Unit Test Framework

A unit testing framework is a software tool designed to support the development and automated execution of tests that verify small units (such as functions or classes) of code for correctness. In the Scala ecosystem, the most widely used framework is **ScalaTest**, though others like **JUnit, MUnit**, and **ScalaCheck** are also available.

Overview: Unit Testing in Scala

1. What is a Unit Testing Framework?

- A unit testing framework automates the execution of test cases that validate individual code units.
- It supports **assertions**, **test discovery**, **setup/teardown routines**, and detailed reporting of results.

2. Popular Scala Unit Testing Frameworks

- ScalaTest (most flexible and widely used): Supports multiple testing styles such as **FunSuite**, **FlatSpec**, **WordSpec**, etc.
- **JUnit** (Java's classic test framework, interoperable with Scala)
- **MUnit** (lightweight, functional, inspired by ScalaTest)
- **ScalaCheck** (property-based testing)

3. Key Features of ScalaTest:

- Multiple styles: **FunSuite, FlatSpec, FunSpec, WordSpec, FreeSpec, PropSpec, FeatureSpec**
- Behavior-driven development (BDD) support
- **Assertions** and **matchers**
- **Mocking** and code coverage plugins
- Easy integration with build tools (sbt, Maven, Gradle)

ScalaTest Testing Styles (with code samples)

FunSuite (JUnit-like, Simple Unit Tests)

```scala
import org.scalatest.funsuite.AnyFunSuite

class CalculatorTest extends AnyFunSuite {
  test("multiply by zero should be zero") {
    assert(Calculator.multiply(5, 0) == 0)
  }

  test("division by zero should throw ArithmeticException") {
    intercept[ArithmeticException] {
      Calculator.divide(10, 0)
    }
  }
}
```

Use for: straightforward, independent tests.

FlatSpec (Readable “X should Y” format)

```scala
import org.scalatest.flatspec.AnyFlatSpec

class CalculatorSpec extends AnyFlatSpec {
  "Multiplication" should "return 0 when either factor is 0" in {
    assert(Calculator.multiply(7, 0) == 0)
  }
}
```

Use for: readable specifications, migration from XUnit/Java.

FunSpec (BDD/nested, structured)

```scala
import org.scalatest.funspec.AnyFunSpec
```

```
class CalculatorFunSpec extends AnyFunSpec {
  describe("Calculator") {
    it("should multiply numbers correctly") {
      assert(Calculator.multiply(3, 4) == 12)
    }
  }
}
```

Use for: hierarchical, human-readable behavioral specs.

WordSpec & FreeSpec (More expressive BDD)

```
import org.scalatest.wordspec.AnyWordSpec

class CalculatorWordSpec extends AnyWordSpec {
  "A Calculator" should {
    "return 0 when multiplying by zero" in {
      assert(Calculator.multiply(123, 0) == 0)
    }
  }
}
```

Use for: expressive, uniform, nested BDD.

PropSpec (Property-based)

```
import org.scalatest.propspec.AnyPropSpec

class CalculatorPropSpec extends AnyPropSpec {
  property("multiplying any Int by zero is zero") {
    for (i <- -1000 to 1000) {
      assert(Calculator.multiply(i, 0) == 0)
    }
  }
}
```

Use for: programmatically generated, property-based tests.

FeatureSpec (acceptance/feature-driven)

```
import org.scalatest.featurespec.AnyFeatureSpec

class CalculatorFeatureSpec extends AnyFeatureSpec {
  Feature("Multiplication") {
    Scenario("Multiplying any number by zero results in zero") {
      assert(Calculator.multiply(55, 0) == 0)
    }
  }
}
```

Use for: user story or business scenario-driven testing.

Code Coverage & Mocking

- Code coverage plugins: integrated with sbt (Scoverage), IDE plugins
- Mocking libraries: ScalaMock, Mockito

Typical Workflow

1. Add ScalaTest to your sbt project:
    a. `libraryDependencies += "org.scalatest" %% "scalatest" % "3.2.19" %`
2. Write unit tests in `src/test/scala`
3. Run tests: `sbt test`

```
libraryDependencies += "org.scalatest" %% "scalatest" % "3.2.19" %
```

Official Website: https://www.scalatest.org/user_guide/selecting_a_style