TCtest 5.0

User Manual

Hugo Wood
2011-10-24

Contents

1 What is TCtest?

TCtest is an automated testing framework for Thales Control. Its primary goal is to run sequences of actions against one or several Thales Control installations. The user defines such sequences (referred to as workflows) and installations in configuration files, and then matches the formers with the latters in a test suite. The TCtest launcher analyzes these files, sets up Thales Control, runs all supplied tests, and finally, writes reports carrying the results.

1.1 The Test Tree

TCtest is built around 4 type of objects that form what is called the test tree. These 4 types are:

- services

  They are the leaves of the tree. They are incarnated by a Java class library with a main class that performs a very small action and returns a result indicating if this action was carried out with success or not.

- workflows

  Workflows are logical sequences of services that perform a coherent task. They can be seen as a way to aggregate services to form macro-services. They also add context to the services executions. A workflow may reference a service multiple times and a service may be referenced by several workflows. They are defined by users using XML.

- test cases

  Test cases are instanciations of a workflow, on top of which they add context. Several test cases may refer to the same workflow.

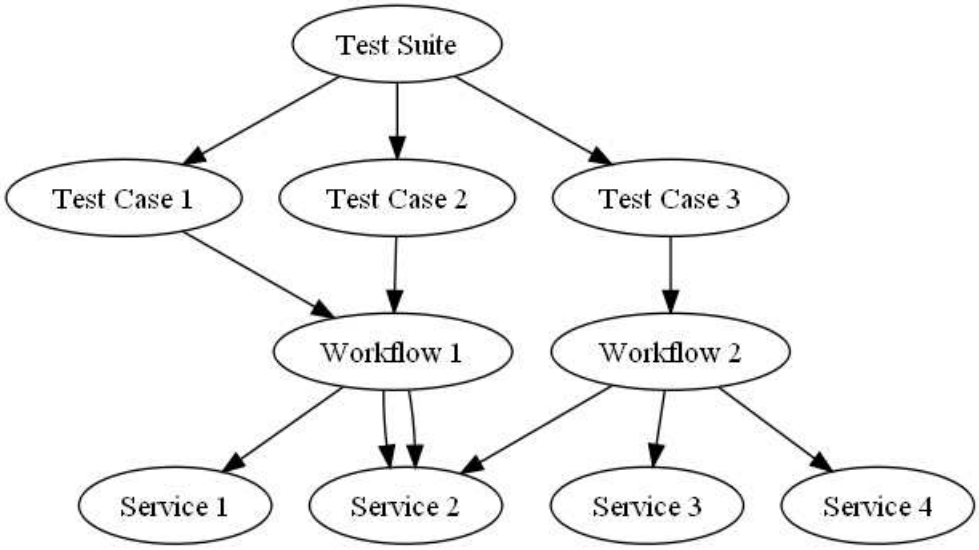
- the test suite

It is the root of the tree. A test suite can have any number of test cases. The test suite and its test cases are defined in one XML file.

Each node has a context, which is basically a set of name-value pairs. Contexts cascade down the tree and are ultimatly consumed by services code. If values with the same name are found, the child node has priority. Contexts are just like parameters for services. Therefore, most services require that some of these parameters have a value. Parameters are assigned when defining the tree. At which node a parameter is assigned is not significant, as long as all required parameters have all been assigned when the service is run. If not, an exception is thrown. To learn which parameters are required for a service, refer to its documentation. For built-in services, see Built-in Services.

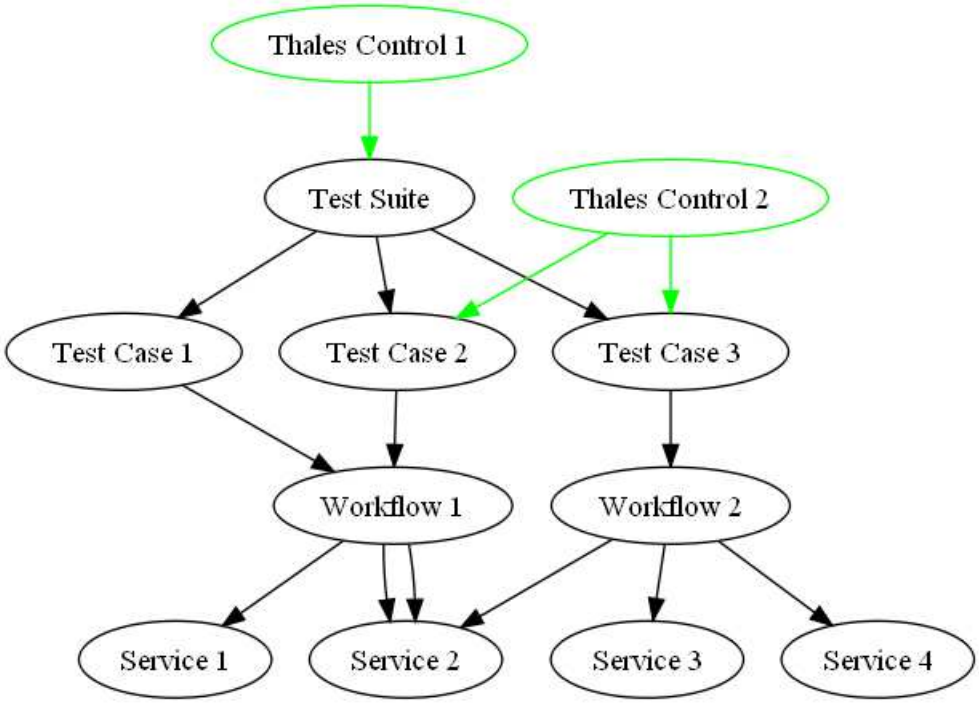
At startup, TCtest loads up the services it founds, parses the user XML files, builds the test tree, and runs it.

Here is an example of a test tree:



1.2 Where is Thales Control?

You may have noticed that the test tree does not involve Thales Control. This comes from the fact that even if TCtest was built for testing Thales Control, it can actually test anything. To test Thales Control, the tree is decorated with Thales Control instances that the user attaches to test suites or test cases. A test case can have no Thales Control, inherit a Thales Control from its parent suite, or have its own Thales Control. A Thales Control can be shared between test cases.



2 Setting Up

2.1 Global Settings

The TCtest launcher takes as an argument a path to an XML file that contains information about its environment (that is to say a series of paths to directories that the program should use to search for input) and global settings. The `settings` root element supports the following children:

- `installerRepository` (optional)

  A directory that serves as root when providing a relative path in the `installer` element of a Thales Control definition (see Defining Thales Control instances).

- `packageRepository` (optional)

  A directory that serves as root when providing a relative path in the `packages` element of a Thales Control definition (see Defining Thales Control instances).

- `tcDescriptionRepository` (optional)

  Where to find Thales Control descriptions. The directory must contain a collection of folders, each of which must contain a `tc.xml` file (see Defining Thales Control instances for the specifications of such files). If the Thales Control defined in a folder in meant to be installed, this folder should also contain a `default.py` file and a `definition.py` file describing the instance components. Refer to the documentation of the Thales Control installer to learn how to write such files.

- `tcInstallationRepository` (optional)

  Where to install Thales Control. Each installation is placed in its separate subfolder.

- `workflowRepository` (required)

  Where to find available workflows. The directory must contain a collection of folders, each of which must contain a `workflow.xml` file describing a sequence of services (see Defining workflows).

- `dataRepository` (required)

  Where to find input data for the tests, and to write ouput.

- `extensionRepository` (multiple)

  Where to find extensions. The directories must contain a collection of TCtest extension JAR files.

- `report` (optional, multiple)

  How and where to write reports. A `report` element has 2 attributes, `writer` and `output`. The `writer` value should be an extension name. The specified extension will be executed by the launcher at reporting time. The `output` value should be a path to a file where to write the report. The path can be relative to the data repository or absolute.

Note
All paths in this file can be either relative to the launcher's working directory, or absolute, except where stated otherwise.

Here is an example of what the file might look like:

```
<settings>
        <installerRepository>C:\TCtest\installers</installerRepository>
        <packageRepository>C:\TCtest\installers</packageRepository>
        <tcDescriptionRepository>C:\TCtest\definitions</tcDescriptionRepository>
        <tcInstallationRepository>instances</tcInstallationRepository>
        <workflowRepository>C:\TCtest\tests</workflowRepository>
        <dataRepository>data</dataRepository>
        <extensionRepository>C:\TCtest\services</extensionRepository>
        <extensionRepository>C:\TCtest\reporting</extensionRepository>
        <report writer="reporting-rst" output"report.rst"/>
        <report writer="reporting-tusar" output"D:\TCtest\reports\report.xml"/>
</settings>
```

2.2 Defining a Test Tree

A test tree is, with the exception of services, defined in XML files. For services, see Extending.

2.2.1 Defining Workflows

A workflow file has to be named `workflow.xml` and placed in a subfolder of a workflow repository. The subfolder's name is considered to be the name of the workflow that is used to refer to it in a test suite file.

Root element `workflow`:

- `parameter` (multiple)

  Name-values pairs that will be injected in services. The name and the value are specified through the `name` and `value` attributes.

- `service` (multiple)

  The services that the workflow should run, in order.

Element `service`:

- `name` (required)

  The name of the service. Either the name of the jar file, or if the latter is something like tctest-*-<version>.jar, simply the part matched by the *. This name system offers the possibility for workflows to be indenpendent of the version of services.

- `parameter` (multiple)

  Name-values pairs that will be injected in services. The name and the value are specified through the `name` and `value` attributes.

Example:

```
<workflow>

        <parameter name ="JobName" value="myJob"/>

        <service>
                <name>hudson-createjob</name>
                <parameter name="ConfigFile" value="config.xml"/>
        </service>

        <service>
                <name>hudson-deletejob</name>
        </service>

</workflow>
```

2.2.2 Defining Thales Control Instances

A Thales Control description file has to be named `tc.xml` and placed in a subfolder of a Thales Control description repository. The subfolder's name is considered to be the name of the instance that is used to refer to it in a test suite file.

Note

`tc.xml` files are refered to as *description* files rather than *definition* files to avoid confusion with the `definition.py` required by the Thales Control installer.

Here is the XML specification for Thales Control description file:

Root element `thalesControl`:

- `installer` (optional)

Path to a Thales Control installation script, or to a folder containing an `install.py` file. The path can be absolute, or relative to the installer repository. If not specified, the launcher will assume that no installation is required. In this case, TCtest will have to way of learning the URLs of the different components of the instance automatically, so they should be provided manually as parameters.

- `packages` (optional)

  Path to a Thales Control package directory. The path can be absolute, or relative to the package repository. If `installer` is specified, then `packages` is required, otherwise it is ignored.

- `home` (optional)

  Path to a directory where to install this Thales Control. The path can be absolute, or relative to the Thales Control installation repository. The default value is the name of the Thales Control.

- `parameter` (multiple)

  Name-values pairs that will be injected in services. The name and the value are specified through the `name` and `value` attributes.

Examples:

```
<thalesControl>
        <installer>C:\ThalesControl-4.4.6\install.py</installer>
        <packages>C:\packages4ThalesControl-4.4.6</packages>
</thalesControl>
```

```
<thalesControl>
        <installer>ThalesControl-4.4.6</installer>
        <packages>packages4ThalesControl-4.4.6</packages>
        <home>tc1</home>
</thalesControl>
```

```
<thalesControl>
        <home>C:\TC1\</home>
        <parameter name="HudsonURL" value="http://TC1:8080/hudson"/>
        <parameter name="SonarURL" value="http://TC1:8080/sonar"/>
</thalesControl>
```

2.2.3 Defining Test Suites

A test suite file can be stored anyway, as its path is passed as command-line argument to the TCtest launcher.

Here is the test suite XML specification:

Root element `testSuite`:

- `name` (`string`, optional)

  The name of the test suite. The name is used for logging purposes and reporting. The default is "Unnamed"

- `thalesControl` (`string`, optional)

  The name of a Thales Control (see Defining Thales Control Instances). Test cases inherit this Thales Control if they do not specify one them-selves.

- `test` (`test`, multiple)

  Test case definition XML elements. See below.

Element `test`:

- `name` (optional)

  The name of the test. The name is used for logging purposes and reporting. The default is "Unnamed".

- `workflow` (required)

  This must match the name of a directory in the test repository.

- `thalesControl` (optional)

  The name of the Thales Control against which to run the test. If the element is absent, the test case inherits the Thales Control from its test suite. To specify that the test needs no Thales Control, the value should be set to `none`.

- `workdir` (optional)

  A directory where the test should read and write files. The path can be relative to the data repository or absolute. The default value is the name of the test.

- `parameter` (multiple)

  Name-values pairs that will be injected in services. The name and the value are specified through the `name` and `value` attributes.

Example:

```
<testSuite>

        <test>
                <name>test42</name>
                <workflow>workflow2</workflow>
                <thalesControl>thalesControl38</thalesControl>
                <workdir>D:\tctest\workflow2\test42</workdir>
                <parameter name="Url" value="http://url.thales"/>
                <parameter name="Path" value="C:\path"/>
        </test>

        <test>
                <workflow>workflow4</workflow>
                <parameter name="Job" value="myjob"/>
        </test>

</testSuite>
```

2.3 Setting Up Extensions

Some extensions require initialization parameters. These parameters must be provided in a XML file that has the same name as the JAR, but with the `.xml` extension. This file must have an `extension` root element with one or more `parameter` child elements. The specifications of that element is as usual (`name` and `value` attributes). See the extension documentation to learn the required parameters.

3 Running

Use the TCtest launcher to run a test suite. Provide the global configuration file and the test suite file as command-line arguments.

```
java -jar tctest-launcher-5.0.jar settings.xml testSuite.xml
```

User-provided paths are checked out, Thales Control instances are installed and launched and tests are run against them.

Tests are run in parallel as much as possible. The limitation is that 2 tests cannot run against the same Thales Control at the same time.

4 Extending

New features can be added to TCTest by writing new extensions. An extension is a Java class library packaged in a JAR file.

4.1 How the launcher uses extensions

Extensions can be divided in two kinds. The *services* and the *report writers*. Services are called by workflows and are where the actual testing takes place. Report writers are called after all the tests have been run. They are meant to compile test results and persist them into a file, a database or anything else. However, any extension can be called (see Global Settings and the `report` element), so possibilities are endless.

4.2 Writing the features

There are two constraints for the class library to be a compatible extension. Firstly, it has to include a class implementing the `ExtensionTask` interface from the API. Secondly, this same class has to expose a public constructor with specific annotations.

4.2.1 Implementing `ExtensionTask`

This class addressing this constraint is considered to be the extension main class.

The `ExtensionTask` interface exposes only one method:

```
Result run(Context context)
```

The `Context` interface provides named values that the method can consume, and a working directory that the extension should use to read and write files. When writing a extension, a developper should use as much values as needed, and given them a meaningful name. The burden of filling in the values is delagated to the users of the extension, not the developper. The `Context` interface exposes the following members:

- `T getVar(String name)`

  Returns the value matching the given name.

- `File getWorkingDirectory()`

  Path to the working directory.

The return value of `run` is of type `Result`. Use the `ResultFactory` to provide `Result` instances. There are 3 kinds of results : success, failure, and error. The first two indicate an expected end of execution, while the third indicates an unexpected end of execution.

- Success

  The extension was run successfully. If the extension is meant to test something, then a success indicates a passed test.

  ```
  return ResultFactory.success("Meaningful message");
  ```

- Failure

  The extension was run successfully, but the test was not passed. Notice that this result kind is only useful for extension that test something.

  ```
  return ResultFactory.failure("Meaningful message");
  ```

- Error

  An exception thrown during the execution, or the extension cannot run due to invalid input.

  ```
  try { }
  catch (Exception e) {
          return ResultFactory.error(e, "Meaningful message");
  }
  ```

Note

The `ResultFactory` also exposes a method `fromBoolean` that lets you create a success or a failure depending on the value of a `boolean`.

Note

In the case of report writers, the `context` parameters contains only two values named `Content` and `Writer`. `Content` returns a `Result` instance that should be the data source from which the report is compiled. `Writer` returns a `Writer` instance in which output should be written.

4.2.2 Exposing a compatible constructor

Usually an extension will not have to expose a constructor with parameters, as everything is provided by the `context` parameter of the `run` method. However, some extensions sometimes require that expensive operations are carried out only once for every subsequent run invocation (loading a file, connecting to a database...). If this is the case of your extension, your main class has to expose a public constructor annotated with the `Inject` annotation from Google Guice. It can take as many parameters as necessary, but they should be of type `String`. Each parameter must be annotated with the `Named` annotation, and the name given matches the name given in the extension initialization configuration file (see Setting Up Extensions). For instance, a service with a main class `MyService` exposing this contructor:

```
@Inject
public MyService(@Named("param1") String param1,
                 @Named("param2") String param2)
```

can be initialized by a configuration file in this manner:

```
<extension>
        <parameter name="param1" value="value1"/>
        <parameter name="param2" value="value2"/>
</extension>
```

Note

The parameter value for the `Named` annotation does not need to match the constructor parameter name.

Note

The required annotations are located in the `com.google.inject` and `com.google.inject.name` packages. Use the following `import` clauses:

```
import com.google.inject.Inject;
import com.google.inject.name.Named;
```

These annotations are also defined in `javax.inject`. Therefore, it should be possible to write extensions that do not depend on Google Guice. However, this has not been tested.

4.3 Packaging

In order for the service to work, it has to be packaged with all its dependencies. Also, its manifest should declare a `Main-Class` attribute so that the launcher knows which class has the `run` method. Using Maven, both these constraints can de addressed by the Maven shade plug-in.

Here is how to configure this plug-in in the POM to build a compatible service:

```
<plugin>
        <groupId>org.apache.maven.plugins</groupId>
```

```
        <artifactId>maven-shade-plugin</artifactId>
        <version>1.4</version>
        <executions>
                <execution>
                        <phase>package</phase>
                        <goals>
                                <goal>shade</goal>
                        </goals>
                        <configuration>
                                <transformers>
                                        <transformer implementation="org.apache.maven.plugins.shade.resour
                                                <mainClass>[Full main class name goes here]</mainClass>
                                        </transformer>
                                </transformers>
                        </configuration>
                </execution>
        </executions>
</plugin>
```

4.4 Deploying

Once you have successfully compiled and packaged your service, place the resulting JAR file one of your extension repositories, and create a initialization file if necessary. The launcher should pick up your extension automatically.

5 Built-in services

TCtest comes with a collection of built-in services. Description of and required parameters for these services are given below:

Diff

Generates a unified diff between two files.

- Name: `diff`
- Parameters

  `Original` Path to the original file. `Revised` Path to the revised file. `Output` Path to the output file.

Execute Command

Executes a command through the system shell.

- Name: `executecommand`
- Parameter `Command`: command to execute

Extract Database Data

Extracts the data of a database into an XML file using Apache DB DdlUtils.

- Name: `extractdbdata`
- Initialization Parameters

  `JdbcDrivers` A folder where to find JDBC drivers.

- Parameters

  `DatabaseURL` `JdbcDriver` JDBC driver class name. `User` `Password` `Database` Name of the database that will be written in the output file. `Catalog` `Schema` `OutputFile`

Extract Database Schema

Extracts the schema of a database into an XML file using Apache DB DdlUtils.

- Name: `extractdbschema`
- Initialization Parameters

  `JdbcDrivers` A folder where to find JDBC drivers.

- Parameter: same as Extract Database Data

Hudson Create Job

- Name: `hudson-createjob`
- Parameters

  `HudsonURL` URL to Hudson. `JobName` Name of the job to create. `ConfigFile` Path to a job configuration file.

Hudson Delete Job

- Name: `hudson-deletejob`
- Parameters

  `HudsonURL` URL to Hudson. `JobName` Name of the job to delete.

Hudson Launch Build

- Name: `hudson-launchbuild`
- Parameters

  `HudsonURL`

  URL to Hudson.

  `JobName`

  Name of the job to build

  `Wait`

  Should the service wait for the build to be complete? The value is converted to the boolean value `true` if and only if it is equal, ignoring case, to the string "`true`". Anything else is `false`.

Hudson Retrieve Job Configuration

- Name: `hudson-retrievejobconfig`
- Parameters

  `HudsonURL`

  URL to Hudson.

  `JobName`

  Name of the job of which to retrieve the configuration file.

  `Output`

  Path to a file in which to save the job configuration.

Hudson Validate Plug-in

Validates a Hudson plug-in by parsing its configuration against an XML schema.

- Name: `hudson-validateplugin`
- Parameters

  `PluginName`

  Name of the plug-in to validate

  `PluginVersion`

  Version of the plug-in to validate

  `Dtkit`

URL to a DTKIT web service

`ConfigFile`

Path to the job configuration file

`XsdFile`

Path to the schema against which to validate the plug-in configuration

`OutputDir`

Path to a folder in which to save the results

Hudson Validate Last Build

Validates that the last build of a job was successfull.

- Name: `hudson-validatelastbuild`
- Parameters

`HudsonURL` URL to Hudson. `JobName` The name of the job of which the build should be validated

Hudson Validate Console Output

Validates the console output of the last build of a job against regular expressions.

- Name: `hudson-validateconsoleoutput`
- Parameters

`HudsonURL` URL to Hudson. `JobName` The name of the job of which the build output should be validated. `Expect` A regular expression that the console output must match. Leave blank to ignore. `Reject` A regular expression that the console output must not match. Leave blank to ignore.

SchemaCrawler

Executes SchemaCrawler with the given command-line options. Refer to the SchemaCrawler documentation to learn the role of each parameter.

- Name: `schemacrawler`
- Initialization Parameters

`JdbcDrivers` A folder where to find JDBC drivers.

- Parameters

`Command InfoLevel DatabaseURL JdbcDriver User Password SchemaFilter TableFilter ProcedureFilter ExcludeColumns SortTables OutputFormat OutputFilePath`

6 Troubleshooting

6.1 Known issues

1. External processes do not exit properly