Earn your stripes - Tiger Workshop (vers. 2)

Einführung

Der Workshop soll einen kleinen Blick in Tiger hinein ermöglichen. Tiger ist ein Testframework. Der Fokus und die Stärke ist das **Interface-Blackbox-Testing** von Komponenten. Aufgrund seiner flexiblen Anlage ermöglicht Tiger damit eine Vielzahl an Testszenarien und Einsatzmöglichkeiten.

Um diese Einführung so praxisnah und einfach wie möglich zu gestalten wird hier beispielhaft eine kleine Testsuite fürs das Abtesten einer externen Webseite gebaut. In vielen kleine Schritten, sogenannten **Stages**, werden verschiedene Techniken von Tiger vorgestellt. Der Teilnehmer soll, allein oder in einer Gruppe, zunehmend kompliziertere Techniken einsetzen, um das System abzutesten. Wer einmal nicht mitkommt, kann für die nächste Stage einfach den entsprechenden Branch auschecken und weitermachen. Abgerundet wird der Aufbau durch **Zusatzaufgaben** (für die ganz Fleißigen).

Zur Einführung gibt es noch eine **Pre-Stage**, welche einen Einstieg in das Test-Projekt bietet. Der Fokus hier ist mehr maven & serenity und weniger Tiger. Der eigentlich Workshop beginnt daher NACH der Pre-Stage.

Pre-Stage: Hello World

Ziel ist es ein Maven-Projekt zu erstellen, Tiger mit möglichst der neuesten Version in die Pom einzubauen, ein feature-File zu schreiben mit einem einfachen Banner-anzeigen. Nutzt Java 17.

- Lege ein neues Projekt an, z.B. "tiger-workshop".
- In der Pom als dependency die tiger-test-lib gerne mit der neuesten Tigerversion eintragen

```
            <dependency>
          <groupId>de.gematik.test</groupId>
          <artifactId>tiger-test-lib</artifactId>
          <version>3.7.1</version>
      </dependency>
```

- Das ganze soll mit "`mvn verify`" abgeprüft werden.
- Des weiteren serenity-cucumber und serenity-core als dependency mit aufnehmen, des weiteren wird noch die logback-classic gebraucht, für die serenity Version nehmt 4.2.3:

```
      <properties>
        <maven.compiler.source>17</maven.compiler.source>
        <maven.compiler.target>17</maven.compiler.target>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <tiger.version>3.7.1</tiger.version>
        <serenity.version>4.2.16</serenity.version>
        <version.logback>1.5.17</version.logback>
    </properties>
    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>ch.qos.logback</groupId>
                <artifactId>logback-classic</artifactId>
                <version>${version.logback}</version>
            </dependency>
        </dependencies>
    </dependencyManagement>

    <dependencies>
        <dependency>
            <groupId>de.gematik.test</groupId>
            <artifactId>tiger-test-lib</artifactId>
            <version>${tiger.version}</version>
        </dependency>
        <dependency>
            <groupId>net.serenity-bdd</groupId>
            <artifactId>serenity-cucumber</artifactId>
            <version>${serenity.version}</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>net.serenity-bdd</groupId>
            <artifactId>serenity-core</artifactId>
            <version>${serenity.version}</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
```

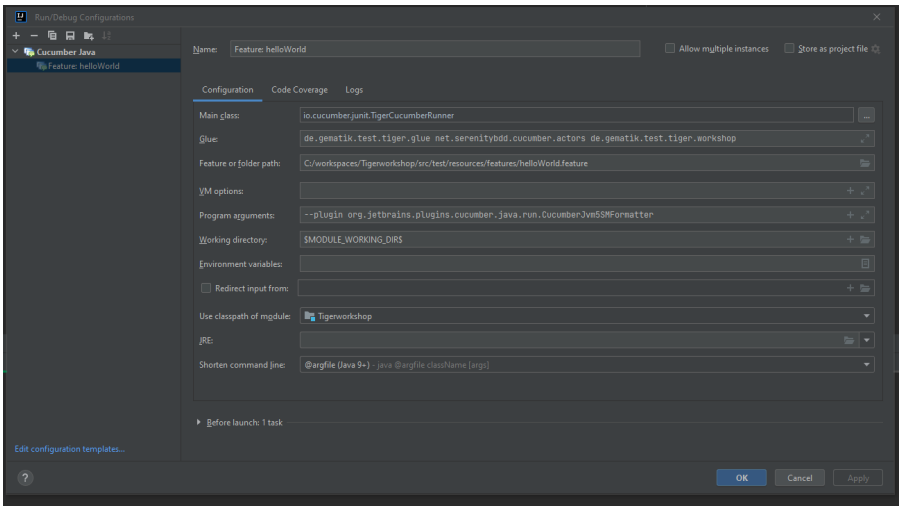
- Lege unter `src/test/resources/features` eine neue `helloWorld.feature` Datei an
- In der Datei soll folgendes eingetragen werden:

```
  Feature: Hello World
Scenario: is everything up and running
  Given TGR show banner "Hello World!"
```

- Im Rootverzeichnis erstellt ihr eine vorerst leere `tiger.yaml` Datei.
- Das Ganze soll mit "`mvn verify`" abgeprüft werden.

Ausführen in der IDE (hier: IntelliJ, prinzipiell kann jede IDE verwendet werden, auch eine reine Ausführung auf der Console ist möglich)

- Eine neue Run-Configuration für Cucumber-Java anlegen
  - Main Class auf *io.cucumber.junit.TigerCucumberRunner* abändern
  - als Glue folgende Pakete eintragen
    - de.gematik.test.tiger.glue
    - net.serenitybdd.cucumber.actors
    - de.gematik.test.tiger.workshop
  - Program arguments eintragen
    - --plugin org.jetbrains.plugins.cucumber.java.run.CucumberJvm5SMFormatter
  - Für Windows-User wichtig: die Kommandozeile verkürzen @argfile (Java 9+)



Hinweis: Geht man auf "Edit configuration templates..." unten links im Bild, kann man dort die Defaultwerte für MainClass, GlueCode, etc eintragen. So muss man die Run Configuration nicht jedes Mal neu aufsetzen, wenn man z.B. ein neues Feature File laufen lassen will.

Für IntelliJ ist es sinnvoll das "Gherkin" und das "Cucumber for Java" Plugin zu installieren.

Serenity BDD Exkurs

Serenity BDD besteht in der Regel aus sogenannten .feature Files, welche die Tests enthalten und GlueCode, welcher die Implementierung der Tests enthält. Der Clou dabei ist, dass diese Tests auch von Managern und Nicht-Softwareentwickler verstanden werden, da sie in fast natürlicher Sprache geschrieben werden. Und zwar derart:

.feature File

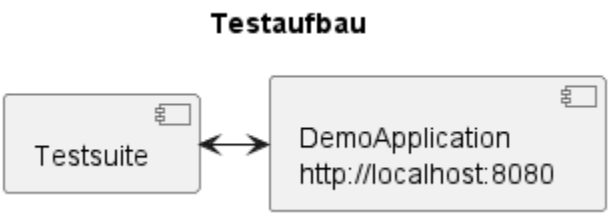
```
  Feature: Hello World
Scenario: is everything up and running
  Given TGR show banner "Hello World!"
      Then XYZ should be checked
```

Der GlueCode ist in einer oder mehreren Javaklassen, die sich im `test` Package befinden und die eigentliche Implementierung von z.B. "XYZ should be checked" enthalten. Die Wörter "Given", "Then", "When", "and" oder resp. "Gegeben", "Wenn", "Dann", "Und" haben KEINE tiefere Bedeutung für die Ausführung des Testfalls, außer dass sie sich gut in die Testfallbeschreibung fügen.

Stage 1: Healthcheck ans System

Hier findet ihr den Tiger-Workshop, wo ihr euch das Projekt auschecken könnt. Es gibt für jeden Stage einen Branch, so dass ihr jederzeit diesen auschecken könnt und weitermachen könnt, auch wenn ihr die letzte Stage nicht mitgemacht habt.

Für den Tigerworkshop haben wir folgenden Testaufbau: Unser Testprojekt kommuniziert mit einem einfachen Webserver und stellt Anfragen an das System. Der Webserver ist für uns eine Blackbox, dessen ServiceController wir mit seinen Restschnittstellen benutzen wollen. Der Tiger startet zuvor die DemoApplication als external Jar.



Erstellt in der `tiger.yaml` einen neuen Server mit Namen "`demo`" vom Typ "`externalJar`". Das Demo-Jar liegt im Projekt. Gebt die Source in der `tiger.yaml` entsprechend mit local: Path-Zum-Jar an. Die healthcheck-URL ist `http://localhost:8080/service/status` (Bei Schwierigkeiten kannst du hier und hier Hilfe bekommen).

Um bei Fehlern ein schnelles Entwicklungserlebnis zu haben, kannst du `startupTimeoutSec` auf 5 setzen (Der Default ist 20, der Testfall bricht dann schneller ab, wenn du etwas falsch gemacht hast).

Die DemoApplication wird dann unter http://localhost:8080/ erreichbar sein, die OpenApi unter http://localhost:8080/swagger-ui/index.html.

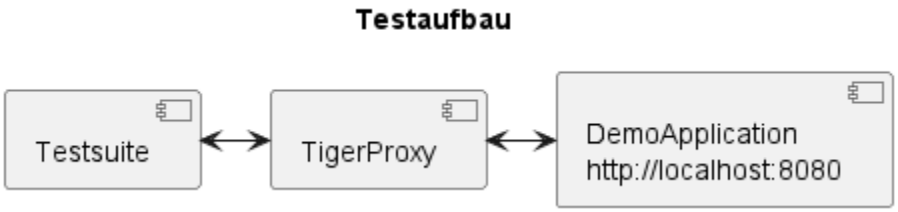
Jetzt wollen wir eine REST Anfrage an diese URL schicken und die Antworten auswerten.

Dafür erstellen wir einen Gluecode Step, wo wir z.B. über SerenityRest die URL (z.B http://localhost:8080/service/status) abfragen. Daran denken, dass das Verzeichnis mit in den GlueCode bei der Run Configuration mit aufgenommen werden muss.

Stage 2: Tiger Proxy

Bislang haben wir den Server direkt angesprochen. Nun werden wir den Tiger-Proxy nutzen. Der Tiger-Proxy ist ein Proxy-Server, welcher den Verkehr einfach weiterleitet und für uns später in der Testsuite zur Verfügung stellt. Dabei kann er mehrere Quellen vereinigen, vielfältige Strukturen parsen und Daten strukturiert zur Verfügung stellen. Am Schluss kann er für uns eine übersichtliche Datei bereitstellen, in welcher der gesamte Netzwerkverkehr aufbereitet nachlesbar ist.

In einer normalen Tiger-Testumgebung wird immer ein Tiger-Proxy gestartet, der lokale Tiger-Proxy. Der Traffic, der über den lokalen Tiger-Proxy läuft, steht für uns direkt in der Testsuite zur Verfügung. Weiterhin wird er beim Starten als Default-Proxy gesetzt, wodurch er schon in der vorigen Stage die gesamte Kommunikation mitgeschnitten und weitergeleitet hat. Beim Start wird außerdem für jeden Server, der Teil der Testumgebung ist, eine Route gesetzt. Domain-Name ist hierbei der Servername aus der `tiger.yaml`.



Wir wollen nun den Tiger-Proxy explizit verwenden, den aufgezeichneten Traffic anschauen und auf diesem direkt Assertions durchführen.

- Dazu ändern wir zunächst den REST-Aufruf im Gluecode: Von `http://localhost:8080` zu http://demo/
  - *Dadurch abstrahieren wir von dem konkreten Server. Die Testsuite weiß nicht mehr, welcher Demoserver selbst angesprochen wird. Das macht ein nachträgliches Austauschen einfacher.*
- Neuen Schritt in der Testsuite einfügen `TGR find first request to path ""` (Den Pfad entsprechend ergänzen. **Wichtig:** Hier ist nur der Pfad einzutragen, nicht die gesamte URL. Hostname und Protokoll können also entfallen)
  - *Dieser Schritt verschiebt den Zeiger des aktuellen Nachrichtenpaars, welches in der Testsuite betrachtet wird. "current request" und "current response" nutzen immer diesen Zeiger. Neue empfangene Nachrichten verschieben den Zeiger NICHT automatisch! Mehr dazu in der nächsten Stage.*
- ```
  TGR current response body matches:
  """
  OK
  """
  ```
  - *Dies vergleicht den HTTP-Body der current Response mit dem gegebenem String. Dies kann ein exakter Match oder ein Regex sein.*

Tipp: Tiger hat viele vordefinierte Gluecode Anweisungen, die auch ständig erweitert werden. Bitte schaut euch bei Interesse im Tigerprojekt die vorhandenen Klassen an.

Debug-Tipp: Im Verzeichnis `target/rbellogs` stehen nun die Aufzeichnungen des Verkehrs bereit, der über den Tiger-Proxy geleitet wurde. Dieser ist sauber nach Szenarien getrennt und eine gute Hilfe, wenn mal etwas nicht auf Anhieb funktioniert (Sehen wir dort überhaupt den Verkehr? Wie sehen die Nachrichten aus? Was ist die Fehlermeldung des Servers?)

Stage 3: Current request/response

Intern merkt sich die Testsuite alle Nachrichten, die der Tiger-Proxy auffängt. In diesen Nachrichten gibt es wiederum einen Zeiger, der "current request /response" Zeiger. Er kann nur durch Suchfunktionen bewegt werden (bewegt sich also NICHT durch das Auffangen einer neuen Nachricht automatisch weiter). Um gut mit den Testschritten der Art "current request matches" arbeiten zu können müssen wir diesen Zeiger korrekt bewegen. Das können wir mit den Testschritten "`TGR find first request to path {}`", "`TGR find next request to path {}`" und "`TGR find last request to path {}`" erledigen.

- Baue vor "I control the health endpoint" einen weiteren Schritt ein, der eine Abfrage gegen `/service/hello` schickt
  - *Mit zwei verschiedenen Anfragen können wir sicherstellen, dass wir die richtige Antwort kontrollieren.*
- Ändere das Suchen nach der Nachricht: `When TGR find first request to path "/service/.*"`
  - *Führe jetzt den Test einmal aus: Er wird fehlschlagen. Zuerst wird die hello Abfrage gefunden, die einen anderen Inhalt als der Health Check Call hat. Das gilt natürlich nicht wenn deine Abfrage nicht generisch war.*

- Nun führen wir zuerst den *Health Check Call* vor der hello Abfrage aus: Checke, ob das Attribute "$.body" gleich "OK" ist.
  - *Ausführen nicht vergessen. Der Test sollte jetzt erfolgreich sein.*

- Würden wir vorherigen Schritt nicht machen wollen, können wir mit: `When TGR find next request to path "/service/.*"` und/oder mit `When TGR find last request to path "/service/.*"` die korrekte Request ermitteln
  - *Dies sucht die Nachricht ausgehend von der aktuellen Position des Zeigers, nicht vom Anfang der Nachrichten.*

- den gesuchten Request finden und die Response vergleichen

Da der "current request/response" Zeiger auch über Scenarios hinweg agiert, ist es sinnvoll, zu Beginn eines Scenarios den Schritt "`TGR clear recorded messages`" auszuführen und die vorangegangenen Nachrichten zu löschen.

Zusatzaufgabe
  ```

- Schreibe ein neues Feature Login "/service/performLogin" mit username und password als QueryParameter
- Teste den ResponseCode auf 400

Stage 4: Workflow UI/Web UI/Rbel-Path

Jetzt wollen wir weitere REST Anfragen an die URL schicken und uns die Workflow UI anschauen. Die Workflow UI und auch die WebUI ist ein Frontend, welches gerade beim Aufsetzen der Tests sehr hilfreich sein kann.

Dafür setzen wir in der tiger.yaml folgenden Eintrag auf true:

```
lib:
  activateWorkflowUi: true
  trafficVisualization: true
```

Nach dem Starten des Scenarios oder des Featurefiles öffnet sich die Workflow UI im Defaultbrowser. Erklärungen zur Workflow UI gibt es hier und hier im Manual. Erklärungen zur WebUI sind hier zu finden.

RBel-Path ist eine an XPath/JsonPath angelehnte Abfragesprache. Mit ihr kann man in beliebige, mit dem Tiger-Proxy aufgefangene Nachrichten hineinschauen. Der entscheidende Unterschied ist, dass die Struktur unabhängig vom eigentlichen Aufbau her aufgedröselt wird. Das Root-Element ist in unserem Fällen hier immer direkt die HTTP-Nachricht. Mit "`$.body`" erhalten wir also den HTTP-body, mit "`$.header`" die Header-Struktur. Wenn die Nachricht eine HTML-Seite wäre, könnten wir mit "`$.body.html.header`" das Header-Tag der HTML-Struktur auswählen. Wir bauen hier ein einfaches Beispiel, mit dem wir in die JSON-Struktur der Rückgabe des Testtreibers hineinschauen. Mehr über Rbel-Path kannst du hier nachlesen.

- Neuen Testschritt hinzufügen: `TGR print current response as rbel-tree`
  - *Dieser Schritt gibt die aktuell ausgewählte Antwort in der Kommandozeile als Baumstruktur aus*
- Kurzes Schauen auf den Baum in der Konsole. Zum Vergleich kann man sich den RBel-Log daneben legen (zu finden unter target/rbellogs).
- Nun ersetzen oder ergänzen wir die Vergleiche aus der vorigen Stage. Zuerst den Login
  - `Then TGR current response with attribute "$.body.status" matches "400"`
  - Mehr Infos zur Traffic Analyse in der WebUI findet ihr in diesem Video.

Zusatzaufgabe: Registration-Test vervollständigen

Für die folgenden Tests bauen wir einen vollständigen Login- und Registrierungstest. Dafür können wir uns die Schnittstellen des Controller der Demoapplikation hier genauer anschauen.

Dann brauchen wir:

- Eine Vorbedingung "user x existiert nicht bzw. wird gelöscht"
- Einen parametrisierten Login-Schritt mit Username und Passwort (haben wir aus vorherigem Stage)
- Einen Login-Lauf mit einem nicht bekannten User, der fehlschlagen soll (haben wir aus vorherigem Stage)
- Einen Registrierungslauf, der erfolgreich sein soll, und ein anschließendes Login, das ebenfalls erfolgreich sein muss

Hinweis: Versuch so viel wie möglich die vorgefertigten GlueCode Steps des Tigers zu nutzen und so wenig wie möglich eigenen GlueCode zu schreiben.

Stage 5: Configuration & Placeholder

Per Platzhalter werden Daten, vllt username/pw, eingefügt. Und in der Rückgabe werden dann nicht feste Werte verglichen sondern einfach genau diese Platzhalter wieder. `mvn verify` nicht vergessen.

Eines der Kern Features, die Tiger mitbringt, ist das Einlesen und Verwalten von Daten & Konfiguration. Dieses Feature nutzen wir schon die ganze Zeit, nämlich um die tiger.yaml einzulesen und die Testumgebung zu manipulieren. Die `tiger.yaml` ist die einzige Datei, die automatisch eingelesen wird. Der Basekey der `tiger.yaml` ist "`tiger`". Diesen kann man benutzen, um im feature-File auf Variablen aus dem `tiger.yaml` zuzugreifen. Um Testdaten oder Default-Werte aus weiteren Dateien einzulesen, müssen wir dafür Sorge tragen, dass diese anderen Dateien auch eingelesen werden. Dazu tragen wir die entsprechenden Dateien unter `additionalConfigurationFiles` in der `tiger.yaml` ein. Das wird hier näher erklärt.

- Lege eine neue YAML-Datei an in welcher du einen Nutzernamen und ein Passwort hinterlegst. Die Struktur kannst du dir hierbei selbst überlegen.
- Trage die Datei in die `tiger.yaml` unter `additionalConfigurationFiles` ein:

```
additionalConfigurationFiles:
  -
    filename: demoData.yaml
    baseKey: demo
```

  - Der Filename muss auf die einzulesende Datei verweisen, relativ vom Root-Verzeichnis der Testsuite aus.
  - BaseKey ist ein Schlüssel welcher als Prefix für alle Schlüssel in einer YAML-Datei dient. Der BaseKey für die `tiger.yaml` ist `tiger` .
  - Man kann auch auf einen BaseKey verzichten, in diesem Falle wäre der Basekey "`tiger`". Es ist lediglich ein Mechanismus, um die Verwaltung von Namespaces zu vereinfachen.
- Ersetze nun in dem feature-File die Username- und Password-Parameter beim Einloggen und Registrieren mit Platzhaltern der Form `${gewählterBaseKey.gewählterUser.username}` . Diese Platzhalter sollen auf die Werte aus der Daten-Yaml Datei verweisen.
- Im Gluecode müssen nun die `{string}` durch `{tigerResolvedString}` ersetzt werden, die Methodenparameter bleiben String. Durch `tigerResolvedString` erkennt Tiger, dass dieser String zuvor aufgelöst werden muss, so dass auch tatsächlich Werte für die entsprechenden Platzhalter hinterlegt werden.

- Fertig! Führe nun einen Testlauf aus und schaue in das RBel-Log hinein. Dort sollten die ersetzten Werte auftauchen (wenn du alles richtig gemacht hast).

Zusatzaufgabe

- Für weitere Tests kann es nützlich sein, mehrere Test-User zu hinterlegen.
  - Baue in die Data-YAML eine Map an Test-Usern ein
  - In dem Aufruf des Glue-Codes musst du nun den Namen angeben. Verwende hierfür ebenfalls eine Variable!
- Man kann natürlich auch eine Variable verwenden, um diese dann im TigerConfiguration Editor zu ändern und mittels Replay in der WorkflowUI ein Scenario erneut abzuspielen. Probiere es aus.

Mehr Infos zur TigerConfiguration findet ihr in diesem Video.

Stage 6: Umgebungsabhängige Konfiguration

Wir führen zwei Testumgebungen ein: "local" und "external". Es sollen unterschiedliche Daten in Abhängigkeit der Umgebung gewählt werden. Weiterhin muss natürlich die Umgebung selbst angepasst werden.

Es gibt in unserem Projekt jetzt zwei verschiedene Arten von Testumgebungen: Eine externe und eine lokale Testumgebung. Das ist eine sehr normale Situation: Lokal will man beim Entwickeln der Testsuite mit Simulatoren arbeiten, auf dem Buildserver oder bei einem echten Abnahmetest mit einem externen System. Mit dem oben dargestellten Konfigurations-Mechanismus hat man eigentlich schon alle Tools an der Hand, um ziemlich elegant mit diesem Problem umzugehen. Wir wollen das hier für uns einmal nachbauen.

- Lege zwei weitere YAML-Dateien an: `tiger-local.yaml` und `tiger-external.yaml`
- Kopiere aus der bisherigen tiger.yaml den gesamten `servers` -Node in die `tiger-local.yaml` . Und lösche die Server aus der tiger.yaml. Diese YAML nutzen wir, um lokal eine aktiv verwaltete Umgebung anzustarten.
- In die `tiger-external.yaml` kopierst du auch den `servers` -Node auf
  - ```
    servers:
      demo:
        type: externalUrl
        source:
          - http://localhost:8080
        healthcheckUrl: http://localhost:8080/service/status
    ```
    .
  - Führe das demo.jar aus. Der Defaultport ist 8080.
- Nun müssen wir noch die richtige der beiden YAML-Dateien einlesen. Wir fügen dazu in der `tiger.yaml` einen neuen Eintrag unter `additionalConfigurationFiles` ein. Das besondere: Der Dateiname wird selber wieder mit einem Platzhalter ergänzt: `tiger-${environment}.yaml mit baseKey "tiger".`
- Jetzt kannst du das Setup schon ausprobieren: `mvn verify -Denvironment=local` und im IntelliJ mit `environment=local` in den Environment variables in den run configurations.

Das ist schon sehr cool, aber es gibt eine entscheidende Schwäche: Was ist mit Default-Werten? Vielleicht will ich immer local nehmen, außer ich will tatsächlich gegen die externe -Umgebung gehen.

- Lege eine neue Datei `defaults.yaml` an. Diese binden wir auch wieder als additionalConfigurationFiles mit ein, aber diesmal OHNE baseKey, dafür an oberster Stelle der additionalConfigurationFiles.
- In diese Datei schreibst du den default-Wert für `environment` mit rein.
- Und ausprobieren: `mvn verify` startet nun die Testsuite mit der Default-Umgebung.

Stage 7: RbelPath reloaded

blocked URL

Wir asserten direkt in das zurückgegebene JWT hinein.

Oben wurde schon erwähnt, dass RBel-Path gerade bei ineinander verschachtelten Datenstrukturen seine Fähigkeiten voll ausspielen kann. Das werden wir jetzt bei dem zurückgegebenen JWT (Json Web Token) demonstrieren. Ein JWT ist eine kompakte Struktur, mit der signierte Daten ausgetauscht werden können. Sie besteht aus drei Teilen, Base64-URL kodiert und mit einem Punkt unterteilt. Die Teile heißen Header, Body und Signatur. Der Header beschreibt, wie die Signatur zu checken ist, der Body ist die signierte Payload, die Signatur beglaubigt den Body (oder auch Body und Header).

- Schaue zuerst auf das Rbel-Log und betrachte das JWT (sollte als Antwort auf /performLogin kommen). Das JWT kann auf jwt.io näher untersucht werden.
- Rbel-Tree zeigen (`TGR print current response as rbel-tree`)
- Username im Token selbst asserten

Zusatzaufgabe

- Asserte auf einen korrekten Algorithmus im Header
- Recursive-descent: Suche nach jedem Vorkommen von `name` in der gesamten Nachricht (`$..nbf` gibt alle Einträge von nbf in der gesamten Struktur zurück)

Mehr Infos zum RbelPath und JEXL findet ihr in diesem Video.

Stage 8: JSON-Checker (optional)
    ```

Noch eine Stufe eleganter: Wir vergleichen gleich das ganze JSON, sprinkeln ein paar Platzhalter rein. Dabei müssen wir natürlich noch einen Zeitstempel ignorieren.

Hierfür gehen wir zu dem erfolglosen service/performLogin und schauen uns die Response an.

Als Hilfsmittel beim Testen von Protokollen wurde ein direkter JSON-Vergleich aus dem feature-File heraus entwickelt. Das hilft dabei, sehr ausdrucksstark eine Schnittstelle zu beschreiben.

- Neuen Vergleich einbauen "`TGR current response at "$.body" matches as JSON:`"
    - Beachte dabei, dass Strings die über mehrere Zeilen gehen (im feature-File), mit """ beginnen und enden
- Für das Datum entweder ein "cooles" Regex bauen oder per "`${json-unit.ignore}`" den Wert ignorieren.

Tipps

Wollt ihr in eurem Gluecode direkt auf den TigerGlueCode zugreifen/nutzen, macht das am besten mit der @Steps Annotation von serenity. Diese instanziiert die annotierten Klassen.

```
    import net.serenitybdd.annotations.Steps;

public class TestsuiteGlueCode {

    @Steps
    HttpGlueCode httpGlueCode;
    @Steps
    RBelValidatorGlue rBelValidatorGlue;
...
}
```