Kwik

Jul 31, 2020

Contents

CHAPTER 1

Kwik

Property-based testing library for Kotlin.

Main features:

- Test-engine agnostic
- Multiplatform
- No reflection
- Configurable built-in generators
- Easy way to create and combine generators
- Seeded generation for reproducible results

Planned features:

- Shrinking

Have a look at the *setup* and *usage*

1.1 Status

The project is incubating and its API is subject to changes.

Please give it a try and write a feedback in the issues or discuss on gitter.

1.2 How it looks like

```
class PlusOperatorTest {

    @Test
    fun isCommutative() = forAll { x: Int, y: Int ->
```

(continues on next page)

(continued from previous page)

```
        x + y == y + x
    }

    @Test
    fun isAssociative() = forAll(iterations = 1000) { x: Int, y: Int, z: Int ->
        (x + y) + z == x + (y + z)
    }

    @Test
    fun zeroIsNeutral() = forAll(seed = -4567) { x: Int ->
        x + 0 == x
    }
}
```

For more information, read *how to write tests* and have a look at the available *generators*

1.3 Motivation

Property based testing is great and very powerful. But despite the fact that many good libraries already exist, none of them fully fit my needs.

The known alternatives either:

- Are bound to a specific test-engine
- Can only be used when compiling kotlin to Java (and cannot be used in multi-platform projects)
- Relies on reflection, making the tests slower and make some errors detectable only at runtime
- Do not allow enough freedom and safety to customize existing generators
- Force the user to add unwanted dependencies in the classpath

1.4 Setup

Example of setup using gradle.

```
repositories {
    jcenter()
}

dependencies {
    testCompile("com.github.jcornaz.kwik:kwik-core-jvm:$kwikVersion")
}
```

For more information, read the *setup*

1.5 Contribute

See how to contribute

CHAPTER 2

Setup

2.1 Make sure to setup a test engine

Kwik is not a test-engine, but only an assertion library.

So before being able to use Kwik you have to setup a test-engine for your project. If the project is for the JVM (Java), you probably want to use Junit or Spek.

Note: If you choose to use kotest as a test-engine, be aware that it includes a similar property-based testing API.

In order to not get confused by mixing the two libraries, you may exclude the `kotlintest-assertions` artifact or introduce some rules in your IDE/linter to prevent usages of the package `kotlin.io.kotest.property`.

2.2 Add the required repository to your build system

- Stable versions are published on jcenter
- Alpha, beta and release-candidates are published on https://dl.bintray.com/kwik/preview

2.3 Add the artifact dependency

- The group id is `com.github.jcornaz.kwik`
- Pick the artifact id that suits your platform:
  - `kwik-core-common`
  - `kwik-core-jvm`
  - `kwik-core-linux`

  - `kwik-core-windows`

- Pick a version from: https://github.com/jcornaz/kwik/releases

2.3.1 Example with gradle for Kotlin/JVM

```
repositories {
    jcenter()
}

dependencies {
    testCompile("com.github.jcornaz.kwik:kwik-core-jvm:$kwikVersion")
}
```

2.4 Kotlin/JVM configuration

If you compile Kotlin to Java ByteCode, you must target Java 8 or above.

Here is how to configure it with gradle

```
tasks.withType<KotlinJvmCompile> {
    kotlinOptions {
        jvmTarget = "1.8"
    }
}
```

CHAPTER 3

Configuration

Kwik allow you to configure some defaults via system property (for Kotlin/JVM) or environment variable (Kotlin/JVM, or Kotlin/Native on linux),

Note when running Kotlin/JVM the system properties have precedence over the environment variable (in case they are both set)

3.1 Default number of iterations

By default Kwik will evaluate each property 200 times. (each time with different random inputs)

This default can configured by defining the system property `kwik.iterations` or environment variable `KWIK_ITERATIONS`.

That can be especially useful to define a different number of iteration on the CI server

For instance one may write the following gradle setup:

```
tasks.withType<Test> {
    if ("CI" in System.getenv()) {

        // On the CI take more time to try falsifying each property
        systemProperty("kwik.iterations", "10000")
    } else {

        // On the local setup allow the developer specify by command line using `-
↪Dkwik.iterations=`
        systemProperty("kwik.iterations", System.getProperty("kwik.iterations"))
    }
}
```

With the setup above each property would be evaluated 10‘000 times (with different random inputs) when test are executed on the CI server. (to make it work, the server needs to have a `CI` environment variable)

And any developer may run `./gradlew test -Dkwik.properties=10` if he wants a fast feedback loop, evaluating each property only 10 times.

Note: The number of iteration defined when invoking `forAll` has precedence over the system property.

See *how to choose number of iterations for specific property*

3.2 Default seed

By default Kwik will generate a different random seed for each property evaluation, leading to unpredicatable input.

That's generally desirable as over multiple build run, the test will cover more and more the domain of possible input.

But during debugging session, it is likely that one want perfectly reproducible builds. That can be achieved by defining a seed either on the *property evaluation* , or globally via the system property `kwik.seed` or the environment variable `KWIK_SEED`.

CHAPTER 4

Write property tests

4.1 Basic usage

To evaluate a property we must invoke the function `forAll` like this:

```
@Test
fun isCommutative() = forAll { x: Int, y: Int ->
    x + y == y + x
}
```

`forAll` Will generate random inputs and evaluate the content of the lambda 200 times. If the lambda return false, it will immediately throws an `AssertionError` making the test fail.

So the test pass only if the lambda returns true for 200 random inputs.

Note: Kwik can automatically generate values for `Int`, `Double`, `Boolean` and `String`.

For other types we have to *Create a custom generator*

4.2 Use assertions

If writing a lambda that return a boolean is not of your taste, you may alternatively use *checkForAll*. Instead of returning a boolean, we have to throw an exception in case of falsification.

Example:

```
@Test
fun isCommutative2() = checkForAll { x: Int, y: Int ->
    assertEquals(x + y, y + x)
}
```

This alternative can be especially useful to get more descriptive messages. In the example above, a falsification of the property would display the expected and actual values. Theses kind of messages cannot be provided when using *forAll*.

4.3 Choose the number of iterations

By default the property is evaluated 200 times[1]. But we can configure it by setting the argument `iteration`.

For instance, the following property will be evaluated 1000 times:

```
forAll(iterations = 1000) { x: Int, y: Int, z: Int ->
    (x + y) + z == x + (y + z)
}
```

4.4 Use a seed to get reproducible results

Because Kwik use random values, it is by definition non-deterministic. But sometimes we do want some determinism. Let's say, for instance we observed a failure on the CI server once, how can be sure to reproduce it locally?

To solve this problem, Kwik use seeds. By default a random seed is used and printed in the console. If we observe a failure in the CI, we simply look at the build-log to see what seed has been used, then we can pass the seed to `forAll` so that it always test the same inputs.

```
forAll(seed = -4567) { x: Int ->
    x + 0 == x
}
```

Note: The seed can be *set globally*

4.5 Customize generated values

Random input is good. But sometimes, we need to constraint the range of possible inputs.

That's why the function `forAll` accepts *generators*, and all built-in *generators* can be configured.

```
forAll(Generator.ints(min = 0), Generator.ints(max = -1)) { x, y ->
    x + y < x
}
```

4.6 Create a custom generator

But what if we want to test with input types which are not supported by Kwik, like domain-specific ones?

For this we can create a generator by implementing the interface `Generator`.

But most of the time it may be simpler to call `Generator.create`:

[1] The default number of iterations can be *configured via system property*

```
val customGenerator1 = Generator.create { rng ->
    CustomClass(rng.nextInt(), rng.nextInt())
}
```

For enums or finite set of values we can use `Generator.enum()` and `Generator.of()`:

```
val enumGenerator = Generator.enum<MyEnum>()

val finiteValueGenerator = Generator.of("a", "b", "c")
```

Note: You may reuse existing operator to build new ones. This can be done by calling `Genarator.genarate(Random)` of other operators, or by using the available *operators*

4.7 Add samples

Testing against random value is great. But often some value have more interest to be tested than others.

These edge-cases can be added to a generator with the function `withSamples`.

And since `null` and `NaN` are two quite common edge-case, there are dedicated `withNull` and `withNaN` operators.

```
val generator = Generator.ints().withSamples(13, 42)

val generatorWithNull = Generator.ints().withNull()
val generatorWithNaN = Generator.doubles().withNaN()
```

The samples have higher chance to be generated and will be tested more often.

Note: All built-in generators already have some samples included.

For instance `Generator.ints()` will generate `0`, `1`, `-1`, `Int.MAX_VALUE` and `Int.MIN_VALUE` often.

4.8 Skip an evaluation

Sometime we want to exclude some specific set of input. For that, we can call `skipIf` in the property evaluation block.

```
forAll { x: Int, y: Int ->
    skipIf(x == y)

    x != y
}
```

Be careful to not overuse it though as it may slow down the tests. Always prefer creating or configuring custom generators if you can.

4.9 Make sure that a condition is satisfied at least once

All theses random inputs are nice, but we may want to be sure that some conditions are met all the time.

For that, we can call `ensureAtLeastOne`. It will force the property evaluation run as many time as necessary, so that the given predicate gets true.

```
forAll { x: Int, y: Int ->

    // This forces the property to run as many times as necessary
    // so that we make sure to always test the case where x and y are both zero.
    ensureAtLeastOne { x == 0 && y == 0 }

    x * y == y * x
}
```

Be careful to not overuse it either as it may slow down the tests.

CHAPTER 5

Built-in generators

Kwik provide a collection of generators to satisfy a wide variety of uses-cases.

They are all available as extension functions on `Generator.Companion` so that we can find them easily and invoke them like this:

```
val generator = Generator.ints()
```

5.1 Primitives

`Generator.ints(min = Int.MIN_VALUE, max = Int.MAX_VALUE)` Generate integers. Includes the samples: `0`, `1`, `-1`, `min` and `max`.

Note that there are also `positiveInts`, `naturalInts`, `negativeInts` and `nonZeroInts` alternatives

`Generator.longs(min = Long.MIN_VALUE, max = Long.MAX_VALUE)` Generate longs. Includes the samples: `0`, `1`, `-1`, `min` and `max`.

Note that there are also `positiveLongs`, `naturalLongs`, `negativeLongs` and `nonZeroLongs` alternatives

`Generator.floats(min = -Float.MAX_VALUE, max = Float.MAX_VALUE)` Generate longs. Includes the samples: `0.0`, `1.0`, `-1.0`, `min` and `max`.

Note that `NaN`, `POSITIVE_INFINITY` and `POSITIVE_INFINITY` are not generated. To test theses, we can use `withSamples()` or `withNaN()`

Example: `Generator.floats().withNaN().withSamples(Float.POSITIVE_INFINITY)`

Note that there are also `positiveFloats`, `negativeFloats` and `nonZeroFloats` alternatives

`Generator.doubles(min = -Double.MAX_VALUE, max = Double.MAX_VALUE)` Generate doubles. Includes the samples: `0.0`, `1.0`, `-1.0`, `min` and `max`.

Note that `NaN`, `POSITIVE_INFINITY` and `POSITIVE_INFINITY` are not generated. To test theses, we can use `withSamples()` or `withNaN()`

Example: `Generator.doubles().withNaN().withSamples(Double.POSITIVE_INFINITY)`

Note that there are also `positiveDoubles`, `negativeDoubles` and `nonZeroDoubles` alternatives

`Generator.booleans()` Generate booleans

5.2 Strings

`Generator.strings(minLength = 0, maxLength = 50, charset = StringCharsets.printable, exclu`
Generate strings. Use the parameter `charset` and `exclude` to customize the characters which can be used.

Generation include empty ("") and blank (" ") strings as samples.

Note: `StringCharsets` provide few common set of characters such as `alpha`, `alphaNumeric` and others

It is there to help quickly configure the String generator.

By default, it will generate any printable characters.

5.3 Collections

`Generator.lists(elementGen = Generator.default(), minSize = 0, maxSize = 50)`
Generate lists. `elementGen` can be used to define the generator of the elements.

Generation include empty and singleton lists as samples

Note that there is also a `nonEmptyLists` alternative

`Generator.sets(elementGen = Generator.default(), minSize = 0, maxSize = 50)`
Generate sets. `elementGen` can be used to define the generator of the elements.

Generation include empty and singleton sets as samples

Will fail in it takes too much iteration to reach the `minSize` (so make sure the element generator can generate enough different values)

Note that there is also a `nonEmptySets` alternative

`Generator.maps(keyGen = Generator.default(), valueGen = Generator.default(), minSize = 0, r`
Generate sets. `keyGen` can be used to define the generator of the elements.

Generation include empty and singleton maps as samples

Will fail in it takes too much iteration to reach the `minSize` (so make sure the element generator can generate enough different values)

Note that there is also a `nonEmptyMaps` alternative

5.4 Sequences

`Generator.sequences(elementGen = Generator.default(), minSize = 0, maxSize = 50)`
Generate sequences. `elementGen` can be used to define the generator of the elements.

Generation include empty and singleton sequences as samples

Note that there is also a `nonEmptySequences` alternative

5.5 Enums

`Generator.enum<T>()` Create a generator for the given enum type `T`.

The enum must contains at least one enumeration.

5.6 Java

`Generator.uuids()` Create a generator for UUID

CHAPTER 6

Generator operators

Few operators are available as extension function on `Generator` to easily derive existing generators.

`withSamples(vararg samples: T, probability: Double)` Add the given samples into the generated values. The samples will have a higher probability to be generated than the other values.

That probability can be customized using the `probability` argument.

`withNull()` Add `null` into the generated values, making sure is is always tested

`withNaN()` Add `NaN` into the generated values, making sure is is always tested

(for double generators only)

`map(transform: (T) -> R)` Apply a transformation to all elements emitted by the source generator

`andThen(transform: (T) -> Generator<R>)` Like map, it applies a transformation to all elements emitted by the source generator. The only difference is that `transform` returns a `generator` instead of a value. You may see it like a `flatMap`.

`filter(predicate: (T) -> Boolean)` Filter elements emitted by the source generator, so that only elements matching the predicate are emitted.

Be aware that the property evaluation will then have to generate more values.

Always favor other method of creating a generators or at least make sure that most of values will pass the predicate.

`filterNot(predicate: (T) -> Boolean)` filter elements emitted by the source generator, so that only elements not matching the predicate are emitted.

Be aware that the property evaluation will then have to generate more values.

Always favor other method of creating a generators or at least make sure that most of values won't pass the predicate.

6.1 Combining exiting operators

`combineWith(other: Generator<B>, transform: (A, B) -> R)` Combine the generated values of both generators.

Generated values will start by a combination of the 5 first samples of both generators. Then samples of each generator have a higher probability to appear than other random values.

Not specifying the transform, will combine the value in pairs.

`plus(other: Generator<T>)` (can be used as `+`) Merge the generated values of both operators. (each generator having the same probability to used)

`frequency(vararg weightedGenerators: Pair<Double, Generator<T>>)` Returns a generator that randomly pick a value from the given list of the generator according to their respective weights.