Volver

U3.2.1 - Introducción a Kotest

Volver

Rafael Alberti
Instituto de Educación Secundaria

Volver

Introducción a Kotest

Volver

¿Qué es Kotest?

- Framework de pruebas escrito en Kotlin.
- Multiplataforma: JVM, JS y Native.
- Compuesto por tres subproyectos:
  - Núcleo de pruebas.
  - Biblioteca de aserciones.
  - Pruebas de propiedad.
- Compatible con otros frameworks.
- Usaremos plataforma JVM.

Volver

Configuración en JVM

- Usa la plataforma JUnit.
- En `build.gradle.kts` se incluyen dependencias:

```
dependencies {
    testImplementation(kotlin("test"))
    testImplementation("io.kotest:kotest-runner-junit5-jvm:5.5.4")
    testImplementation("io.kotest:kotest-runner-junit5:5.5.4")
    testImplementation("io.kotest:kotest-assertions-core:5.5.4")
    testImplementation("io.kotest:kotest-property:5.5.4")
}
```

Volver

Estilos de prueba en Kotest

Volver

Estilo BDD (BehaviorSpec)

- Usa bloques `given`, `when`, `then`.
- Similar a pruebas de comportamiento.

```
class CardPaymentTests : BehaviorSpec({
    given("I have sufficient balance") {
        `when`("I make a card payment") {
            then("The card payment should be successful") {
                // test code
            }
        }
    }
})
```

Volver

Estilo ShouldSpec

- Usa `should` para describir expectativas.
- Se agrupan con bloques `context`.

```
class PaymentTests : ShouldSpec({
    context("CardPayments") {
        should("Make a card payment") {
            // test code
        }
    }
})
```

Volver

Estilo FeatureSpec

- Usa `feature` y `scenario`.
- Inspirado en pruebas tipo Cucumber.

```
class HomePageTests : FeatureSpec({
    feature("signup") {
        scenario("should allow user to signup with email") {
            // test code
        }
    }
})
```

Volver

Estilo DescribeSpec

- Usa `describe` e `it`.
- Estilo común en JS y Ruby.

```
class PaymentTests : DescribeSpec({
    describe("CardPayments") {
        it("Should make a card payment") {
            // test code
        }
    }
})
```

Volver

Aserciones en Kotest

Volver

¿Qué son las aserciones?

- Verifican condiciones esperadas tras ejecutar un test.
- Kotest tiene su propia biblioteca de aserciones.
- Dos categorías:
  - Core matchers.
  - External matchers.
- Permiten comparaciones expresivas y precisas.

Volver

Ejemplos de aserciones core

```
result.shouldBe(expected)
result.shouldBeTrue()
result.shouldBeTypeOf<Double>()
result.shouldContain("substring")
```

- Comparación directa de resultados.
- Validación de tipos, textos, mapas, fechas, archivos.
- Uso sencillo y claro.

Volver

Pruebas de excepciones

Volver

Validación de excepciones esperadas

- Uso de `shouldThrow<T>()` para capturar errores.
- Permite validar tipo y contenido del mensaje.

```
val exception = shouldThrow<ValidationException> {
    // test code
}
exception.message should startWith("Invalid input")
```

Volver

Ganchos de ciclo de vida

Volver

Uso de `beforeTest` y `afterTest`

- Ejecutan lógica antes o después de cada prueba.
- Similares a setup/teardown de JUnit.

```
class TransactionStatementSpec : ShouldSpec({
    beforeTest {
        // Preparación
    }
    afterTest { (test, result) ->
        // Limpieza
    }
})
```

Volver

Pruebas basadas en datos

Volver

Uso de `withData` y clases de datos

- Similares a pruebas parametrizadas.
- Reutiliza lógica de test con distintas entradas.

```
data class TaxTestData(val income: Long, val taxClass: TaxClass, val expe

withData(
    TaxTestData(1000, ONE, 300),
    TaxTestData(1000, TWO, 350)
) { (income, taxClass, expected) ->
    calculateTax(income, taxClass) shouldBe expected
}
```

Volver

Pruebas no deterministas

Volver

Uso de `eventually`

- Útil para funciones asincrónicas o resultados tardíos.
- Reintenta la condición durante un tiempo definido.

```
eventually({
    duration = 5000
    interval = FixedInterval(1000)
}) {
    transactionRepo.getStatus(120) shouldBe "COMPLETE"
}
```

Volver

Integración con Mocking

Volver

Uso de `mockk` con Kotest

- Kotest no tiene biblioteca propia de mocks.
- Se recomienda usar `mockk`.

```
val provider = mockk<ExchangeRateProvider>()
every { provider.rate("USDEUR") } returns 0.9

val service = ExchangeService(provider)
service.exchange(Money(1200, "USD"), "EUR") shouldBe 1080
```

Volver

Cobertura de pruebas

Volver

Uso de Jacoco con Kotest

- Genera informes de cobertura tras las pruebas.
- Integración sencilla en `build.gradle.kts`.

```
tasks.test {
    finalizedBy(tasks.jacocoTestReport)
}
```

- Informe HTML en `$buildDir/reports/jacoco/test`.

Volver

Agrupación por etiquetas

Volver

Uso de anotaciones `@Tags`

- Permite ejecutar o excluir pruebas según contexto.
- Útil para ignorar tests lentos en Pull Requests.

```
@Tags(NamedTag("SlowTest"))
class SlowTests : ShouldSpec({})
```

Volver

Conclusión y recursos

Volver

Resumen y próximos pasos

- Kotest ofrece múltiples estilos y flexibilidad.
- Aserciones claras y ganchos potentes.
- Soporte para pruebas no deterministas y parametrizadas.
- Integración con herramientas externas como MockK y Jacoco.
- Ejemplos disponibles en GitHub.

Volver