Java Test

itera

Types of test

- Unit
- Integration
- DB
- Web
- Component
- System
- Performance/Load/Speed
- Smoke
- Regression
- User Acceptance
- Black Box
- Behavioural
- Compliance
- ...

Types of test - workshop

Way too many to cover in one workshop - we will take a look at the following:

- Design for testing (brief)
- Unit test (JUnit - including parametric test)
- Matchers (JUnit/Hamcrest/AssertJ)
- Mocking/Spying (Mockito)
- Integration (Spring)
- DB (Spring + flyway)
- Kotlin (kotest and mockk)

Design for testing

- Follow SOLID - well designed code is usually easier to test
- Injection - prefer constructor to setters or injected properties
- Unit tests give more value where they test logic rather than boilerplate
- Structure of a test
- Naming conventions
- Use of `@VisibleForTesting` [1]

[1] VisibleForTesting simply documents why access to a method or value is more open than it should be. It does nothing for enforcement - but can be used by static code analysis.

Injection

Classes often have dependencies. These can be provided in several ways - e.g.:

- Constructor parameters
- Setter methods
- Annotated properties

By using constructor properties - it forces you to create a complete instance - this is good practice both for coding in general as well as testing - for example - the instance property can be set final.

Annotation based properties are even worse - how do you set them from the test code without starting the annotation system (for example spring).

```java
class ConstructorInjected {
  // The internal property can be final
  private final Service service;

  // In spring 4.3 - classes with a single constructor no longer need the @Autowired annotation
  public ConstructorInjected(Service service) {
    this.service = service;
  }
}

class SetterInjected {
  // We lose the final marker
  private Service service;

  public void setService(Service service) {
    this.service = service;
  }
}

class AnnotatedProperty {
  @Autowired
  private final Service service;
}
```

Structure

GivenWhenThen [2]

This came originally from behaviour driven development - but it applies well to most tests. The test structure is simply:

- Given - setup your initial state
- When - the action to be tested
- Then - the expected results

[2] https://martinfowler.com/bliki/GivenWhenThen.html

Naming Conventions

- Both class and test method names are used in the test results so they need to be descriptive.
- Certain frameworks pick files based on filename [3]. For example failsafe which we will see under integration tests. A common convention is <Name>Test for unit, <Name>IT for integration test (this is configurable).
- Test method names should be consistent. [3]
- Kotlin test method names are perhaps one of the few places where we can use this form of method naming to advantage (gives a very readable test result output):

```
fun `short description of the test`() {}
```

[3] Prior to annotation use this was often the way testing frameworks distinguished between tests, test suites, integration tests etc. The same applied to methods - setup, teardown and which methods were actual tests.

Unit test with JUnit 5

- The test function is marked with `@Test`
- We use the built in JUnit assertEquals

Example: SimpleJunitTest

Assertions

There are multiple ways to assert in tests. JUnit has its inbuilt set. Some other popular libraries are Hamcrest and AssertJ.

- Hamcrest - `assertThat(result, equalTo(5))`
- AssertJ - `assertThat(result).isEqualTo(5)`

Which to use us a matter of personal preference and/or project standards.

Parametric

A parametric test allows us to reuse the same test with a range of different test data sets.

The test method is annotated to tell JUnit that it is parameterized and also where to get the data from.

There's a bunch of different sources available[4] - we'll use MethodSource.

Example: SimpleParametricTest

[4] https://junit.org/junit5/docs/current/api/org.junit.jupiter.params/org/junit/jupiter/params/provider/package-summary.html

Unit tests in a real application

Issue - we need to provide a full implementation of the repository to test a non-related method.

Things to consider:

- Poor separation of concerns?
- Mocking (we'll see this later)?
- In this instance - the calculation method could be static

Example: DummyJavaServiceTest

Parametric tests in a real application

The issues here are the same as for the simple test.

Example: DummyJavaServiceParametricTest

Mocking

In the above two examples - mocking is not really the solution - they should likely be refactored with SOLID in mind.

However - there are situations were mocking a dependency allows you to test a higher level component.

For example - we want to test a service - but to have test control over what the repository responds. This allows for unit testing of the service without starting up the entire application [5]

[5] We will do this in integration testing

Simple Mocking example

JUnit needs some help to allow for mocking so we add an extension to the test class and set up our mock dependency:

```
@ExtendWith(MockitoExtension.class)
class DummyJavaServiceMockTest {
  @Mock
  DummyRepository dummyRepository;
}
```

Example: DummyJavaServiceMockTest

We can now use that repository in our tests and tell it what to do under certain conditions e.g.:

```
@Test
void testServiceBackendCheck() {
    // When the repository isUp() is called then we will return value true
    when(dummyRepository.isUp()).thenReturn(true);

    // Instantiate test service with mock repo
    DummyJavaService service = new DummyJavaService(dummyRepository);

    // Test
    Assertions.assertThat(service.backendCheck()).isTrue();
}
```

Simple spying example - argument capture

We want to know something about an internal call that our test candidate makes.

For that we'll use argument capture.

Example: DataJavaServiceMockTest

As well as using a mocked repository we add a Captor:

```
@Captor
ArgumentCaptor<Long> captor;
```

We can use this when configuring the mock to capture an argument value:

```
when(repository.findById(captor.capture()))
  .thenReturn(Optional.of(new DataJava(1L, "qwerty")));
```

And we can test that this was in fact called with the correct value:

```
Assertions.assertThat(captor.getValue()).isEqualTo(1L);
```

Verification

We can also check that certain expectations match - how many times a mocked method is called, order of calls etc.

For the previous example - we can verify that the findById method is called only once:

```
verify(repository, times(1)).findById(any());
```

Here we use `any()` as matcher - we could also choose to verify with a concrete parameter value.

Integration tests

These are tests that spin up the application and test it under a running condition.

We use the failsafe plugin for maven for these.

One of the default filename matchers for failsafe is **IT.java - we will use that.

Integration with spring

For integration tests with spring we can use:

```
@ExtendWith(SpringExtension.class)
```

This annotation also allows us to specify what spring configuration we want to use.

We will actually use this for the DB tests later on - but as we are using spring boot - we can use the spring boot annotation that applies this extension as well as bootstrapping spring boot for us:

```
@SpringBootTest
```

Example: DummyJavaServiceIT

Spring boot with MockMvc

Spring boot test provides us with a mock mvc engine to test web calls to controllers.

Annotate the test class:

```
@SpringBootTest
@AutoConfigureMockMvc
```

and you get a MockMvc object you can use to call your application.

Example: DummyJavaControllerIT

DB testing

For this we will use h2 in memory db and flyway for db migrations.

The migrations are under src/main/resources rather than src/test/resources so that we can click around in the online db interface. However - you can use src/test/resources for test only data.

DB Console

Start the TestApplication then head to http://localhost:8080/h2

JDBC URL: jdbc:h2:mem:testdb
Username: sa
Password: empty

DB Repository test

We will use two annotations for this:

```
@ExtendWith(SpringExtension.class)
@DataJpaTest
```

Inject the repository you want to test:

```
@Autowired
private DataJavaRepository repository;
```

Example: DataJavaRepositoryIT

Kotlin

Kotlin can be used to create all the tests we have seen so far - e.g. compare:

- DataJavaRepositoryIT
- DataKotlinRepositoryIT

The same annotations and injection of repository is used. The only difference here is that we used kotest matchers rather than JUnit assertions.

Kotest Specs

Kotest also has multiple styles (specs) to choose between.

For the list (10 as of when this was written) see styles.md[6]

We'll take a look at FunSpec.

Example: DummyJavaServiceFunSpecTest

This is not quite the simplest structure - it uses init rather than the FunSpec constructor - but that allows for the beforeTest setup call.

[6] https://github.com/kotest/kotest/blob/master/doc/styles.md

Kotest with Mockk

Kotlin can also use Mockito and similar java mock libraries - but there is a nice kotlin one called mockk.

Two examples - one mock tests the DummyJavaService and the other the DataKotlinService:

- DummyJavaServiceMockkFunSpecTest
- DataKotlinServiceMockkFunSpecTest

Maven testing

All of the above tests can be run within a modern java IDE. However - we use a build system for our projects - most often maven (gradle can also be used in a similar fashion). This will also be how the tests are run when using a CI system.

There are three main sets of configuration in the pom.xml file.

- Surefire plugin - runs unit tests
- Failsafe plugin - runs integration tests
- Jacoco - generates code coverage

Surefire will run under mvn test, and failsafe under mvn verify [7]

Jacoco sets itself up under pre-integration-test and builds the result in post-integration-test so will also be triggered by verify.

[7] https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html