Functional programming with Scala

Unit and Integration Testing

Said BOUDJELDA

Senior Software Engineer @SCIAM
GitHub @bmscomp

efrei, 2025

Table of Contents

Why Testing Matters

- **Confidence**: Code works as expected
- **Regression Prevention**: Catch breaking changes
- **Documentation**: Tests describe behavior
- **Refactoring Safety**: Change code without fear
- **Design Feedback**: Tests reveal design issues

Good tests are your safety net

Testing Pyramid

Level	Speed	Scope
Unit Tests	Fast	Single function/class
Integration Tests	Medium	Multiple components
E2E Tests	Slow	Full system

Rule: More unit tests, fewer integration tests, minimal E2E

Popular Testing Frameworks

- **ScalaTest**: Most popular, many styles
- **MUnit**: Lightweight, fast compilation
- **uTest**: Minimal boilerplate
- **Specs2**: BDD-style specifications
- **ZIO Test**: For ZIO applications

We'll focus on ScalaTest and MUnit

ScalaTest Setup

```
// build.sbt
libraryDependencies += "org.scalatest" %% "scalatest"
    % "3.2.15" % Test

class CalculatorSpec extends AnyFlatSpec with Matchers
    :
  "A Calculator" should "add two numbers" in {
    val result = Calculator.add(2, 3)
    result should be(5)
  }
```

Basic ScalaTest structure

Testing Pure Functions

```scala
object MathUtils:
  def factorial(n: Int): Int =
    if n <= 1 then 1 else n * factorial(n - 1)

class MathUtilsSpec extends AnyFlatSpec with Matchers:
  "factorial" should "calculate correctly" in {
    MathUtils.factorial(5) should be(120)
  }
```

Testing pure functions is straightforward

Property-Based Testing

```
import org.scalatestplus.scalacheck.
    ScalaCheckPropertyChecks

class MathPropertiesSpec extends AnyPropSpec with
    ScalaCheckPropertyChecks:
  property("addition is commutative") {
    forAll { (a: Int, b: Int) =>
      MathUtils.add(a, b) should equal(MathUtils.add(b
          , a))
    }
  }
```

Generate test cases automatically

Testing with Mocks

```
trait UserRepository:
  def findById(id: Long): Option[User]

class UserServiceSpec extends AnyFlatSpec with
    MockitoSugar:
  "UserService" should "find user" in {
    val mockRepo = mock[UserRepository]
    when(mockRepo.findById(1L)).thenReturn(Some(user))
  }
```

Integration Testing Concepts

- **Database Integration**: Test with real/embedded DB
- **HTTP Integration**: Test REST API endpoints
- **Message Queues**: Test async communication
- **External Services**: Test third-party integrations
- **Container Testing**: Use Testcontainers

Integration tests verify component interaction

Database Integration Testing

```
class UserRepositoryIntegrationSpec extends
    AnyFlatSpec:
  val db = Database.forConfig("h2mem")
  val userRepo = new UserRepository(db)

  "UserRepository" should "save users" in {
    val user = User(0L, "Alice", "alice@test.com")
    val saved = Await.result(userRepo.save(user), 2.
        seconds)
    saved.name should be("Alice")
  }
```

HTTP API Testing

```
class UserApiSpec extends AnyFlatSpec with
    ScalatestRouteTest:
  val userService = mock[UserService]
  val routes = new UserRoutes(userService).routes

  "UserAPI" should "create user" in {
    Post("/users", userData.parseJson) ~> routes ~>
        check {
      status should be(StatusCodes.Created)
    }
  }
```

Test HTTP endpoints in isolation

Testcontainers Integration

```
import com.dimafeng.testcontainers.PostgreSQLContainer
import com.dimafeng.testcontainers.scalatest.
    TestContainerForEach

class PostgresIntegrationSpec extends AnyFlatSpec
  with TestContainerForEach:

  override val containerDef = PostgreSQLContainer.Def
      ()

  "PostgreSQL" should "work with real database" in {
    withContainers { postgres =>
      val userRepo = new UserRepository(postgres.
          jdbcUrl)
      val result = Await.result(userRepo.save(user),
          3.seconds)
    }
  }
```

Testing Best Practices

- **AAA Pattern**: Arrange, Act, Assert
- **One Assertion**: Test one thing at a time
- **Descriptive Names**: Clear test method names
- **Independent Tests**: No test interdependence
- **Fast Execution**: Keep unit tests fast
- **Test Data Builders**: Use factories for setup
- **Clean Resources**: Proper setup/teardown

Good tests are readable and maintainable

Test Data Builders

```
class UserBuilder:
  private var name: String = "John Doe"
  def withName(name: String): UserBuilder = { this.
      name = name; this }
  def build(): User = User(1L, name, "john@example.com
      ")

// Usage: UserBuilder().withName("Alice").build()
```

Testing Async Code

```
class AsyncServiceSpec extends AnyFlatSpec with
    ScalaFutures:
  "AsyncService" should "handle futures" in {
    val service = new AsyncUserService()
    val futureResult = service.findUserAsync(1L)

    whenReady(futureResult) { user =>
      user.name should be("Alice")
    }
  }
```

Handle asynchronous operations properly

Testing with ZIO

```scala
object UserServiceSpec extends ZIOSpecDefault:
  def spec = suite("UserService")(
    test("should create user") {
      for {
        service <- ZIO.service[UserService]
        user    <- service.createUser("Alice", "alice@
            test.com")
      } yield assertTrue(user.name == "Alice")
    }
  ).provide(UserServiceLive.layer)
```

ZIO Test provides functional testing

Test-Driven Development (TDD)

- **Red**: Write failing test first
- **Green**: Write minimal code to pass
- **Refactor**: Improve code while keeping tests green
- **Benefits**: Better design, 100% test coverage
- **Challenges**: Requires discipline and practice

TDD drives better software design

TDD Example

```scala
// 1. Red - Write failing test
class StringUtilsSpec extends AnyFlatSpec:
  "reverse" should "reverse a string" in {
    StringUtils.reverse("hello") should be("olleh")
  }

// 2. Green - Make it pass
object StringUtils:
  def reverse(s: String): String = s.reverse
```

Write test first, then implementation

Behavior-Driven Development (BDD)

- **Given-When-Then**: Structure test scenarios
- **Readable Tests**: Business-friendly language
- **Collaboration**: Bridge between technical and business
- **Living Documentation**: Tests as specifications
- **ScalaTest Support**: Feature and scenario DSL

BDD focuses on behavior and outcomes

BDD with ScalaTest

```
class UserRegistrationSpec extends FeatureSpec with
    GivenWhenThen:
  feature("User Registration") {
    scenario("Valid user signs up") {
      given("a valid email address")
      val email = "user@example.com"

      when("user registers")
      val result = UserService.register(email)

      then("user should be created")
      result should be(defined)
    }
  }
```

Test Doubles

- **Mocks**: Verify interactions and behavior
- **Stubs**: Return predefined responses
- **Fakes**: Working implementations for testing
- **Spies**: Record calls to real objects
- **Dummies**: Objects passed but never used

Choose the right test double for your needs

Stubs vs Mocks

```
// Stub - returns data
val stubRepo = stub[UserRepository]
when(stubRepo.findById(1L)).thenReturn(Some(user))

// Mock - verifies behavior
val mockRepo = mock[UserRepository]
service.deleteUser(1L)
verify(mockRepo).delete(1L)
```

Stubs provide data, mocks verify behavior

Parameterized Tests

```
class MathSpec extends AnyFlatSpec with
    TableDrivenPropertyChecks:
  val examples = Table(
    ("input", "expected"),
    (0, 1), (1, 1), (5, 120)
  )

  forAll(examples) { (input, expected) =>
    MathUtils.factorial(input) should be(expected)
  }
```

Test multiple inputs efficiently

Test Fixtures

```scala
trait UserFixture:
  def withUsers[T](testCode: List[User] => T): T = {
    val users = List(
      User(1L, "Alice", "alice@test.com"),
      User(2L, "Bob", "bob@test.com")
    )
    testCode(users)
  }

class UserServiceSpec extends AnyFlatSpec with
   UserFixture:
  "UserService" should "process users" in withUsers {
     users =>
    // Test with fixture data
  }
```

Performance Testing

- **Load Testing**: Normal expected load
- **Stress Testing**: Beyond normal capacity
- **Spike Testing**: Sudden load increases
- **Volume Testing**: Large amounts of data
- **Tools**: JMeter, Gatling, ScalaMeter

Ensure your application scales

ScalaMeter Example

```
import org.scalameter._

class PerformanceSpec extends AnyFlatSpec:
  "List operations" should "be performant" in {
    val sizes = Gen.range("size")(1000, 5000, 1000)

    val time = measure {
      val list = (1 to 1000).toList
      list.map(_ * 2).filter(_ > 100)
    }

    time should be < 10.millis
  }
```

Mutation Testing

- **Concept**: Introduce bugs to test your tests
- **Mutants**: Modified versions of your code
- **Kill Rate**: Percentage of mutants caught
- **Benefits**: Reveals weak tests
- **Tools**: Stryker4s, PIT for Scala

Test quality matters more than coverage

Contract Testing

- **Consumer-Driven**: Consumers define expectations
- **Provider Testing**: Verify contract compliance
- **Independent Deployment**: Test service boundaries
- **Microservices**: Essential for distributed systems
- **Tools**: Pact, Spring Cloud Contract

Test service interactions safely

Snapshot Testing

```scala
class ApiResponseSpec extends AnyFlatSpec:
  "API" should "return expected JSON" in {
    val response = api.getUser(1L)
    val json = response.asJson.spaces2

    // Compare with saved snapshot
    json should matchSnapshot("user-response.json")
  }
```

Capture and compare complex outputs

Testing Microservices

- **Service Tests**: Individual service testing
- **Contract Tests**: API compatibility
- **Component Tests**: Service with dependencies
- **End-to-End Tests**: Full system scenarios
- **Chaos Testing**: Failure resilience

Layer your testing strategy

Testing Configuration

```
// application-test.conf
database {
  url = "jdbc:h2:mem:test"
  driver = "org.h2.Driver"
}

class ConfigSpec extends AnyFlatSpec:
  "Config" should "load test settings" in {
    val config = ConfigFactory.load("application-test"
        )
    config.getString("database.url") should include("
        h2:mem")
  }
```

Separate test configuration

Continuous Testing

- **Fast Feedback**: Run tests on every change
- **Test Categories**: Unit, integration, acceptance
- **Parallel Execution**: Speed up test suites
- **Flaky Test Management**: Identify and fix unstable tests
- **Test Reporting**: Visibility and metrics

Automate testing in your pipeline

Test Organization

- **Directory Structure**: Mirror production code
- **Naming Conventions**: Consistent test naming
- **Test Categories**: Use tags and suites
- **Shared Utilities**: Common test helpers
- **Test Resources**: Manage test data files

Organize tests for maintainability

Error Handling in Tests

```
class ErrorHandlingSpec extends AnyFlatSpec:
  "Service" should "handle errors gracefully" in {
    assertThrows[ValidationException] {
      UserService.createUser("", "invalid-email")
    }
  }

  it should "return proper error messages" in {
    val result = UserService.validateEmail("invalid")
    result.left.value should include("Invalid email")
  }
```

Testing Anti-Patterns

- **Ice Cream Cone**: Too many E2E tests
- **Testing Implementation**: Test behavior, not internals
- **Fragile Tests**: Brittle tests that break easily
- **Slow Tests**: Long-running test suites
- **Mystery Guest**: Hidden test dependencies

Avoid these common testing mistakes

Test Coverage and Metrics

- **Line Coverage**: Percentage of code executed
- **Branch Coverage**: All decision paths tested
- **Mutation Testing**: Test quality assessment
- **Performance Testing**: Load and stress testing
- **Security Testing**: Vulnerability assessment

Tools: scoverage, sbt-scoverage, ScalaMeter

Continuous Integration

- **Automated Testing**: Run tests on every commit
- **Parallel Execution**: Speed up test suites
- **Test Categories**: Unit, integration, acceptance
- **Quality Gates**: Coverage thresholds
- **Test Reports**: Visibility into test results

CI/CD pipeline ensures code quality

Conclusion

- **Test Early**: Write tests as you code
- **Test Often**: Automate everything
- **Test Smart**: Focus on critical paths
- **Test Fast**: Keep feedback cycles short
- **Test Real**: Use integration tests strategically

Remember: Good tests enable confident refactoring and feature development

Thank You!

Questions?

Said BOUDJELDA
Senior Software Engineer @SCIAM
GitHub: @bmscomp