Tiger User Manual

TIGER@gematik.de

Version 0.19.1 - 2022-03-09

Contents

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, 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)

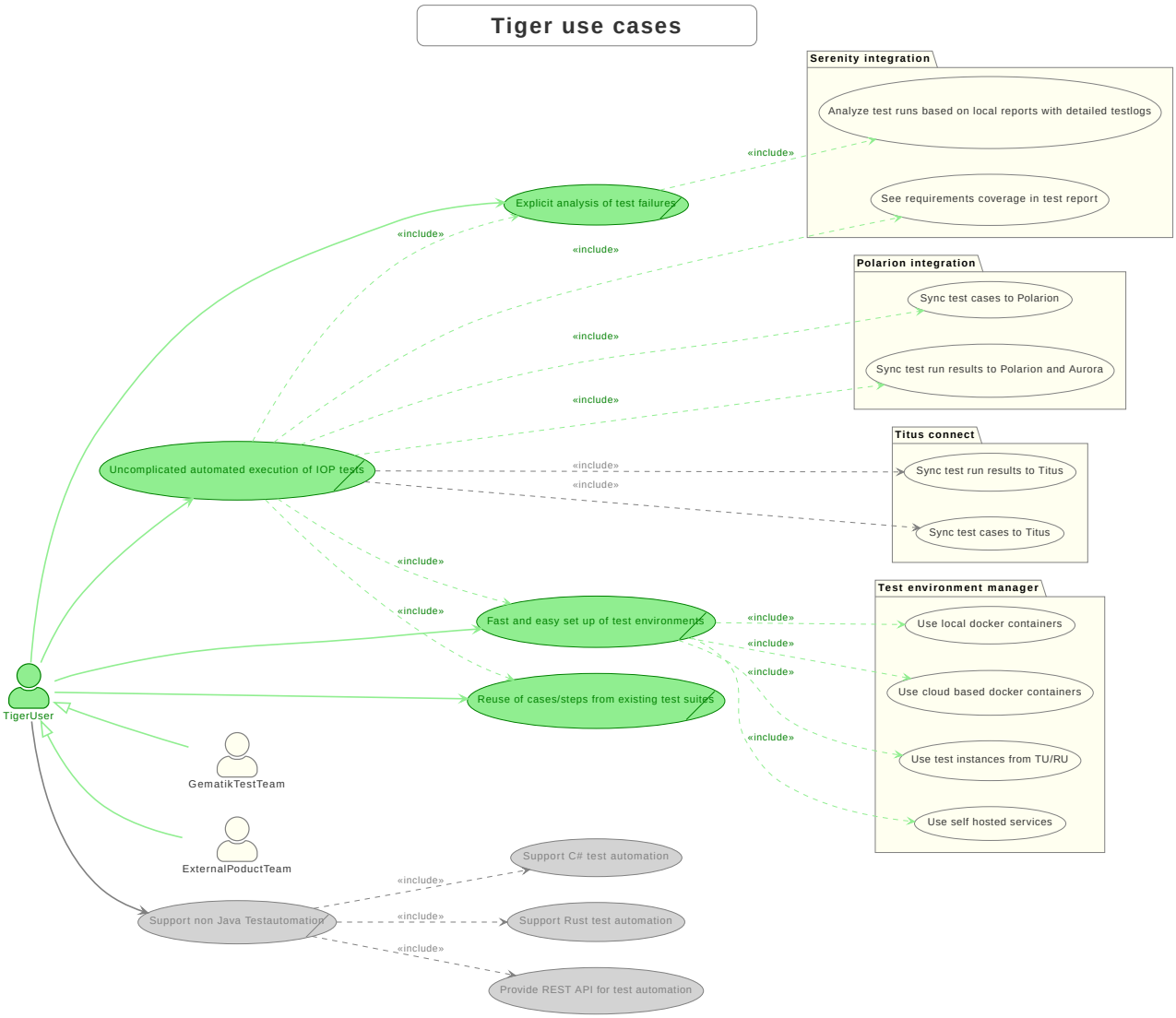

Figure 2. Tiger use cases

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, 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-testenv.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.

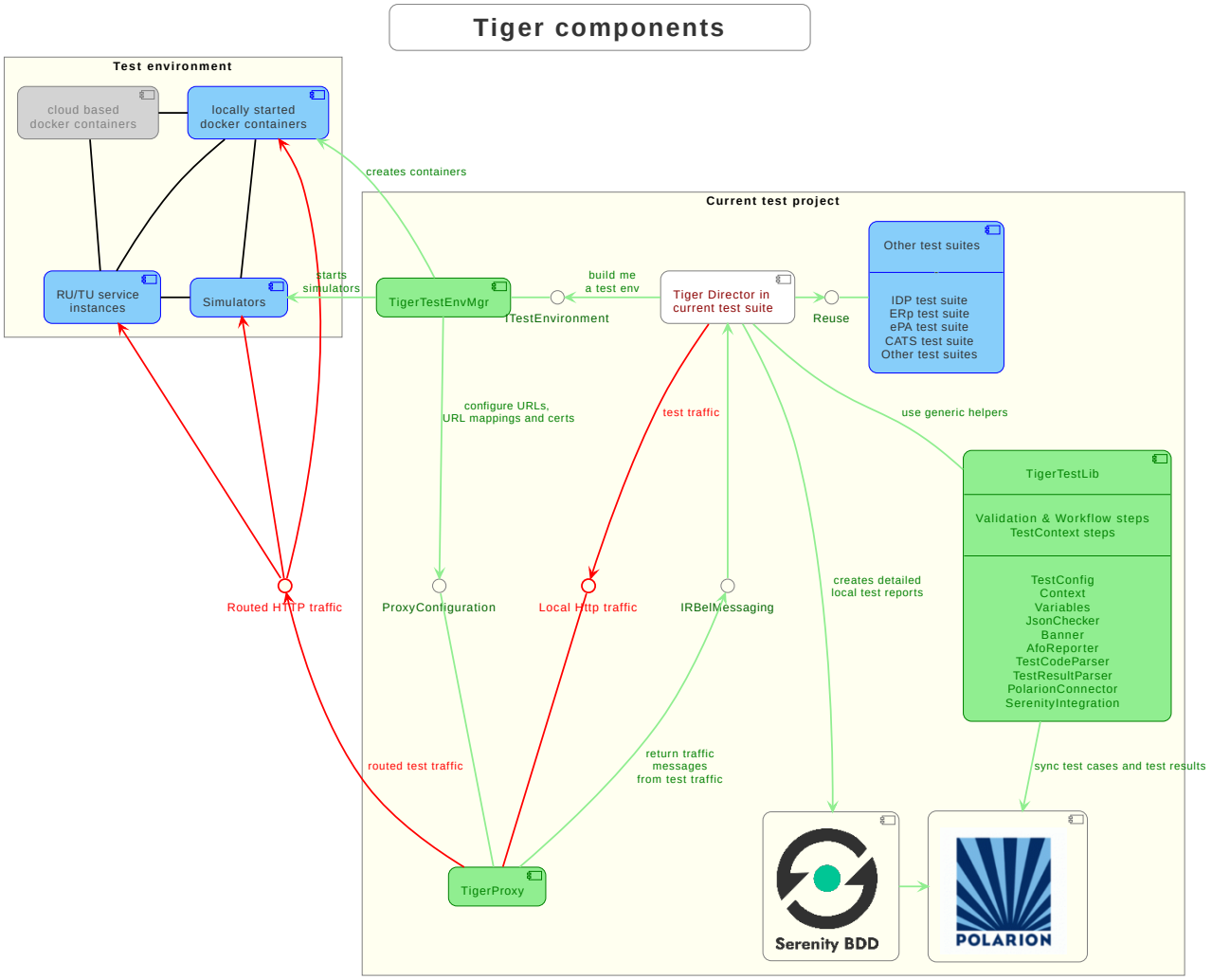

Figure 3. Tiger components

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

Operating system requirements

- Open JDK >= 11
- Maven >= 3.6
- IntelliJ >= 2021.2.3

On Windows please refrain from using Powershell or DOS command line windows but stick with GitBash

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 test library
- 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 driver generator plugin
- Maven Surefire plugin
- Serenity maven plugin

Listing 1. Simple Tiger Maven pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<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>1.0-SNAPSHOT</version>

    <properties>
        <maven.compiler.source>11</maven.compiler.source>
        <maven.compiler.target>11</maven.compiler.target>

        <!-- 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 gematik internal Nexus
          https://build.top.local/nexus/#nexus-search;quick~tiger-test-lib
        -->
        <version.tiger>0.19.1</version.tiger>
        <version.serenity.core>3.1.16</version.serenity.core>
        <version.serenity.maven.plugin>3.1.16</version.serenity.maven.plugin>
    </properties>

    <!-- tag::dependencies[] -->
    <dependencies>
        <dependency>
            <groupId>de.gematik.test</groupId>
            <artifactId>tiger-test-lib</artifactId>
            <version>${version.tiger}</version>
            <type>test-jar</type>
        </dependency>
        <dependency>
            <groupId>de.gematik.test</groupId>
            <artifactId>tiger-test-lib</artifactId>
            <version>${version.tiger}</version>
        </dependency>
        <!-- Optional if you have JUnit5 dependencies
        but use JUnit4 for your driver classes
        <dependency>
            <groupId>org.junit.vintage</groupId>
            <artifactId>junit-vintage-engine</artifactId>
            <version>5.8.2</version>
        </dependency>
        -->
        <!-- needed for the JUnit driver class @CucumberOptions annotation -->
        <dependency>
            <groupId>net.serenity-bdd</groupId>
            <artifactId>serenity-cucumber</artifactId>
            <version>${version.serenity.core}</version>
            <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>de.gematik.test</groupId>
                <artifactId>tiger-bdd-driver-generator-maven-plugin</artifactId>
                <version>${version.tiger}</version>
                <executions>
                    <execution>
                        <configuration>
                            <skip>false</skip>
                            <!-- optional, defaults to the templated located at
                            /src/main/resources/driverClassTemplate.jtmpl
                            in the tiger-driver-generator-maven-plugin module.
                            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 -->
                            <basedir>
${project.basedir}/src/test/resources/features</basedir>
                            <!-- mandatory -->
                            <includes>
                                <include>**/*.feature</include>
                            </includes>
                            <!-- mandatory -->
                            <glues>
                                <glue>de.gematik.test.tiger.hooks</glue>
                                <glue>de.gematik.test.tiger.glue</glue>
                                <!-- add your packages here -->
                            </glues>
                            <!-- optional -->
                            <driverPackage>
                                de.gematik.test.tiger.examples.bdd.drivers
                            </driverPackage>
                            <!-- optional -->
                            <!--suppress UnresolvedMavenProperty -->
                            <driverClassName>Parallel${ctr}IT</driverClassName>
                        </configuration>
                        <phase>generate-test-sources</phase>
                        <id>default-testSources</id>
```

```
                    <goals>
                        <goal>generate-drivers</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
        <!-- end::generator-plugin[] -->

        <!-- tag::surefire-plugin[] -->
        <!-- Runs the tests by calling the JUnit driver classes -->
        <!-- To filter features / scenarios use the system property
             -Dcucumber.filter.tags -->
        <plugin>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>3.0.0-M5</version>
            <configuration>
                <includes>
                    <!-- adapt to the class names of your driver classes -->
                    <include>**/Parallel*IT.java</include>
                </includes>
                <!-- on purpose, to ensure the serenity plugin is called
afterwards.
                It's check goal wil bail out in case of test failures,
                AFTER the serenity report has been generated -->
                <testFailureIgnore>true</testFailureIgnore>
            </configuration>
        </plugin>
        <!-- end::surefire-plugin[] -->

        <!-- tag::serenity-plugin[] -->
        <!-- Creates the SerenityBDD test report and
             fails the build if there were test failures -->
        <plugin>
            <groupId>net.serenity-bdd.maven.plugins</groupId>
            <artifactId>serenity-maven-plugin</artifactId>
            <version>${version.serenity.maven.plugin}</version>
            <dependencies>
                <dependency>
                    <groupId>net.serenity-bdd</groupId>
                    <artifactId>serenity-core</artifactId>
                    <version>${version.serenity.core}</version>
                </dependency>
                <!-- Optional if you want to also have a single HTML page mailable
report
                -->
                <dependency>
                    <groupId>net.serenity-bdd</groupId>
                    <artifactId>serenity-single-page-report</artifactId>
                    <version>${version.serenity.core}</version>
                </dependency>
            </dependencies>
```

```
                <!-- Optional if you want to also have a single HTML page mailable
report -->
                <configuration>
                    <reports>single-page-html</reports>
                </configuration>
                <executions>
                    <execution>
                        <id>serenity-reports</id>
                        <phase>post-integration-test</phase>
                        <goals>
                            <goal>aggregate</goal>
                            <goal>reports</goal>
                            <goal>check</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
            <!-- end::serenity-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 driver generator plugin

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

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>de.gematik.test</groupId>
                <artifactId>tiger-bdd-driver-generator-maven-plugin</artifactId>
                <version>${version.tiger}</version>
                <executions>
                    <execution>
                        <configuration>
                            <skip>false</skip>
                            <!-- optional, defaults to the templated located at
                            /src/main/resources/driverClassTemplate.jtmpl
                            in the tiger-driver-generator-maven-plugin module.
                            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 -->
                            <basedir>
${project.basedir}/src/test/resources/features</basedir>
                            <!-- mandatory -->
                            <includes>
                                <include>**/*.feature</include>
                            </includes>
                            <!-- mandatory -->
                            <glues>
                                <glue>de.gematik.test.tiger.hooks</glue>
                                <glue>de.gematik.test.tiger.glue</glue>
                                <!-- add your packages here -->
                            </glues>
                            <!-- optional -->
                            <driverPackage>
                                de.gematik.test.tiger.examples.bdd.drivers
                            </driverPackage>
                            <!-- optional -->
                            <!--suppress UnresolvedMavenProperty -->
                            <driverClassName>Parallel${ctr}IT</driverClassName>
                        </configuration>
                        <phase>generate-test-sources</phase>
                        <id>default-testSources</id>
                        <goals>
                            <goal>generate-drivers</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
```

Mandatory configuration properties

- **List[include] includes** (mandatory)
  list of include patterns for feature files in Ant format (directory/**.feature)
- **List[glue] glues** (mandatory)
  list of packages to be included as glue or hooks code

Optional configuration properties or properties with default values

- String basedir (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: Junit4SerenityTestDriver${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).
    - ${glues}
      comma separated list of glue/hook packages as specified by the glues configuration property.

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 io.cucumber.junit.CucumberOptions;
import net.serenitybdd.cucumber.CucumberWithSerenity;
import org.junit.runner.RunWith;

@RunWith(CucumberWithSerenity.class)
@CucumberOptions(
    features = {"src/test/resources/features/YOURFEATURE.feature"},
    plugin = {"json:target/cucumber-parallel/1.json"},
    monochrome = false,
    glue = {"de.gematik.test.tiger.hooks", "de.gematik.test.tiger.glue",
        "ANY ADDITIONAL PACKAGES containing GLUE or HOOKS code"})
public class Parallel1IT {

}
```

2.3.2. SureFire plugin

The surefire 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

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>
                <artifactId>maven-surefire-plugin</artifactId>
                <version>3.0.0-M5</version>
                <configuration>
                    <includes>
                        <!-- adapt to the class names of your driver classes -->
                        <include>**/Parallel*IT.java</include>
                    </includes>
                    <!-- on purpose, to ensure the serenity plugin is called
afterwards.
                    It's check goal wil bail out in case of test failures,
                    AFTER the serenity report has been generated -->
                    <testFailureIgnore>true</testFailureIgnore>
                </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. Optionally Failsafe plugin

We do NOT support the `maven-failsafe-plugin` at the moment.

Although the failsafe plugin is used quite often with spring boot applications under test to trigger start up / tear down of the app
via pre and post integration hooks, we found that the same
can be achieved utilizing spring annotated JUnit driver classes.

For more details please google @SpringBootTest, follow this link and/or check the `/src/test/jtmpl/SpringBootDriver.jtmpl file in the tiger-admin module.

2.3.4. Serenity Maven plugin

The `serenity-maven-plugin` has three goals, which will aggregate the data coming from the test results as XML and JSON, create the HTML reports and
fail the maven job if any test run via the surefire plugin failed.

```
            <!-- Creates the SerenityBDD test report and
                 fails the build if there were test failures -->
            <plugin>
                <groupId>net.serenity-bdd.maven.plugins</groupId>
                <artifactId>serenity-maven-plugin</artifactId>
                <version>${version.serenity.maven.plugin}</version>
                <dependencies>
                    <dependency>
                        <groupId>net.serenity-bdd</groupId>
                        <artifactId>serenity-core</artifactId>
                        <version>${version.serenity.core}</version>
                    </dependency>
                    <!-- Optional if you want to also have a single HTML page mailable
report
                    -->
                    <dependency>
                        <groupId>net.serenity-bdd</groupId>
                        <artifactId>serenity-single-page-report</artifactId>
                        <version>${version.serenity.core}</version>
                    </dependency>
                </dependencies>
                <!-- Optional if you want to also have a single HTML page mailable
report -->
                <configuration>
                    <reports>single-page-html</reports>
                </configuration>
                <executions>
                    <execution>
                        <id>serenity-reports</id>
                        <phase>post-integration-test</phase>
                        <goals>
                            <goal>aggregate</goal>
                            <goal>reports</goal>
                            <goal>check</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

- embedds 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-testenv.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.

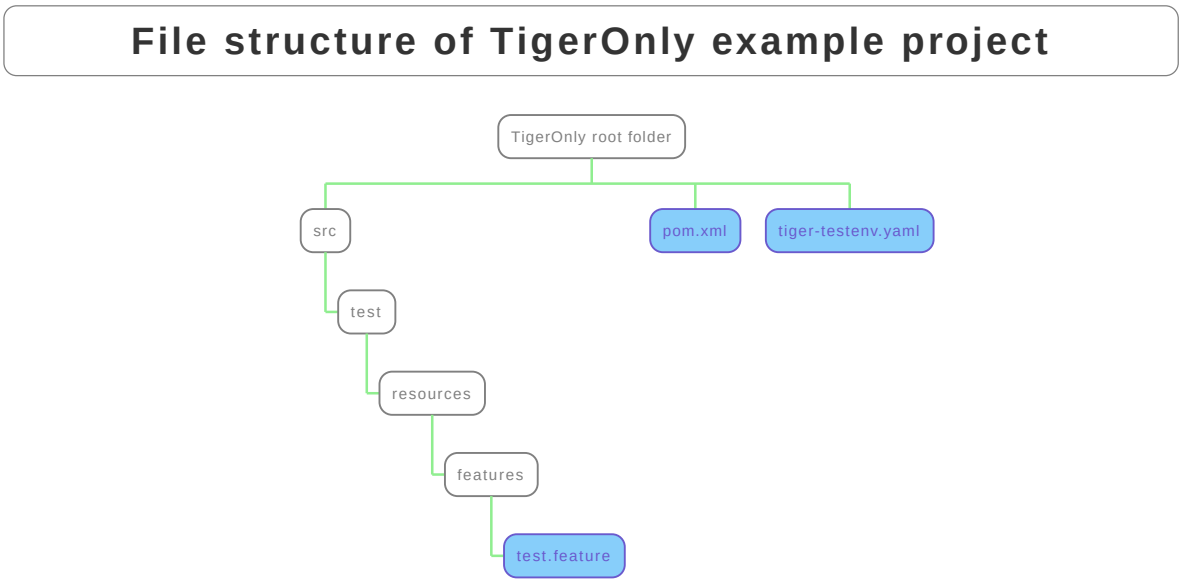

Figure 4. File structure of TigerOnly example project

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: net.serenitybdd.cucumber.cli.Main
- Glue:
  - de.gematik.test.tiger.glue

    - de.gematik.test.tiger.hooks
    - 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.

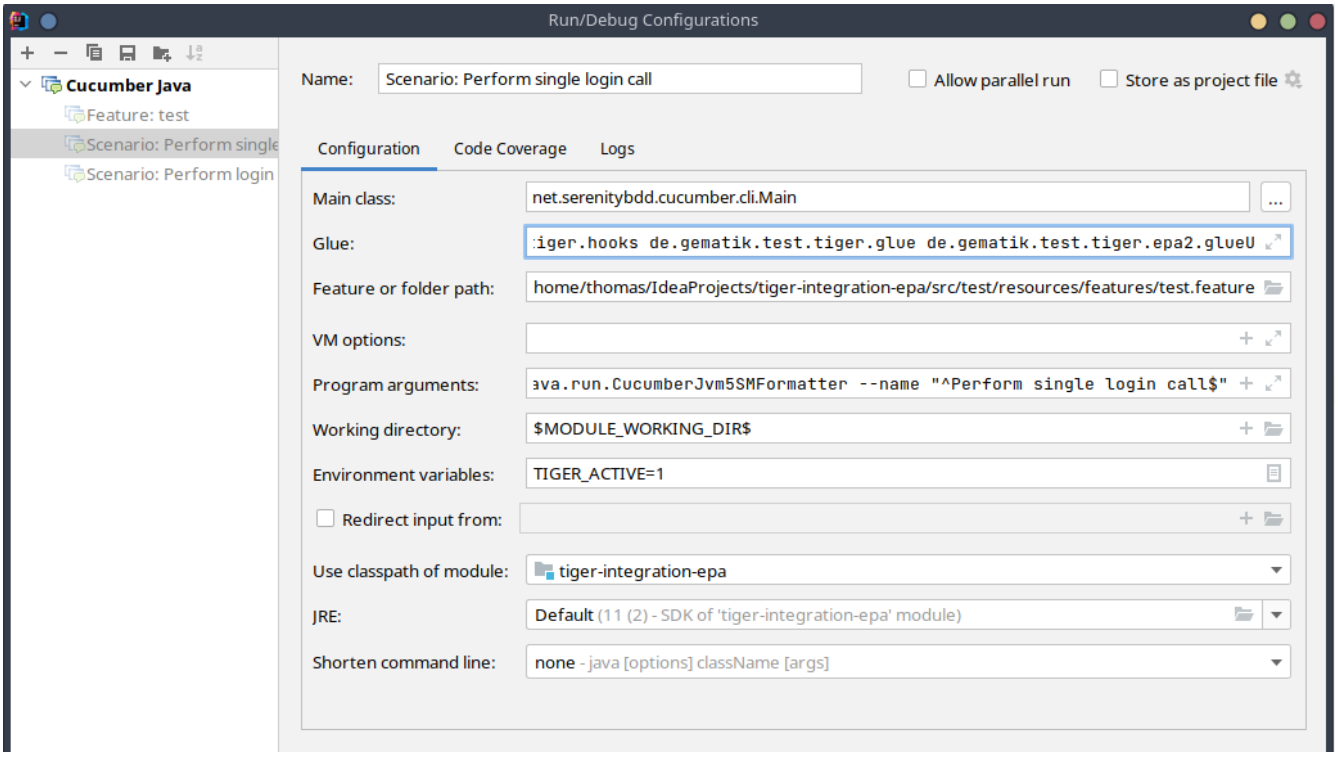

Figure 5. Run/Debug settings for IntelliJ

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-testenv.yaml configuration file and initialize the local Tiger Proxy for the test suite.
To choose a different test environmental configuration file you may set the environment variable TIGER_TESTENV_CFGFILE.

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

If the environment variable is not set, then the test environment manager first checks for a file named tiger-testenv-${COMPUTERNAME}.yaml, and if this also does not exist, it searches for a file named tiger-testenv.yaml.
If none of these files exist it will fail the start up.

In the startup 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 either compose the tiger-testenv.yaml file manually or much preferred
use the Tiger Admin UI.

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

3.1. Supported server nodes and their configuration

The Tiger testenviroment manager currently supports the following list of server ndoes.

- **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.
- **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 of 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.

3.1.1. YAML configuration files in a nutshell

Before you start writing your own tiger-testenv.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.

3.1.2. General properties

The general properties apply to each node type.

Listing 4. General properties

```
serverKey_xxx:
  # OPTIONAL host name of this node when accessing it via the test suite
  # (rerouted via the local test suite Tiger Proxy).
  # Defaults to the server key (serverKey_xxx)
  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
  # 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
  # type specific property for Tiger Proxy and docker container nodes
  version: string
  # OPTIONAL list of pki certs and keys to initialize
  # the local Tiger Proxy of the test suite with
  # Default value is empty array
  # For more details see "PKI configuration" section below
  pkiKeys: []
  # OPTIONAL type specific list of environmental variable assignments to be used
  # when starting the server node.
  # Each entry has to have the format ENVVARNAME=VALUE
  # Has NO EFFECT on external Url nodes.
  # Default value is empty array
  environment:
    - ENVVAR1=VALUE1
    - ENVVAR2=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
```

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 5. Type specific properties

```
# type specific sub structure for external jar, external url and Tiger Proxy nodes
externalJarOptions:
  # used by external jar and Tiger Proxy nodes
  workingDir: string
  # used by all node types
  healthcheck: string
  # only used by external jar nodes
  options: []
  # used by external jar and Tiger Proxy nodes
  arguments: []

# type specific sub structure for Tiger Proxy nodes
tigerProxyCfg:
  serverPort: int
  proxiedServer: string
  proxiedServerProtocol: [HTTP|HTTPS]
  # Here a normal Tiger Proxy configuration can be used.
  # This is explained in more depth down below
  proxyCfg:
    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:
  # only used by docker compose nodes
  serviceHealthchecks: []
  # all properties below only used by docker container nodes
  proxied: boolean
  oneShot: boolean
  entryPoint: string
```

The configuration of the Tiger Proxy is explained in detail in the section Configuring the local test suite Tiger Proxy

3.1.3. PKI configuration in pkiKeys

The pkiKeys property contains a list of certificates and keys to be provided to the local Tiger Proxy of the test suite.
Each entry has to provide a unique id, type and pem property.

Listing 6. PKI configuration

```
pkiKeys:
    # MANDATORY unique key/certificate id
  - id: disc_sig
    # MANDATORY one of [Certificate|Key]
    type: Certificate
    # MANDATORY base64 encoded multiline string representing the certificate / key.
    pem: "MIICsTCCAligAwIBAgIHA61I5ACUjTAKBggqhkjOPQQDAjCBhDELMAkGA1UEBhMC
REUxHzAdBgNVBAoMFmdlbWF0aWsgR21iSCBOT1QtVkFMSUQxMjAwBgNVBAsMKUtv
.....
xiKK4dW1R7MD334OpOPTFjeEhIVV"
  - id: disc_enc
    type: Key
    pem: "ISUADOGBESBXEZOBXWEDHBXOU..."
```

3.1.4. 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)

3.1.5. 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.

To customize the docker container you may alter the entry point command line and add the Tiger Proxy certificate to the container's operating system list of trusted certificates.
For containers that should exit after a single command you may enable the oneShot property.

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

  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

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

3.1.6. 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.

Listing 8. Docker compose configuration

```
  type: compose
  dependsUpon: csv string
  startupTimeoutSec: int

  # 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
  dockerOptions:
    # MANDATORY list of URLs to check for a successful start up
    # of the docker compose's services
    serviceHealthchecks:
      - http://service1:8001/
      - https://service2:9001/
```

Listing 9. Demis docker compose example

```
demis_001:
  type: compose
  source:
    - classpath:/de/gematik/test/tiger/testenvmgr/demis/demis_localhost.yml
  startupTimeoutSec: 180
  dockerOptions:
    serviceHealthchecks:
      - http://notification-gateway:9042
      - http://notification-portal:9041/welcome
```

3.1.7. 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.

```
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

  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.
    workingDir: /home/user/test/myspecificjar
    # 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.
    healthcheck: http://127.0.0.1:8080
    # 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
  pkiKeys: []
  environment: []
  urlMappings: []
  exports: []
```

3.1.8. 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

  externalJarOptions:
    # 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.
    healthcheck: https://www.medizin.de/healthyState.jsp

  # The following properties are explained in the General properties section above
  pkiKeys: []
  # IGNORE for this type as it has no effect
  environment: []
  urlMappings: []
  exports: []
```

3.1.9. 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 startup time can be minimized, and it is always guaranteed to start the current version.

Listing 12. Tiger Proxy configuration

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

  tigerProxyCfg:
    # 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.
    serverPort: 8080
    # OPTIONAL server key of the node this proxy shall be used as reverse proxy for.
    # If set the routes in the proxyCfg section 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
    proxyProtocol: http
    # configures the proxy itself. For more details
    # please check the chapter about the local test suite Tiger Proxy below
    proxyCfg:
      ...
      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
  pkiKeys: []
  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. 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
```

3.2.1. 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.

3.2.2. 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.

3.2.3. 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 dont worry, it is just the server that uses them.

3.2.4. Local ePA2 reference nodes

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

3.2.5. 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.

3.2.6. Local KonSim node

This template provides a Konnektor simulation (as external jar).
See https://wiki.gematik.de/display/PTP/KonSim for more information.

3.2.7. Local ePA FdV Sim

This template provides FdV simulation, usable for ePA.

3.2.8. Local DEMIS reference nodes

This template provides the DEMIS Meldeportal as local docker compose.

3.3. 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-testenv.yaml file.
For more information about what the Tiger Proxy and how it works see the chapter Tiger Proxy

```
# Flag whether to activate the local Tiger Proxy
# Default value is true
localProxyActive: true

# the block where all the Tiger Proxy configuration properties are located
tigerProxy:
  # the port under which the server will be booted
  port: 7777
  # logLevel of the proxy-server. DBEUG and TRACE will print traffic, so use with
care!
  proxyLogLevel: TRACE
  # section to configure whether and where the proxy should dump
  # a traffic HTML report on shutdown
  fileSaveInfo:
    # should the cleartext http-traffic be logged to a file?
    writeToFile: true
    # configure the file name
    filename: "foobar.tgr"
    # default false
    clearFileOnBoot: true
  # 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 true)
      activateRbelLogging: false

  # 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.
    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
  # 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
  # This will share the WebUI-Resources (various CSS-files) from the Tiger Proxy
  # locally, thus enabling usage when no internet connection exists
  localResources: 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.
    serverSslSuites:
      - "TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA"

    # 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 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:
  - .

  # 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"
```

3.4. 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 short time, you might want to keep your test environment running and not shut it down after each run.
To do so, you need to create a customized test environment configuration file (call it tiger-testenv-standalone.yaml,
set the env var TIGER_TESTENV_CFGFILE accordingly) containing all the server nodes needed and deactivate the local Tiger Proxy in this configuration file.
Now download the
tiger test environment jar-with-all-dependencies file v0.17.1
from maven or go to maven index folder to choose a more recent version.

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-testenv-standalone.yaml
java -jar tiger-testenv-mgr-${VERSION}-jar-with-all-dependencies.jar
```

Now before starting your test suite scenarios you need to

- disable / remove the test nodes in your default tiger-testenv.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-testenv-standalone.yaml to enable a persistent test environment during the development of test suite scenarios
- tiger-testenv-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-testenv.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.

3.5. Using Environment variables and system properties

3.5.1. 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 nodesare 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. Excurse: What are proxies, reverse, forward

There are a lot of different kind 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 has two types: Tiger Proxy and Tiger Standalone Proxy. Both types have a proxy-port (configurable via `tigerProxy.port`),
which supports both http- and https-traffic (so you do not have to differentiate between the two).

The Tiger Standalone Proxy has an admin-port on top (configurable via `server.port`). This provides a webui to monitor the traffic,
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 (eg. http://foobar), which defines a forward-proxy route, or relative (eg. /blub), defining a reverse-proxy-route. Please note: You can freely add parts (eg. http://foobar/extra/part) to further specify the mapping.

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. 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.

4.3.4. basicAuth

You can add optional basic-auth configuration which will be added to the forwarded message. Theoretically this could also be done via modifications, but this a more convenient approach.

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

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 a 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 default CA used by the Tiger Proxy can be found here: https://github.com/gematik/app-Tiger/blob/master/tiger-standalone-proxy/src/main/resources/CertificateAuthorityCertificate.pem

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.

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

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 fullfilled 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 forwards 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.
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).

```
tigerProxy:
    proxyId: IBM
    trafficEndpoints:
      - http://another.tiger.proxy:<serverPort>
    # 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)
```

Please be advised to use the server-port (`server.port`) here, not the proxy-port (`tigerProxy.port`). The traffic from routes with `activateRbelLogging: false` 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`. 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
    },
    "reveiver": {
      "hostname": "hostname/ip address of receiver",
      "port": portAsInt
    }
}
```

4.7. 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:

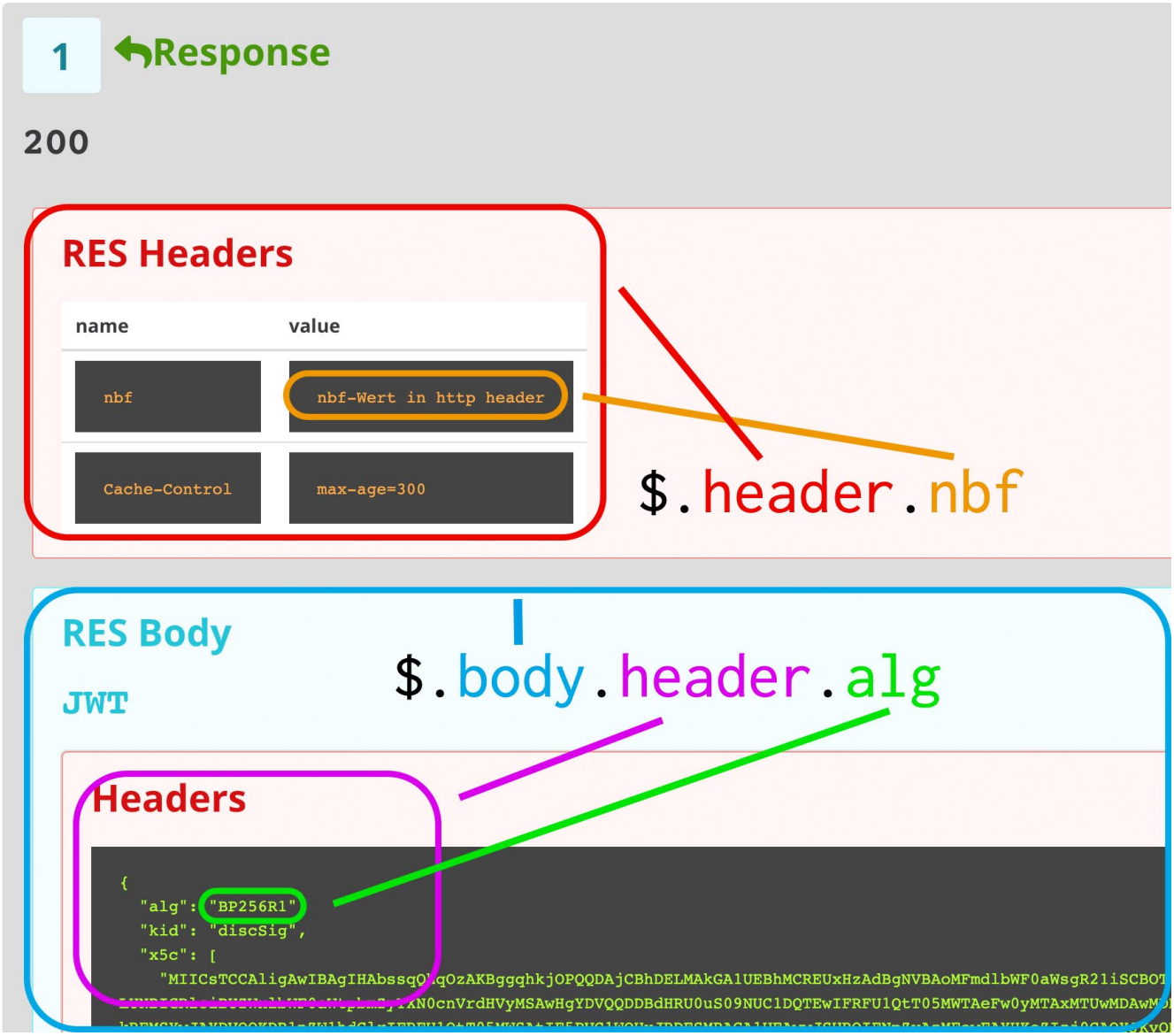

Figure 6. Rbel-Path expression in a HTTP-Response

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.)

Figure 7. Multiple body references

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

```
$.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).

4.7.1. 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: https://commons.apache.org/proper/commons-jexl/reference/syntax.html

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
- `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.
- `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`.

4.7.2. 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:

```
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 three
    .printContent(true) // should the content of each element be printed?
    .build()
    .execute();
```

4.8. 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 fat 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
```

4.9. Additional configuration

There are some additional configuration-flags in the tiger-proxy:

4.9.1. Performance

Below some properties along with their respective default values:

```
tigerProxy:
    activateRbelParsing: true
    activateAsn1Parsing: false
    activateVauAnalysis: false
```

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 loose all benefit of performing Rbel-Analysis.

activateAsn1Parsing

This is off by default. ASN.1 objects are very common in crypto applications. While parsing them will enable you to directly have a look inside certificates it comes with a penalty in performance and also clutters the object-tree. Often it's enough to know that there is a certificate, only in some scenarios is the content of interest. If the latter is of interest to you activate ASN.1 parsing.

activateVauAnalysis

VAU-Analysis adds information about the current session to every single VAU-message. If you are not trying to analyze EPA-VAU messages leave this option turned off. If you do enabling it will give you additional information about the messages.

Chapter 5. Tiger Test library

As outlined in the overview section the Tiger test libarary 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.

```
rbelPathDebugging: false
# 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
rbelAnsiColors: true
# 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.
activateMonitorUI: false
# Flag whether to start a small Java Swing UI to display the current steps / banner
messages
# 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
```

5.2. Cucumber and TigerTestHooks

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

That's why it is mandatory to add the package de.gematik.test.tiger.hooks to the glue packages list.

The TigerTestHooks 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.

The @Before method calls the TigerDirector once
to initiate the Tiger test environment manager, the local Tiger Proxy and optionally the monitoring UI and parses the current scenario / feature file.
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 the @Before method also registers a Restassured request filter,
parses the details and adds a curl Command details button to the Serenity BDD report.
The curl command shown in this section allows to repeat the performed REST request, for manual test failure analysis.

The @After method saves all collected RbelMessages as HTML file to the `target/rbellogs` folder, attaches it to the SerenityBDD report as test evidence and logs the current test run state (scucess/failed rate) to the console.

The TigerDirector in the future will become a general high level access class to all the major features of the Tiger test framework.
Such things as syncing test cases and test reports with Polarion, creating requirements coverage reports,
using the Tiger test framework from non Cucumber test drivers, ...etc.

For now, the TigerDirector is only for internal use and does not provide any implementation of the aforementioned features.

5.2.1. How to use Tiger in a non Cucumber scenario

If you are using TestNG, plain JUnit or other Test harness frameworks, you have to ensure that you call the four methods defined
in the TigerTestHooks class at the appropriate times.

- The @Before method must be called before each Test scenario
- The @BeforeStep method MIGHT be called before each test step
- The @After method MIGHT be called after each Test scenario

If you provide a dummy Scenario instance with dummy steps matching your test code to the method you might use all methods.
Else you just pass in a null Scenario to the methods or do not call the other methods in your test run.

But take note that this will not pop up UI Workflow banner messages and will not save RBelMessage HTML files.
As a workaround you might use the TigerDirector to manually trigger UI Monitor popups via its method `updateStepInMonitor(Step step)`.

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 matching given value in request
- Find first / next matching request
- Clear all recorded messages
- Specify timeout for filtering request

With the `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).

Listing 13. Tiger validation filtering steps example

```
Feature Tiger validation steps

  Scenario: Example steps

    # clear all previousely recorded messages (requests and responses)
    Given TGR clear recorded messages
    # restrict filtering requests to a specific host
    And TGR filter requests based on host "testnode.example.org"
    # restrict filtering requests to a specific HTTP method
    And TGR filter requests based on method "POST"
    # specify timeout for filter queries
    And TGR set request wait timeout to 20 seconds
    # find the first request matching criteria
    When TGR find request to path "/path/blabla" with "$..tag.value.text" matching
"abc.*"
    # validate response to the filtered request using RBelPath
    Then TGR current response with attribute "$..answer.result.text" matches "OK.*"
    # find the next request matching criteria
    When TGR find next request to path "/path" with "$..value.text" matching "abc.*"

    ....
```

5.3.2. Validating responses

Core features

- Assert that the boty of the resposne matches regex

- Assert that a given RBelPath node matches 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 14. Tiger response validation steps example

```
Feature Tiger validation steps

  Scenario: Example steps
    ...
    # find the first request matching criteria
    When TGR find request to path "/path/blabla" with "$..tag.value.text" matching
"abc.*"
    # validate response to the filtered request using RBelPath
    Then TGR current response with attribute "$..answer.result.text" matches "OK.*"
    # find the next request matching criteria
    When TGR find next request to path "/path" with "$..value.text" matching "abc.*"
    # validate response to the filtered request comparing body content
    Then TGR current response body matches
    """
         body content
    """
    # validate response to the filtered request based upon JSONChecker
    And TGR current response at "$..tag" matches as JSON
    """
      {
        "arr1": [
          "asso", "bsso"
        ]
      }
    """
    # validate response to the filtered request based upon XML comparison
    And TGR current response at "$..tag" matches as XML
    """
      <arr1>
        <entry index="1">asso</entry>
        <entry index="2">bsso&</entry>
      </arr1>
    """
```

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
- "txtignoreempty" 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.

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 to ignore 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.
Also 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 15. 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 $NULL, meaning this value of this key is equal to null.
- Usage of regular expression (e.g. ".*" and "[\\d]*") to match values.
- Usage of "__" preceeding a json key.- This indicates that the entry is optional but if it exists it must match the given value.
- if a value is specified as "${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.

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.

5.3.3. Complete set of steps in validation glue code

```
// copied from module /tiger-test-lib
// /src/test/java/de/gematik/test/tiger/glue/RBelValidatorGlue.java

/**
 * Specify the amount of seconds Tiger should wait when filtering for
 * requests / responses
 * before reporting them as not found.
```

```
 */
@Gegebensei("TGR setze Anfrage Timeout auf {int} Sekunden")
@Given("TGR set request wait timeout to {int} seconds")

/**
 * clear all validatable rbel messages. This does not clear the recorded messages
later on
 * reported via the rbel log HTML page or the messages shown on web ui of Tiger
Proxies.
 */
@Wenn("TGR lösche aufgezeichnete Nachrichten")
@When("TGR clear recorded messages")

/**
 * filter all subsequent findRequest steps for hostname. To reset set host name to
 * empty string "".
 *
 * @param hostname host name (regex supported) to filter for
 */
@Wenn("TGR filtere Anfragen nach Server {string}")
@When("TGR filter requests based on host {string}")

/**
 * filter all subsequent findRequest steps for method.
 *
 * @param method method to filter for
 */
@Wenn("TGR filtere Anfragen nach HTTP Methode {string}")
@When("TGR filter requests based on method {string}")

/**
 * reset filter for method for subsequent findRequest steps.
 */
@Wenn("TGR lösche den gesetzten HTTP Methodenfilter")
@When("TGR reset request method filter")

/**
 * find the first request where the path equals or matches as regex and memorize it
 * in the {@link #rbelValidator} instance
 *
 * @param path path to match
 */
@Wenn("TGR finde die erste Anfrage mit Pfad {string}")
@When("TGR find request to path {string}")

/**
 * find the first request where path and node value equal or match as regex and
memorize it
 * in the {@link #rbelValidator} instance.
 *
 * @param path     path to match
```

```
 * @param rbelPath rbel path to node/attribute
 * @param value    value to match at given node/attribute
 */
@Wenn("TGR finde die erste Anfrage mit Pfad {string} und Knoten {string} der mit
{string} übereinstimmt")
@When("TGR find request to path {string} with {string} matching {string}")

/**
 * find the NEXT request where the path equals or matches as regex and memorize it
 * in the {@link #rbelValidator} instance.
 *
 * @param path path to match
 */
@Wenn("TGR finde die nächste Anfrage mit dem Pfad {string}")
@When("TGR find next request to path {string}")

/**
 * find the NEXT request where path and node value equal or match as regex and
memorize it
 * in the {@link #rbelValidator} instance.
 *
 * @param path     path to match
 * @param rbelPath rbel path to node/attribute
 * @param value    value to match at given node/attribute
 */
@Wenn("TGR finde die nächste Anfrage mit Pfad {string} und Knoten {string} der mit
{string} übereinstimmt")
@When("TGR find next request to path {string} with {string} matching {string}")

/**
 * assert that there is any message with given rbel path node/attribute matching given
value.
 * The result (request or response) will not be stored in the {@link #rbelValidator}
 * instance.
 *
 * @param rbelPath rbel path to node/attribute
 * @param value    value to match at given node/attribute
 * @deprecated
 */
@Wenn("TGR finde eine Nachricht mit Knoten {string} der mit {string} übereinstimmt")
@When("TGR any message with attribute {string} matches {string}")

//
=============================================================================
====
//
//    S T O R E   R E S P O N S E   N O D E   I N   C O N T E X T
//
//
=============================================================================
====
```

```
/**
 * store given rbel path node/attribute text value of curren tresponse.
 *
 * @param rbelPath path to node/attribute
 * @param varName  name of variable to store the node text value in
 */
@Dann("TGR speichere Wert des Knotens {string} der aktuellen Antwort in der Variable
{string}")
@Then("TGR store current response node text value at {string} in variable {string}")

//
================================================================================
====
//
//    R E S P O N S E   V A L I D A T I O N
//
//
================================================================================
====

/**
 * assert that response body of filtered request matches.
 *
 * @param docString value / regex that should equal or match
 */
@Dann("TGR prüfe aktuelle Antwort stimmt im Body überein mit:")
@Then("TGR current response body matches")

/**
 * assert that response of filtered request matches at given rbel path node/attribute.
 *
 * @param rbelPath path to node/attribute
 * @param value    value / regex that should equal or match as string content with
MultiLine
 *                 and DotAll regex option
 */
@Dann("TGR prüfe aktuelle Antwort stimmt im Knoten {string} überein mit {string}")
@Then("TGR current response with attribute {string} matches {string}")

/**
 * assert that response of filtered request matches at given rbel path node/attribute.
 *
 * @param rbelPath  path to node/attribute
 * @param docString value / regex that should equal or match as string content with
MultiLine
 *                  and DotAll regex option supplied as DocString
 */
@Dann("TGR prüfe aktuelle Antwort im Knoten {string} stimmt überein mit:")
@Then("TGR current response at {string} matches")
```

```
/**
 * assert that response of filtered request matches at given rbel path node/attribute.
 *
 * @param rbelPath path to node/attribute
 * @param value    value / regex that should equal or match as string content with
MultiLine
 *                 and DotAll regex option
 * @deprecated
 */
@Then("TGR current response at {string} matches {string}")

/**
 * assert that response of filtered request matches at given rbel path node/attribute
 * assuming its JSON or XML
 *
 * @param rbelPath     path to node/attribute
 * @param mode         one of JSON|XML
 * @param oracleDocStr value / regex that should equal or match as JSON or XML content
 * @see JsonChecker#assertJsonObjectShouldMatchOrContainInAnyOrder(String, String,
boolean)
 */
@Dann("TGR prüfe aktuelle Antwort im Knoten stimmt als {word} überein mit:")
@Then("TGR current response at {string} matches as {word}")

/**
 * assert that response of filtered request matches at given rbel path node/attribute
 * assuming its XML with given list of diff options.
 *
 * @param rbelPath       path to node/attribute
 * @param diffOptionsCSV a csv separated list of diff option identifiers to be applied
 *                       to comparison of the two XML sources
 *                       <ul>
 *                           <li>nocomment ... {@link
DiffBuilder#ignoreComments()}</li>
 *                           <li>
 *                             txtignoreempty ...
 *                             {@link  DiffBuilder#ignoreElementContentWhitespace()}
 *                           </li>
 *                           <li>txttrim ... {@link
DiffBuilder#ignoreWhitespace()}</li>
 *                           <li>
 *                             txtnormalize ... {@link
DiffBuilder#normalizeWhitespace()}
 *                           </li>
 *                       </ul>
 * @param xmlDocStr      value / regex that should equal or match as JSON content
 * @see <a href="https://github.com/xmlunit/user-guide/wiki/DifferenceEvaluator">
 *        More on DifferenceEvaluator
 *      </a>
 */
@Dann("TGR prüfe aktuelle Antwort im Knoten {string} stimmt als XML mit folgenden diff
```

```
Optionen {string} überein mit:")
@Then("TGR current response at {string} matches as XML and diff options {string}")
```

5.3.4. 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.

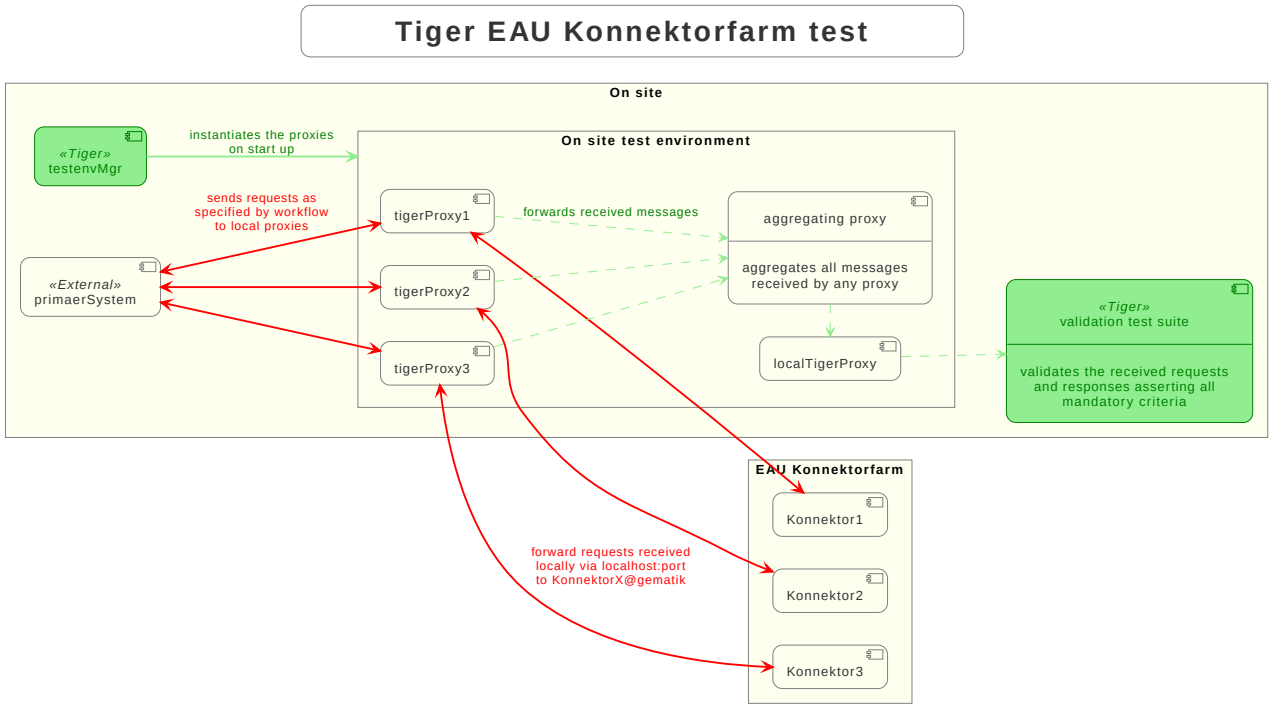

Figure 8. Tiger EAU Konnektorfarm test environment

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.

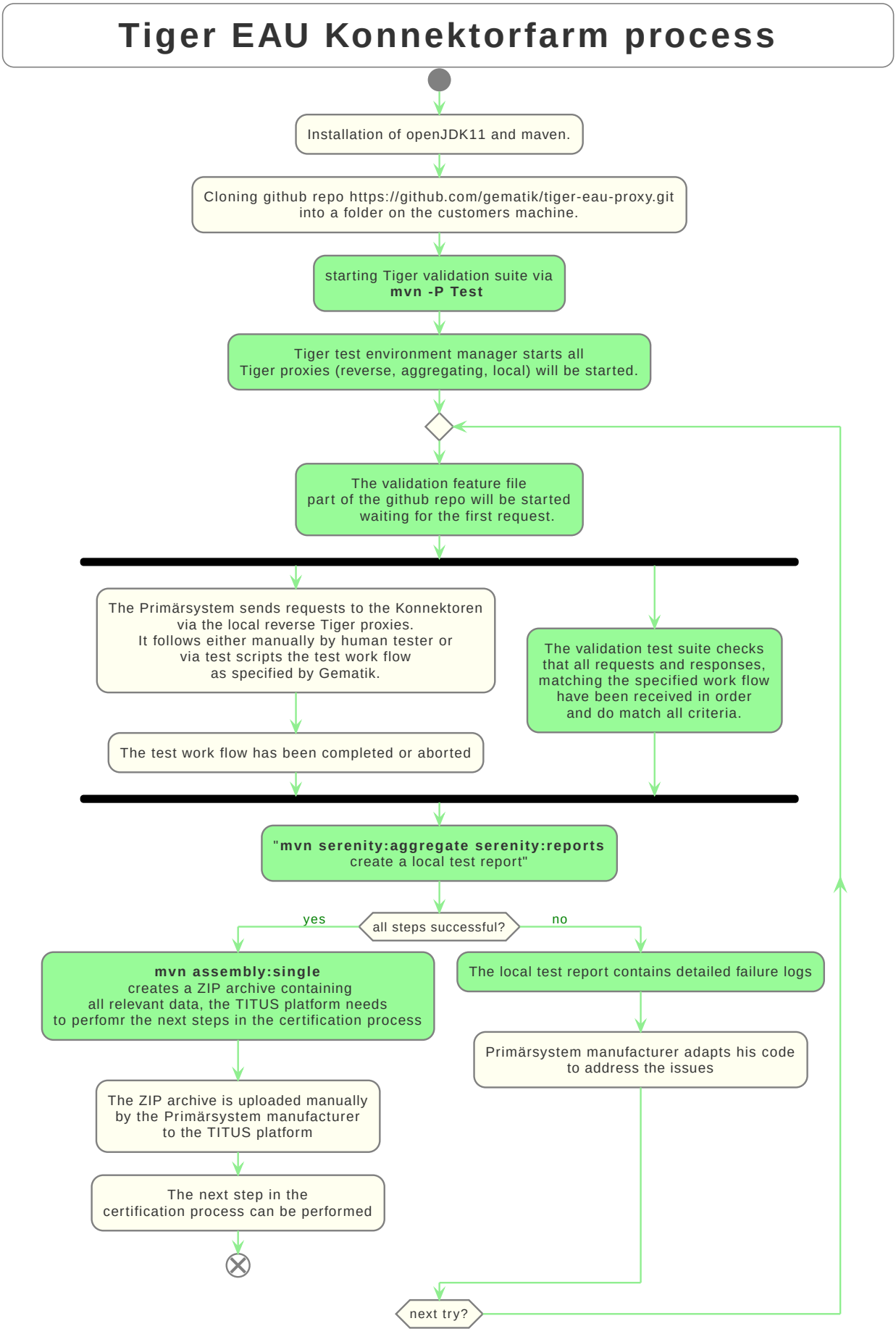

Figure 9. Tiger EAU Konnektorfarm process

5.3.5. Workflow UI

The Workflow UI is one of the new experimental features which are currently introduced to Tiger.If activated via the tiger.yaml configuration file (see Tiger test lib configuration), any TGR banner step will be displayed in the Monitor UI popup 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.

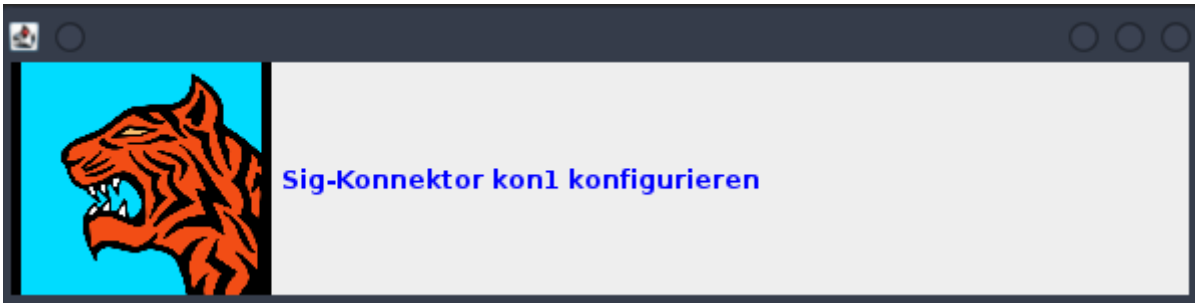

Figure 10. Workflow UI popup

Listing 16. Current message steps for Workflow UI

```
// copied from module /tiger-test-lib
// /src/test/java/de/gematik/test/tiger/glue/TigerGlue.java

    @Gegebensei("TGR zeige {word} Banner {string}")
    @Given("TGR show {word} banner {string}")

    @Gegebensei("TGR zeige {word} Text {string}")
    @Given("TGR show {word} text {string}")

    @Gegebensei("TGR zeige Banner {string}")
    @Given("TGR show banner {string}")

    @When("TGR wait for user abort")
    @Wenn("TGR warte auf Abbruch")
```

The last step allows to pause the validation test suite and is mainly used in demo scenarios allowing the manual tester
to perform demo transactions that will be logged and saved to HTML reports but are nto validated.

5.4. 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

5.4.1. 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.
For more details please check the code and its usages in the Tiger test framework.

5.4.2. 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 REST
- Serenity Screenplay REST

5.5. Test library configuration

To configure your test library add a tiger.yaml file to the project root.
The following configuration properties are supported:

```
# flag whether to activate monitoring UI
activateMonitorUI: false
# flag whether to activate rbel path debugging
rbelPathDebugging: false
# flag whether to print rbel dumps to console using ANSI color codes or not
rbelAnsiColors: true
# flag whether to add a curl command details button to SerenityRest Restassured calls
# in the Serenity BDD report
addCurlCommandsForRaCallsToReport: true
```

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.

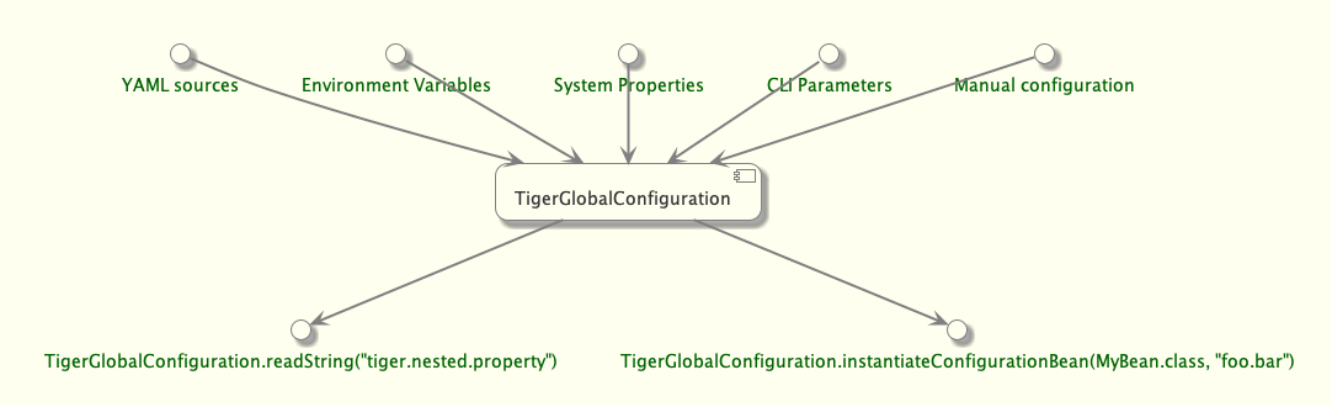

Figure 11. The TigerGlobalConfiguration with inlets and outlets

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):

- Manually added properties (TigerGlobalConfiguration.putString("foo.bar")). Here there are three sub-distinctions:
  - Global properties
  - Test properties
  - Thread-based properties
- Command-line properties
- System-Properties (System.setProperty)
- Environment-Variables (export "FOO_BAR" = 42)
- YAML-Files (tiger-testenv.yaml)

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.

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", SourceType.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 use 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 17. 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
    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}
      healthcheck: http://127.0.0.1:19307
```

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

6.5. Examples

Some examples to clarify:

6.5.1. 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 18. tiger-testenv.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"
```

6.5.2. Example 2

In the above example let's say you only want to customize the port. This can be done by using placeholders:

Listing 19. tiger-testenv.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"
```

6.5.3. 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 20. 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.

6.6. 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.

Chapter 7. Tiger User interfaces

7.1. Admin UI

The Tiger Admin UI is a separate module that can be downloaded from maven

https://repo1.maven.org/maven2/de/gematik/test/tiger-admin

Choose a current version and download the `tiger-admin-x.y.z.jar` to a local folder on your computer.

Run the following command to get the Tiger Admin UI up and running:

```
java -jar tiger-admin-x.y.z.jar --server.port=8080
```

Now open your firefox browser and go to http://127.0.0.1:8080.

Voila you are ready to start working with the Tiger Admin UI

7.2. Workflow UI

The Workflow UI is currently an experimental feature implemented as PoC.
We plan to have this feature in product grade quality in the first half of 2022.
Till then there is not much more than in the Workflow UI section above.

7.3. Standalone Tiger Proxy UI

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 `SERVERPORT` being the configured server port of the Tiger Proxy.

7.3.1. Bottom menu bar

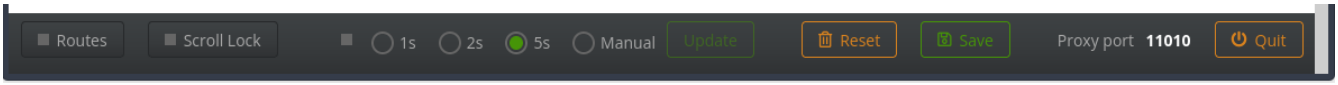

Figure 12. Tiger Proxy bottom menu bar

The displayed buttons can trigger the following actions:

- **Routes** ... allows you to modify the routes configured on this Tiger Proxy
- **Scroll Lock** ... allows you to lock the scroll position. Incoming messages wills till be added to the list at the bottom of the page.

- **Update buttons** allow you to specify how often the list shall be updated. The Update button is only active if you have choosen manual update mode. To the left of the 1s choice there is a small LED that will indicate active retrieval of new data (green) or failure while retrieving data (red).
- **Reset** … allwos you to delete all recorded messages so far. This will delete all messages on the Tiger Proxy!
- **Save** … allows you to save the current HTML page.
- **Quit** … allows you to quit the Tiger Proxy instance

Chapter 8. Links to test relevant topics

- 3-Amigos
  - presumably first mentioned in George Dinwiddie's blog (2009)
  - John Ferguson's Blog about 3 Amigos
  - Becky Carter's blog
- Cucumber
  - Product web site
  - 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 9. FAQs

9.1. 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