Testing in Future Space
Why you needn’t Await for the Future[ScalaTest]

Bill Venners
Artima, Inc.
Escalate Software, LLC
Northeast Scala Symposium
March 4, 2016
// Java’s original Servlet API encouraged blocking
protected void doGet(HttpServletRequest req, HttpServletResponse resp);
// Java’s original Servlet API encouraged blocking
protected void doGet(HttpServletRequest req, HttpServletResponse resp);

// Can block in Play if you don’t care about what others think
def index = Action { request =>
  val futureInt = Future { intensiveComputation() }
  val result = Await.result(futureInt, 30 seconds) // blocks
  Ok("Got result: " + result)
}
// Java’s original Servlet API encouraged blocking
protected void doGet(HttpServletRequest req, HttpServletResponse resp);

// Can block in Play if you don’t care about what others think
def index = Action { request =>
  val futureInt = Future { intensiveComputation() }
  val result = Await.result(futureInt, 30 seconds) // blocks
  Ok("Got result: " + result)
}

// Can return a future response to Play
def index = Action.async {
  val futureInt = Future { intensiveComputation() }
  futureInt.map(i => Ok("Got result: " + i))
}
// Good use case for blocking on futures is testing

test("This test blocks") {
    val futureInt = Future { intensiveComputation() }
    val result = Await.result(futureInt, 30 seconds) // blocks
    result should be (42)
}

Three Rules of Reactive Programming
Three Rules of Reactive Programming

1. NEVER EVER BLOCK!
Three Rules of Reactive Programming

1. NEVER EVER BLOCK!

2. NEVER EVER BLOCK!

3. Well, maybe it is OK to block sometimes in your tests.
Three Rules of Reactive Programming

1. NEVER EVER BLOCK!

2. NEVER EVER BLOCK!

3. Well, alright maybe it is OK sometimes to block in your tests.
// Good use case for blocking on futures is testing

test("This test blocks") {
  val futureInt = Future { intensiveComputation() }
  val result = Await.result(futureInt, 30 seconds) // blocks
  result should be (42)
}

// Why? If we can return a future response to a web
// framework, why can’t we return a future assertion to
// a test framework?
Good use case for blocking on futures is testing

test("This test blocks") {
  val futureInt = Future { intensiveComputation() }
  val result = Await.result(futureInt, 30 seconds) // blocks
  result should be (42)
}

Why? If we can return a future *response* to a web
// framework, why can’t we return a future *assertion* to
// a test framework?

test("This test blocks") {
  val futureInt = Future { intensiveComputation() }
  futureInt.map(i => result should be (42))
}
1. You can’t block!
import org.scalatest.FunSuite
import org.scalatest.concurrent.ScalaFutures
import scala.concurrent.Future
import scala.scalajs.concurrent.JSEExecutionContext.Implicits.runNow

class SampleServiceTest extends FunSuite with ScalaFutures {

  test("getData") {

    val x = SampleService.getData(""")

    assert(x.futureValue.contains("total_rows"))

  }
}
Currently in org.scalatest.

```scala
// Currently in org.scalatest.
Suite:

def withFixture(test: () => Outcome): Outcome = {
  test()
}

// Users can override in their own suites:
override def withFixture(test: () => Outcome): Outcome = {
  // Setup fixture
  try test()
  finally { /* cleanup fixture */ }
}
```

ScalaTest 2.x
// Currently in org.scalatest.Suite:
def withFixture(test: () => Outcome): Outcome = {
  test()
}

// Users can override in their own suites:
override def withFixture(test: () => Outcome): Outcome = {
  // Setup fixture
  try super.withFixture(test)
  finally { /* cleanup fixture */ } 
}
def withFixture(test: () => Outcome): Outcome

// Users can make SuiteMixin traits that override withFixture:
trait SeveredStackTraces extends SuiteMixin {
  this: Suite =>

  abstract override def withFixture(test: NoArgTest): Outcome = {
    super.withFixture(test) match {
      case Exceptional(e: StackDepth) => Exceptional(e.severedAtStackDepth)
      case o => o
    }
  }
}
ScalaTest 2.x Summary

- Users can define withFixture methods.

- can compose withFixture(() => Outcome) methods by stacking traits.

- According to the types, the test has already completed once the test function returns.
type Assertion = Succeeded.type

scala> val x = 1
x: Int = 1

scala> assert(x == 1)
res3: org.scalatest.Assertion = Succeeded

scala> assert(x == 2)
org.scalatest.exceptions.TestFailedException: 1 did not equal 2
...

scala> x should equal (1)
res5: org.scalatest.Assertion = Succeeded

scala> x should equal (2)
org.scalatest.exceptions.TestFailedException: 1 did not equal 2
...

ScalaTest 3.0.x

Outcome << sealed trait >>

Succeeded << object >>
Failed(
  ex: Throwable
)
Canceled(
  ex: TestCanceledException
)
Pending << object >>
class SampleServiceSuite extends AsyncFunSuite {

  test("getData") {

    val future = SeedService.getData"

    future map { sd => assert(sd.contains("total_rows")) } 
  }

  // Note: Result type of test is Future[Assertion],
  // though we also provide an implicit conversion from
  // Assertion to Future[Assertion]
def withFixture(test: () => Outcome): Outcome = {
  test()
}

This won’t work for async styles, because:

- According to the types, the test has already completed once the test function returns.
// Now in org.scalatest.TestSuite:
def withFixture(test: () => Outcome): Outcome = {
  test()
}

// In org.scalatest.AsyncTestSuite:
def withFixture(test: () => FutureOutcome): FutureOutcome = {
  test()
}
// SuiteMixin traits that overrode withFixture will need to be changed:

trait SeveredStackTraces extends TestSuiteMixin { this: TestSuite =>

  abstract override def withFixture(test: NoArgTest): Outcome = {
    super.withFixture(test) match {
      case Exceptional(e: StackDepth) => Exceptional(e.severedAtStackDepth)
      case o => o
    }
  }
}

ScalaTest 3.0.x
// Users can override in their own async suites:
override def withFixture(test: () => FutureOutcome): FutureOutcome = {
    // Setup fixture
    complete {
        super.withFixture(test)
    } lastly {
        // cleanup fixture
    }
}
// Has result type StringIndexOutOfBoundsException
intercept[StringIndexOutOfBoundsException] { 
  "hi".charAt(3)
}

// Has result type Assertion
assertThrows[StringIndexOutOfBoundsException] { 
  "hi".charAt(3)
}

ScalaTest 3.0.x

Added assertThrows in 3.0
// Wouldn’t work
future map { sd => assertThrows[Exception] { … } }

// Wouldn’t work
assertThrows[Exception] {
  future
}
ScalaTest 3.0.x

Added recoverTo methods in AsyncSuite in 3.0.x

// Has result type Future[IllegalStateException]
recoverToExceptionIf[IllegalStateException] {
  emptyStackActor ? Peek
}

// Has result type Future[Assertion]
recoverToSucceededIf[IllegalStateException] {
  emptyStackActor ? Peek
}
ScalaTest 3.0.x

- Tests execute one after another
- Default **SerialExecutionContext**
- We tried to make async consistent with sync
- Before & After work
- **ParallelTestExecution** works, even on Scala.js!
- Plan to release 3.0 final for ScalaDays NYC

Lots more to the story
Q $\implies$ A