Student Instructions

These are the student instructions for the TDDE46 lab on REST API evaluation, testing and exploration.

```
TODO: UPDATE SETUP AND IMAGE FETCHING INSTRUCTIONS. WHERE DO WE STORE IT? LOCAL REGISTRY?
HOW TO AVOID PORT CONFLICTS IF USING SHARED ENVIRONMENT? COPY INSTRUCTIONS FROM DOCKER LAB.
TODO: UPDATE ALL IMAGE VERSION REFERENCES BY SEARCING FOR "VERSION" IN THE DOCUMENT.
```

1. Introduction

In this lab you will be given the opportunity to investigate, evaluate and explore an Authenticator service, which exposes a REST API. It is provided as a Docker image, and through a series of exercises you will reflect on it from multiple perspectives and apply both manual and automated testing to it. Apart from these instructions, you need a separate Requirements document. This document lists the Authenticator service requirements that you will test, and provides a description of the service and its purpose.

1.1 Prerequisites

To complete this lab, an environment with Docker installed and running is required. Apart from that, no experience or in-depth understanding of containerization in general or Docker in particular is needed; detailed instructions will be given for all Docker operations. If you do not have Docker installed, see https://docs.docker.com/get-started/ to get started.

To build the automated verification project you need Maven. If you don't have Maven set up, see https://maven.apache.org/download.html and https://maven.apache.org/install.html.

Prior to the lab, spend some time reading up on Representational State Transfer (REST) and RESTful API design. There are multiple views on what good RESTful design looks like, and sampling some of them by simply searching the Internet is a useful exercise. Also take a brief look at the canonical source on REST: Fielding, R. T., & Taylor, R. N. (2000). *Architectural styles and the design of network-based software architectures*. You don't need to read the whole thing, but Section 5 is useful background reading.

1.2 Preparation

1. Open your favorite shell.
2. Run `docker ps` to verify that Docker is running. You should see something like this:

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS              PORTS               NAMES
```

This tells you that no containers are currently running.

1. Fetch the Authenticator service image: `PLACEHOLDER`

2. Create a container from the image: `docker run -d -p <port>:8080 tdde46/authenticator:VERSION` where `<port>` is the port you wish to use on your host machine. If you're running in your own local environment you can use any port number (e.g. `8080`), but if you're running in a shared remote environment you need to select a unique port number to avoild collisions. `PLACEHOLDER: EXPLAIN PORT SELECTION SCHEME.` You should see something like this:

   ```
   $ docker run -d -p 8080:8080 tdde46/authenticator:VERSION
   03e4aa138d5a45a76839077e47564d0f3000268e83a0e1d99402fd4be24185ef
   ```

   Replace VERSION with 1.0.0 or 1.0.0.

   The `-d` means detached mode, allowing you to continue using your shell while the container runs in the background. `-p <port>:8080` publishes the container's port 8080 to your host machine's port 8080.

3. Verify that the container is running by running `docker ps` once more. You should see something like this:

   ```
   $ docker ps
   CONTAINER ID        IMAGE                          COMMAND                  CREATED
   STATUS              PORTS                                NAMES
   03e4aa138d5a        tdde46/authenticator:VERSION   "/bin/sh -c 'java --…"   34 minutes ago
   Up 34 minutes       0.0.0.0:8080->8080/tcp, 8081/tcp   frosty_mendeleev
   ```

4. Open your favorite browser and point to to `http://<hostname>:<port>/ping`, where `<hostname>` is `localhost` if you're running on a local machine, or the hostname of the server if you're running in a remote environment. You should get something like `{"id":1,"content":"Pong!"}` on Content-Type `application/json`. If at first you don't get a response, wait a few more seconds for the service to spin up.

5. Now you're all set!

2 Exercises

The exercises are a mix between tasks to execute and questions to reflect on. Completed tasks are shown to and assessed by the lab assistant. For reflection exercises, begin by collectively phrasing your thoughts as text, then present the text to your lab assistant and discuss it with them. A benchmark to aim for is approximately 250 words per reflection exercise. If you end up with less there's a fair chance you haven't exhausted the subject.

2.1 Service Quality Assessment Task

In the Requirements document you will find a description of the Authenticator service along with its requirements. Read the description and browse through the requirements, then explore the REST API of the service. To do this, point your browser to `http://localhost:8080/swagger`. The service exposes an interactive Swagger documentation of its API, which lets you try out its exposed methods. Go through each of the methods, read its documentation in Swagger and try it out for yourself to see what kind of responses you get back.

```
Tip: In the Swagger UI, to find the correct syntax of a request body (e.g. for POSTing to /permissions/create)
you can click "Model" under the Description of the body parameter.
```

```
Tip: You can of course access the service using other clients as well. For instance, try using
curl from your shell. Just make sure your request header accepts application/json content,
or you will receive an HTTP 406 code!
```

Now consider the Authenticator service and its API from the following quality perspectives:

- How would you judge its usability? Are the methods easy to use and easy to understand? Is the output easy to use and easy to understand? How RESTful is the service?
- How would you judge its security? Are there any obvious security concerns with this service? If so, how would you improve it?

Also consider the Requirements document.

- Are the requirements understandable?
- Are the requirements comprehensive?
- Are the requirements unambiguous?

2.2 Manual Requirement Verification Task

Use the Swagger UI (or some other client of your choice) to manually verify each of the requirements on the Authenticator service.

2.3 Test Methodology Reflection 1

What are your reflections on the manual verification of the requirements? Is this a reasonable way of testing? What are the pros and cons? Some things to consider:

- Are there any particular types of faults that this type of testing might be well suited for?
- How would you judge the upfront investment and effort (CAPEX) required for this type of testing?
- How would you judge the continual investment and effort over time (OPEX) required for this type of testing?
- Assume that your job description was to run these and similar tests 40 hours a week. How would you feel about that as an engineer?

2.4 Automated Requirement Verification Task

Use TestNG and RestAssured to create automated test cases verifying each of the requirements on the Authenticator service. You will find a pre-configured project with a few implemented tests to help you get started in the authenticator-test project, supplied to you along with these instructions. You can execute the tests from inside of your favorite IDE or from your shell using `mvn test -Dv=1.0.0`.

2.5 Test Methodology Reflection 2

What are your reflections on the automated verification of the requirements? Did you prefer it to the manual testing? What are the pros and cons in this case? Some things to consider:

- Did you encounter any faults that you didn't identify during manual testing?
- Could all requirements be verified? Did you achieve a clean mapping of requirements to test cases? Is that desirable? Why, or why not?
- How would you judge the upfront investment and effort (CAPEX) required for this type of testing?
- How would you judge the continual investment and effort over time (OPEX) required for this type of testing?
- Which do you find most fun and rewarding - manually performing and repeating these tests, or creating automated test cases to do it for you?

2.6 Non-Functional Requirements Reflection

Review the list of requirements again. Note that there are no non-functional requirements. What is the implication of this? Would it be possible to design a service that is functionally correct, but that cannot serve its stated purpose due to non-functional aspects? Perhaps that is already the case in the provided implementation? Recall what you have seen of the API and its behavior - is there any behavior you have spotted that might be problematic? If so, which of the test methods you have tried would be most likely to uncover it?

Reflect on which non-functional requirements might be relevant for the Authenticator service. Which of the test methods you have tried would you recommend for ensuring that the service stays compliant over time with these requirements? How would you design a process that ensures that the service stays within acceptable non-functional boundaries throughout its life-cycle?

2.7 Testability Reflection

How would you judge the testability of the Authenticator service in general? Is the provided API conducive to manual and/or automated testing?

Requirement 2.4.4 specifies time intervals where the validity of a permission is to be queried. What are the implications of this? In particular, think of a non-trivial software project with many thousands of automated test cases of services like this one, and think of a continuous delivery scenario where many new versions are created every hour. How will such time intervals impact the ability to continuously deliver? What might be done differently in the service implementation to enhance its testability in this regard?

2.8 Requirements Phrasing Reflection

Review the Authenticator service requirements one final time. How are the requirements phrased? Are they phrased "positively" or "negatively", in the sense that they state what *shall* happen or what *shall not* happen? In testing one often speaks of "happy path" testing as opposed to "sad path" testing. What might the correlation be between these types of testing on the one hand, and how the requirements are phrased on the other? How might this affect whether certain types of faults go undetected? How might one go about phrasing requirements and implementing tests designed to detect faults rather than verifying assumptions?