



Tiger User Manual

TIGER@gematik.de

Version 3.7.8 - 2025-05-15

Contents

1. Overview	1
1.1. Use cases	2
1.2. Components	2
2. Getting started	6
2.1. Requirements	6
2.2. Maven in a nutshell	6
2.3. Maven plugin details	11
2.4. Example project	17
2.5. How to contact the Tiger team	18
2.6. IntelliJ	18
3. Tiger test environment manager	20
3.1. Tiger.yaml files and how they are chosen and loaded	20
3.2. Supported server nodes and their configuration	21
3.3. Provided node templates	34
3.4. Configuring the local test suite Tiger Proxy	36
3.5. Standalone mode vs. implicit startup with test suite	40
3.6. Using Environment variables and system properties	41
4. Tiger Proxy	43
4.1. Excuse: What are proxies, reverse, forward	43
4.2. Tiger Proxy basics	43
4.3. Understanding routes	44
4.4. TLS, keys, certificates a quick tour on proxies	48
4.5. Modifications	52
4.6. Mesh set up	53
4.7. Adding notes to messages	55
4.8. Understanding RBelPath	56
4.9. Running Tiger Proxy as standalone JAR	63
4.10. Additional configuration	64
4.11. Understanding filtering	66
5. Tiger Test library	68
5.1. Tiger test lib configuration	68
5.2. Cucumber and Hooks	69
5.3. Using the Cucumber Tiger validation steps	70
5.4. Modifying RbelObjects (RbelBuilder)	72
5.5. Using the HTTP client steps	74
5.6. Exemplaric scenario Konnektorfarm EAU validation	81
5.7. Using Tiger test lib helper classes	84
5.8. Synchronizing BDD scenarios with Polarion test cases (Gematrik only)	85

5.9. JUnit test report when using Scenario Outlines	85
6. Tiger Configuration	87
6.1. Inlets	87
6.2. Key-translation	88
6.3. Thread-based configuration	88
6.4. Placeholders	88
6.5. RbelPath-style retrieval	89
6.6. Fallback values	90
6.7. Localized configuration	90
6.8. Examples	91
6.9. Pre-Defined values	92
6.10. Inline JEXL	92
6.11. Configuration Editor	93
7. Tiger User interfaces	98
7.1. Workflow UI	98
7.2. Postpone start of test scenarios	110
7.3. Standalone Tiger Proxy Log	110
7.4. Explanation of JEXL Expressions	116
8. Tiger Zion	120
8.1. Simple canned response	120
8.2. Looping (tgrFor)	121
8.3. Conditional rendering (tgrIf)	123
8.4. Backend request	124
8.5. Nested response	125
8.6. Matching path variables	125
8.7. tgrEncodeAs	127
8.8. RbelWriter content structures	128
9. Tiger Extensions	132
9.1. Tiger Konnektor Management Extension	132
9.2. Tiger On FHIR Extension	132
9.3. Tiger CATS Extension	132
9.4. Tiger Cloud Extension	132
9.5. Tiger PSSIM Extension	132
9.6. Tiger Robot Extension	132
10. Rest API	134
10.2. Models	137
11. Links to test relevant topics	140
12. Frequently asked questions	141
12.1. Maven	141
12.2. Extensions	142
12.3. Workflow UI	142

12.4. Other topics	143
12.5. Extensions	145
12.6. Workflow UI	146
12.7. Other topics	147

Chapter 1. Overview

To get a quick introduction to the core concepts and features of the Tiger test framework check out our video at

<https://youtu.be/eJJZDeuFlyI?autoplay>



Figure 1. Tiger product pitch video

Tiger is a framework for interface-driven BDD black-box-testing.

Tiger is a toolbox that supports and guides you when writing test suites. It lets you focus on writing the tests and solves typical problems that every team encounters (configuration, setting up the test environment, parametrization, result reporting, test running). How, you ask?

- Tiger does not focus on components but on the interactions between them. The Tiger Proxy captures the traffic between components.
- Tiger Proxy parses the traffic and builds a tree-structure which abstracts away the encoding (XML, JSON...) and lets you focus on the data.
- The Tiger test environment manager handles dockers, helm charts, JARs and external servers, boots the configured setup and routes the traffic, all with zero lines of Java, all in YAML only.
- A complete configuration toolkit, which combines multiple source and supports custom configuration of your testsuite as well, again with zero lines of Java.
- Common tasks (JSON-validation, message-filtering, scenario configuration, configuration of simulators...) can be performed with the Tiger test library, which can be seamlessly imported into BDD test suites.
This allows you to build mighty test suites with zero lines of java.
- If you want to write custom steps and glue code our Java-API has got you covered by supporting common tasks (crypto, serialization...) for you. So the little lines you have to write are be powerful and descriptive?!

1.1. Use cases

In our first dive we focused on what Tiger should stand for and how we could improve the situation of test teams.

Core business use cases

- Fast and easy set up of test environments
- Uncomplicated automated execution of IOP tests
- Explicit analysis of test failures
- Reuse of cases/steps from existing test suites
- (non Java test automation support is not implemented yet)

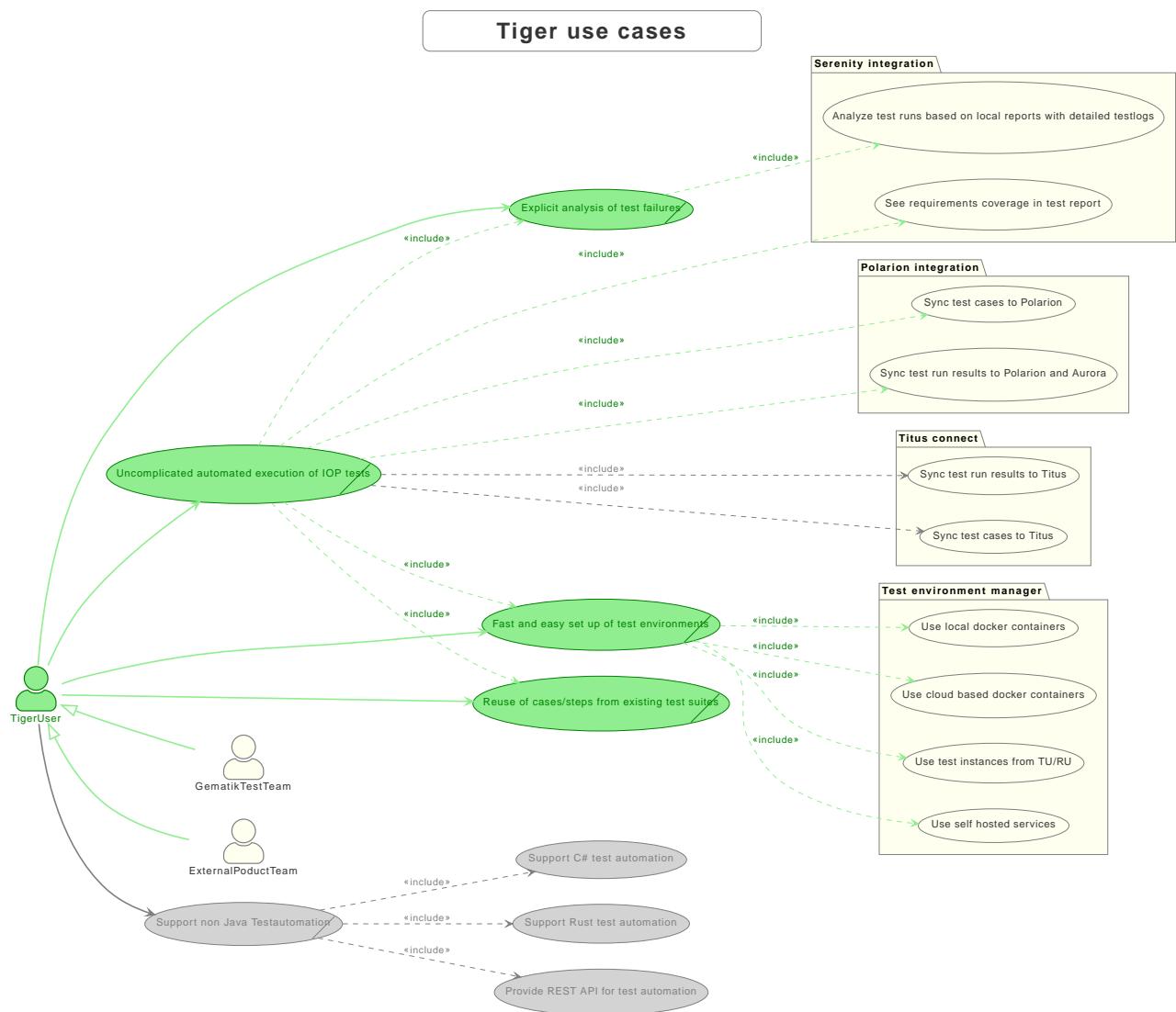


Figure 2. Tiger use cases

1.2. Components

Tiger has a clear separation in three components, each of them having a clear purpose, described in the next subsections:

- Tiger Proxy
- Tiger Testenvironment Manager
- Tiger Test library

1.2.1. Tiger Proxy

The Tiger Proxy at its core is an extended Mock server, that has the following additional core feature set:

- **Rerouting** - allows rerouting requests based on a configured lookup table
- **Modifications** - allows modifying the content of requests / responses on the fly
- **Mesh set up** - allows forwarding traffic data from one proxy to another for aggregated validations
- **TLS man in the middle** - allows tracing TLS encrypted traffic
- **RBel logging** - breaks up and parses each request / response received. This includes decryption of VAU and encrypted JWT.
Structured data like JSON, XML, JWT is displayed in a sophisticated HTML report.

1.2.2. Tiger test environment manager

The Tiger test environment manager provides methods to configure and instantiate multiple server nodes in your test environment and offers the following core feature set:

- **Instantiating test nodes** - docker containers, docker compositions, helm charts, external Jars** and
accessing server instances via external URL configurations
- **Instantiating preconfigured server nodes** - for common test scenarios like ePA, ERp, IDP, Demis
- **Automatic shutdown** - on tear down of test run, all the instantiated test nodes are ended
- **Highly configurable** - Multitude of parameters and configuration properties
- **Flexible environment management** - exporting and importing environment variables and system properties to other test nodes
- **Customizing configuration properties** - via command line system properties or environment variables

1.2.3. Tiger test library

The Tiger test library provides the following core features:

- **Validation** - BDD steps to filter requests and validate responses
- **Workflow UI** - BDD steps to support tester guidance in test workflows
- **Content assertion** - BDD steps to assert JSON / XML data structures
- **Product Integration** - Synchronisation with Polarion, Serenity BDD and screenplay pattern

1.2.4. Working together

The Testenvironment Manager instantiates all test nodes configured in the `tiger.yaml` config file. It also instantiates one local Tiger Proxy for the current test suite. This Tiger Proxy instance (and others created in the test environment if using a mesh setup) traces all requests and responses forwarded via this proxy and provides them to the test suite for further validation.

For each server node instantiated, the local Tiger Proxy adds a route so that the instantiated server node can be reached by the test suite via HTTP and the configured server hostname.

Each Tiger Proxy can be configured in a multitude of ways: as reverse or forward proxy with special routing features and modifications of content easily configurable, or in a mesh setup as proxy forwarding traffic to other Tiger Proxies...

The BDD or JUnit test suite can integrate the Tiger test library to validate messages (requests and responses) sent/received over Tiger Proxies using features such as RBelPath, VAU decryption, JSON checker and XML checker.

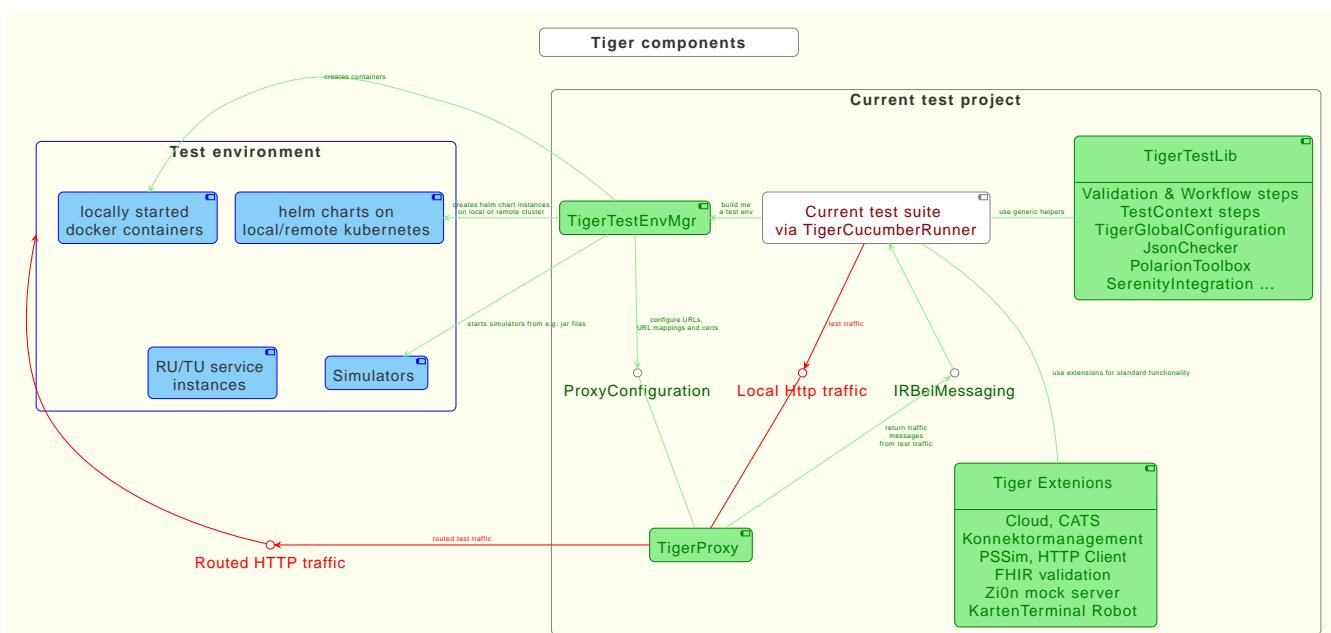


Figure 3. Tiger components

1.2.5. Tiger extensions

As Tiger evolves we have implemented quite a nice set of extensions that eases your job as tester in areas not directly fitting the core of Tiger. The currently or soon available extensions are:

- **Cloud** extension provides the docker, docker compose and helm chart server types for the Tiger test environment mgr
- **CATS** extension provides BDD steps to configure and interact with the Cats Card Terminal simulator of gematik

- **Konnektormanagement** extension provides BDD steps to administer Konnektors
- **PSSim** extension provides BDD steps to simulate a Primärsystem
- **HTTP Client** extension follows the zero code philosophy and provides BDD steps to perform http requests without having to write any line of code
- **FHIR validation** extension provides BDD steps to perform FHIR scheme based / FHIRPath based validations (planned release early spring 2023)
- **Kartenterminal Robot** extension provides BDD steps to control the card terminal robot currently constructed at gematik labs (release mid 2023)

Chapter 2. Getting started

Tiger is based on Java, Maven and Serenity BDD - so saddle the horses, check the operating system requirements and hit the road.



We do not at the moment have any plans to support gradle or other build environments.

But if you are using it in your projects feel free to contact us, and we might find a way to support your specific build environment.

If you don't have time right now to look through the whole documentation, you can directly jump to our [Example project](#) section.

2.1. Requirements

System requirements

- Open JDK >= 17
- Maven >= 3.6

When you are developing a testsuite (further), you should use an IDE (we recommend IntelliJ >= 2021.2.3 because of the IntelliJ Cucumber plugin).



On Windows you can use Git Bash or Powershell

2.2. Maven in a nutshell

In order to use Tiger with your BDD/Cucumber/Serenity based test suite you need to add a few dependencies to integrate with Tiger

- Current version of Tiger-bom in your dependencyManagement section
- Tiger test library in your dependencies section
- or the current version of Tiger test library as test-jar artefact



The second dependency is needed so that the IntelliJ Cucumber plugin detects the Steps/Glue code provided by the Tiger test library.

And to trigger the test suite's execution, you will need to add these plugins

- Tiger maven plugin
- Maven FailSafe plugin

Listing 1. Simple Tiger Maven pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
 ~ Copyright 2024 gematik GmbH
```

```

~ Licensed under the Apache License, Version 2.0 (the "License");
~ you may not use this file except in compliance with the License.
~ You may obtain a copy of the License at
~
~   http://www.apache.org/licenses/LICENSE-2.0
~
~ Unless required by applicable law or agreed to in writing, software
~ distributed under the License is distributed on an "AS IS" BASIS,
~ WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
~ See the License for the specific language governing permissions and
~ limitations under the License.
~
-->

<project xmlns="http://maven.apache.org/POM/4.0.0"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>de.gematik.test.tiger.examples</groupId>
  <artifactId>TigerTestBDD</artifactId>
  <version>1.2.0-SNAPSHOT</version>

  <properties>
    <maven.compiler.source>17</maven.compiler.source>
    <maven.compiler.target>17</maven.compiler.target>

    <version.maven.failsafe>3.3.1</version.maven.failsafe>
    <!-- please adapt Tiger version property to the most current one obtained from
-->
    <!-- maven central:
        https://mvnrepository.com/artifact/de.gematik.test/tiger-test-lib
        or from gemitik internal Nexus
        https://nexus.prod.ccs.gematik.solutions/#browse/search=keyword%3Dtiger-
test-lib
    -->
    <version.tiger>3.7.2</version.tiger>
  </properties>

  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>de.gematik.test</groupId>
        <artifactId>tiger-bom</artifactId>
        <version>${version.tiger}</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>

```

```

</dependencyManagement>

<!-- tag::dependencies[] -->
<dependencies>
    <dependency>
        <groupId>de.gematik.test</groupId>
        <artifactId>tiger-test-lib</artifactId>
    </dependency>
    <dependency>
        <groupId>io.cucumber</groupId>
        <artifactId>cucumber-junit-platform-engine</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>
<!-- end::dependencies[] -->

<build>
    <plugins>
        <!-- tag::generator-plugin[] -->
        <!-- optional plugin to dynamically create JUnit driver classes on the
fly.
        You may omit this plugin if you have written your driver classes manually.
        -->
        <plugin>
            <groupId>com.mycila</groupId>
            <artifactId>license-maven-plugin</artifactId>
            <configuration>
                <skip>true</skip>
            </configuration>
        </plugin>
        <plugin>
            <groupId>de.gematik.test</groupId>
            <artifactId>tiger-maven-plugin</artifactId>
            <version>${version.tiger}</version>
            <executions>
                <execution>
                    <configuration>
                        <!-- optional -->
                    <glues>
                        <glue>de.gematik.test.tiger.glue</glue>
                        <!-- add your packages here -->
                    </glues>
                    <!-- optional -->
                </execution>
            </executions>
        </plugin>
    </plugins>
<featuresDir>${project.basedir}/src/test/resources/features</featuresDir>
        <!-- optional -->
        <includes>
            <include>**/*.feature</include>
        </includes>
        <!-- optional -->
        <driverPackage>

```

```

        de.gematik.test.tiger.examplesbdd.drivers
    </driverPackage>
    <!-- optional -->
    <!--suppress UnresolvedMavenProperty -->
    <driverClassName>Driver${ctr}IT</driverClassName>
    <!-- optional, defaults to the templated located at
        /src/main/resources/Driver4ClassTemplate.jtmpl
        in the tiger-maven-plugin module.
    This template will create a junit4 compliant driver class.
    Use separate template file if you have spring boot apps to
test
    or need to do some more fancy set up stuff.

<templateFile>${project.basedir}/..../XXXX.jtmpl</templateFile>
    -->
    <!-- optional -->
    <skip>false</skip>
</configuration>
<phase>generate-test-sources</phase>
<id>generate-tiger-drivers</id>
<goals>
    <!-- mandatory -->
    <goal>generate-drivers</goal>
    <!-- optional. This will attach the Tiger-Agent to the VM
running the
    tests. This, in turn, enables tiger to access and store
masterSecrets of TLS
    connections. This can be used to decipher TLS-traffic in
wireshark. -->
        <goal>attach-tiger-agent</goal>
    </goals>
</execution>
<execution>
    <id>generate-tiger-report</id>
    <goals>
        <goal>
            generate-serenity-reports
        </goal>
    </goals>
    <configuration>
        <!-- optional - directory where serenity reports are
created -->
            <reportDirectory>
${project.build.directory}/site/serenity</reportDirectory>
        <!-- optional - directory with the .feature files being
executed -->
            <requirementsBaseDir>
src/test/resources/features</requirementsBaseDir>
        <!--optional - when set to true, the serenity report is
automatically open in the default browser -->
            <openSerenityReportInBrowser>

```

```

false</openSerenityReportInBrowser>
    <!-- optional - A comma separated list of report types to
be generated. -->
        <reports>html,single-page-html,json-summary</reports>
    </configuration>
</execution>
</executions>
</plugin>
<!-- end::generator-plugin[] -->

<!-- tag::failsafe-plugin[] -->
<!-- Runs the tests by calling the JUnit driver classes -->
<!-- To filter features / scenarios use the system property
    -Dcucumber.filter.tags -->
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-failsafe-plugin</artifactId>
    <version>${version.maven.failsafe}</version>
    <executions>
        <execution>
            <goals>
                <goal>integration-test</goal>
                <goal>verify</goal>
            </goals>
        </execution>
    </executions>
    <configuration>
        <includes>
            <!-- adapt to the class names of your driver classes -->
            <include>**/Driver*IT.java</include>
        </includes>
    </configuration>
</plugin>
<!-- end::failsafe-plugin[] -->
</plugins>
</build>
</project>

```

For a successful startup you also need a minimum Tiger test environment configuration yaml file in your project root:

Listing 2. Minimum Test environment configuration

```

# minimum viable test environment specification
# default local Tiger Proxy
tigerProxy:
# no server nodes
servers: {}

```

and finally a minimal feature file under src/test/resources/features:

Listing 3. Minimum Cucumber feature file

```
Feature: Test Tiger BDD

Scenario: Dummy Test
  Given TGR set global variable "key01" to "value01"
  When TGR assert variable "key01" matches "v.*\d\d"
```

With these three files in place you can run the simple dummy test scenario defined in the feature file by issuing

```
mvn verify
```

2.3. Maven plugin details

This section is for the ones that love to know all the details.

If you are happy that everything works and don't bother to understand all the bits / properties and settings just skip this section and head over to the [Example project](#) section.

2.3.1. Tiger maven plugin

This plugin allows to dynamically generate the JUnit driver classes that are then used in the Surefire plugin to start the test runs.

And replaces the serenity maven plugin to generate Serenity BDD test reports.

Generate Drivers goal



You may decide to manually write your own JUnit driver classes in which case you can omit this plugin.

To activate this feature in your maven project add the following plugin block to your <build><plugins> section:

```
<!-- optional plugin to dynamically create JUnit driver classes on the fly.
You may omit this plugin if you have written your driver classes manually.
-->
<plugin>
  <groupId>com.mycila</groupId>
  <artifactId>license-maven-plugin</artifactId>
  <configuration>
    <skip>true</skip>
  </configuration>
</plugin>
<plugin>
  <groupId>de.gematik.test</groupId>
  <artifactId>tiger-maven-plugin</artifactId>
```

```

<version>${version.tiger}</version>
<executions>
    <execution>
        <configuration>
            <!-- optional -->
            <glues>
                <glue>de.gematik.test.tiger.glue</glue>
                <!-- add your packages here -->
            </glues>
            <!-- optional -->
        </configuration>
    </execution>
<featuresDir>${project.basedir}/src/test/resources/features</featuresDir>
    <!-- optional -->
    <includes>
        <include>**/*.feature</include>
    </includes>
    <!-- optional -->
    <driverPackage>
        de.gematik.test.tiger.examplesbdd.drivers
    </driverPackage>
    <!-- optional -->
    <!--suppress UnresolvedMavenProperty -->
    <driverClassName>Driver${ctr}IT</driverClassName>
    <!-- optional, defaults to the templated located at
    /src/main/resources/Driver4ClassTemplate.jtmpl
    in the tiger-maven-plugin module.
    This template will create a junit4 compliant driver class.
    Use separate template file if you have spring boot apps to
test
        or need to do some more fancy set up stuff.

<templateFile>${project.basedir}/..../XXXX.jtmpl</templateFile>
    -->
    <!-- optional -->
    <skip>false</skip>
</configuration>
<phase>generate-test-sources</phase>
<id>generate-tiger-drivers</id>
<goals>
    <!-- mandatory -->
    <goal>generate-drivers</goal>
    <!-- optional. This will attach the Tiger-Agent to the VM
running the
masterSecrets of TLS
wireshark. -->
        <goal>attach-tiger-agent</goal>
    </goals>
</execution>
<execution>

```

```

<id>generate-tiger-report</id>
<goals>
    <goal>
        generate-serenity-reports
    </goal>
</goals>
<configuration>
    <!-- optional - directory where serenity reports are
created -->
    <reportDirectory>
${project.build.directory}/site/serenity</reportDirectory>
    <!-- optional - directory with the .feature files being
executed -->
    <requirementsBaseDir>
src/test/resources/features</requirementsBaseDir>
    <!--optional - when set to true, the serenity report is
automatically open in the default browser -->
    <openSerenityReportInBrowser>
false</openSerenityReportInBrowser>
    <!-- optional - A comma separated list of report types to
be generated. -->
    <reports>html,single-page-html,json-summary</reports>
</configuration>
</execution>
</executions>
</plugin>

```

Mandatory configuration properties

- **List[glue] glues** (mandatory)
list of packages to be included as glue or hooks code

Optional configuration properties or properties with default values

- List[include] includes (mandatory)
list of include patterns for feature files in Ant format (directory/**.feature)
- String featuresDir (default: local working directory)
root folder from where to apply includes and excludes
- List[exclude] excludes (default: empty)
list of exclusion patterns for feature files in Ant format (directory/**.feature)
- boolean skip (default: false)
flag whether to skip the execution of this plugin
- String driverPackage (default: de.gematik.test.tiger.serenity.drivers)
package of the to be generated driver class
- String driverClassName (default: Driver\${ctr})
Name of the to be generated driver class.



The ctr token \${ctr} is mandatory!

For more details see section below

- String templateFile (default: null which means that the plugin will use the built-in template file)
Optional path to a custom template file to be used for generating the driver Java source code file.
 - The plugin currently supports the following list of tokens:
 - \${ctr}
counter value that is unique and incremented for each feature file.
 - \${package}
will be replaced with package declaration code line of the driver class.
Either empty or of the pattern "package xxx.yyy.zzz;" where xxx.yyy.zzz is replaced with the configured driverPackage configuration property.
 - \${driverClassName}
name of the driver class (with the ctr token already being replaced with the incrementing counter value).
 - \${feature}
path to the feature file(s).
 - \${gluesCsv}
comma separated list of glue/hook packages without curly braces.

Manually creating driver classes

For each feature (or use wildcards / directories for single driver class) you can implement a driver class based on the example code below.

```
package de.gematik.test.tiger.integration.YOURPROJECT;

import static io.cucumber.junit.platform.engine.Constants.FILTER_TAGS_PROPERTY_NAME;
import static io.cucumber.junit.platform.engine.Constants.GLUE_PROPERTY_NAME;
import static io.cucumber.junit.platform.engine.Constants.PLUGIN_PROPERTY_NAME;

import org.junit.platform.suite.api.ConfigurationParameter;
import org.junit.platform.suite.api.IncludeEngines;
import org.junit.platform.suite.api.SelectClasspathResource;
import org.junit.platform.suite.api.Suite;

@Suite
@IncludeEngines("cucumber")
@SelectClasspathResource("features/YOURFEATURE.feature")
@ConfigurationParameter(key = FILTER_TAGS_PROPERTY_NAME, value = "not @Ignore")
@ConfigurationParameter(key = GLUE_PROPERTY_NAME, value =
"de.gematik.test.tiger.glue,ANY ADDITIONAL PACKAGES containing GLUE or HOOKS code")
@ConfigurationParameter(
    key = PLUGIN_PROPERTY_NAME,
    value = "io.cucumber.core.plugin.TigerSerenityReporterPlugin,json:target/cucumber-
parallel/1.json")
```

```
public class DriverIT {}
```

Generate Reports goal

The second execution block in the example XML section above will trigger the report creation.

With the parameter `reports` you can configure which reports get generated.

The following reports types are available

- `html` - A fancy detailed overall report (`index.html`)
- `single-page-html` - A simple HTML single page report for emailing (`serenity-summary.html`)
- `json-summary` - A summary report in json format, useful for displaying in CI tools (e.g.: Jenkins)

By setting the parameter `openSerenityReportInBrowser` to true, the `html` report will automatically open in the default browser.

Start Tiger test environment in standalone mode

Adding the plugin as shown below will allow you to start a test environment in standalone mode by starting mvn as follows: `mvn tiger:setup-testenv`.

Please be aware that this is a blocking call, you may specify a timeout configuration property `autoShutdownAfterSeconds` with timeout in seconds.

To prematurely stop the process either press Ctrl+C in your console or kill it with operating system specific kill commands / tools.

In order to customize the tiger yaml to be used either set the environment variable `TIGER_TESTENV_CFGFILE` or set the system property `tiger.testenv.cfgfile`.

```
<plugin>
  <groupId>de.gematik.test</groupId>
  <artifactId>tiger-maven-plugin</artifactId>
  <version>${version.tiger}</version>
</plugin>
```

2.3.2. FailSafe plugin

The failsafe plugin will trigger the test run.

It is important to activate the `testFailureIgnore` property, to ensure that even if the test fails, the serenity report is created.

To filter the scenarios/features to be run you may pass in the Java system property `cucumber.filter.tags`.

You can do so either within the `<systemPropertyVariables>` tag or via the command line using `-Dcucumber.filter.tags`

The "not @Ignore" is the default setting for maven verify as well as for IntelliJ, therefore scenarios that should be ignored are to be tagged with `@Ignore`.

If the user uses the cucumber option "`-Dcucumber.options`" to set own tags then the default setting of "not @Ignore" is overridden.

The same counts for own tag settings in the IntelliJ run configuration.

For more details about how to use filter tags see <https://cucumber.io/docs/cucumber/api/#tags>

```
<!-- Runs the tests by calling the JUnit driver classes -->
<!-- To filter features / scenarios use the system property
    -Dcucumber.filter.tags -->
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-failsafe-plugin</artifactId>
    <version>${version.maven.failsafe}</version>
    <executions>
        <execution>
            <goals>
                <goal>integration-test</goal>
                <goal>verify</goal>
            </goals>
        </execution>
    </executions>
    <configuration>
        <includes>
            <!-- adapt to the class names of your driver classes -->
            <include>**/Driver*IT.java</include>
        </includes>
    </configuration>
</plugin>
```



We do not recommend the parallel test execution with Tiger at the moment.

Reason is that when using Tiger Proxies with the Tiger test library validation feature parallel execution may lead to messages from different threads / forked processes ending up in the wrong listening queue making it very complicated to make sure your validations are working as expected in different timing situations.

2.3.3. Custom Tiger Failsafe Provider - Experimental

As an alternative to the driver generation with the tiger-maven-plugin, you can configure the tiger-failsafe-provider which enables the maven-failsafe-plugin to discover cucumber tests and execute them with the tiger framework.

With provider, you do not need to configure the driver generation with the tiger-maven-plugin.

To activate the tiger failsafe provider you need to add it as a dependency to the failsafe configuration:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-failsafe-plugin</artifactId>
    <version>3.5.3</version>
```

```

<configuration>
    <forkedProcessExitTimeoutInSeconds>18000</forkedProcessExitTimeoutInSeconds>
    <!-- OPTIONAL - these properties are already set to these default values. You
only need to include them if you
want to change the default value -->
    <systemPropertyVariables>
        <tiger.features>src/test/resources/features</tiger.features>
        <tiger.filter.tags>not @Ignore</tiger.filter.tags>
        <tiger.glue>de.gematik.test.tiger.glue</tiger.glue>
        <tiger.plugin>
            io.cucumber.core.plugin.TigerSerenityReporterPlugin</tiger.plugin>
        </systemPropertyVariables>
    </configuration>
    <dependencies>
        <dependency>
            <!-- by adding the failsafe provider as a dependency, it will be used when
the failsafe plugin executes -->
            <groupId>de.gematik.test</groupId>
            <artifactId>tiger-failsafe-provider</artifactId>
            <version>3.7.7-SNAPSHOT</version>
            <!-- OPTIONAL if there is a dependency on your classpath bringing the
groovy-xml library transitively, you need to
exclude it, so that there are no conflicts in the loaded versions -->
            <exclusions>
                <exclusion>
                    <groupId>org.apache.groovy</groupId>
                    <artifactId>groovy-xml</artifactId>
                </exclusion>
            </exclusions>
        </dependency>
    </dependencies>
    <executions>
        <execution>
            <goals>
                <goal>integration-test</goal>
                <goal>verify</goal>
            </goals>
        </execution>
    </executions>
</plugin>

```

2.4. Example project

In the [/doc/examples/tigerOnly](#) folder of this project you will find an example for a minimum configured maven project that

- embeds Tiger
- allows to use its Cucumber steps and
- allows to easily configure your test environment

All you need is to set up three files:

- a Maven `pom.xml` file to declare the dependencies and the plugins needed
- a `tiger.yaml` to declare your test environment (servers needed, proxy routes,...). This is currently "empty".
- a `test.feature` file containing a test scenario and dummy test steps to be performed.

File structure of TigerOnly example project

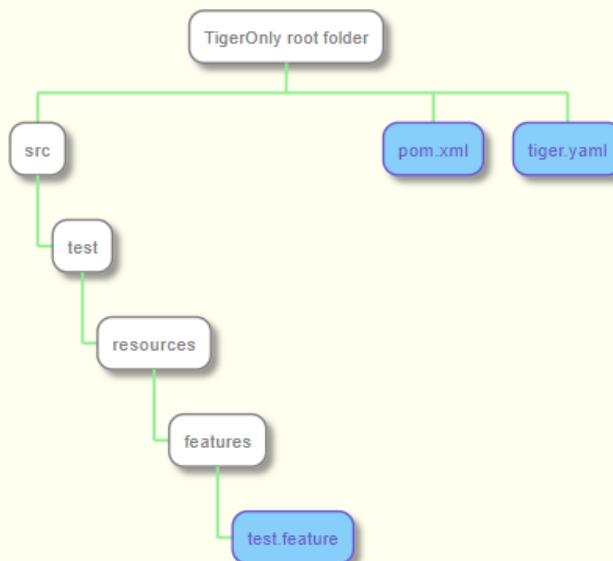


Figure 4. File structure of TigerOnly example project

2.5. How to contact the Tiger team

You can reach us via

- GitHub <https://github.com/gematik/app-Tiger>
- or email TIGER@gematik.de

2.6. IntelliJ

We recommend to use latest version of IntelliJ at least version 2021.1.

2.6.1. Run/Debug settings

To be able to successfully start scenarios/features you first need to configure the Run/Debug settings of your project:

Run/Debug settings for Java Cucumber template

- Main class: `io.cucumber.junit.TigerCucumberRunner`

- Glue:
 - de.gematik.test.tiger.glue
 - net.serenitybdd.cucumber.actors
if you are using the screenplay pattern (PREFERRED!)
 - additional packages specific to your test suite
- VM Options:
 - Java proxy system properties (see [Proxy configuration](#) below)
- Environment variables:
 - Proxy environment variables (see [Proxy configuration](#) below)

Best is to add these settings to the **Configuration Templates** for Cucumber Java.

Depending on the version of IntelliJ these settings are located either on the top icon bar or at the bottom left as link.

Else you would have to apply these settings to any new Debug/Run Configuration, like when you start a new scenario, which was never executed before.

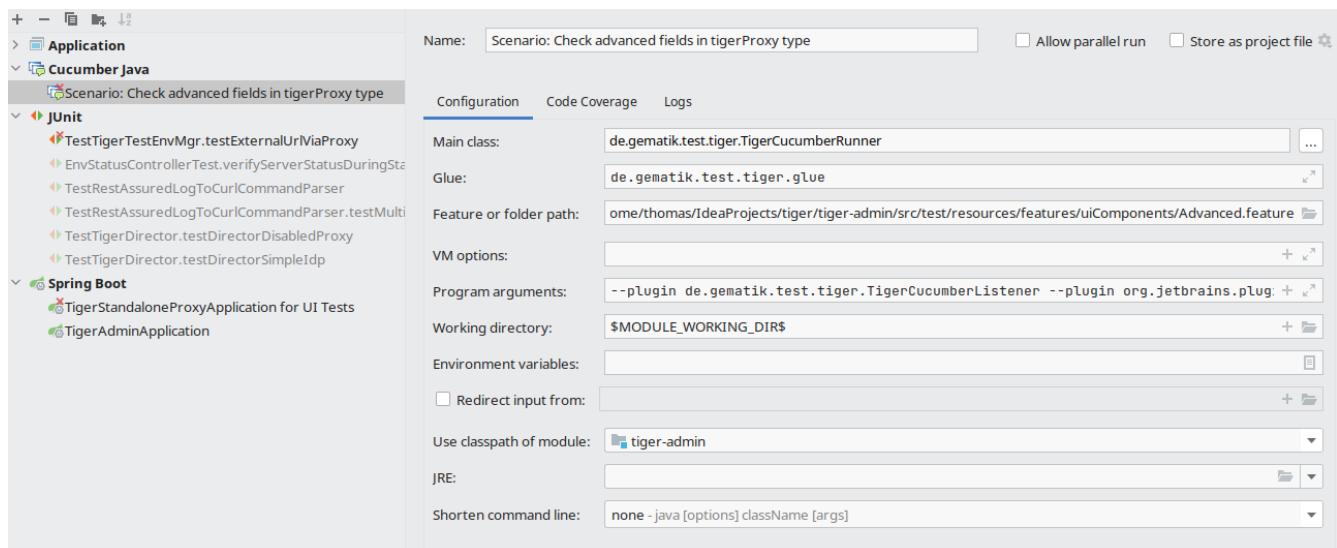


Figure 5. Run/Debug settings for IntelliJ

2.6.2. Proxy configuration

If you are located behind a proxy please make sure to set the environment variables HTTPS_PROXY and HTTP_PROXY as well as the Java system properties http.proxyHost, http.proxyPort, https.proxyHost and https.proxyPort appropriately so that the internet connections are routed properly through your **company proxy**.

Please also make sure IntelliJ has its proxy settings configured appropriately for HTTP and HTTPS so that it can download the dependencies for the IntelliJ build environment too.



BOTH settings (environment variables and system properties) are required as Maven and Java code and HTTP client libraries use both settings.

Chapter 3. Tiger test environment manager

As outlined in [the overview section](#) the test environment manager is one of the three core components of the Tiger test framework.

Its main task is to start various test server nodes configured in the `tiger.yaml` configuration file and initialize the local Tiger Proxy for the test suite.

3.1. Tiger.yaml files and how they are chosen and loaded

The test environment manager first checks if the env variable is set and tries to load the configuration file from this value.

If this file does not exist the test environment manager tries to load the configuration from `tiger.yaml`.

If none of these files exist it will fail the start-up.

To choose a different test environment configuration file you may set the environment variable `TIGER_TESTENV_CFGFILE`.

If the environment variable is not set it searches for files named `tiger.yaml` or `tiger.yml`.

If none of these files exist it will fail the start-up.

It then loads further yaml-files:

- `tiger-${hostname}.yaml` and `tiger-${hostname}.yml` are read and give the possibility to make computer-dependent configuration.

The hostname is the name of your own computer in the network (on Windows-machines typically the computername).

- The list given under `tiger.additionalYamls` is read.

Each list-entry has two properties:

- `filename` pointing to the file to be read.

This can be relative to the `tiger.yaml` (primary) or relative to the working-directory (secondary).

Keep in mind that placeholders can be used in the filename!

- `baseKey`, an optional attribute, which gives you the chance to prefix every property from the given file with this key (keep in mind that the `tiger.yaml` has a `baseKey` of `tiger`)

In the start-up phase it also informs the local Tiger Proxy about the hostnames each node has configured, so that the local Tiger Proxy can create appropriate routing entries in its own configuration.

To configure your test environment you can compose the `tiger.yaml` file manually.

The nodes configured in the yaml file will be started asynchronously unless the `dependsUpon` property is set.

3.2. Supported server nodes and their configuration

The Tiger testenvironment manager currently supports the following list of server nodes.

- **Docker container** is a node based on instantiating a specific docker image that is either locally available or downloaded from a remote docker repo configured in the source property.
- **Docker compose** is a node that you can use to start a group of services defined in one to several compose yaml files configured in the source list.
- **Helm charts** is a node that installs/updates a given helm chart on a local or remote kubernetes cluster (configuring a local context for remote clusters has to be done outside of Tiger)
- **External jar** is a node that is started by running `java -jar XXXX.jar` after downloading a Jar archive from the configured source URL.
- **External URL** is a symbolic node that is actually maintained outside the realm of the test environment manager.

The main purpose is to allow the test suite to access this external server via a constant URL, regardless of what the actual access URL of the server is.

So if you change the location of the external server has no adaptations effect on the test suite.

- **Tiger Proxy** is a specialized external Jar node that allows you to instantiate standalone Tiger Proxy nodes in your test environment in several locations to track, log and validate traffic between any two nodes.
For this to work, you must either be able to force a proxy on the nodes or [use a reverse proxy set up scenario](#).
- **Helm Chart** is a node that will be added to tiger via a plugin mechanism that starts a helm chart within a kubernetes environment.
- **httpbin** is a node mainly useful for testing purposes.
It starts a [httpbin](#) server which provides mock endpoints.

3.2.1. YAML configuration files in a nutshell

Before you start writing your own `tiger.yaml` configuration files, make sure you have worked with yaml files before and know its syntax and structure.

If unsure take a [20 minutes primer](#), although not everything in the video is relevant, it gives a good introduction to indenting properties and structures and specifying values in a yaml file.

3.2.2. General properties

The general properties apply to each node type.

Listing 4. General properties

```
serverKey_xxx:  
  # OPTIONAL hostname of this node when accessing it via the test suite  
  # (rerouted via the local test suite Tiger Proxy).  
  # Defaults to the server name (serverKey_xxx)  
  # For docker compose and helm chart this property must NOT be set!  
  hostname: string
```

```

# MANDATORY one of [tigerProxy|docker|compose|externalJar|externalUrl]
type: string
# OPTIONAL name of a template to apply.
# Default value is empty
template: string
# OPTIONAL comma separated list of keys of server nodes that must be started
# before this node is set up.
# Default value is empty
dependsUpon: csv string
# OPTIONAL duration in seconds to wait for a successful start-up of the server node
# Default value is 20
startupTimeoutSec: int
# OPTIONAL duration in milliseconds to wait between polls for the server node to
start up
# Default value is the property
"tiger.internal.externalServer.startupPollIntervalMs" which defaults to 1000
startupPollIntervalMs: int
# MANDATORY type specific property in that for some types it's a list,
# for some others it's a single URL
source:
- source entry 1
- source entry 2
# used by all node types, for external URL this property is OPTIONAL and fallback is
the source URL
healthcheckUrl: string
# OPTIONAL only declare the server healthy once the specified return code
# is given
healthcheckReturnCode: int
# type specific property for Tiger Proxy and docker container nodes
version: string
# OPTIONAL the logs of the server are also written to a file, if no logFile is
# specified a default name will be used (default is "./target/serverLogs/[key of
server in tiger.yaml].log")
logFile: ./target/serverLogs/serverKey_xxx.log

# OPTIONAL type specific list of environmental variable assignments to be used
# when starting the server node.
# Each entry has to have the format ENV_VAR_NAME=VALUE
# Has NO EFFECT on external Url nodes.
# Default value is empty array
environment:
- ENV_VAR1=VALUE1
- ENV_VAR2=VALUE2
- http://tsl --> https://download-ref.tsl.ti-dienste.de
# OPTIONAL list of routes to be added to the local Tiger Proxy of the test suite.
# Default value is empty array
urlMappings:
- https://www.orf.at --> https://eitzen.at
# OPTIONAL list of system properties that will be provided to following nodes.
# Each entry has to have the format system.property.name=VALUE
# Default value is empty array

```

```

exports:
  - systemProp1=Value1
  - systemProp2=Value2

```

Here is a little example how the server names are set and used and how there server is reachable via the Tiger Proxy.

Listing 5. Example with three external jar servers

```

servers:
  # here the server name is "identityServer" and
  # the server is reachable under "identityServer" via the Tiger Proxy
  identityServer:
    type: externalJar
    source:
      - local:../octopus-identity-service/target/octopus-identity-service-1.0-
SNAPSHOT.jar
      healthcheckUrl: http://localhost:${tiger.ports.identity}/status
    externalJarOptions:
      options:
        - -Dhttp.proxyHost=127.0.0.1
        - -Dhttp.proxyPort=${tiger.ports.proxyPort}
      arguments:
        - --server.port=${tiger.ports.identity}
        - --services.shopping=http://myShoppingServer

  # here the server name is "shoppingServer"
  # but the server is reachable under "myShoppingServer" via the Tiger Proxy because
  # hostname is set
  shoppingServer:
    hostname: myShoppingServer
    type: externalJar
    source:
      - local:../octopus-shopping-service/target/octopus-shopping-service-1.0-
SNAPSHOT.jar
      healthcheckUrl: http://localhost:${tiger.ports.shopping}/inventory/status
    externalJarOptions:
      options:
        - -Dhttp.proxyHost=127.0.0.1
        - -Dhttp.proxyPort=${tiger.ports.proxyPort}
      arguments:
        - --server.port = ${tiger.ports.shopping}
        - --services.identity=http://identityServer

testClient:
  type: externalJar
  source:
    - local:../octopus-example-client/target/octopus-example-client-1.0-SNAPSHOT.jar
  healthcheckUrl: http://localhost:${tiger.ports.client}/testdriver/status
  externalJarOptions:

```

```

options:
  - -Dhttp.proxyHost=127.0.0.1
  - -Dhttp.proxyPort=${tiger.ports.proxyPort}
arguments:
  - --server.port=${tiger.ports.client}
  # here are the examples how the servers are reachable
  - --services.shopping=http://myShoppingServer
  - --services.identity=http://identityServer

```

The general properties are followed by the type specific substructures, which configure specific aspects of each node type.

Their meaning and format are explained in the related section.

Listing 6. Type specific properties

```

# type specific sub structure for external jar, Tiger Proxy, docker and helm chart
nodes

externalJarOptions:
  # used by external jar and Tiger Proxy nodes
  workingDir: string
  # only used by external jar nodes
  options: []
  # used by external jar and Tiger Proxy nodes
  arguments: []
  # flag whether to forward log output from external jar processes to the workflow
UI
  activateWorkflowLogs : true
  # flag whether to forward log output from external jar processes to workflow UI
and console
  activateLogs: true

# type specific sub structure for Tiger Proxy nodes
tigerProxyConfiguration:
  # Here a normal Tiger Proxy configuration can be used.
  # This is explained in more depth down below
  adminPort: int
  proxiedServer: string
  proxiedServerProtocol: [HTTP|HTTPS]
  proxyRoutes:
    # defines a forward-proxy-route from this server
    - from: http://foobar
      # to this server
      to: https://cryptic.backend/server/with/path

# type specific sub structure for docker container and compose nodes
dockerOptions:
  # all properties below only used by docker container nodes
  proxied: boolean
  oneShot: boolean
  entryPoint: string

```

```

# type specific sub structure for helm charts
:
    # context to install the helm chart to
    :
        # name for the helm chart
        :
            # working directory for local helm and kubectl calls
            :
                # name space to install the helm chart to
                :
                    # flag whether to show more detailed infos about
                    # the helm chart installation in the console
                    :
                        # list of regex names for pods to be running to signal
                        # successful startup of helm chart */
                        :
                            # list of key value pairs to be used by the helm chart
                            :
                                # comma separated list of port forwardings
                                # Entries can be either "podNameRegex:xxxx", which is shorthand for
                                # "podNameRegex:xxxx:xxxx" or
                                # "podNameRegex:xxxx:yyyy" where xxxx is the local port
                                # and yyyy is the port in the pod
                                :
                                    # list of regex for pod names logs should be shown
                                    :

```

The configuration of the Tiger Proxy is explained in detail in the section [Configuring the local test suite Tiger Proxy](#)

3.2.3. Configuring PKI identities in Tiger Proxy's tls section

PKI identities can be supplied in a number of ways (JKS, BKS, PKCS1, PKCS8).

In every place a string can be given.

It could be one of

- "my/file/name.p12;p12password"
- "p12password;my/file/name.p12"
- "cert.pem;key.pkcs8"
- "rsaCert.pem;rsaKey.pkcs1"
- "key/store.jks;key"
- "key/store.jks;key1;key2"
- "key/store.jks;jks;key"

Not supported pathname strings:

- "D:\\myproject\\key\\store.jks;key"

Supported pathname string on all platforms:

- "myproject/key/store.jks;key"

Please notice, that double backslashes ("\\") are not supported as file separators, since they are not accepted on all platforms.

Invalid pathname strings will also produce an exception.

Each part can be one of:

- filename
- password
- store-type (accepted are P12, PKCS12, JKS, BKS, PKCS1 and PKCS8)

If you want, you can also describe the components in a map:

```
tls.forwardMutualTlsIdentity:  
  filename: myIdentity.p12  
  password: "changeit"  
  storeType: P12
```

In this case both the storeType and the password are not mandatory and would be guessed (the store-type via the file extension and the password via a default-list).

PKI identity passwords

Tiger will attempt to decrypt a given P12 file with a list of common passwords, among these are:

```
"00", "123456", "gematik", "changeit"
```

Users can insert additional passwords by configuring the `tiger.yaml` as follows

```
lib:  
  additionalKeyStorePasswords: ["foo", "bar", "baz"]
```

3.2.4. Docker Container node

The docker container node allows to instantiate a local docker container from the configured image.

The exposed port of the docker container is available as a special token in the substitution process of the exports entries (`#{PORT:xxxx}` where xxxx is the port being exposed inside the container).

To customize the docker container you may alter the entry point command line.

Additionally, Tiger will automatically add the Tiger Proxy certificate to the container's operating system list of trusted certificates.

This modification can be disabled by setting the property `dockerOptions.proxied` to false.

E.g.:

```
servers:  
  exampleDockerServer:  
    type: docker  
    dockerOptions:  
      proxied: false # default is true
```

For containers that should exit after a single command you may enable the oneShot property.

You can also copy files to the container by configuring the source and destinations paths of files or folder to be copied.

If there is no health check configured inside the docker image, Tiger will try to guess a healthcheck url by assuming the first exposed port as a get request to localhost to check for a successful startup of the docker container (e.g. <http://127.0.0.1:xxxx>).

If no port is exposed at all, the startupTimeoutSec property will determine the wait period, after which Tiger assumes the container is up and running.

If you have your local docker environment set up hosting the docker containers on a remote docker hub server, you may set the environment variable **TIGER_DOCKER_HOST** to allow the health check url determined on runtime to point to the remote host instead of localhost.



To use this server type you must include the tiger-cloud-extension dependency!

Listing 7. Docker container configuration

```
dockerContainer_001:  
  hostname: myDockerContainer  
  type: docker  
  dependsUpon: csv string  
  startupTimeoutSec: int  
  
  # MANDATORY URL from where to download the docker image.  
  source:  
    - dockerhubrepo.somewhere.org/repo/project/docker.image  
  # OPTIONAL version of the docker image to download.  
  version: 0.1.2  
  # OPTIONAL the logs of the docker container are also written to a file, if no  
  logFile is  
  # specified a default name will be used  
  logFile: ./target/serverLogs/dockerContainer_001.log  
  
  dockerOptions:  
    # OPTIONAL Flag whether the container shall be modified by  
    # o adding the Tiger Proxy certificate to the container operating system.  
    # o adding docker.host.internal to the container's /etc/hosts file.  
    # Default value is true.
```

```

proxied: true
# OPTIONAL Flag whether the container is a one shot container or not.
# One shot meaning it will execute a command and then stop.
# Default value is false.
oneShot: false
# OPTIONAL The entry point command line to be used to start up this container
# overwriting any configured entry point in the docker image.
# Default value is empty meaning to use the configured entry point command line.
entryPoint: chmod a+x /startup.sh && /startup.sh
# OPTIONAL list of files to be copied to the container
copyFiles:
  # path to the file or the folder to copy inside the container
  - sourcePath: ./example/path/file_to_copy.txt
    # path inside the container where the file should be copied to
    destinationPath: /path/in/container/file_to_copy.txt
    # OPTIONAL the file mode of the copied file as octal representation (see
    https://en.wikipedia.org/wiki/File-system_permissions#numericNotation
    fileMode: 0633
    # a complete folder can also be copied instead of a single file
    - sourcePath: ./example/copy_folder
      destinationPath: /path/in/container/copy_folder

# The following properties are explained in the General properties section above
environment: []
urlMappings: []
exports: []

```

3.2.5. Docker Compose node

The docker compose node is a very tricky type of node because we use testcontainer library, which is not exactly up to date in terms of docker compose support.

So many of the yaml compose files will need to be modified to work with the testcontainer library.

For now, we support the ePA2 FD module and the DEMIS Meldeportal.

If you want to use your own compose files, please note that Tiger copies and processes your yml files to the target/tiger-testenv-mgr/\${serverId} folder, replacing all variable/property expressions (for details check [this chapter](#)).

The processing/copying flattens the file hierarchy, thus you must not depend on any additional file resources in your docker compose files.

Each copied compose file will have a random UUID appended to its filename.



To use this server type you must include the tiger-cloud-extension dependency!

Listing 8. Docker compose configuration

```

type: compose
dependsUpon: csv string

```

```

startupTimeoutSec: int
# OPTIONAL the logs of the docker compose are also written to a file, if no logFile
is
# specified a default name will be used
logFile: ./target/serverLogs/dockerCompose.log

# MANDATORY list of yaml files to use to start up the services.
# The entries can either be file paths or if starts with
# classpath:.... a reference to a yaml file contained in the class path
# (it could also be located inside a jar that is in the class path)
source:
- classpath:/de/gematik/test/tiger/testenvmgr/epa/titus-epa2.yml
- classpath:/de/gematik/test/tiger/testenvmgr/epa/titus-epa2-local.yml

```

Listing 9. Demis docker compose example

```

demis_001:
type: compose
source:
- classpath:/de/gematik/test/tiger/testenvmgr/demis/demis_localhost.yml
startupTimeoutSec: 180

```

3.2.6. External Jar node

The External Jar node is along with the Docker container node the most important/used node for test environments.

Any Jar archive executable which can be started with the `java -jar` command can be configured as an external Jar node.

The options list are arguments added immediately after the java executable, while the arguments list is appended after the `-jar` argument.

The working directory is the place where the jar file is downloaded to and executed from.
So if your jar archive expects some configuration files make sure to choose the folder appropriately.

If using the `local:` prefix you can also use wildcards to find any matching jar-files.
Tiger will use the following order to try to find a matching file:

- In the working directory a file with the filename contained in the source
- From the working directory a file with a relative path equal to the source
- In the working directory a file with a filename matching the source (eg. `app-*.jar`)
- From the working directory a file with a relative path equal and matching the filename of the source (eg. `../target/app-*.jar`)

```
java ${options} -jar externalJar.jar ${arguments}
```

Listing 10. External jar configuration

```
externalJar_001:  
    hostname: mySpecialJar  
    type: externalJar  
    dependsUpon: csv string  
    startupTimeoutSec: int  
  
    # MANDATORY SINGLE ENTRY URL from where to download the Jar archive.  
    # If the entry starts with "local:" followed by a file path the jar archive  
    # is expected to be available at that location and no download is performed.  
    # Only one entry is expected for this node type. Additional entries are silently  
    ignored.  
    source:  
        - http://myjars.download.org/myproject/myjar.jar  
    # MANDATORY URL to check for the successful startup of this node.  
    # A successful start is indicated by ANY answer on this URL.  
    # Any status is accepted as long as there is an answer.  
    # If set to "NONE" no check is performed and  
    # the test environment manager will wait for the startup timeout.  
    healthcheckUrl: http://127.0.0.1:8080  
    # OPTIONAL only declare the server healthy once the specified return code  
    # is given  
    healthcheckReturnCode: int  
    # OPTIONAL the logs of the externalJar are also written to a file, if no logFile is  
    # specified a default name will be used  
    logFile: ./target/serverLogs/externalJar_001.log  
  
externalJarOptions:  
    # OPTIONAL folder from where to start the external jar.  
    # The downloaded jar file will be stored and executed from here  
    # The default value is empty, which means that the operating-system-specific  
    # temporary folder will be used.  
    # hint: when the jar file is taken from a local directory and is set in source  
    # and the workingDir is set then the workingDir has to be the directory where  
    # the jar file is located  
    workingDir: /home/user/test/myspecificjar  
    # OPTIONAL Options to pass in to the java executable call.  
    options: []  
    # OPTIONAL provide additional arguments to the jar archive call.  
    # Default value is empty.  
    arguments:  
        - --testarg1  
        - -singledasharg2  
        - --paramarg3=testvalue1  
  
    # The following properties are explained in the General properties section above  
environment: []  
urlMappings: []  
exports: []
```

By default, the JVM used to start the JAR-File is taken from the `java.home` system property, thus using the same JVM with which Tiger was started.

To change the JVM used you can set the property `tiger.lib.javaHome` (e.g. by setting `-Dtiger.lib.javaHome`, by setting `TIGER_LIB_JAVAHOME` in the environment or by setting `lib.javaHome` in the `tiger.yaml`).

3.2.7. External URL node

The symbolic node type that will not start a server instance, but simply allows external services to be used via the configured hostname.

This is achieved by the test environment manager instructing the local Tiger Proxy to provide a route for the symbolic hostname to the external URL of the service.

So, in the following example, the test suite can send HTTP(S) requests to the server "http://myExternalServer" via the local Tiger Proxy, which will be rerouted to the external URL "https://www.medizin.de".

If it is ever necessary to change the external URL, the test suite does not have to be modified, only the routing configuration for the node has to be changed.

Given the nature of this type, the environment section has no effect and is not to be used.

Listing 11. External URL configuration

```
externalUrl_001:  
  hostname: myExternalServer  
  type: externalUrl  
  dependsUpon: csv string  
  startupTimeoutSec: int  
  
  # MANDATORY URL of the external server  
  source:  
    - https://www.medizin.de  
  
  # OPTIONAL URL to check for successful startup of this node.  
  # A successful start is indicated by ANY answer on this URL.  
  # Any status is accepted as long as there is an answer.  
  # If the value is not set, then no health check is carried out  
  # in the startup phase, instead the startupTimeout is waited for.  
  # After this timeout it is assumed that the server is up.  
  healthcheckUrl: https://www.medizin.de/healthyState.jsp  
  # OPTIONAL only declare the server healthy once the specified return code  
  # is given  
  healthcheckReturnCode: int  
  # OPTIONAL the logs of the externalUrl are also written to a file, if no logFile is  
  # specified a default name will be used  
  logFile: ./target/serverLogs/externalUrl_001.log  
  
  # The following properties are explained in the General properties section above  
  # IGNORE for this type as it has no effect  
  environment: []
```

```
urlMappings: []
exports: []
```

3.2.8. Helm Chart node

The helm chart node allows to start a helm chart from the configured source (local helm chart file / folder or remote helm chart).

The helm chart is started and the server is ready when all pods are up and running, if port-forward is used (if exposedPorts are set), then port-forwarding is also done and the startup is finished and the service can be used for testing.



To use this server type you must include the tiger-cloud-extension dependency!

Listing 12. Helm chart configuration

```
servers:
  testHelmChart_Nginx:
    type: helmChart
    startupTimeoutSec: 50
    # MANDATORY repository from where to download the docker image
    # if the helm chart is stored on the local file system that the
    # workingDir should be set.
    source:
      - bitnami/nginx
    # OPTIONAL version of the image
    version: 1.1.0
    helmChartOptions:
      # The kubernetes context
      context:
        # OPTIONAL if no working directory is set the default . is used.
        # if the helm chart is stored on the local file system the workingDir
        # should be set.
      workingdir:
        # OPTIONAL prints out debug messages if set to true, default is false.
        debug: true
        # OPTIONAL override the POD_NAMESPACE environment variable if set.
        # if not set, "default" will be used.
      nameSpace: buildslaves
      # MANDATORY pod name of the helm chart
      podName: test-tiger-nginx
      # OPTIONAL key-value pairs that will be used for starting the helm chart
      values:
        # OPTIONAL should contain a list of pods for the health check, regex can be
        # used.
      healthcheckPods:
        - test-tiger-nginx-.*
      # OPTIONAL contains a list of regex to identify the pods whose logs
      # should be forwarded to the console and Tiger Workflow UI.
      logPods:
        - test-tiger-nginx.*
```

```

# OPTIONAL contains a list that will be used for the port forwarding,
# if empty no port forwarding is done. The syntax is:
#
<POD_NAME_OR_REGEX>,<LOCAL_PORT>:<FORWARDING_PORT>[,<LOCAL_PORT>:<FORWARDING_PORT>]*

exposedPorts:
  - test-tiger-nginx.*:8080:80

```

3.2.9. Tiger Proxy node

The most complex and versatile node type.

The Tiger Proxy will be started as an embedded spring boot application.

This way the start-up time can be minimized, and it is always guaranteed to start the current version.

Listing 13. Tiger Proxy configuration

```

tigerProxy_001:
  hostname: myTigerProxy
  type: tigerProxy
  dependsUpon: csv string
  startupTimeoutSec: int

tigerProxyConfiguration:
  # OPTIONAL port of the web user interface and the proxy management
  # (e.g. rbel-message forwarding)
  # Default value is empty, which means a random port will be used.
  # The chosen port is stored with the key tiger.internal.localproxy.admin.port in
  # the TigerGlobalConfiguration
  adminPort: 8080
  # OPTIONAL server name of the node this proxy shall be used as reverse proxy for.
  # If set the routes will be configured appropriately.
  # Default value is empty.
  proxiedServer: externalJar_001
  # OPTIONAL port of the proxy, where the proxy expects to receive proxy requests
  # Default value is empty, which means a random port will be used.
  proxyPort: 3128
  # OPTIONAL protocol the proxy is expecting requests in. One of [http|https]
  # Default value is http
  proxiedServerProtocol: http
  # configures the proxy itself. For more details
  # please check the chapter about the local test suite Tiger Proxy below
  ...
  proxyRoutes:
    - from: http://foobar
      # defines a forward-proxy-route from this server...
      to: https://cryptic.backend/server/with/path
      # to this server
  ...
  # The following properties are explained in the General properties section above

```

```
environment: []
urlMappings: []
exports: []
```

The configuration of the Tiger Proxy is explained in detail in the section [Configuring the local test suite Tiger Proxy](#)

3.2.10. httpbin node

The httpbin simply starts a [httpbin](#) server.

This provides several endpoints against which you can perform all kinds of http requests.

The server port on which the server starts can be configured.

Listing 14. httpbin configuration

```
httpbin:
  type: httpbin
  serverPort: ${free.port.0}
  healthcheckUrl: http://localhost:${free.port.0}/status/200
```

3.3. Provided node templates

Besides these basic nodes we also support tailored templates for nodes like IDP, ePA, ERp and DEMIS.

This should allow you to bring up project specific test environments very fast.

All currently supported templates can be found in the tiger-testenv-mgr modul in the yaml file at /src/main/resources/de/gematik/test/tiger/testenvmgr/templates.yaml

To use such a template, just use the template attribute:

```
myPersonalTestIDPInTheRU:
  template: idp-rise-ru
```

or if you want to have an environment with a local reference implementation of the ERezept Fachdienst

```
myLocalTestIDP:
  template: idp-ref
  hostname: idp

myLocalTestERp:
  template: erzpt-fd-ref
  dependsUpon: myLocalTestIDP
```

3.3.1. Local IDP reference nodes

This template provides the reference implementation of the IDP server as a local docker container. The docker image is loaded from a gematik internal docker registry server.

The system property IDP_SERVER is set to the URL of the Discovery Document end point and is available for all subsequently initiated test environment nodes.

3.3.2. External IDP RISE instance nodes

The idp-rise-ru template provides the RU instance of RISE's IDP server as an "external URL". The system properties IDP_SERVER and GEMATIK_TESTCONFIG are set to the URL of the Discovery Document end point and a config-file for the IDP test suite respectively. They are available for all subsequently initiated test environment nodes.

The idp-rise-tu template provides the TU instance accordingly.

3.3.3. Local ERp reference nodes

This template provides the reference implementation of the eRezept server as a local docker container.

The docker image is loaded from a gematik internal docker registry server.

Make sure that an IDP server node is instantiated before the ERp FD is started and that it is available under <http://idp> or adapt the environment variable configuration.

A large list of environment variables is set.

But don't worry, it is just the server that uses them.

3.3.4. Local ePA2 reference nodes

This template provides the gematik reference Aktensystem simulation as docker compose.

3.3.5. Local PSSim node

This template provides a Primärsystem simulation (as a jar), usable for ePA.

See <https://wiki.gematik.de/display/PTP/epa-ps> for more information.

3.3.6. Local KonSim node

This template provides a Konnektor simulation (as external jar).

See <https://wiki.gematik.de/display/PTP/KonSim> for more information.

3.3.7. Local ePA FdV Sim

This template provides FdV simulation, usable for ePA.

3.3.8. Local DEMIS reference nodes

This template provides the DEMIS Meldeportal as local docker compose.

3.4. Configuring the local test suite Tiger Proxy

The local Tiger Proxy for the test suite can be configured by using the following section(s) in the `tiger.yaml` file.

For more information about what the Tiger Proxy is and how it works see the chapter [Tiger Proxy basics](#)

```
# Flag whether to activate the local Tiger Proxy. The local tiger proxy field will be
# null if this property is set to false
# Default value is true
localProxyActive: true

# Specifiy additional yaml-files to read in during startup
additionalConfigurationFiles:
  -
    # the path to the file to read
    filename: specialEnvironment.yaml
    # the key to which to map the given file. "tiger" is the base-key for the
    # tiger.yaml-file
    baseKey: tiger

# the block where all the Tiger Proxy configuration properties are located
tigerProxy:
  # the port under which the server will be booted
  adminPort: 7777
  # logLevel of the proxy-server. DEBUG and TRACE will print traffic, so use with
  # care!
  proxyLogLevel: TRACE
  # section to configure whether and where the proxy should dump a tiger traffic file
  # on shutdown
  fileSaveInfo:
    # should the traffic be logged to a file?
    writeToFile: true
    # configure the file name
    filename: "foobar.tgr"
    # default false, removes any potentially existing saved tgr traffic file on
    # startup.
    clearFileOnBoot: true
    # read traffic from a file at startup
    sourceFile: "sourceFile.tgr"
    # filter messages read from file (JEXL expression)
    readFilter: "message.statusCode == '200'"

  # a list of routing entries the proxy should apply to traffic
  proxyRoutes:
    # defines a forward-proxy-route from this server...
    - from: http://foobar
      # to this server
      to: https://cryptic.backend/server/with/path
      # reverse proxy-route. http://<tiger-proxy>/blub will be forwarded
```

```

- from: "/blub"
  to: "https://another.de/server"
  # the traffic for this route will NOT be logged (default is false)
  disableRbelLogging: true
  hosts:
    - "www.google.com"
    - "www.bing.com"
    # For reverse proxy-routes this allows to differentiate requests based on the
    # host-header. This is useful when the DNS-resolvement of the client can be influenced,
    # but not the ports used.

  # a list of modifications that will be applied to every proxied request and response
  modifications:
    # a condition that needs to be fulfilled for the modification to be applied
    # (uses JEXL grammar)
    - condition: "isRequest"
      # which element should be targeted?
      targetElement: "$.header.user-agent"
      # the replacement string to be filled in.
      # This modification will replace the entire "user-agent" in all requests
      replaceWith: "modified user-agent"

    - condition: "isResponse && $.responseCode == 200"
      targetElement: "$.body"
      # The name of this modification.
      # This can be used to identify, alter or remove this modification.
      # The name is optional, but if it is used, it has to be unique.
      name: "body replacement modification"
      # This will replace the body of every 200 response completely with the given json-
      # string
      # (This ignores the existing body. For example this could be an XML-body.
      # Content-Type-headers will NOT be set accordingly).
      replaceWith: "{\"another\":{\"node\":{\"path\":\"correctValue\"}}}"
    - targetElement: "$.body"
      # The given regex will be used to target only parts of targeted element.
      regexFilter: "ErrorSeverityType:(Error)|(Warning)"
      # This modification has no condition,
      # so it will be applied to every request and every response
      replaceWith: "ErrorSeverityType:Error"

  # can be used if the target-server (to) is behind another proxy
  forwardToProxy:
    hostname: 192.168.110.10
    port: 3128
    type: HTTP
    noProxyHosts:
      - "localhost"
    # The Tiger Proxy will route google.com to google.com even if no route is set.
    # The traffic routed via this "forwardAll"-routing will be logged by default
    # (meaning it will show up in the Rbel-Logs and be forwarded to tracing-clients)
    # This can be deactivated by setting this flag to false

```

```

activateForwardAllLogging: true
# Limits the rbel-Buffer to approximately this size.
# Note: When Rbel debugging is activated the size WILL vastly exceed this limit!
rbelBufferSizeInMb: 1024
# If set to false disables traffic-analysis by Rbel.
# Deactivating will not impede proxy-forwarding nor
# the traffic-endpoints.
activateRbelParsing: true
# While parsing the Tiger Proxy can block the communication from completing.
# The end answer from the Tiger Proxy is only transmitted when parsing is completed
# (and the message pair can be seen in the log). When 'false' the parsing is done
# asynchronous.
# Default is true ONLY for the local Tiger Proxy, otherwise default is false!!
parsingShouldBlockCommunication: false
# This will share the WebUI-Resources (various CSS-files) from the Tiger Proxy
# locally, thus enabling usage when no internet connection exists
localResources: true
# When active the host-headers are rewritten even for a reverse-proxy-route
rewriteHostHeader: true
# This option can be used to deactivate TLS-Termination. Only deactivate this in
# conjunction with a directReverseProxy
activateTlsTermination: true

tls:
# Can be used to define a CA-Identity to be used with TLS. The Tiger Proxy will
# generate an identity when queried by a client that matches the configured route.
# If the client then in turn trusts the CA this solution will provide you with a
seamless
# TLS experience. It however requires access to the private-key of a trusted CA.
serverRootCa: "certificate.pem;privateKey.pem;PKCS8"
# Alternative solution: now all incoming TLS-traffic will be handled using this
identity.
# This might be easier but requires a certificate
# which is valid for the configured routes
serverIdentity: "certificateAndKeyAndChain.p12;Password"
# Defines which SSL-Suites are allowed. This will delete all default-suites and
only add the one
# defined here. This configures the server-side of the proxy. Available values can
be found here:
#
https://docs.oracle.com/javase/7/docs/technotes/guides/security/SunProviders.html
serverSslSuites:
- "TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA"
# This configures the SSL-Suites for the client-side. Available values can be
found here:
#
https://docs.oracle.com/javase/7/docs/technotes/guides/security/SunProviders.html
clientSslSuites:
- "TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA"
# Define which TLS protocols the server will allow/use. Available values can be
found here:

```

```

#
https://docs.oracle.com/javase/7/docs/technotes/guides/security/SunProviders.html
  clientSupportedGroups:
    - "brainpoolP256r1"
    - "brainpoolP384r1"
    - "prime256v1"
    - "secp384r1"
  # Define the groups to be offered in the "client hello" message. More information
  can be found here:
  # https://datatracker.ietf.org/doc/html/rfc8446#section-4.2.7
  serverTlsProtocols:
    - "TLSv1.2"

  # This identity will be used as a client-identity for mutual-TLS when forwarding
  to
  # other servers. The information string can be
  # "my/file/name.p12;p12password" or
  # "p12password;my/file/name.p12" or
  # "cert.pem;key.pkcs8" or
  # "rsaCert.pem;rsaKey.pkcs1" or
  # "key/store.jks;key" or
  # "key/store.jks;key1;key2" or
  # "key/store.jks;jks;key"
  #
  # Each part can be one of:
  # * filename
  # * password
  # * store-type (accepted are P12, PKCS12, JKS, BKS, PKCS1 and PKCS8)
  forwardMutualTlsIdentity:
  "directory/where/another/identityResides.jks;changeit;JKS"
  # domain which will be used as the server address in the TLS-certificate
  domainName: deep.url.of.server.de
  # Alternate names to be added to the TLS-certificate
  # (localhost and 127.0.0.1 are added by default)
  alternativeNames:
    - localhost
    - 63.54.54.43
    - foo.bar.server.com
  # the file into which the master-secrets are written. For this to work
  # the tiger-java-agent has to be attached to the JVM
  # (eg. -javaagent:tiger-java-agent-3.1.4.jar). This can be done by
  # executing the goal 'attach-tiger-agent' to the tiger-maven-plugin.
  masterSecretsFile: "masterSecretsFile.txt"

  # the given folders are loaded into RBel for analysis. This is only necessary to
  decrypt
  # traffic when analyzing it. It has no effect on the proxy-functions themselves.
  keyFolders:
    -
      -
        -
          -
            -
              -
                -
                  -
                    -
                      -
                        -
                          -
                            -
                              -
                                -
                                  -
                                    -
                                      -
                                        -
                                          -
                                            -
                                              -
                                                -
                                                  -
                                                    -
                                                      -
                                                        -
                                                          -
                                                            -
                                                              -
                                                                -
                                                                    -
                                                                      -
                                                                        -
                                                                          -
                                                                            -
                                                                              -
                                                                                -
                                                                                  -
                                                                                    -
                                                                

```

```

skipParsingWhenMessageLargerThanKb: 8000
# Filter out any messages (or message parts) from displaying
skipDisplayWhenMessageLargerThanKb: 512

# A list of upstream Tiger Proxies. This proxy will try to connect to all given
sources to
# gather traffic via the STOMP-protocol. If any of the given endpoints are not
accessible
# the server will not boot. (fail fast, fail early)
trafficEndpoints:
- http://another.tiger.proxy:<proxyPort>
trafficEndpointConfiguration:
# the name for the traffic Endpoint. can be any string, which will be
# displayed at /tracingpoints
name: "tigerProxy Tracing Point"

```

3.5. Standalone mode vs. implicit startup with test suite

If your test environment is very "expensive" to start or if you are developing your test suite scenarios thus starting many test runs in a short time, you might want to keep your test environment running and not shut it down after each run.

To do so, you can simply use the tiger maven plugin to start your test environment in standalone mode.

First prepare a standalone test environment configuration file (call it for example tiger-standalone.yaml) containing all the server nodes needed and with a deactivated the local Tiger Proxy section.

Now set the env var TIGER_TESTENV_CFGFILE or the Java system property tiger.testenv.cfgfile to point to this file.

And add the plugin block to your pom.xml

```

<plugin>
    <groupId>de.gematik.test</groupId>
    <artifactId>tiger-maven-plugin</artifactId>
    <version>${version.tiger}</version>
</plugin>

```

If you start the test environment manager standalone, it will keep the nodes running until you enter quit into the console or kill the process with Ctrl + C or the operating equivalent commando to the UNIX command kill \${PROCESS_ID}.

In the latter case it is not guaranteed that all processes are cleanly shut down.

Please check your process list with operating system specific tools.

```
export TIGER_TESTENV_CFGFILE=...../tiger-standalone.yaml
```

```
mvn tiger:setup-testenv
```

In case you also need cloud extension server types (docker, helmchart) make sure to add the Tiger cloud extensions as dependency to the **plugin block**.

Now before starting your test suite scenarios you need to

- disable / remove the test nodes in your default **tiger.yaml** (either by setting the property active to false or remove the server node entry completely).

If you forget to do this, two nodes will be instantiated (one from the standalone test environment manager and the second during test run from the test environment manager started via the test suite hooks).

- and add routes for each node to the local Tiger Proxy.

If you forget to do this, your test suite will not be able to access the test nodes under their configured hostname as this configuration is only known to the standalone test environment manager and NOT to the local tiger proxy started by the test suite hooks.

Best practice is to have three test environment configuration files:

- tiger-standalone.yaml to enable a persistent test environment during the development of test suite scenarios
- tiger-nonodes.yaml for the test suite that will instantiate no nodes but only configure the routes to the nodes from the standalone test environment manager
- **tiger.yaml** a complete configuration that can be used in CI or after the test suite development is completed.

The first and the latter most of the time are identical besides the root level flag localProxyActive. So you may skip the first and just use it with two different values being set.

3.6. Using Environment variables and system properties

3.6.1. Token/variable substitution

Entries in the exports list of a node will be parsed and specific tokens will be substituted:

- \${PORT:xxxx} will be replaced with the port on the docker host interface
- \${NAME} will be replaced with the hostname of the node

All exports entries of a node will be present when subsequent nodes are instantiated and can be used in the following properties:

Docker node:

- source list
- environment list

Tiger Proxy node:

- from/to route URLs

External URL node:

- source list

External Jar node:

- options list

Chapter 4. Tiger Proxy

4.1. Excuse: What are proxies, reverse, forward

There are a lot of different types of proxies.

Here we talk only about HTTP and HTTPS proxies!

4.1.1. Forward proxies

Forward proxies work like a switch-station: You send a packet to your destination, via proxy.

The proxy receives the packet, sees the address and can send that packet to wherever he sees fit.

To use a forward proxy, the sender has to be aware of it and send the packet accordingly.

This allows the creation of virtual domains, something we use extensively in tiger.

A forward proxy can always read the entire content of your communication, something we also use heavily.

Lastly a forward proxy acts as a man-in-the-middle, enabling the penetration of TLS-traffic.

We also use this, but we will explain it in more depth later.

4.1.2. Reverse proxies

Reverse proxies also receive traffic and may reroute them at their own discretion.

But unlike a forward proxy a reverse proxy is invisible to the sender.

Reverse proxies act like normal servers and are addressed as such.

They then send the received packet to its actual destination and return the answer to the original caller.

The reverse proxy can also read the complete traffic.

The eventual destination is opaque to the original caller.

This also enables path-rewriting (for example the GET <http://reverse.proxy.de/my/deep/url> might be mapped to <http://gematik.de/deep/url>, eliminating the /my)

A reverse proxy also terminates https, always.

This is less of a problem with a reverse proxy since it is technically not a man-in-the-middle attack, due to the traffic being addressed to the reverse proxy.

4.2. Tiger Proxy basics

The Tiger Proxy is a proxy-server.

It comes in two flavours: Tiger Proxy and Tiger Standalone Proxy.

The standalone tiger proxy is started from a JAR-file.

The test environment manager boots the main tiger proxy (local tiger proxy) and also any additional ones (normal tiger proxy, not standalone).

Both types have a proxy-port (configurable via `tigerProxy.proxyPort`), which supports both http-

and https-traffic, (so you do not have to differentiate between the two). Additionally, they have an admin-port (configurable via `tigerProxy.adminPort`). This provides the Tiger Proxy Log to monitor the traffic (described in detail [here](#)), a rest-interface to customize the behavior (add/delete route, add/delete modifications) and a web-socket interface to stream rbel-messages between multiple Tiger Proxies.

4.3. Understanding routes

Routes are the fundamental mechanic of how the Tiger Proxy handles traffic.

They can be for a forward- or reverse-proxy.

A route has the following properties:

4.3.1. from

From where should the traffic be collected?

This can either be an absolute URL (e.g. `http://foobar`), which defines a forward-proxy route, or relative (e.g. `/blub`), defining a reverse-proxy-route.

Please note: You can freely add parts (e.g. `http://foobar/extra/part`) to further specify the mapping.

You can add multiple routes that match the same URL.

If multiple matches are found the most specific route is select.

For example if you have two routes `/` and `/foo` then for a request to `/foo/bar` the route `/foo` will be selected.

A route will only match when the proxy-mode is met.

A reverse-proxy-route will thus not match when a forward-proxy request is received.

To disable this proxy-mode matching you can set the flag `matchForProxyType` to false (default is true).

4.3.2. to

The target of the mapping.

This has to be an absolute URL.

The Tiger Proxy will, upon receiving a request to this mapping, execute a matching request to the defined host.

An example.

Consider the following route:

```
tigerProxy:  
  proxyRoutes:  
    - from: http://my.domain/  
      to: http://orf.at/
```

The "http://" in the **from property** indicates that we have a forward-proxy route defined.

So when we execute: (assuming the Tiger Proxy is started locally under the port 1234)

```
curl -x http://localhost:1234 http://my.domain/news
```

The result will match the following curl

```
curl http://orf.at/news
```

Additional headers are kept and just patched through.

The same goes for the body and the HTTP-Method.

Added parts of the from-URL are stripped when forwarding.

Meaning: If you have a mapping

```
tigerProxy:  
  proxyRoutes:  
    - from: http://my.domain/deep/  
      to: http://orf.at/blub/
```

and you execute GET <http://my.domain/deep/deeper>, you will get the result of GET <http://orf.at/blub/deeper> (the /deep in between has been eliminated along with my.domain).

4.3.3. Trailing Slashes

Trailing slashes in routes may be significant for the server.

Thus, the handling in the tiger-proxy is important.

To achieve a consistent behavior while maintaining ease of use, the following rules apply:

- If the request is longer than the from-path of the route (eg. if the "from" is "/foo" and the request is "/foo/bar", then "/bar" is the deep-path), the trailing slash behavior from the request is taken. This only is the case for more path-fragments (/foo/bar), and not simply a longer string (/foobar), as this would not be matched to the route.
- If the to-URL has a trailing slash, the request is forwarded with a trailing slash.
- Finally, if the request ends with a slash and the from-URL does NOT end with one, a trailing slash is added.
(The logic being the slash in the from-URL expresses an anticipated slash.
Thus, a slash being present in the request is simply "expected".)

This leads to the following examples:

from	to	request	forwarded to
"/webapp/"	"/api/"	"/webapp"	"/api/"
"/webapp/"	"/api"	"/webapp"	"/api"
"/webapp/"	"/api/"	"/webapp/"	"/api/"
"/webapp/"	"/api"	"/webapp/"	"/api"
"/webapp"	"/api/"	"/webapp"	"/api/"
"/webapp"	"/api"	"/webapp"	"/api"

from	to	request	forwarded to
"/webapp"	"/api/"	"/webapp/"	"/api/"
"/webapp"	"/api"	"/webapp/"	"/api/"
"/webapp<irrelevant>"	"/api<irrelevant>"	"/webapp/foo"	"/api/foo"

4.3.4. hosts

In some instances you might be pressed to "hosts" multiple servers on one tiger-proxy. This can happen when you can influence DNS-Resolvement, but neither the path nor the port used. In this case you can use the hosts-property to define which hosts should be routed to which target.

```
tigerProxy:
  proxyRoutes:
    hosts:
      - "www.google.com"
      - ".*.bing.com"
```

The entries are matched first as a total match, failing that as regular expressions.

If you want to add ports you are free to do so: "myHost:80".

This however will necessitate that the ports on all entries in this specific list are given (this does not apply to other routes).

4.3.5. Multiple targets

Sometimes you can only at runtime know which target to use.

This can be achieved by using the **to**-property as a list of targets:

```
tigerProxy:
  proxyRoutes:
    - from: /
      to:
        - http://orf.at/blub/
        - http://ard.de/bla/
```

When booting the tiger-proxy the proxy tries to reach the target host by sending a HEAD-request to the host (dropping the path).

If the server sends a valid HTTP (or HTTPS) response, the target is considered reachable and the route is used.

If no target can be reached an exception is thrown.

The return code is ignored.

4.3.6. criterions

As an additional measure for a more fine-grained matching you can use criterions. These are JEXL-expressions which have to be fulfilled for the route to be used.

This can be leveraged to make the routing decision dependent on the content of the message.

```
tigerProxy:  
  proxyRoutes:  
    - from: /  
      to: http://orf.at/blub/  
      criterions:  
        - $.header.foo == 'bar'
```

This will only forward messages where the header contains a key "foo" with the value "bar".

4.3.7. disableRbelLogging

You can deactivate the rbel-Logging on a per-Route basis.

Rbel is a versatile and powerful tool, but the analysis of individual messages consumes a lot of both CPU and memory.

Deactivating it for routes in which it is not needed is therefore a good idea.

4.3.8. Authentication

You can add optional authentication configuration which will be added to the forwarded message. Here either Basic access authentication or Bearer Token can be used:

```
tigerProxy:  
  proxyRoutes:  
    - from: http://my.domain/basicAuth/  
      to: http://orf.at/blub/  
      authentication:  
        username: "test1"  
        password: "pwd2"  
    - from: http://my.domain/bearerToken/  
      to: http://orf.at/blub/  
      authentication:  
        bearerToken: "blubblab"
```

Please note that the phrase "Bearer " will be added automatically.

Please do not add it yourself!

4.3.9. matchForProxyType

By default, a proxy route will only match when the proxy-mode defined by the `from` attribute is met. For example, a route with `from: /foo` will not match with a forward-proxy request like so:

```
curl -x localhost:59163 http://backend.domain/
```

To enable matching for both proxy-modes you can use the `matchForProxyType`-flag:

```
tigerProxy:
```

```
proxyRoutes:  
  - from: /foobar  
    to: http://orf.at/blub/  
    matchForProxyType: false
```

This will match with both of the following requests:

```
curl -x localhost:59163 http://backend.domain/foobar  
curl http://localhost:59163/foobar
```

4.4. TLS, keys, certificates a quick tour on proxies

A fundamental part of a proxy setup is TLS.

Since a proxy is a constant man-in-the-middle attack TLS is designed to make this exact scenario (eavesdropping while forwarding) impossible.

Since a lot of the traffic in the gematik context is security-relevant and thus TLS-secured this point is a very relevant one.

Fundamentally breaking into TLS requires two things:

- A certificate which the server can present which is valid for the given domain
- The certifying CA (or a CA reachable via a certification path) has to be part of the client truststore

There are different ways to reach these two requirements.

Which one should be taken is dependent on the setting and the client used (most importantly, of course: can you alter the truststore for the test-setup?)

Here are a few things to know and ways in which to enable TLS:

4.4.1. TLS and HTTPS-Proxy

TLS can be done via a http- or a https-proxy.

The proxy-protocol does NOT equate to the client-server-protocol.

To minimize the headache in configuration it is therefore strongly recommended to simply always use the http-proxy (sidenote: using a http-proxy does NOT reduce the security of the overall protocol).

The security still relies on server-certificate-verification.)

If, however, you can not avoid using the https-proxy you have to make sure that you add the given certificate to your truststore.

In class TigerProxy.java in Tiger there are methods such as SSLContext getConfiguredTigerProxySslContext(), X509TrustManager buildTrustManagerForTigerProxy() and KeyStore buildTruststore() which can help you configure the SSLContext in your case, if you use HTTP 3rd party libraries (Unirest, okHttp, RestAssured, etc.) as well as vanilla Java.

If you encounter any problems, please contact us.

4.4.2. Dynamic server identity

For successfully breaking into TLS traffic the Tiger Proxy needs to present a certificate which features the domain-name of the server.

Since the domain-names are known only at runtime, we generate the needed certificate on-the-fly during the first connection.

For a forward-proxy this is easy: The client sends not only the path, but the complete URL to the proxy, letting him handle DNS-resolution.

So when the Tiger Proxy receives a new request the necessary domain-name is given by the client.

A new, matching, certificate is generated (these are cached) and presented.

To complete the setup the client-truststore needs to be patched.

The CA used by the Tiger Proxy is dynamically generated on each startup.

For a reverse-proxy the domain name, which should be used, is unknown to the Tiger Proxy (DNS-resolution is done on the client-side).

Thus, a domain-name needs to be provided, which should be used for certificate-generation:

```
tigerProxy:  
  tls:  
    domainName: deep.url.of.server.de
```

4.4.3. Client-sided truststore modification

When using a non-default certificate (which will almost always be the case for the Tiger Proxy) the modification of the client-truststore is necessary.

For cases where the client is running in the same JVM as the target Tiger Proxy (which is the typical case for a tiger-based testsuite) there exists helper method to make this task easier.

Depending on your HTTP- or REST- or SOAP-API you will need to choose the exact way yourself. The following two examples might give you some idea of what to do.

```
Unirest.config().sslContext(tigerProxy.buildSslContext());
```

```
OkHttpClient client = new OkHttpClient.Builder()  
  
  .proxy(new Proxy(  
    Proxy.Type.HTTP,  
    new InetSocketAddress(  
      "localhost",  
      tigerProxy.getPort())))  
  
  .sslSocketFactory(  
    tigerProxy.getconfiguredTigerProxySslContext().getSocketFactory(),  
    tigerProxy.buildTrustManagerForTigerProxy())
```

```
.build();
```

4.4.4. Custom CA

If you can not or don't want to alter the client-truststore you have two choices: You can either provide a custom CA to be used (and trusted by the client) or you can give the certificate to be used by the Tiger Proxy.

To set a custom CA to be used for certificate generation simply specify it:

```
tigerProxy:  
  tls:  
    serverRootCa: "certificate.pem;privateKey.pem;PKCS8"  
  # for more information on specifying PKI identities in tiger see "Configuring PKI  
  identities"
```

4.4.5. Fixed server identity

The final, easiest and most unflexible way to solve TLS-issues is to simply give a fixed server-identity.

This identity will be used for all routes, but only if it matches the domain-name given by the client during the handshake.

As a fallback (if the domain-name does not match) the dynamic server-identity will be used.

```
tigerProxy:  
  tls:  
    serverIdentity: "certificateAndKeyAndChain.p12;Password"
```

If you want to use the fixed-server identity only for requests to matching hosts and you have NOT supplied a `serverRootCa`, you can use the `allowGenericFallbackIdentity`-property:

```
tigerProxy:  
  tls:  
    serverIdentity: "certificateForFoobar.p12;Password"  
    allowGenericFallbackIdentity: true
```

If `certificateForFoobar.p12` contains a certificate for the FQDN `foobar.com`, then a request to `foobar.com` will use this certificate.

If the request is to `barfoo.com` the generic server-identity will be used.

If you have supplied a `serverRootCa` this CA will always be used for any request for which an exact match is not found.

4.4.6. Multiple server identities

Sometimes you might want to use different server-identities for different hosts that are proxied. This can be achieved by using the `serverIdentities`-property.

Simply list the properties, the proxy will automatically try to choose the correct one.

```
tigerProxy:  
  tls:  
    serverIdentities:  
      - "someIdentity.p12;00"  
      - "anotherIdentity.p12;changeit"
```

4.4.7. OCSP stapling

If you want the Tiger Proxy to use OCSP stapling you can directly specify the OCSP-Signer to use in the configuration.

```
tigerProxy:  
  tls:  
    ocspSignerIdentity: "myOcspSigner.p12;Password"
```

The server will then use this OCSP-Signer to create a fake OCSP-Response during the TLS-handshake.

4.4.8. TLS Decryption in wireshark

Sometimes you might want to look at decrypted TLS-traffic in wireshark.

To achieve this we need to extract the masterSecrets of every connection from the Tiger Proxy and provide them to wireshark.

This is actually pretty straight forward, with one big caveat: It is very insecure to access the masterSecrets of a TLS-connection, so we need to attach a Java-Agent to the VM.

When you are using the Tiger Proxy in a normal tiger-testsuite you can simply add the `<goal>attach-tiger-agent</goal>` goal to the tiger-maven-plugin:

```
<plugin>  
  <groupId>de.gematik.test</groupId>  
  <artifactId>tiger-maven-plugin</artifactId>  
  <version>${project.version}</version>  
  <executions>  
    <execution>  
      <phase>generate-test-sources</phase>  
      <goals>  
        <goal>generate-drivers</goal>  
        <goal>attach-tiger-agent</goal>  
      </goals>  
    </execution>  
  </executions>  
</plugin>
```

This will modify the `argLine` property used by failsafe to start the testsuite and attach the

TigerAgent to the VM.

Next we need to set the filename where to write the masterSecrets to:

```
tigerProxy:  
  tls:  
    masterSecretsFile: "myMasterSecrets.txt"
```

The Tiger Proxy will write the secrets to the given file.

The final step is to import the masterSecrets into wireshark:

Go to Edit → Preferences.

Open the Protocols tree and select TLS.

Alternatively, select a TLS packet in the packet list, right-click on the TLS layer in the packet details view and open the Protocol preferences menu.

Set the **(Pre)-Master-Secret log filename** to the file containing the master secrets.

Now wireshark should decrypt TLS traffic routed through the Tiger Proxy on-the-fly.

4.5. Modifications

Modifications are a powerful tool to alter messages before forwarding them.

They can be applied to requests and responses, to routes in forward- and reverse-proxy-mode.

You can choose to modify only specific parts of the message and only alter messages, if certain conditions are met.

Response messages support so called "reason phrases" which are small text explanations to the response code, e.g. "200 OK", ("OK" is a reason phrase).

You can add, modify and remove reason phrases.

Below is a sample configuration giving insight into how modifications are organized:

```
tigerProxy:  
  modifications:  
    # a list of modifications that will be applied to every proxied request and  
    # response  
  
    -  
      # The following modification will replace the entire "user-agent" in all  
      # requests  
      -  
        condition: "isRequest"  
        # a condition that needs to be fulfilled for the modification to be applied  
        # (JEXL grammar)  
        targetElement: "$.header.user-agent"  
        # which element should be targeted?  
        replaceWith: "modified user-agent"  
        # the replacement string to be filled in.  
  
        # The following modification will replace the body of every 200 response  
        # completely with the given json-string
```

```

# (This ignores the existing body. For example this could be an XML-body.
Content-Type-headers will NOT be set accordingly)

condition: "isResponse && $.responseCode == 200"
targetElement: "$.body"
name: "body replacement modification"
# The name of this modification. This can be used to identify, alter or remove
this modification. A name is optional
replaceWith: "{\"another\":{\"node\":{\"path\":\"correctValue\"}}}"

# The following modification has no condition, so it will be applied to every
request and every response

targetElement: "$.body"
regexFilter: "ErrorSeverityType:(Error)|(Warning)"
# The given regex will be used to target only parts of targeted element.
replaceWith: "ErrorSeverityType:Error"

```

4.6. Mesh set up

One of the fundamental features of the Tiger Proxy is mesh set up AKA rbel-message forwarding. This transmits the information about the messages, which the proxy has logged, to other Tiger Proxies (where they will be logged as well).

This enables the creation of "proxy-meshes", staggered Tiger Proxies.

In a mesh set up the "remote tiger proxy" is the one which intercepts the traffic and sends the information.

Conversely, the "receiving tiger proxy" receives the information about the message from the remote tiger proxy.

The "local tiger proxy" is the main tiger proxy booted by the testsuite.

If you configured it to receive traffic from another tiger proxy (which should always be the case when you are doing a mesh set up) then it is also a receiving tiger proxy.

Common scenario for this approach might be the use of multiple reverse-proxies on the root level (e.g. when the client only allows the configuration of the server IP or domain, but no path-prefix) or the aggregation of traffic across machine-boundaries (e.g. one constantly running Tiger Proxy which is used by a testsuite on another machine).

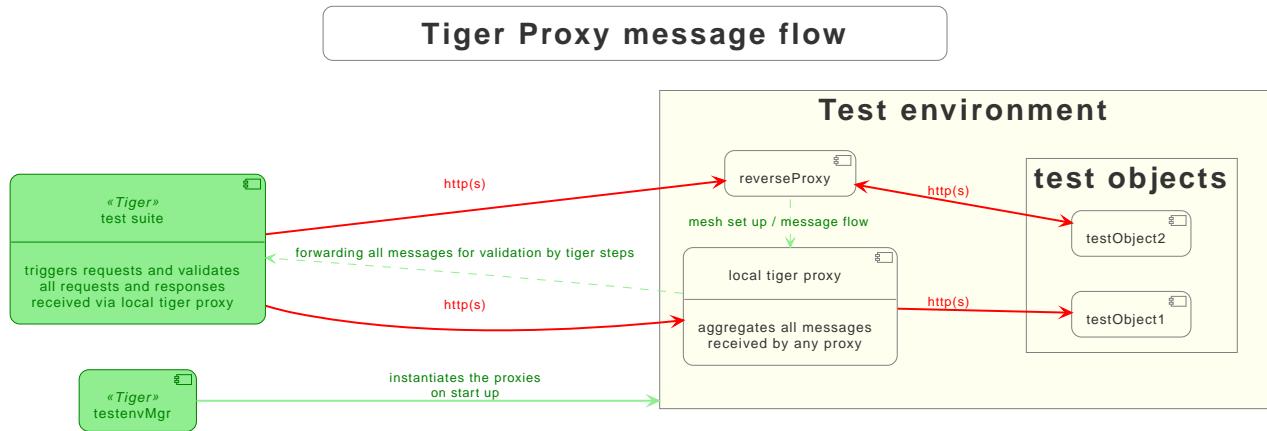


Figure 6. Tiger Proxy message flow

In the above picture the test object 2 would not be accessible directly by the test suite, thus using the reverse proxy allows circumventing network restrictions.

The reverse proxy could either be started by the test environment manager or as standalone process.

```

tigerProxy:
  proxyId: IBM
  trafficEndpoints:
    - http://another.tiger.proxy:<adminPort>
      # A list of upstream Tiger Proxies. This proxy will try to connect to all given
      sources to
      # gather traffic via the STOMP-protocol.
      skipTrafficEndpointsSubscription: false
      # If false then the subscription is tested at the beginning and if any of the
      given endpoints are not accessible the
      # server will not boot. (fail fast, fail early)
      # default of skipTrafficEndpointsSubscription is false
      downloadInitialTrafficFromEndpoints: true
      # Should the traffic currently available (cached) in the remote be download upon
      initial connection?
      # default of downloadInitialTrafficFromEndpoints is true
      failOnOfflineTrafficEndpoints: false
      # Should the Tiger Proxy fail on startup, when a traffic endpoint is offline?
      Default is true.
      # Ignoring this error might lead to an unexpected testing environment!

```

Please be advised to use the server-port (`server.port`) here, not the proxy-port (`tigerProxy.proxyPort`).

The traffic from routes with `disableRbelLogging: true` will not show up here.



If you are setting up a Tiger Proxy to run constantly and simply forward traffic to a testsuite that is booted ad-hoc you might run into performance-problems.
This is due to the Rbel-Logger being a very hungry beast.
To stop Rbel from parsing all message simply add `tigerProxy.activateRbelParsing: false`.

`false`.

This will vastly reduce memory and CPU consumption of the application, while still forwarding logged traffic.

4.6.1. Mesh API

The Tiger Proxies use [STOMP](#) a simple/streaming text oriented messaging protocol via web socket to forward received traffic.

For an external client to receive these traffic data, it must subscribe to the traces topic reachable at the subscription path `/topic/traces`.

To do so the client must connect to the traffic endpoint URL of the Tiger Proxy.

This is answered with HTTP status 100 and then redirected to web socket protocol via the same port.

For each received traffic data pair (request/response) the Tiger Proxy will push a web socket message to all subscribed clients.

This JSON encoded message consists of:

* UUID string * http request as base64 encoded data * http response as base64 encoded data * hostname and port of sender (if retrievable, worst case only IP address or empty) * hostname and port of receiver (if retrievable, worst case only IP address or empty)

```
{  
  "uuid": "UUID string",  
  "request": "base64 encoded http request",  
  "response": "base64 encoded http response",  
  "sender": {  
    "hostname": "hostname/ip address of sender",  
    "port": portAsInt  
  },  
  "receiver": {  
    "hostname": "hostname/ip address of receiver",  
    "port": portAsInt  
  }  
}
```

4.7. Adding notes to messages

To add notes to certain messages, you can add a list of criteria with messages that shall be added to any message in the rbel log matching the specific criterion.

```
tigerProxy:  
  proxyId: meshProxy2  
  notes:  
    - message: "This is a note on the HTTP method"  
      jexlCriterion: "isRequest && path == '$.method'"  
    - message: "Hackers were here..."
```

```
jexlCriterion: "element.decryptedUsingKeyId == 'mySuperSecretKey'"
```

4.8. Understanding RBelPath

RBeL-Path is a XPath or JSON-Path inspired expression-language enabling the quick traversal of captured RBeL-Traffic (navigation of the RbelElement-tree).

A simple example:

```
assertThat(convertedMessage.findRbelPathMembers("$.header"))
    .containsExactly(convertedMessage.getFacetOrFail(RbelHttpMessageFacet.class)
        .getHeader());
```

or

```
assertThat(convertedMessage.findElement("$.header"))
    .get()
    .isSameAs(convertedMessage.getFacetOrFail(RbelHttpMessageFacet.class).getHeader());
);
```

(The first example executes the RbelPath and returns a list of all matching element, the second one returns an Optional containing a single result.

If there are multiple matches an exception is given.)

RBeL-Path provides seamless retrieval of nested members.

Here is an example of HTTP-Message containing a JSON-Body:

1 Response

200

RES Headers

name	value
nbf	nbf-Wert in http header
Cache-Control	max-age=300

\$.header.nbf

RES Body

JWT

Headers

`$.body.header.alg`

Figure 7. Rbel-Path expression in a HTTP-Response

The following message contains a JWT (Json Web Token, a structure which contains of a header, a body and a signature).

In the body there is a claim (essentially a Key/Value pair represented in a JSON-structure) named `nbf` which we want to inspect.

Please note that the RBeL-Path expression contains no information about the types in the structure. This expression would also work if the HTTP-message contained a JSON-Object with the corresponding path, or an XML-Document.

```
assertThat(convertedMessage.findRbelPathMembers("$.body.body.nbf"))
    .containsExactly(convertedMessage.getFirst("body").get()
        .getFirst("body").get()
        .getFirst("nbf").get()
        .getFirst("content").get());
```

(The closing `.getFirst("content")` in the assertion is due to a fix to make RbelPath in JSON-Context easier: If the RbelPath ends on a JSON-Value-Node the corresponding content is returned.)

RES Body

JWT

Headers

```
{
  "alg": "BP256R1",
  "kid": "discSig",
  "x5c": [
    "MIICsTCCAligAwIBAgIHAbssqQhqOzAKBggqhkjOPQQDAjCBhDELMakGA1UEBhMCREUxHzAdBgNVBAoLUNBIGR1ciBUZWx1bWF0aWtpbmZyYXN0cnVrdHVyMSAwHgYDVQQDBdHRU0uS09NUC1DQTEwIFRFU1QtT05MkRFMSYwJAYDVQQKDB1nZW1hdGlrIFRFU1QtT05MWSAtIE5PVC1WQUxJRDESBAGA1UEAwJSURQIFNpZyAzMr/bz6BTcQO5pbeum6qQzYD5dDCcriw/VNPPZCQzXQPg4StWyy500q9TogBEmOjge0wgeowDgYDVR0PAQH/BAUAEwEggQwIQYDVR0gBBowGDAKBggqghQATASBSzAKBggqghQATASBIzAfBgNVHSMEGDAWgBQo8Pjmqch3zENLy9laGNhLmd1bWF0aWsZGUvb2NzcC8wHQYDVR0OBBYEFC94M9Lgw44lNgoAbkPaomnLjs8/MAwGA1UdEwEBU/YGNlRc7+kBHcCIBuzba3GspqSmoP1VwMeNNKNaLsgV8vMbDJb30aqaiX1"
  ]
}
```

Body

\$.body .body .nbf

```
{
  "authorization_endpoint": "http://localhost:8080/sign_response",
  "alternative_authorization_endpoint": "http://localhost:8080/alt_response",
  "sso_endpoint": "http://localhost:8080/sso_response",
  "pairing_endpoint": "http://localhost:8080/pairing",
  "token_endpoint": "http://localhost:8080/token",
  "jii_disc": "http://localhost:8080/discoveryDocument",
  "issuer": "https://idp.central.idp.splitdns.ti-dienste.de",
  "jwks_uri": "http://localhost:8080/jwks",
  "exp": 1614425703,
  "nbf": 1614339303,
  "iat": 1614339303,
  "url_pkcs12_end": "https://localhost:8080/128bit/jwks.json"
}
```

Figure 8. Multiple body references

You can also use wildcards to retrieve all members of a certain level:

```
$.body.[*].nbf
```

or

```
$.body.*.nbf
```

Alternatively you can recursively descend and retrieve all members:

```
$..nbf
```

and

```
$.body..nbf
```

will both return the same elements (maybe amongst other elements).

To use keys containing spaces, escape them via `['foo bar']`, like so:

```
$.body,['foo bar'].key
```

Please note that the keys in the bracket are URL unescaped.

So to use special characters please URL encode them (Space is a special case since + and ' ' are allowed, depending on the exact position).

4.8.1. Arrays

To make things easy and consistent, the entries of an array are simply stored as a map with the index as key.

So the following expression will return the first element of the array:

```
$.body.array.0
```

4.8.2. Differentiating between multiple elements

When a key is present multiple times, all elements are returned.

To differentiate between them, you can use the index:

```
$.body.entry[0]
```

would give the first element in the following XML:

```
<body>
  <entry>first</entry>
  <entry>second</entry>
</body>
```

4.8.3. Alternate keys

To find alternating values, concatenate them using the pipe symbols, like so:

```
$.body,['foo'|'bar'].key
```

This expression will explore both subtrees to try to find the following nodes

`$.body.foo.key` and `$.body.bar.key`.

Please note that only elements that are present are returned.

So if only always one of the two elements is present, only a single element will be returned.

4.8.4. Case-insensitive matching

Sometimes it can be helpful to match keys in a case-insensitive manner.

To achieve this you can use the `~`-operator:

`$.body.[~'foo'].key`

This will match `$.body.foo.key` and `$.body.FOO.key` (and any other case-insensitive match).

To find multiple case-insensitive matches, concatenate them using the pipe symbols, like so:

`$.body.[~'foo'|~'bAR'].key`.

With this expression, the following nodes will be found: `$.body.foo.key`, `$.body.FOO.key`, `$.body.bar.key` and `$.body.BAR.key` (and any other potential matches).

4.8.5. JEXL expressions

RBeL-Path can be integrated with JEXL-expression, giving a much more powerful and flexible tool to extract certain element.

This can be done using the syntax from the following example:

```
$..[?(key=='nbf')]
```

The expression in the round-brackets is interpreted as JEXL.

The available syntax is described in more detail [here](#) or <https://commons.apache.org/proper/commons-jexl/reference/syntax.html>

Please note that these Jexl-Expression can not be nested inside each other deeper than one level
(You can write a RbelPath that contains a Jexl-Expression.

And this Jexl-Expression can even contain a RbelPath.

But the inner RbelPath can not contain another Jexl-Expression).

The variables that can be used are listed below:

- `element` contains the current RBeL-Element
- `parent` gives direct access to the parent element of the current element.
Is `null` if not present
- `message` contains the HTTP-Message under which this element was found.

It contains:

- `method` is the HTTP-Method (or null if it is a response)
- `url` is the request URL (or null if it is a response)
- `statusCode` is the status response code (or null if it is a request)
- `request` is a boolean denoting whether this message is a request

- `response` is a boolean denoting whether this message is a response
- `header` is a map containing all headers (as `Map<String, List<String>>`)
- `bodyAsString` is the body of the message as a raw string, or null if none given
- `body` is the RbelElement of the message-body, or null if none given
- `request` is the corresponding HTTP-Request.
If `message` is a response, then the corresponding Request will be returned.
If `message` is a request, then the `message` itself will be returned.
- `response` is the corresponding HTTP-Response.
If `message` is a request, then the corresponding Response will be returned.
If `message` is a response, then the `message` itself will be returned.
- `key` is a string containing the key that the current element can be found under in the parent-element.
- `path` contains the complete sequence of keys from `message` to `element`.
- `type` is a string containing the class-name of `element` (eg `RbelJsonElement`).
- `content` is a string describing the content of `element`.
The actual representation depends heavily on the type of `element`.

Additionally you can always reference the current element (via `@.`) or the root element (via `$.`) in any JEXL-expression.

Lets explain this using an example.

For more detailed information on JEXL expressions please refer to [Detailed JEXL-expressions](#).

4.8.6. Nested RbelPath expressions

Consider the following rbel tree:

Rbel Tree

```
└─body (eyJ0eXAiOiJpZHAtbGlzdCtqd3QiLCJraWQiOiJwdWtfZmVkbW...) (RbelJwtFacet)
  ├─header ({"typ":"idp-list+jwt","kid":"puk_fedmaster_sig","a...") (RbelJsonFacet)
  |  ├─typ ("idp-list+jwt") (RbelJsonFacet)
  |  |  └─content (idp-list+jwt) (RbelValueFacet)
  |  |  └─kid ("puk_fedmaster_sig") (RbelJsonFacet)
  |  |  |  └─content (puk_fedmaster_sig) (RbelValueFacet)
  |  |  └─alg ("ES256") (RbelJsonFacet)
  |  |  |  └─content (ES256) (RbelValueFacet)
  |  └─body ({"iss":"https://app-ref.federationmaster.de","iat"...}) (RbelJsonFacet)
  |    ├─iss ("https://app-ref.federationmaster.de") (RbelJsonFacet)
  |    |  └─content (https://app-ref.federationmaster.de) (RbelValueFacet,RbelUriFacet)
  |    |    └─basicPath (https://app-ref.federationmaster.de) (RbelValueFacet)
  |    ├─iat (1681810461) (RbelJsonFacet)
  |    |  └─content (1681810461) (RbelValueFacet)
  |    ├─exp (1681896861) (RbelJsonFacet)
  |    |  └─content (1681896861) (RbelValueFacet)
  |    └─idp_entity ([{"iss":"https://idpsek.dev.gematik.solutions","or...}) (RbelJsonFacet)
        └─0 ({"iss":"https://idpsek.dev.gematik.solutions","org...}) (RbelJsonFacet)
          ├─iss ("https://idpsek.dev.gematik.solutions") (RbelJsonFacet)
          |  └─content (https://idpsek.dev.gematik.solutions) (RbelValueFacet,RbelUriFacet)
          |    └─basicPath (https://idpsek.dev.gematik.solutions) (RbelValueFacet)
          |    └─organization_name ("gematik") (RbelJsonFacet)
          |      └─content (gematik) (RbelValueFacet)
          |    └─logo_uri (null) (RbelJsonFacet,RbelValueFacet)
          |    └─user_type_supported ("IP") (RbelJsonFacet)
          |      └─content (IP) (RbelValueFacet)
        └─signature (070. 0H0, !@H0D n@`@\r\0D0K0\lr@|@a@0@V\@L@[0D@ 0@Y...) (RbelJwtSignature)
        └─isValid (Value: true) (RbelValueFacet)
        └─verifiedUsing (Value: puk_fedmaster_sig) (RbelValueFacet)
```

Figure 9. Nested Rbel tree with array

At `$.body.body.idp_entity` we have an array with potentially multiple entries (here there is only one, entry `0`).

We want to select an entry where the `iss`-claim matches our expectation.

We can achieve this with using a nested Rbel-Path inside the JEXL-Expression:

```
$.body.body.idp_entity.[?(@.iss.content=='https://idpsek.dev.gematik.solutions')]
```

Here the `@.` references the current element: For each array entry the expression is tested, with `@.` always referring to the current entry.

To access elements starting from the root you can use `$.` like so:

```
$.body.body.idp_entity.[?(@.iss.content==$body.body.idp_entity.0.iss.content)]
```

You can use recursive descent here as well:

```
$.body.[?(@..content == 'ES256')] would yield $.body.header.
```

Let's unpack this expression:

- `$.body` selects the http body
- `.` then selects a child (of the http-body, meaning either `header`, `body` or `signature`)
- The JEXL-selector `[?(@..content == 'ES256')]` is then tested on each of the candidates.
 - In turn `@..` executes a recursive descent, meaning it will select all child nodes individually

- `content` selects only the elements which have a key matching `content`.
So we end up with all nodes in the respective subtrees that are named `content`.
- The JEXL-expression `* == 'ES256'` is then selected for every member of the subtree (so for the header it will test `$.body.header.typ.content`, `$.body.header.kid.content` and `$.body.header.alg.content`).
The individual results are then reduced using (so the overall expression matches if there is ANY matching element)
- Since only one of the subtrees does fulfill the expression only this subtree is returned (and NOT the element itself, i.e. `$.body.header.alg.content`)

Please note that since the RbelPath-expressions are executed prior to the JEXL-expression the negation might yield unexpected results.

Currently it is not recommended to use these. (e.g. `$.body.[?(not (@.. == 'ES256'))]`)

4.8.7. Debugging Rbel-Expressions

To help users create RbelPath-Expressions there is a Debug-Functionality which produces log message designed to help.

These can be activated by `RbelOptions.activateRbelPathDebugging()`.

Please note that this is strictly intended for development purposes and will flood the log with quite a lot of messages.

Act accordingly!

When you want to debug RbelPath in BDD test suites, you can add a `tiger.yaml` file to your project root and add the following property (for more details see [this chapter](#)):

```
lib:
  rbelPathDebugging: true
```

To get a better feel for a RbelElement (whether it being a complete message or just a part) you can print the tree with the `RbelElementTreePrinter`.

It brings various options:

```
RbelElementTreePrinter.builder()
  .rootElement(this) //the target element
  .printKeys(printKeys) // should the keys for every leaf be printed?
  .maximumLevels(100) // only descend this far into the tree
  .printContent(true) // should the content of each element be printed?
  .build()
  .execute();
```

4.9. Running Tiger Proxy as standalone JAR

If you only want to run a Tiger Proxy instance without test environment manager or test library you may do so (e.g. in certain tracing set-ups).

A spring boot executable JAR is available via [maven central](#).

Supplying an application.yaml file allows you to configure the standalone proxy just like an instance started by the test environment manager.

All properties can be used the same way as described in [this chapter](#).

There is however one additional property for the standalone proxy specifically:

```
# flag whether to load all resources (js,css) locally or via CDN/internet.  
# useful if you have no access to the internet in your environment  
localResources: false
```

4.10. Additional configuration

There are some additional configuration-flags in the Tiger Proxy:

4.10.1. Performance

Below some properties along with their respective default values:

```
tigerProxy:  
  activateRbelParsing: true  
  parsingShouldBlockCommunication: false  
  activateTrafficLogging: true  
  activateRbelParsingFor:  
    - epa-vau
```

activateRbelParsing

Deactivating this flag turns off all Rbel-Analysis of the incoming traffic.

This is a huge deal in terms of memory- and CPU-consumption, but you will lose all benefit of performing Rbel-Analysis.

activateRbelParsingFor

This option can activate various optional Rbel-Converters.

Currently supported are:

- **pop3** for RbelPop3CommandConverter and RbelPop3ResponseConverter
- **smtp** for RbelSmtpCommandConverter and RbelSmtpResponseConverter
- **mime** for RbelMimeConverter and RbelEncryptedMailConverter
- **asn1** for RbelAsn1Converter
- **epa-vau** for RbelVauEpaConverter and RbelVauEpaKeyDeriver
- **erp-vau** for RbelErpVauDecryptionConverter
- **epa3-vau** for RbelVauEpa3Converter
- **sicct** for RbelSicctCommandConverter and RbelSicctEnvelopeConverter
- **X500** for RbelX500Converter

- **X509** for RbelX509Converter and RbelX500Converter
- **ldap** for RbelLdapConverter
- **b64gzip** for RbelBase64GzipConverter
- **OCSP** for RbelOcspResponseConverter

These optional converters can also be deactivated at runtime via glue code.

You can deactivate parser during a specific test case.

See the [Complete set of steps in validation glue code](#).

parsingShouldBlockCommunication

If blocking is enabled the Tiger Proxy will only return the response when message parsing is completed.

This is inadvisable in high-speed scenarios.

It, however, greatly simplifies the test suite (after the communication is concluded the parsed message appears in the log).

Therefore, the blocking is deactivated by default.

The only exception is the local Tiger Proxy, which WILL block communication until parsing is completed.

For all Tiger Proxies this default behavior can be changed.

directReverseProxy

To enable the use of the TigerProxy for non-HTTP scenarios you can use the option **directReverseProxy**:

```
tigerProxy:
  directReverseProxy:
    hostname: 127.0.0.1
    port: 3858
    ignoreConnectionErrors: false
```

This will directly forward any request to the given host.

This is a form of reverseProxy, only also applicable for non-http-traffic.

HTTP traffic will still be forwarded through use of a global reverse proxy.

Other traffic will be directly forwarded, rerouted directly on the TCP layer.

Messages transmitted can still be parsed via RBel.

If the optional flag **ignoreConnectionErrors** is true, no connection errors will be logged, the default is false.

4.10.2. activateTrafficLogging

This flag controls whether the Tiger Proxy will log all traffic.

If activated every request and response is noted in the log.

This can lead to a verbose and bloated log.

If you are not interested in the traffic log, but only in the Rbel-Analysis, you can deactivate this flag.

Default is true.

4.10.3. rewriteHostHeader

This flag activates the rewriting of the host-header.

If activated the host-header will be rewritten to the target host (only applicable for reverse proxy routes).

Default is false.

4.10.4. rewriteLocationHeader

This flag activates the rewriting of the location-header for 3xx responses.

If activated the location-header will be modified so the client will still use the proxy to reach the new location.

Default is true.

4.10.5. maxLoopCounter

By default, the Tiger Proxy will keep count of the number of requests it executes on itself (e.g. when a request is forwarded to the Tiger Proxy itself).

If this counter exceeds the value of `maxLoopCounter` the request will be aborted.

The default value is 10 (Meaning after 10 loops the request will end in an error).

Note: This behavior can only be customized for all Tiger Proxies.

4.11. Understanding filtering

The filtering of messages in the tiger proxy consists of three main stages.

These are:

- Traffic filter (`trafficEndpointFilterString` / `readFilter`, Determines which messages are accepted into the tiger proxy)
- Tiger Proxy Log filter (Which messages are displayed in the Tiger Proxy Log?)
- Pagination (Look around in smaller pages of messages)

Lets dive a bit deeper!

4.11.1. Traffic filter

At the core of the Tiger Proxy sits a `RbelLogger` instance.

Here the messages are parsed and stored.

Three sources feed into the `RbelLogger`:

- Messages intercepted in the Tiger Proxy
- Messages relayed using a mesh setup
- Messages imported from a file

Messages that are intercepted are automatically stored (the exception being the `tigerProxy.activateForwardAllLogging`-property, which can deactivate the logging of traffic not specifically forwarded via a route).

For messages in a mesh setup and from a source file filter expressions can be defined to limit the messages that are actually stored.

These can be defined using the `tigerProxy.trafficEndpointFilterString` (for mesh setups) and `tigerProxy.fileSaveInfo.readFilter` (for tgr-files) respectively.

When messages pass the filter, partner messages (request/response pairs) are kept intact.

So when you filter for messages that have a return code of 200 the corresponding requests do not match the filter expression.

They are however kept in memory since the partner, the response in that case, do match.

Filter expressions are [JEXL-expressions](#).

4.11.2. Tiger Proxy Log filter

When you display the messages on the Tiger Proxy Log you have the ability to filter out certain messages to be displayed exclusively.

The messages, which are filtered out, do still remain stored in the Tiger Proxy.

Consequently, this has no effect if you store a TGR file (be it via the Tiger Proxy Log or the YAML).

The menu on the right side will only show the messages being filtered out to avoid confusion.

However, the messages numbers do reference the order in the main Tiger Proxy store.

This way they are consistent across different Tiger Proxy Log filters (message #10 will always refer to the same message, regardless of the Tiger Proxy Log filter being applied).

Filter expressions are [JEXL-expressions](#).

4.11.3. Pagination

Finally, pagination is applied in the Tiger Proxy Log.

This comes after the Tiger Proxy Log-Filter has been applied.

So when would filter out every second message via a Tiger Proxy Log-Filter every page would still contain 20 (or whatever page size you have set) messages.

Chapter 5. Tiger Test library

As outlined in [the overview section](#) the Tiger test library is one of the three core components of the Tiger test framework.

Its main goal is to provide extensive support for BDD/Cucumber testing and integrating the local Tiger Proxy with the test environment manager and the started test environment.



As of now we do not support multithreaded / parallel test runs.

5.1. Tiger test lib configuration

In the root folder of your test project you may place a *tiger.yaml* configuration file to customize the Tiger test library integration and activate / deactivate certain features.

```
lib:  
  # Flag to activate tracing at the Rbel Path Executor.  
  # If activated the Executor will dump all evaluation steps of all levels to the  
  console  
  # when traversing through the document tree  
  # Deactivated by default  
  rbelPathDebugging: false  
  # Flag whether the Executor's dump shall be in ANSI color.  
  # If you are working on operating systems (Windows) that do not support  
  # Ansi color sequences in their console you may deactivate the coloring with this  
  flag.  
  # Activated by default.  
  rbelAnsiColors: true  
  # Flag whether to start a browser window to display  
  # the current steps / banner messages / rbel logs  
  # when executing the test suite.  
  # This feature can be used to instruct the tester to follow  
  # a specific test workflow for manual tests.  
  # Deactivated by default  
  activateWorkflowUi: false  
  # Flag whether to add a curl command details button to  
  # SerenityRest Restassured calls  
  # in the Serenity BDD report  
  addCurlCommandsForRaCallsToReport: true  
  # Flag whether to create the RBEL HTML reports during  
  # a testsuite run, activated by default  
  createRbelHtmlReports: true  
  # maximum amount of seconds to wait / pause execution via pop up in the workflow  
  ui, default is 5 hours.  
  pauseExecutionTimeoutSeconds: 18000  
  # Customize your Rbel-Logs with your own logo. Must be PNG format.  
  # Path should be relative to execution-location  
  rbelLogoFilePath: "myLogo.png"  
  # If you want to use a fixed port for the workflowUI you can set it here.
```

```

workflowUiPort: 8123
# Flag whether to clear the initial traffic after the environment startup phase
# activated by default
clearEnvironmentStartupTraffic: true

# flag whether to start the local Tiger Proxy (default) or to omit it completely.
# if you have the local Tiger Proxy deactivated you will NOT be able to
# validate / log any traffic data from test requests / responses.
localProxyActive: true

# section to allow adapting the logging levels of packages/class loggers similar to
spring boot
logging:
  level:
    # activate tracing for a specific class
    de.gematik.test.tiger.testenvmgr.TigerTestEnvMgr: TRACE
    # activate tracing for all classes and subpackages of a package
    de.gematik.test.tiger.proxy: TRACE
    # activate tracing for the local Tiger Proxy. This logger has a special name due
    to its importance in the tiger test framework
    localTigerProxy: TRACE

```

5.2. Cucumber and Hooks

As Tiger focuses on BDD and Cucumber based test suites all the setup and tear down as well as steps based actions are performed.

That's why it is mandatory to use the TigerCucumberRunner, which internally registers the plugin **io.cucumber.core.pluginTigerSerenityReporterPlugin** to the plugins list.

The LocalProxyRbelMessageListener class initializes a static single RBelMessage listener to collect all messages received by the local Tiger Proxy and provides those messages via a getter method to the Tiger filter and validation steps.

At startup of the TigerCucumberRunner the TigerDirector gets called once to initiate the Tiger test environment manager, the local Tiger Proxy (unless it's configured to be not active) and optionally the workflow UI.

It adds a RbelMessage Listener once to the local Tiger proxy and also clears the RbelMessages queue before each scenario / scenario outline variant.

Utilizing the close integration of SerenityBDD and RestAssured at startup also a Restassured request filter, which parses the details and adds a curl Command details button to the Serenity BDD report, is registered.

The curl command shown in that section in the report allows to repeat the performed REST request, for manual test failure analysis.

After each scenario / data variant all collected RbelMessages are saved as HTML file to the **target/rbellogs** folder, and attached to the SerenityBDD report as test evidence.

The current test run state (success/failed rate) is logged to the console.

5.3. Using the Cucumber Tiger validation steps

The Tiger validation steps are a set of Cucumber steps that enable you to search for requests and associated responses matching certain criteria.

All of that without need to write your own code.

Basic knowledge about RBelPath and regular expressions are sufficient.

In order to use these steps you must ensure that the relevant traffic is routed via the local Tiger Proxy of the test suite or construct a [Tiger Proxy mesh](#) set up.

5.3.1. Filtering requests

Core features

- Filter for server, method, path, RBelPath node existing / matching given value in request
- Find first / next / last matching request
- Find absolute last request (no path input needed)
- Find first / next / last request containing a RBelPath node
- Clear all recorded messages
- Specify timeout for filtering request

With the `TGR find next request ...` steps you can validate a complete workflow of requests to exist in a specific order and validate each of their responses (see next chapter).

5.3.2. Validating responses

Core features

- Assert that the body of the response matches regex
- Assert that a given RBelPath node exists
- Assert that a given RBelPath node matches regex
- Assert that a given RBelPath node does not match regex
- Assert that a given RBelPath node matches a JSON struct using the JSONChecker feature set
- Assert that a given RBelPath node matches an XML struct using the XMLUnit difference evaluator

Listing 15. Tiger response validation steps example

```
Feature Tiger validation steps
```

```
Scenario: Example steps
```

```
    Given TGR clear recorded messages
    And TGR filter requests based on host "testnode.example.org"
    And TGR filter requests based on method "POST"
    And TGR set request wait timeout to 20 seconds
    When TGR find first request to path "/path/path/blabla" with "$..tag.value.text"
        matching "abc.*"
```

```

And TGR find first request to path "/path/path/blabla" containing node "$..tag"
Then TGR current response with attribute "$..answer.result.text" matches "OK.*"
But TGR current response with attribute "$..answer.reason.text" does not match
"REQUEST.*"
And TGR current response body matches:
"""
    body content
"""

And TGR current response at "$..tag" matches as JSON:
"""
{
  "arr1": [
    "asso", "bss0"
  ]
}
"""

And TGR current response at "$..tag" matches as XML:
"""
<arr1>
  <entry index="1">asso</entry>
  <entry index="2">bss0</entry>
</arr1>
"""

Given TGR find message with ".$pop3Body.mimeHeader.subject" matching ".*test.*"
Given TGR find next message with ".$pop3Status" matching "-ERR"
Given TGR find last message with ".$pop3Command" matching "RETR"

```

5.3.3. Validating requests

Core features

- Assert that the body of the request matches regex
- Assert that a given RBelPath contains node
- Assert that a given RBelPath node matches regex
- Assert that a given RBelPath matches regex
- Assert that a given RBelPath node matches as JSON or XML
- Assert that a given RBelPath node does not match regex

Listing 16. Tiger request validation steps example

Feature Tiger request validation steps

Background:

Given TGR clear recorded messages

Scenario: Test validation Request

When TGR send PUT request to "http://httpbin/put" with body "{'foo': 'bar'}"

Then TGR find last request to path ".*"

```

Then TGR current request body matches:
"""
{'foo': 'bar'}
"""

Then TGR current request contains node "$.body.foo"
Then TGR current request with attribute "$.body.foo" matches "bar"
Then TGR current request at "$.body" matches:
"""
{'foo': 'bar'}
"""

Then TGR current request at "$.body" matches as JSON:
"""
{
  "foo": "${json-unit.ignore}"
}
"""

Then TGR current request with attribute "$.body.foo" does not match "foo"

```

5.3.4. Regex matching

When comparing values (e.g. in the `TGR current response body matches:`) generally the algorithms check for equality and only check for regex matches if they were not equal.

5.3.5. Complete set of steps in validation glue code

Unresolved directive in tigerTestLibrary.adoc - include::./RBelValidatorGlueCommentsOnly.adoc[]

5.4. Modifying RbelObjects (RbelBuilder)

5.4.1. Introduction

Tiger supports modifying JSON, XML and several token formats.

After loading in an object from a string or a file the RbelObject can be modified in multiple ways:

- Changing a primitive value at a certain path
- replacing a primitive node by an object node and vice versa
- adding new nodes and primitive values as child path of an existing path
- adding new nodes to an array/list

After the adjustments the values of the modified RbelElements can be asserted.

The object can be serialised back to a string.

Jexl Expressions are implemented, as for reading a file or for serializing:

`'!{rbelObject:serialize("myRbelObject")}'`

5.4.2. List of all possible Steps

Using the Rbel builder steps

Unresolved directive in tigerTestLibrary.adoc
include:::/RbelBuilderGlueCodeCommentsOnly.adoc[]

5.4.3. Usage examples

Listing 17. Tiger Rbel Builder steps example

Feature: Tiger - RbelBuilder

Scenario: Replace/change/add certain values/nodes in a rbel object

changing primitive value

Given TGR creates a new Rbel object 'someObjName' with content '{"address": {"street": "Friedrichstr 136", "city": "Berlin", "postalCode": "10115"}}'

Then TGR sets Rbel object 'someObjName' at '\$.address.street' to new value 'Hauptstrasse'

Then TGR asserts '!{rbelObject:serialize("someObjName")}' equals '{"address": {"street": "Hauptstrasse", "city": "Berlin", "postalCode": "10115"}}' of type JSON

adding primitive value

Given TGR creates a new Rbel object 'someAddress' with content '{"address": {"city": "Berlin", "postalCode": "10115"}}'

Then TGR sets Rbel object 'someAddress' at '\$.address.street' to new value 'Friedrichstr'

Then TGR asserts '!{rbelObject:serialize("someAddress")}' equals '{"address": {"street": "Friedrichstr", "city": "Berlin", "postalCode": "10115"}}' of type JSON

replacing object nodes

Given TGR creates a new Rbel object 'phoneNumbers' with content '{"phoneNumbers": [{"type": "home", "number": "030-1234567"}, {"type": "mobile", "number": "0176-123456788"}]}'

When TGR sets Rbel object 'phoneNumbers' at '\$.phoneNumbers.1' to new value '{"type": "work", "number": "0176-199999"}'

Then TGR asserts '!{rbelObject:serialize("phoneNumbers")}' equals '{"phoneNumbers": [{"type": "home", "number": "030-1234567"}, {"type": "work", "number": "0176-199999"}]}' of type JSON

adding new nodes to an array

When TGR extends Rbel object 'phoneNumbers' at path '\$.phoneNumbers' by a new entry '{"type": "mobile", "number": "0176-123456788"}'

Then TGR asserts '!{rbelObject:serialize("phoneNumbers")}' equals '{"phoneNumbers": [{"type": "home", "number": "030-1234567"}, {"type": "work", "number": "0176-199999"}, {"type": "mobile", "number": "0176-123456788"}]}' of type JSON

5.5. Using the HTTP client steps

The Tiger HTTP client steps are a set of Cucumber steps that enable you to perform simple HTTP requests, with bodies, default and custom headers.

Listing 18. Tiger response validation steps example

```
Feature: HTTP/HTTPS GlueCode Test feature
```

Background:

```
Given TGR clear recorded messages
```

Scenario: Simple Get Request

```
Given TGR clear recorded messages
```

```
When TGR send empty GET request to "http://httpbin/"
```

```
Then TGR find last request to path "./*"
```

```
And TGR assert "!{rbel:currentRequestAsString('$.method')}" matches "GET"
```

```
And TGR assert "!{rbel:currentRequestAsString('$.path')}" matches "\/?"
```

Scenario: Get Request to folder

```
When TGR send empty GET request to "http://httpbin/get"
```

```
Then TGR find last request to path "./*"
```

```
And TGR assert "!{rbel:currentRequestAsString('$.method')}" matches "GET"
```

```
And TGR assert "!{rbel:currentRequestAsString('$.path')}" matches "\/get\/?"
```

Scenario: PUT Request to folder

```
When TGR send empty PUT request to "http://httpbin/put"
```

```
Then TGR find last request to path "./*"
```

```
And TGR assert "!{rbel:currentRequestAsString('$.method')}" matches "PUT"
```

```
And TGR assert "!{rbel:currentRequestAsString('$.path')}" matches "\/put\/?"
```

Scenario: PUT Request with body to folder

```
When TGR send PUT request to "http://httpbin/put" with body "{ 'hello': 'world!' }"
```

```
Then TGR find last request to path "./*"
```

```
And TGR assert "!{rbel:currentRequestAsString('$.method')}" matches "PUT"
```

```
And TGR assert "!{rbel:currentRequestAsString('$.path')}" matches "\/put\/?"
```

```
And TGR assert "!{rbel:currentRequestAsString('$.body.hello')}" matches "world!"
```

Scenario: PUT Request with body from file to folder

```
When TGR send PUT request to "http://httpbin/put" with body "!{file('pom.xml')}"
```

```
Then TGR find last request to path "./*"
```

```
And TGR assert "!{rbel:currentRequestAsString('$.method')}" matches "PUT"
```

```
And TGR assert "!{rbel:currentRequestAsString('$.path')}" matches "\/put\/?"
```

```
And TGR assert
```

```
"!{rbel:currentRequestAsString('$.body.project.modelVersion.text')}" matches "4.0.0"
```

```
# application/octet-stream is used since no rewriting is done, so unknown/default  
MIME-type is assumed
```

```
And TGR assert "!{rbel:currentRequestAsString('$.header.Content-Type')}" matches  
"application/octet-stream.*"
```

Scenario: DELETE Request without body

```
When TGR send empty DELETE request to "http://httpbin/delete"
Then TGR find last request to path ".*"
And TGR assert "!{rbel:currentRequestAsString('$.method')}" matches "DELETE"
And TGR assert "!{rbel:currentRequestAsString('$.path')}" matches "\/delete\/?"
```

Scenario: Request with custom header

```
When TGR send empty GET request to "http://httpbin/get" with headers:
| foo    | bar   |
| schmoo | lar   |
Then TGR find last request to path ".*"
And TGR assert "!{rbel:currentRequestAsString('$.header.foo')}" matches "bar"
And TGR assert "!{rbel:currentRequestAsString('$.header.schmoo')}" matches "lar"
```

Scenario: Request with default header

```
Given TGR set default header "key" to "value"
When TGR send empty GET request to "http://httpbin/get"
Then TGR find last request to path ".*"
And TGR assert "!{rbel:currentRequestAsString('$.header.key')}" matches "value"
When TGR send POST request to "http://httpbin/post" with body "hello world"
Then TGR find last request to path ".*"
And TGR assert "!{rbel:currentRequestAsString('$.header.key')}" matches "value"
And TGR assert "!{rbel:currentRequestAsString('$.body')}" matches "hello world"
```

Scenario: Request with custom and default header, check headers

```
Given TGR set default header "key" to "value"
When TGR send empty GET request to "http://httpbin/get" with headers:
| foo | bar |
Then TGR find last request to path ".*"
And TGR assert "!{rbel:currentRequestAsString('$.header.foo')}" matches "bar"
And TGR assert "!{rbel:currentRequestAsString('$.header.key')}" matches "value"
```

Scenario: Get Request with custom and default header, check body, application type url encoded

```
Given TGR set local variable "configured_state_value" to "some weird ${value$}"
Given TGR set local variable "configured_param_name" to "my_cool_param"
When TGR send GET request to "http://httpbin/get" with:
| ${configured_param_name} | state           | redirect_uri   |
| client_id             | ${configured_state_value} | https://my.redirect |
Then TGR find last request to path ".*"
And TGR assert "!{rbel:currentRequestAsString('$.path.state.value')}" matches
"${configured_state_value}"
And TGR assert "!{rbel:currentRequestAsString('$.path.state')}" matches
"state=!{urlEncoded('some weird ${value$}')}"
And TGR assert "!{rbel:currentRequestAsString('$.path.my_cool_param')}" matches
"${configured_param_name}=client_id"
And TGR assert "!{rbel:currentRequestAsString('$.header.Content-Type')}" matches
"application/x-www-form-urlencoded.*"
```

Scenario: Post Request with custom and default header, check body, application type url encoded

```
Given TGR set local variable "configured_state_value" to "some weird ${value$}"
```

```

Given TGR set local variable "configured_param_name" to "my_cool_param"
When TGR send POST request to "http://httpbin/post" with:
| ${configured_param_name} | state           | redirect_uri      |
| client_id            | ${configured_state_value} | https://my.redirect |
Then TGR find last request to path ".*"
And TGR assert "!{rbel:currentRequestAsString('$.body.state')}" matches
"!{urlEncoded('some weird $value§')}"
And TGR assert "!{rbel:currentRequestAsString('$.body.my_cool_param')}" matches
"client_id"
And TGR assert "!{rbel:currentRequestAsString('$.header.Content-Type')}" matches
"application/x-www-form-urlencoded.*"
And TGR assert "!{rbel:currentRequestAsString('$.body.redirect_uri')}" matches
"!{urlEncoded('https://my.redirect')}"

```

Scenario: Request with custom and default header, check application type json

```

Given TGR set default header "Content-Type" to "application/json"
When TGR send POST request to "http://httpbin/post" with:
| ${configured_param_name} |
| client_id             |
Then TGR find last request to path ".*"
And TGR assert "!{rbel:currentRequestAsString('$.header.Content-Type')}" matches
"application/json"

```

Scenario Outline: JEXL Rbel Namespace Test

```

Given TGR send empty GET request to "http://httpbin/html"
Then TGR find first request to path "/html"
Then TGR current response with attribute "$.body.html.body.h1.text" matches
"!{rbel:currentResponseAsString('$.body.html.body.h1.text')}"

```

Examples: We use this data only for testing data variant display in workflow ui, there is no deeper sense in it

txt	txt2	txt3	txt4	txt5	
text2	21	31	41	51	
text2	22	32	42	52	

Scenario: Simple first test

```

Given TGR send empty GET request to "http://httpbin/html"
Then TGR find first request to path "/html"
Then TGR current response with attribute "$.body.html.body.h1.text" matches
"Herman Melville - Moby-Dick"

```

Scenario: Test Find Last Request

```

Given TGR send empty GET request to "http://httpbin/anything?foobar=1"
Then TGR send empty GET request to "http://httpbin/anything?foobar=2"
Then TGR find last request to path "/anything"
Then TGR current response with attribute "$.responseCode" matches "200"
Then TGR current response with attribute "$.body.url.content.foobar.value" matches
"2"

```

Scenario: Test find last request with parameters

```

Given TGR send empty GET request to "http://httpbin/anything?foobar=1"

```

```
Then TGR send empty GET request to "http://httpbin/anything?foobar=1&xyz=4"
Then TGR send empty GET request to "http://httpbin/anything?foobar=2"
Then TGR find last request to path "/anything" with "$.path.foobar.value" matching
"1"
Then TGR current response with attribute "$.body.url.content.xyz.value" matches
"4"
```

Scenario: Test find last request

```
Given TGR send empty GET request to "http://httpbin/anything?foobar=1"
Then TGR send empty GET request to "http://httpbin/anything?foobar=2"
Then TGR send empty GET request to "http://httpbin/anything?foobar=3"
Then TGR send empty GET request to "http://httpbin/status/404?other=param"
Then TGR find the last request
Then TGR current response with attribute "$.responseCode" matches "404"
Then TGR assert "!{rbel:currentRequestAsString('$.path.other.value')}" matches
"param"
```

Scenario: Get Request to folder and test param is url decoded when access via \$.path and ..value is url decoded

```
When TGR send empty GET request to "http://httpbin/get?foo=bar%20and%20schmar"
Then TGR find last request to path ".*"
And TGR assert "!{rbel:currentRequestAsString('$.path.foo.value')}" matches "bar
and schmar"
And TGR assert "!{rbel:currentRequestAsString('$.path.foo')}" matches
"foo=bar%20and%20schmar"
```

Scenario: Test deactivate followRedirects

```
When TGR disable HttpClient followRedirects configuration
And TGR send empty GET request to "http://httpbin/redirect-
to?url=!{urlEncoded('http://httpbin/status/200')}"
Then TGR find the last request
Then TGR current response with attribute "$.responseCode" matches "302"
And TGR current response with attribute "$.header.Location" matches
"http://httpbin/status/200"
When TGR reset HttpClient followRedirects configuration
And TGR send empty GET request to "http://httpbin/redirect-
to?url=!{urlEncoded('http://httpbin/status/200')}"
Then TGR find the last request
Then TGR current response with attribute "$.responseCode" matches "200"
```

Scenario: Test check filter POST request

```
Given TGR send POST request to "http://httpbin/post" with body "{ 'foobar': '4' }"
And TGR send empty GET request to "http://httpbin/anything?foobar=22"
And TGR filter requests based on method "POST"
Then TGR find last request to path ".*"
Then TGR current response with attribute "$.body.data.foobar" matches "4"
```

Scenario: Test check filter GET request

```
Given TGR send empty GET request to "http://httpbin/anything?foobar=22"
And TGR send POST request to "http://httpbin/post" with body "{ 'foobar': '4' }"
And TGR filter requests based on method "GET"
```

```
Then TGR find last request to path "./*"
Then TGR current response with attribute "$.body.args.foobar.0" matches "22"
```

Scenario: Test check filter method reset

```
Given TGR reset request method filter
# check resetting it works even if done twice
And TGR reset request method filter
And TGR send empty GET request to "http://httpbin/anything?foobar=22"
And TGR send POST request to "http://httpbin/post" with body "{ 'foobar': '4' }"
Given TGR reset request method filter
Then TGR find the last request
Then TGR current response with attribute "$.body.data.foobar" matches "4"
```

Scenario: Test check filter POST request

```
Given TGR send empty GET request to "http://httpbin/anything?foobar=66"
When TGR find last request with "$.path.foobar.value" matching "66"
Then TGR current response with attribute "$.body.args.foobar.0" matches "66"
```

5.5.1. Complete set of steps in HTTP client glue code

Unresolved directive in tigerTestLibrary.adoc - include:::/HttpGlueCodeCommentsOnly.adoc[]

5.5.2. XMLUnit Diff Builder

Using the validation steps `TGR current response at {string} matches as XML`: or
`TGR current response at {string} matches as XML and diff options {string}`: you are able to compare the content of any RbelPath node in the response.

The latter method even allows passing in the following options to the XMLUnit's DiffBuilder:

- "nocomment" for DiffBuilder::ignoreComments
- "txtnignoreempty" for DiffBuilder::ignoreElementContentWhitespace
- "txttrim" for DiffBuilder::ignoreWhitespace
- "txtnormalize" for DiffBuilder::normalizeWhitespace

Per default the comparison algorithm will ignore mismatches in namespace prefixes and URIs. Comparison is also performed on similarity and not equal content.

For more detailed explanation about the XMLUnit difference evaluator we refer to the [online documentation of the XMLUnit project](#).

5.5.3. JSONChecker

Using the validation step `TGR current response at {string} matches as JSON`: you are able to compare the content of any RbelPath node in the response to the doc string beneath the step, with the help of the JSONChecker comparison algorithm.

The purpose of JSONChecker class is to compare JSON structures, including checking for the integrity of the whole RbelPath node, as well as matching values for particular keys.

To make sure all the attributes in your JSON RbelPath structure are present, such features as \${json-unit.ignore}, \$NULL, optional attributes, regular expressions and lenient mode can come in handy.

\${json-unit.ignore} is a parameter which allows ignoring certain values in your RbelPath node while comparing, and the result of such comparison always returns true.

It also works when \${json-unit.ignore} is used in a JSON array or nested JSON object.

This parameter should be placed as a value of a key.

To ignore some attributes in the JSON structure, you can set a boolean value checkExtraAttributes as false.

In this case if you miss one attribute in your doc string, the comparison will still be equal to true.

To check whether the value for a particular key is null, you can either use null or parameter \$NULL at the place of the value.

Checking whether a nested key is null also works with JSONChecker.

Four underscores "__" before the JSON keys indicate that these keys are optional and will be checked for the value ONLY if the value exists in the test JSON RBelPath node.

Please note that checking whether a nested key is optional, is not yet possible with JsonChecker.

JSON Arrays are compared in lenient mode, meaning that the order of elements in JSON array doesn't matter.

Identifying missing keys is made easy in JSONChecker with the help of parameter \$REMOVE.

If you specify the name of the key and then \$REMOVE parameter as its value, the comparison will result in true, if the key is indeed missing and false, if the key is present.

It is worth noting that even if the value of the key is null, the key doesn't count as missing.

Last but not least, regular expressions, which can be used for matching the whole JSON element, as well as particular values.

It will be first checked, whether the expected value is equal to the actual one, and only afterwards, if the actual value matches a regular expression.

It should also be noted, that although JSONChecker can match multilevel JSON objects at a high level, it is not yet possible to access nested attributes out of the box.

We are working on it :)

Listing 19. Simple adapted example from the IDP test suite

```
{  
    "alg": "dir",  
    "enc": "A256GCM",  
    "cty": "$NULL",  
    "exp": "[\\d]*",  
    "___kid": ".*",  
    "dummyentry": "${json-unit.ignore}",  
    "dummyarray": [ "entry1", "entry2" ],  
    "dummyarray2": "${json-unit.ignore}"  
}
```

The example above shows three main features of the JSONChecker.

- Value specified as \$NULL, meaning this value of this key is equal to null.
- Usage of regular expression (e.g. ".*" and "[\d]*") to match values.
- Usage of "__" preceding a json key: This indicates that the entry is optional but if it exists it must match the given value.
- if a value is specified as "\${json-unit.ignore}", there is no check performed at all.
This applies also to objects and arrays as seen in the dummyarray2 entry.
- if we match key dummyEntry2 to the value of \$REMOVE, it will return true, because this key does not exist.

JSON Schema Validation

Using the validation step `TGR current response at {string} matches as JSON_SCHEMA`: you are able to assert that the content of any RbelPath node complies with the JSON Schema given in the doc string beneath the step.

We use the JSON Schema version 2020-12 for validation with the additional feature that we can have placeholders in the schema that will be replaced with values from the Tiger Global Configuration.

For example:

```
TGR current response at "$.body" matches as JSON_SCHEMA:
```

```
"""
{
  "type": "object",
  "properties": {
    "name": {
      "type": "string"
    },
    "age": {
      "type": "integer",
      "minimum": 0
    },
    "email": {
      "type": "string",
      "format": "email"
    },
    "address": {
      "type": "string",
      "const": "${tiger.value.from.config}"
    }
  },
  "required": [
    "name",
    "age"
  ]
}
```

```
}
```

```
"""
```

would validate a message with a JSON body like:

```
{
  "name": "hello",
  "age": 5,
  "email": "hello@example.com",
  "address": "Friedrichstraße"
}
```

assuming that the value of `#{tiger.value.from.config}` set in the Tiger Global Configuration is `Friedrichstraße`.

For full details how to specify a JSON Schema, refer to the external resource: [Understanding JSON Schema](#).

In [this list](#) you can find a list of online validators which you can use to test your schema.

Remember to test under JSON Schema version 2020-12.

5.6. Exemplaric scenario Konnektorfarm EAU validation

The EAU Konnektorfarm scenario is a scenario where customers can use their Primärsystem to test signing and verifying documents via a set of Konnektoren and that this works interoperable.

For this purpose a phalanx of local Tiger Proxies is set up as reverse proxies for each Konnektor being hosted at the gematik location.

Any message that is forwarded by any of these proxies is forwarded to an aggregating Tiger Proxy which in turn forwards all the received messages to the local Tiger Proxy for assertion via the validation test suite.

Tiger EAU Konnektorfarm test

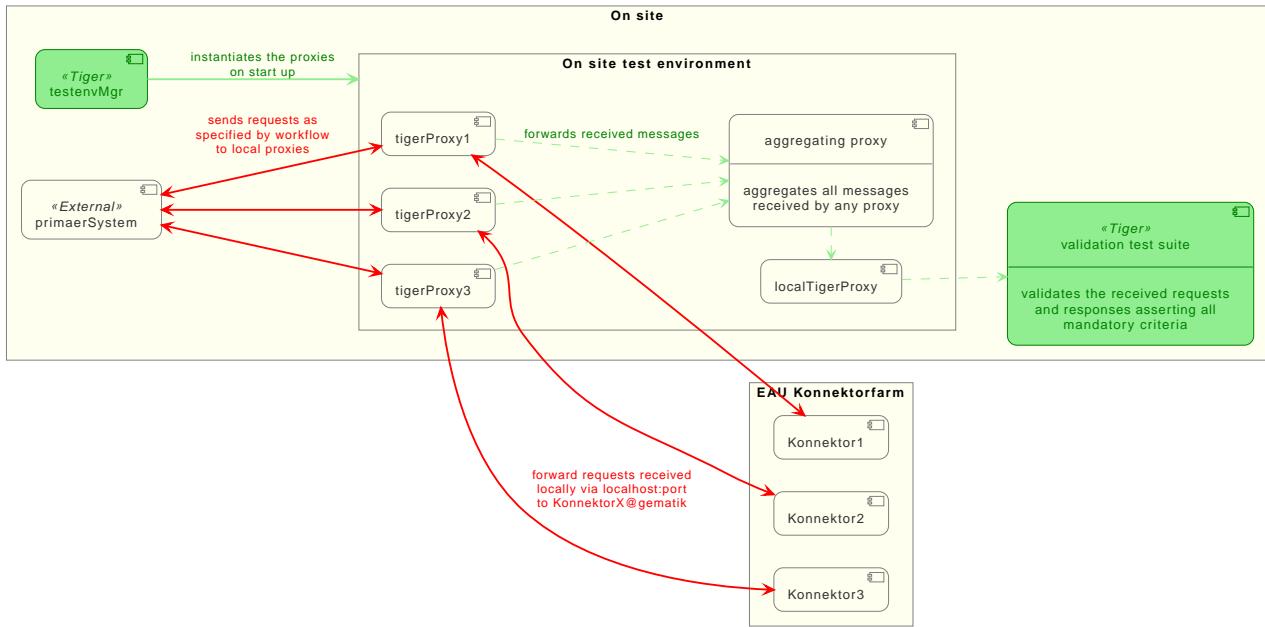


Figure 10. Tiger EAU Konnektorfarm test environment

So after starting the validation test suite (and the test environment), the customer / Primärsystem manufacturer must perform the specified workflow.

The test suite meanwhile will wait for a given order of requests/responses matching specified criteria to appear.

If all is well, at the end the test report JSON files will be packed into a zip archive and can be uploaded to the Titus platform for further certification steps.

Tiger EAU Konnektorfarm process

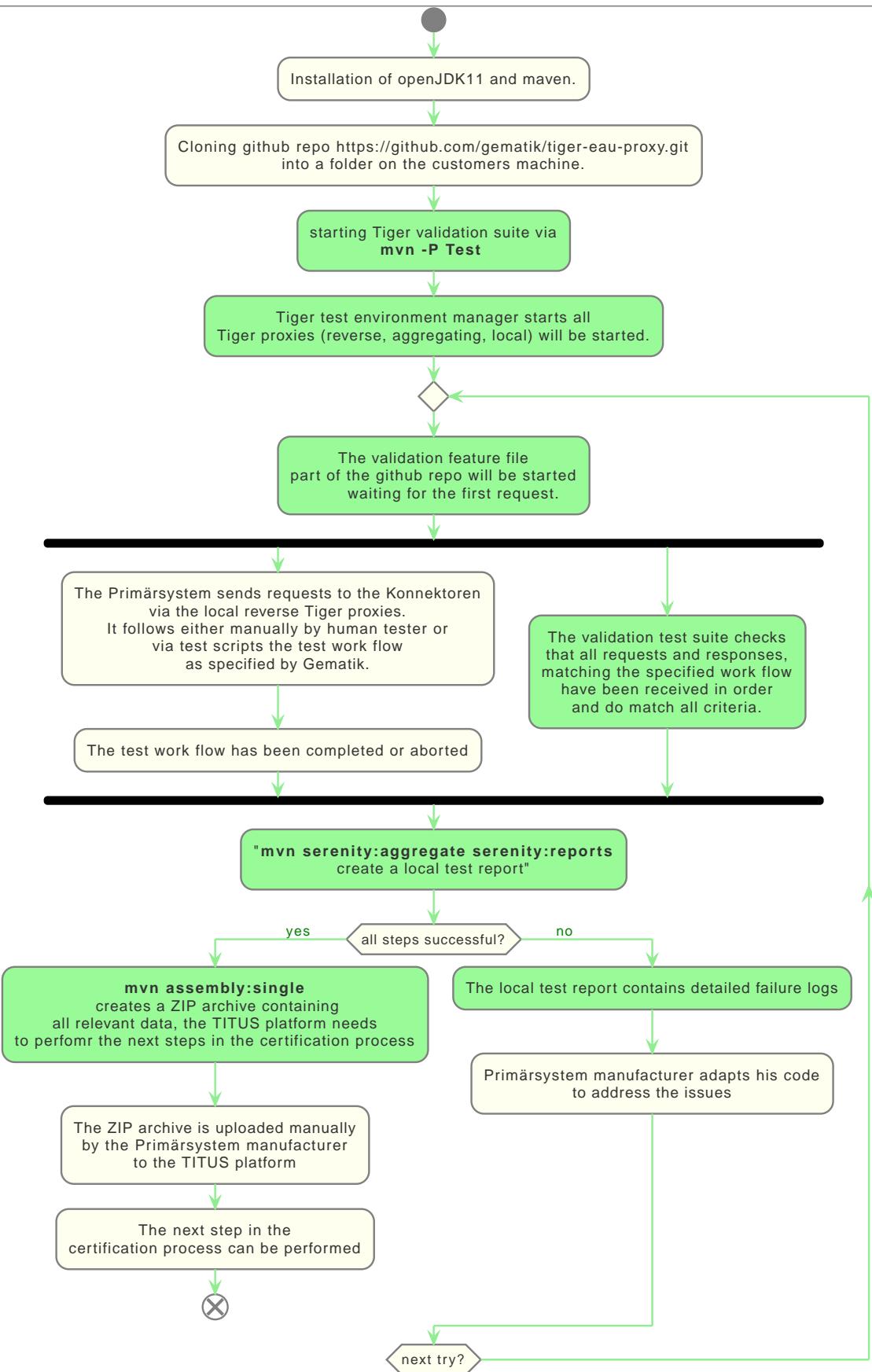


Figure 11. Tiger EAU Konnektorfarm process

5.7. Using Tiger test lib helper classes

If you don't want to use the Tiger test framework but only pick a few helper classes the following classes might be of interest to you:



All classes listed here are part of the tiger-common module

For BDD feature files Tiger provides some helper steps too.

5.7.1. Complete set of helper steps in Tiger glue code

Unresolved directive in tigerTestLibrary.adoc - include:::/TigerGlueCommentsOnly.adoc[]

5.7.2. Banner

If you want to use large ASCII art style log banners you may find this class very helpful.

Supports ANSI coloring and a set of different fonts.

Furthermore, all banner messages are displayed and highlighted in the Workflow UI For more details please check the code and its usages in the Tiger test framework.

5.7.3. TigerSerializationUtil

This class supports you in converting String representation of YAML and JSON data to an Java JSONObject or extract that or other loaded data to Java Maps.

If you are planning to implement test data management or configuration sets, we propose to use the TigerGlobalConfiguration class described [in detail here](#).

5.7.4. TigerPkiIdentityLoader, TigerPkiIdentity

The loader class allows to easily instantiate PKI identities from given files.

For more details on the format and the supported file types please check [this section in the test environment chapter](#).

5.7.5. Performing REST calls with Tiger

Tiger is closely integrated with SerenityBDD, which in turn has integrated the RestAssured library, so if you use the `SerenityRest` helper class, you will get detailed information about each call inside the test report.

The Tiger test library configuration also provides a flag to add curl command details to each of these calls, so that you can easily reproduce the REST call manually in case of test failure analysis.

For more information about REST testing in Tiger/SerenityBDD please check these two documents:

- [Serenity](#)
- [Serenity Screenplay REST](#)

5.8. Synchronizing BDD scenarios with Polarion test cases (Gematrik only)

Within gematik we maintain test cases via feature files being committed to git repositories. To keep traceability to the requirements maintained in Polarion we have a Tiger sub project that synchronizes test cases in Polarion with the scenarios in our feature files. It is a one way synchronisation, where the master are the feature files.

To use this feature the scenarios need a minimal set of mandatory annotations:

- **@TCID:xxx** - a unique test case identifier, where 'xxx' matches the value of the custom field "cfInternalId" in Polarion
- **@PRODUKT:p,p,p** - reference to the custom field "cfProductType". You add this annotation above each feature, not each scenario. 'p' is a product, one is mandatory but it can be a list.

And following optional annotations exist:

- **@AFO-ID:xxx** - a link to a defined requirement (Anforderung) in Polarion, where 'xxx' matches the custom field "cfAfoId"
- **@AF-ID:xxx** - a link to a defined requirement (Anwendungsfall) in Polarion, where 'xxx' matches the custom field "cfAfId"
- **@AK-ID:xxx** - a link to a defined requirement (Akzeptanzkriterium) in Polarion, where 'xxx' matches the work item id
- **@PRIO:n** - priority number (1-4), default is '1'
- **@MODUS:xxx** - describes the way of testing, default is 'automatisch'
- **@STATUS:xxx** - describes the status of the test, default is 'implementiert'
- **@TESTFALL:n** - describes if the test case is a negative testcase or positive one, default is 'positiv'
- **@TESTSTUFE:n** - describes the test type, default is '3' (which is E2E-Test)
- **@DESCRIPTION** - if your test case has a description, and you use this annotation, the description will be parsed.
If not, the description stays empty and won't overwrite the one already existing in Polarion

If a scenario is identified that has no test case with a matching TCID, it will be created automatically in the sync run.

Background blocks will be merged to each scenario before exporting its steps to Polarion.

For more details on how to perform the synchronisation, all choices for the annotations and how to upload generated test run reports to Polarion and Aurora, please check the README.md in the PolarionToolbox project on the Gematrik GitLab.

5.9. JUnit test report when using Scenario Outlines

When using Scenario Outlines in your feature files, the JUnit test report will show each example

that has run in the scenario outline as a single test case.

To create a name for each example that is referencing the scenario outline, the system property `cucumber.junit-platform.naming-strategy.short.example-name` is set to the value `pickle` in case it is not set by the environment.

Chapter 6. Tiger Configuration

Configuration is an integral part of testing.

To make this task easier for you and to make configuration the various parts of the system as easy as possible Tiger has a central configuration store: `TigerGlobalConfiguration`.

It combines properties from multiple source and feeds into various parts of the system.

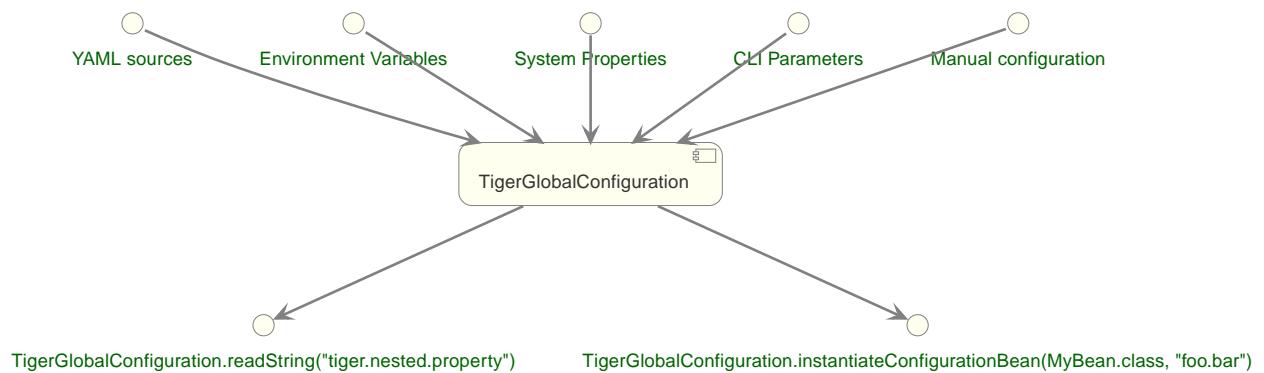


Figure 12. The `TigerGlobalConfiguration` with inlets and outlets

This allows a vastly simplified retrieval and configuration of nearly all aspects of the system. It is therefore recommended reusing this system for your own testsuite as well.

6.1. Inlets

The following inlets are considered in the `TigerGlobalConfiguration` (ordered from most to least important, meaning if a property occurs in multiple sources the one at the top is considered first):

- Exports from `ScopedExecutor`
- Thread-local exports
- Exports done in glue code
- Exports during runtime (`TigerGlobalConfiguration.putValue()`)
- Command-line properties
- System-Properties (`System.setProperty`)
- Environment-Variables (`export "FOO_BAR" = 42`)
- Full-text YAML file (value of `tiger.yaml` configuration key)
- Additional YAML-Files (`additionalConfigurationFiles:`)
- Host YAML-File (`tiger-<hostname>.yaml`)
- Main YAML-File (`tiger.yaml`)
- Interne Defaults

6.2. Key-translation

To easily convert between the multiple sources the TigerGlobalConfiguration offers key-translation:

`tiger.foo.bar` is equal to `TIGER_FOO_BAR` is equal to `tIgER.foo.BaR`

- When the key consists only of letters and underscores then the underscores are converted to points.
- Names are compared without considering the case.
- Keys that contain '{', '}' or '|' are forbidden.
To allow a clean startup on systems that have values like this configured the given characters are replaced by '_.'

6.3. Thread-based configuration

To enable execution of multiple tests simultaneously some data has to be stored in a thread-based manner (the first step could for example store the result of a request in a variable, the second step could read it from that variable).

To enable this simply reference the Thread-context when storing a variable:

```
TigerGlobalConfiguration.putValue("foo.value", "bar", ConfigurationFileType  
.THREAD_CONTEXT);
```

When retrieving the variable you could simply ask for `foo.value`: Only when you are in the thread that stored this variable you will find it again.

6.4. Placeholders

The TigerGlobalConfiguration supports the use of placeholders.

Say for example you have a test-environment with two servers, "A" and "B".

For the server "A" you have two choices: Either a real URL in the internet or a locally booted server.

The user can choose which to activate by setting "active" of the server to use.

The server "B" should now use the activated server, without having to set it manually while booting.

You could achieve this by exporting the URL (`servers.myServer.exports`) and referencing it in an argument which is passed into server "B" (`serverAUrl=${serverA.url}`).

The first part here before the equal is the name of the environment variable passed into server "B" while booting, the second part behind the equal is the name of the property. compare this to the exports in the two serverA-options):

Listing 20. Configuring using placeholders and exports

```
servers:  
  serverAInternet:  
    active: true
```

```

type: externalUrl
source:
  - https://my.real.server/api
exports:
# The string SERVERA_URL is split internally into SERVERA and URL, which are then
considered
# as lowercase keys
  - SERVERA_URL=https://my.real.server/api
serverALocal:
  active: false
  type: externalUrl
  source:
    - https://localhost:8080/api
  exports:
    - SERVERA_URL=https://localhost:8080/api
serverB:
  type: externalJar
  source:
    - http://nexus/download/server.jar
  healthcheckUrl: http://127.0.0.1:19307
  externalJarOptions:
    arguments:
# The second part is the placeholder which will be resolved using the internal value
store.
# The string "serverA.url" is split into "serverA" and "url", again considered as
lowercase,
# which then matches to "SERVERA_URL",
  - --serverAUrl=${serverA.url}

```



Placeholders which can not be resolved will not lead to errors but rather they will simply not be replaced.

6.5. RbelPath-style retrieval

The placeholders also support RbelPath-style expressions to allow for more flexible, dynamic retrieval of properties.

Consider for example the following YAML:

```

myMap:
  anotherLevel:
    key1:
      value: foobar
      target: schmoo
    key2:
      value: xmas
      target: blublub
  hidden:
    treasure:

```

To retrieve values from this map you can use the following expressions:

- `${myMap.anotherLevel.[?(@.value=='foobar')].target}` will resolve to "schmoo", retrieving the node `myMap.anotherLevel.key.target`.
- The same value can be retrieved via `${..[?(@.target=='schmoo')].target}`. This expression uses the recursive descent mechanic of RbelPath.
- `${..buried}` will resolve to "deep", retrieving the node `myMap.hidden.treasure.buried`.

6.6. Fallback values

Sometimes a default value is desired when a given key is not set.

To define such a value, just use the pipe (|) after the key, like so:

```
 ${foo.bar|orThisValue}
```

This will first test for the presence of "foo.bar" as a configuration key.

If that key is not found, the fallback value "orThisValue" will be used.

6.7. Localized configuration

It is possible to set a local variable in the TigerGlobalConfiguration which will only be active for the duration of the test case execution.

This can be achieved with the glue code step:

```
TGR setze lokale Variable {tigerResolvedString} auf {tigerResolvedString}
TGR set local variable {tigerResolvedString} to {tigerResolvedString}
```

The variables will be removed from the TigerGlobalConfiguration after the test case execution.

Bear in mind that this does not work with threading: The values are added to the global store and are removed automatically, but will still be visible by any parallel thread that attempts to read the TigerGlobalConfiguration.

It is also possible to set a variable that is local to the feature file where it is defined.

With the glue code step:

```
TGR setze lokale Feature Variable {tigerResolvedString} auf {tigerResolvedString}
TGR set local feature variable {tigerResolvedString} to {tigerResolvedString}
```

a variable can be set that will be removed from the TigerGlobalConfiguration when the execution of the feature file is finished.

6.8. Examples

Some examples to clarify:

6.8.1. Example 1

Say you have an environment configured in your testenv.yaml.

You want the Tiger Proxy to forward traffic on one route to your backend-server.

This will normally be a local server, but on the build-server you want to address another host.

You can simply set an environment variable to do the job for you.

Below are the relevant snippets:

Listing 21. tiger.yaml with the Tiger Proxy routing everything to the local server

```
tigerProxy:  
  proxyRoutes:  
    - from: /  
      to: http://127.0.0.1:8080
```

In the buildserver you can now simply overwrite the "to"-part of this route like so:

```
export TIGERPROXY_PROXYROUTES_0_TO = "http://real.server"
```

6.8.2. Example 2

In the above example let's say you only want to customize the port.

This can be done by using placeholders:

Listing 22. tiger.yaml with the Tiger Proxy routing everything to the local server

```
tigerProxy:  
  proxyRoutes:  
    - from: /  
      to: http://127.0.0.1:${backend.server.port}
```

This time we don't overwrite the complete to-url but only the port like so:

```
export BACKEND_SERVER_PORT = "8080"
```

6.8.3. Example 3

Now we want to assert that the reply coming from the server has the correct backend-url in the XML that is returned to the sender.

To do this we have to reference the configured URL from above, since the value could be different on every execution.

We can solve this using placeholders:

Listing 23. The testsuite

```
TGR current response with attribute "$.body.ReplyStructure.Header.Sender.url"  
matches "http://127.0.0.1:${backend.server.port}"
```

The glue-code in Tiger automatically resolves the placeholders.

6.9. Pre-Defined values

Tiger adds some pre-defined values to make your life easier configuring the environment. Currently, these are:

- `free.port.0 - free.port.255`: Free ports that are randomly determined at startup but stay fixed during the execution.
This enables side effect free execution of the testsuite.

6.10. Inline JEXL

In addition to the `${foo.bar}` syntax allowing the retrieval of configuration values there exists the `!{'foo' != 'bar'}` syntax allowing the execution of JEXL expressions.

The JEXL-syntax is described in more depth here: <https://commons.apache.org/proper/commons-jexl/reference/syntax.html>

To give you more power and flexibility when creating inline-JEXL-expression you can access several namespaces from inside the JEXL expression.

You will find two predefined namespaces and also the ability to add your own, allowing further customization.

6.10.1. The default namespace

The default-namespace of the inline JEXL-expression carries the following functions:

- `file(<filename>)` loads the given file and returns it as a UTF-8 parsed string.
- `sha256` returns the HEX-encoded SHA256-value of the given string.
- `sha256Base64` returns the Base64-encoded SHA256-value of the given string.
- `sha512` returns the HEX-encoded SHA512-value of the given string.
- `sha512Base64` returns the Base64-encoded SHA512-value of the given string.
- `md5` returns the HEX-encoded MD5-value of the given string.
- `md5Base64` returns the Base64-encoded MD5-value of the given string.
- `base64Encode` returns the Base64-Encoding of the given string (non-url safe).
- `base64UrlEncode` returns the Base64-URL-Encoding of the given string.
- `base64Decode` decodes the given Base64-String (URL and non-url) and converts it into a UTF-8 string.

An example of a function-invocation in the default namespace:

```
!{file('src/test/resources/testMessage.json')}
```

This will load the given file and replace any placeholders found in it.

6.10.2. The rbel namespace

To give you direct access to the messages sent please use the rbel-namespace:

- `currentResponse` returns the current response, optionally filtered by a given Rbel-path
- `currentResponseAsString` returns the string-representation of the current response, optionally filtered by a given Rbel-path
- `currentRequest` returns the current request, optionally filtered by a given Rbel-path
- `currentRequestAsString` returns the string-representation of the current request, optionally filtered by a given Rbel-path

This can be done like so

```
!{rbel:currentResponseAsString('$.body.html.head.link.href')}
```

This will immediately return the `href`-attribute of the link in question as a string.

6.10.3. Adding custom namespaces

You can easily register additional namespaces by calling `TigerJexlExecutor.registerAdditionalNamespace(<namespace-prefix>, <namespace class or object>)`.

6.11. Configuration Editor

The configuration editor allows to view and edit the tiger configuration during a test run.

The editor is part of the [Workflow UI](#) and can be opened by clicking the gears icon in the sidebar ([Figure 13](#)).

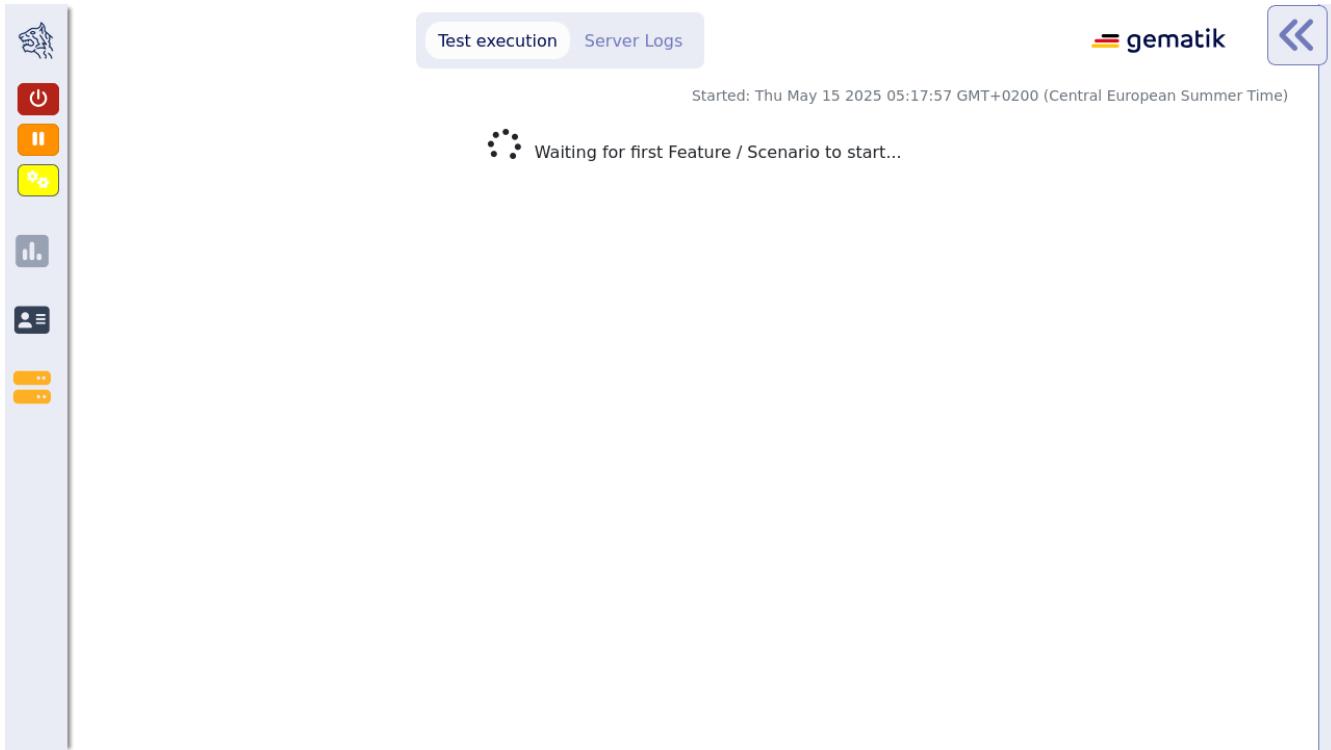


Figure 13. Open the configuration editor by clicking the gears icon in the sidebar.

The configuration editor displays a table where you can view the current configuration properties loaded in the Tiger global configuration (Figure 14).

This includes properties from all [inlet sources](#).

If a property is defined multiple times in different sources, only the one with higher importance is displayed.

Tiger Global Configuration Editor				
	Source	Key	Value	Action
☰	tgr.testenv.cfg.check.mode		myEnv	
☰	tgr.testenv.cfg.delete.mode		deleteEnv	
☰	tgr.testenv.cfg.edit.mode		editEnv	
☰	tgr.testenv.cfg.multiline.check.mode		Lorem ipsum dolor sit amet, consetetur sadipscing elit...	
📄	tgrTestAdditionalYaml.cfgEditor.checkKey		checkAdditionalValue	
📄	tgrTestAdditionalYaml.cfgEditor.deleteKey		deleteAdditionalValue	
📄	tgrTestAdditionalYaml.cfgEditor.editKey		editAdditionalValue	
⚙️	tgrTestPropCfgCheckMode		myProp	
⚙️	tgrTestPropCfgDeleteMode		deleteProp	
⚙️	tarTestPropCfaEditMode		

Figure 14. The Tiger global configuration editor

The editor allows sorting and filtering each column so that you can easily find a specific property (Figure 15).

Given that the Tiger global configuration includes many environment variables and system properties which are not directly relevant to Tiger, the filtering functionality proves to be especially useful.

Tiger Global Configuration Editor

The screenshot shows the 'Tiger Global Configuration Editor' interface. At the top, there are buttons for 'Clear filters', 'Export', 'Import', and 'Refresh'. Below this is a table with three columns: 'Source', 'Key', and 'Value'. A filter dialog is open over the 'Key' column, containing dropdown menus for 'Contains' and 'tgr', and radio buttons for 'AND' and 'OR'. The table rows show various configuration keys like 'tgr.testenv.cfg.check.mode', 'tgr.testenv.cfg.delete.mode', etc., with their corresponding values and edit/delete icons.

Source	Key	Action
	tgr.testenv.cfg.check.mode	<input type="button" value="Edit"/> <input type="button" value="Delete"/>
	tgr.testenv.cfg.delete.mode	<input type="button" value="Edit"/> <input type="button" value="Delete"/>
	tgr.testenv.cfg.edit.mode	<input type="button" value="Edit"/> <input type="button" value="Delete"/>
	tgr.testenv.cfg.multiline.check.mode	<input type="button" value="Edit"/> <input type="button" value="Delete"/>
File	tgrTestAdditionalYaml.cfgEditor.checkKey	<input type="button" value="Edit"/> <input type="button" value="Delete"/>
File	tgrTestAdditionalYaml.cfgEditor.deleteKey	<input type="button" value="Edit"/> <input type="button" value="Delete"/>
File	tgrTestAdditionalYaml.cfgEditor.editKey	<input type="button" value="Edit"/> <input type="button" value="Delete"/>
Prop	tgrTestPropCfgCheckMode	<input type="button" value="Edit"/> <input type="button" value="Delete"/>
Prop	tgrTestPropCfgDeleteMode	<input type="button" value="Edit"/> <input type="button" value="Delete"/>
Prop	tarTestPropCfaEditMode	<input type="button" value="Edit"/> <input type="button" value="Delete"/>

Figure 15. Example of filtering the column key by the text 'tgr'

The values of existing configuration properties can be edited by double-clicking the value cells. This opens an input field where you can input a new value (Figure 16).

Tiger Global Configuration Editor

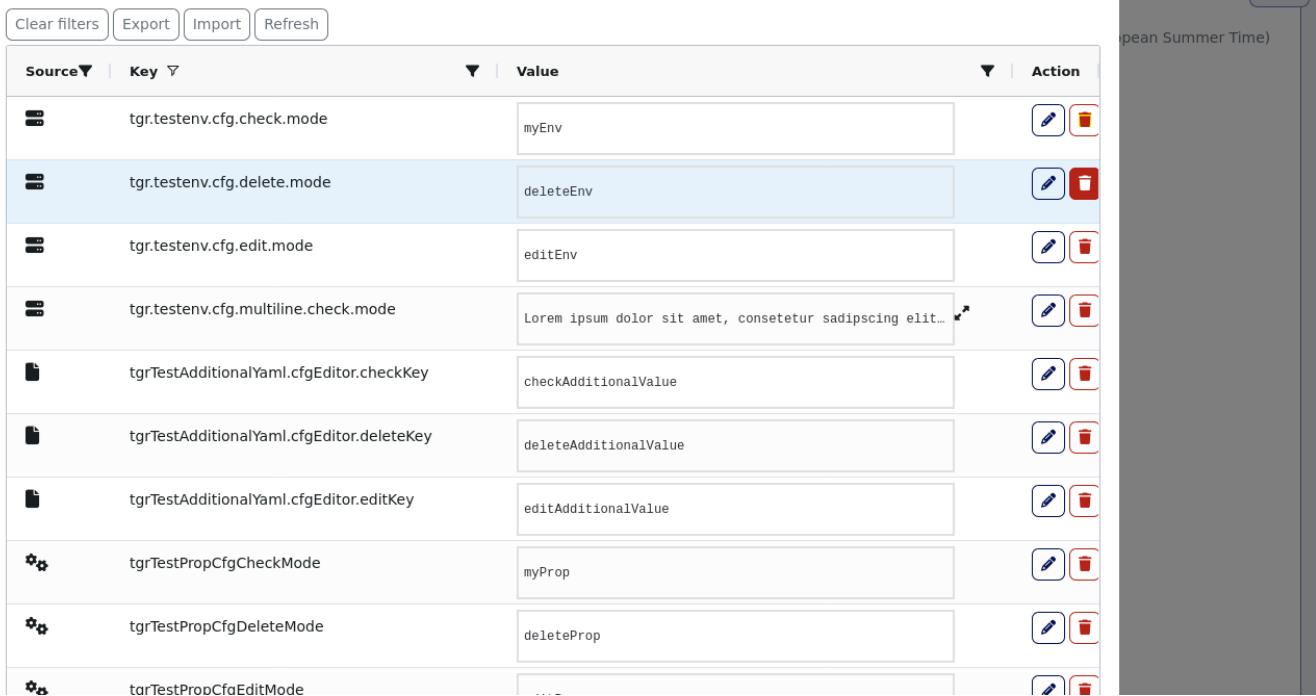
This screenshot shows the same configuration editor interface as Figure 15, but with a different state. The 'myEnv' value cell for the 'tgr.testenv.cfg.check.mode' row is now highlighted with a yellow background, indicating it is selected. A modal dialog box is overlaid on the cell, containing a 'Save' button (which is dark blue) and a 'Cancel' button. The rest of the table and interface elements are visible in the background.

Source	Key	Action
	tgr.testenv.cfg.check.mode	<input type="button" value="Edit"/> <input type="button" value="Delete"/>
	tgr.testenv.cfg.delete.mode	<input type="button" value="Edit"/> <input type="button" value="Delete"/>
	tgr.testenv.cfg.edit.mode	<input type="button" value="Edit"/> <input type="button" value="Delete"/>
	tgr.testenv.cfg.multiline.check.mode	<input type="button" value="Edit"/> <input type="button" value="Delete"/>
File	tgrTestAdditionalYaml.cfgEditor.checkKey	<input type="button" value="Edit"/> <input type="button" value="Delete"/>
File	tgrTestAdditionalYaml.cfgEditor.deleteKey	<input type="button" value="Edit"/> <input type="button" value="Delete"/>
File	tgrTestAdditionalYaml.cfgEditor.editKey	<input type="button" value="Edit"/> <input type="button" value="Delete"/>
Prop	tgrTestPropCfgCheckMode	<input type="button" value="Edit"/> <input type="button" value="Delete"/>
Prop	tgrTestPropCfgDeleteMode	<input type="button" value="Edit"/> <input type="button" value="Delete"/>
Prop	tarTestPropCfaEditMode	<input type="button" value="Edit"/> <input type="button" value="Delete"/>

Figure 16. Double clicking a value cell opens the cell editor.

Additionally, you can remove existing configuration properties by clicking the delete button (Figure 17)

Tiger Global Configuration Editor



Source	Key	Value	Action
	tgr.testenv.cfg.check.mode	myEnv	
	tgr.testenv.cfg.delete.mode	deleteEnv	
	tgr.testenv.cfg.edit.mode	editEnv	
	tgr.testenv.cfg.multiline.check.mode	 Lorem ipsum dolor sit amet, consetetur sadipscing elit... 	
	tgrTestAdditionalYaml.cfgEditor.checkKey	checkAdditionalValue	
	tgrTestAdditionalYaml.cfgEditor.deleteKey	deleteAdditionalValue	
	tgrTestAdditionalYaml.cfgEditor.editKey	editAdditionalValue	
	tgrTestPropCfgCheckMode	myProp	
	tgrTestPropCfgDeleteMode	deleteProp	
	tarTestPropCfaEditMode	

Figure 17. Clicking the delete button removes the property from the Tiger global configuration.



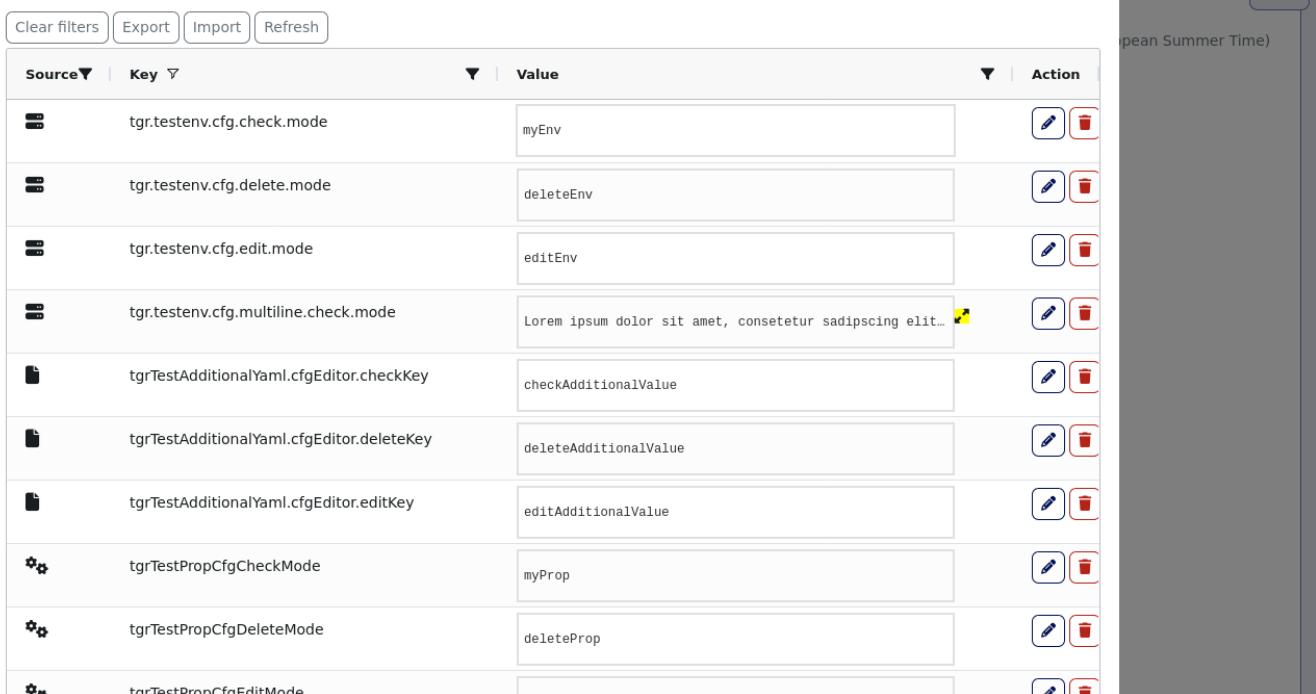
Editing or removing configuration properties will not affect already ran tests.

If you want to use edited properties in a specific test, then you should pause the test before editing the configuration.

In [Workflow UI](#) you can see how to use custom steps to pause the test suite.

Some variables in the table have multiline values, causing the text to appear truncated initially. These cells are equipped with an expand icon (Figure 18), indicating the availability of additional content.

Tiger Global Configuration Editor

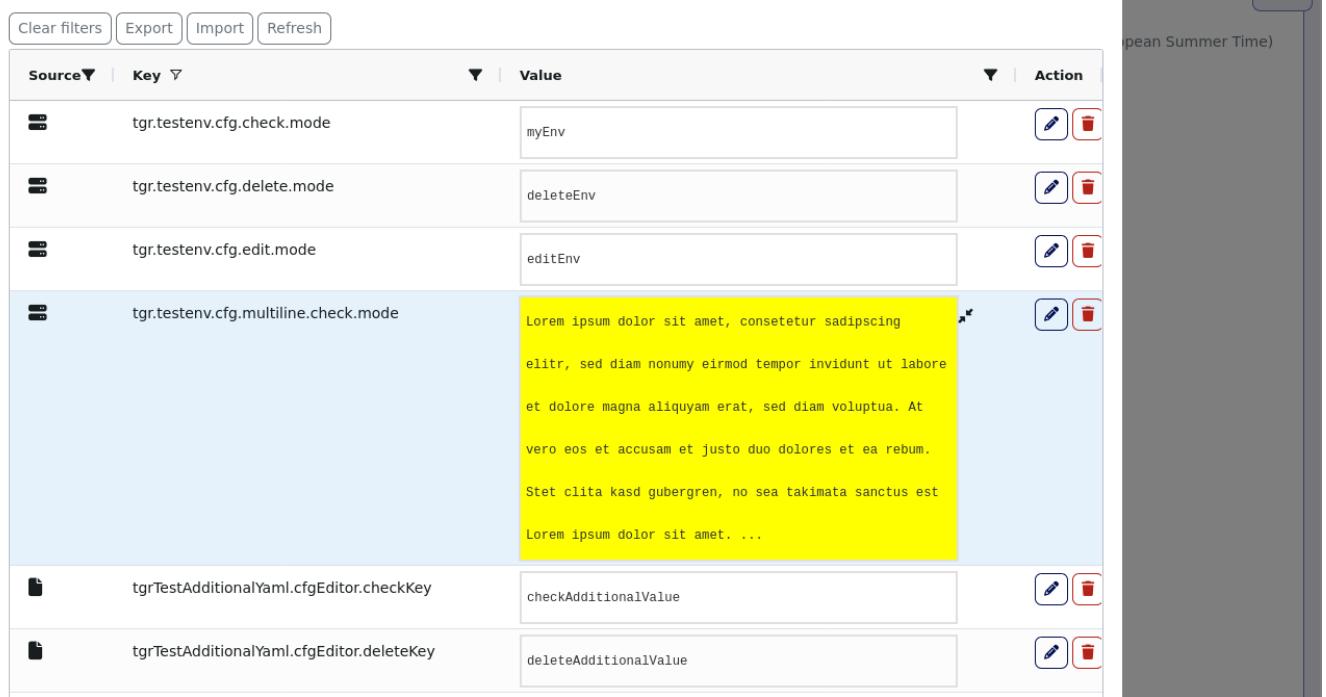


Source	Key	Value	Action
	tgr.testenv.cfg.check.mode	myEnv	
	tgr.testenv.cfg.delete.mode	deleteEnv	
	tgr.testenv.cfg.edit.mode	editEnv	
	tgr.testenv.cfg.multiline.check.mode	 Lorem ipsum dolor sit amet, consetetur sadipscing elit... 	
	tgrTestAdditionalYaml.cfgEditor.checkKey	checkAdditionalValue	
	tgrTestAdditionalYaml.cfgEditor.deleteKey	deleteAdditionalValue	
	tgrTestAdditionalYaml.cfgEditor.editKey	editAdditionalValue	
	tgrTestPropCfgCheckMode	myProp	
	tgrTestPropCfgDeleteMode	deleteProp	
	tarTestPropCfaEditMode	

Figure 18. Clicking on the expand icon reveals the full multiline content.

Clicking the expand icon uncover the complete multiline content, ensuring it is fully visible within the cell.

Tiger Global Configuration Editor



Source	Key	Value	Action
	tgr:testenv.cfg.check.mode	myEnv	edit delete
	tgr:testenv.cfg.delete.mode	deleteEnv	edit delete
	tgr:testenv.cfg.edit.mode	editEnv	edit delete
	tgr:testenv.cfg.multiline.check.mode	Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est lorem ipsum dolor sit amet.	edit delete
	tgrTestAdditionalYaml.cfgEditor.checkKey	checkAdditionalValue	edit delete
	tgrTestAdditionalYaml.cfgEditor.deleteKey	deleteAdditionalValue	edit delete

Figure 19. Click the expand icon to view the full multiline content.

To hide the multiline content and return to a truncated view, simply click on the collapse icon. This action collapses the multiline content, returning the text to its truncated state.

Chapter 7. Tiger User interfaces

7.1. Workflow UI

The Workflow UI is a feature for a better user experience during the test run of feature file(s).

If activated via the [tiger.yaml configuration file](#), the Workflow UI will be opened in the current browser window during the test run and shows the status and logs of the servers as well as the results and request calls of the scenarios and feature files during the test run.

If no browser is open at the time a new instance will be launched.

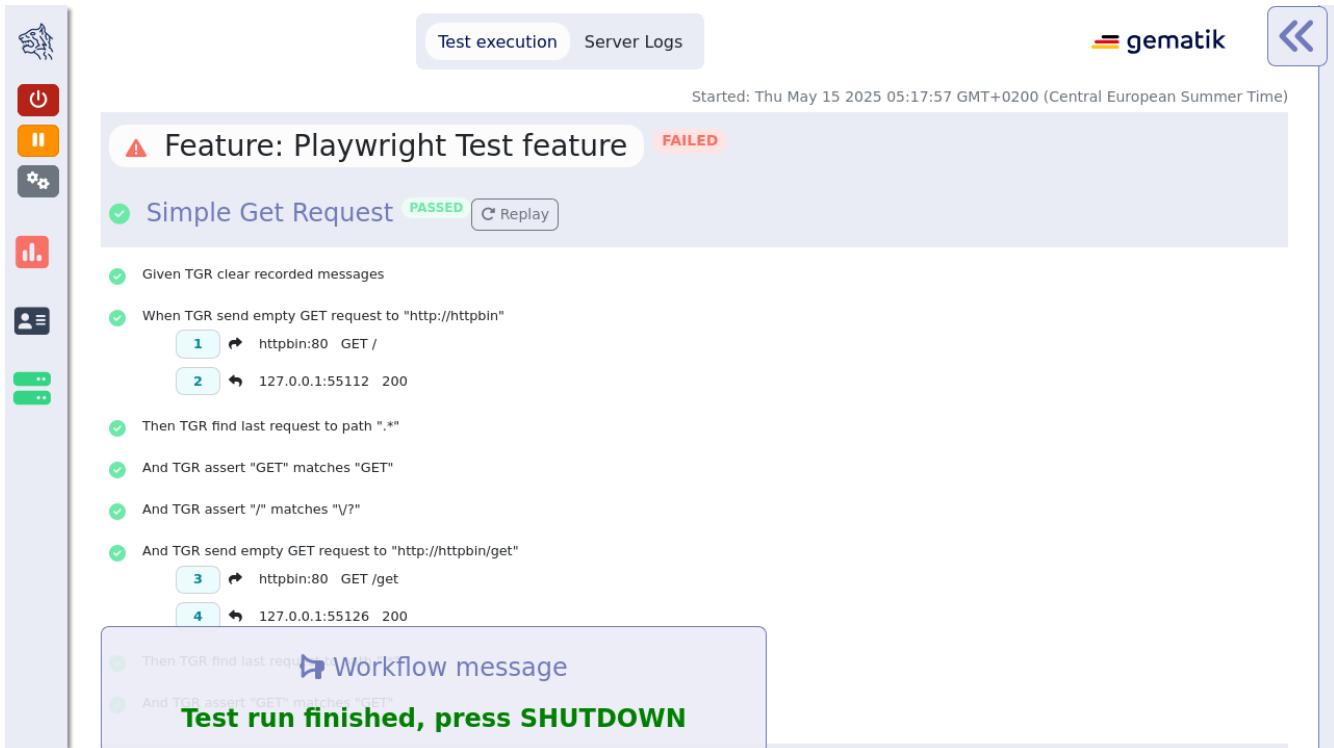


Figure 20. Workflow UI

The image above shows the initial startup of the Workflow UI.

The Workflow UI is divided into three sections: the [status bar](#), the [main window](#) with [test execution](#) and [server logs](#) and the [Rbel log details](#) (a slimmed down version of the [WebUI](#)).

7.1.1. Status Bar

The section on the left is called status bar as shown in the picture below.

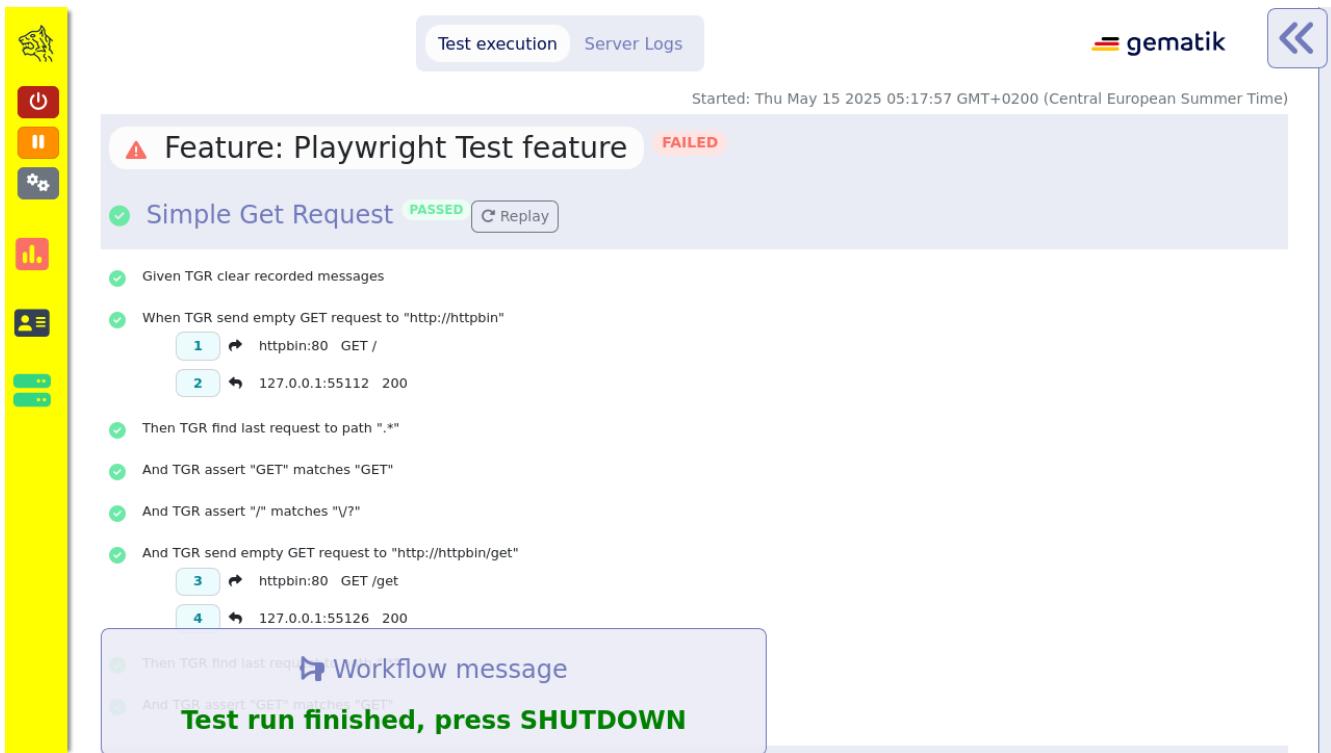


Figure 21. the status bar is situated on the left

When the user clicks on the tigers head on the top left the status bar slides open as shown below.

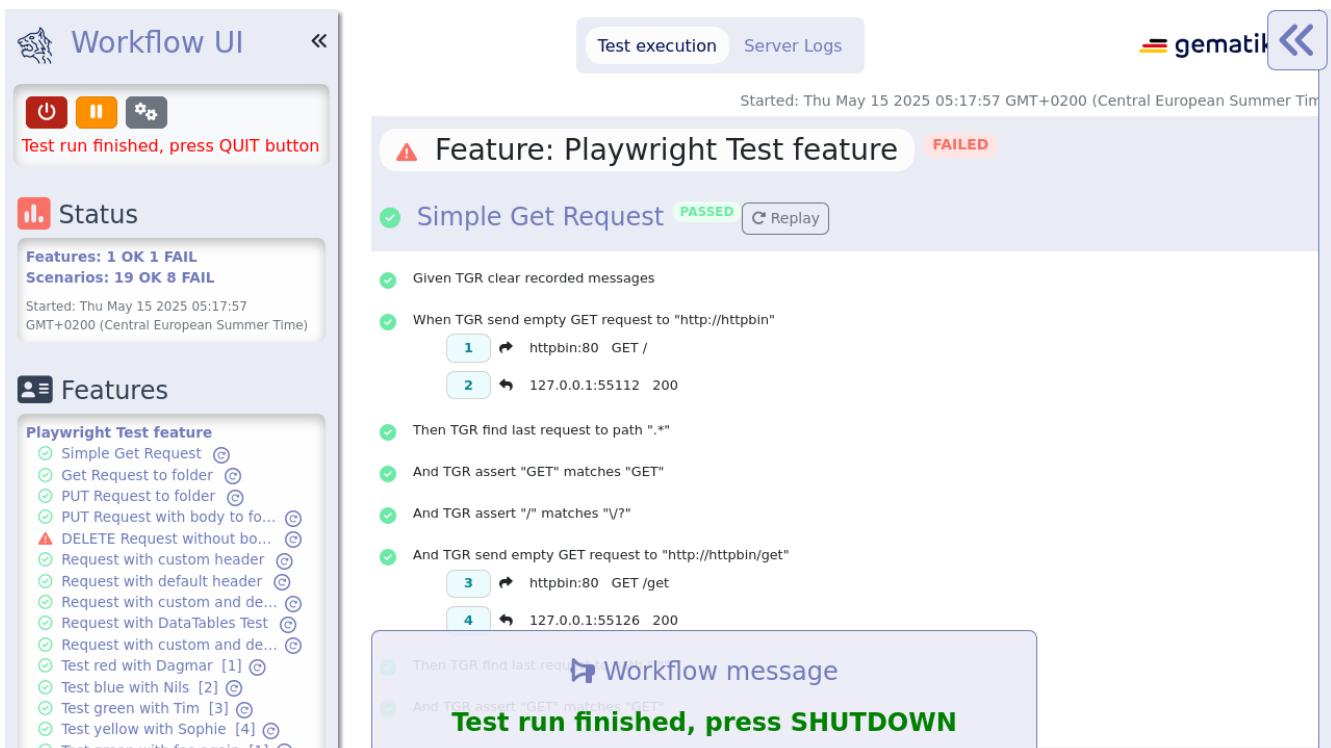


Figure 22. open status bar

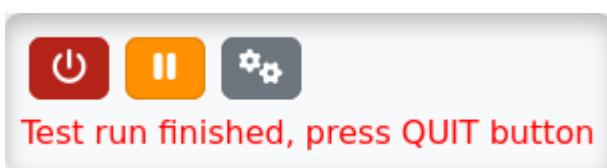


Figure 23. status bar buttons: abort, pause, configuration editor

The first button stops the test execution and terminates the servers.

As seen in the following screenshot the background color of the status bar changed to red and at a banner is shown that tells the user that the test execution has been aborted.

Once the Workflow UI has quit, searches and filtering on the [Rbel log details](#) as well as on the [Web UI](#) are no longer possible.

The screenshot shows the Workflow UI interface. At the top, there are tabs for 'Test execution' and 'Server Logs'. The status bar indicates the start time: 'Started: Thu May 15 2025 05:17:57 GMT+0200 (Central European Summer Time)'. On the left, a vertical toolbar features icons for power, pause, settings, and other functions. The main area displays a test result for 'Feature: Playwright Test feature' which failed. Below the test summary, a list of test steps is shown, each with a green checkmark indicating success. One step involves sending a GET request to 'httpbin:80'. A modal dialog box is overlaid on the screen, containing a power-off icon and the text 'Quit on user request! Please check console'.

Figure 24. test execution has stopped on user request

By clicking on the second button the test execution pauses.

The background color of the status bar and the pause button change to indicate the pause as shown in the following picture.

The screenshot shows the Workflow UI interface. The status bar indicates the start time: 'Started: Thu May 15 2025 05:17:57 GMT+0200 (Central European Summer Time)'. On the left, a vertical toolbar features icons for power, play, and settings. The main area displays a test result for 'Feature: Playwright Test feature' which failed. Below the test summary, a list of test steps is shown, each with a green checkmark indicating success. One step involves sending a GET request to 'httpbin:80'. A modal dialog box is overlaid on the screen, containing a speaker icon and the text 'Workflow message'. A banner at the top of the screen says 'Test run finished, press QUIT button'.

Figure 25. test execution is paused

The test execution will be resumed once the user clicks on the green play button.
The third button opens the Configuration Editor which is explained in detail in [this](#) section.

Below the buttons the status box shows how many feature files and how many scenarios were executed and also the amount of failed tests are shown.



Figure 26. status box

In the feature box below each scenario name is displayed.

The names are linked to the test and when the user clicks on the scenario the test is shown in the test execution on the main section.

The green icon in front of the name indicates a passed scenario, the red exclamation mark indicates a failed scenario.

The numbers in square brackets indicate that this is part of an outline scenario, meaning a test scenario that is run multiple times with different test data.

Playwright Test feature

- ✓ Simple Get Request [©](#)
- ✓ Get Request to folder [©](#)
- ✓ PUT Request to folder [©](#)
- ✓ PUT Request with body to fo... [©](#)
- ⚠ DELETE Request without bo... [©](#)
- ✓ Request with custom header [©](#)
- ✓ Request with default header [©](#)
- ✓ Request with custom and de... [©](#)
- ✓ Request with DataTables Test [©](#)
- ✓ Request with custom and de... [©](#)
- ✓ Test red with Dagmar [1] [©](#)
- ✓ Test blue with Nils [2] [©](#)
- ✓ Test green with Tim [3] [©](#)
- ✓ Test yellow with Sophie [4] [©](#)
- ✓ Test green with foo again [1] [©](#)
- ✓ Test red with bar again [2] [©](#)
- ✓ Test Find Last Request [©](#)
- ✓ Test find last request with p... [©](#)
- ⚠ Test find last request [©](#)
- ⚠ JEXL Rbel Namespace Test [1] [©](#)
- ⚠ JEXL Rbel Namespace Test [2] [©](#)
- ⚠ JEXL Rbel Namespace Test [3] [©](#)
- ⚠ JEXL Rbel Namespace Test [4] [©](#)
- ⚠ JEXL Rbel Namespace Test [5] [©](#)
- ⚠ Request a non existing url [©](#)
- ✓ Request for testing tooltips [©](#)

Zweite Feature-Datei

- ✓ Test zeige HTML [©](#)

Figure 27. feature box

Local Tiger Proxy URL http://localhost:36131
Local Tiger Proxy UI http://localhost:46007/
Local Tiger Proxy started

?

httpbin(RUNNING)

httpbin READY ▾
Checking configuration httpbin...
Adding routes to local tiger proxy for server httpbin...
Pre-start HttpBin-Server httpbin
Server httpbin up & healthy
httpbin httpbin started
httpbin started
httpbin READY

?

remoteTigerProxy(RUN...

<http://localhost:45541/> ↗

remoteTigerProxy READY ▾
Checking configuration remoteTigerProxy...
Adding routes to local tiger proxy for server remoteTigerProxy...
Pre-start Tiger Proxy remoteTigerProxy
Starting Tiger Proxy remoteTigerProxy at 45541...
Server remoteTigerProxy up &

Figure 28. server box

The server box above displays the configured servers, its status (e.g. STARTING, RUNNING, STOPPED) and some outputs of its logs.

When the icon color before the server name is green then the server is up and running correctly.

Below the server box the version number and the build date of the currently used tiger release is displayed.

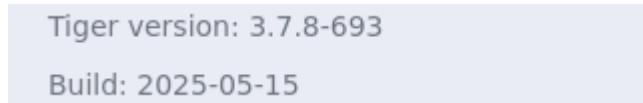


Figure 29. tiger version and build

The status bar can be minimized by clicking on the double arrow or by clicking on any of the icons in the status bar (e.g. status box icon, feature box icon, server box icon, tiger head icon).

7.1.2. Main window

The main window of the Workflow UI has two sections: the [test execution](#) and the [server logs](#) which can be selected by the two buttons on top of the Workflow UI as seen in the picture below.

Server logs

By clicking on the server logs button on top of the main window the user can have a look at the log files of each server.

There the user can use several filter options to search in the log files.

There are the following server buttons: you can see all logs of all servers, or only the logs of one or more servers by clicking on the corresponding buttons.

The user can also search via text input after a certain text phrase.

It is also possible to distinguish between the different log levels.

In the picture below only the httpbin server is selected.

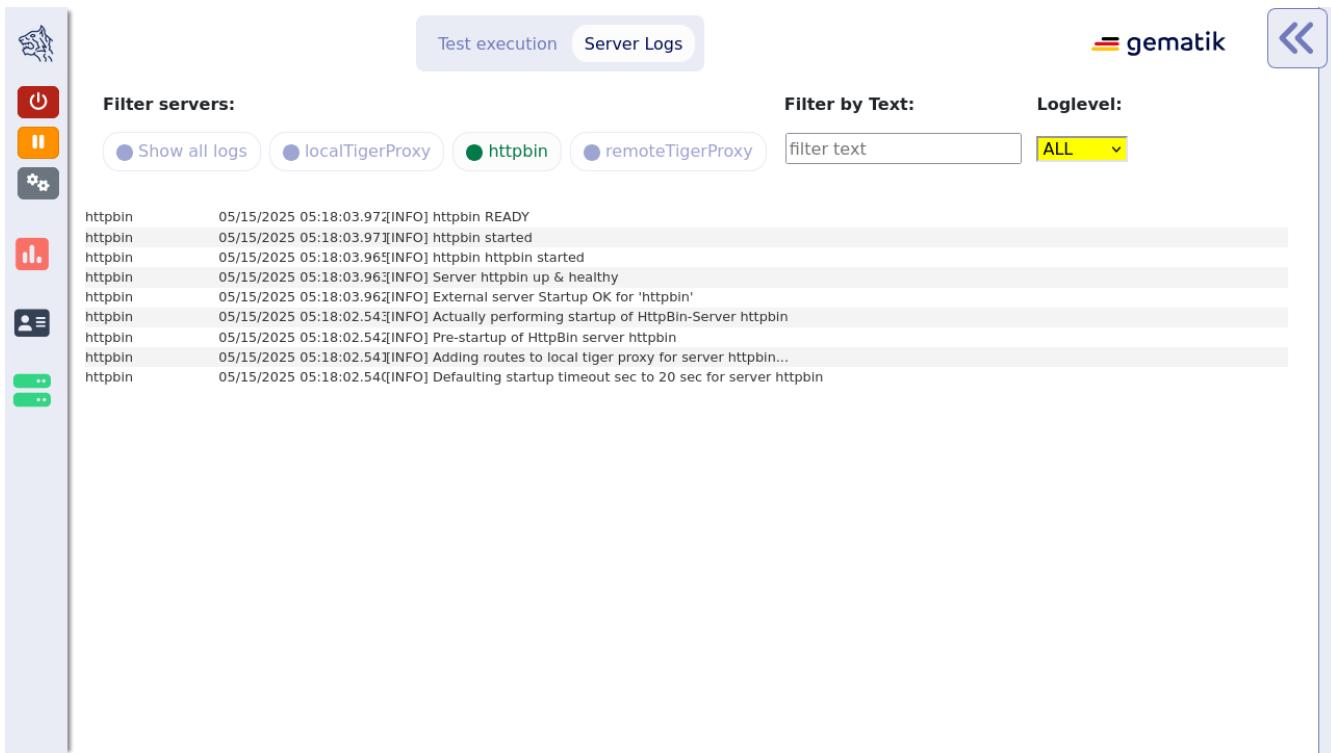


Figure 30. Server Logs with httpbin server and all log levels selected

Test execution

In the test execution tab the user sees the executed features and their scenarios as well as their execution status.

A test can be either passed or failed.

In the example below the scenario has passed but the feature itself has failed, which means that at least one of the scenarios of the feature has failed.

Feature: Playwright Test feature FAILED

Simple Get Request PASSED C Replay

- Given TGR clear recorded messages
- When TGR send empty GET request to "http://httpbin"
 - 1 ↳ httpbin:80 GET /
 - 2 ↳ 127.0.0.1:55112 200
- Then TGR find last request to path ".*"
- And TGR assert "GET" matches "GET"
- And TGR assert "/" matches "V?"
- And TGR send empty GET request to "http://httpbin/get"
 - 3 ↳ httpbin:80 GET /get
 - 4 ↳ 127.0.0.1:55126 200
- Then TGR find last req ↳ **Workflow message**
- And TGR assert "GET" matches "GET"

Test run finished, press SHUTDOWN

Figure 31. Test execution

The screenshot shows the gematik Test execution interface. At the top, there are tabs for 'Test execution' and 'Server Logs', with the date 'Started: Thu May 15 2025 05:17:57 GMT+0200 (Central European Summer Time)' displayed. On the left, a vertical sidebar contains icons for power, pause, settings, and other navigation options. The main content area displays a feature named 'Feature: Playwright Test feature' with a status of 'FAILED'. Below this, a scenario named 'Simple Get Request' is shown with a status of 'PASSED' and a 'Replay' button. The log section details four requests: 1. Given TGR clear recorded messages; 2. When TGR send empty GET request to "http://httpbin"; 3. Then TGR find last request to path "./*"; 4. And TGR assert "GET" matches "GET". A banner at the bottom right of the log area says 'Test run finished, press SHUTDOWN'.

Figure 32. execution status for scenarios and features

Beside the status at the end of the feature/scenario name the user can also see the status at the icon before the name.

During text execution or while [pausing](#) the Workflow UI there is a third status the feature/scenario can have which is "pending".

The icon before the name would be a spinner icon to indicate that status.

TGR banner step will be displayed at the bottom of the Workflow UI and will stay there till the next banner step replaces the message.

This way you can instruct manual testers to follow a specified test workflow.

This feature is used in the EAU Konnektorfarm validation test suite to guide the Primärsystem manufacturers through the interoperability combinations of signing/verifying documents against all Konnektors available.

Additionally, a test scenario can be replayed.

When clicking the replay button next to the scenario name, the scenario will be rerun again.

If placeholder variables were modified with the configuration editor, the new values will be used when replaying the scenario.



Figure 33. The replay button

Alternatively, a scenario can be replayed by clicking the small replay button in the feature box in the sidebar.



Figure 34. Small replay buttons in the sidebar

The communication requests that are called during the step execution are displayed beneath the step that initiated the request.

When the user clicks on the light blue rectangle with the number (whereas uneven numbers are requests, even number are responses) of the request then the Rbel Log view opens on the right hand side of the Workflow UI as shown on the screenshot below.

The screenshot shows the Rbel Log Details view on the left and the Tiger Proxy Log on the right. The Rbel Log Details view displays a failed Cucumber test titled "Feature: Playwright Test feature". The test step "Simple Get Request" has passed. The log details show a sequence of requests and responses between httpbin:80 and localhost:55112. The Tiger Proxy Log on the right shows a single log entry for a GET request to the root path at 127.0.0.1:55112, timestamped 05:18:07.89062133+02:00. The log entry includes the request headers (Accept: */*, Proxy-Connection: Keep-Alive, User-Agent: Apache-HttpClient/4.5.14 (Java/17.0.14), Accept-Encoding: gzip, deflate, content-length: 0, Host: localhost:42487) and a note that the request body is empty.

Figure 35. Rbel Log Details

In the Rbel Log Details view the RbelMessages are displayed that are also saved as HTML files as described in the [Cucumber and Hooks](#) section.

Next to the headline there is a link to the [WebUI](#) (aka Tiger Proxy UI) which opens the WebUI in a new tab as shown in the picture below.

This screenshot is identical to Figure 35, showing the Rbel Log Details view on the left and the Tiger Proxy Log on the right. The difference is that the "Test execution" tab is selected in the Rbel Log Details view, and there is a blue link labeled "Tiger Proxy Log" next to the "Server Log" tab. This link is intended to open the full WebUI interface.

Figure 36. Link to open the Tiger Proxy Log

The Rbel Log Details view is described in the [WebUI](#) section as it is a slimmed down version of the WebUI.

In order to increase/decrease the width of the Rbel Log Details view the user can drag the border

between the main window and the Rbel Log Details view.

The Rbel Log Details can be minimized by clicking on the double arrow on the top left of the Rbel Log Details section.

7.1.3. Traffic Visualization

An additional feature of the Workflow UI is the traffic visualization.

This feature allows to visualize the traffic between the servers under test in a sequence diagram.

The feature needs to be explicitly enabled in the `tiger.yaml` configuration file.

```
lib:  
  trafficVisualization: true
```

This will enable a third section in the main window of the Workflow UI where a sequence diagram is displayed.

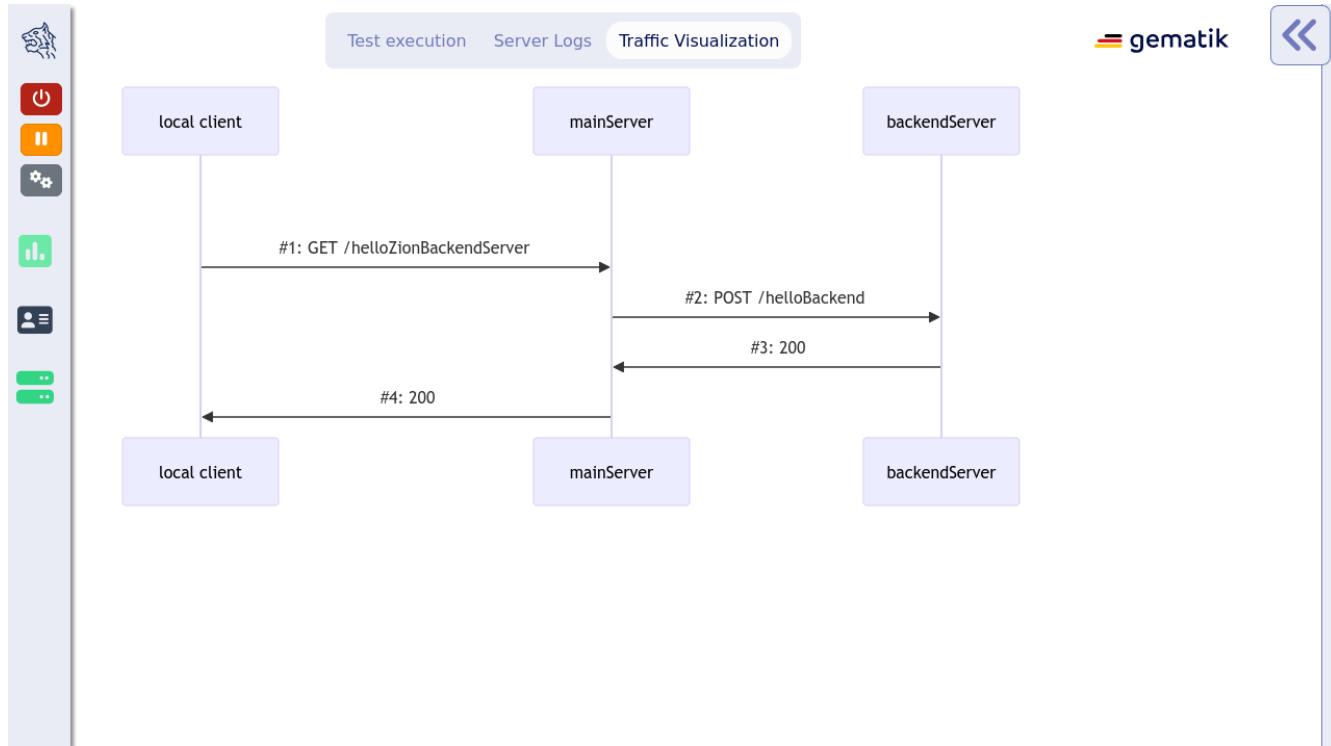


Figure 37. Traffic Visualization

The sequence diagram shows the messages that were exchanged between the servers under test. By clicking on a message in the sequence diagram, the corresponding Rbel Log Details will be displayed in the Rbel Log Details section.

The traffic visualization currently supports the following server types: `externalJar`, `externalUrl`, `zion`, `docker` and `compose`.

For messages to show up in the sequence diagram they need to be routed through the Tiger Proxy. This is the case for all the messages originating in the local tiger client and their responses. If there are additional messages originating on one the servers under test, they need to be routed through the Tiger Proxy as well.

In Zion servers this is automatically configured.

For external jars this can be achieved by configuring the servers with the following VM options:

```
externalJarOptions:  
  options:  
    - -Dhttp.proxyHost=127.0.0.1  
    - -Dhttp.proxyPort=${tiger.tigerProxy.proxyPort}
```

For the server types docker and compose, we do not yet support the visualization of messages originating on a client port of these servers.

7.2. Postpone start of test scenarios

It is possible to configure the Workflow UI to not start all tests automatically at startup.

When setting the following configuration option in the `tiger.yaml`:

```
lib:  
  runTestsOnStart: false # default is true
```

the tests will only be discovered and displayed in the Workflow UI.

The user can then choose which test scenario he/she wants to execute.

This can be done by clicking the play buttons in either the Test Execution pane or in the Status Bar.

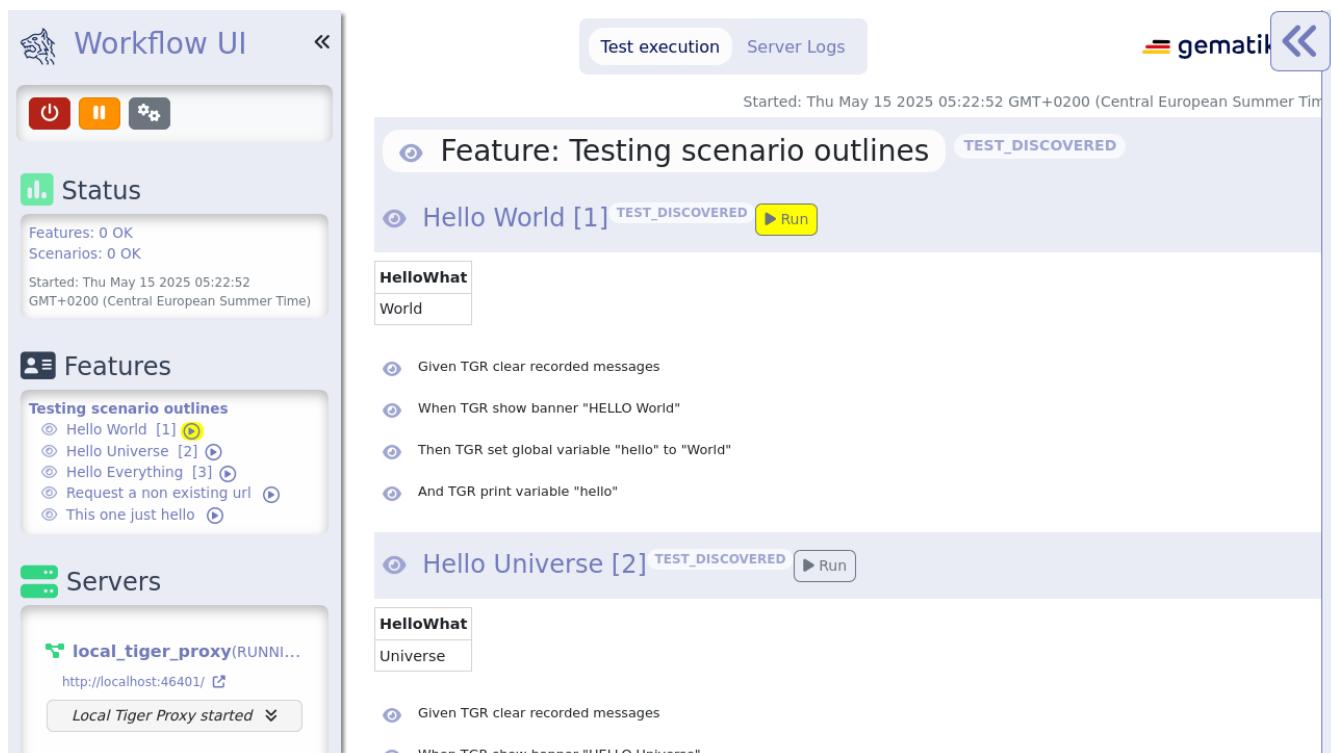


Figure 38. Play Buttons

7.3. Standalone Tiger Proxy Log

To watch the recorded messages and to be able to analyze issues at test run time already you can visit the Tiger Proxy web user interface at:

`http://127.0.0.1:${SERVERPORT}/webui`

With `ADMINPORT` being the configured server port of the Tiger Proxy.

When the user works with the [Workflow UI](#) the Tiger Proxy UI can be opened via a link in the Rbel Log Details view in a new browser tab.

7.3.1. Overview

The following screenshot shows the WebUI.

On the left side the request/response pairs are displayed.

The user can see the request type and the error code of the response as well as the timestamp of the request.

The screenshot displays the Tiger Proxy WebUI interface. At the top, there is a header bar with the Tiger Proxy logo, search and filter buttons, and a dropdown menu labeled "Rbel Path...". Below the header, the main area shows a list of log entries on the left and detailed message information on the right.

Left Panel (Log Entries):

- #1 → 2 Request 5:18:07 AM.890
HTTP GET /
- #2 ← 1 Response 5:18:07 AM.919
HTTP 200
- #3 → 4 Request 5:18:09 AM.523
HTTP GET /get
- #4 ← 3 Response 5:18:09 AM.551
HTTP 200
- #5 → 6 Request 5:18:11 AM.204
HTTP GET /get
- #6 ← 5 Response 5:18:11 AM.212
HTTP 200
- #7 → 8 Request 5:18:13 AM.8
HTTP PUT /put
- #8 ← 7 Response 5:18:13 AM.41
HTTP 200
- #9 → 10 Request 5:18:14 AM.861
HTTP PUT /put

Right Panel (Message Detail):

Message #1:

1 → 2 GET /
05:18:07.89062133+02:00 127.0.0.1:55112 → httpbin:80

REQ Headers:

Accept	/*
Proxy-Connection	Keep-Alive
User-Agent	Apache-HttpClient/4.5.14 (Java/17.0.14)
Accept-Encoding	gzip, deflate
content-length	0
Host	localhost:42487

REQ Body Empty

Message #2:

2 → 1 200

Figure 39. Tiger Proxy Log

On the top right the tiger version and the build date are displayed.

In the middle the full request and response messages are shown with detailed header and body.

Filter Modal

When a lot of messages are recorded, it is sometimes hard to find the message you are looking for. Therefore, the user can filter the messages with a Rbel-Path or a regex using either the filter modal as shown in the picture below or the JEXL Debugging modal described [here](#).

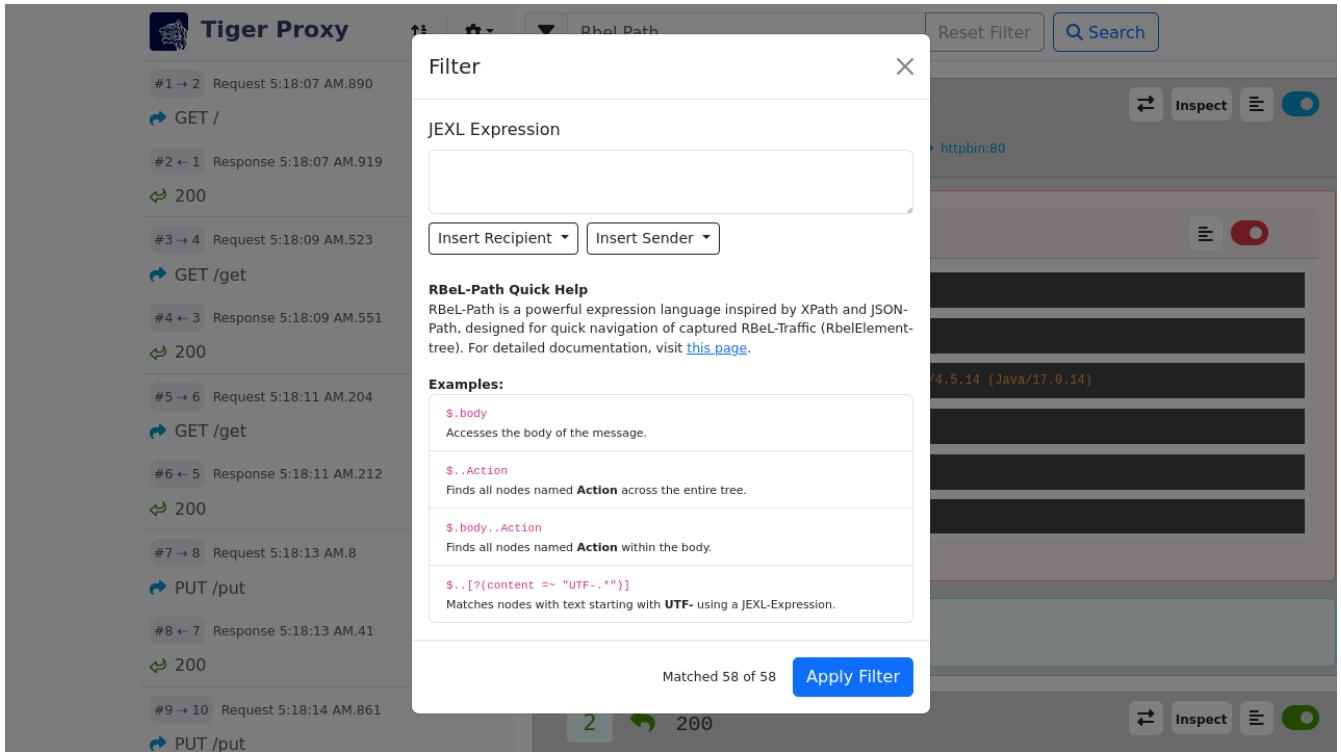


Figure 40. Filter Management

RBel-Path/JEXL Debugging Modal

When the user wants to inspect a Rbel-Path or have a look at some JEXL expressions, the user can click on the corresponding button in the top right corner of the request or the response that is highlighted in the following screenshot.

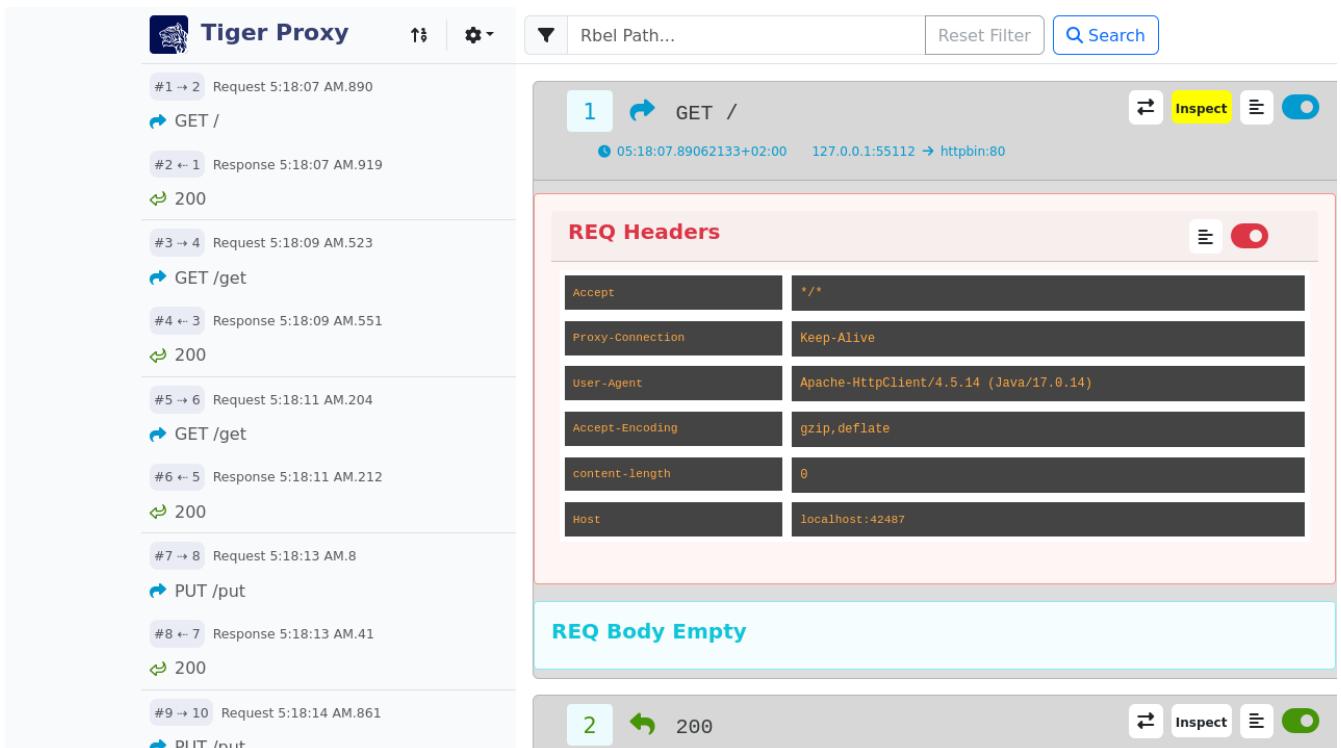


Figure 41. Access RBel-Path/JEXL Debugging

The picture below shows the RBel-Path tab.

The user can execute the RBel-Path on the request or the response and the result is displayed in the

bottom part of the modal.

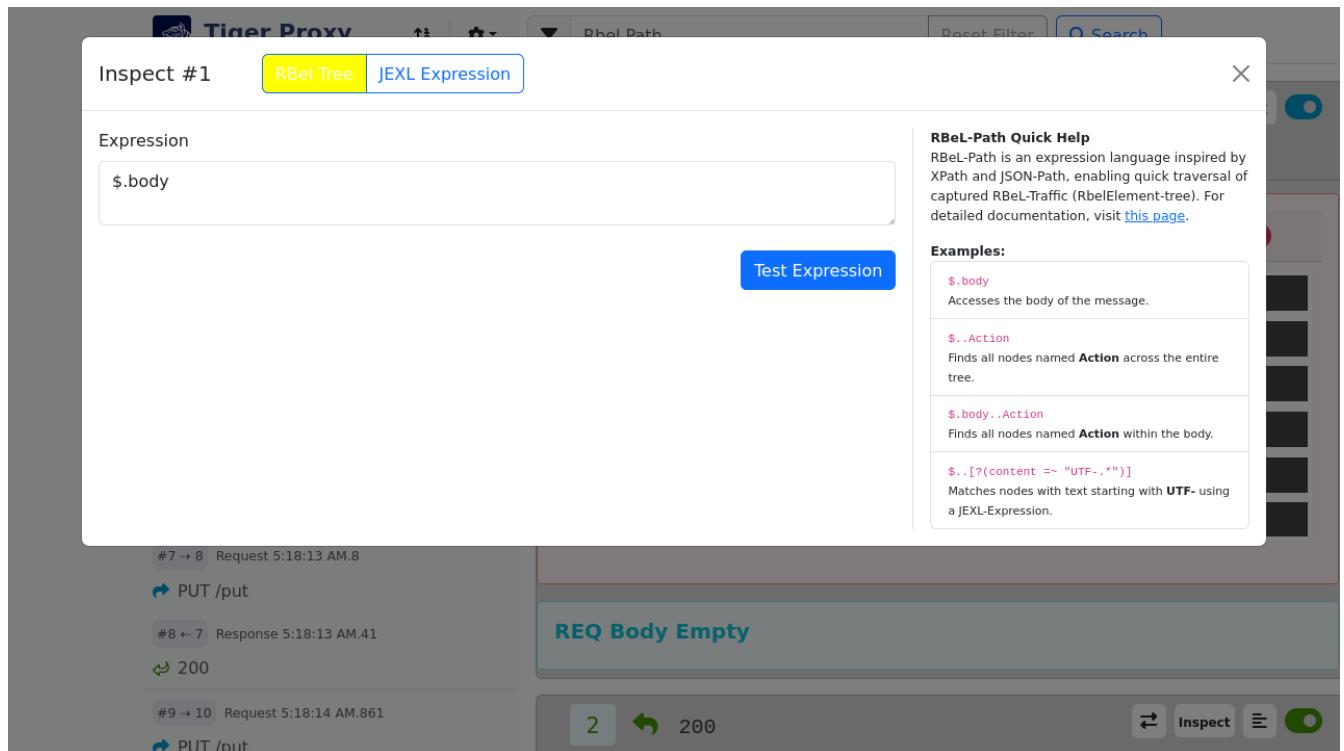


Figure 42. RBel-Path

For more information in the Rbel-Path check out [this](#) section.

The picture below shows the JEXL Debugging tab.

The user can execute the JEXL expression on the request or the response and the result is displayed in the bottom part of the modal.

Further information on JEXL expressions can be found in [Explanation of JEXL Expressions](#).

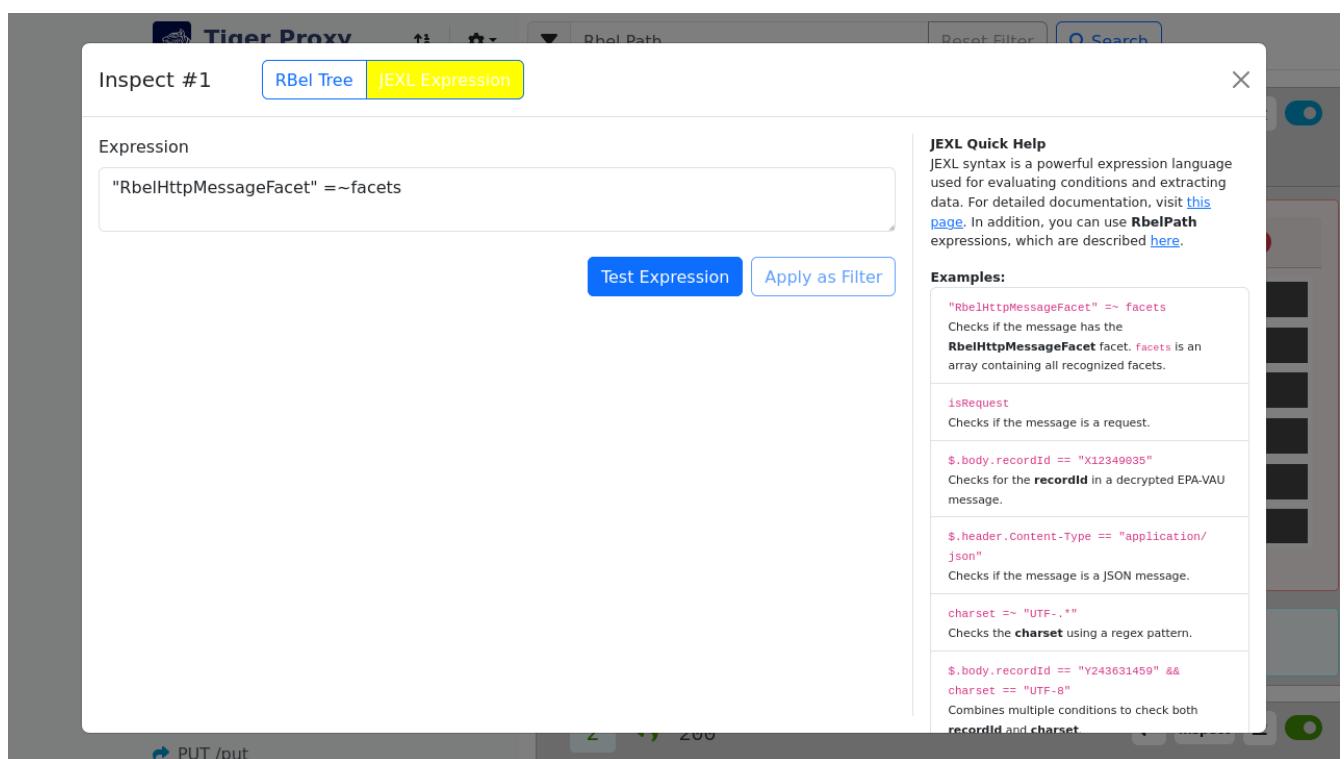


Figure 43. JEXL Debugging

When the user wants to filter the messages with a JEXL expression, the user can click on the "Use as filter" button in the modal.

Behind the settings icon are some modals that are explained in more detail in the following sections.

Settings

Behind the settings icon several actions can be triggered:

- **Message Options** ... has two possibilities (hide headers and hide details) which collapses either all headers (request headers as well as response headers) or all the detailed information of the requests and responses
- **Export** ... allows you to export all or the currently filtered received messages as an HTML page or as a machine-readable tgr file.
- **Configure Routes** ... allows you to modify and add the routes configured on this Tiger Proxy
- **Reset Messages** ... allows you to reset all the messages and import a previously stored traffic file.
- **Quit Proxy** ... quits the Tiger Proxy.

Routing Modal

The user can add/delete routes in the routing modal which is shown in the following screenshot.

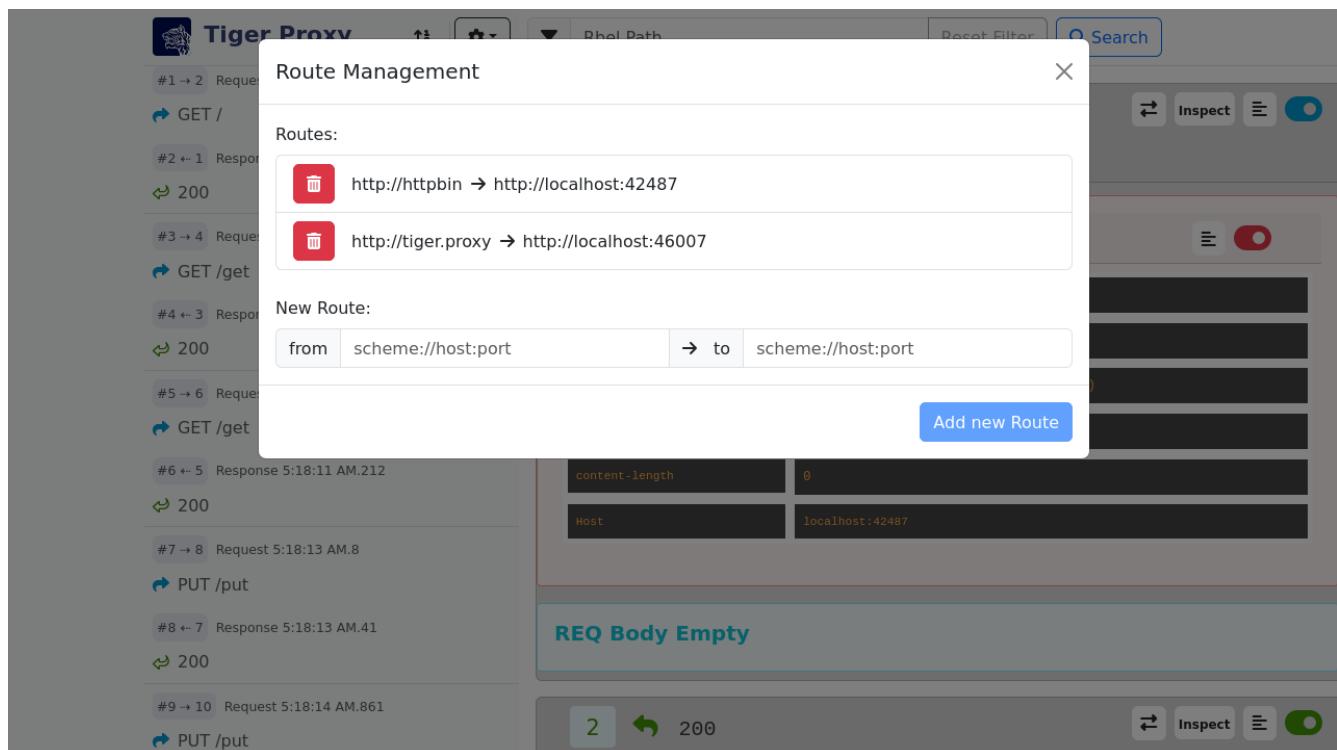


Figure 44. Route Management

Message Content

The user can have a look at the request/response message content of the header, body or both by clicking on the corresponding button in the top right corner of the request or the response that is

highlighted in the following screenshot.

The screenshot shows the Tiger Proxy interface. On the left, a list of messages is displayed with numbers and descriptions. On the right, a detailed view of message #1 is shown. The top part of the view shows the request details: method GET, path /, timestamp 05:18:07.89062133+02:00, and source 127.0.0.1:55112 → httpbin:80. Below this is the 'REQ Headers' section, which lists various headers with their values. The 'REQ Body Empty' section indicates that the response body is empty. At the bottom, the response summary shows code 200. There are several buttons at the top right: 'Inspect' (highlighted in yellow), 'List' (highlighted in green), and a toggle switch.

Figure 45. Buttons to show message content

The picture below shows the content of the whole response.

The screenshot shows the Tiger Proxy interface. A modal window titled 'Raw Content #2' is displayed in the center, containing the full HTTP response for message #2. The response is as follows:

```
HTTP/1.1 200 OK
Content-Length: 14
Content-Type: text/plain; charset=UTF-8
Connection: keep-alive

Hello World!!!Content-Length: 14
Content-Type: text/plain; charset=UTF-8
Connection: keep-aliveHello World!!!
```

The background of the interface is dimmed to indicate that the modal is active. The rest of the interface, including the message list and other panels, is visible but less prominent.

Figure 46. Example of the content of a response

Switching between request/response

Since the order in the list is based upon the reception of the corresponding message it can be hard to find the corresponding request or response to a given message.

To make this easier the user can switch between the request and the response by clicking on the

corresponding button in the top right corner of the request or the response that is highlighted in the following screenshot.

The screenshot shows the Tiger Proxy application interface. On the left, there is a list of proxy interactions:

- #1 → 2 Request 5:18:07 AM.890 GET /
- #2 ← 1 Response 5:18:07 AM.919 200
- #3 → 4 Request 5:18:09 AM.523 GET /get
- #4 ← 3 Response 5:18:09 AM.551 200
- #5 → 6 Request 5:18:11 AM.204 GET /get
- #6 ← 5 Response 5:18:11 AM.212 200
- #7 → 8 Request 5:18:13 AM.8 PUT /put
- #8 ← 7 Response 5:18:13 AM.41 200
- #9 → 10 Request 5:18:14 AM.861 PUT /put

On the right, a detailed view of the first request (#1) is shown:

- REQ Headers:**

Accept	/*
Proxy-Connection	Keep-Alive
User-Agent	Apache-HttpClient/4.5.14 (Java/17.0.14)
Accept-Encoding	gzip, deflate
Content-Length	0
Host	localhost:42487
- REQ Body Empty**
- RESPONSE:** 200

Figure 47. Buttons to switch between request and response

7.4. Explanation of JEXL Expressions

In the [Workflow UI](#) and in the [WebUI](#) you can inspect the requests and response messages. For that you can use RbelPath and/or JEXL expressions. This section should give you a brief review on the JEXL expressions.

Important to know is that an JEXL expression is usually a "condition1 operator condition2" expression which is compared.

Therefor the following operators could be used.

7.4.1. Operators

Operator	Description
r	
and &&	cond1 <i>and</i> cond2 and cond1 <i>&&</i> cond2 are equivalent
or	cond1 <i>or</i> cond2 and cond1 <i> </i> cond2 are equivalent
not !	The ! operator can be used as well as the word <i>not</i> , e.g. !cond1 and not cond1 are equivalent
==	Equality, e.g. cond1 == cond2 null is only ever equal to null, that means when you compare null to a non-null value, the result is false.

Operator	Description
r	
!=	Inequality
>	Greater than
<	Less than
>=	Greater than or equal
≤	Less than or equal
=~	In or match, can be used to check that a string matches a regular expression. For example "abcdef" =~ "abc.*" returns true. It also checks whether any collection, set or map contains a value or not; in that case, it behaves as an "in" operator. "a" =~ ["a","b","c","d","e","f"] returns true.
!~	Not in or not-match, can be used to check that a string does not match a regular expression. For example "abcdef" !~ "abc.*" returns false. It also checks whether any collection, set or map does not contain a value. "a" !~ ["a","b","c","d","e","f"] returns false.
=^	startsWith, for example "abcdef" =^ "abc" returns true
!^	startsNotWith, "abcdef" !^ "abc" returns false
=\$	endsWith, for example "abcdef" =\$ "def" returns true
!\$	endsNotWith, for example "abcdef" !\$ "def" returns false
Empty	The unary empty operator behaves like the corresponding function empty().
size	The unary size operator behaves like the corresponding function size().

Jexl Debugging

Execute Jexl on selected message `$.responseCode == "404"`

JEXL syntax Help

Condition is true: `$.responseCode == "404"`

JEXL context

```
@
{
    "receiver": null,
    "sender": {
        "port": "80",
        "domain": "winstone"
    },
}
```

Figure 48. Rbel Path Expression

7.4.2. Access on Array, Lists and Maps

To access maps in JEXL/RbelPath the point notation is used. In case of lists use the number of the list entry you want to access, starting with 0, 1, 2 and so on.

Jexl Debugging

Execute Jexl on selected message `$.body.pairing_entries.0.name == "idpClient"`

Figure 49. The access of the elements of a list is done with the number, starting with 0. For maps the point notation is used.

7.4.3. Access JEXL contexts

There are predefined JEXL contexts which can be used for the query, for example `isRequest`, `isResponse`, `charset`, `content` or also more complex contexts like `response.statuscode`, `request.url`, `message.method` etc. For more details check the InlineJexlToolbox

Jexl Debugging

Execute Jexl on selected message `response.headers.'Content-Type'.0==^"text"`

JEXL syntax Help

Condition is true: `response.headers.'Content-Type'.0==^"text"`

Figure 50. Use single quotes when using JEXL contexts with a hyphen.

7.4.4. More Examples

`message.headers.'content-length'.0 == "0"` → Use single quotes when using JEXL contexts with a hyphen.

`@.body.0.name.content =^ "Jasmin"` → check whether the content starts with "Jasmin"

`$.body.recordId == "X12349035"` → checks for the recordId of a decrypted EPA-VAU-message

`$.header.Content-Type == "application/json"` → check if the message is a JSON-message

`request.method == "GET"` → check if request is da GET request

`charset =~ "UTF-.*"` → check the charset with a regex

`empty(response.url)==true` oder auch `empty(response.url)` → url is not set

`$.body.recordId == "Y243631459" && charset == "UTF-8"` → combines the two criterions

7.4.5. POST Form / GET parameters

When accessing parameters POST and GET are handled differently.

POST form data requests contain the data as Url encoded query string in the body of the request.

There is no easy way to decode this data generically within Rbel/JEXL.

To help you ease the situation for POST we do have a helper JEXL inline method:

```
!{urlEncoded('value')}
```

To access POST form data you may use `$.body.paramname` which will return the URL encoded value.

For GET requests you have two options:

- `$.path.paramname` which will return the string "paramaname=URLENCODED_VALUE" or
- `$.path.paramname.value` which will return the URL decoded original value.

For further help on JEXL please check out the official website (<https://commons.apache.org/proper/commons-jexl>).

Chapter 8. Tiger Zion

Tiger Zion is a server that is highly customizable.

It serves as a stand-in, an interactive, Zero-Line mock for more complicated servers.

It can be stateful, do backend-requests, give conditional answers (allowing for error cases, edge cases).

It can return JSON, XML, JWT, JWE and many more, nest these formats into each other.

The underlying templating language supports loops and conditions.

It can be used both as a standalone server or as a server directly inside of tiger.

In order to use zion, make sure that you add the tiger-zion dependency in the most recent version.

```
<dependency>
    <groupId>de.gematik.test</groupId>
    <artifactId>tiger-zion</artifactId>
    <version>${tiger.version}</version>
</dependency>
```

8.1. Simple canned response

To start lets write a server that has one endpoint: Return a simple JSON.

We do this as part of a tiger testsuite.

Listing 24. tiger.yaml

```
servers:
  zionServer:
    type: zion
    zionConfiguration:
      # define the server port.
      # In a real setup you would leave this empty (then a random port will be used)
      serverPort: 8080
      mockResponses:
        # a map with all the responses
        helloWorld:
          # the name is arbitrary. It can be used to alter the response later on
          requestCriterions:
            # This is a list of JEXL expressions. Only when all expressions are met is
            # this response considered
            - message.method == 'GET'
            - message.path == '/helloWorld'
        response:
          statusCode: 200
          body: '{"Hello":"World"}'
```

This will give us:

```
$ curl localhost:8080/helloWorld
{"Hello": "World"}
```

8.2. Looping (tgrFor)

Let's now make this a bit more interactive: Return a map of all given HTTP headers.
And please return it as an XML!

Because XMLs are large and get complicated easily we don't want to write it directly inside the [tiger.yaml](#).

So we reference an external file, where we do the heavy lifting:

Listing 25. tiger.yaml

```
servers:
  zionServer:
    type: zion
    zionConfiguration:
      serverPort: 8080
    mockResponses:
      helloWorld:
        requestCriterions:
          - message.method == 'GET'
          - message.path == '/helloWorld'
        response:
          statusCode: 200
          bodyFile: complicatedResponse.xml
```

Listing 26. complicatedResponse.xml

```
<?xml version="1.0" encoding="utf-8" ?>
<requestDescription>
  <headers>
    <header>
      <tgrFor>header : request.headers.entrySet()</tgrFor>
      ${header}
    </header>
  </headers>
</requestDescription>
```

Let's try it out:

```
$ curl localhost:8080/helloWorld
<?xml version="1.0" encoding="UTF-8"?>

<requestDescription>
  <url>http://localhost:8080/helloWorld</url>
```

```

<path>/helloWorld</path>
<headers>
    <header>host=[localhost:8080]</header>
    <header>accept=[/*/*]</header>
    <header>user-agent=[curl/7.88.1]</header>
</headers>
</requestDescription>

```

Here you see a **tgrFor** construct.

As the name leads on, it is used for looped rendering.

It reside WITHIN the element which needs to be considered, which is a bit different compared to other templating languages.

It just has to be beneath the element in the logical tree of the document.

So the **tgrFor** can be a tag within **<header>**, but it also could have been an attribute.

It can also be used in a JSON-fragment: The tgrFor then has to be a field in the object that you want looped.

The **tgrFor** must consist of three parts: The name of the loop-variable (here: **header**), followed by colon and then the loop expression to be evaluated.

The loop expression must yield a Java-Collection as a result.

Some examples:

1..42

{ 'one' , 2, 'more'}

{ 'one' : 1}.entrySet()

The loop-variable will be set for every iteration in the TigerConfiguration.

So to access it directly, use **\${myLoopVariable}**.

You can then also combine the loop-variable with JEXL like so:

Listing 27. complicatedResponse.xml

```

<?xml version="1.0" encoding="utf-8" ?>
<requestDescription>
    <headers>
        <header>
            <tgrFor>header : {1,2,3}</tgrFor>
            ${header} and !{ ${header} + 1}
        </header>
    </headers>
</requestDescription>

```

will lead to

```

<?xml version="1.0" encoding="UTF-8"?>
<requestDescription>
  <headers>
    <header>1 and 2</header>
    <header>2 and 3</header>
    <header>3 and 4</header>
  </headers>
</requestDescription>

```

8.3. Conditional rendering (tgrIf)

To make the presence of elements conditional you can use the **tgrIf** statement.
Consider the following example:

Listing 28. complicatedResponse.xml

```

<?xml version="1.0" encoding="utf-8" ?>
<requestDescription>
  <checkIf tgrIf="1 &lt; 5" logic="still applies" />
</requestDescription>

```

This will give us

```

<?xml version="1.0" encoding="UTF-8"?>

<requestDescription>
  <checkIf logic="still applies"></checkIf>
</requestDescription>

```

The tgrIf statement just consist of one single JEXL expression.

The result must be of type boolean.

Please note that the tgrIf-statement, like the tgrFor, has to be beneath the target element in the document tree.

This can be done via an attribute in XML, but it can also be done using a tag:

Listing 29. complicatedResponse.xml

```

<?xml version="1.0" encoding="utf-8" ?>
<requestDescription>
  <checkIf logic="still applies">
    <tgrIf>1 &lt; 5</tgrIf>
  </checkIf>
</requestDescription>

```

Here are some examples for other possible criteria:

```
$.header.connection == 'Keep-Alive'
```

This will only be true if the matching header is present.

```
$.header.host =~ 'local.*'
```

You can also use the more complex JEXL operators (here `=~`, comparing using a regex).

8.4. Backend request

To simulate complex interactions you can execute backend requests.

The following example should clarify the mechanism:

We want measure the length of the response by google to a query:

Listing 30. tiger.yaml

```
servers:
  zionServer:
    type: zion
    zionConfiguration:
      serverPort: 8080
      mockResponses:
        helloWorld:
          requestCriterions:
            - message.method == 'GET'
            - message.path == '/helloWorld'
          backendRequests:
            tokenCheck:
              url: "https://google.com/?${.path.myParam.value}"
# the method is optional. GET is the default, POST if a body is present
              method: GET
              assignments:
                length: "$.header.Content-Length"
            response:
              statusCode: 200
              body: "{$responseLength":'${length}', 'testedPath':'${.path.myParam.value}'}"
```

To test, we sent:

```
$ curl "localhost:8080/helloWorld?myParam=d fds"
{"responseLength": "1566", "testedPath": "d fds"}
```

The request is sent, the path is taken from the `myParam` query-parameter of the initial request. Afterwards, the value of the `Content-Length`-Header is stored in the variable named `length` in the

TigerGlobalConfiguration.

We then sent a response with status-code 200 and json-body.

Here we first reference the measured `length` variable from the backend-request and next we return the testedPath, taking the parameter from the initial request.

8.5. Nested response

To reduce the overhead when simulating conditional responses you can use the `nestedResponses` mechanism.

This allows subdividing responses.

Consider the following example, where we check the password of the calling party (which is given in cleartext in the request header).

Listing 31. tiger.yaml

```
servers:
  zionServer:
    type: zion
    zionConfiguration:
      serverPort: 8080
      mockResponses:
        passwordCheckResponse:
          requestCriterions:
            - message.method == 'GET'
            - message.path == '/helloWorld'
        nestedResponses:
          correctPassword:
            importance: 10
            requestCriterions:
              - "?{$.header.password}" == 'geheim'
            response:
              statusCode: 200
              body: '{"Hello":"World"}'
          wrongPassword:
            importance: 0
            response:
              statusCode: 405
              body: '{"Wrong":"The password !{$.header.password} is not correct"}'
```

The two answers are both considered.

Since they are stored in the YAML as a map, the order in the YAML is of no significance.

Rather you have to specify the importance of a response, with a higher number meaning a higher importance meaning the response will be considered first.

8.6. Matching path variables

In many REST-Apis it is usual to include variables as part of the resource path.

Zion allows to configure a response that will match a path and extract the given variables.

The assigned values can then be used in the response or be used in additional matching criteria. Here is an example:

Listing 32. tiger.yaml

```
servers:
  zionServer:
    type: zion
    zionConfiguration:
      serverPort: 8080
    mockResponses:
      users:
        request:
          path: "/users/{userId}"
          method: "GET"
          additionalCriterions:
            - "${userId}" == '123'
        response:
          statusCode: 200
          body: "{id:'${userId}', 'username': 'Tiger'}
```

The matching of the response is made with the new configuration entry `request`. Here we defined the `path` and `method` that should match and `additionalCriterions`. In the path we can see a variable defined with `{userId}`.

When making the following request:

```
$ curl "localhost:8080/users/123"
{id:'123', 'username': 'Tiger'}
```

the variable `userId` will be matched with the requested url and be assigned the value of "123". This value can then be used in the `additionalCriterions` and in the `body`.

The matching of paths using the `request` configuration can also be made using nested responses. The path to match will combine paths specified in the different levels of the nested response. For example:

Listing 33. tiger.yaml

```
servers:
  zionServer:
    type: zion
    zionConfiguration:
      serverPort: 8080
    mockResponses:
      users:
        request:
          path: "/users"
        nestedResponses:
```

```

getSpecificUser:
  request:
    path: "/{userId}"
    method: "GET"
    additionalCriterions:
      - "'${userId}' == '123'"
  response:
    statusCode: 200
    body: "{id:'${userId}', username: 'Tiger'}"
addUser:
  request:
    method: "POST"
    path: ""
  response:
    statusCode: 201
    headers:
      Location: "/users/456"

```

Here we have two nested responses in the `/users` path.

One will match GET requests to the path `/users/{userId}` and the other will match POST-Requests to the path `/users`.

8.7. tgrEncodeAs

One of the core capabilities of Zion is the ability to switch between media types.

You can return XML, JSON, JWT and many more types.

You can also embed one into the other.

As an example we want to return a JSON containing a freshly signed JWT (JSON Web Token). For this we use the following response body file:

Listing 34. complicatedResponse.json

```
{
  "myToken": {
    "tgrEncodeAs": "JWT",
    "header": {
      "alg": "BP256R1",
      "typ": "JWT"
    },
    "body": {
      "name": "Max Power",
      "iat": {
        "tgrAttributes": ["jsonNonStringPrimitive"],
        "value": "!{currentTimestamp()}"
      }
    },
    "signature": {
      "verifiedUsing": "idpEnc"
    }
}
```

```
}
```

will lead to

```
{
  "myToken": "eyJhbGciOiJCUDI1NjIxIiwidHlwIjoiSl0In0.eyJ0YWlkIjogIk1heCBQb3dlciIsImhdCI6IDE20Dg2MzQ5MjR9.aOnFmxSkzvo9fJjnDSFCeX0G5-IP3XFQPZCRyZFB0EyBAgV2Dy3ImEjz_DpFRqSqtkHdkCcVT_e6aBejN_A2g"
}
```

We see the keyword `tgrEncodeAs` being used here.

Currently the following values are supported: `XML`, `JSON`, `JWT`, `JWE`, `URL`, `BEARER_TOKEN`.

A JWT consists of three parts: header, body, signature.

The given nodes are searched and taken.

The description of the JWT also could have been done in XML.

We then see another mode-switch being done in the `iat`-claim in the body of the JWT: `iat` is the Unix-Timestamp at which the token was issued.

For our faked ad-hoc token we of course want to use the current time for this claim.

Unfortunately the `iat` claim is a number, which precludes the direct use of a JEXL-expression.

To solve this problem make the claim complex, add the `"jsonNonStringPrimitive"` attribute to the resulting node and set the value to the desired value.

This also works for floating-point and boolean values.

8.8. RbelWriter content structures

In this paragraph we'll take a look at the various structures that can be serialized with the RbelWriter, which sits at the core of the Zion-Server.

The following examples are kept in JSON (apart from the Bearer token example).

Please note that the same result can be achieved from XML (or any other format for that matter).

8.8.1. JWT

```
{
  "tgrEncodeAs": "JWT",
  "header": {
    "alg": "BP256R1",
    "typ": "JWT"
  },
  "body": {
    "claim": "value"
  },
  "signature": {
    "verifiedUsing": "idpEnc"
  }
}
```

```
}
```

The three parts denote the different part of a JWT: The header claims (header), body claims (body) and signature properties (signature).

RbelWriter will automatically try to sign the given JWT.

For this the key at `$.signature.verifiedUsing` is consulted and a matching key is searched in the tiger key-database.

This will be filled at start-time by digging through the root-directory of the application recursively and trying to open all found key-files with various default passwords.

Please note that the header-claims have to match the given key, otherwise the signing operation can't be completed successfully.

8.8.2. JWE

```
{
  "tgrEncodeAs": "JWE",
  "header": {
    "alg": "ECDH-ES",
    "enc": "A256GCM"
  },
  "body": {
    "some_claim": "foobar",
    "other_claim": "code"
  },
  "encryptionInfo": {
    "decryptedUsingKeyId": "idpEnc"
  }
}
```

As with the JWT, for the JWE all relevant claims are to be found in the appropriate sections.

The signature has been replaced by the `encryptionInfo`-section.

Here you need to specify the key to be used for the encryption.

Here in this example we are using a public/private key-pair (with the same name as before).

Again the header claims have to match the key used.

Apart from a public/private key-pair you can also use direct keys to encrypt your JWE.

Here is an example:

```
{
  "tgrEncodeAs": "JWE",
  "header": {
    "alg": "dir",
    "enc": "A256GCM"
  },
  "body": {
    "some_claim": "foobar",
    "other_claim": "code"
  }
}
```

```

},
"encryptionInfo": {
  "decryptedUsingKey": "YVI2Ym5wNDVNb0ZRTWFmU1Y1ZTZkRTg1bG9za2tscjg"
}
}

```

As we are using a AES 256 bit key the supplied key has to exactly carry 256 bits, Base64 encoded.

8.8.3. URL

To generate a URL you can also use the RbelWriter.

Consult the following structure:

```
{
  "tgrEncodeAs": "url",
  "basicPath": "http://blub/fdsa",
  "parameters": {
    "foo": "bar"
  }
}
```

The parameters will be added as query-parameters.

This can be rather useful to construct more complex parameters, for example a dynamically generated JWT.

8.8.4. Bearer Token

A Bearer token can also be serialized directly via RbelWriter.

This is very relevant if you want to, for example, generate a JWT on the fly and use it as the Bearer token.

The straight-forward way would be to directly write the Bearer token like so:

```

Bearer {
  "tgrEncodeAs": "JWT",
  "header": {
    "alg": "BP256R1",
    "typ": "JWT"
  },
  "body": {
    "claim": "value",
  },
  "signature": {
    "verifiedUsing": "idpEnc"
  }
}
```

Here the whole arrangement will be parsed as a Bearer token (Which is essentially the Word

`Bearer`, followed by a space and any string).

The value of the `Bearer` token will be parsed as a JSON.

When the result is then serialized, the `tgrEncodeAs` is interpreted and the JSON will be written as a Base64-encoded JWT.

The following example will produce the same result.

However the overall structure is a JSON, which will be written as a `Bearer` token (`"tgrEncodeAs": "BEARER_TOKEN"`).

The content of the token is taken from the child-node with the name `BearerToken`.

```
{  
  "tgrEncodeAs": "BEARER_TOKEN",  
  "BearerToken": {  
    "tgrEncodeAs": "JWT",  
    "header": {  
      "alg": "BP256R1",  
      "typ": "JWT"  
    },  
    "body": {  
      "claim": "value"  
    },  
    "signature": {  
      "verifiedUsing": "idpEnc"  
    }  
  }  
}
```

Chapter 9. Tiger Extensions

Tiger has certain extensions that fulfil certain tasks.

The different extensions are shortly described in the following sections.

9.1. Tiger Konnektor Management Extension

The Tiger-Konnektor-Management-Extension provides an interface to the KMS system of different connector providers.

It also provides cucumber feature steps to access the different connectors.

This is an internal gematik extension and can be found on GitLab <https://gitlab.prod.ccs.gematik.solutions/git/Testtools/tiger/tiger-konnektor-management-extensions>.

9.2. Tiger On FHIR Extension

The Tiger-On-Fhir-extension provides a set of simple BDD steps which can be used to check for valid FHIR content therein.

FHIR stands for Fast Healthcare Interoperability Resources.

The tiger-on-fhir can be found on GitHub <https://github.com/gematik/tiger-on-fhir> and it uses the Gematik Referenzvalidator located on GitHub <https://github.com/gematik/app-referencevalidator>.

9.3. Tiger CATS Extension

CATS stands for Card Terminal Simulator.

Tiger-Cats-Extensions offers the option to use the REST interface of CATS as Java functionality or as BDD steps.

This is an internal gematik extension and can be found on GitLab <https://gitlab.prod.ccs.gematik.solutions/git/Testtools/tiger/tiger-cats-extensions>.

9.4. Tiger Cloud Extension

The Tiger-Cloud-Extension allows to embed docker image based containers, docker compose scripts and even helm charts to local or remote kubernetes clusters.

The GitHub repo is <https://github.com/gematik/tiger-cloud-extension>.

9.5. Tiger PSSIM Extension

The Tiger-PSSIM-Extension is an extension for simulating a Primärsystem (PS) in your tests.

It provides a wide range of BDD Steps and covers the majority of PS functionalities.

This is an internal gematik extension and can be found on GitLab <https://gitlab.prod.ccs.gematik.solutions/git/Testtools/tiger/tiger-pssim-extension>.

9.6. Tiger Robot Extension

The Tiger-Robot-Extension is an extension designed to control the Cardterminal Robot created by

the gematik.

This is an internal gematik extension and can be found on GitLab

<https://gitlab.prod.ccs.gematik.solutions/git/Testtools/tiger/tiger-robot-extension>.

Chapter 10. Rest API

The tiger test environment provides a Rest API which allows the automation of test execution and retrieval of the corresponding test results.

The Rest API is disabled by default.

You can enable it by setting the following configuration entry in the tiger.yaml

```
lib:  
  enableTestManagementRestApi: true # set to true to enable the Rest API. default is  
false
```

To enable the Rest API it is required to have the `cucumber-junit-platform-engine` in the class path.

E.g.:

```
<dependency>  
  <groupId>io.cucumber</groupId>  
  <artifactId>cucumber-junit-platform-engine</artifactId>  
  <version>7.19.0</version>  
  <scope>test</scope>  
</dependency>
```

This is transitively included if you are importing the `tiger-test-lib`.

But it is not included if you just import the `tiger-testenv-mgr`.

The following section provides an overview of the available endpoints and their functionality.
Additionally, the OpenApi specification yaml can be found in the [tiger repository](#).

10.1. Tests

getAvailableTests

`GET /tests`

get list of available tests

Description

gets the list of available tests that were discovered by the JUnit Platform.

Parameters

Return Type

array[*TestDescription*]

Content Type

- application/json

Responses

Table 1. HTTP Response Codes

Code	Message	Datatype
200	List of available tests	List[<i>TestDescription</i>]
400	Bad Request	<i>Error</i>
500	Something went wrong server internally	<i>Error</i>
0	The default error response	<i>Error</i>

getTestResults

GET /tests/runs/{testRunId}

get the results of a test run

Description

gets the results of a test run. This includes a global test result and the results for each test case. In case of failing tests, the failure message is also included.

Parameters

Path Parameters

Name	Description	Required	Default	Pattern
testRunId	testRunId for which to get the test results	X	null	

Return Type

TestExecutionResult

Content Type

- application/json

Responses

Table 2. HTTP Response Codes

Code	Message	Datatype
200	the test results for the given testRunId	<i>TestExecutionResult</i>
404	The specified resource was not found	<i>Error</i>

Code	Message	Datatype
400	Bad Request	<i>Error</i>
500	Something went wrong server internally	<i>Error</i>
0	The default error response	<i>Error</i>

postExecutionRequest

POST /tests/runs

request the execution of a selection of tests

Description

sends a request for the execution of a selection of tests. The tests to be executed are specified with a TestExecutionRequest which includes uniqueIds, tags and file paths. The execution request is queued for execution and will be executed as soon as any previously running tests finish. The response includes the id of the test run and the url where to find the results of the execution.

Parameters

Body Parameter

Name	Description	Required	Default	Pattern
TestExecutionRequest	test execution request <i>TestExecutionRequest</i>	X		

Return Type

TestExecutionInformation

Content Type

- application/json

Responses

Table 3. HTTP Response Codes

Code	Message	Datatype
202	Request was received and test run is started	<i>TestExecutionInformation</i>
404	The specified resource was not found	<i>Error</i>
400	Bad Request	<i>Error</i>
500	Something went wrong server internally	<i>Error</i>
0	The default error response	<i>Error</i>

postExecutionRequestAllTests

POST /tests/runs/all

request execution of all tests

Description

sends a request for the execution of all tests. The execution request is queued for execution and will be executed as soon as any previously running tests finish. The response includes the id of the test run and the url where to find the results of the execution.

Parameters

Return Type

TestExecutionInformation

Content Type

- application/json

Responses

Table 4. HTTP Response Codes

Code	Message	Datatype
202	Request was received and test run is started	<i>TestExecutionInformation</i>
404	The specified resource was not found	<i>Error</i>
400	Bad Request	<i>Error</i>
500	Something went wrong server internally	<i>Error</i>
0	The default error response	<i>Error</i>

10.2. Models

10.2.1. Error

Default error object with information about the occurred error

Field Name	Required	Nullable	Type	Description	Format
errorCode	X		String	A code identifying this error	
errorMessage	X		String	A readable message describing the error	

10.2.2. *ExecutionResult*

the result of an executed test

Field Name	Required	Nullable	Type	Description	Format
result	X		[String]		<i>Enum:</i> PENDING, RUNNING, SUCCESSFUL, ABORTED, FAILED,
failureMessage			String		

10.2.3. *TestDescription*

The description of a test case

Field Name	Required	Nullable	Type	Description	Format
uniqueId			String	unique identifier of the tests which is generated by the test platform.	
sourceFile			String	source file from where the test was discovered	
displayName			String	display name of the test	
tags			Set of [string]	tags associated with the test as specified in the feature file	

10.2.4. *TestExecutionInformation*

information of which tests were started and where to find the results of the test run

Field Name	Required	Nullable	Type	Description	Format
testRunId			UUID		uuid
resultUrl			URI		uri
testsToExecute			List of <i>TestDescription</i>		

10.2.5. *TestExecutionRequest*

Request the execution of a subset of tests. Multiple specifications will be combined with AND. If one of the tags, sourceFiles, or testUniqueIds is an empty list, it will not be considered in the selection of tests. If all are empty, all tests are selected.

Field Name	Required	Nullable	Type	Description	Format
tags			List of [string]		
sourceFiles			List of [string]		
testUniqueIds			List of [string]	the unique ids generated by the test platform. See the endpoint `GET /tests` for the available tests.	

10.2.6. *TestExecutionResult*

the result of the executed tests

Field Name	Required	Nullable	Type	Description	Format
testRunStarted			Date		date-time
testRunFinished			Date		date-time
result			<i>ExecutionResult</i>		
tests			List of [<i>TestExecutionResult_tests_inner</i>]		

10.2.7. *TestExecutionResultTestsInner*

Field Name	Required	Nullable	Type	Description	Format
test			<i>TestDescription</i>		
result			<i>ExecutionResult</i>		

Chapter 11. Links to test relevant topics

- 3-Amigos
 - presumably first mentioned in [George Dinwiddie's blog](#) (2009)
 - [John Ferguson's Blog about 3 Amigos](#)
- Cucumber
 - [Product website](#)
 - [Guru99's Intro to Gherkin](#)
 - [Cucumbers Gherkin reference](#)
- [Serenity BDD](#)
- SOLID
 - [Explaining all five concepts with simple Geometry](#)
 - [In depth discussion of the 5 principles](#)
- [Separation of concerns principle](#)
- Screenplay Pattern
 - [Nice overview of what the screenplay pattern is about](#)
 - [From Page Objects to SOLID Screenplay](#)
- FIRST principle for Unit tests
 - [AgileOtters Blog](#)

Chapter 12. Frequently asked questions

12.1. Maven

12.1.1. FM01 Which Serenity are we currently using?

You can find the Serenity compatible with each Tiger version in the [ReleaseNotes](ReleaseNotes.md)

12.1.2. FM02 When using maven, no tests are executed.

Please first make sure that either the surefire or failsafe plugin is enabled and shown as running in the console.

If you use Junit4 test annotations, you have to make sure that the junit vintage engine from the Junit5 library is included in the dependencies.

```
<dependency>
    <groupId>org.junit.vintage</groupId>
    <artifactId>junit-vintage-engine</artifactId>
    <version>${version.junit5}</version>
</dependency>
```

12.1.3. FM03 When running tests in Tiger, the test run aborts with a java.lang.NoSuchMethodError.

More specifically, the error is as follows:

```
Exception in thread 'main' java.lang.NoSuchMethodError: 'java.util.Set
org.json.JSONObject.keySet()'
```

This is due to a dependency conflict which may be solved by an exclusion in the tiger-test-lib:

```
<exclusion>
    <groupId>com.vaadin.external.google</groupId>
    <artifactId>android-json</artifactId>
</exclusion>
```

12.1.4. FM04 I don't see any log output, there are only warnings about outdated versions at the beginning

Apparently you included SLF4J V2 dependencies.

We currently use the logback classic 1.2.x branch, which is delivered in the most recent SpringBoot version.

This is not compatible to SLF4J 2.x.x.

12.1.5. FM05 There are selenium version conflicts when I want to run my project with SpringBoot and tiger with selenium

SpringBoot deploys an outdated selenium version.

To solve the conflicts, please use the versions stated in the release notes via dependency management in the maven pom.xml.

12.2. Extensions

12.2.1. FE01 When I start a docker image in tiger.yaml, the TestenvironmentManager's startup fails

Please make sure that you added the tiger-cloud-extension dependency in the most recent version.

```
<dependency>
    <groupId>de.gematik</groupId>
    <artifactId>tiger-cloud-extension</artifactId>
    <version>x.y.z</version>
</dependency>
```

12.2.2. FE02 When using the tiger-cloud-extensions, healthcheck at docker servertypes in Gematik SW factory fails

Tiger expects that Docker Daemon starts the container locally.

However, if this is not the case, you may use the environment variable TIGER_DOCKER_HOST to share on which server instance the container is started and the HealthcheckURL is adjusted accordingly.

For purposes of the Gematik SW-factory, the following code snippet is recommended for the pipeline script:

```
stage('Test') {
    environment {
        TIGER_DOCKER_HOST = dockerGetCurrentHostname()
    }
    steps {
        mavenVerify(POM_PATH)
    }
}
```

12.3. Workflow UI

12.3.1. FW01 In the workflow UI scenarios are listed twice and are refreshed the same time (as if they ran parallelly)

Usually, this only happens when the test suite is started in intellij and TigerCucumberListener is

delivered as a plugin in TigerCucumberListener.

This is no longer necessary since v1.3 because the listener is added automatically.

Due to this manual adjustment, two listeners are running that communicate the scenarios twice to the workflow UI.

If this happens in a mvn call, please check the tiger-maven-plugin configuration or the generated driver classes in terms of additional plugins in CucumberOptions.

12.3.2. FW02 After having pressed shutdown in the workflow UI, I cannot see messages in RbelLog Details Pane anymore

By stopping the test runs, the workflow UI backend is terminated as well.

You may recognize this by the light-red color of the side bar.

However, navigating in the RbelLog Details Pane requires a running backend.

In addition, RbelPath- and JEXL inspect dialogues are not working.

12.4. Other topics

12.4.1. FO01 How can I change the logging levels of loggers used by Tiger

Inside the tiger.yaml file, you can add a section logging.level: and add a list of packages / classes and the desired logging level.

```
logging:  
  level:  
    de.gematik.test.tiger.testenvmgr.TigerTestEnvMgr: TRACE  
    de.gematik.test.tiger.lib.TigerDirector: TRACE  
    de.gematik.test.tiger.proxy: TRACE  
    localTigerProxy: TRACE
```

12.4.2. FO02 Docker container creation fails

Use the command below to remove all unused containers.

Or look for containers starting with "tiger", stop and remove them.

```
docker system prune
```

Last resort:

```
netcfg -d
```

and restart docker

12.4.3. FO03 Adding alternative names programatically throws SSLEException

When using directly the method `de.gematik.test.tiger.proxy.TigerProxy.addAlternativeName()` to add multiple alternative names to the TLS certificate of the tiger proxy the following exception may come up:

```
12:17:48.604 [MockServer-EventLog13] ERROR o.mockserver.log.MockServerEventLog - 58165  
exception creating SSL context for serverfailed to set certificate and key  
javax.net.ssl.SSLException: failed to set certificate and key
```

The tiger proxy uses a mockserver internally which creates a SSLContext when handling the first request.

Adding additional names after the first request will not update the created SSLContext and the exception will be thrown.

A workaround for this behaviour is to explicitly restart the internal mockserver after adding an alternative name.

E.g.:

```
TigerProxy proxy = TigerDirector.getTigerTestEnvMgr().getLocalTigerProxyOrFail();  
proxy.addAlternativeName(host);  
proxy.restartMockserver();
```

12.4.4. FM01 What serenity do we use currently?

Das zu jeder Tiger Version kompatible Serenity findet ihr in den [ReleaseNotes](ReleaseNotes.md)

12.4.5. FM02 Bei der Nutzung von maven werden keine Tests ausgeführt

Bitte stell zuerst sicher, dass entweder das surefire oder das failsafe plugin aktiviert ist und auch in der Konsole als ausgeführt angezeigt wird.

Solltest Du Junit4 Test Annotationen verwenden so musst Du noch sicherstellen, dass die junit vintage engine aus der Junit5 Library in den dependencies mit angeführt ist.

```
<dependency>  
    <groupId>org.junit.vintage</groupId>  
    <artifactId>junit-vintage-engine</artifactId>  
    <version>${version.junit5}</version>  
</dependency>
```

12.4.6. FM03 Beim Ausführen von Tests im Tiger bricht der Testlauf mit einem java.lang.NoSuchMethodError ab

Genauer geht es um folgenden Fehler:

```
Exception in thread 'main' java.lang.NoSuchMethodError: 'java.util.Set org.json.JSONObject.keySet()'
```

Der Grund hierfür ist ein Dependency Konflikt und kann durch eine Exklusion in der tiger-test-lib dependency aufgelöst werden:

```
<exclusion>
  <groupId>com.vaadin.external.google</groupId>
  <artifactId>android-json</artifactId>
</exclusion>
```

12.4.7. FM04 Ich sehe keine Log-Ausgabe, lediglich am Anfang stehen Warnungen über veraltete Versionen

Du hast anscheinend Dependencies zu SLF4J V2 eingebunden.

Wir verwenden derzeit den logback classic 1.2.x branch, da dieser in der von uns verwendeten Spring Boot Version mitgeliefert wird.

Dieses ist NICHT kompatibel zu SLF4J 2.x.x!

12.4.8. FM05 Wenn ich in meinem Projekt Spring Boot und Tiger mit Selenium nutzen will, gibt es Versionskonflikte bei Selenium

Spring Boot liefert eine veraltete Version von Selenium aus.

Um die Konflikte zu lösen, bitte die in den ReleaseNotes angeführten Versionen über DependencyManagement im maven pom.xml lösen.

12.4.9. FM06 Wenn ich Scenario Outlines nutze, werden die Tests im junit Report nicht mit dem Namen des Scenarios angezeigt, sondern z.B. als Examples.Example #1.1.

Vermutlich ist die System Property `cucumber.junit-platform.naming-strategy.short.example-name` (meist im File `junit-platform.properties`) auf etwas anderes als "pickle" gesetzt.

Wenn diese Property nicht explizit gesetzt wurde, wird sie von Tiger automatisch auf "pickle" gesetzt.

12.5. Extensions

12.5.1. FE01 Wenn ich in der tiger.yaml ein Docker image starten will, so schlägt der Startup des TestenvironmentManagers fehl.

Stelle sicher, dass du die `tiger-cloud-extension` in der aktuellsten Version als dependency hinzugefügt hast.

```
<dependency>
  <groupId>de.gematik</groupId>
```

```

<artifactId>tiger-cloud-extension</artifactId>
<version>x.y.z</version>
</dependency>

```

12.5.2. FE02 Wenn ich die tiger-cloud-extensions nutze, schlägt der Healthcheck bei docker Servertypen in der Gematik SW Factory fehl

Normalerweise geht Tiger davon aus, dass der Docker Daemon die Container am lokalen Rechner startet.

Sollte dies nicht so sein, so kann man Tiger mit der Umgebungsvariable TIGER_DOCKER_HOST mitteilen, auf welchem Rechner die Container gestartet werden und die HealthcheckURL wird dementsprechend angepasst.

Für die Gematik SW-Factory empfiehlt sich folgendes Code Snippet für das Pipeline-Skript:

```

stage('Test') {
    environment {
        TIGER_DOCKER_HOST = dockerGetCurrentHostname()
    }
    steps {
        mavenVerify(POM_PATH)
    }
}

```

12.6. Workflow UI

12.6.1. FW01 In der Workflow UI sind die Szenarios doppelt aufgeführt und werden auch zeitgleich aktualisiert (es scheint, als ob sie parallel ablaufen)

Passiert eigentlich nur, wenn die Testsuite aus IntelliJ gestartet wurde und in der RuntimeConfiguration der TigerCucumberListener als plugin mitgegeben wird.

Dies ist seit v1.3 nicht mehr notwendig, weil der Listener automatisch hinzugefügt wird.

Durch den manuellen Eintrag laufen also dann zwei Listener, welche die Szenarien dann auch doppelt an die Workflow UI kommunizieren ...

Sollte dieser Effekt auch bei einem mvn call auftreten, dann bitte die Konfiguration des tiger-maven-plugins überprüfen, bzw. die generierten Treiberklassen bezüglich zusätzlicher Plugins in den CucumberOptions checken.

12.6.2. FW02 Nachdem ich auf Shutdown in der Workflow UI gedrückt habe, kann ich die Nachrichten in der RbelLog Details Pane nicht mehr ansehen

Durch das Beenden des Testlaufs ist das Backend der Workflow UI auch beendet worden.

Dies kannst Du auch daran erkennen, dass die linke Seitenleiste nun blass rot eingefärbt ist.

Das Navigieren in der RbelLog Details Pane benötigt aber das Backend und klappt daher zum jetzigen Zeitpunkt nicht mehr.

Auch die RbelPath- und JEXL Inspect Dialoge sind nicht mehr funktional.

12.7. Other topics

12.7.1. FO01 How can I change the logging levels of loggers used by Tiger

Inside the tiger.yaml file you can add a section logging.level: and add a list of packages / classes and the desired logging level.

```
logging:  
  level:  
    de.gematik.test.tiger.testenvmgr.TigerTestEnvMgr: TRACE  
    de.gematik.test.tiger.lib.TigerDirector: TRACE  
    de.gematik.test.tiger.proxy: TRACE  
    localTigerProxy: TRACE
```

12.7.2. FO02 Docker container creation fails

Use the command below to remove all unused containers.

Or look for containers starting with "tiger", stop and remove them.

```
docker system prune
```

Last resort:

```
netcfg -d
```

and restart docker

12.7.3. FO03 Adding alternative names programatically throws SSLEException

When using directly the method `de.gematik.test.tiger.proxy.TigerProxy.addAlternativeName()` to add multiple alternative names to the TLS certificate of the tiger proxy the following exception may come up:

```
12:17:48.604 [MockServer-EventLog13] ERROR o.mockserver.log.MockServerEventLog - 58165  
exception creating SSL context for serverfailed to set certificate and key  
javax.net.ssl.SSLException: failed to set certificate and key
```

The tiger proxy uses a mockserver internally which creates a SSLContext when handling the first request.

Adding additional names after the first request will not update the created SSLContext and the exception will be thrown.

A workaround for this behaviour is to explicitly restart the internal mockserver after adding an

alternative name.

E.g.:

```
TigerProxy proxy = TigerDirector.getTigerTestEnvMgr().getLocalTigerProxyOrFail();
proxy.addAlternativeName(host);
proxy.restartMockserver();
```