Faculty Elektrotechnik / Informatik

Master Thesis

Homework Manage System with Git-Support and hierarchical File database for Data management

Submitted by: Hao Gao
Matriculation number: 33101387

Supervised by: Prof. Dr. Albert Zündorf
Universität Kassel
Prof. Dr. Gerd Stumme
Universität Kassel

Kassel, March 8, 2016

Declaration of Authorship

I hereby declare that I have authored this thesis independently, that I have not used other than the declared resources, and that i have explicitly marked all material which has been quoted either literally or by content from the used sources. According to my knowledge, the content or parts of this thesis have not been presented to any other examination authority and have not been published.

Kassel, March 8, 2016

Hao Gao

Contents

1 Introduction **1**

2 Background and Related Work **2**

2.1 Git 2
2.2 Playframework 3
2.3 WebSocket 4
2.4 Related works 5

3 Design **6**

3.1 Common functions 6
3.1.1 User management 8
3.1.2 Course management 11
3.1.3 Assignment management 12
3.1.4 Communication system 12
3.2 Specific functions 13
3.2.1 Submission management using Git 14
3.2.2 Automatic evaluation 16
3.2.3 Dynamic data management 17

4 Implementations **21**

4.1 Dynamic data management 21
4.1.1 Realtime database generation 21
4.1.2 Data models 24
4.1.3 File structure 26
4.2 Integration of Git 27
4.2.1 Access control using Gitolite 27
4.2.2 Local mode 29
4.2.3 Remote mode 32
4.2.4 Automatic evaluation 33
4.3 User management 37
4.3.1 Registration system 37

4.3.2 Authentication and authorization system 40
4.3.3 Self-management 42
4.4 Course management 42
4.4.1 Creating a new course 43
4.4.2 Course enrollment 44
4.5 Assignment management 46
4.6 Communication system 48
4.6.1 Public forum 48
4.6.2 Private instant message 48

5 Tests **51**
5.1 Fluentlenium 51
5.2 JUnit 52
5.3 Result 53

6 Deployment and maintenance **54**
6.1 Deployment 54
6.2 Maintenance 55

7 Summary and Future Work **57**

Glossary **58**

Bibliography **59**

Chapter 1
Introduction

Using a web-based Learning Management System (LMS) to handle the teaching and learning process in higher education is becoming increasingly common[1]. At the same time, more and more students choose Computer Science as their major. In the winter semester of year 2013 in Germany alone, there were 33700 students enrolled in computer science, which is 4% more than in 2012[2].

The current LMS system already provides a lot of features such as creating online courses and handout course materials and assignments. However they are still short on features to meet specific needs of computer science education[3]. A common lecture format for computer science students is that they have to finish a lecture project throughout the whole semester. In order to avoid data loss, most students use a version control system (VCs) to manage their project, such as Git[4], and the lecturer would usually provides a remote Git server to let the students host their project, so that the teacher can check out the student's repository to make a review from time to time. And when deadlines for projects have passed, teachers have to register the points of each project into the LMS system by hand. This is a common scenario when using LMS for computer science lectures. Under this circumstance, the teaching staff has to maintain two systems: LMS and VCs. And they need to constantly change between these two systems to make an evaluation. On the other hand, the current LMS system requires large amounts of maintenance work. Based on the research of Debbi Weaver from Swinburne University[5], the academic staff require more local IT support to solve some technical issues with the LMS platform–a big distraction from the teaching task. In this work a new LMS system–the HMS (Homework Management System)–will be introduced using the version control system Git to manage submissions from students. With Git-based submission management, semester projects can be directly hosted within HMS, additionally a newly designed evaluation system based on the Git can automatically collect the points of the review made by the teacher and register the points directly into the student's account, so that the teacher and students no longer need to change between the systems. Furthermore, HMS uses a file-based embedded database and a unified storage structure and can run as a stand-alone java application on any sever, so that the work of maintenance can be reduced to a minimal level. The detail design of the data management will be introduced in the design chapter.

2 Chapter 2
Background and Related Work

In order to make this thesis more understandable, the used technologies and their major characteristics will be described in detail in this chapter. The first part introduces the version control system, Git, which is used to manage the submitted file by the HMS system. Then, the Playframework which the HMS is based on will be described. Finally, a communications protocol–used to build the instant message system in the HMS–will be introduced. Additionally, two related systems, which were developed in the Faculty Software Engineering Department of the University of Kassel will be discussed.

2.1 Git

Git is a widely used version control system. It was initially designed and developed in 2005 by Linus Trovalds with the following features:

Distributed

Every client will fully mirror the repository when they checkout. If the server dies, any of the repositories of the client can be used to restore the repository on the server, so that a distributed version control system can avoid the risk of losing all the data.

Doesn't save differences

Git does not store the differences between the state of a file. Instead it will take a snapshot, every time a user checks in, and it will memorize the state of the files at that moment and save a reference with it.

No server needed

Most operations in Git can be operated locally. It doesn't need any information from other sources. The changes made offline can be easily loaded onto the server later.

Data integrity

Git uses SHA-1[6] hash to check-sum files and directories before they are stored–the hash value that will be referred to them and saved later into the database of Git, so that it is impossible to make changes inside a repository without Git detecting them. Besides, all the actions made by the user with Git, will only be added to the database of Git. It will not erase or modify anything, which is not undoable. So losing data is very difficult when using Git.

Workflow

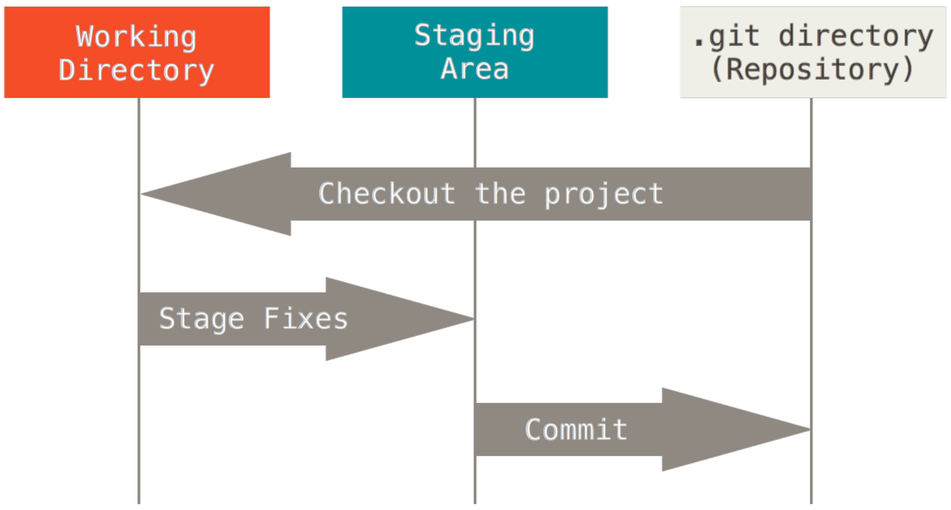


Figure 2.1: Workflow of Git[7]

Figure 2.1 demonstrates the basic workflow of Git. First the user will modify the files inside the working directory, then the snapshot of these files will be added to the staging area. When the user commits the changes, the data inside the staging area will be permanently saved inside the Git directory that stores the metadata and object database for the project[7].

2.2 Playframework

Playframework[8] is an open source web application framework based on the Java. A play application follows the MVC architectural pattern[9].

As Figure 2.2 shows, this architecture splits the web application into two layers: The presentation layer and the model layer. The model layer representation of raw data, mostly uses a persistent mechanism such as database at this layer. The presentation

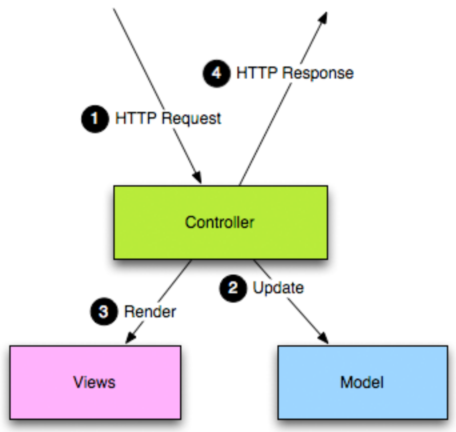


Figure 2.2: MVC architecture[10]

layer is divided into two parts: The view and the controller. The view renders the raw data from the model into a form, which can be interacted with by the user. The controller receives and responds to events, that normally come from the HTTP Request.

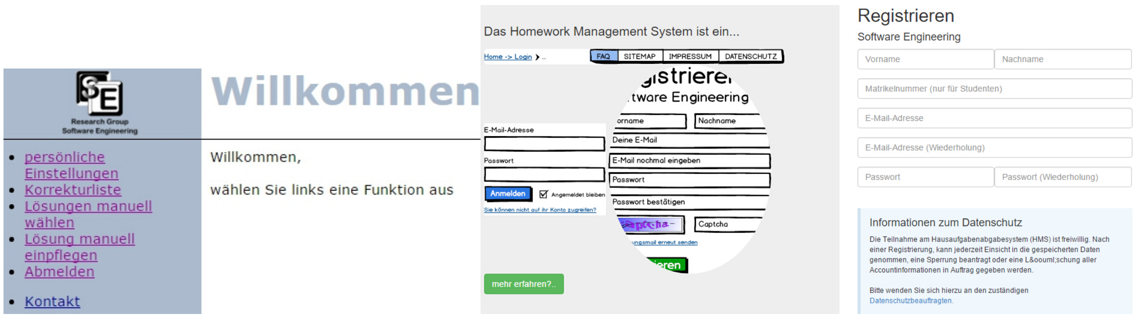
Play is also completely RESTful and stateless, fully supported by the asynchronous HTTP programming model so that the concurrent real time data can be easily integrated within play applications, and long-lived requests will also be served without interfering with other threads. Unlike other java frameworks, Playframework comes with an embedded HTTP server so that a play application can be easily executed without any container such as a Tomcat[11]. So, a play application can run on any computer with a JVM[12] as a normal, stand-alone java application and without having to do complicated server configurations[10].

2.3 WebSocket

WebSocket[13] is a protocol designed for real time communications that provides a persistent, bidirectional, fast connection between the client and server. Beside the WebSocket protocol, there is also a WebSocket API, which was developed by the World Wide Web Consortium (W3C). This API enables the developer to perform actions like opening and closing the WebSocket channels or sending and receiving messages within their applications. The WebSocket API is supported by major modern browsers[14].

2.4 Related works

Previous to the HMS system–introduced by this work–there were already two projects that existed at faculty software engineering. On the left side of Figure 2.3 is the first homework management system used inside the faculty. It is a simple, static php page attached to the lecture homepage combined with a sql database. It only provides basic functions: user registration, homework submission, solution submission and evaluation registration.



Informationen zum Datenschutz

Die Teilnahme am Hausaufgabenabgabesystem (HMS) ist freiwillig. Nach einer Registrierung, kann jederzeit Einsicht in die gespeicherten Daten genommen, eine Sperrung beantragt oder eine L&oouml;schung aller Accountinformationen in Auftrag gegeben werden.

Bitte wenden Sie sich hierzu an den zuständigen Datenschutzbeauftragten.

Figure 2.3: Earlier systems

On the right side of Figure 2.3 the second system is shown, which is more advanced. It is a stand-alone web application based on Playframework, and provides more function than the first system, such as lecture management, assignment management, and a simple feedback system. However it still has some weak points which were discussed in Chapter 1; for instance, neither systems provide an easy and quick solution to backup the semester-related data, nor do they consider the special nature of the homework in the area of computer science, which contains a lot of source code and multiple files.

In the rest of this thesis, a new HMS system will be introduced, which provides a new way to manage the activities around homework–specially the homework of computer science and including a new approach to manage the data persistently so that the cost of running and maintaining the system can be minimized.

3 Chapter 3 Design

HMS is a web platform to manage the activities related to homework including handing out the related materials to the students, collecting the handed in homework and managing the evaluation of the assignment. In this chapter the detailed design of functions to support those activities will be discussed. Furthermore the problems of the current alternative system will be analyzed and the solutions to those problems will be introduced.

The functions of the HMS system are divided into two parts: The first part is to develop the common features, which are similar to other web platforms, for instance, the "register a user account" feature. The second part of the design is to develop the special core features–the dynamic data management and the git based homework management–which make the HMS system a better platform compared to others.

3.1 Common functions

The main task of HMS is handling the process of handing in and handing out the homework between the students and teachers. As Figure 3.1 shows, there are several steps within this process from the prospective of the user. First, considering the user role of students, the student should be able to register a HMS account. After logging into the system, the students can browse all the available courses in the system and subscribe to their target course. Then it is possible for the students to view the homepage of the course and download the available homework, and finally, uploading the solutions to the homework accordingly. The process of uploading homework to the system is finished on the student's side. Now take a look at the side of teacher and assistant. Beside the registration and log in process, the user group of teacher can create new course and new assignment materials and also collect the handed in homework and give them back to the students once the evaluation is finished. The assistants, however, cannot create a new course, but should be able to add new assignments and evaluate the handed in homework as well.

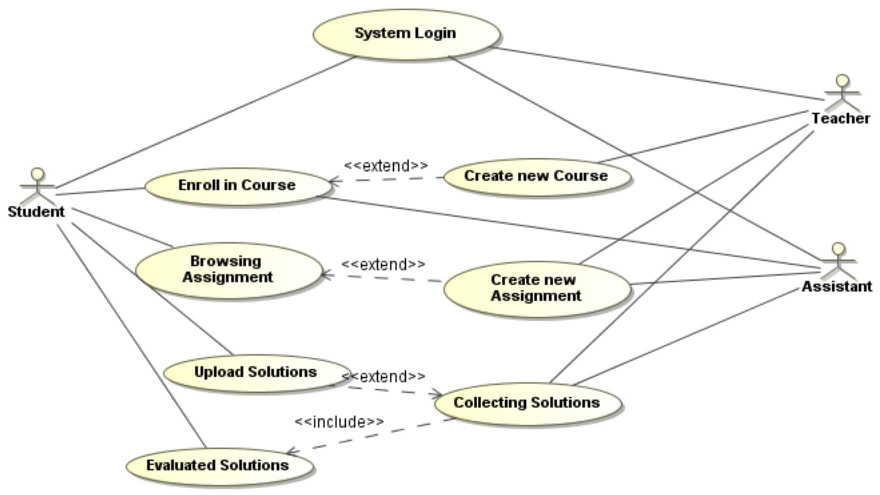


Figure 3.1: The use case from the perspective of different user roles.

In summary, in order to realize the main task of the HMS, it should have the following capabilities:

1. User management including "Registration", "Role based access control" and"Self-Management"
2. Course management including "Creation of Course","Modification of Course"
3. Assignment management including "Creation of Assignment", "Distribution of Assignments", "Collecting of Assignment" and "Evaluation of Assignment"

Besides the above listed capabilities, the HMS should also provide a way for the students and teachers to communicate with each other. Foremost, a private message system will be needed for the students and teachers to exchange information about problems with assignments or evaluations individually. Also, the new message system should work as an instant message system, and, in this way, the questions or the problems between the students and teachers can be resolved more efficiently. Second, a course forum is also not a bad idea, since a common scenario is that more students may have the same question for a new assignment; if a student writes a new post with this question, and the teacher gives an answer to the question, the other students with the same question can also get the answer and avoid asking the same questions again. This saves time on both sides. So the communications system for the HMS has two parts:

1. Instant private message system
2. A public course forum for each course

3.1.1 User management

The user management of the HMS system consists of several modules (Figure 3.2): First is the registration module. With this module, the user can use their email address to register an account in the HMS system. Second is the self-management module. After the user has logged into the system, they should be able to change their email and password or other personal details. Third is the user role control module, which is necessary for the system to arrange the proper functions for the current user. Based on their user role, the user obtains a start-up role at registration. Later on, the user role can be changed by the system administrator. In the rest of this section, all the modules will be discussed in detail.

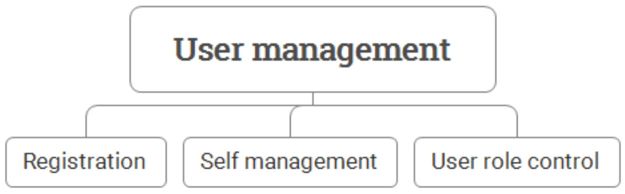


Figure 3.2: The modules of user management

Registration

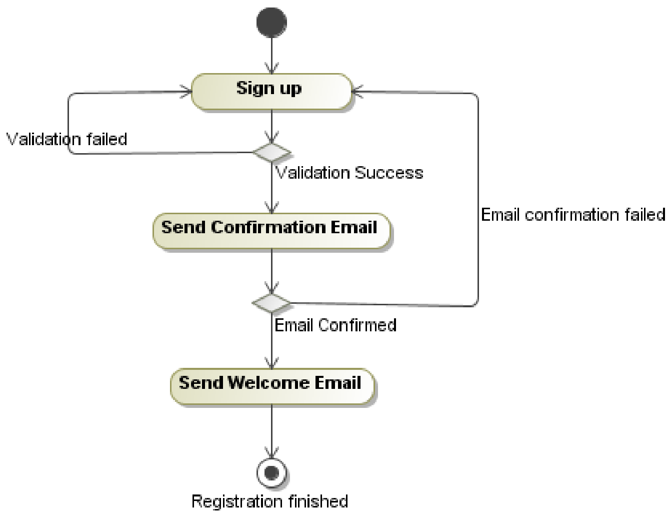


Figure 3.3: Registration activity

Figure 3.3 shows the workflow when a user registers a new account. The user first uses the registration form to fill in all the relevant information–the password, email, and student number–after clicking the sign-up button, and the whole process begins.

First the validation of the registration form will be performed to check whether the user has filled in all the required fields and without error. Once the user has passed the form validation step, the HMS system will then send a confirmation email with a confirmation URL to the email address from the registration form. Once the user clicks the confirmation link, the user will be redirected to the website and can directly start using the account. Otherwise, the user has to start the registration process over. The step of email address confirmation is important because this procedure allows the system to check that the user actually signed up for the account and guarantee that the email of this user is valid and ready to receive the system information.

Self-management

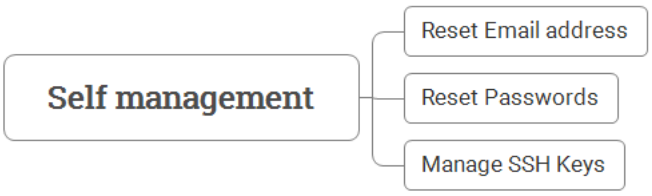


Figure 3.4: The functions of self-management

It is very common that users may forget their passwords or, even worse, their registered email becomes invalid, so it is necessary to develop functions that users can use to reset their emails and passwords. The resetting of email addresses or passwords works similar to registration. If a user chooses to reset their email address, first they will be asked to type in the new email address, and after that, the user clicks the reset button, and the system should send a new confirmation email to the new address, where the user clicks the hyperlink in the email, and a new web page will be generated so the user can confirm the address change. If a user chooses to reset their password, they just need to click the "reset password button", and the system will send another confirmation email with a hyperlink that the user can use to type in their new password. Besides the modification of passwords and email addresses, there is another function that should be added to self-management; thus, the HMS system uses the Git server to control access to the homework files, and this Git server use ssh to authenticate the connections. The system should also provide a function where users can add their ssh public key to the Git server, so that they can connect their computer directly to this Git server. Figure 3.4 gives an overview of all the components of user self-management.

	Admin	Default	Students	Assistants	Teachers
System Functions	●	○	○	○	○
Create Lecture	○	○	○	○	●
Create Assignment	○	○	○	○	●
Evaluation	○	○	○	●	●
Join Lecture	○	○	●	●	●
Hand in Homework	○	○	●	○	○
Forum & Message	○	○	●	●	●
Self Manage	○	●	●	●	●

Figure 3.5: Different user roles in the HMS system

Role-based access control

HMS system users have different roles to perform different actions: for instance, a teacher can create a new course but students cannot, so it is important to have a subsystem to distinguish the user roles, so that the HMS can serve proper functions for the user.

Figure 3.5 shows all the user roles in the HMS system: first, is the *system admin*, whose job is to manage other user's role and the system data (backup the database and related files in certain times). Any other functions of the HMS are irrelevant to the system admin. Second is the *default user*. The user with this role cannot do many things other than updating their personal data (passwords and email). The third user role is *students*, where, with a student account, the user can browse all available courses, join the course, and download and upload homework, all while using the communication features like "chat" and "forum". The fourth user role is *assistants*. Besides all the functions of students, assistants can review all the student's homework and make an evaluation. The last one is the *teacher* role. The teacher account has all the functions of the assistant account but additionally the teacher can create new courses.

In the rest of this thesis, the use of teacher, assistant and student will specifically indicate the user group with a certain role. For instance, teacher means a user group with user role "teacher".

3.1.2 Course management

The typical course provided by the faculty of software engineering and their evaluation methods are listed as follows:

- Bachelor Programming Methodology: Evaluation through final exam
- Bachelor Design Pattern: Evaluation from multiple sub project
- Master Software Engineering 2: Evaluation from semester project
- Master Compiler Construction: Evaluation from multiple sub project
- Master Graph Model : Evaluation through final exam

The above courses can be divided into two categories: In the first type, of course, the students have to hand in various homework, and the points gained from that homework are usually used as a prerequisite for the final exam. In the second type of course, the students will get a semester assignment–normally a whole software project–and the points gained from this project are usually the final points for the course. Additionally, every student will get a Git repository after they have signed in as users on the course, and this repository will also worked in two modes according to the type of course. The details of Git working modes will be discussed later in section 3.2.1.

Type	Precondition to exam	Git repository Mods	Evaluation
Type I	1. Students have to hand in a minimum amount of valid homework 2. Valid homework requires normally for students to gain more than 50% on an assignment 3. A student should gain at least 50% of total points on the final exam	Git repository works under local mods (Student can only hand in the homework through the course homepage)	need detail evaluation of assignments (number of valid hand-ins, percentage of gained points)
Type II	None, Students just need to hand in the final project (may consist of a sub. project)	Git repository works under remote mods (Student can use the course repository as any remote Git repository)	Only final evaluation (or multiple sub.evaluation)

Figure 3.6: Features of different types of courses

Figure 3.6 shows the different features of different types of courses.The functions of ”course management” should take all the features from above into consideration.

3.1.3 Assignment management

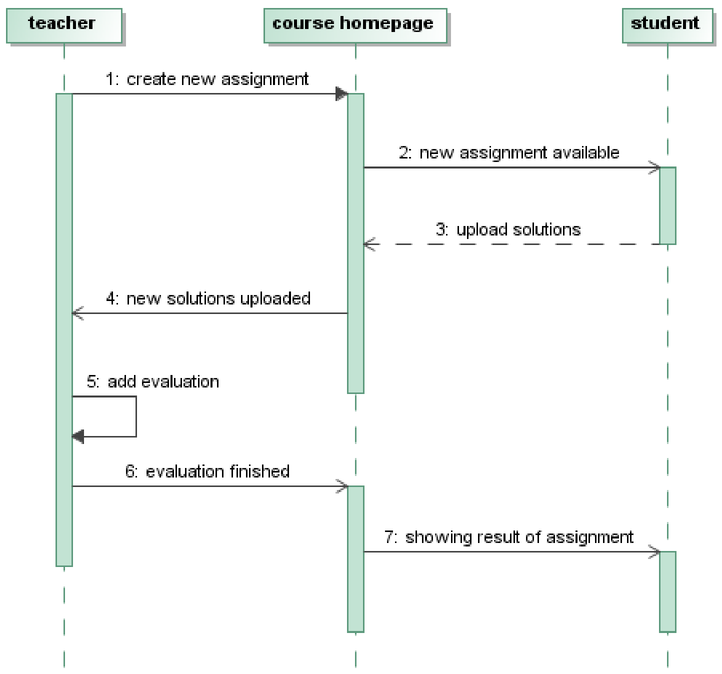


Figure 3.7: Sequence diagram for assignment management

Figure 3.7 shows the sequence of handing in and handing out a assignment. After creating a new course, the teacher can start adding assignments to the course. When a new assignment has been successfully added to the course, the student's course homepage will show this new assignment with a downloadable link, where the student can download the assignment and start working on the new assignment. Later, when students have finished the assignment, the solution can be uploaded to the system manually or directly pushed into the system using a Git client, based on the lecture type. Then the teacher can start reviewing them. When the evaluation has been successfully saved into the system, the student will get the results at the course homepage.

3.1.4 Communication system

Forum

The HMS system has a standard client-server structure, where in both the client and server communicate with each other over the internet using HTTP protocol[15]. HTTP has a typical "Request-Response" pattern; the web client sends a request to the web server, and the web server serves a response according to the web client request. It is a simple but powerful solution to provide a two-way conversation for two parties over one channel[16]. The forum function within the HMS system works also along this pattern.

Instant message

The standard HTTP protocol, however, is not suited for the instant message system, because of the Request-Response message pattern, wherein the user has to manually ask for a content upgrade. But an instant message system needs to automatically update the chat contents on both sides of a conversation while a new message has been added[17]. Therefore, a full-duplex communication system, "WebSocket"[18], will be used to back up the message module. Figure 3.8 illustrates a simple scenario of a dialog based on WebSocket. After logging into their HMS account, user A and user B are both connected to the HMS web socket server. Later on, user A sends a new message to user B. First, the request from user A passes to the WebSocket server, the server processes the request accordingly and serves the response not only to user A but also automatically to user B, ensuring both sides of the conversation can have their message received in real time.

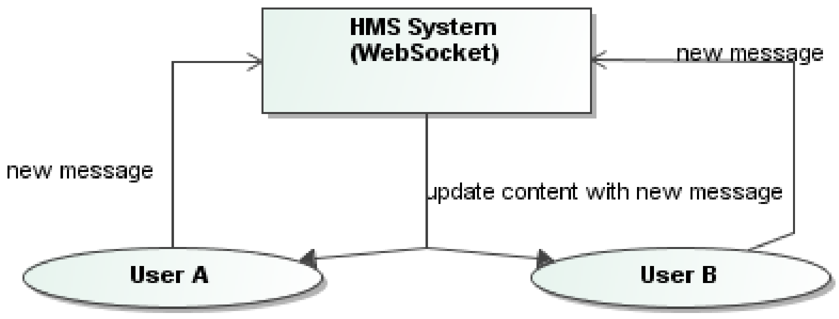


Figure 3.8: Web socket message pattern

3.2 Specific functions

There are three major problems in other homework management platforms:

1. Management of file submission
2. Centralized data persistent (single database)
3. Complicated backup process

The first problem is the management of the student's submission, because, besides simply uploading the file to the server, there will usually be more functions that need to be added to support different needs related to hand-in homework. For instance, the documentation of Moodle[19] has suggested several submission types related to file submission[20]:

1. Student submits work and the teacher downloads it later

2. Student submits work multiple times
3. Student submits work with a response

And the teacher needs to set several settings to make these types work properly, which introduces unnecessary extra cost not only for development but also for the user of the system. On the other hand, the teachers may need to trace the work history to comprehend the work of students (avoiding plagiarism), which is not provided by the current system.

The second and third problems are related to each other, because all the data is persistent in a single database, and the backup procedure is complicated and time-consuming. All these platforms, including HMS, are designed for a faculty of a university. Once in a while the faculty need to archive or backup the old data from the past semester. A possible way is to query out all the related data based on the semester, then dump these data into a file and save the course-related files somewhere else. Moodle also uses this approach to make a course backup[21]. Since the amount of data will increase rapidly after years of usage, this approach can only consume more time.

The HMS has introduced two new approaches to avoid the problems from above:

1. Git-based file submission
2. Dynamic data management

These two methods will be introduced and described in the rest of this section.

3.2.1 Submission management using Git

Instead of uploading a file to a simple folder on the server, the students will obtain a Git repository when they first enter the course. Using a Git repository that is different than a normal file folder has several advantages: First, the implementation of different submission types is not needed anymore and don't need users to set extra settings.

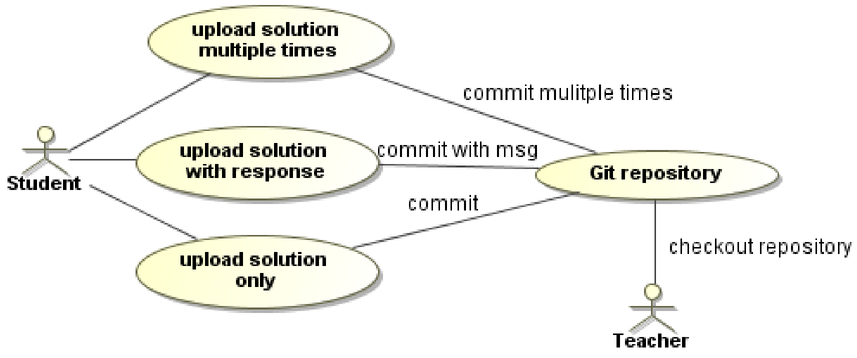


Figure 3.9: Workflow of file submission using Git

Figure 3.9 shows the use case for all submission types[20] when using the Git repository. No matter what submission type was chosen by the user, the only difference is how many commitments were made. Especially by the submission type "upload solution multiple times". Since the nature of a Git repository is to track the history of all file changes, the teacher can easily review all the changes that were made between the uploads with any Git client. Second of all, using a Git repository in a "Type II" course is more convenient for students and teachers to manage the project, since the assignment of "Type II" course is usually a semester project, the students can use this repository to host their project directly on the HMS so that the teacher won't need to set up an extra Git server for students.

Since there are two types of courses (Figure: 3.6), the Git repository will also work under two modes to fit the property of the course:

1. Local modes (for "Type I" course)
2. Remote modes (for "Type II" course)

Local modes

For the "Type I" course students will get different assignments throughout the whole semester, and every assignment needs to be evaluated individually, so the repository should have a proper file structure to distinguish the student submissions based on the assignment. Figure 3.10 shows the file structure inside the local Git repository. With this file structure, the teacher can manage the student submissions more efficiently.

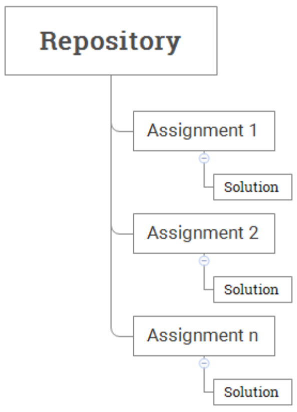


Figure 3.10: File structure of a repository in local modes

In order to maintain this file structure and avoid introducing unnecessary errors, the address of the Git repository is hidden from the student. Furthermore students don't have access rights to the repository at all; they can only use the course homepage to upload their solutions.

Remote modes

For the "Type II" course students will obtain the repository address and its full access rights as soon as they join the course, and use this repository to host their semester project. This part works similarly to using a remote repository. Students can push the changes from their local machine directly into the HMS server, without logging into the HMS system.

3.2.2 Automatic evaluation

Until now, evaluating a submission from a student requires a teacher or assistant first to check out the repository of a student using a Git client and make an evaluation before logging into the HMS system using the function described in Section 3.1.3 to register the score of the student. Constantly switching between two working systems is not only inconvenient but can also easily introduce errors when registering points.

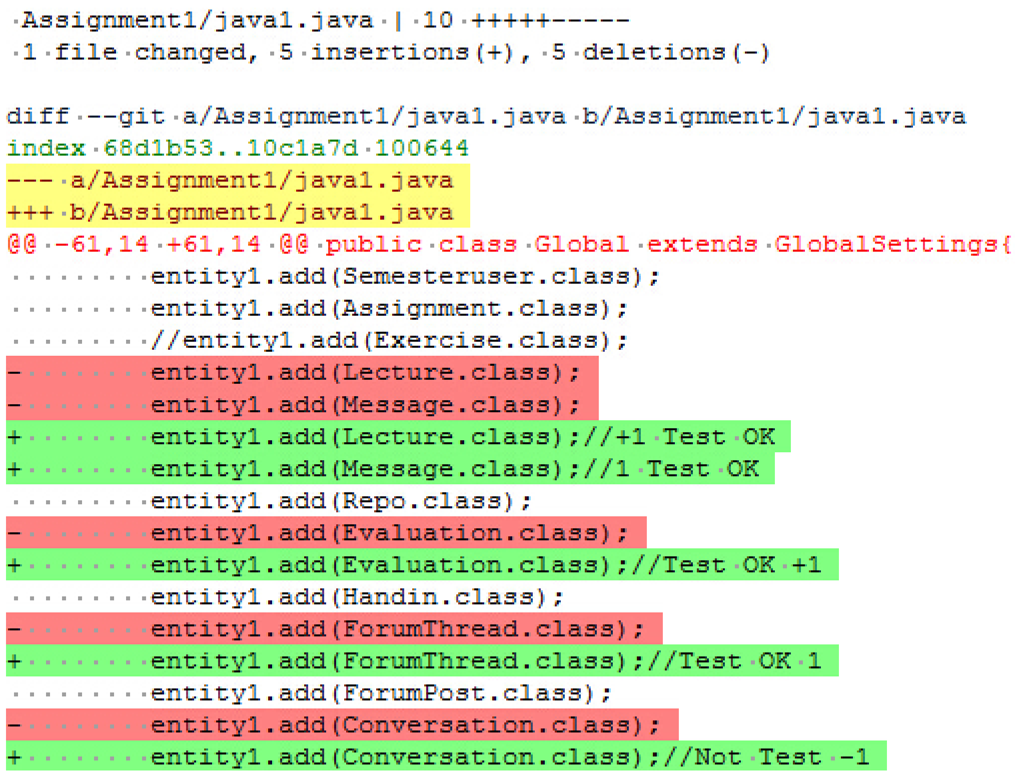
```
List<Class> entity1 = new ArrayList<Class>();
        entity1.add(Semesteruser.class);
        entity1.add(Assignment.class);
        entity1.add(Lecture.class);//+1 Test OK
        entity1.add(Message.class);//1 Test OK
        entity1.add(Repo.class);//-1 Test Not OK
        entity1.add(Evaluation.class);//Test OK +1
        entity1.add(Handin.class);
        entity1.add(ForumThread.class);//Test OK 1
        entity1.add(ForumPost.class);
        entity1.add(Conversation.class);//Not Test -1
```

Figure 3.11: Evaluation of a submission

The second advantage of using Git to manage the submission is that it makes the process of registering the points and comments fully automatic and avoids the situation of changing the working system. Based on the results of interviewing the colleagues of the software engineering faculty, the common behavior of reviewing and evaluating a student submission is to add comments at the end of a line as shown in Figure 3.11.

If the reviewer commits the changes, these changes can be easily retrieved by comparing the student's submission and its reviewed version made by the reviewer using Git command `git diff` and the results will be saved in Unified Diff Format[22].

Figure 3.12 shows the changes made by the reviewer in Unified Diff Format. The two-line header which starts with symbols "- - -" and "+ + +" indicate the changes that have been made in the file *java1.java*. The next line shows the exact change location of the files, and, in this case, the changes begin from line 61 to the next



```
·Assignment1/java1.java·|·10·+++++-----
·1·file·changed,·5·insertions(+),·5·deletions(-)

diff·--git·a/Assignment1/java1.java·b/Assignment1/java1.java
index·68d1b53..10c1a7d·100644
---·a/Assignment1/java1.java
+++·b/Assignment1/java1.java
@@·-61,14·+61,14·@@·public·class·Global·extends·GlobalSettings{
·········entity1.add(Semesteruser.class);
·········entity1.add(Assignment.class);
·········//entity1.add(Exercise.class);
-········entity1.add(Lecture.class);
-········entity1.add(Message.class);
+········entity1.add(Lecture.class);//+1·Test·OK
+········entity1.add(Message.class);//1·Test·OK
·········entity1.add(Repo.class);
-········entity1.add(Evaluation.class);
+········entity1.add(Evaluation.class);//Test·OK·+1
·········entity1.add(Handin.class);
-········entity1.add(ForumThread.class);
+········entity1.add(ForumThread.class);//Test·OK·1
·········entity1.add(ForumPost.class);
-········entity1.add(Conversation.class);
+········entity1.add(Conversation.class);//Not·Test·-1
```

Figure 3.12: Unified Diff Format

14 lines. The line begins with symbols "-" and "+" contains the actual difference between the two versions[22].

The Unified Diff Format can be easily parsed based on these specific symbols, so that the actual changes made by the reviewer can be extracted. After analyzing the comments made by the reviewer, most of the comments have the following formats:

1. //points text
2. //text points

The comments either start with a number or end with a number, using the regular expression, the points can be collected programmatically. This makes the automatic evaluation possible. The only precondition is that the reviewer has to commit the changes based on the submissions from students, and using the comments format from above, so that the points can be automatically collected by the system and registered into the database without logging into the system.

3.2.3 Dynamic data management

To do a backup or achieve the whole process as easily as possible, the HMS system introduced a new approach to manage the data. First of all, the HMS system uses multiple databases instead of one central database.

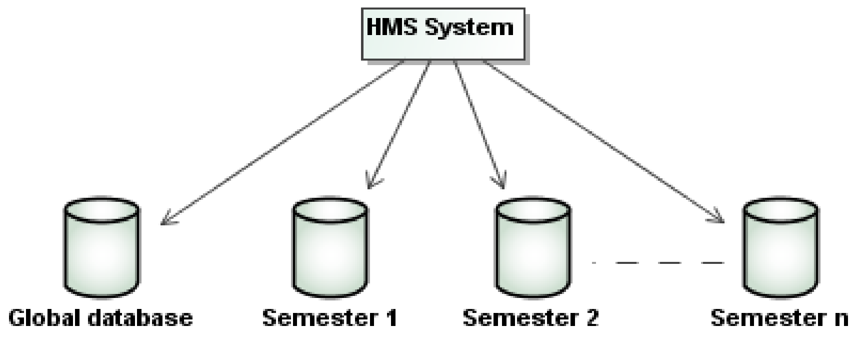


Figure 3.13: Multiple databases based on semester

Figure 3.13 shows the database structure in the HMS system. The global database contains all the authentication data including ssh public keys, user names, user emails, user role and passwords. This database is only used for the user management (Section 3.1.1). Another database is the semester database. Since the data structure of every semester is the same, the data model for each semester can be reused in every new semester. Every time a new semester begins, a new clean database for this semester will be generated using the same data structure in real time. Therefore, in this case, there is no need to query the semester data to back up the database.

Also, the database in this project should work under the embedded modes using file storage. The embedded database runs directly in the application that uses them, and it requires no extra server and no maintenance for the database itself. Another advantage of the embedded database is the speed, because all the database operations happen inside the application process[23]. Since all the relevant data of the database are saved in a single file, backing up the semester database only means copying the database file somewhere else.

In this project, the H2[24] database engine will be used as the default database engine. First, H2 supports the embedded mode (file storage) and it is purely written in java. Another reason is that the H2 database supports a mixed mode (Figure 3.14); that is, a combination of the embedded mode and the server mode. The first application (in this case, the HMS) will use the database as an embedded mode, but it also starts a server so that the other application (a SQL query tool) can still side-load the database. A normally embedded database runs within the application, so it is hidden from the end user[23], and so there is no way the user can side-load the database. But in real life, maintaining the HMS system–directly accessing and manipulating the database using a SQL query tool–is sometimes more efficient than the usual routine. It is also important to notice that if the application is shut down, the server mode will also close all the connections[25], and therefore, side-loading a database using remote mode can only take place when the HMS is still running. However the database file generated by the

H2 engine can still be loaded from the H2 web-based managing tools without needing the HMS system to be online. This feature is especially important for an archived H2 database, meaning that all the contents of the database can still be freely accessed without extra work

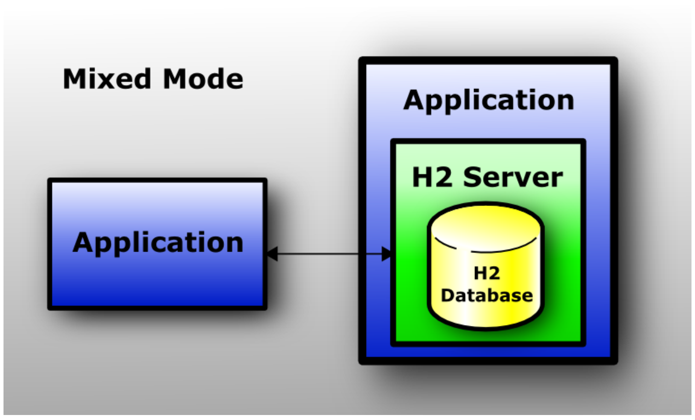


Figure 3.14: Mixed Mode of h2 server[25]

Besides backing up the database, the files from student submission and course materials should also be backed up at the same time. Using a unified file saving structure can make this process easier.

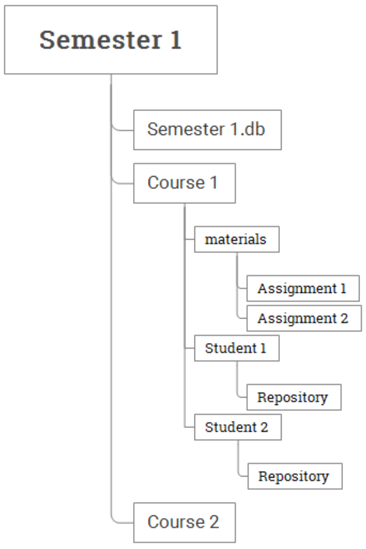


Figure 3.15: Semester file structure

Figure 3.15 shows the file structure for one semester. The uploaded course materials and student's repository are saved with the database file under the same folder. Using a clear file structure to host the uploaded files along with the commit history of the student's repository can help the maintainer of the system more easily distinguish the files without looking into the database. Combine these two methods, and the backup procedure of the HMS system only needs the maintainer to relocate the semester folder, without doing any database operation.

This approach requires the system to create a database in the run-time. Therefore the standard database configuration of Playframework cannot be used. The details of implementing dynamic data management will be demonstrated in the next chapter.

Chapter 4

4 Implementations

In this chapter the concrete implementation including used tools, extra library and also the methodology to fulfill the requirements of the functions from Chapter 3 will be presented. Because the data persistent and management are preconditions for other functions to work, they are the key features of the HMS system. The implementation of dynamic data management and Git-based file submission will be discussed first.

4.1 Dynamic data management

Dynamic data management consists of two parts: First, the HMS system can dynamically generate a new database every time a new semester starts, so that the created database will only contain the data related to that semester. Second the physical files, including the students' submissions and the course materials, are saved under a unified structure (Figure 3.15), so that the usage of each file can be easily determined without involving the database.

4.1.1 Realtime database generation

The HMS system is based on the Playframework and Playframework uses Ebean[26] ORM to access the database. ORM is a technique to convert objective-oriented programming language (in this case, Java) into its persistence as a relational database, so the data can freely exchange between a java object and database table[27]. By default the configuration of Ebean and the database are done by editing the configuration file of Playframework (Figure 4.1). The developer needs to define the details of a database and define the java data model for a specific Ebean server, which will handle all the database operations. After the application has been started, there is no way to modify this configuration again in the run time. Therefore, this method cannot be used.

In order to dynamically manage the database, all of the configuration has to be done programmatically. Ebean supports database configuration programmatically. But it still needs a little modification to make it compatible with Playframework.

```
# Database configuration
# You can declare as many datasources as you want.
# By convention, the default datasource is named `default`
# db.default.driver=org.h2.Driver
# db.default.url="jdbc:h2:file:~/data/playdb"
# db.default.user=sa
# db.default.password=""
# Ebean configuration
# You can declare as many Ebean servers as you want.
# By convention, the default server is named `default`
# ebean.default="models.*"
```

Figure 4.1: Database configuration in Playframework

```java
public static void createServer(String name, List<Class> entity) {
    ServerConfig config = new ServerConfig();
    config.setName(name);

    DataSourceConfig h2Db = new DataSourceConfig();
    h2Db.setDriver("org.h2.Driver");
    h2Db.setUsername("hms");
    h2Db.setPassword("test");
    h2Db.setUrl("jdbc:h2:tcp://localhost/~/data_dynamic/" + name + "/" + name)
    config.setDataSourceConfig(h2Db);

    Path p= Paths.get(System.getProperty("user.home"),"data_dynamic",name,name+".
        h2.db");
    File f = p.toFile();
    if(f.exists()){
    config.setDdlGenerate(false);
    config.setDdlRun(false);}
    else
    {
    config.setDdlGenerate(true);
    config.setDdlRun(true);
    }
    ...
    for(int i=0;i<entity.size();i++){
    config.addClass(entity.get(i));
    EbeanServer server = EbeanServerFactory.create(config);
    }
```

Figure 4.2: Create an Ebean server in run time

Figure 4.2 is a static method which creates an Ebean server in run time. This method needs two parameters as input: First is the name of the database, usually the name of a semester–for instance, “WS2016 “. Second is a list of java class, which contains all the java data model that is related to this database. Ebean needs to use this to create the table for the database. The configuration of the Ebean server and database is straightforward: After setting all the parameters to the Ebean server and data source, the Ebean server will be created by the class “EbeanServerFactory”. In the HMS, the decision to create a new database is made when the teacher creates a new course. Creating a new course needs the teacher to give the name of the semester. After creating the new course, the request is sent to the server. The server will first check whether this semester was already registered in the semester table of the global database. If it is already registered, the course will be simply saved to that semester database. If the input semester cannot be found in the global database, a new database with the input semester name will be first created, then the new course will be written in this database. After finishing, the part of creating a database in run time must be done. There is only one thing to be added to make it work with Playframework. Since the databases are configured during the run time, when the application is restarted, it will not restore those databases automatically, because the Playframework was originally designed to use the configuration file to track the database configuration, and in this case the configuration file is empty. The workaround for this issue is to track the database file, and reload all these databases before the application has restarted. Playframework has already provided a method to do some actions before actually starting the application, and this can be used to reload the database.

```java
public class Global extends GlobalSettings{
    @Override
    public void onStart(Application application) {
        super.onStart(application);
        List<Class> entity = new ArrayList<Class>();
        entity.add(User.class);
        ...
        entity.add(SSH.class);

        List<Class> entity1 = new ArrayList<Class>();
        entity1.add(Semesteruser.class);
        ...
        entity1.add(Conversation.class);

        try {
            Server h2server = Server.createTcpServer("-tcpAllowOthers");
            h2server.start();
        } catch (SQLException e) {
            e.printStackTrace();
        }

        createServer("global", entity);
        List<Semester> database == Semester.getallsemester();
        for (int i = 0; i < database.size(); i++) {
          createServer(database.get(i).semester, entity1);
        }
        }
    }
}
```

Figure 4.3: Reload the database at application start

Figure 4.3 demonstrates the process of reloading the database. List *entity* contains the data model for the “global” database and List *entity1* contains the data model for the ”semester” database (Figure 3.13), the global database will be first reconstructed, because the name of the other semester database is saved within the global database, which then using a loop to reconstruct other semester databases. It also should be noticed that between the lines of 12 and 21 in Figure 4.2, there are additional implementations of database file detection. If the database file is present, the Ebean server should not regenerate table relations because this action will erase all the data previously saved within this database.

It also should be noticed that, because there are multiple databases, the name of the database should always be given every time when there is a CRUD (Create, Read, Update, Delete) operation that has taken place; for instance, saving a new course to the semester "WS2016", should look like: `newcourse.save("WS2016");`

4.1.2 Data models

There are two types of databases in HMS: One is a global database, used for authenticating and tracking the information of newly created semester databases and lecture repositories of students. Another is the semester database, which saves all the data relating to the course, assignments, evaluations and communications.

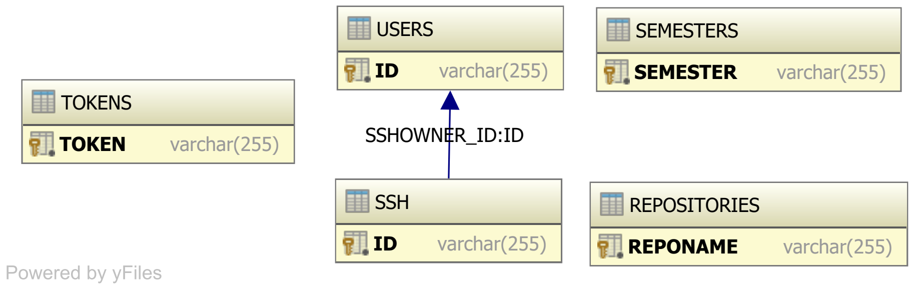


Figure 4.4: Global database

Figure 4.4 shows the data model for the global database. There are five tables defined. The first are the *Users*, which saves all the data from the user registration, for instance, the user ID, email address, password, et cetera. Second is the *SSH*, which saves all the ssh keys for the user, and because one user may have multiple ssh keys, there is a "One-To-Many" relation between the *Users* and *SSH*. The third is *Semesters*, containing all the names of the already created semester database. The fourth is *Tokens*, used to temporarily save the confirmation token for registration, change email and password actions. The last one is *Repositories*, containing all the names of the students' repositories.

Figure 4.5 represents the data models of the semester database. First is the Lecture–all the activities around the homework management is about the lecture. A lecture includes assignment (Assignment), Git repositories (Repos), a forum (Thread) and the lecture evaluation (Evaluation) for students. And each assignment contains a lot of hand-ins (Handins) from the students. Besides the lecture-related data, the data of the chat system is semester-related and independent from the lecture. At last, all these data are related to the *Semesteruser*.

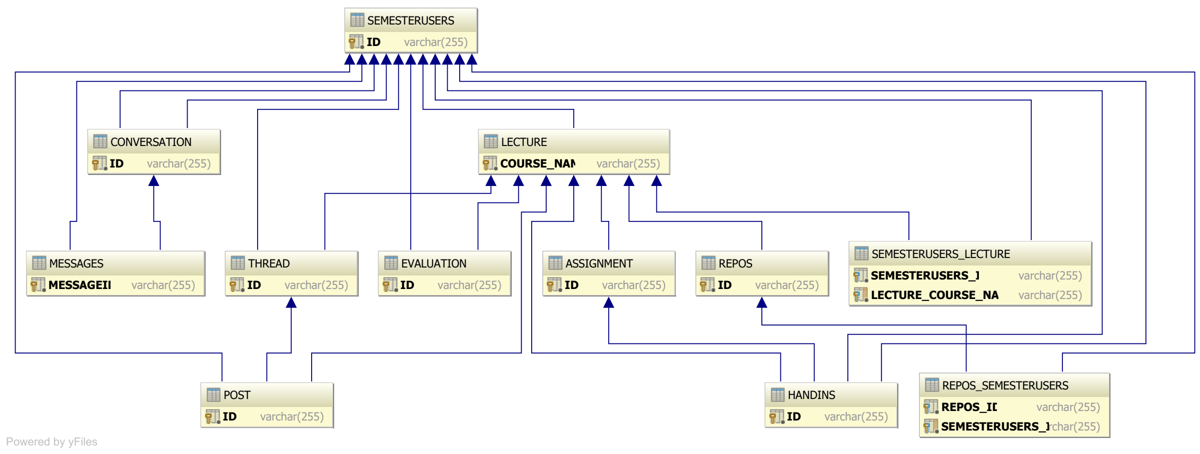


Figure 4.5: Semester database

Semesteruser and *User* are both subclass extensions from super class *Abstractuser* as shown in Figure 4.6.

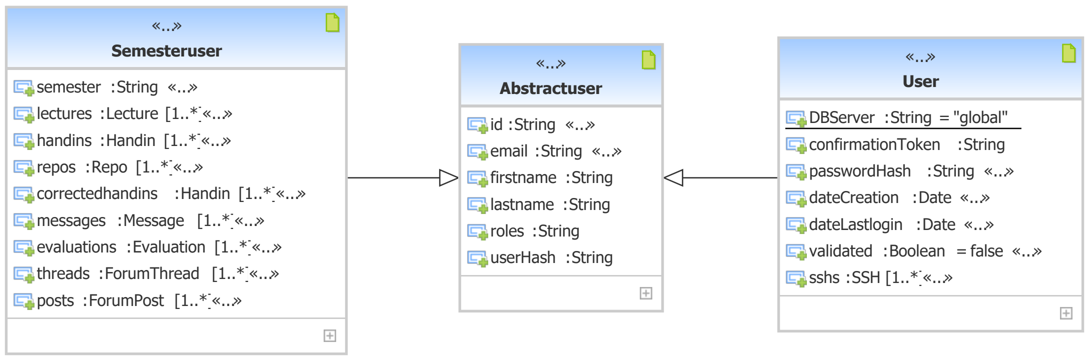

«..»
Semesteruser
semester :String «..»
lectures :Lecture [1..*] «..»
handins :Handin [1..*] «..»
repos :Repo [1..*] «..»
correctedhandins :Handin [1..*] «..»
messages :Message [1..*] «..»
evaluations :Evaluation [1..*] «..»
threads :ForumThread [1..*] «..»
posts :ForumPost [1..*] «..»
«..»
Abstractuser
id :String «..»
email :String «..»
firstname :String
lastname :String
roles :String
userHash :String
«..»
User
DBServer :String = "global"
confirmationToken :String
passwordHash :String «..»
dateCreation :Date «..»
dateLastlogin :Date «..»
validated :Boolean = false «..»
sshs :SSH [1..*] «..»

Figure 4.6: Class diagram Semesteruser

Ebean supports the JPA (Java Persistence API)[28] annotation "@MappedSuperclass". This annotation designates a class whose mapping information is applied to the classes that inherit from it, but will not be able to be generated for the mapped superclass itself[29]. The common data, which will be used both for User and Semesteruser, are defined in the class "Abstractuser ", and the database-specific data will be defined in the subclass. The field variable of the User class is related to the authentication data, for instance, the passwords and ssh keys. On the other hand, the Semesteruser class contains only the data related to the lecture activities. Using this method can first avoid redundant code, and second, the semester data can be completely separate from the other data. When a registered user wants to sign up for a new course in a semester, the common user information (e.g. user ID and email address) should be first transformed from the "User" to "Semesteruser".

```
public static Semesteruser getSemesteruserfromUser(String database,User user){
        Semesteruser semesteruser = null;
        try {
            semesteruser = Semesteruser.findByEmail(user.email, database);
        } catch (Exception e) {
            semesteruser = null;
        }

        if (semesteruser == null) {
            semesteruser = new Semesteruser();
            semesteruser.email = user.email;
            semesteruser.firstname = user.firstname;
            semesteruser.id = user.id;
            semesteruser.lastname = user.lastname;
            semesteruser.roles = user.roles;
            semesteruser.userHash=user.userHash;
            semesteruser.semester = database;
            semesteruser.save(database);
            return semesteruser;
        }
        else{
            return semesteruser;
        }
    }
```

Figure 4.7: User To Semesteruser

As Figure 4.7 shows, when a user wants to sign up for a course, the system will first find their Semesteruser information. If the Semesteruser object for the current user was not found, a new Semesteruser object will be generated in the semester database based on the user information from the global database. Using this method can keep the amount of data of the Semesteruser in each semester database as small as possible.

4.1.3 File structure

Besides the database design, the semester-related files should also be saved under a unified structure (Figure 3.15). Defining the upload path within the Playframework is straightforward: After the file is uploaded to the server, the file can be moved to the desired location by the method `FileUtils.moveFile();`.

Saving the course repository of students is a little complicated. The HMS system using the Gitolite[30] to manage the Git repository and repository created by the Gitolite is bare repository which doesn't contain a working directory[31], so it is pointless to just copy a bare repository to another place. Also the Gitolite has its own file structure and cannot be changed. So, a possible way is using Git Java API JGit[32] to make a clone of this bare repository to the desired destination.

Figure 4.8 shows the use of JGit[32] to clone a bare repository from Gitolite to a "local path", which fits the file structure of HMS. This procedure also makes submission of a file to Gitolite repository possible. The detailed usage of Gitolite will be closely discussed in the next section.

```
Repo repo=Repo.findRepoByLectureAndOwner(assignment.semester, semesteruser,
     assignment.lecture);
Git git = Git.cloneRepository();
     git.setURI(repo.repofilepath)
     git.setDirectory(localPath)
     git.call();
```

Figure 4.8: Clone repository using JGit

4.2 Integration of Git

The implementation of this function should consider following requirements:

1. Access control: students can only have access to their own repository, and the teacher or assistant of the course can fully access the student's repository when needed
2. Local and remote modes: to support two types of course (Figure 3.6)

The most important requirement is the access control over the Git repository. Because Git by itself does not do any access control, it relies on the transport medium (authentication of the HMS system) to do authentication, and file permissions of the operation system to do authorization (read or write permission)[33]. Without proper access control over the Git repository, it is then impossible to integrate the Git into the HMS. Since the basic needs of managing homework are to keep the students' submissions only between the marker and students themselves.

The first part of this section will present a method using Gitolite[30] and Java-Gitolite-manager[34] to solve the access control problem of Git repository. And the usage of Git repository for both course types will be introduced in the other part of this section.

4.2.1 Access control using Gitolite

Access control over the student's repository is pretty simple. As Figure 4.9 shows, only the students themselves have full access rights to their course repository. Teachers and assistants have to grant access rights before evaluation.

Repository Access level	Read	Write
Student	●	●
Other Student	○	○
Teacher	require permission	require permission
Assistant	require permission	require permission

Figure 4.9: Access level for different user

This access rule can be easily managed by Gitolite. After properly installing the Gitolite in the server, it will generate a "gitlite-admin" repository under current user's home folder. Within this folder there is a plain text file, which will be used by Gitolite to specify the access rules.

```
repo foo
 RW = alice bob
 R  = carol david
```

Figure 4.10: Configuration of access rules with Gitolite

Figure 4.10 is a simple configuration of the access rules for the repository "foo". For this repository, Alice and Bob both have read and write access, but Carol and David can only read the content of the repository. The server maintainer only needs to modify this configuration file and push the changes using a Git command back to the Gitolite, and all of the changes will have been adopted automatically by Gitolite. Additionally, the ssh public keys of the user of this repository are also needed to be copied into "Gitolite-admin" repository, since Gitolite use the ssh mechanism to authenticate the user.

Java-Gitolite-Manager

Gitolite gives the possibility to add access control to Git repository, but it still needs a system maintainer to edit the configuration file manually. Therefore, it still can not be directly used in the HMS system. Delft University of Technology has developed a java library, which enables developers to manage the Gitolite configuration directly from Java.

```java
ConfigManager manager = ConfigManager.create("/home/gitolite-admin");
Config config = manager.get();

User user = config.createUser("alice");
user.setKey("desktop", "ssh-rsa AAAB3Nz...");

config.createRepository("foo")
     .setPermission(user, Permission.ALL);
manager.apply();
```

Figure 4.11: Gitolite configuration from Java

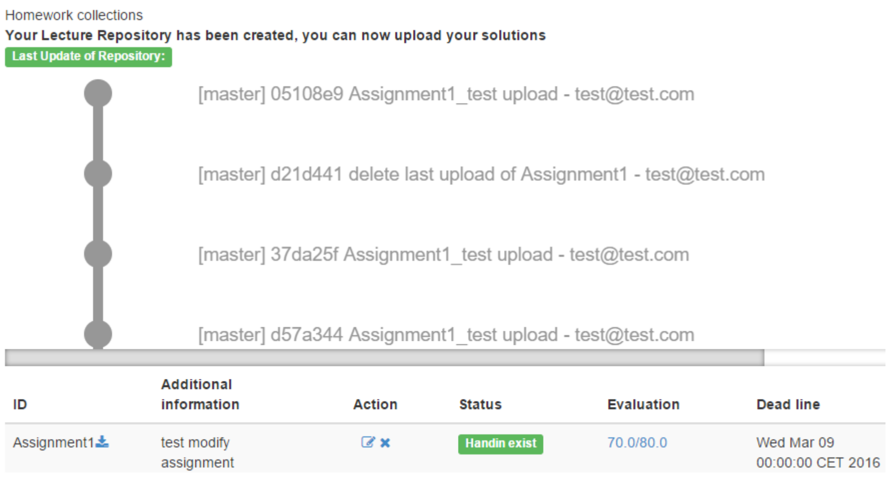
Figure 4.11 is an example of how to use "Java-Gitolite-Manager" to configure the Gitolite from Java. First, an instance of *ConfigManager* is created with the path of repository "Gitolite-admin", then a user with user name "Alice" is created alone with the ssh public key of user Alice. After the user has been created, a new repository "foo" will also be created and user Alice will be added to this repository with full permission granted.

The HMS system uses the same procedure as the example in Figure 4.11–every time a student joins a course, no matter what this course type is, a repository will be generated for this student if the ssh public of this student is present. If the teacher needs to access the student's repository, the HMS system will first get out the repository and set a correct permission for teacher use, `setPermission(teacher, Permission.ALL);`. Then teacher can clone this repository remotely.

4.2.2 Local mode

Under local mode, students need to use the course homepage to upload their solutions and the actual repository address is hidden from the student. Figure 4.12 shows the homework submission part for the course under local mode. The gray dot above shows the current status of the repository, under the indicator of repository status, is the homework area, where a student can download the assignment's material and use the predefined actions: commit and revert.

The left icon under the action column is used for uploading the solutions. The icon on the right is the predefined action for deleting the last submission; students can use this to revert their last submission.


Homework collections

Your Lecture Repository has been created, you can now upload your solutions

Last Update of Repository:

[master] 05108e9 Assignment1_test upload - test@test.com

[master] d21d441 delete last upload of Assignment1 - test@test.com

[master] 37da25f Assignment1_test upload - test@test.com

[master] d57a344 Assignment1_test upload - test@test.com

ID	Additional information	Action	Status	Evaluation	Dead line
Assignment1	test modify assignment		Handin exist	70.0/80.0	Wed Mar 09 00:00:00 CET 2016

Figure 4.12: Homework submission for course under local mode

The repository created by Gitolite is a bare repository. A bare repository is a named file directory with a ".git" suffix which contains only the administrative and control files of the repository, and it doesn't have any copy of the files that present in a normal Git repository[35]. The student's submission then cannot be directly uploaded to this repository. Also, the HMS system requires a unified file structure to save the physical files. The solution to this problem is using JGit to clone the bare repository to a normal repository at a desired location on the server.

JGit

JGit is a Java library for working with Git repository. With JGit all the operations for a Git repository can be realized from Java[32]. Besides cloning the bare repository presented in Figure 4.8, the HMS system also needs to upload the students' submission to the cloned repository and, using JGit, commit the changes and push the changes back to the Gitolite bare repository, so that the teacher can clone the student's bare repository remotely.

Commit

After a student has chosen the commit action, a dialog (Figure 4.13) will pop up. Using this form a student can upload their files with a commit message; if the commit message is empty, a default commit will be made by the HMS system.

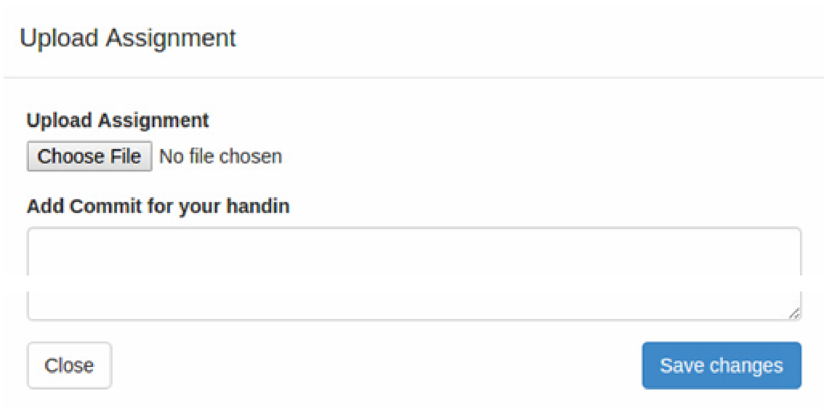


Figure 4.13: Commit dialog

Figure 4.14 shows what happened after the submission of a file and a commit message were passed onto the HMS system. First, the Gitolite bare repository will be cloned to the *localPath*. After the clone operation is finished, the uploaded file from the student will be copied into the newly created local repository. Then ,use JGit to commit the changes, and after that all the changes will be pushed through to the master branch of the Gitolite bare repository. In the end, a new "hand in" object will be generated and saved into the database under the name of the student. This object will be used to indicate that the student has submitted a solution that is ready to be evaluated.

```
public static Result handinhomework(...){
        try{
        MultipartFormData body = request().body().asMultipartFormData();
        FilePart homeworkfile = body.getFile("homeworkfile");
        if (homeworkfile != null) {
            String fileName = homeworkfile.getFilename();
            Git git = Git.cloneRepository()
            git.setDirectory(localPath);

            FileUtils.moveFile(file, new File(localPath, des+fileName));
            git.add().addFilepattern(des+fileName).call();
            git.commit().setMessage(commit).setAuthor(...).call();
            RefSpec refSpec = new RefSpec("master");
            git.push().setRemote("origin").setRefSpecs(refSpec).call();
            git.getRepository().close();

            Handin handin= new Handin();
            handin.save(semester);}
```

Figure 4.14: Commit submission

Delete Submission

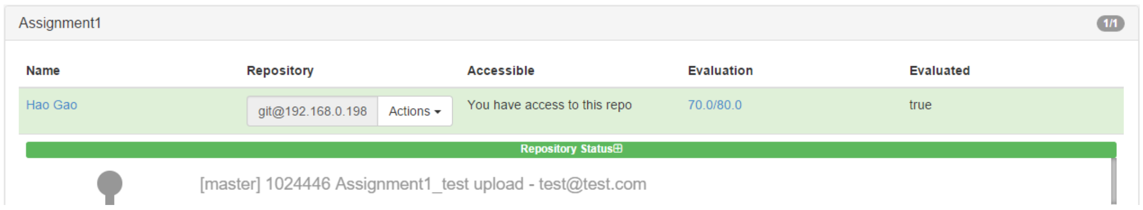
If a student didn't satisfy the first submission for an assignment, the last submission can also be deleted before the deadline. As Figure 4.15 shows, the local repository created by the submission action will be first updated to the latest state. Second, since the submission under the local modes are saved under a unified structure (Figure 3.10), the folder of relevant assignments will be deleted. In the end the delete action will also be committed by the HMS system and the changes will be pushed back to the Gitolite bare repository. Finally, the related hand in object from the submission will also be deleted and regenerate by the next submission action.

```
 Git git = Git.cloneRepository();
        String subfolder=assignment.title;
        git.rm().addFilepattern(subfolder).call();
        git.commit()
    .setMessage(commit)
    .setAuthor(semesteruser.lastname, semesteruser.email)
    .call();

 Handin handin=Handin
        .getHandinofassignmentofstudentinlecture();
    if(handin!=null){
    handin.delete(semester);}
```

Figure 4.15: Delete last submission

Check out by Teacher

After a student has submitted their solutions to the repository, teachers can check out the students' repository for evaluation after granting access to the student's repository.


Assignment1 1/1

Name	Repository	Accessible	Evaluation	Evaluated
Hao Gao	git@192.168.0.198 Actions	You have access to this repo	70.0/80.0	true

Repository Status

[master] 1024446 Assignment1_test upload - test@test.com

Figure 4.16: Checkout student repository

Figure 4.16 shows the user interface for evaluating students' submissions. It shows the address of the student repository and its status, and whether the teacher already has access to this repository. If the user has already submitted a solution, the row of this user will be in the color green; otherwise it is red. The teacher can use the action button to choose an action towards the students repository. The student repository can be directly checked out into the Git client "Source tree", or the teacher can just copy the address of the repository and check out in a favorite Git client.

The precondition of checkout for the student repository is to grant an access. When teacher press the "Grand Access" button, the method *grandaccess()* in Figure 4.17 will be executed.

```
public static boolean grandaccess(...) {
if(!admincredential.sshs.isEmpty()
    &&!studentrepo.owner.contains(currentadmin)) {
    User teacher = config.ensureUserExists(teacher.id);
    String reponame = lecture.courseName + "_" + student.id;

    Repository repository = config.ensureRepositoryExists(reponame);
    repository.setPermission(teacher, Permission.ALL);

    manager.applyAsync(config);
    studentrepo.owner.add(currentadmin);

  }
```

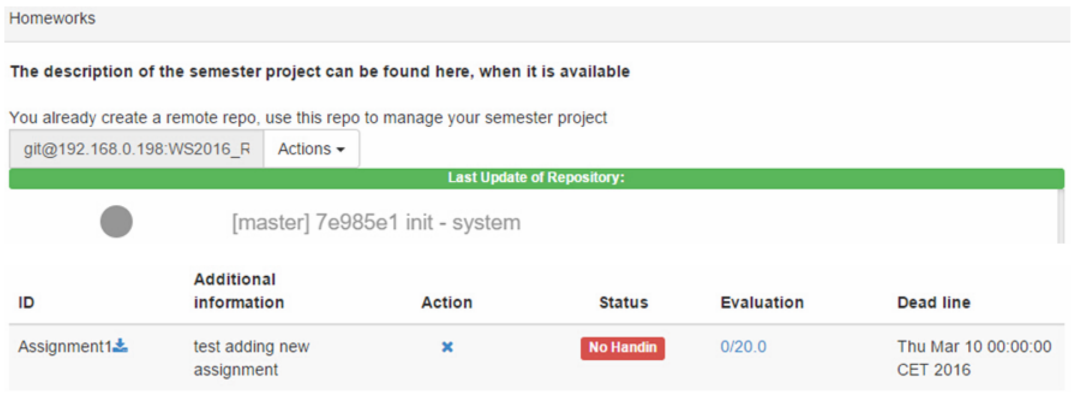
Figure 4.17: Grand access to student's repository

First the ssh key of the teacher must be present, since Gitolite needs the ssh key to authenticate the user. Second, if the teacher already has access to this repository, the system should avoid running this method again. If the preconditions are fulfilled, the teacher will be added to the repository with full permission. Also the information about the teacher already has access to this repository, and will be saved into the database.

4.2.3 Remote mode

Remote mode is much simpler than local mode. Figure 4.18 shows the user interface of homework area in a *Type II* (Figure 3.6) course under remote mode. The address of

Git repository will be directly given to the students. Students will use this repository to host their semester project, like using a normal remote Git repository. Thus, all the changes pushed by the student will be directly saved into the Gitolite bare repository, and an additional clone action of the bare repository–which is same as the submission action (Figure 4.14) in the local modes–will be performed in order to maintain the unified file structure.


Homeworks

The description of the semester project can be found here, when it is available

You already create a remote repo, use this repo to manage your semester project

git@192.168.0.198:WS2016_R Actions

Last Update of Repository:

[master] 7e985e1 init - system

ID	Additional information	Action	Status	Evaluation	Dead line
Assignment1	test adding new assignment	×	No Handin	0/20.0	Thu Mar 10 00:00:00 CET 2016

Figure 4.18: Remote mode

4.2.4 Automatic evaluation

The common behavior of a reviewer rating a student's submission is to add comments directly inside the file of a submission (Section 3.2.2). And these comments can be extracted based on the information from the Unified Diff Format, and specific points inside these comments can be automatically collected and assigned, related to the hand-in of a student by HMS.

To achieve this goal, there are three problems that need to be dealt with:

1. HMS needs to be aware when a reviewer pushes an evaluation back to repository using a Git client.
2. The comments extracted from Unified Diff are actually made by the reviewer, not by students or anyone else.
3. Points are collected from the comments and the points are registered with the correct person and lecture.

The solution to these problems will be introduced in the rest of this section.

Detect changes from repository

Although HMS already has the ability to make changes to Git repositories by using Gitolite, it cannot directly detect changes of the repository made outside of the system. Like when a reviewer pushes changes back to the repository directly using other Git clients, the HMS will not be aware of these changes. This is because the HMS using Gitolite to manage the access control over Git repositories and these repositories can be accessed remotely using ssh, which isn't supported by the Playframework.

So the HMS system has to detect the changes of the repositories actively. One possible method is to poll the file system looking for changes. However, this approach cannot detect the changes in time and is inefficient. Another way is to use a native Java API "WatchService" from JDK 7[36]. This API enables the system to register any directory with different change events, such as file deletion, file creation or file modification. When the watch service detects any of these events, they will be handled as needed. The Watch Service API can also take advantage of native file change notification implemented by the file system itself, so that polling the file system can be avoided[37].

```java
public class RepoWatcher implements Runnable {
    public WatchService watchService;
    public RepoWatcher(WatchService repowatcher){
        this.watchService=repowatcher;
    }
    @Override
    public void run() {
       try{
            WatchKey key = watchService.take();
            while(key!=null){
            key.pollEvents();
            Path dir=(Path)key.watchable();
            GitEvaluation(dir.toString());
            key.reset();
            key=watchService.take();
            }
       }catch (Exception e){
            Logger.debug(e.getMessage());
       }

    }
}

public class Global extends GlobalSettings{
    @Override
    public void onStart(Application application) {
        try {
            setWatchService();
            RepoWatcher repoWatcher = new RepoWatcher(getWatchService());
            Thread watchThread = new Thread(repoWatcher, "repoWatcherThread");
            watchThread.start();
        } catch (IOException e) {
            Logger.debug(e.getMessage());
        }
}
```

Figure 4.19: File Watch Service

Figure 4.19 shows the implementation of Watch Service inside the HMS system. Inside the global class, the method *setWatchService()* initiate a Watch Service of the default file system. Then, the object of Watch Service will pass to a runnable class *RepoWatcher*. Then a WatchKey, which contains the detailed change information, will be obtained from the *watchService.take()*. The *take()* method returns a queued key. If there are not any changes detected by the Watch Service, this method waits. On the other hand, if a change has been detected by the Watch Service, key will be not a null object, and this key will enter an infinite loop. Inside the loop, the address of changed repository will be got and passed to the method *GitEvaluation()*. This method will start the evaluation process. After this key is used, it will be cleared and wait for the next changes. The *RepoWatcher* runs in a separate thread, *watchThread*, and will be started before HMS actually runs. Everytime a new repository is created, it can be easily added to the Watch Service as follows:

```
Path addToWatch= Paths.get(System.getProperty("user.home"), "repositories",
    reponame + ".git", "refs", "heads");
addToWatch.register(getWatchService(), ENTRY_MODIFY);
```

Authentication

After detecting the changes, the next step is to determine who made the changes, because the precondition of automatic evaluation to work is to get the changes between the commits made by reviewer and students. Based on the different submission methods in the different lecture modes, the Authentication of automatic evaluation works also differently in Local and Remote mode.

a. Local mode

The situation under local mode is straightforward: Each repository will only have two committers: hms and reviewer. Because under this mode the students don't have any access rights to the repository and can only use the lecture homepage to upload the submission. Every time a student uses the homepage to upload a submission, the HMS system will commit the changes.

So under this mode, when the HMS system detects the changes, it will try to find out the name of the committer. If the name of the committer was "hms", then this is a submission from the students. Otherwise, it is the evaluation submission from the reviewer.

```java
RevWalk walk= new RevWalk(repository);

ObjectId head=repository.resolve(Constants.HEAD);

RevCommit headCommit=walk.parseCommit(head);

String committerofhead=headCommit.getAuthorIdent().getName();
```

Figure 4.20: Get the name of committer

Figure 4.20 shows a code snippet used in the HMS system to get the name of the committer using JGit API[32], since the latest change of a repository will be saved inside the HEAD which is a reference to the last commit of a repository[7]. The HMS will always try to obtain the identity information from the commit referenced by the HEAD.

b. Remote mode

Under remote mode, using only the name or email address of a committer to determine the identity becomes unreliable since, under this mode, students also have full access to the repository and the identity of a committer can be easily changed using any Git client. The results of an evaluation can be easily manipulated because of this. For instance, a student using Git push command submits a solution to the repository normally, and later on they change their name and email address to a teacher of the lecture, and make comments to their submission by themselves, and push their own commented version to the repository. The results are thus manipulated.

The solution to this issue is that when a student submits a solution to the repository of the lecture, the HMS will revoke all the access rights of this student to this repository immediately when it detects the changes of the repository, so that the next committer can only be the reviewer, because only the reviewer has access to the repository. This, however, raises another problem: How to determine that a change was a submission from a student. This requires a student to use a key word, "Assignment" inside their commit messages. If the HMS detects a change and the commit message of this change contains the keyword, HMS will think this submission is from students, and should revoke their access rights to this repository temporarily. Until the next commit, which is definitely from the reviewer, finishes the evaluation, the HMS will reassign the access rights back to the students until the next submission. However, this method will require the reviewer to not use the keyword "Assignment" inside their commit messages.

Collecting and registering the result of the evaluation

If the HEAD commit was made by the reviewer and the commit before the HEAD was made by the students or hms, then the HMS will start the evaluation phase.

```
        ObjectId newhead = repository.resolve("HEAD^{tree}");
        ObjectId oldHead = repository.resolve("HEAD^^{tree}");
        ByteArrayOutputStream changes = new ByteArrayOutputStream();
        DiffFormatter formatter = new DiffFormatter(changes);
        formatter.setRepository(repository);
        formatter.format(oldHead, newhead);
        String diffresult = changes.toString();
        evaluationResult = CommitParser(diffresult);
```

Figure 4.21: Get evaluation

Figure 4.21 demonstrates the method to obtain the Git diff result between the current HEAD commit and the commit before the HEAD into a string object. The result, which is similar to the result in Figure 3.12 will be passed to the method *CommitParse()*. In this method, the *string diffresult* will be parsed using the Java Regex[38], and the total points from the comments made by the reviewer will be calculated. In order to register the points conveniently, the name of the repository is intentionally designed as follows:

`WS2016_LocalLectureTest_7352212`

The first part is the semester information, while the second part is the name of a lecture, and the last part is the ID of the student. With this naming method, the detail information about the lecture and the student can be easily obtained from the related database, so that the evaluation result can be registered accordingly.

4.3 User management

The implementation of the user management system consists of the following parts:

1. Registration system
2. Authentication and Authorization system
3. Self-management system

4.3.1 Registration system

The registration system of the HMS is modified from the demo project PlayStartApp[39]. Figure 4.22 shows the registration form on the HMS homepage. A new user needs to at least type in their email address, first name, last name, and the password. The

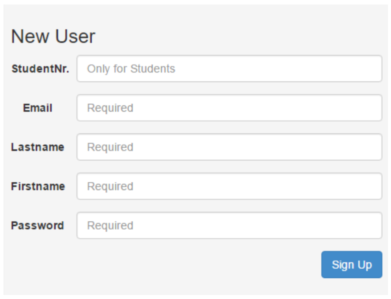


Figure 4.22: Registration form on homepage

student number is optional because of the multi-user role management of the system. Student users will get a student user role as a default if they type in their student number. Other users will get a random user ID and a "default user" role. The system administrator will change their user role afterwards. Figure 4.23 shows what happens when users click the sign up button on the registration form.

```
public static Result save() {
        Form<Application.Register> registerForm = form(Application.Register.class)
            .bindFromRequest();
        if (registerForm.hasErrors()) {
            return badRequest(index.render());
        }

        Application.Register register = registerForm.get();
        Result resultError = checkBeforeSave(registerForm, register.email);

        if (resultError != null) {
            return resultError;
        }

        try {
            User user = new User();
            if(register.id==null||register.id.isEmpty()){
                user.id= CreateExternalId.generateId();
            }
            else{
            user.id=register.id;}
            user.roles=UserRoll.Students.toString();
            user.confirmationToken = UUID.randomUUID().toString();
            user.save("global");
            sendMailAskForConfirmation(user);
    }
```

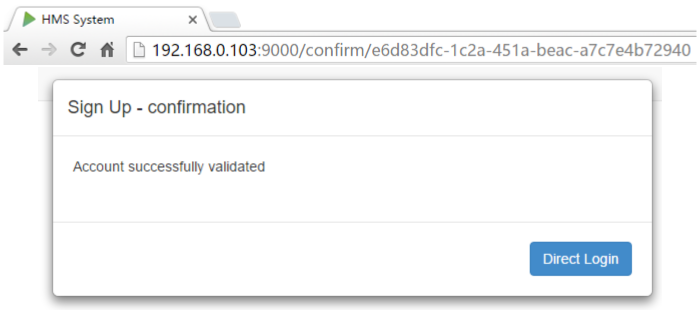
Figure 4.23: Save user registration

The sign up button is bonded with the function "Signup.save()". At first, the data from the *registerForm* will be checked as to whether all the required fields have been filled. If there is an error, the user will be redirected to the homepage and do the registration process again. Second, if the data are correct, the email address from the *registerForm* will be checked by the function *checkBeforeSave()* to insure that the email address has not be taken by other users. If both tests have been proofed, the next step is to check the user ID. If the user ID has been given, then this user should be saved with the user role of students; if the user ID is empty, the user will obtain a user role of default user. After saving the user registration data alone with the confirmation token to the database, a confirmation email with user confirmation token will be sent to the user email address by the function *sendMailAskForConfirmation()*, in Figure4.24. At this step the user data are successfully saved into the database and wait to be confirmed by the user.

```
private static void sendMailAskForConfirmation(User user){
        String subject = Messages.get("mail.confirm.subject");
        urlString += "/confirm/" + user.confirmationToken;
        URL url = new URL(urlString);
        String message = Messages.get("mail.confirm.message", url.toString());
        Mail.Envelop envelop = new Mail.Envelop(subject, message, user.email);
        Mail.sendMail(envelop);
    }

public static Result confirm(String token) {
        User user = User.findByConfirmationToken(token,"global");
        if (User.confirm(user,"global")) {
            sendMailConfirmation(user);
            user.dateCreation=new Date();
            user.save("global");
            return ok(views.html.account.signup.confirm.render(user));
        }
    }
```

Figure 4.24: Confirm registration

The user needs to use the hyperlink (contains a confirmation token) to confirm their registration. After the user has clicked the confirmation link, the confirmation token will be passed to the method *confirm()*. The system then, uses this token to find the correct user record within the database. If the user is found then a welcome email will be sent and mark this user as confirmed in the database.


HMS System
192.168.0.103:9000/confirm/e6d83dfc-1c2a-451a-beac-a7c7e4b72940
Direct Login

Sign Up - confirmation

Account successfully validated

Figure 4.25: Registration successful

If the details of the user are successfully updated in the database, the system will render a modal and the user can directly log into their account and start using the system. Figure 4.25 shows the result of a successful registration. The URL from above is the confirmation link with the user token.

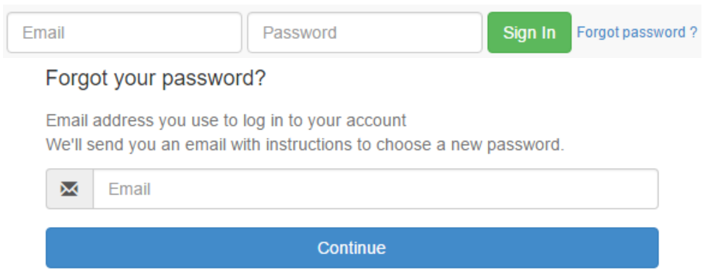
4.3.2 Authentication and authorization system

Authentication

The authentication system consists of two parts:

1. Authentication of HMS system
2. Authentication of Git server

The first part is the authentication of the HMS system. This part controls the user login activities.



Forgot your password?

Email address you use to log in to your account
We'll send you an email with instructions to choose a new password.

Figure 4.26: Login and password recovery

Figure 4.26 shows the component of authentication of a user login. The login procedure is straightforward. If the inputted email address and password have a match inside the database, the user will be redirected to the homepage. It also should be noticed that, due to security reasons, the password will be saved as SHA-256[40] hash in the database.

Another function of the authentication system is to recover the user password. Figure 4.27 shows the procedure for when the user resets their password with an email address.

```
public static Result runAsk() {
        Form<AskForm> askForm = form(AskForm.class).bindFromRequest();
        User user = User.findByEmail(email,"global");
        if (user == null) {
            sendFailedPasswordResetAttempt(email);
            return ok(views.html.account.reset.runAsk.render());
        }
        try {
            Token.sendMailResetPassword(user,"global");
            return ok(views.html.account.reset.runAsk.render());
        } catch (MalformedURLException e) {
            Logger.error("Cannot validate URL", e);
        }
        return badRequest(ask.render(askForm));
    }
```

Figure 4.27: Reset password

With *runAsk()*, system first checks whether a user with the input email exists in the database. For security reasons, if the user does not exist, the system shouldn't expose any results to the user; instead, send an email to notify the person that the reset has failed. If the user exist, the other actions will be the same as the registration, and the user will receive an email with a hyperlink with a reset token. They can click this hyperlink, and it will generate a password reset page. At this page the user can modify their password.

The second part of authentication related to the Git server: because the Git function is using Gitolite to manage the access rights of the student's repository, it has a separate authentication system using the ssh public key mechanism. The detail implementation has already been discussed in Section 4.2.

Authorization

Besides the authentication of a user, the HMS system still need to authorize the user proper access rights based on the user role (Figure 3.5). Playframework uses actions to serve HTTP requests. An action is basically a Java method that processes the data from the HTTP request[10]. If an unauthorized person can mock a correct HTTP request, this person can basically execute any actions implemented in server logic. In order to prevent unauthorized actions, Playframework comes with a built-in authenticator action called *Secured*[10]. In the case of the HMS system, some actions can only be executed by the user role of the teacher, the additionally secured class implementation should distinguish the current user role and decide whether the actions should be executed for the current user.

```
public class Securedteacher extends Security.Authenticator{
@Override
public String getUsername(Http.Context ctx) {
    User current=User.findByEmail(ctx.session().get("email"), "global");
    if(current!=null) {
        if (current.roles.equals(UserRoll.Teachers.toString())) {
            return ctx.session().get("email");
        } else {
            return null;
        }
    }else{
        return null;
    }
        }

@Override
public Result onUnauthorized(Http.Context ctx) {
    User current=User.findByEmail(ctx.session().get("email"), "global");
    return ok(forbidden.render(current));
        }
}

@Security.Authenticated(Securedteacher.class)
public static Result createlecture() {...}
```

Figure 4.28: Secured actions

Figure 4.28 shows the secured class implementation and its usage for the user role of teacher. At first use the *HTTP.Context* session to get a current user, then compare the current user role to the required user role. If they are identical, the email address of the current user will be returned; otherwise, *null* will be returned. According to the documentation of the Playframework, if method *getUsername()* returns a string, annotated action, *createlecture()* will be executed for the current user. If *getUsername()* returns value null, the method *onUnauthorized()* will be executed. In this example, a web page "forbidden" will be generated for the current user, and notify the user that this action cannot be accomplished with their current user role.

4.3.3 Self-management

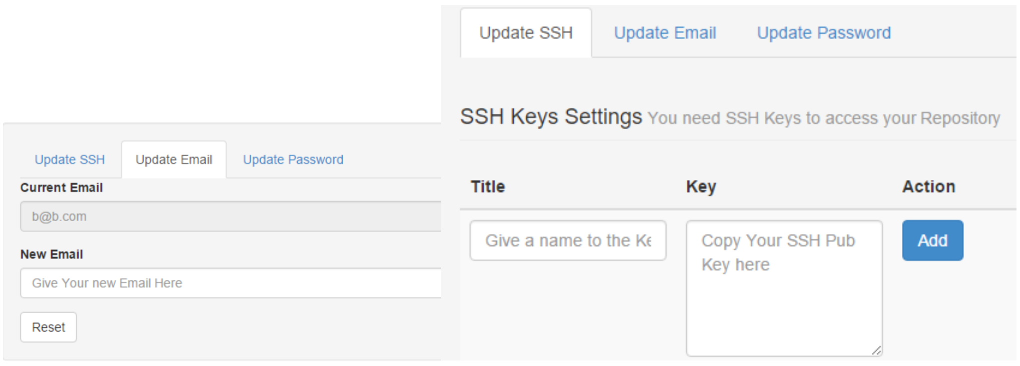


Figure 4.29: Self-management

Figure 4.29 shows the component of the self-management system. At the settings page of the user, the user can update their ssh, password and email; they can update the password and email using the same mechanisms from the Section 4.3.2. If a user wants to change their email or password, the HMS system will always send a confirmation email with a confirmation token. Only after the user has confirmed the changes from the URL within the email, can the changes then be saved into the database. It is a necessary step to offer more security to sensitive data. And in order to use the Git repository which is managed by Gitolite (Section 4.2), the user also needs to add the ssh public key from their working terminals.

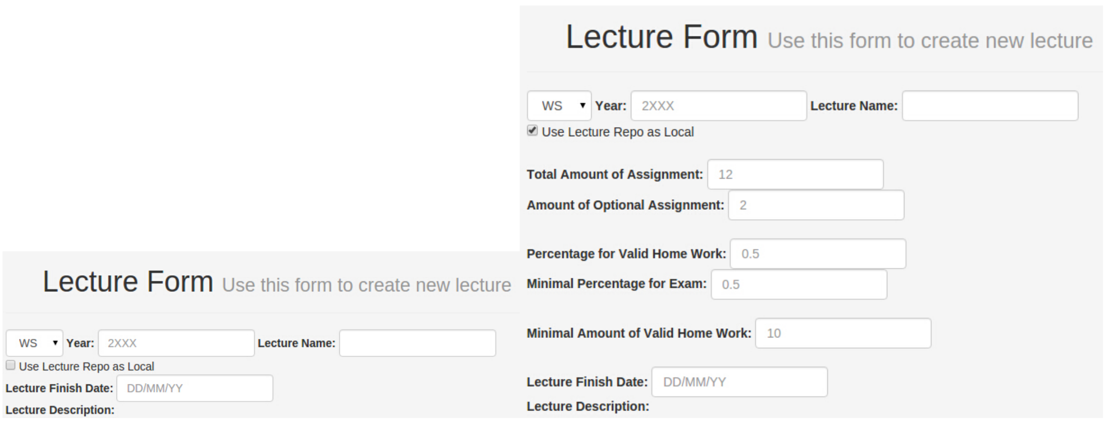
4.4 Course management

The management of a course includes two parts: Creating a new course and the enrollment of the course participants.

4.4.1 Creating a new course

Because of the different types and features of a course (Figure 3.6), the function of creating new course must be able to cover these two types of courses. Also, creating a new course is related to the database generation, so it should also have the ability to decide whether a new semester database should be generated.

User interface



Lecture Form Use this form to create new lecture

WS ▾ Year: 2XXX Lecture Name:

☐ Use Lecture Repo as Local

Lecture Finish Date: DD/MM/YY

Lecture Description:

Lecture Form Use this form to create new lecture

WS ▾ Year: 2XXX Lecture Name:

☑ Use Lecture Repo as Local

Total Amount of Assignment: 12

Amount of Optional Assignment: 2

Percentage for Valid Home Work: 0.5

Minimal Percentage for Exam: 0.5

Minimal Amount of Valid Home Work: 10

Lecture Finish Date: DD/MM/YY

Lecture Description:

Figure 4.30: GUI of creating a new course

Figure 4.30 shows the user interface of creating a new course. The form from left is used to create the "Type II" course with a remote Git repository. This type of course only contains a semester project and didn't have a normal final exam so there isn't much information needed for creating the "Type II" course. Only the deadline and a description of the course are needed. Another form on the right side is for the "Type I" course. In the "Type I" course students need to do various homework assignments and earn enough points to enter the final exam. So when creating this kind of course, the teacher should provide the precondition for the final exam, like the number of assignments and how many points are needed to enter the final exam. All these settings can still be modified after the course has been created.

Server-side logic

After sending the course creation form to the server, the data from the form will be processed by the *createlecture()* method. Figure 4.31 shows the process for how the newly created courses are saved into a semester database.

The first thing to check is the semester name of the course. If the semester name of the course does not show in the semester tracking table of global database, it simply means a new semester has begun. So the first thing the system will do is generate a new database for this course and then save the new semester into the semester tracking table. After that, the course will be saved under the correct semester database.

If the semester name of a new course are present, the course will be simply saved into the related semester database.

```
if(!createlectureForm.hasErrors()) {
        String semester = createlectureForm.get().yearprefix
                + createlectureForm.get().year;
        User globaluser=User.findByEmail(ctx().session().get("email"),"global");

        if (Semester.findsemester(semester) == null) {
            List<Class> entity = new ArrayList<Class>();
            entity.add(Semesteruser.class);
            ...
            entity.add(Conversation.class);
            createServer(semester, entity);
            Semester addsemester = new Semester();
            addsemester.semester = semester;
            addsemester.save("global");
        }
        Lecture lecture = new Lecture();
        lecture.semester = semester;
        lecture.courseName = createlectureForm.get().coursename;
        ...
        lecture.closingdate = createlectureForm.get().closingdate;

        Semesteruser semesteruser=Semesteruser.getSemesteruserfomrUser(semester, globaluser);
        lecture.lasteditor = semesteruser;
        if (!lecture.attendent.contains(lecture.lasteditor)) {
            lecture.attendent.add(lecture.lasteditor);
        }
        try {
        lecture.save(lecture.semester);
        ...
    }
```

Figure 4.31: Logic of creating a new course

4.4.2 Course enrollment

After creating the course, it is time to let the students or other teachers and assistants join the course. The first part of course enrollment is to list all of the available courses to the users of the HMS.

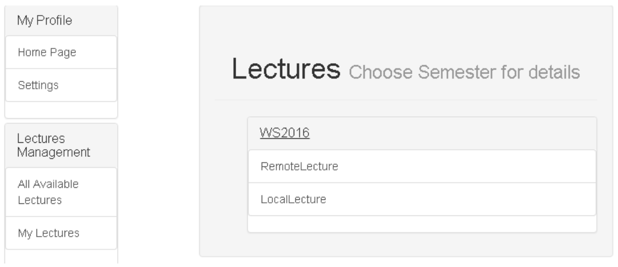
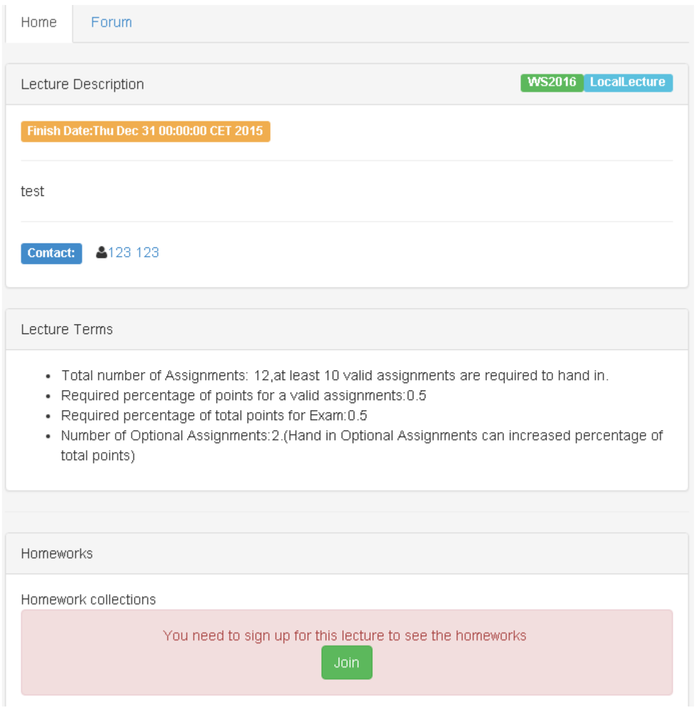


Figure 4.32: Browsing the course

Figure 4.32 shows all the available courses under the semester "WS2016". When the user clicks on one of the courses, the homepage of the course will show up and ask the user to join the course (Figure 4.33).



Lecture Description

Finish Date:Thu Dec 31 00:00:00 CET 2015

test

Contact: 123 123

Lecture Terms

- Total number of Assignments: 12,at least 10 valid assignments are required to hand in.
- Required percentage of points for a valid assignments:0.5
- Required percentage of total points for Exam:0.5
- Number of Optional Assignments:2.(Hand in Optional Assignments can increased percentage of total points)

Homeworks

Homework collections

You need to sign up for this lecture to see the homeworks

Figure 4.33: Course homepage before sign up

Before actually signing up for the course, the user can only view the lecture description and the lecture terms. The other functions of this lecture are blocked.

Join the new course

If a user has a user role above "Defaultuser", this user can join a course. If the user has a student user role, after clicking the "Join" button, three things will happen:

Figure 4.34 shows the three processes for when a student joins the course. First, the current student user will be added to the participants' list of the target lecture. At the same time, a new evaluation object will be generated for this student. The evaluation object will be used to save the student's total performance for this lecture. At last, the student will get a Gitolite bare repository generated by the method *createRemoteRepo()*.

On the other hand, if a user has a teacher or an assistant role, the system only needs to add the user to the course participants' list, since the evaluation and repository are only needed by the student.

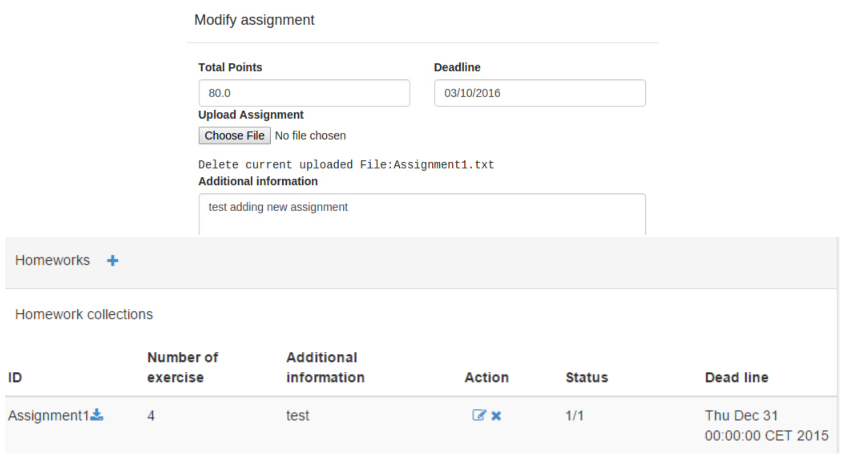
```
public static Result addSemesterusertoLecture(String user, String semester,String
    lecturename){
...
if(Lecture.addSemesterusertoLecture(semester, semesteruser, lecture)){
    if(semesteruser.roles.equals(UserRoll.Students.toString())){
        Evaluation eval= new Evaluation();
        eval.lecture=lecture;
        eval.student=semesteruser;
        eval.save(semester);
        semesteruser.update(semester);
    try{
        String repopath= RepoManager
                    .createRemoteRepo(currentuser, lecture, request().
                        getHeader("Host"));
        if(repopath!=null){
            Repo newrepo = new Repo();
            newrepo.course=lecture;
            ...
            newrepo.save(lecture.semester);
            }
        else{
            ...
        }
    }catch(Exception e){
     ...}
    }
```

Figure 4.34: Add a student to a course

4.5 Assignment management

The management of assignments consists of three parts: The first part is creating an assignment. The second part is collecting and evaluating the student submissions, and the third part is returning the assignments' results back to the student. The evaluation of the students' submissions has already been discussed in the previous section. This section will focus on creating assignments and distributing the results to students.

Creating assignments


Modify assignment

Total Points	Deadline
80.0	03/10/2016

Upload Assignment
Choose File No file chosen

Delete current uploaded File:Assignment1.txt
Additional information
test adding new assignment

Homeworks +

Homework collections

ID	Number of exercise	Additional information	Action	Status	Dead line
Assignment1	4	test		1/1	Thu Dec 31 00:00:00 CET 2015

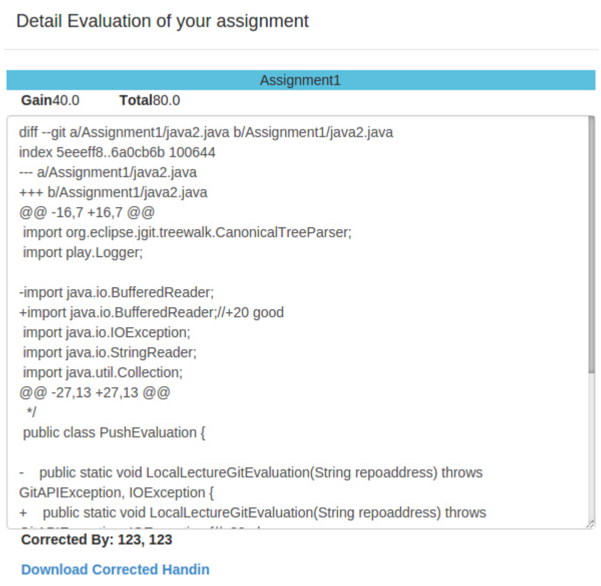
Figure 4.35: Creation of an assignment

Figure 4.35 shows the homework area on the teacher's course homepage. The form from above is used to modify the details of an assignment, and there are several things that the teacher needs to input: The details of the exercise, the deadline of this assignment, and lastly, the teacher can upload the related materials to the assignment. The teacher can also give additional information to clarify the problem inside the assignment. Created assignments will be shown below on the homework section of the homepage. The details of the assignment can still be modified and be deleted after creation, but after a student has uploaded a solution, the assignment cannot be deleted anymore.

On the student side, the created assignment will immediately appear on the course homepage and be ready to be worked on (Figure 4.12).

Feedback of result

After the evaluation of a submission, the results will be given back to the students. The students can direct click the points in the column under the evaluation (Figure 4.12) for details which were shown in Figure 4.36.


Detail Evaluation of your assignment

Assignment1

Gain40.0 **Total**80.0

```
diff --git a/Assignment1/java2.java b/Assignment1/java2.java
index 5eeeff8..6a0cb6b 100644
--- a/Assignment1/java2.java
+++ b/Assignment1/java2.java
@@ -16,7 +16,7 @@
 import org.eclipse.jgit.treewalk.CanonicalTreeParser;
 import play.Logger;

-import java.io.BufferedReader;
+import java.io.BufferedReader;//+20 good
 import java.io.IOException;
 import java.io.StringReader;
 import java.util.Collection;
@@ -27,13 +27,13 @@
  */
 public class PushEvaluation {

-    public static void LocalLectureGitEvaluation(String repoaddress) throws
GitAPIException, IOException {
+    public static void LocalLectureGitEvaluation(String repoaddress) throws
```

Corrected By: 123, 123

Download Corrected Handin

Figure 4.36: Assignment results

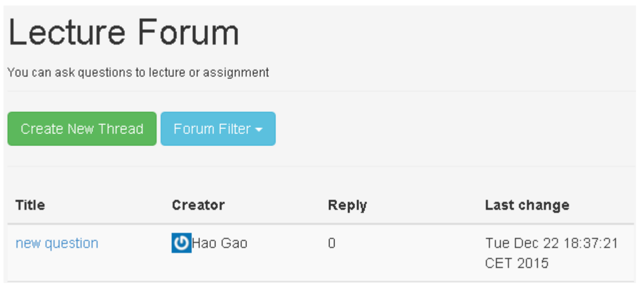
The Unified Diff will be used directly as feedback, so that the students can track down the teacher's comments more easily. Additionally, if the lecture is under local modes, students can also directly download the corrected version using the link in the left-hand corner.

4.6 Communication system

There are two ways for the user of the HMS to communicate with each other: A public course forum and a private instant message system.

4.6.1 Public forum

The forum is used for students to share their questions about the assignments or the course, using the forum to discuss the question can avoid the same questions from being repeatedly asked by different students. Figure 4.37 is the main page of a course forum. The area above is the forum functions, and the forum thread will be listed underneath. The latest modified thread will always be listed at the top.



Lecture Forum

You can ask questions to lecture or assignment

Title	Creator	Reply	Last change
new question	Hao Gao	0	Tue Dec 22 18:37:21 CET 2015

Figure 4.37: Course forum

4.6.2 Private instant message

The HMS system uses the WebSocket protocol (Section 3.1.4) to realize the instant message system. Playframework supports WebSocket natively. But WebSocket can't be directly handled by standard Playframework actions. A possible way to use WebSocket in Play is using function callbacks.

```java
@Security.Authenticated(Securedstudents.class)
@BodyParser.Of(BodyParser.Json.class)
public static WebSocket<String> socket(){
   User currentuser=User.findByEmail(ctx().session().get("email"),"global");
   if(currentuser!=null) {
        return WebSocket.whenReady((in, out) -> {
             Chatsocket.start(currentuser.email, in, out);
        });
   }
}
```

Figure 4.38: WebSocket callbacks

Figure 4.38 is the WebSocket callback function. When the WebSocket is ready, both in and out channels of this connection will be obtained by the system. The channels and the email of the current user will be passed on to the method *start()* (Figure 4.39).

```
public static HashMap<String ,WebSocket.Out<String>> connections;
public static void start(String useremail ,WebSocket.In<String> in , WebSocket.Out<
    String> out){
    connections.put(useremail ,out);
    in.onMessage(new F.Callback<String >() {
    @Override
    public void invoke(String event) throws Throwable {
    JsonNode inmsg = Json.parse(event);
    if(inmsg.findPath("event").asText().equals("allconversations")){
        ...
                out.write(result.toString());
            }
    }

    if(inmsg.findPath("event").asText().equals("chatcontent")){
                ...
                out.write(result.toString());
        }
    }

    if(inmsg.findPath("event").asText().equals("newmessage")){
                ...
                conversation.update(semester);
                out.write(result.toString());
                if(connections.get(other)!=null){
                connections.get(other).write(notification.toString());}
        }
    }
}
});
}
```

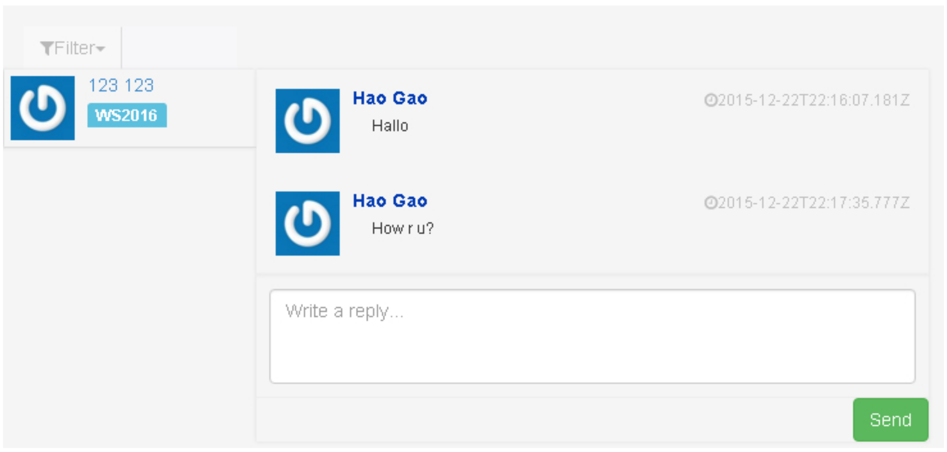
Figure 4.39: Process WebSocket data

The data from the in channel will be processed in the method from Figure 4.39. First the outgoing channel will be saved with its user name inside a hash map, so that the outgoing channel can be easily picked up later. Then the inbound message will be processed by the *invoke()* method. The inbound messages are saved in a JSON[41] string, The first thing to do is to parse the inbound message. The inbound message has two keys: event and data. *Event* is defined by the message type and *Data* stores the actual data from the client. The client will generate three different types of messages based on the user actions. Three example messages are listed below:

1. `{"event":"allconversations","data":{"semester":"WS2016","email":"b@b.com"}}`
2. `{"event":"chatcontent","data":{"convid":"1","semester":"WS2016"}}`
3. `{"event":"newmessage","data":{"semester":"WS2016","convid":"1","content":"How r u?","other":"a@a.com"}}`

The first message is sending a request to the server for all the conversations of user “b@b.com” in semester “WS2016 ”. The second message requests the chat content from the conversation with user ID “1 ” in conversation WS2016. The third message will be generated when a user sends a new message to another user, and it contains the information about the conversation ID, the content of the new message and the email address of the other participants of this conversation.

According to the information from the inbound messages, the server can serve outgoing messages to a different receiver. For the first two types of messages, the system will simply return the result through the outgoing channel of same user. For the third type of message, the system will first save the content of the new message into related conversations and finally into the database, then the updated conversation object will be formatted into a JSON string and sent back to both participants of the conversation–as long as the other participant is also connected to the WebSocket. The outgoing channel of the other participant will be picked up from the hashmap defined in the first line from Figure 4.39.



Hao Gao 2015-12-22T22:16:07.181Z
Hallo

Hao Gao 2015-12-22T22:17:35.777Z
How r u?

Write a reply...

Figure 4.40: GUI of instant message

Figure 4.40 shows the user interface of instant message on the client’s side. When the user chooses the semester number in the filter, the first type message will be sent to the server. If the user clicks the name of another user in the conversation, a second type of message will be sent and the conversation content will be shown. Sending the new reply will generate a third type of message and the chat content will be immediately updated.

5 Chapter 5 Tests

Testing a web application is challenging due to the nature of web applications. First, web applications have a client/server structure, with asynchronous HTTP calls and response to synchronize the state of each side. Second, web application is a mix of different technologies and programming languages; for instance, in the HMS system, JAVA was used on the server side and HTML5[42], CSS[43],JavaScript[44], Scala Template[10] on the client side. Third, a web application have to manipulate the Document Object Model (DOM) on the client side to serve dynamic content[45]. Therefore, only using the unit test cannot test a web application meaningfully. The test of the HMS system is divided into two parts: The first part is using Fluentlenium[46] to test the GUI, and the second part is using JUnit to test the Server-Logic.

5.1 Fluentlenium

Fluentlenium is a framework that helps developers write Selenium[47] tests[48]. Selenium is an in-browser programming system, which allows developers to directly drive the real web browser programmatically. It also has direct access to the DOM elements of the web page on the client side, and can assert expected client behavior defined by the developer. Since Selenium works within the browser, it can be used to test the dynamic behavior of JavaScript and the server response through the interaction between the browser and user[49, 50]. Fluentlenium provides a fluent interface to the Selenium web driver, so that the programming of Selenium tests will be much easier and the code will be more readable. Figure 5.1 demonstrates one of many test scenarios written with Fluentlenium: Registering a user account in the HMS system and getting a confirmation email. First, the web browser opens a page at “http://localhost:9000 ”. Second, a HTML form will be filled out and submitted to the server. At last, after receiving the response from the server, the accepted behavior, in this case, a success label should be displayed correctly with the text "*You will receive a confirmation email soon. Check your email to activate your account.*" If this text is incorrect, the test will fail.

Other scenarios, for instance, “creating a new course by teacher”, ”uploading a new solution to the assignment by student” are similarly programmed as in Figure5.1.

```
@Test
public void a_testRegistration(){
goTo("http://localhost:9000");
fill("#SignUpEmail").with("123@123.com");
fill("#SignUpLastname").with("123");
fill("#SignUpFirstname").with("123");
fill("#SignUpPassword").with("123");
click("#SignUpSubmit");
await().atMost(5, TimeUnit.SECONDS).until(".label-success").areDisplayed();
assertThat(find(".label-success").getText()).isEqualTo("You will receive a
    confirmation email soon. Check your email to activate your account.");
}
```

Figure 5.1: A test Scenario of Fluentlenium

5.2 JUnit

Since the server side of the HMS system is written in Java, the server logic will be tested by the JUnit. In order to save the testing time, it is important to test the server logic without starting the whole HMS system.

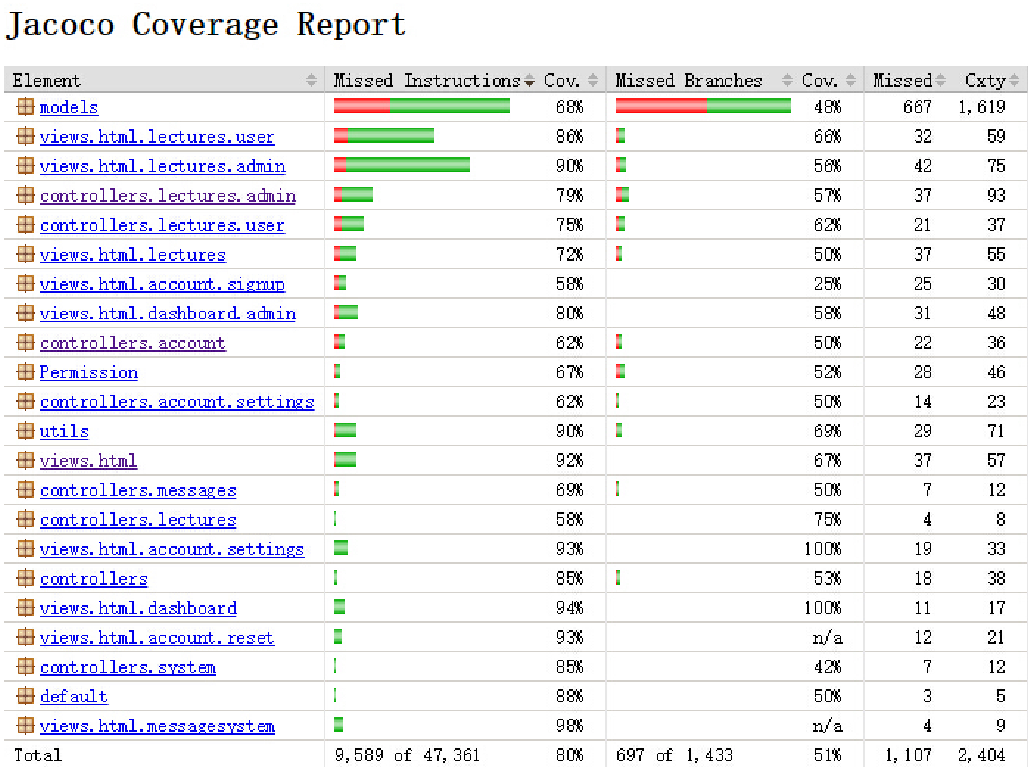
```
@Test
public void testDeleteSSH() {
        User owner= new User();
        owner.save("global");
        SSH ssh= new SSH();
        ssh.save("global");

        FakeRequest request=new FakeRequest("POST","/settings/ssh_delete?sshid=1")
            ;
        Result result = route(request);
        assertThat(status(result)).isEqualTo(OK);
}
```

Figure 5.2: Controller Test

Figure 5.2 is a unit test for controller "deleteSSH", using the class "FakeRequest" from the Playframework. The developer can easily test the behavior of the controller without actually starting the application. In Figure 5.2 a fake HTTP post request will be sent to the controller with a URL query "sshid=1 ". Since before this request a new ssh has already been saved into the database, the request for "deleting a ssh with id=1 " should be successful. In this case the request status should be equal to "200" or "OK".

5.3 Result

Figure 5.3 shows the test coverage from the test procedure from both aspects (GUI and Logic), which were calculated by the Jacoco for SBT[51]. However, due to the incompatability of SBT and Jacoco, many scala methods which were generated by Playframework during run-time were marked by the Jacoco sbt version as not covered, especially in the database tests.



Jacoco Coverage Report

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty
models		68%		48%	667	1,619
views.html.lectures.user		86%		66%	32	59
views.html.lectures.admin		90%		56%	42	75
controllers.lectures.admin		79%		57%	37	93
controllers.lectures.user		75%		62%	21	37
views.html.lectures		72%		50%	37	55
views.html.account.signup		58%		25%	25	30
views.html.dashboard.admin		80%		58%	31	48
controllers.account		62%		50%	22	36
Permission		67%		52%	28	46
controllers.account.settings		62%		50%	14	23
utils		90%		69%	29	71
views.html		92%		67%	37	57
controllers.messages		69%		50%	7	12
controllers.lectures		58%		75%	4	8
views.html.account.settings		93%		100%	19	33
controllers		85%		53%	18	38
views.html.dashboard		94%		100%	11	17
views.html.account.reset		93%		n/a	12	21
controllers.system		85%		42%	7	12
default		88%		50%	3	5
views.html.messagesystem		98%		n/a	4	9
Total	9,589 of 47,361	80%	697 of 1,433	51%	1,107	2,404

Figure 5.3: Test Coverage

Chapter 6

6 Deployment and maintenance

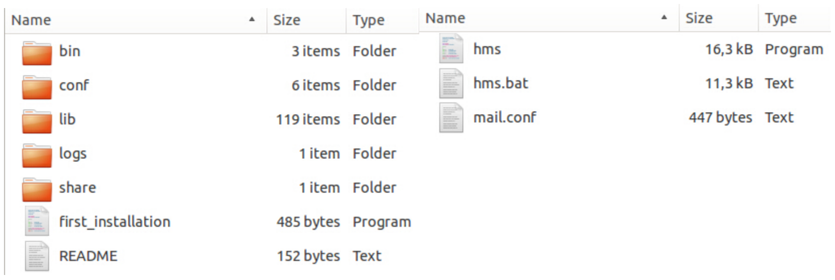
After describing every step of the development phase of the HMS system, it is time to put the system into real practice.

The first part of this chapter will describe the steps for deploying the HMS system to a private server running with Ubuntu 15.04[1], and the second part will demonstrate the maintenance procedures–i.e. backing up semester data and managing the user role–of the HMS system.

6.1 Deployment

Since the HMS system was written in Java and needs Gitolite to manage the Git repository (Section 3.2.1), the user has to make sure the Gitolite server and Java SDK 8.0 have already been installed on the server before running the HMS system. Besides installing the required software, a mail server also needs to be configured, so that the HMS system can send out email. Then, the user can execute the start script within the distribution package of HMS to bring the system online.

The content of the distribution package are listed on the left side of Figure 6.1.



Name	Size	Type
bin	3 items	Folder
conf	6 items	Folder
lib	119 items	Folder
logs	1 item	Folder
share	1 item	Folder
first_installation	485 bytes	Program
README	152 bytes	Text

Name	Size	Type
hms	16,3 kB	Program
hms.bat	11,3 kB	Text
mail.conf	447 bytes	Text

Figure 6.1: Distribution package of HMS

The script “first_installation” will be used to automatically install and configure the Gitolite server. On the right side of Figure 6.1, the files inside “bin” folder are shown.

[1]The installation procedure has also been tested on the Ubuntu 14.04LTS, however the automatic installation of the Gitolite is not working with Ubuntu 15.10, so Gitolite needs to be installed manually

The script “hms” is for starting the whole HMS system, and the file “mail.conf” needs to be configured to fit the mail server.

Installation of Gitolite

First of all, the server maintainer should create a new user with the name “git” on the Ubuntu server. Then copy the distribution package of HMS into a folder and execute the script “first installation” as follows:

```
1. chmod +x first_installation
2. ./first_installation
```

The script will first install Git and Gitolite under the user “git”, then generate the ssh key pair for the Git user and configure the Gitolite server based on this ssh key. At last the administrator repository of Gitolite will be cloned into the home folder and will then be ready for use. If the installation process was successful, the user can find two new directories under the home folders: “repositories” and “gitolite-admin”.

Configuring the mail server

The mail servers are configured by the file “mail.conf”, and the configurations are: `smtp.host=mail.gmail.com` and `smtp.port=587`, both of which have to be correctly set according to the mail server which is used for the HMS system.

Starting the HMS system

After setting up the Gitolite server and the mail server, the HMS system can finally be started by executing the script “hms” with the configuration file of mail server:

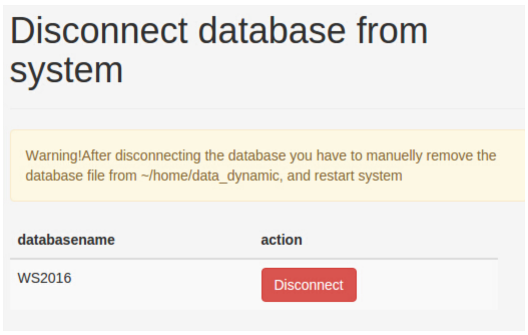
```
./hms -Dconfig.file=mail.conf
```

6.2 Maintenance

The start-up phase of the HMS system will generate an account for the system administrator, and the user name and password of the account will be saved into a folder named “HMS_Config” under the “git” home directory. The administrator account provides two functions: backing up semester data and managing the user roles.

Backup

Since the database file and files, which are generated by the activities of the lecture during the semester, are saved under a unified structure (Figure 3.15), it only needs the maintainer to move these files to other place. And additionally, one should remove the information of this semester from the system, so that after rebooting the HMS, the system will not try to reload the database of this semester (Figure 4.3), and all the users of the HMS cannot access the data of this semester anymore. Figure 6.2 shows the user interface of removing semester information from the system. After the maintainer clicks the "Disconnect" button, the information for semester "WS2016 " will be deleted and the database of "WS2016 " will not be reloaded after rebooting the HMS.



Disconnect database from system

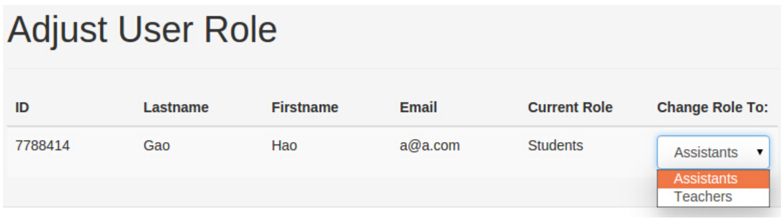
Warning!After disconnecting the database you have to manuelly remove the database file from ~/home/data_dynamic, and restart system

databasename	action
WS2016	Disconnect

Figure 6.2: Disconnecting the semester database

Manage user role

Besides maintaining the data of the HMS, the administrator of the HMS also needs to take care of the user role. Based on the request from the user, the administrator can adjust the user role from the "User Management" panel (Figure 6.3).



Adjust User Role

ID	Lastname	Firstname	Email	Current Role	Change Role To:
7788414	Gao	Hao	a@a.com	Students	Assistants

Figure 6.3: Manage user role

Chapter 7

7 Summary and Future Work

The Homework Management System (HMS) introduced by this thesis has two major differences compared to other Learning Management Systems (LMSs). The first difference is that the HMS uses Git repository to manage the homework submissions. Git-based submission management has several advantages: First, the complicated implementation of submission strategies can be saved because the features of Git repository have already covered most of them (Section 3.2.1). Second, automatic evaluations using the result from "git diff " are possible (Section 3.2.2) so that the teacher can register the points directly into the student‘s account simply using the push function of Git without logging into the HMS system at all.

Another difference is that, in order to reduce the additional work besides the teaching activity such as backup and server maintenance, HMS uses a special mechanism to manage the data. First, HMS is based on the Playframework and embedded database. With this combination the HMS can easily run on any computer with JVM pre-installed as a stand-alone Java application; it doesn't require an additional server and database configuration. Second, the file-based embedded database will be generated separately for each semester, so that each database will only hold the data specific to that semester, and the file database will be saved together with other files under a unified storage structure on the host computer, so that the backup procedure is almost as easy as copying the semester-related files to another location (Section 3.2.3).

The current work of HMS focuses on improving the low level functions–e.g. introducing new methods to manage the homework and data. Some other high level features related to the educational activities are not supported in the current state or are only partially supported–for instance, the course enrollment can only support self-enrollment which allows users to enroll themselves, and the admin of the lecture however cannot control the enrollment. In the future work of the HMS, these high level features should be added continuously based on the requirements of the educational activities.

Nomenclature

API	Application Programming Interface
DOM	Document Object Model
GUI	Graphical User Interface
HMS	Homework Management System
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
LMS	Learning Management System
ORM	Object-relational mapping
VCs	Version Controle System
W3C	World Wide Web Consortium

Bibliography

[1] ROMERO, Cristóbal ; VENTURA, Sebastián ; GARCÍA, Enrique: Data mining in course management systems: Moodle case study and tutorial. In: *Computers & Education* 51 (2008), Nr. 1, S. 368–384

[2] *Statistisches Bundesamt: Zahl der Erstsemester im Studienjahr 2013*. https://www.destatis.de/. Version: 11 2013. – Last visited on 12.01.2016

[3] RÖSSLING, Guido ; JOY, Mike ; MORENO, Andrés ; RADENSKI, Atanas ; MALMI, Lauri ; KERREN, Andreas ; NAPS, Thomas ; ROSS, Rockford J. ; CLANCY, Michael ; KORHONEN, Ari u. a.: Enhancing learning management systems to better support computer science education. In: *ACM SIGCSE Bulletin* 40 (2008), Nr. 4, S. 142–166

[4] *Git*. https://git-scm.com/. – Last visited on 22.10.2015

[5] WEAVER, Debbi ; SPRATT, Christine ; NAIR, Chenicheri S.: Academic and student use of a learning management system: Implications for quality. In: *Australasian Journal of Educational Technology* 24 (2008), Nr. 1, S. 30–41

[6] STEVENS, Marc Martinus J. u. a.: *Attacks on hash functions and applications*. Mathematical Institute, Faculty of Science, Leiden University, 2012

[7] CHACON, S. ; STRAUB, B.: *Pro Git*. Apress, 2014 (The expert's voice). https://books.google.de/books?id=jVYnCgAAQBAJ. – ISBN 9781484200766

[8] *Playframework*. https://www.playframework.com/. – Last visited on 16.09.2015

[9] LEFF, Avraham ; RAYFIELD, James T.: Web-application development using the model/view/controller design pattern. In: *Enterprise Distributed Object Computing Conference, 2001. EDOC'01. Proceedings. Fifth IEEE International* IEEE, 2001, S. 118–127

[10] *Playframework Documentation Version 2.3.x*. https://playframework.com/documentation/2.3.x/. – Last visited on 16.09.2015

[11] *Apache Tomcat*. http://tomcat.apache.org/. – Last visited on 21.02.2016

[12] Lindholm, Tim ; Yellin, Frank ; Bracha, Gilad ; Buckley, Alex: *The Java virtual machine specification*. Pearson Education, 2014

[13] *WebSocket*. http://www.websocket.org/. – Last visited on 23.01.2016

[14] Wang, V. ; Salim, F. ; Moskovits, P.: *The Definitive Guide to HTML5 WebSocket*. Apress, 2013 (Books for professionals by professionals). https://books.google.de/books?id=YIgdrqKZzYMC. – ISBN 9781430247418

[15] Microsystem, Sun: *Distributed Application Architecture*. 06 2009

[16] Hohpe, G. ; Woolf, B.: *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Pearson Education, 2012 (Addison-Wesley Signature Series (Fowler)). https://books.google.de/books?id=qqB7nrrna_sC. – ISBN 9780133065107

[17] Day, Mark ; Rosenberg, Jonathan ; Sugano, Hiroyasu: A model for presence and instant messaging. 2000. – Forschungsbericht

[18] *Glossary: WebSockets*. https://developer.mozilla.org/en-US/docs/Glossary/WebSockets. Version: 2015. – Last visited on 20.12.2015

[19] *Moodle*. https://moodle.org/. – Last visited on 20.09.2015

[20] *Documentation of moodle*. https://docs.moodle.org/30/en/Main_page. – Last visited on 02.12.2015

[21] *Moodle documentation: Course Backup*. https://docs.moodle.org/30/en/Course_backup. – Last visited on 30.09.2015

[22] *GNU.org: Unified-Format*. http://www.gnu.org/software/diffutils/manual/html_node/Unified-Format.html. – Last visited on 12.01.2016

[23] Chaudhri, A.B. ; Rashid, A. ; Zicari, R.: *XML Data Management: Native XML and XML-enabled Database Systems*. Addison-Wesley, 2003 https://books.google.de/books?id=7LNhd0eQulQC. – ISBN 9780201844528

[24] *H2 Database Engine*. http://www.h2database.com/html/main.html. – Last visited on 02.12.2015

[25] *H2 Documentation*. http://www.h2database.com/html/features.html#embedded_databases. – Last visited on 11.12.2015

[26] *Ebean ORM for Java*. http://ebean-orm.github.io/. – Last visited on 23.11.2015

[27] K, S.K.: *Spring And Hibernate*. McGraw-Hill Education (India) Pvt Limited, 2009 https://books.google.de/books?id=NfNbbhBRc0kC. – ISBN 9780070077652

[28] *Java Persistence API*. http://www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html. – Last visited on 26.11.2015

[29] *ObjectDB JPA Reference*. http://www.objectdb.com/api/java/jpa/. – Last visited on 05.10.2015

[30] *Gitolite*. http://gitolite.com/gitolite/index.html. – Last visited on 10.10.2015

[31] *Git on the Server*. https://git-scm.com/book/en/v2/Git-on-the-Server-Getting-Git-on-a-Server. – Last visited on 01.10.2015

[32] *JGit User Guide*. http://wiki.eclipse.org/JGit/User_Guide. – Last visited on 10.11.2015

[33] *Gitolite all-in-one documentation*. http://gitolite.com/gitolite/gitolite.html. – Last visited on 10.10.2015

[34] *Java Gitolite Manager*. https://github.com/devhub-tud/Java-Gitolite-Manager. – Last visited on 16.10.2015

[35] LOELIGER, Jon: Collaborating with GIT. In: *Linux Magazine, June* (2006)

[36] *Java Development Kit*. http://www.oracle.com/technetwork/articles/javase/index-jsp-138363.html. – Last visited on 11.09.2015

[37] *Watching a Directory for Changes*. https://docs.oracle.com/javase/tutorial/essential/io/notification.html. – Last visited on 16.01.2016

[38] *Java Documentation*. https://docs.oracle.com/javase/7/docs. – Last visited on 15.09.2015

[39] YESNAULT: *PlayStartApp*. https://github.com/yesnault/PlayStartApp. – Last visited on 11.09.2015

[40] GILBERT, Henri ; HANDSCHUH, Helena: Security analysis of SHA-256 and sisters. In: *Selected areas in cryptography* Springer, 2003, S. 175–193

[41] *JSON*. http://www.json.org/. – Last visited on 13.10.2015

[42] *HTML5*. https://www.w3.org/TR/html5/. – Last visited on 23.11.2015

[43] *Cascading Style Sheets*. https://www.w3.org/Style/CSS/. – Last visited on 24.11.2015

[44] *JavaScript*. https://www.javascript.com/. – Last visited on 10.12.2015

[45] GAROUSI, Vahid ; MESBAH, Ali ; BETIN-CAN, Aysu ; MIRSHOKRAIE, Shabnam: A systematic mapping study of web application testing. In: *Information and Software Technology* 55 (2013), Nr. 8, S. 1374–1396

[46] *Fluentlenium*. https://github.com/FluentLenium/FluentLenium. – Last visited on 25.01.2016

[47] *SeleniumHQ*. http://www.seleniumhq.org/. – Last visited on 12.01.2016

[48] *Fluentlenium Github Documentation*. https://github.com/FluentLenium/FluentLenium. – Last visited on 25.01.2016

[49] Van Deursen, Arie ; Mesbah, Ali: Research issues in the automated testing of ajax applications. In: *SOFSEM 2010: Theory and Practice of Computer Science*. Springer, 2010, S. 16–28

[50] Brown, C T. ; Gheorghiu, Gheorghe ; Huggins, Jason: *An introduction to testing web applications with twill and selenium*. " O'Reilly Media, Inc.", 2007

[51] *Jacoco for Sbt*. https://github.com/sbt/jacoco4sbt. – Last visited on 12.01.2016

List of Figures

2.1 Workflow of Git[7] 3
2.2 MVC architecture[10] 4
2.3 Earlier systems 5

3.1 The use case from the perspective of different user roles. 7
3.2 The modules of user management 8
3.3 Registration activity 8
3.4 The functions of self-management 9
3.5 Different user roles in the HMS system 10
3.6 Features of different types of courses 11
3.7 Sequence diagram for assignment management 12
3.8 Web socket message pattern 13
3.9 Workflow of file submission using Git 14
3.10 File structure of a repository in local modes 15
3.11 Evaluation of a submission 16
3.12 Unified Diff Format 17
3.13 Multiple databases based on semester 18
3.14 Mixed Mode of h2 server[25] 19
3.15 Semester file structure 19

4.1 Database configuration in Playframework 22
4.2 Create an Ebean server in run time 22
4.3 Reload the database at application start 23
4.4 Global database 24
4.5 Semester database 25
4.6 Class diagram Semesteruser 25
4.7 User To Semesteruser 26
4.8 Clone repository using JGit 27
4.9 Access level for different user 27
4.10 Configuration of access rules with Gitolite 28
4.11 Gitolite configuration from Java 28
4.12 Homework submission for course under local mode 29

4.13 Commit dialog 30
4.14 Commit submission 31
4.15 Delete last submission 31
4.16 Checkout student repository 32
4.17 Grand access to student's repository 32
4.18 Remote mode 33
4.19 File Watch Service 34
4.20 Get the name of committer 36
4.21 Get evaluation 37
4.22 Registration form on homepage 38
4.23 Save user registration 38
4.24 Confirm registration 39
4.25 Registration successful 39
4.26 Login and password recovery 40
4.27 Reset password 40
4.28 Secured actions 41
4.29 Self-management 42
4.30 GUI of creating a new course 43
4.31 Logic of creating a new course 44
4.32 Browsing the course 44
4.33 Course homepage before sign up 45
4.34 Add a student to a course 46
4.35 Creation of an assignment 46
4.36 Assignment results 47
4.37 Course forum 48
4.38 WebSocket callbacks 48
4.39 Process WebSocket data 49
4.40 GUI of instant message 50

5.1 A test Scenario of Fluentlenium 52
5.2 Controller Test 52
5.3 Test Coverage 53

6.1 Distribution package of HMS 54
6.2 Disconnecting the semester database 56
6.3 Manage user role 56