System Documentation

The main entry point into the project is the official project readme, which can be found here: https://github.com/learning-layers/LAPPS/blob/develop/README.md

The project can be divided into a frontend, responsible for the client-side representation (i.e. the web page) and a backend, which handles the data.

The backend currently connects to a database, but for the future it is planned to switch to the Social Semantics Server.
https://github.com/learning-layers/SocialSemanticServer

Frontend

JSDoc

The current documentation is updated automatically every day and can be found at http://layers.dbis.rwth-aachen.de/jenkins/job/LAPPS/JavaScript_Documentation/

Tools Overview

- *Node.js*:
  Cross-platform runtime environment to create network applications. It uses the Google V8 JavaScript engine for code execution and can be used in a JavaScript development environment.

- *Karma:*
  Testing environment to test JavaScript in real browsers. It runs in the background and re-runs tests whenever it detects a change in the JavaScript code. It supports multiple testing frameworks, such as Jasmine.

- *PhantomJS*
  Headless WebKit scriptable with JavaScript.Can be used for testing of web applications over the command line without using any actual browser and is especially useful in continuous integration systems.

- *Protractor*
  End-to-end test framework for AngularJS applications. It opens up a browser and simulates a user interacting with the web page.

- *UglifyJS*
  JavaScript minifier to reduce the file size of scripts, which allows faster page load. It generates very compact, but for humans hardly readable code, which is then used for deployment.

- *AngularJS*
  Web application framework to create dynamic single-page web applications running on the client side. It follows the MVC pattern and two-way data binding by introducing new custom HTML tags and attributes.

- *Bootstrap*
  Framework for responsive web design. It allows the easy creation of web pages, that scale depending on the display device size. This way a web page can also be used on a mobile device, like a smartphone, without the need to create a special app for it.

- *jQuery*
  Cross-platform JavaScript library which simplifies the use of JavaScript in web applications. It makes working with DOM elements, animations and Ajax requests much easier and is used by many webpages.

Structure

- app
  - api
    - Contains the automatically generated API definitions of the backend.
  - assets
    - Data like images, icons and css style sheets.
  - bower_components
    - External libraries are stored here. They are downloaded automatically.
  - components
    - Contains the individual pages (each has one folder) with their view, controllers and directives.
  - core
    - appModule.js glues everything together (i.e. module management) and provides configuration and routing of the AngularJS application.
  - shared
    - Contains elements shared among pages, such as services and the header. platformService.js handles platform selection in the frontend (what platforms are available, their icons and names, etc.). userProvicer.js handles the login with OIDC and stores user data, like token (to send in requests).
  - index.html
    - The initial page of the frontend. Responsible for loading all required scripts.
- deploy
    - ‘Compiled’ and minified version of the webpage (fast page load due to small sizes and only few files). The contents can be generated automatically and used directly on a web server without any external dependencies.
- out
    - Automatically generated JSDoc documentation in .html format.
- test
  - e2e
    - Protractor test specifications. They do not get executed in the Jenkins tests. Currently only Chrome is supported as the testing browser.
  - unit
    - Karma test specifications. They are automatically run by Jenkins.
  - utils
    - Helper functions for the tests.
      Swagger2HttpBackend.js allows to use the swagger client for $httpBackend mocks in AngularJS.
      karmaHelper.js contains mockup-JSON objects for http responses.

mockUserProvider replaces the real userProvider and does not connect to the layers oidc server (but provides some mock user data).
    - karma.conf.js
        - Configuration file for Karma tests.
    - protractor.conf.js
        - Configuration file for Protractor tests.
- patchLibs.js
    - Some external libraries do not work perfectly as needed with this project. This script patches them automatically after installation.
- fetchApi.js
    - Script to fetch the current swagger documentation from the backend manually.
- deploy.js
    - Script to create a deployable website in /deploy.
      It automatically reads the index.html and groups and concats scripts. CSS, HTML and JS are minified so: **Attention**: You need to write minification compatible AngularJS code, if you want to use the deploy files afterwards:
      https://scotch.io/tutorials/declaring-angularjs-modules-for-minification

- bower.json
    - Contains library dependencies of the web page. The libraries are automatically downloaded.
- package.json
    - Contains library dependencies for the tools used (i.e. these libraries are not referenced by the web page) and some commands to run in the console.

Important console commands

- `npm install` --- install dependencies (will be done automatically when running the start command)
- `npm test` --- test frontend with karma
- `npm start` --- start on node server (port 8000)
- `npm run protractor` --- start e2e tests (start the server with npm start first in another console)
- `npm run doc` --- create jsdoc documentation in out/
- `npm run deploy` --- create a deployable containing minified files in deploy/
- `npm run api` --- fetches the up to date Swagger api from the deploy server for frontend usage
- `npm run apilocal` --- fetches the up to date Swagger api from the local server for frontend usage

Tests

- Karma Tests:
    - The Karma tests utilize PhantomJS and are used for continuous integration on Jenkins. No browser is needed. The tests check basic functionality of the controllers, services and directives without actually testing the API.
    - API calls are intercepted automatically and instead a mock-response is returned. (see https://docs.angularjs.org/api/ngMock/service/$httpBackend). This way the karma tests still pass when the backend has errors or the server crashed. But the correctness of the API cannot be ensured

- Protractor Tests:
    - Protractor uses Chrome to simulate the input of a user. It cannot cover all possibilities due to the page complexity. Also rendering errors (due to CSS) cannot be caught. But it can generally check if the page displays some results and the routing between pages works correctly.
    - Unlike the Karma tests real API calls are used, so the tests will fail if the backend server is not running.

Work Flow

1. After downloading and installing Node.js you should make sure to download all external dependencies. Open the console in the context of the LAPPS-frontend directory (i.e. where the package.json is located):
2. `npm install` will install all required libraries and resolve all dependencies. Now you should have everything you need for development.
3. You might also want to run `npm run api` if there were changes in the backend regarding the swagger API. This command will download the latest swagger API definitions into the project.
4. Check the API documentation at http://buche.informatik.rwth-aachen.de:9080/lapps-0.3-SNAPSHOT/swagger-documentation/ .
   Sadly the swagger UI does not show the nicknames of the methods, which are required if you want to use the swagger client in AngularJS. But they can be found out easily:
    a. Look them up in the backend code.
    b. Look them up in lappsAPI.js. You can use some of the online JSON formatters to make it more readable.
5. After making your changes in /app make sure to register new scripts in index.html.
   If you have created a new page, make sure to adjust the $routeProvider configuration in appModule.js. If you create new directories, check the deploy.js if they are included

correctly in jsGroups or copyPaths respectively (only needed, if you want to use the deploy script later).

6. Run the Karma tests with `npm test`.
7. You can run the page on a test server (localhost:8000) with `npm start`.
8. To run the Protractor tests you need to have a running test server (as above) and a Chrome installation: `npm run protractor`.
9. To create a documentation of the code run `npm run doc`.
10. If you want to create deployable files use `npm run deploy`.
    Make sure your angular code is minification compatible (https://scotch.io/tutorials/declaring-angularjs-modules-for-minification).
    Of course you can deploy the contents of /app directly and it will work, but it would increase the page loading times drastically (and also requires much more storage space).

Backend

JavaDoc

The current documentation is updated automatically every day and can be found at http://layers.dbis.rwth-aachen.de/jenkins/job/LAPPS/de.rwth.dbis.layers.lapps%24LAPPS-backend/javadoc/

Tools Overview:

- *Java Enterpise Edition*
  Java EE (Enterprise Edition) is a specification about a platform that runs within the Java environment. It includes different aspects like web services, persistence layer, message service, multi-tier architectures, etc.

- *Javadoc*
  Javadoc is the standard API documentation for Java. It generates an HTML documentation from code comments with special tags. The current documentation is updated automatically every day and can be found at http://layers.dbis.rwth-aachen.de/jenkins/job/LAPPS/de.rwth.dbis.layers.lapps%24LAPPS-backend/javadoc/

- *Jersey*
  Reference implementation of Java API for RESTful Services (JAX-RS) to build light and
  robust web services. Jersey provides his own API which provide more features.

- *Swagger*
  Swagger is a specification and framework implementation for describing, producing, consuming, and visualizing RESTful web services. The Sawgger UI will be updated automatic and can be viewed at http://buche.informatik.rwth-aachen.de:9080/lapps-1.0/swagger-documentation/

- *OpenID Connect*
  OpenID Connect is an authentication protocol on top of the OAuth 2. It works according to the RESTparadigm and uses JSON for its payloads. Its key feature is that it enables developers to authenticate users among different apps without the need of storing locally any kind of user credentials. Which makes it also more comfortable for the users, who do not need to create a separate account for each service they want to use.

- *Apache Maven*

Maven is a tool for managing builds, reporting and documentation of Java projects from one single project object model (POM) file.

- *Grizzly HTTP Server*
  The Grizzly framework was designed with the aim to take advantage of the Java NIO (Non-Blocking) API. The Grizzly HTTP server is used in the Glassfish Application server, where it takes care of the network operations. Since it is embeddable and light-weight and provides support for the deployment of our Jersey RESTful services, we quite naturally have chosen it for our development and test environment.

- *Tomcat*
  Tomcat is a widely used Web server implementation of the Java Servlet and JavaServer Pages (JSP) specifications. It is the default Web Container of another certified Application server - WildFly (formerly known as JBoss). Our team has chosen it for our production environment, because it is lightweight, stand-alone and largely extensible.

Tests:

We created automated unit tests for all the JAX-RS endpoints. The tests provide their own dummy data and remove it again after execution.

Important console commands:

- `mvn clean` --- clean
- `mvn test` --- test backend
- `mvn exec:java` --- compile and start on jetty server (port 8080)
- `mvn javadoc:javadoc` --- generates JavaDoc documentation
- `mvn package` --- generate a servlet that can be deployed to an application server
- `mvn test -Dtest`=DataGenerator --- generates mockup data in the database

Database Access

To avoid putting login data for the database into a publically accessible file on github, the login data is stored in a settings.xml (in C:\Users\.m2 for Windows Systems):

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0

http://maven.apache.org/xsd/settings-1.0.0.xsd">
    <profiles>
```

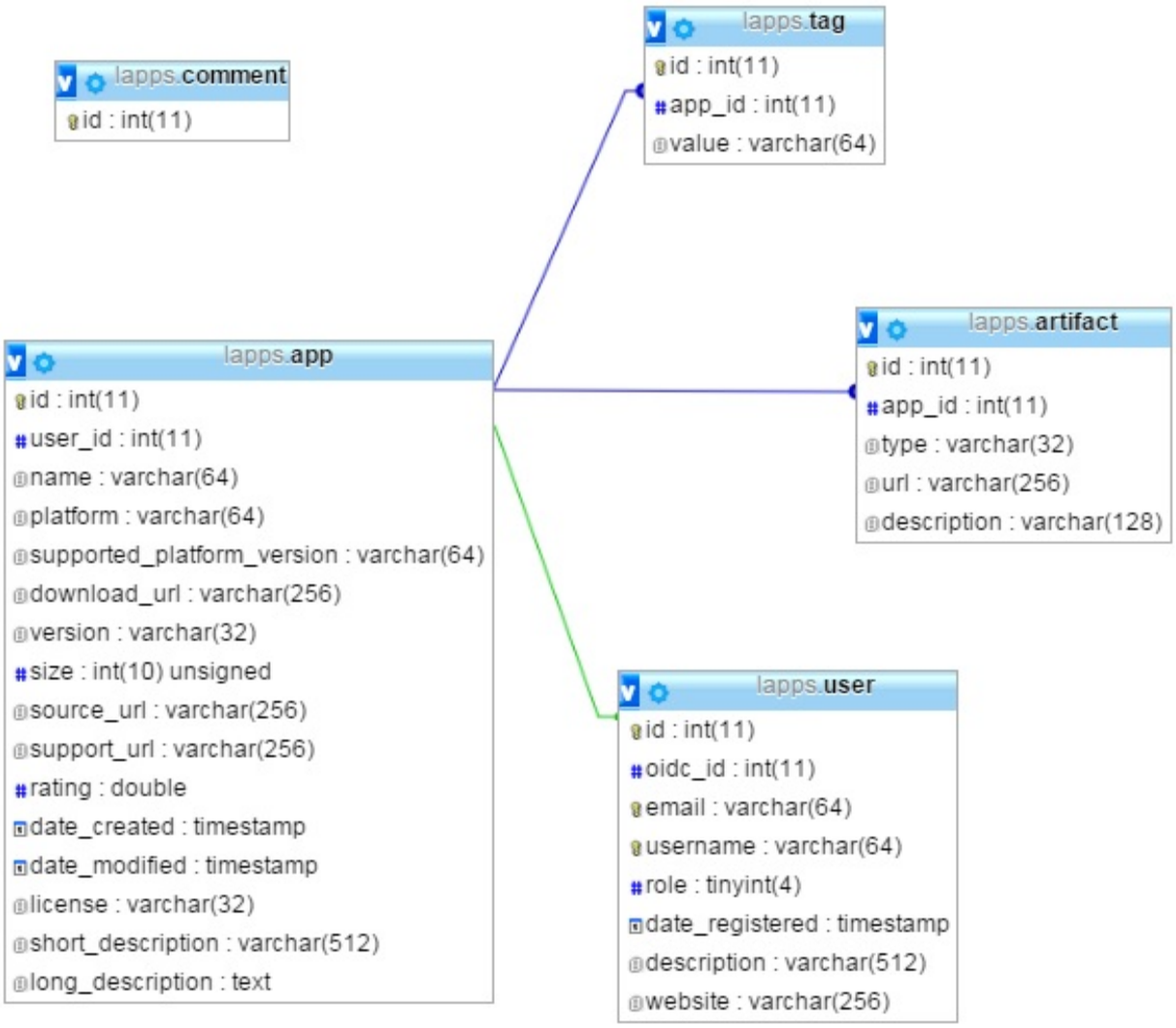
```
        <profile>
            <id>BucheDatabase</id>
            <properties>

<jdbc.url>jdbc:mysql://buche.informatik.rwth-aachen.de:3306/henm1415g
1</jdbc.url>
                <jdbc.username>banana</jdbc.username>
                <jdbc.password>pineapple</jdbc.password>
            </properties>
        </profile>
    </profiles>
    <servers>
        <server>
            <id>LabCourseTomcatServer</id>
            <username>banana</username>
            <password>apple</password>
        </server>
    </servers>

    <activeProfiles>
        <activeProfile>BucheDatabase</activeProfile>
    </activeProfiles>
</settings>
```

Either create your own database/tomcat setup or ask us for the credentials.

Database



A well known challenge in application development is the mapping between the (volatile) data, residing on the application level to the (non-volatile) persistence storage. In the context of object-relational application development and relational storage mechanisms (Relational

Database Management Systems), the problem is widely referred to as Object-Relational Mapping (ORM). As the way data is stored onto the persistence layer most often differs from the so called business or domain entities, used on the application business layer, the task of mapping the data would usually include writing numerous database queries for each and every business entity. Most often large part of the queries regarding different entities are similar if not equal in its logic, that making the work more mundane. The Object-Relation Mapping aims at 'covering' the relational layer and enabling the application programmer to express queries about the data in terms of the business data model (i.e. in terms of objects) and not relations. It also supports (most often) automatically updating and reading the entities, once a certain 'mapping' between the business data model and the relational data model is provided.

As a mature object-oriented language, Java has its means to address the issue and namely - **Java Persistence API (JPA)**. Several implementations are known with EclipseLink being the reference implementation. Another implementation leveraging the API and building upon it is **Hibernate**, also chosen in our project because of its solid background and community support. With the help of Hibernate the application programmer is able to annotate its business entities (objects) thus mapping them to (existing) database relations. Other than some more specific queries, Hibernate is able to do most of the read and update work without any further interference.

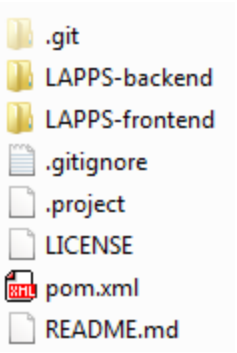
In our project, classes are mapped to the tables in straight-forward “one-to-one” manner. Though, the basics of “theoretical” normal forms are slightly violated in our approach due to the purposes of readability and faster query processing.

How to set up the project for further development

This document has the goal to guide the reader through all the steps necessary to get LAPPS up and running on a local environment. It also introduces most of the technologies used in the project and gives an overview over the software prerequisites for everyone interested in contributing to the project.

The reader is also encouraged to use the readme.md file at the GitHub project’s page as a reference regarding the technologies and tools used:
https://github.com/learning-layers/LAPPS/blob/master/README.md

Initial step: Clone the repo
The tools needed before setting up the front-end for development are git and nodejs. A clone of the repository under https://github.com/learning-layers/LAPPS.git should download the source code of both the front-end and the back-end subprojects into folder named “LAPPS” with the following contents:



Front-end

The front-end is built entirely with Javascript with the help of AngularJS framework and Bootstrap CSS framework. The application can either be used via the web-interface or via the RESTful API it exposes. The tools used in the development process are Node.js (Javascript run-time environment running the V8 Google engine), Karma, PhantomJS, Protractor (the latter three related to testing) and UgligyJS (minifies and obfuscates Javascript code).

Step 1: Resolve front-end dependencies
Open a console and navigate to the LAPPS-frontend folder, shown on Fig.1. Type “npm install” to install all the packages the front-end needs. What files are being downloaded is described in the “package.json” file. For more information on that matter in general, the reader is to search for npm – the Node.js packet manager.
After the step is complete, make sure that a “node_modules” folder is present (and not empty) under the current directory and also that an “app/bower_components” folder is to be seen as well.

Step 2 (optional): Update Swagger API by changes on the back-end
Run “npm run api” to download the latest Swagger API definitions.

Step 3: Start the front-end
Run “npm start” to start the node.js server on port 8000. This server is used when there is no need for end-to-end communication (e.g. when testing the client only). To make the front-end communicate with the actual back-end server, just start the latter directly (see below for instructions on how to set it up) and navigate to the proper address.

Here’s a list of the more important commands and front-end scripts:

- `npm install` --- install dependencies (will be done automatically when running the start command)
- `npm test` --- test frontend with karma
- `npm start` --- start on node server (port 8000)
- `npm run protractor` --- start e2e tests (start the server with npm start first in another console)
- `npm run doc` --- create jsdoc documentation in out/

- `npm run deploy` --- create a deployable containing minified files in deploy/
- `npm run api` --- fetches the up to date Swagger api from the deploy server for frontend usage
- `npm run apilocal` --- fetches the up to date Swagger api from the local server for frontend usage

Back-end

The back-end is built on Java 7 with the help of Jersey JAX-RS reference implementation, Jackson JSON processing library, Hibernate JPA implementation, JUnit and Swagger framework.The project is built with Maven and tested on Grizzly embedded Web server, although the application can (and is) deployed on Apache Tomcat as well.

Step 1: Resolve back-end dependencies
Eclipse usually downloads all needed jar files automatically upon start. If that is not the case, please refer to the Maven documentation for more detailed information. Once downloaded, the libraries are usually available under “Libraries/Maven Dependencies” in Eclipse.

Step 2: Put database credentials under “C:\Users\.m2” (for Windows)
The file holding the database credentials should either be obtained from third party or newly created if another database instance is used.

Step 3: Start the back-end
Run “mvn exec:java” to execute the Main class of the app, which in turns starts the embedded Grizzly server. Swagger UI is available under http://localhost:8080/ (can be changed in Main.java). To run the Web interface it should be deployed manually.

Here’s a list of the more important Maven commands:

- `mvn clean` --- clean
- `mvn test` --- test backend
- `mvn exec:java` --- compile and start on jetty server (port 8080)
- `mvn javadoc:javadoc` --- generates JavaDoc documentation
- `mvn package` --- generate a servlet that can be deployed to an application server
- `mvn test -Dtest=DataGenerator` --- generates mockup data in the database