Testing with Kotlin

Patterns and practices from the trenches

Who am I

I’m Bruno, CTO @ WATERDOG, we do digital innovation and product development projects for clients in traditional industries (aerospace, mining, automotive, etc.);

Talk outline

- Why test?
- Why use Kotlin?
- Technical deep dive
  - Leveraging Kotlin
  - Terminology
  - Libraries & unit test examples
  - Integration test examples
- Key take-aways

“There's no such thing as a free lunch”

Fact of life

Why test?

Why test

- Yup, there's an up-front/setup cost: remember no free lunches;
- Deliver "value" faster (more features, less bugs - however you want to measure it);
- Safer to iterate and modify your code - this is especially important for products;
- Provides real-value as a process: operational implementation of design principles, e.g. SOLID;
- How can we reduce the cost of setting up testing? Better tools :)

Why use Kotlin?

Why use Kotlin?

- Code should be optimized for reading: clarity, conciseness and expressiveness;
- JVM ecosystem (libraries, accumulated ops knowledge);
- Nice language features:
  - Smart casts
  - Final variables
  - Data classes
  - Sealed classes
  - String templating
  - Collections API
- Nice language features (cont.):
  - Separate interfaces for reading and writing on data structures;
  - Structured concurrency;
  - High order functions and Lambdas;
  - Extension functions;
  - Null-safety (ish) - beware platform types;
- Tooling (intellij rocks!)
- Happily using Kotlin in production for the last 18 months.

“Context is king”

The following examples are not universal truths

Technical deep dive

Before we jump into some code...

- The following code examples, are related to a fictional company that installs and operates IoT devices on industrial facilities to collect environmental readings (e.g. temperature);

Leveraging Kotlin

AKA put the compiler to work

Leveraging Kotlin: Data classes

Data classes

- Optimized for holding data: great for value objects, DTOs;
- Auto-generated methods: equals, hashCode, copy, toString

Caveats:

- Data classes are final: you cannot inherit from them;
- Constructor shenanigans: needs to have a non-empty primary constructor;

```
data class IoTDevice(

    val deviceId: IoTDeviceId,

    val label: String,

    val status: DeviceStatus,

    val installedAt: InstallationLocation,

    val sensors: List<Sensor>

){ … }
```

Leveraging Kotlin: Sealed classes

Sealed classes:

- Express limited class hierarchies - think of it as Enum on steroids;
- The sealed class behaves like an abstract class, with the key difference that the compiler knows what classes belong to that hierarchy;
- Plays really well with the `when` keyword and smart-casts;

```
sealed class Sensor:

data class TemperatureSensor(...)

data class HumiditySensor(...)

data class CarbonMonoxideSensor(...)
```

Leveraging Kotlin: Smart casts

Smart casts:

- Tends to work pretty well, and avoids ugly type-juggling;

```
// Sealed classes usage:

when(sensor){

    is TemperatureSensor -> sensor.readTemp()

    is HumiditySensor -> sensor.readHumidity()

    is CarbonMonoxideSensor -> sensor.readCO()

    // No need for an else branch;

    // Compiler will scream if you miss any of

    // the classes in the hierarchy

}
```

Leveraging Kotlin: Contracts (1)

Contracts:

- Contracts allow a function to explicitly describe its behavior in a way which is understood by the compiler.

Caveats:

- Custom contracts are still experimental

```
// Old-school style, it works but...

fun configure(props: Map<String, String>){

    val requiredProp = props["required-prop"]

    if(requiredProp != null){

        requiredProp.substring(1)

        ....

    }

}
```

Leveraging Kotlin: Contracts (2)

Contracts:

- Contracts allow a function to explicitly describe its behavior in a way which is understood by the compiler.

Caveats:

- Custom contracts are still experimental

```
// This will not fail due to an NPE, but beware nesting

// especially when there are multiple properties

fun configure(props: Map<String, String>){

    props["required-prop"]?.let{ requiredProp ->

        requiredProp.substring(1)

        ....

    }

}
```

Leveraging Kotlin: Contracts (3)

Contracts:

- Contracts allow a function to explicitly describe its behavior in a way which is understood by the compiler.

Caveats:

- Custom contracts are still experimental

```
// Note that the compiler will know that 'requiredProp'

// not null - and the intent is super clear

fun configure(props: Map<String, String>){

    val requiredProp = props["required-prop"]

    requiredNotNull(requiredProp){ "A message" }

    requiredProp.substring(1)

    ....

}
```

Terminology

Some not-so-precise terms I’m gonna use

Terminology

- Unit tests
  - Run very fast;
  - Doesn’t do I/O (database, network, FS);
  - Can be executed at the same time as other tests;
  - Run assertions only for a single concept/entity;
- Integration tests
  - Test how different components interact;
  - Can do I/O;
  - May be slower to execute;

Libraries & unit test examples

Libraries: JUnit5

- URL: https://junit.org/junit5/
- Why:
  - Better exception handling
  - Nested tests
  - Parameterized tests
  - Tags
  - Test instance lifecycle control
  - Dynamic tests
  - JUnit Vintage test engine to ease migration from JUnit 3 or 4

- Nice comparison with JUnit4: https://sormuras.github.io/blog/2018-09-13-junit-4-core-vs-jupiter-api.html
- Gradle:

```
testImplementation("org.junit.jupiter:junit-jupiter-api:5.4.1")

testImplementation("org.junit.jupiter:junit-jupiter-params:5.4.1")

testRuntimeOnly("org.junit.jupiter:junit-jupiter-engine:5.4.1")
```

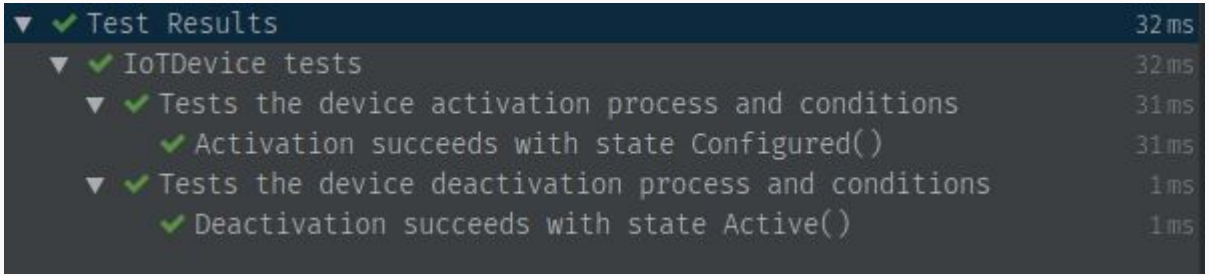
@Nested

Example of the @Nested annotation and readable method names using backticks. Use it with @DisplayName, for even better readability.

```
@DisplayName("IoTDevice tests")
class IoTDeviceTests {

    @Nested
    @DisplayName("Tests the device activation process and conditions")
    inner class DeviceActivationProcessTests {
        @Test
        fun `Activation succeeds with state Configured`() { … }
    }

    @Nested
    @DisplayName("Tests the device deactivation process and conditions")
    inner class DeviceDeactivationProcessTests {
        @Test
        fun `Deactivation succeeds with state Active`() { … }
}
```



Exception handling

Improved exception handling vs. JUnit4

```
@Test
fun `Activation fails on an active device`() {
    // Given a device with the Active status
    val device = IoTDevice(
        "my-device",
        "A test device",
        DeviceStatus.Active,
        location(),
        listOf(Sensors.carbonMonoxide())
    )

    // Expect: IllegalStateException
    val exception = assertThrows<IllegalArgumentException> {
        device.activate()
    }
    exception.message `should equal` "A nice error message"
}
```

@ParameterizedTest (1)

Example of the @ParameterizedTest annotation. There can be different data souces:

https://junit.org/junit5/docs/current/user-guide/#writing-tests-parameterized-tests

```
// Test with all the values of the DeviceStatus enum, except 'Configured'
@ParameterizedTest
@EnumSource(
    value = DeviceStatus::class,
    mode = EnumSource.Mode.EXCLUDE,
    names = ["Configured"]
)
fun `Activation fails on the other states`(status:DeviceStatus){
    // Given a device
    val device = IoTDevice(
        "my-device",
        "A test device",
        status,
        location(),
        sensors()
    )

    // Expect: IllegalStateException
    assertThrows<IllegalArgumentException> { device.activate() }
}
```

@ParameterizedTest (2)

Example of the @ParameterizedTest annotation. There can be different data souces:

https://junit.org/junit5/docs/current/user-guide/#writing-tests-parameterized-tests

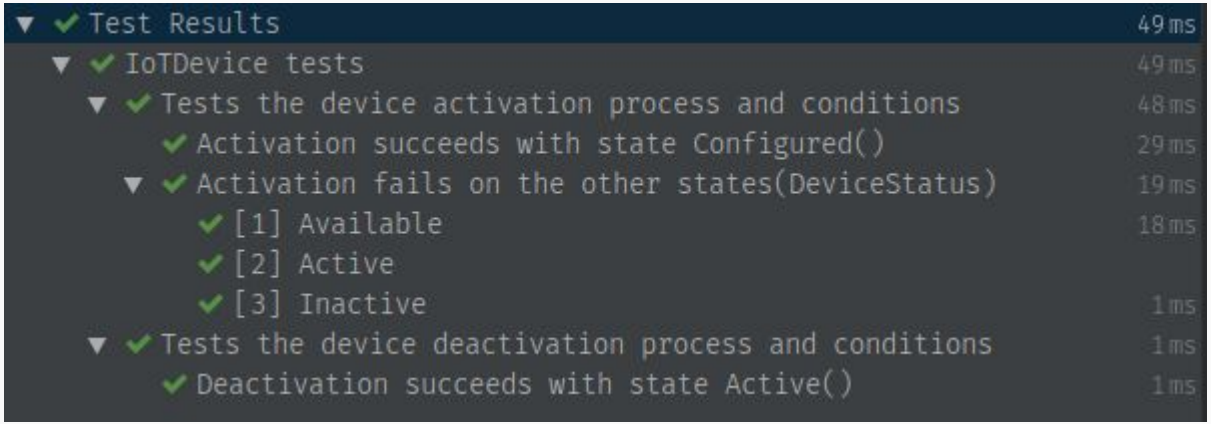
```
// CsvSource is also very useful - this is a contrived example :P
@ParameterizedTest
@CsvSource("Inactive", "Active", "Available")
fun `Activation fails on the other states`(statusName: String) {
    // Given a device
    val status = DeviceStatus.valueOf(statusName)
    val device = IoTDevice(
        "my-device",
        "A test device",
        status,
        location(),
        listOf(Sensors.carbonMonoxide())
    )

    // Expect: IllegalStateException
    assertThrows<IllegalArgumentException> { device.activate() }
}
```

@ParameterizedTest (3)

Example of the @ParameterizedTest annotation. There can be different data souces:

https://junit.org/junit5/docs/current/user-guide/#writing-tests-parameterized-tests



@Tag

The @Tag is useful to mark tests as unit or integration tests.

We can then use gradle to check whether or not to run the tests.

In this case, we will only run the “Slow” test in our CI environment.

IoTDeviceServiceTests.kt

```
@Tag("Slow")
@DisplayName("IoTDeviceService integration tests")
class IoTDeviceServiceTests { … }
```

Build.gradle.kts

```
val isCiBuild = System.getenv("CI_ENV") == "true"
tasks {
    withType<Test> {
        useJUnitPlatform{
            if(!isCiBuild) {
                excludeTags("Slow")
            }
        }
    }
}
```

@TestInstance

By default each test method will run on its own test class instance. We can easily override that and run all the tests under the same class instance.

This is might be useful if we need to rely on some class state.

```
// This is the default
@TestInstance(TestInstance.Lifecycle.PER_METHOD)
class IoTDeviceTests { ... }

// We could potentially use this if we needed
// to use some shared state in out tests.
@TestInstance(TestInstance.Lifecycle.PER_CLASS)
class IoTDeviceTests { ... }
```

Libraries: Mockk

- URL: https://mockk.io/
- Why:
  - Mocking for kotlin
  - Support for suspending functions

Mockk

Basic mocking examples, returns and answers

https://mockk.io/#answers

```
// Example target interface
interface IoTDeviceRepository {
   fun save(device: IoTDevice)
   fun get(deviceId: IoTDeviceId): IoTDevice?
}


// Simple mocking
val repo = mockk<IoTDeviceRepository>()


// Functions that return Unit
every { repo.save(any()) } just Runs


// Return a canned result (AKA stub)
every {repo.get(any())} returns IoTDevice(...)


// Capture input and match with a bunch of known instances
val id = slot<IoTDeviceId>()
val device1 = IoTDevice(...)
val device2 = IoTDevice(...)
val devices = mapOf(
   "device1" to device1,
   "device2" to device2
)

every { repo.get(capture(id)) } answers { devices[id.captured] }
```

Mockk

Basic mocking examples, matchers and throwing errors

https://mockk.io/#answers

```
// Example target interface
interface IoTDeviceRepository {
   fun save(device: IoTDevice)
   fun get(deviceId: IoTDeviceId): IoTDevice?
}


val repo = mockk<IoTDeviceRepository>()
// Matcher: when the argument is 'FORCE_ERROR' bad things happen
every { repo.get(cmpEq("FORCE_ERROR")) }
    throws IllegalArgumentException("Invalid id")


// Example Test
@Test
fun `When an invalid device id is requested an exception is thrown`(){
   // Given a properly configured service
   val repo = deviceRepo()

   // Expect: an IllegalArgumentException when an invalid id is called
   assertThrows<IllegalArgumentException> {
       repo.get("FORCE_ERROR")
   }
}
```

Mockk

Mocking suspension functions

```
// Example target interface:
interface InstallationLocationRepository {
   suspend fun get(id: LocationId): InstallationLocation
}

val repo = mockk<InstallationLocationRepository>()
val stubLocation = InstallationLocation(
   locationId = "change",
   clientId = "Test Client",
   physicalLocation = PhysicalLocation(0.0, 0.0, 0.0)
)

val idSlot = slot<LocationId>()
coEvery { repo.get(capture(idSlot)) } answers {
    stubLocation.copy(locationId = idSlot.captured)
}
```

Libraries: Kluent

- URL: https://github.com/MarkusAmshove/Kluent
- Why:
  - Assertion library
  - Provides more readable assertions

Kluent

Fancy assertions, if that's your thing

```
// JUnit - Check if two things are equal
assertEquals("a test device", result.label)
// Kluent
result.label `should equal` "a test device"

// JUnit - nullability check
assertNotNull(result)
// Kluent
result.`should not be null`()

// JUnit - collection check
sensors.forEach{ sensor -> assert(result.sensors.contains(sensor))}
// Kluent
result.sensors `should contain all` sensors
```

Libraries: Misc

- Ktlint: https://github.com/pinterest/ktlint
  - Lint/code style
- Spek: https://spekframework.org/
- Hamkrest: https://github.com/npryce/hamkrest

Integration test examples

Testing REST APIs with Ktor

- Ktor is a lightweight framework to write asynchronous servers;
- Let's see some tests for the following endpoint:

```
# OpenAPI 3.0
/device:
  post:
    operationId: createDevice
    summary: "Creates a new device"
    tags: [ "DeviceManagement" ]
    requestBody:
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/CreateDeviceCommand'
    responses:
      "201":
        description: "Created"
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/IoTDevice'
```

Ktor - test setup

First we need to setup our test application, via the usual Ktor DSL

```
// Implementation of the handler for the endpoint
// Note the dependency inversion principle in action
fun Route.iotDeviceApi(service: IoTDeviceService) {
    post("/device"){ ... }
}

fun Application.testModule(service: IoTDeviceService) {
    install(DefaultHeaders)
    install(ContentNegotiation) {
        register(
            ContentType.Application.Json,
            JacksonConverter(JsonSettings.mapper)
        )
    }
    install(Routing) {
        // We register our endoint with the server
        iotDeviceApi(service)
    }
}
```

Ktor

Second - we need to write the actual test, and wire up our test application with the code we wish to test, using the provided `withTestApplication` call

```
// The service implementation we're going to use
private val service = IoTDeviceServiceImpl(
        deviceRepo(),
        locationRepo()
)

// Just an "alias" to our jackson object mapper
private val mapper = JsonSettings.mapper

@Test
fun `Successful creation of an IoT Device`(): Unit = withTestApplication({
    testModule(service)
}, {
    // Given a create device command
    val cmd = CreateIoTDeviceCommand(
        label = "A new device",
        installationLocationId = UUID.randomUUID().toString(),
        sensors = listOf(Sensors.carbonMonoxide())
    )

    // When the endpoint is invoked with the command
    // serialized as JSON and a content type of 'application/json'
    with(handleRequest(HttpMethod.Post, "/device") {
        setBody(mapper.writeValueAsString(cmd))
        addHeader(
            HttpHeaders.ContentType,
            ContentType.Application.Json.toString()
        )
    }) {
        //Then we get a 201 as well as a new IoTDevice as JSON
        response.status() `should equal` HttpStatusCode.Created
        val receivedDevice: IoTDevice = mapper.readValue(response.content!!)
        with(receivedDevice){
            label `should equal` cmd.label
            sensors `should contain same` cmd.sensors
            installedAt.locationId `should equal` cmd.installationLocationId
        }
    }
})
```

Ktor

The `withTestApplication` tends to add a lot of clutter, let's get rid of it...

```
// Moved some of the boilerplate into its own function so our tests
// look less cluttered
fun <R> testWith(
    service: IotDeviceService,
    body: TestApplicationEngine.() -> R
) = withTestApplication({
    testModule(service)
}, body)

@Test
fun `Successful creation of an IoT Device`(): Unit = testWith(service) {
    // Given a create device command
    val cmd = CreateIoTDeviceCommand(
        label = "A new device",
        installationLocationId = UUID.randomUUID().toString(),
        sensors = listOf(Sensors.carbonMonoxide())
    )
    ...
}
```

Testing with Testcontainers

- Let’s say we want to persist our data to Cassandra;
- In order to test we need access to a Cassandra cluster;
- That’s a PITA to setup, so we need something easier to use…
- Enter Testcontainers (https://www.testcontainers.org/)

- Requirements:
  - Docker (on your dev machine or CI env)

Testcontainers

This is the class we want to test, and these are the dependencies we need to use Testcontainers and Cassandra

CassandraIoTDeviceRepository.ts

```
// Note the constructor: we're going to need to grab a cassandra session
// and since we want to run an integration test, we won't use Mockk

class CassandraIoTDeviceRepository(
    private val session: CqlSession,
    private val locationRepo: InstallationLocationRepository
) : IoTDeviceRepository {
    override fun save(device: IoTDevice) { ... }

    override fun get(deviceId: IoTDeviceId): IoTDevice? { ... }
    }
}
```

build.gradle.kts

```
dependencies {

    ...

    testImplementation("org.testcontainers:junit-jupiter:1.11.3")

    testImplementation("org.testcontainers:cassandra:1.11.3")

}
```

Testcontainers

As well as some boilerplate to ease our lives...

CassandraTestContainer.kt

```kotlin
// This is a very ugly :(
// See: https://github.com/testcontainers/testcontainers-java/issues/318
class KCassandraContainer : CassandraContainer<KCassandraContainer>()

// Singleton to hold our cassandra container, we don't want to
// be creating a new container for each test that needs this
object CassandraTestContainer {
    val instance by lazy { initCassandra() }

    private fun initCassandra(): KCassandraContainer {
        val instance = KCassandraContainer()
        instance.setWaitStrategy(CassandraQueryWaitStrategy())
        instance.startupAttempts = 1
        instance.withInitScript("database.cql")
        instance.start()
        return instance
    }
}
```

Testcontainers

In order to usen Testcontainers, we need to annotate our test with the @Testcontainers annotation

Our class also has some boilerplate to setup the CqlSession

```
@Tag("Slow")
@Testcontainers
@DisplayName("Cassandra IoT Device repository integration test")
class CassandraIoTDeviceRepositoryTests {

    private val cassandra = CassandraTestContainer.instance

    private val containerHost = InetSocketAddress(
        cassandra.containerIpAddress,
        cassandra.getMappedPort(9042)
    )

    private val session by lazy {
        CqlSession.builder()
            .withLocalDatacenter("datacenter1")
            .addContactPoint(containerHost)
            .withKeyspace("kt_lx")
            .build()
    }

    @Test
    fun `An IoTDevice can be successfully persisted`() {... }
}
```

Testcontainers

At this point our test is not that surprising :)

```
@Test
fun `An IoTDevice can be successfully persisted`() {
    // Given a correctly configured repo
    val repo = CassandraIoTDeviceRepository(session, locationRepo)
    val location = runBlocking {
       locationRepo.get(UUID.randomUUID().toString())
    }

    // And an IoTDevice to persist
    val originalDevice = IoTDevice(
        UUID.randomUUID().toString(),
        "A test device",
        DeviceStatus.Available,
        location,
        listOf(Sensors.humidity())
    )

    // When the save method is called on the repo
    repo.save(originalDevice)

    // The said IoTDevice is persisted, that is,
    // we can fetch it from the database
    val storedDevice = repo.get(originalDevice.deviceId)
    storedDevice `should equal` originalDevice
}
```

Testcontainers

Using other types of containers

```
// A generic Alpine linux container
class KAlpineContainer : GenericContainer<KAlpineContainer>("alpine:3.2") {
    init {
        this
         .withExposedPorts(80)
         .withEnv("MAGIC_NUMBER", "42")
         .withCommand(
                "/bin/sh", "-c",
                "while true; do echo \"\$MAGIC_NUMBER\" | nc -l -p 80; done"
            )
    }
}
```

```
// same deal with our singleton object we had with Cassandra
object AlpineServer {
    val instance by lazy { init() }

    private fun init():KAlpineContainer {
        val instance = KAlpineContainer()
        instance.setWaitStrategy(HostPortWaitStrategy())
        instance.startupAttempts = 1
        instance.start()
        return instance
    }
}
```

```
@Test
fun `Alpine test`() {
    val server = AlpineHttpServer.instance
    val address = InetSocketAddress(server.containerIpAddress, server.getMappedPort(80))
    val socket = Socket()
    socket.connect(address)
    ...
}
```

Key take-aways

Stuff to retain and think about

Key take-aways

- One of the largest benefits of testing has to do with the implications it has in the process of building software;
- Kotlin brings to the table a nice set of features that allow it to be used in production (language, ecosystem, tooling);
- There is no "one true way" of delivering software, however as software becomes ever more prevalent, as practitioners, we need to up our game, especially in terms of our safety culture - so test your stuff ;)

Thanks!

Bruno Felix

CTO - WATERDOG

bruno.felix@waterdog.mobi
@felix19350