Sunshine Documentation

Dmytro Serdiuk

May 25, 2020

Contents

Sunshine allows you to manage suits of your automated tests directly from Java code. It supports TestNG, JUnit4, and JUnit5.

CHAPTER 1

Concept

Modern build tools such as Maven or Gradle have a test phase of the build lifecycle to execute the unit tests of an application. This capability is used in the automated testing to manage the execution of automated tests (instead of unit tests). As the result, all automated tests are located under `src/test` as well as any test execution configurations are managed through the build tools.

The used solution has 3 main gaps:

1. **There is no place for unit tests as it is already owned by automated tests.** But sometimes unit tests are useful even for the automated testing as they allow to check some specific implementations to be used during the automation.
2. **Hard to configure tests suites.** This is because you need to run all unit tests at the same time during the project's build cycle. And build tools support it very well. But for automated tests, it is often required to run only a subset of tests (or even only single test). The build tools are not so good at doing this.
3. **There is no chance to create a JAR file and distribute automated tests like a standard Java archive.** That's why in addition to a Java installation, build tool and other dependencies are required to run the tests. And this is a pain especially if you are supporting the tests execution or trying to dockerize them.

What does Sunshine offer?

It wraps a test runner and allows programmatic configuration of the tests suites. This decouples automated tests execution from a build tool and moves a project with automated tests toward the regular Java project structure.

Want to see a small demonstration? Please check out the video

The code is located on https://github.com/extsoft/jcat. Feel free to check it out and make your own experiments.

Now you are ready to move further with Getting started.

CHAPTER 2

Getting started

You need to follow several steps to integrate Sunshine with your automated tests.

2.1 Step 1: Make proper project's structure

Suppose, there is a Maven project with automated tests

```
.
├── README.md
├── pom.xml
└── src
    ├── main
    └── test
```

Usually, the code is located under `src/main` directory and automated tests under `src/test`. According to *Sunshine's concept*, the code and automated tests have to live together in `src/main`.

So, please move everything to `src/main` except unit tests.

2.2 Step 2: Add Sunshine library

Based on the xUnit engine you are using, please select an appropriate library and add to your build configuration. Available libraries:

-
-
-

To find out more references, please visit *artifacts page*.

2.3 Step 3: Configure code packaging

The recommended packaging is an `uber-JAR` - also known as a `fat JAR` or `JAR with dependencies` - is a JAR file that contains not only a Java program but embeds its dependencies as well.

How to configure an uber-JAR depends on a build tool is used by a project. But regardless of a way of the configuration, a JAR's entry point has to be defined.

There are predefined classes to be used as the entry points for each test runner. However, Sunshine was designed to encourage you to create your own configurations based on the project's requirements. Please visit entry points page to find out more.

CHAPTER 3

Artifacts

Sunshine consists of 4 libraries

- `sunshine-core` provides interfaces and common implementations
- `sunshine-junit4` wraps JUnit 4 to allow the creation of entry points
- `sunshine-junit5` wraps JUnit 5 to allow the creation of entry points
- `sunshine-testng` wraps TestNG to allow the creation of entry points

Since `0.4.x` version Sunshine uses `org.tatools` group ID. If you need the earlier version please use `io.github.tatools` instead.

3.1 sunshine-core

Artifacts:

3.2 sunshine-junit4

The library is tested on JUnit 4 `4.11`.

Artifacts:

3.3 sunshine-junit5

The library is tested on Jupiter `5.5.2` and Platform `1.5.2`.

Artifacts:

3.4 sunshine-testng

The library is tested on TestNG `6.11`.

Artifacts:

CHAPTER 4

Entry points

An entry point is a Java class which is configured for a JAR and will be used when

```
java -jar my.jar
```

is executed.

4.1 Design custom entry point

The `org.tatools.sunshine.core` package provides 3 main interfaces which are a base for any Sunshine's wrapper:

- `Suite` defines a test suite and may be adapted to a test runner
- `Kernel` runs encapsulated `Suite` on desired test runner
- `Star` reports `Kernel`'s execution status to command line interface

Also, there is a `SunshineSuite` interface with several core implementations which allow seeking and filtering of the classes to be treated as tests. Sunshine uses a string representation of a class as an input to a filter which uses Java pattern matching to filter classes. For instance, there is a `LoginTest` class in `com.example.mypackage` package. It will be converted to `com.example.mypackage.LoginTest` and then put to a filter.

Each wrapper provides its own implementations at least for `Suite` and `Kernel`. For instance, if the classes located in `pro.extsoft.comments.tests` package have to be executed by TestNG and status of the execution has to be reported to the CLI, the code snippet will look like

```
new Sun(
    new TestNGKernel(
        new LoadableTestNGSuite(
            new RegexCondition("^pro.extsoft.comments.tests(.+)?")
        )
    )
).shine();
```

You can wrap it with any additional logic which is required for your tests execution. For instance, some input arguments can be read to generate test data before testing, etc.

If you need an example, please take a look for jcat repository.

4.2 Default TestNG entry point

Class: `org.tatools.sunshine.testng.Sunshine`

The class exposes the following behavior:

1. `java -jar my.jar` seeks for classes with word `Test` in the class name (a filter uses `(.+)(Test)([\w\d]+)?` regex), runs discovered tests, and reports results to the CLI.
2. `java -Dtests-regex="^(com.company.smoke)(.+)" -jar my.jar` does the same as the previous one except a class name has to match with `^(com.company.smoke)(.+)` regex.
3. `java -jar my.jar testng.xml` runs tests defined in `testng.xml` file (it can be either TestNG XML or TestNG YAML).

4.3 Default JUnit 4 entry point

Class: `org.tatools.sunshine.junit4.Sunshine`

The class exposes the same behavior as default TestNG entry point exposes except the last option. All tests will be executed using JUnit 4.

4.4 Default JUnit 5 entry point

Class: `org.tatools.sunshine.junit5.Sunshine`

The class exposes the same behavior as default TestNG entry point exposes except the last option. All tests will be executed using JUnit 5.

4.5 Code snippets for different use cases

4.5.1 Specify regex

There is a regex which has used to filter classes from current classpath.

TestNG sample

```
new Sun(
    new TestNGKernel(
        new LoadableTestNGSuite(
            new VerboseRegex(
                new RegexCondition("^my.package(.+)?")
            )
        )
    )
).shine();
```

JUnit 4 sample

```
new Sun(
    new Junit4Kernel(
        new JunitSuite(
            new VerboseRegex(
                new RegexCondition("^my.package(.+)?")
            )
        )
    )
).shine();
```

4.5.2 Specify classes directly

There are some classes which have to be treat as a test suite.

TestNG sample

```
new Sun(
    new TestNGKernel(
        new LoadableTestNGSuite(
            new SuiteFromClasses(
                TestNGTest1.class,
                TestNGTest2.class,
                TestNGTest3.class
            )
        )
    )
).shine();
```

JUnit 4 sample

```
new Sun(
    new Junit4Kernel(
        new JunitSuite(
            new SuiteFromClasses(
                JUnitTest1.class,
                JUnitTest2.class,
                JUnitTest3.class
            )
        )
    )
).shine();
```

4.5.3 Sequential execution of `Kernels`

There is a suite of tests which needs to be checked with several different listeners. For instance, if a listener is not configured, the tests will use some cache, otherwise, no cache before each test. And we have to run the suite with and without the listener.

There is a specific `Kernel` called `SequentialExecution` which allows execution of described use case:

```
final Kernel<ITestNGListener> suite = new TestNGKernel(....);
new Sun(
    new SequentialExecution<>(
```

(continues on next page)

(continued from previous page)

```
        suite,
        suite.with(new CleanCacheListener())
    )
).shine();
```

Note: Please read the *Concept* to understand how it works before moving to *Getting started*.

Want to contribute? Please visit GitHub repository.