependsOnMethods = {"login"})

public void accessDashboard() {

// only runs if login is successful

}

Use this to enforce flow like: login → perform action → logout

**Use dependsOnGroups for Structured Execution**

This helps when multiple tests belong to a group, and another test depends on all of them passing.

@Test(groups = "setup")

public void createUser() {}

@Test(groups = "setup")

public void setPermissions() {}

@Test(dependsOnGroups = {"setup"})

public void testLoginWithPermissions() {}

🔹 Helps organize tests modularly and avoid hardcoding method names.

Always Write Independent Tests Where Possible

Use Descriptive Method Names for Clarity

Make your dependencies clear by naming methods meaningfully:

@Test

public void userRegistration() {}

@Test(dependsOnMethods = {"userRegistration"})

public void userLogin() {}

🔹 Improves readability and maintainability of the suite.

**Document Dependencies Clearly**

Leave comments or maintain a flow diagram if the dependencies get complex. This helps in debugging and onboarding new team members.

@Test(timeOut = 3000)

Using timeOut in TestNG is a good way to fail slow-running tests that exceed expected execution time — ensuring performance thresholds and preventing test suite hang-ups. However, improper use can cause false negatives, flaky tests, or missed exceptions.

* **Use timeOut for Performance-Sensitive Tests**
* Only apply timeOut when the execution time is critical (e.g., API response tests, UI interactions, or backend jobs).

@Test(timeOut = 5000) // 5 seconds

public void testApiResponse() {

// Call to REST API expected to return within 5 sec }

Don't overuse timeouts on all tests — apply it where performance matters.

**Set Realistic Timeout Thresholds**

Avoid using arbitrary or overly tight timeouts. Base them on:

* Historical execution time
* System/environment variability
* Some margin for safe overhead

❌ Bad: timeOut = 1000 (if the test usually takes ~950 ms)

✅ Good: timeOut = 1500–2000 (adds buffer)

**Avoid Combining timeOut with Heavy Operations**

If your test includes DB setup, file I/O, or large computations, timeout values should consider that overhead.

**Be Cautious with expectedExceptions = ThreadTimeoutException.class**

Means you're expecting the test to fail by timing out, which is useful only in rare situations like:Testing watchdog behavior

Ensuring infinite loops or hangs are detected

Don't Use timeOut for Ordering or Synchronization

Use Logging to Understand Timeout Failures

@Parameters

* To define environment variable
* Use Parameters from testng.xml for Configuration Data
* **When to use:** For values that don’t change frequently like environment URL, username, password, browser type, etc.
* Best Practice:
  + Use @Parameters annotation along with testng.xml.
  + Add default values or null checks to prevent runtime errors.

<!-- testng.xml -->

<parameter name="browser" value="chrome" />

@Parameters("browser")

@BeforeClass

public void setup(String browser) {

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

// Launch Chrome

}

}

*Use testng.xml parameters only for simple config—not large test data.*

*Match parameter names in @Parameters and XML exactly.*

*Maintain clear naming to avoid confusion during debugging.*

***❌ Avoid These Pitfalls:***

* *❌ Avoid hardcoding parameters.*
* *❌ Avoid putting complex logic in @DataProvider.*
* *❌ Don’t mix too many @Parameters and @DataProvider without clarity.*

TestNG.xml file

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

*<suite name="Basic - Test" parallel="classes" thread-count="5" verbose="2" >*

<test name="Parameter Test" >

<parameter name="browser" value="edge"></parameter>

<parameter name="url" value="https://login.yahoo.com/"></parameter>

<parameter name="username" value="nav@gmail.com"></parameter>

<classes>

<class name="testNG.ParameterTest" />

*<class name="testNG.testDependsOn" />*

</classes>

</test>

<test name="PaymentTests">

<classes>

<class name="tests.PaymentTest"/>

</classes>

</test>

</suite>

Use Descriptive Names for Suites and Tests

Avoid generic names like "Suite1" or "Test1" — use meaningful identifiers:

<suite name="SmokeTestSuite">

<test name="UserLoginAPITests">

Assign Multiple Groups to a Test

You simply pass an array of group names to the groups attribute:

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

public void testLoginPageLoad() {

// Test logic

}

Group Related Tests Using <classes> and <packages>

For modular design, group related classes in one <test> block or organize via packages.

<classes>

<class name="tests.LoginTest"/>

<class name="tests.RegistrationTest"/>

</classes>

Or:

<packages>

<package name="tests.api"/>

</packages>

✅ Keeps XML compact and scalable.

**Use <groups> to Include/Exclude Categories**

Group your tests (e.g., smoke, regression, broken) and control which ones run:

<groups>

<run>

<include name="smoke"/>

<exclude name="broken"/>

</run>

</groups>

✅ Supports parallel or staged execution across CI/CD environments.

5. ✅ Use <parameter> to Externalize Configuration

Centralize config values (e.g., browser, env, credentials):

<parameter name="browser" value="chrome"/>

✅ Avoids hardcoding values in tests.

**6. ✅ Leverage Parallel Execution Wisely**

To speed up test runs:

<suite name="ParallelTests" parallel="tests" thread-count="4">

Options:

parallel="tests" → parallel across <test> blocks

parallel="classes" → parallel test classes

parallel="methods" → parallel test methods

⚠️ Make sure your tests are thread-safe when using parallel modes.

**Split testng.xml into Multiple Files for Large Projects**

Example:

smoke-testng.xml

regression-testng.xml

full-suite.xml

✅ Improves modularity and allows targeted test runs.

**Use Suite-Level Configuration for Timeouts or Listeners**

<suite name="MySuite" time-out="10000">

<listeners>

<listener class-name="listeners.TestListener"/>

</listeners>

</suite>

✅ Centralizes timeout handling and reporting.

**Use Comments for Documentation**

Briefly describe what each suite or test section does:

<!-- Runs all login and registration tests for smoke validation -->

<test name="AuthSmokeTests">

✅ Helps team members understand test intent quickly.

**Common Mistakes to Avoid**

**Mistake Recommendation**

Hardcoding values Use <parameter> and config files

Overusing one large testng.xml Split into modular suites

Missing test dependencies Use dependsOnGroups or dependsOnMethods properly

Using parallel incorrectly Ensure thread safety before enabling

**Summary: What Makes a Good testng.xml**

Naming Use descriptive names for suite and tests

Organization Use <classes>, <packages>, and <groups>

Config Externalize values using <parameter>

Execution control Use parallel, thread-count, and listeners

Modularity Split files for smoke/regression/full suite

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

<suite name="First Test" parallel="classes" thread-count="2" time-out="30000" verbose="1" >

<test name="Some Groups" >

<groups>

<run>

<include name="learning"></include>

<include name="ui"></include>

</run>

</groups>

<classes>

<class name="testNG.testDependsOn"></class>

<class name="testNG.TestExceptionTimeout"></class>

<class name="testNG.SeleniumDummyTest"></class>

<class name="testNG.ParameterTest"></class>

<class name="testNG.testSampleTestNGClass"></class>

<class name="testNG.testTestNgGroups"></class>

</classes>

</test>

<test name="Second Test">

<groups>

<run>

<include name="learning"></include>

<include name="ui"></include>

<include name="selenium"></include>

<include name="annotations"></include>

</run>

</groups>

<classes>

<class name="testNG.testDependsOn"></class>

<class name="testNG.SeleniumDummyTest"></class>

<class name="testNG.TestExceptionTimeout"></class>

<class name="testNG.ParameterTest"></class>

<class name="testNG.testSampleTestNGClass"></class>

</classes>

</test>

</suite>

@Test(invocationCount = 5)

@Test(invocationCount = N) tells TestNG to run the same test method N times.

It’s used to:

Use Case Description

Idempotency testing Ensure method behaves consistently on repeated calls

Flakiness detection Reveal intermittent failures

Concurrency validation Test thread safety using threadPoolSize

Performance warm-up Run code multiple times to measure stability or resource handling

Stress testing Simulate multiple calls under load

**Use threadPoolSize to test concurrency safety**

@Test(invocationCount = 10, threadPoolSize = 5)

public void parallelTest() {

// thread-safe code only!

}

☑️ Validate if the tested method works well under concurrent conditions  
☑️ Avoid shared mutable state, or use synchronized, Atomic\*, etc.

**Fail-fast for flaky tests**

Use TestNG retry logic **only if the failure is not consistent**.

**Don’t use retry logic unless testing flakiness**

Avoid retry analyzers unless you're explicitly checking resilience.

**Avoid shared mutable state**

If using shared data:

* Use ThreadLocal
* Use synchronized blocks
* Use concurrent collections (e.g., ConcurrentHashMap)

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

<suite name=*"First Test"* parallel=*"classes"* thread-count=*"2"* time-out=*"30000"* verbose=*"1"* >

<test name=*"Some Groups"* >

<groups>

<run>

<include name=*"learning"*></include>

<include name=*"ui"*></include>

</run>

</groups>

<classes>

<class name=*"testNG.testDependsOn"*></class>

<class name=*"testNG.TestExceptionTimeout"*></class>

<class name=*"testNG.SeleniumDummyTest"*></class>

<class name=*"testNG.ParameterTest"*></class>

<class name=*"testNG.testSampleTestNGClass"*></class>

<class name=*"testNG.testTestNgGroups"*></class>

</classes>

</test>

<test name=*"Second Test"*>

<groups>

<run>

<include name=*"learning"*></include>

<include name=*"ui"*></include>

<include name=*"selenium"*></include>

<include name=*"annotations"*></include>

</run>

</groups>

<classes>

<class name=*"testNG.testDependsOn"*></class>

<class name=*"testNG.SeleniumDummyTest"*></class>

<class name=*"testNG.TestExceptionTimeout"*></class>

<class name=*"testNG.ParameterTest"*></class>

<class name=*"testNG.testSampleTestNGClass"*></class>

</classes>

</test>

</suite>

@DataProvider