Classic
Developer Manual

by Team Classic

1 Introduction

As Classic is a webapplication, the code is split into two parts. There is the frontend and the backend. The frontend has all the logic to render the webpages. The backend has the code to handle all the requests made by the frontend. The following sections explain these two parts in more detail. The deployment diagram of our system is shown in figure 1.

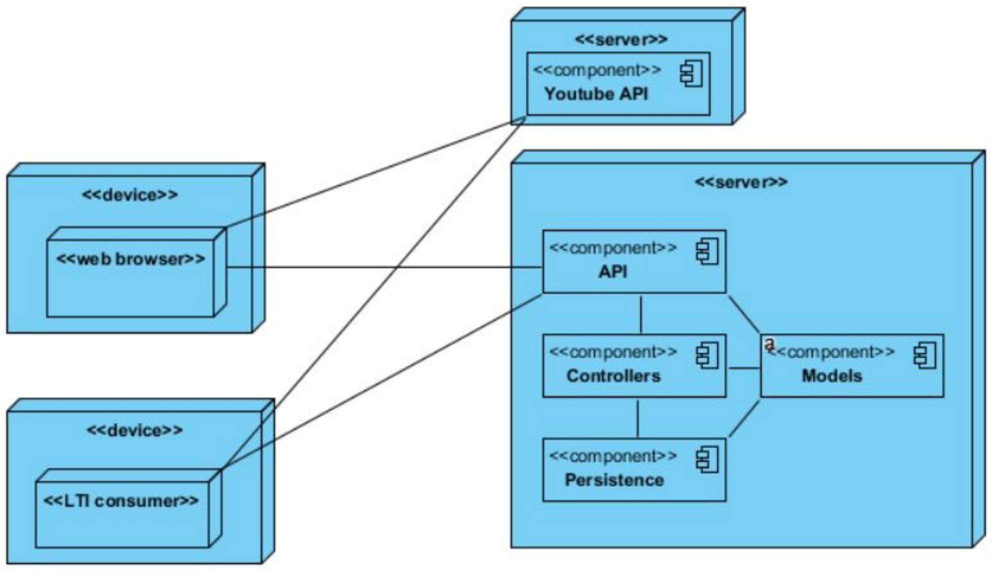

Figure 1: Deployment Diagram

2 Backend

The architecture from the first semester was adapted to better fit the scope of the project. The Publish-Subscribe pattern was dropped and no adapters are used for importing external courses.

The backend of our project was written in Java. We use PostgreSQL for our database and Java Spring for our API. We will now discuss the different components of the backend implementation.

2.1 Models

The model layer represents the business logic of our application. Our system is mainly based on keeping courses and lectures with videos and course notes. videos and course notes can have comments and replies. A comment does not belong to any course material, instead it can be referenced to from multiple course materials. This is what we call a VideoReference or CourseNotesReference. We also have a user and different roles(teacher, student and admin) that define what each user can do.

To illustrate the concept of references we will now give an example. We have a course with a lecture and in that lecture there is a video and there are course notes. If we add a comment to the video we need to add a VideoReference to the comment. The VideoReference includes information about where it refers to. To add a cross-reference to the course notes, assuming the video and course notes talk about the same concept, a CourseNotesReference is created and added to the existing comment. The comment is now a comment of both the course notes and the video. A VideoReference keeps a timestamp while a CourseNotesReference keeps a list of locations. These locations each contain 4 coordinates to form a rectangle and a pagenumber.

2.2 Persistence

The persistence layer takes care of the communication with the database.

This layer contains a set of Data Access Objects (DAOs) that are each responsibly for a specific part of the database. For example the VideoDAO is responsible for every action related to Videos. Every DAO extends the ClassicDatabaseConnection which is an abstract class that takes care of setting up the connection to the database. The resource folder in the persistence module contains the property file "dbconfig.properties" with the database specifications. The other property files in this folder are used for testing purposes.

The persistance module also contains the exception classes used in the backend of the application. There are four exception classes. The first one is The ClassicDatabaseException which is an exception that is thrown when an error occurs in the communication with the database. The second exception is The ClassicNameTakenException that indicates when someone attempts to add an object of which the name is taken. The third exception is the CLassicNotfoundException which indicates that an object that is being searched for in the database cannot be found. The last exception is the ClassicUnauthorizedException used in the user authorization. This exception is thrown when a user makes an unauthorized action.

2.3 Controllers

The controller layer of our application is responsible for the interaction between the API and the persistence layer. It mostly passes down calls and makes sure certain parameters are set. Next to that it also is responsible for uploading, downloading and deleting PDFs.

In the CourseNotesController, we used iText to add the comments from the webapplication to an actual PDF file. We do this by first copying the original PDF file to a new file and then writing the annotations to a ByteArrayOutputStream. Afterwards the contents of this ByteArrayOutputStream are written to the copy we had created earlier. This is done because adding an annotation to a file will create a new file instead of adding it to the original file.

The LTIController houses the main methods concerning Learning Tools Interoperability (LTI) Single Sign On (SSO). It handles the key-secret pairs and has a method to generate the OAuth signature.

2.4 API

To build the classic API, we used the spring framework. More specifically, the spring-boot package. The config package contains all global spring configuration data, as well as the controller and dao object creation. These controller and dao objects can then be passed to the endpoints, making sure that only one instance is created. The main package contains the main class with main method. This launches the application.

To be able retrieve data in the frontend, endpoints were created for each object as defined by our business logic (Video, CourseNotes, Comment, etc.). These endpoints are all very much alike in the way exceptions should be handled and access should be restricted. That is why the *Runner*-class was created. This abstract class has a method *"action"* in which all endpoint logic should be implemented. Once a *Runner*-instance has been created, the *runAndHandle*-method should be called. This takes care of access management and exception handling.

2.4.1 Parsers

All data that is provided to the frontend is provided in the JSON-format. For the transition between the backend models and the JSON-data, the *parser*-package is used. Generally speaking, the parsers implement the following methods;

- *to*: JSON object to backend object
- *toList*: JSON-array to list of backend objects
- *from*: Backend object to JSON object
- *fromList*: List of backend objects to JSON array

2.4.2 Spring authentication

The spring framework provides an authentication system, called spring security. This system needs to be configured correctly, to provide smooth integration between our user data on one hand and spring security on the other hand. This is done in the *security*-package. This package consists of the following components:

- *PasswordHasher*: Incoming passwords are hashed with BCrypt, so no plain text passwords can be extracted from the database.
- *Authority*: The spring equivalent of a Role (defined in the models): Student, Teacher or Admin. These have their own rights concerning access control.
- *ClassicUserDetailsService*: This connects the spring authentication system and our user controllers, to be able to do spring authentication with our user system.
- *SecurityConfig*: Global security configurations. Integrating all of the above components.
- *AuthenticationChecker*: A toolkit for the endpoints, which provides session information and method access control functionality. For access control, this class will read from the *authentication.properties* file and check if the role of the current authenticated user has access to a specific endpoint.

2.4.3 LTI endpoint

The only API endpoint that should be used by external people, is the "/lti" endpoint. This endpoint will only process POST requests and is meant for SSO defined by LTI. It allows users that are logged in on an LTI learning platform to automatically log in into Classic with the credentials of that platform. The POST body should contain all the information as defined by LTI. The endpoint will then check its validity and proceed with the authentication. If a user is not yet registered on the Classic platform, a new user will be automatically generated. The authentication will then be handled by Spring. After a successful validation you will be rerouted to the Classic platform and you will be logged in as "username_on_lti_platform#key".

All LTI usernames are generated in the above format. If a new request is made to the endpoint, and the user is not yet present in the Classic system, the new user will receive a new account with password "*very_secret_lti_user_password*". This password is not allowed for regular users. By this manner we restrict LTI users to only log in via SSO. The regular login endpoint will filter out all requests that have the LTI password.

2.5 Setup for local development

Editing Java code in a Gradle project can be done with your favorite developing tools that support Gradle. For example Eclipse or Netbeans with the Gradle plugin.

When running the code locally, a Spring server will be setup that connects to a local database. This database should be setup as follows:

1. Install posgresql + pgAdmin3.
2. In pgAdmin3 connect to your local database. Default: localhost:5432
3. Select the default database "postgres" and click the "SQL"-button in the action bar.
4. Go to File, then Open.. and open the file "createClassic.sql" and press run
5. Close the window and hit the refresh button untill you see the "Classic" database.
6. Select the "Classic" database and press the "SQL"-button.
7. Go to File, then Open.. and open each updateClassic file and press run

The sql files can be found in the **database** folder of the code directory.

3 Frontend

The frontend was written using the AngularJS framework (https://angularjs.org/). This can be written in any editor of your choice. We used two extensions on this framework: Videogular to work with videos, including the videogular-youtube plugin and PDF.js to work with PDFs. Other dependencies are ui-router, ui-validate, ui-bootstrap, ui-breadcrumbs and ng-cookies.

All Angular controllers can be found in the **controllers** folder. The Angular.js file contains most global configurations including the injected dependencies, routing configuration and the HTTP request interceptor. All other controller files are used as controllers for (part of) a single page. Every controller should be clearly named for its intended purpose (for example, it should be quite clear that the CommentController contains functions to add, edit and delete comments).

HTML views for these controllers can be found in the **views** folder. There are full webpage views in this folder and a few partial views (such as the navigation bar, the comment section and a few forms) in the subfolder **partials**. CSS can be found in the **css** folder.

In the folder **js** there are a few Javascript files, most of them are plugins. The globals.js file contains a few global functions, the REST service URL and the endpoint strings. That way they can be used in every controller and only need to be updated once if it is needed.

The folder **node-modules** contains more plugin files.

3.1 PDF.js

The Course Notes section of the webapplication is built on the PDF.js platform. PDF.js is a general-purpose, web standards-based platform for parsing and rendering PDFs. We used this code to embed a viewer into the page. The code can be found in the **views/pdfjs** folder. We wrote an add-on for PDF.js named classicPdfAddOn.js which is located in the same folder. This add-on adds the functionality to highlight text in the viewer. It also adds a custom scroll function that allows you to scroll to the given coordinates on a PDF page. This add-on also allows you to indicate a highlight from an external event. All interaction of the CourseNotesController to the PDF.js viewer is done through the add-on. We also edited the source code from the PDF.js platform. The source code is in the viewer.js file. All modifications are labeled with a **CLASSIC** tag. We also changed the look and layout of the viewer itself by deleting functionalities that were unnecessary for our product.

3.2 Videogular

Videogular (http://www.videogular.com/) is an open-source HTML5 video player for the AngularJS framework. We opted to use this especially because it has a plugin for YouTube (videogular-youtube) and because of its functionality with cue points (note: these are native cuepoints from the videogular player and not the videogular-cuepoints plugin; as they have the same name, this can be quite confusing). We extended functionality of these cuepoints to also show a popover when the video time passes the time of the cuepoint. Therefor we have modified the file vg-controls.js in **/node_modules/videogular-controls**.

3.3 Setup for local development

1. Install XAMPP
2. Run XAMPP as Administrator
3. In the modules box, next to Apache, click 'Config' and select 'httpd.conf'
4. Replace in the line DocumentRoot "/home/user/www" the directory to your liked one. Use forward slashes.
5. Do the same in the line ¡Directory "/home/user/www"¿

The default DocumentRoot path will be different for windows [the above is for linux]. If after restarting the server you get errors, you may need to set your directory options as well. This is done in the tag in httpd.conf. Make sure the final config looks like this:

```
DocumentRoot "C:\alan"
<Directory "C:\alan">
    Options Indexes FollowSymLinks Includes ExecCGI
    AllowOverride All
    Order allow,deny
    Allow from all
    Require all granted
</Directory>
```

When there are problems with Allow Control Allow Origin headers, install the following Chrome extension: https://goo.gl/SpBzLM

4 Guidelines

4.1 Commits

We will use four kinds of branches:

- **Master**: Every release corresponds with a merge of the development branch in the master branch.
- **Hotfix**: If a release contains a bug, start a hotfix branch from the master branch. When fixed, merge with master and delete.
- **Development**: Commits to this branch can be done for features that take less than a day and will not cause any inconvenience for other team members.
- **Feature branches**: If development of a feature takes longer than 1 day or might cause inconvenience for team members, make a new feature branch from the development branch. Implement the feature. Merge with Development and delete.

4.2 Releases

When releasing on the master branch after merging with the development branch, it is important not to forget to change the `HOME_URL` in *Config.java*, *TestConfig.java* and *ITConfig.java* to the URL of the release instead of the development webpage. The same applies to the URL of the REST service located at the top of the *globals.js* file.

4.3 Testing

- **Shared responsibility**: everyone is responsible for testing and quality. If you write a class you should also write test code for this class.
- **Test automation**: all types of tests (unit, integration, functional, acceptance) should be automated. Manual testing will only be used for usability testing.
- **Test management**: test cases, code, documents and data will be treated with the same importance as production code. This means that test code should also be documented.
- **Test independence**: at the beginning of each test as well as at the end the database should be empty. This ensures tests will not depend on each other and a test cannot cause failures for other tests.

4.3.1 Writing test code in Java

- Each test unit must be fully independent. Each of them must be able to run alone, and also within the test suite, regardless of the order they are called. The implication of this rule is that each test must be loaded with a fresh dataset and may have to do some cleanup afterwards. This is usually handled by setUp() and tearDown() methods.
- Names of functions in test code should clearly express: the method being tested, the expected input and the expected result. Variable names should express the expected input and state. Examples:
  - public void testSum_NegativeNumberAs1stParam_ExceptionThrown(int negative_number, int positive_number)
  - public void testSum_NegativeNumberAs2ndParam_ExceptionThrown(int positive_number, int negative_number)
  - public void testSum(int positive_number1, int positive_number2)
- We will use JUnit

Example:

```
import org.junit.Test;
import static org.junit.Assert.*;

public class TestSum {
    public void testSum(int positive_number1, int positive_number2){
        int sum = positive_number1 + positive_number2;
        assertEquals(sum, sum(positive_number1, positive_number2));
    }
}
```

4.3.2 Test types

Test	Description	Guidelines
Unit	Testing that verifies the implementation of software elements in isolation	Write unit test for all testable classes
Integration	Testing in which software elements, hardware elements, or both are combined and tested until the entire system has been integrated	Write integration tests for interaction between controllers and database, and entire interaction from API to database
Acceptance (System)	Testing based on acceptance criteria to enable the customer to determine whether or not to accept the system	Write Selenium tests for all acceptance tests

4.3.3 Test execution

Tests can be executed in Netbeans/Eclipse with JUnit. Please test the code before pushing to Git. Tests will also run automatically with Jenkins when pushed to the development branch. For some local tests a local database is needed. The section database in the readme of the projects contains instructions on how to set up a local database. Acceptance tests will not be run on the server but will be run manually every week by the test expert on Sunday. To run the acceptance tests we use the Selenium plugin for Firefox.

4.4 Code Quality

To ensure and evaluate the code quality of the project SonarQube was used. The code duplication should be under 5%. Blocker and critical issues should be under 5. All methods should have java doc and this is the responsibility of the developer.

5 Credentials

5.1 Google Developers

https://console.developers.google.com

5.1.1 Account

- Username: teamclassic01@gmail.com
- Password: classic01

5.1.2 Overview - Enabled APIs - Youtube Data API v3

- Usage: gives a graphical overview of the requests/day for the past 30 days
- Quotas: gives a numerical overview of the units used on the current day

5.1.3 Credentials

- Key: AIzaSyBtJIMKyKE5LJ3LhGHbvYyq8oceoyk6V5k
- Gives the key we use for the http-requests in our project to retrieve video information (duration, title, ...)

5.2 Jenkins

http://student-dp8.intec.ugent.be/jenkins/

5.2.1 Account

- Username: staff
- Password: YhX3ZSEa9YaLuHZ

5.3 Sonar

http://student-dp8.intec.ugent.be/sonar/

5.3.1 Account

- Username: admin
- Password: I9OBK8P6S1Wqp8n

5.4 Moodle

http://student-dp8.intec.ugent.be/moodle/

5.4.1 Account

- Username: admin
- Password: WNF-sqfT!x9feDXn