UNIVERSITY OF TWENTE.

Sekhmet: secure chatting during examination

M. Brattinga, D. Ludjen, J. van der Schaaf, J. van der Waal, B. Willemsen

Supervised by: dr. T. van Dijk and prof.dr. M. Huisman

16 April 2021

Abstract

Face-to-face communication during exams can be disturbing -and due to COVID-19, dangerous- for students. To minimize these risks, we built a web app for the University of Twente to facilitate supervisors handling students' questions and making announcements during exams. This was done by creating a real-time chat application where for every student there is a conversation with the student on one side and all supervisors collectively on the other. There is also a global "announcements"-conversation accessible to all students where only supervisors are allowed to write. The authorization is managed through the existing University of Twente single-sign-on-system, cross-referenced with courses and tests imported from existing Canvas administration. The minimum-viable product is functional (minor bugs notwithstanding), however, the use of unfamiliar tools along with a lack of vigilance during development made the final product fall short compared to original designs.

Acknowledgements

We want to thank several people who made it possible to realize this project:

- our supervisors, Tom van Dijk and Marieke Huisman, for their support, constructive feedback and their continuing belief in us even when times were bleak.
- Rafael Dulfer for providing design suggestions on dealing with University of Twente intergrations and together with Floris Breggeman helping to set up the production environment.
- Ard Kosters and André Brands from LISA for their information for the integration with Canvas and the needed information for the Single Sign-On.
- lastly, our testers that were willing to break our system.

Contents

Chapter 1

Introduction

During examinations, communication between supervisors and teachers might be needed. Students might have questions and or supervisors might have announcements about the exam. Currently, teachers or teaching assistants walk to the student who raised his hand, to answer the questions, but this movement and conversing can have a negative impact on the concentration of fellow students. This project brings a solution, which is a chatting system for student to teacher, teacher to student, and teacher to teacher communication, to omit walking and talking entirely.

The product is a web app, such that it works independent of an operating system, to which students and supervisors must log in using their University of Twente credentials. In this application, the student can ask questions during a test. Supervisors can answer these questions, communicate with each other in another chatroom (invisible to the students), and send announcements to the group of students. This last bit is especially advantageous in a room where not all students are taking the same test, since in the current situation you would distract all students in the room, instead of only the ones the announcement is meant for.

The web app has partial Canvas integration so that it can use the database of the university to retrieve some structures without users having to define everything. As the focus is on intuitiveness and ease of use, the Canvas integration and aforementioned university login should make it simple and intuitive to use.

The system has five roles for people: administrator, module coordinator, teacher (supervisor & employee), teaching assistant (supervisor & non-employee), and student. These roles are explained in chapter 2. Everyone in the system has one of these roles for a test (the admin has their role always). When we talk about the permissions of a user, we mean the general permissions of their respective role. It is not possible to add single permissions to users, only to give a role to a user.

Chapter 2

Glossary of Terms

System admin (administrator)
An administrator is a user who has access to the entirety of the system, regardless of who is supposed to "own" any given part. While the system is designed to avoid this being necessary, the administrator is assumed to have access to the technical back-end of the system. An administrator is assumed to not have any of the other roles mentioned here, though the design will not prohibit this as a possibility. The administrator is only used as backup access, for instance, if a coordinator becomes ill or cannot log in anymore.

Student
A student is a student. By default students are only allowed to view tests they are given and ask questions for them. Critically, a student is never a teacher, but they can be a supervisor (TA) in a test they are not also student for.

Teacher
A teacher is an employee at the University of Twente, who may be coordinating a module or supervising a test. Critically, a teacher is never a student.

(Module) Coordinator
A coordinator is a teacher who is the module coordinator for a specific module. Every module has precisely one coordinator. If a teacher imports a module they are part of, that teacher becomes the module coordinator. This task can be transferred.

Supervisor
A supervisor is anyone associated with a test who is not taking that test. While the coordinator is certainly in a supervising role, "supervisors" mainly refer to teachers and TA's that answer questions during tests.

TA

A TA (teaching assistant) is a student who is a supervisor. This means a TA might have a test-taking role in one module while having a supervising one in another. While within a module a TA has the same rights as those supervisors that are teachers, a TA may not make new modules, and may not be a coordinator.

Chapter 3

Stakeholders

There are some affected parties associated with our system. Here we will mention the most prominent ones.

Teachers are part of our direct target users. The program is supposed to make it easier for them to answer questions. However, some may have a negative disposition to other teachers reading their answers, or they don't like the answers being stored.

Students embody the largest subset of our target users. Although it might be less frightening to ask a question in writing, students might have the same problem as teachers about storing data, or that their question can be read by all teachers.

University of Twente is the client. They get the advantages of stored questions, which might influence grading or lecture structures. They also have all the responsibility to remove data and handle the European Privacy Laws (GDPR) on "right to be forgotten" and such.

Other stakeholders might include hardware producers on which our system would run, competitors, people that can't use our system due to dyslexia, or other restrictions. Those are not regarded in this report, since they are either secondarily affected, or they would have a need for special architecture anyway.

Chapter 4

Requirement analysis

4.1 Requirement specification

As this project was commissioned by the University of Twente, most of the requirements are extracted from the project assignment during the first weeks. Further requirements were given by T. van Dijk and M. Huisman during supervisory sessions in the first weeks. We converted all given requirements to be tangible and specific, such that the development team and the clients have the same understanding of what the system will look like. This list of must, could, and won't requirements can be found in section 4.3.

4.2 Requirement prioritization

As the time frame for this project was merely 10 weeks, we had to specify which requirements lay in the scope of this project, and which did not. Therefore, we separated the list of requirements into must and could. This separation was made based on what we determined as a minimum workable system to be used during exams. This includes some basic course and test management, chatting (teachers with each other, student to all teachers, and visa versa), making announcements, editing and deleting functionality for messages, some basic indication to see if another teacher is replying, and a backlog. Any further improvements that would increase the functionality or improve the user experience were considered could requirements.

4.3 Requirements

The list of requirements is shown below, as well as an explanation to what extend those requirements are implemented. All must requirements are implemented. There are no could requirements implemented.

4.3.1 Must

- The system must have a connection with Canvas to allow the import of modules, tests, and people with their role;
  This is implemented based on Canvas API tokens. The full integration with Canvas login is implemented, but not ready for production yet (see section 5.2, Canvas integration).

- Staff and students must be able to log in using University of Twente credentials;
  This is implemented such that everyone with University of Twente credentials can log in via the their Single Sign-On.

- A role-based permission system with different roles for the system admin, the module coordinator, supervising staff, and students must be implemented;
  This is implemented (as described in section 5.2, Authorization) and ensures privacy and security.

- The system must allow conversations to be flagged as being answered;
  This is implemented as chats being automatically assigned to a supervisor when it starts typing. Manually removing the assignment is possible, manually changing to another supervisor is not. The state is not persistent.

- The system must show which chats are unread, and provide the ability to mark chats as unread again;
  This is implemented as unread states for every chat. The state is automatically changed on events, and can manually be changed too. The state is persistent, and shared among all supervisors.

- All teachers must be able to see all communication. Students must not be able to see each other's messages;
  This is implemented as supervisors can access all student chats containing messages, and students are restricted to only see announcements and their own question chat.

- Messages must be persistent
  - After the test, the teacher must be able to see a log of the test until it is formally closed;
  - After the test, only the module coordinator must be able to see the complete log;
  - After re-login, users must see their own chat history if the test is still ongoing;

  This is implemented by storing all messages in the database and retrieving them on loading the chat. A backlog is available for the module coordinator that also includes previous versions of edited messages, and deleted messages.

- Supervisors must be able to make announcements, which remain visible for users who lose connection or log in later;
  This is implemented with the announcement chat and announcement pop-up and notifications. Announcements are stored in the database, and on loading the chat, announcements are the first thing shown.

- Supervisors must be able to communicate with each other via a group chat;
  This is implemented with the supervisor group chat.

- The system must have an intuitive user experience that complies with other apps of the University of Twente and make efficient use of screen space
  The user interface is made as minimalist as possible and follows patterns of other software most users should be familiar with.

- The system must be a web app, able to run on Chromebooks;
  This is implemented. The system is a web app, and is able to run on Chromebooks.

- The system must have a fallback system admin who can access everything;
  This is implemented by the system admin role that can be assigned to a user, which overrules all required permissions.

- It must be possible for supervisors to delete and edit messages, in the sense of making them invisible in the chat while keeping them in the logs;
  This is implemented by giving supervisors the option to edit their own messages, as well as deleting any messages (including of others, and announcements). The full history of edited messages as well as deleted messages are kept in the database and are included in the exported backlog.

- The system must offer a way for tests to be grouped per module;
  This is implemented by linking all tests to a module and showing the tests per module on the overview page.

- The coordinator must be able to delete the complete backlog of a test.
  This is implemented by the download backlog functionality, which includes all data the system has for a particular test.

4.3.2 Could

- Supervisors could be able to communicate with each other via private chats;

- Supervisors could be able to send files as attachment to announcements or chats to be able to distribute files;

- The system could have an indication of waiting time, shown to the teacher interface;

- The system could offer a way to send files (including pictures);
- The system could offer a way to capture a screenshot of Remindo[1], crop it as desired, and send it as a file;
- The system could offer a way to draw on the captured screenshots before sending;
- The system could send chat messages while typing, character by character to improve response time. This is only desired for students to teachers and teachers to teachers, not for teachers to students; *Not implemented. However, the state whether a user is typing, is actually shared. This same system could be extended to send the content of the temporary message.*
- The student could determine the subject of the question before actually asking a question, making it easier for the supervisors to decide who is going to answer that question;

4.3.3 Won't

- The system won't support audio communication;
- The system won't support group tests/exams;
  We believe that the extra infrastructure required for this feature would not be worth its limited use-case
- The system won't support interactive screen sharing

[1] `https://utwente.remindotoets.nl/`

Chapter 5

Design

5.1 Technologies

Front-end

For the front-end, the Vue[1] framework, written in Javascript, is used. The choice of a front-end framework was made to speed up development and allow us to create a better user experience. With such a reactive framework, creating dynamic pages is made easy as it provides some structure and default behavior for certain functionality. As a Vue project is made out of many Vue components, features can be separated into different files to create a clear project structure. The Vue Router[2] is used for routing the front-end page. For state management Vuex[3] is used, as this allows for a shared state between the separate Vue components. Vue Bootstrap[4] is used to speed up layout and style implementation. This bootstrap library contains a lot of default styles, as well as components you can easily implement. For instance forms, pop-ups, menus, layouts, and lists can be used, such that we only had to focus on the behavior, and not on the implementation of default functionality. For the WebSockets, the standard, browser-native WebSockets will be uses, as documented in the MDN Web Docs[5].

Back-end

For the back-end, the Spring framework[6] in Java is used. The Spring framework essentially provides the basic implementations and the core structures that are needed for the creation of a web application, wrapping tomcat[7], using

[1] https://www.vuejs.org
[2] https://router.vuejs.org/
[3] https://vuex.vuejs.org/
[4] https://bootstrap-vue.org/
[5] https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API
[6] https://spring.io/
[7] https://tomcat.apache.org/

Thymeleaf[8] for page serving and inserting the CSRF token on page-load. These implementations can then be applied by simply using the corresponding annotations in the Java classes. The intention is then that this would have allowed us to focus more on the implementation of features, over the setup of infrastructure. The decision on this framework was also based on suggestions from the LISA department. If the software of this project is to be used in the future, which was our aim, it would have to be maintained at the University of Twente. The University of Twente mainly uses Java with Spring as the back-end for their systemsnx and would like new projects to use that too. In this way, they can easily maintain their projects. In addition to Spring framework, the build system Maven[9] is used for dependencies management. For the WebSockets, the standard raw tomcat implementation is used, inherited to be a native part of Spring boot. In order to help manage the communication with the front-end, all of which is done in JSON, we use Gson[10] to manage the dynamic construction of these messages.

Database

The database, in which all data of the application is stored, runs on a MySQL server. Accessed using the Hibernate[11] and JPA[12] frameworks for Java. These libraries access the database automatically and store information in local memory as an entity. It is UTF8-encoded to allow for Unicode[13] characters. The structure will be explained in chapter 5.2.

5.2 Server structure

Database design

The database will be a MySQL database with 6 tables, as seen in figure 5.1.

[8] https://www.thymeleaf.org/
[9] https://maven.apache.org/what-is-maven.html
[10] https://github.com/google/gson
[11] https://hibernate.org/
[12] https://spring.io/projects/spring-data-jpa
[13] https://unicode.org

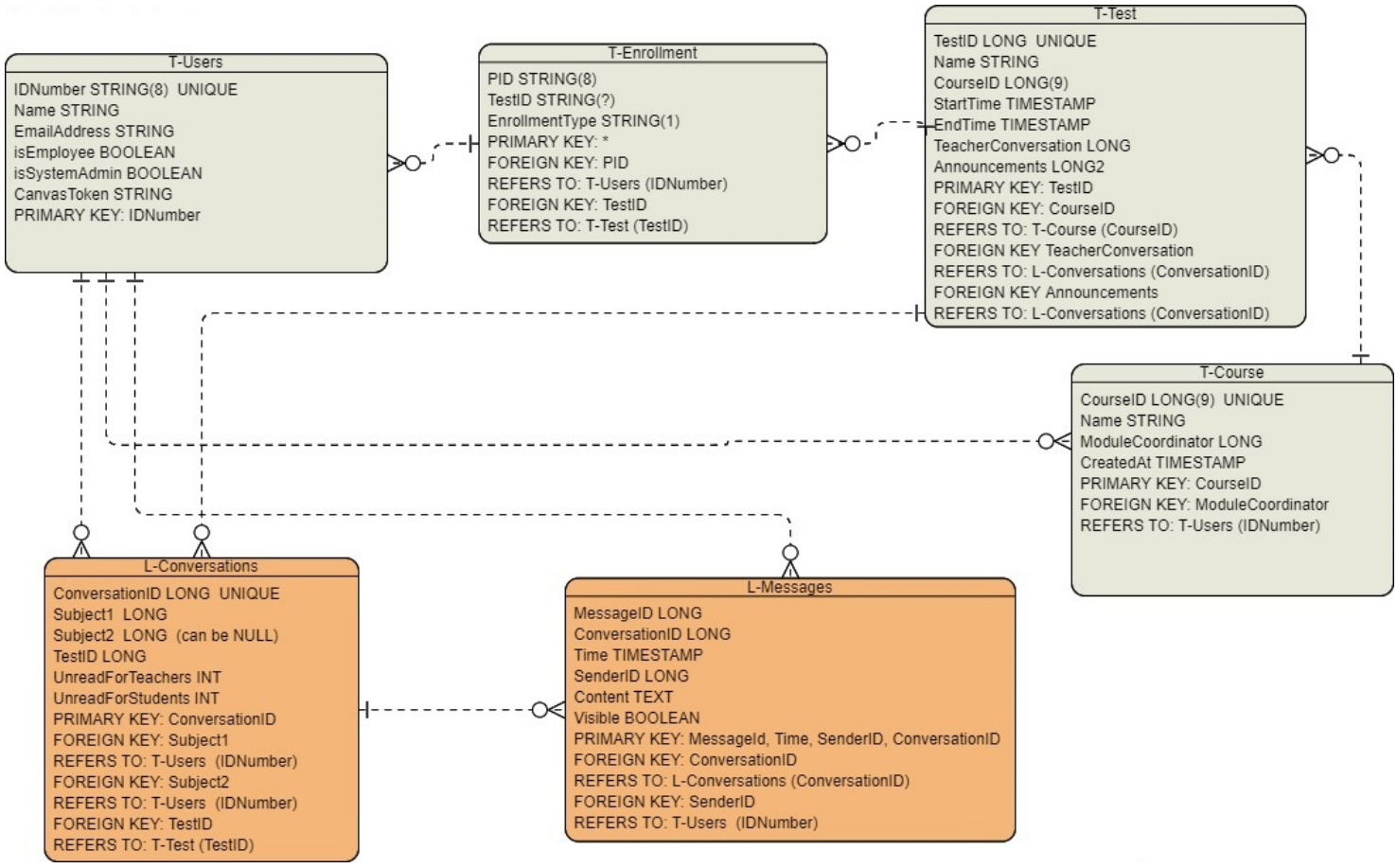

Figure 5.1: UML diagram of database structure

The users table will contain all necessary information on the user. The user ID is the same as the one in the University of Twente's databases, and defines the user uniquely. The name is for ease of use in the user interface: knowing who is chatting with who or for deleting enrollments for example. The email addresses are stored, currently this is used only to allow searching users by email address, and only for the teachers. Lastly it stores whether or not the person is an employee, which allows them to import information from Canvas, and whether a person is a system admin, who is allowed full access, see chapter 2. The Canvas token is also stored here, so that an employee does not have to retrieve it each time they want to access Canvas.

The enrollment table is mostly a pivot table of users and tests, with as inclusion the role for that person for that test. This means that a user can have exactly one role for a test.

The course table contains the basic information of a course, but for our UI we only needed the Canvas ID, which we will also use as a unique ID, and the name. The teacher that imports the course is automatically attached to the course as module coordinator to let the system know who has all responsibilities, see chapter 2. The timestamp of creation is stored for sorting the courses in the UI.

All tests in the tests table are attached to a course. The tests have a start time and end time, which are defined upon starting and ending a test, not beforehand. An active test is defined as a test with a start time and without an end time. The test also stores which conversations are the announcement and teacher conversation for that test.

The conversations table gives a unique conversation ID to every student in a test upon starting that test. This ID is auto-generated. All announcements are also stored in a specific conversation for that test, to make lookup easier.

The messages table stores all sent messages. If a message is altered this message will be saved with the same ID and a different timestamp, so an altered message is a message with the same ID, sender and chat, but a different timestamp. Deleted messages just get their visible field set to false. The reason the primary key of this table is compound is that the front end never needs the server to define the next message ID: it can just keep score of the number of messages they sent (does not even need the amount of received messages because they have a different sender), and then append that.

The database should be able to handle the standard Unicode characters, which include cedillas, accents, and other symbols. This is done by encoding the database in UTF8, which can be completely handled and implemented by the database server.

Classes design

The back-end functions as the server, where business logic and data processing are located. It handles the incoming requests from the front-end through HTTP and REST API. Each request accepts input and provides outputs in the data format JSON. In addition, the back-end also handles authorization processes that decide whether a specific user is allowed access to the APIs. The general structure of the back-end with file names is displayed in figure 5.2.

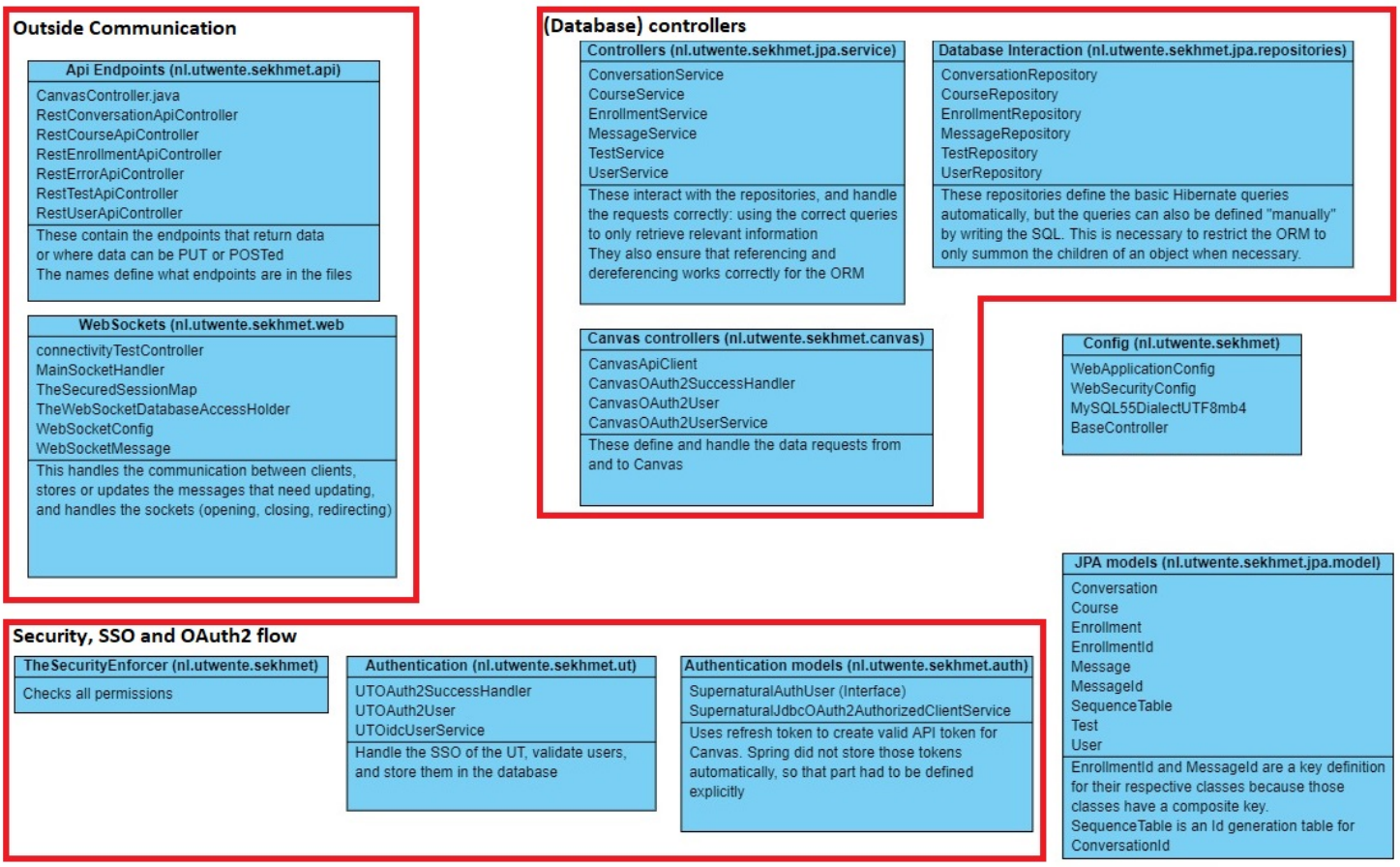

Figure 5.2: Server structure and file names

The back-end is divided into 'layers', interconnected, and each with it's own focus in order to provide a clear overview of the back-end structure especially for system maintenance and/or expansion of the back-end. These layers are data access, service, and controller. figure **??** shows the REST API flow including the connection between the front-end and the controller as well as the connection to the database.

Data Access

The data access layer handles the data retrieval process from the database. The back-end utilizes the Spring Hibernate ORM to create abstract objects from the database. These objects are classified as an entity where each entity is mapped to a table in the database and are defined as Java classes in the back-end. These entity classes provide getters and setters which further eases the process of manipulating the data (or part of the data) in the database.

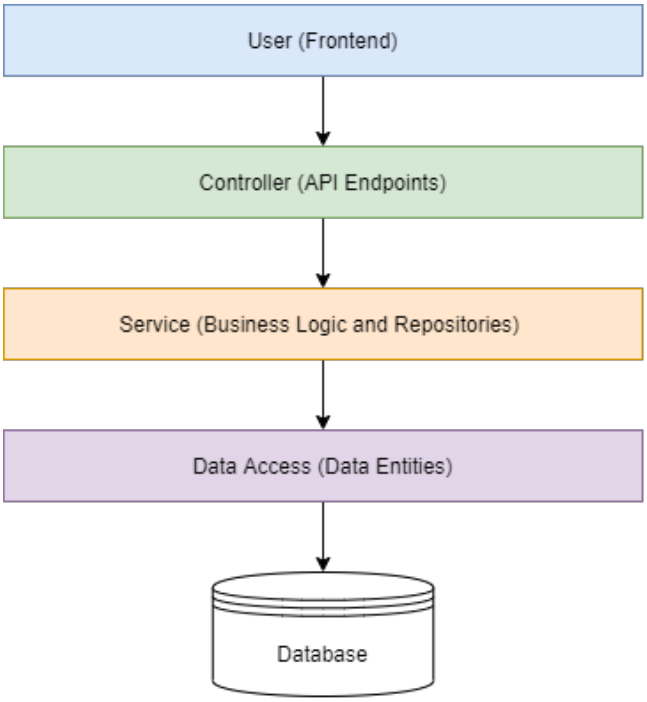

Figure 5.3: Back-end REST API Flow

Service
The service layer is the location of most business logic that happens in the back-end. It connects the data access layer with the controller layer. Repositories, an interface of the JPA (Java Persistence API), are used in order to provide easier access to the database by allowing data retrieval simply by declaring methods inside these repositories which will then return the mentioned entity of the database. The methods inside the repositories are automatically converted to database queries by the Spring Hibernate. Data from the data access layer that require further processing are located in the Service classes where they are grouped according to the repository they mainly access. The service class that mainly accesses the conversation repository is named ConversationService, other classes following a similar pattern. Simple requests such as getting a single entity from the database are called directly from the repositories.

Controller
The controller is the main connection to the front-end where it allows access to the back-end through a HTTP REST API. Types of HTTP request such as GET, POST, PUT, and DELETE are each mapped to a specific URL and a Java method using the @RequestMapping which then will be utilized by the front-end to access the required HTTP requests. All API endpoints apply the authorization checks, determining if a request by a user is allowed. More on authorization will be defined on the subsection Authorization under the section Security (section 5.5).

In the back-end, the REST API endpoints are grouped into Java classes based on the type of entity representation of the database that the endpoint will manipulate. They are grouped into Conversation, Course, Enrollment, Test and User. In addition, there exists a group named Error, however, the endpoints inside of this class do not manipulate any entity representation of the database. Instead, they write into a log file that functions as a storage for error logs. The Java classes of the groups are named as RestConversationApiController for Conversation group, with other groups following similar patterns. WebSocket handlers are also located in this layer and more on this is described in detail in the section Real-time communication section (section 5.3).

Canvas integration

To make setting up a new course with tests as uncomplicated as possible, module coordinators can import courses and tests from Canvas. This is done using the API of Canvas[14]. The API provides access to information from Canvas from the users' point of view. This information is used to set up the course, test(s) and enrollments in the system. The Canvas system also supports OAuth2 to authenticate the user, and give the system access to the users data in Canvas. The implementation of the OAuth2 flow is done using the Spring Security OAuth2

[14] `https://canvas.utwente.nl/doc/api/`

library. Because this library is used and tested by many other developers and users, we can assume most bugs are already found and resolved, thus we can ensure the best possible security for our users.

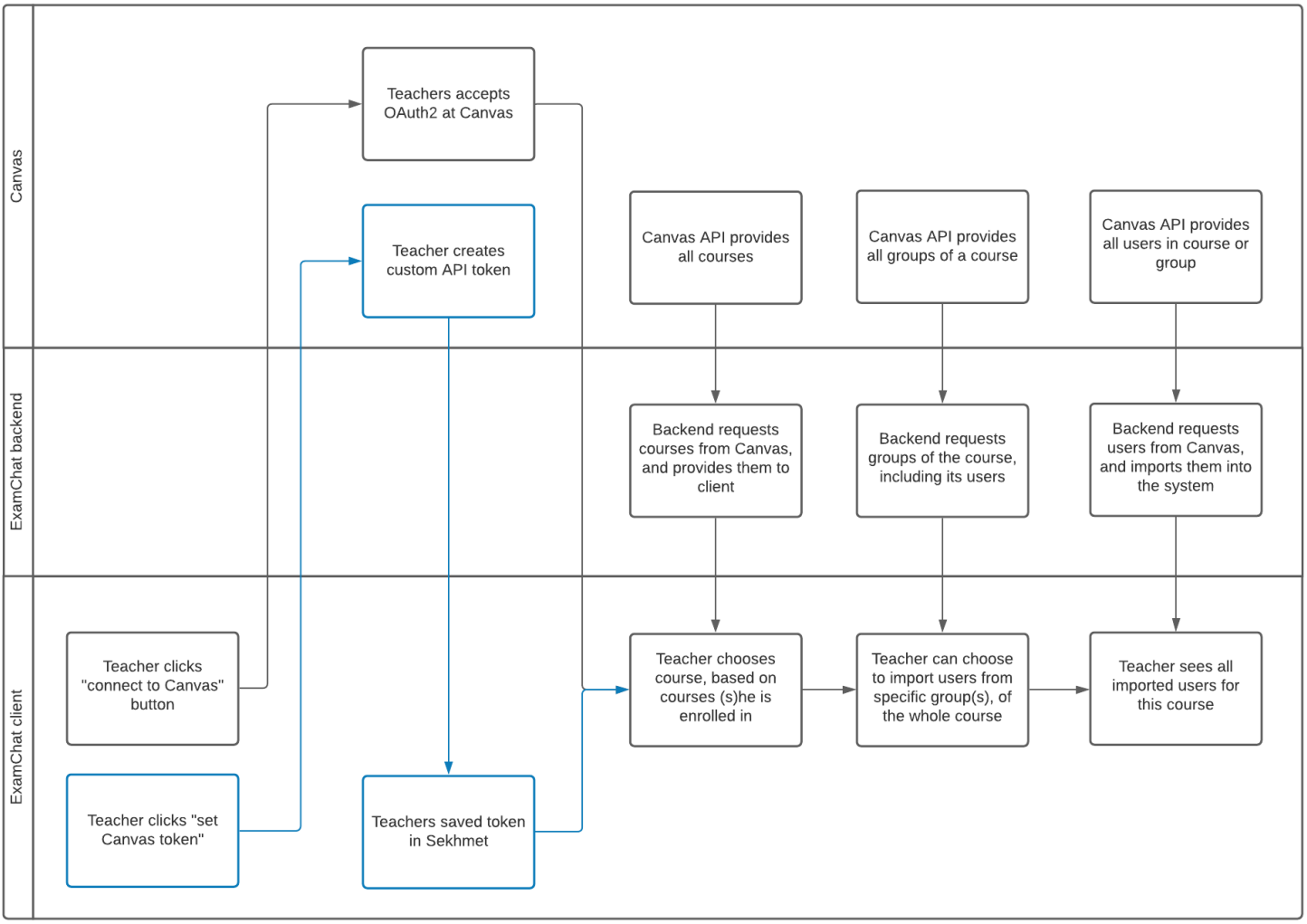

Figure 5.4: Flow of the integration with Canvas. The blue color indicates the alternative flow to authorize a Canvas user

Above in figure 5.4, the flow of the full Canvas integration is shown. At the start, there are two possibilities to start. Both methods are implemented in the system.

The original and intended implementation is using OAuth2, indicated in black

in figure 5.4. This process starts when a teacher clicks the "connect to Canvas" button, so an OAuth2 flow is started between our applications server and Canvas. The teacher is redirected to Canvas, where they have to authorize the Sekhmet application to have access to their account, as can be seen in figure 5.5. Once approved, the teacher gets redirected back to our system. On return, the system requests an access token from Canvas together with a refresh token and stores them. The system can now make requests on behalf of the teacher.

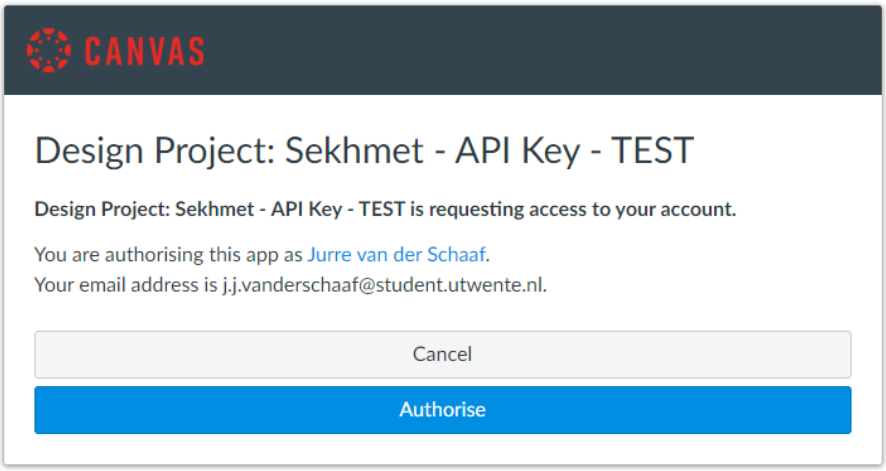

Figure 5.5: Authorizing the Sekhmet application on Canvas

The second way to authorize a Canvas user is indicated in blue in figure 5.4. This process is an alternative to the above because it turned out it was not possible to meet the requirements to use the production Canvas environment of the University of Twente. This alternative process is meant to be replaced by the process above when requirements are met. This process starts with a teacher clicking the 'save Canvas token button'. Instructions are given to help teachers create a custom API token in their Canvas account. Once made, teachers have to paste this API token into a field on our application. After saving, the teacher can proceed with the regular flow of choosing a course to import, where there's no difference in procedure.

Both methods above are implemented and in principle ready to use. The first one, however, has been turned off and temporarily replaced by the second one to make it possible to use the production environment of Canvas.

Now the system can request data from Canvas' API, the integration can actually be used. As a first step in this process, the teachers get a list of all available courses to import. For this, the server first retrieves all courses where the user has the Teacher role. From this list, already imported courses are fil-

tered out. This can be easily done because, on import, the ID in our system is set to the Canvas ID. Once the teacher has chosen the course, the next step is to retrieve all groups and assignments from Canvas of the specific course. The teacher can choose to use the already made assignments in Canvas for a test or can create a custom test. For either of them, the teachers choose to import all students from the course or to import students who are in a group in one or multiple chosen groupsets.

The actual import of users happens in the next and last step in the flow. Once the teacher submits the request of importing the course, all students from the chosen groupsets and/or all students of the course are retrieved, together with all teachers and TAs. The course is created with the same ID and title as in Canvas. Also, the chosen assignments and custom assignments are created. For each user to enroll, they will first be created if they are not in the system already. This check is done based on ID (student ID or employee ID). When inserting, the person name as registered in Canvas is used. Finally, the enrollment of the user to the test is created in the database, with the corresponding role.

5.3 Real-time communication

Client-server connection
For an effective chat application, multiple clients have to be able to send and receive messages in real-time. In order to avoid a repeated-polling system, we use WebSockets between our clients and our server. Most messages have a list of addressees, to which the server will forward the message, provided these clients are allowed to communicate. Any message containing a state significant enough to be stored (such as an actual send message between people as opposed to "user is typing") is then also parsed and saved.

Registering, WebSockets & handlers
In order to set up basic WebSockets in Java Spring, a few components were required. First is the handler, where the behaviour upon receiving a message is defined. Notable about the handler, thanks to the nature of Spring's documentation, is that the lifetime of the handler is unclear. As a result, a few extra helper-singletons have been created where a more efficient implementation would have been possible if we knew for sure these handlers were scoped to a single servlet or connection. This handler then needs to be registered to an endpoint, at which point an interceptor is inserted in order to allow the HTTP-context to be available so that the session authorisation can be carried over to the WebSocket.

TheSecuredSessionHashmap
TheSecuredSessionHashmap is a helper singleton that enables the back-end to act as a bridge for users to send each other messages. It does so by providing the functionality of two key structures in regular networks: DNS and a Firewall.

It is designed to function akin to any standard map object, you attempt to retrieve a WebSocket connection by the user ID of the connected person, and you get the result if that user is also currently connected. Except that if you are not allowed to contact that person they also appear disconnected. To do this, however, all connections need to be registered, unregistered when closed, and because the database query "are these two people allowed to talk" is fairly costly, all verified userID-pairs are saved and have to be wiped at the end of every test.

WebSocket messages & parsing
As with most protocols that contain an element of routing, all messages send to WebSockets are enclosed in a JSON wrapper containing the data the server needs, defined as shown in 5.1.

Attribute Name	Type
messageType	String
receiverIds	Array
message	Object

Table 5.1: WebsocketMessage definition

The possible message types are message_final, message_delete, conversation_unread, conversation_typing, conversation_assigned & nack. Their detailed design and usage will not be discussed here, since those are fairly straightforward, merely containers for the data required to process the new information. We use Gson to cast the incoming text messages to their corresponding models.

5.4 UI design

The main screen of Sekhmet is the chat itself, where communication between supervisors and students happens. To manage the courses and tests, some other screens are available as well. The content of pages is dependent on your role. So, for every role, the page is based on the same template, but certain functions are hidden if you don't have access to them.

Every page contains a navigation bar with the name of the currently logged-in user, as well as a button to log out and a button to go to the courses overview page. On some pages where extra menu buttons should be visible, a sidebar is present (i.e. in the supervisor's chat screen, figure 5.7).

Chat

The chat page is mainly used during exams. There are two main versions of this page, one for students (figure 5.6) and one for supervisors (figure 5.7). The supervisor view shows additional features. First, the design choices of the

student chat page will be explained, after which we will dive into the supervisor features.

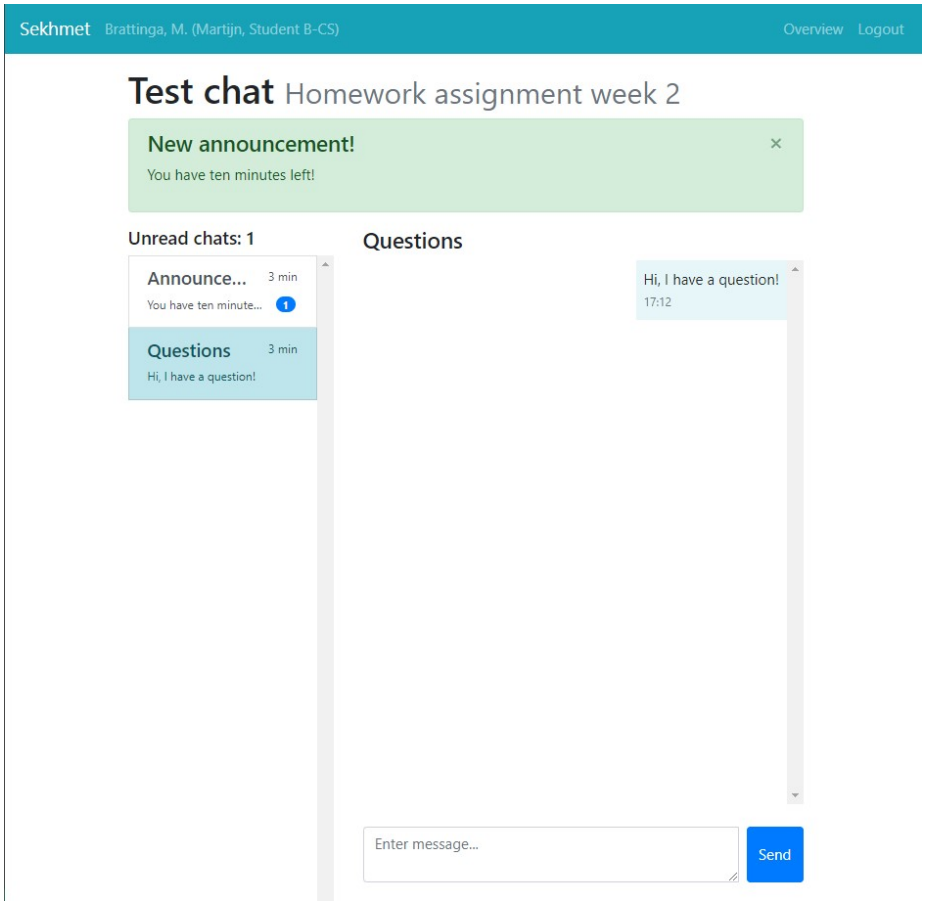

Figure 5.6: Chat screen for students

The chat page starts with the page title, with the name of the test next to it. They are stated at the top, such that there can be no doubt for students or supervisors whether they are in the correct test.

If there is a new announcement, this is displayed in a green banner just underneath the title. The color and location make sure that this announcement hard to miss. This banner only displays the latest announcement, since that is the one deserving the most attention. To read previous announcements, the user can go to the announcement chat.

Switching to another chat, e.g. the announcement chat, can be done in the chat list at the left. This list displays the available chats for the user, ordered

by the timestamp of the latest message. For a student, this defaults to a question chat and an announcement chat. Per chat is indicated if it has unread messages by a blue circle with the number of unread messages, such that users can see which chats require attention. At the top of the chat list, there is a counter of unread chats. This allows for a quick check to see if you have any unread messages in any chat.

At the right side of the chat list, the message list covers the biggest part of the screen. It starts with the name of the chat, then displaying a scrollable list of all messages, and finally a text box with a button to send messages. Every message in the message list contains the content of the message, as well as the timestamp of the message. To distinguish between your own messages and messages of the other party, they are respectively right and left aligned.

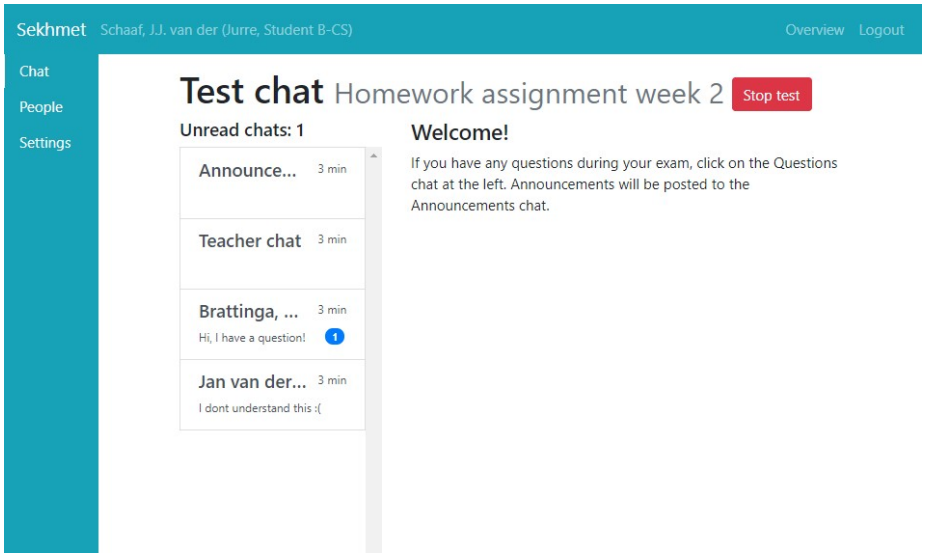

Figure 5.7: Chat screen for supervisors

Supervisors chat
The supervisor chat screen contains some additional features compared to the student view. The sidebar is shown, linking to the test settings and people page. These pages are explained in section 5.4. Starting with the title of the page, next to the test name is a button to start or stop the test (depending on the current state of the test). This prominent position is chosen to remind teachers to open a test if it isn't already, and close the test after the test has ended. As such a prominent position might also trigger unintended button clicks, to close the test a confirmation box (figure 5.8) asks to confirm the decision.

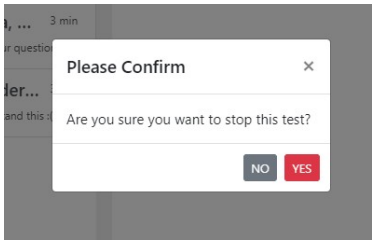

Figure 5.8: Confirmation pop-up for stopping a test

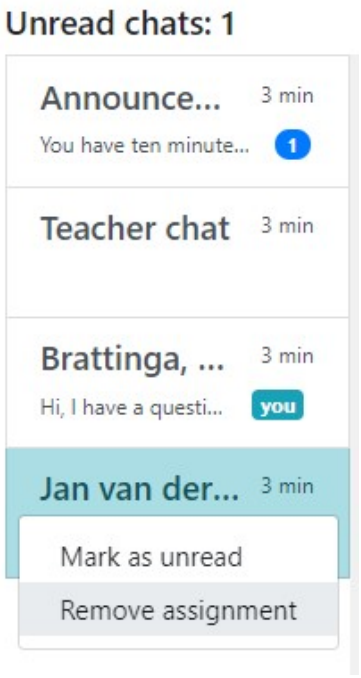

Figure 5.9: Chat list for a supervisor with features

The chat list for supervisors (figure 5.9) shows more chats than in the student view. Besides the announcement chat, it shows a teacher group chat as well as a chat for every student that has sent a message. Every student who hasn't sent a message in this chat before is hidden. By this, a long list of unneeded chats is prevented. Note that a teacher can never initiate a conversation with the student as empty chats are not shown.

For every chat, the supervisor has the option to mark the chat as read or unread. The student chats share this state for all supervisors, so if one supervisor marks a chat as read, this chat is marked as read for all the other supervisors as well. In this way, all supervisors are aware of which students did not get attention yet, and which did. This marking as read can be done manually via the chat options (by clicking three dots that are displayed when hovering a chat in the chat list, and selecting the mark option), as well as automatically when an unread chat is opened by a supervisor. The unread state of the announcement chat and the teacher chat is not stored in the database, and thus not persistent, as this would require storing the unread count for every person separately, and not provide that much more value.

If a supervisor starts typing in a student chat, and the chat is not assigned

to another supervisor yet, that chat will be assigned to the supervisor who just started typing. This is indicated by a green pillow in the chat, with the name of the supervisor who is assigned to that chat. In figure 5.9 the chat with student 'Brattinga' has been assigned to the user, as it states 'you'. This state is not persistent, thus gone if reloaded the page, but the state is shared between all supervisors. It is possible to manually remove the assignment. This assignments enables supervisors to see if another supervisors is replying to a chat.

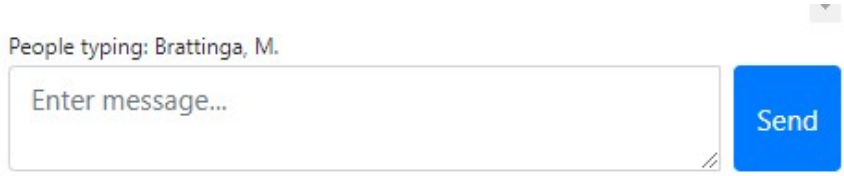

Figure 5.10: Indication that someone is typing in the chat

Another feature that helps supervisors determine if a chatting is currently being answered by another supervisor, is the indication that someone is typing in a chat. If another supervisor is typing in a chat, it is shown just above the input box, as can be seen in figure 5.10. A supervisor is typing when the message box of a chat is not empty, thus if the supervisor is thinking about his reply, but already started typing the reply, other supervisors still know that the other supervisor is replying. This typing state is shared between all supervisors but is not persistent.

Students are not able to see which supervisor is assigned to their chat, or which supervisor is typing in the chat. They also cannot see the sender of the messages.

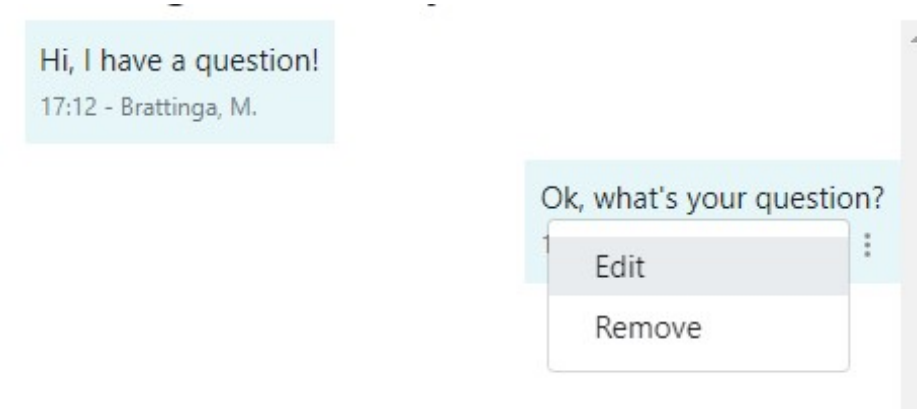

Figure 5.11: Options per chat message available to a supervisor

For supervisors, messages in the message view display in addition to the content and the timestamp, also the sender of the message. This indicates which supervisor has sent the message, which might be helpful information for supervisors. Supervisors also have the option to delete any message and edit their own messages. This is done via the three dots that appear when the user hovers over a message.

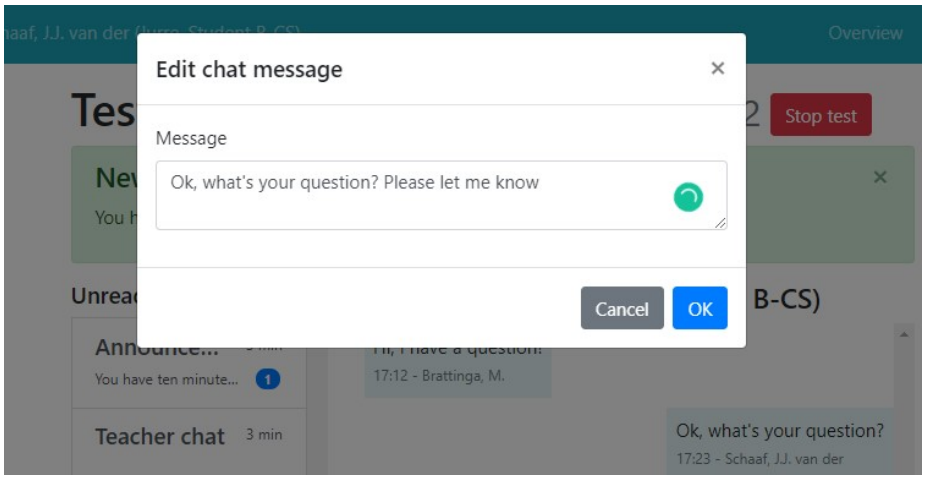

Figure 5.12: Overlay for editing a message

If the supervisor wants to edit its message, a pop-up is displayed with a text area to edit the message. If the blue 'ok' button is pressed, the new message is stored in the database and all people in the chat where the message is edited are notified.

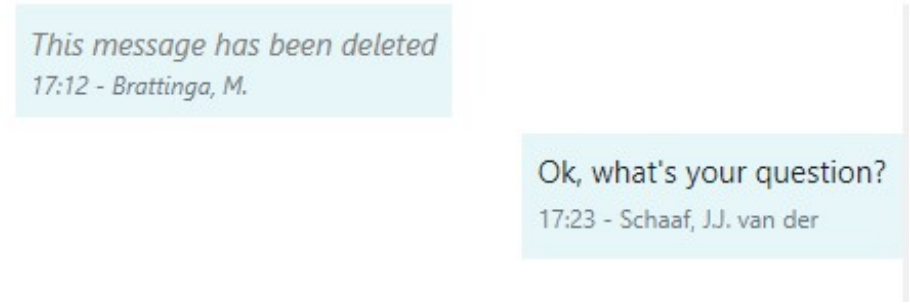

Figure 5.13: Appearance of a deleted message

If a message is deleted, the content of the message for everyone in the chat is replaced by the text 'This message has been deleted' as can be seen in figure 5.13. The original message is saved in the database, but a visibility field is set

to false. The original content is never served to the front-end anymore, but it is present in the back-log of the test. Thus after a test, the module coordinator always can see which messages were sent but deleted later.

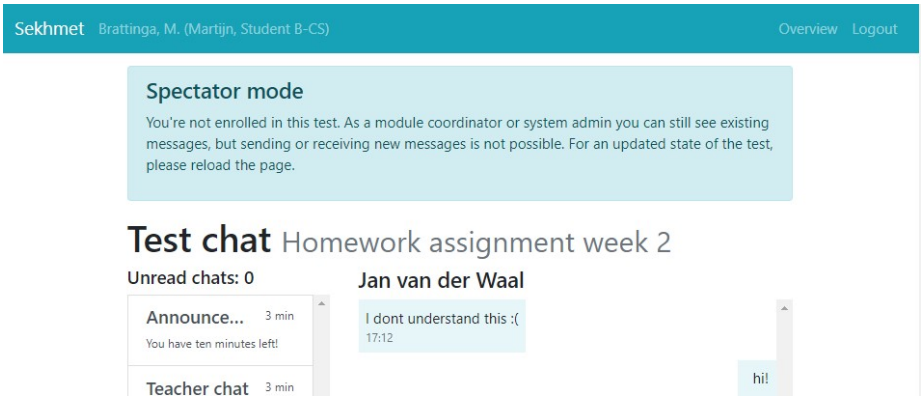

Figure 5.14: Notice of spectator mode

A module coordinator or the system admin might not be enrolled in a test, is still able to see the chat. Since the user is not enrolled, it is not part of any conversation, and thus cannot send or receive chat messages. Therefore, this mode is called spectator mode. To make the user aware that it is not able to send or receive messages, on the top of the chat an informative pop-up is shown (figure 5.14). On loading the page, all chat messages are shown. This is an extra option to view a chat backlog in a slightly more pleasing view than the CSV file.

Test management

There are two pages to alter test properties. One for general test settings, and one for enrollments. The latter one is called the people page. This page (figure 5.15) displays a list of enrollments by role, as well as the ability to add and import new enrollments.

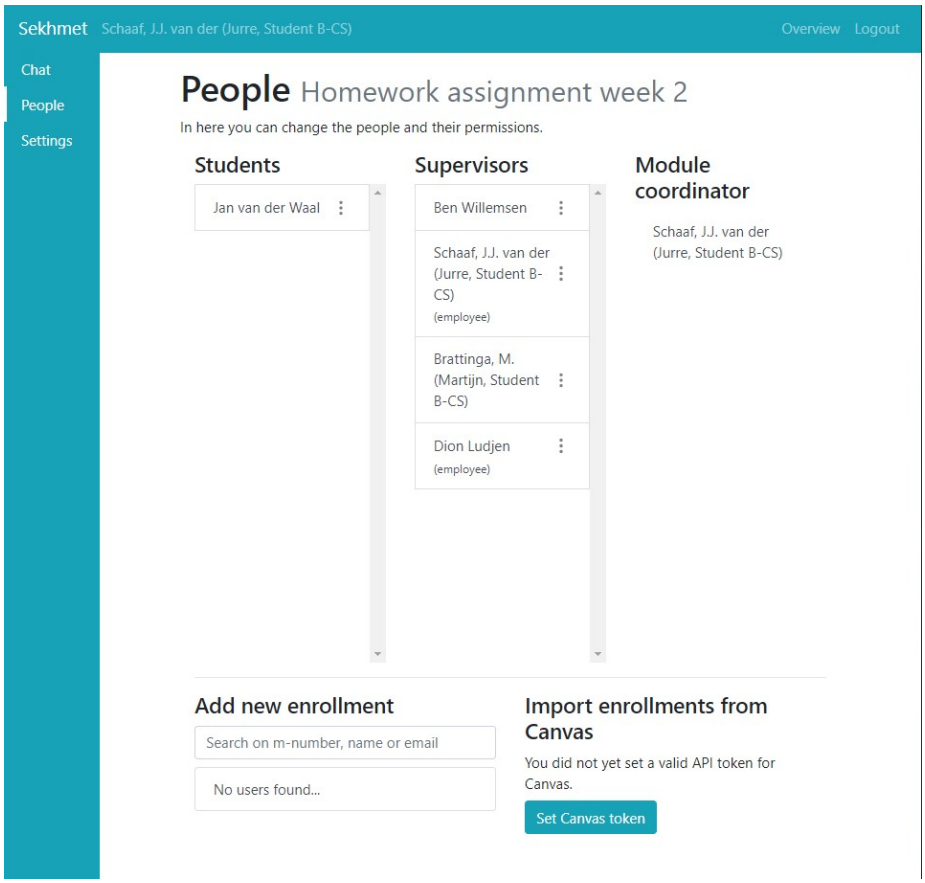

Figure 5.15: Test people page

This people page also starts with a title indicating the name of the test, in the same way, the chat page did. Then a scrollable list of students, a scrollable list of supervisors, and the module coordinator are displayed. By the options menu for enrollment, the three dots next to the name, enrollment can be removed (as in figure 5.16). The module coordinator cannot be changed on this page, as the module coordinator is not test specific, but course specific.

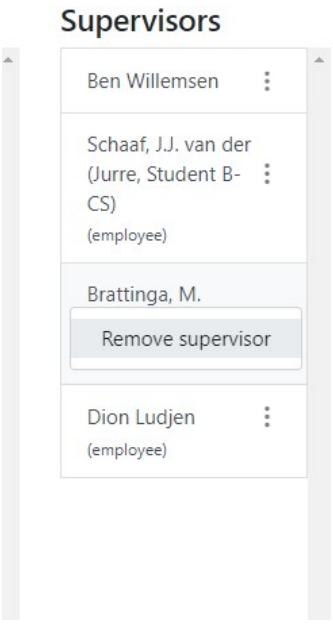

Figure 5.16: Remove enrollment option

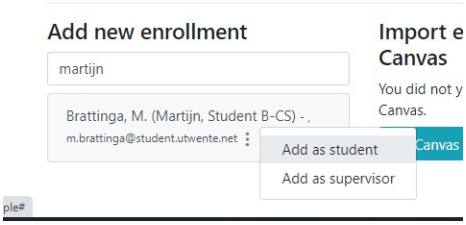

Figure 5.17: Add enrollment option

Below the overview of enrollment there are two options to add enrollments. On the left is the option to manually add a user that exists in the system, on the right is an option to import a student set from canvas.

To manually add an enrollment, the supervisor can search for the user by name, email or student number. Via the options per search result, by clicking on the three dots next to the name, the user can be added as either a student or a supervisor.

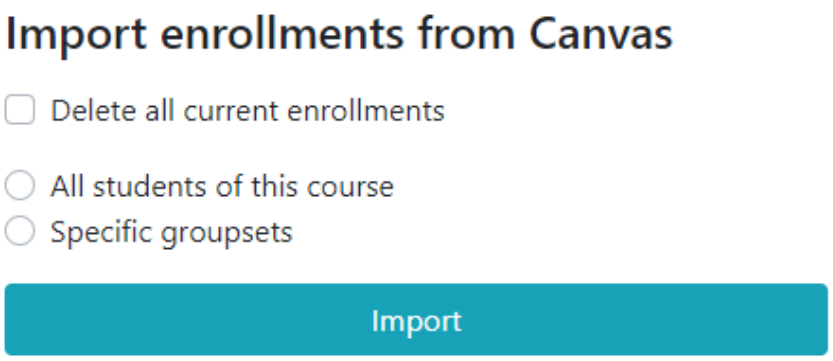

Figure 5.18: Import students to an existing test

To import enrollments from Canvas, the Canvas API token has to be set first. If this canvas token is set, the bottom right section will be as figure 5.18. All enrollments can be deleted, which can be an helpful feature if you want to redo all enrollments. All students of the course can be imported, or specific groupsets can be imported to the test.

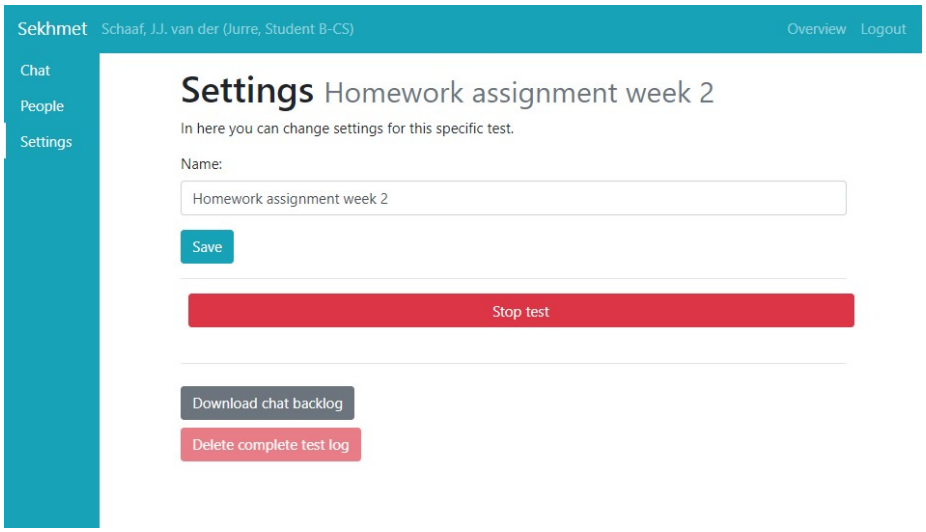

Figure 5.19: Test settings page

Settings for a test that don't have anything to do with enrollments can be found on the test settings page (figure 5.19). This page allows to change the name of the test, to start or stop the test (depending on the current state), to download the backlog, or to remove the backlog.

The backlog buttons are only for the module coordinator. The deletion of the backlog is not possible if a test is open (thus not closed yet). Deleting the backlog also requires accepting a confirmation pop-up first (as in figure 5.8, but with a different text).

Course management

The first page users see after logging in is the page showing all courses with tests, as can be seen in figure 5.20. Note that if a student only has one test active, this screen automatically redirects to that active chat. This prevents a redundant click to open a test, when that is the only possible click a student can do.

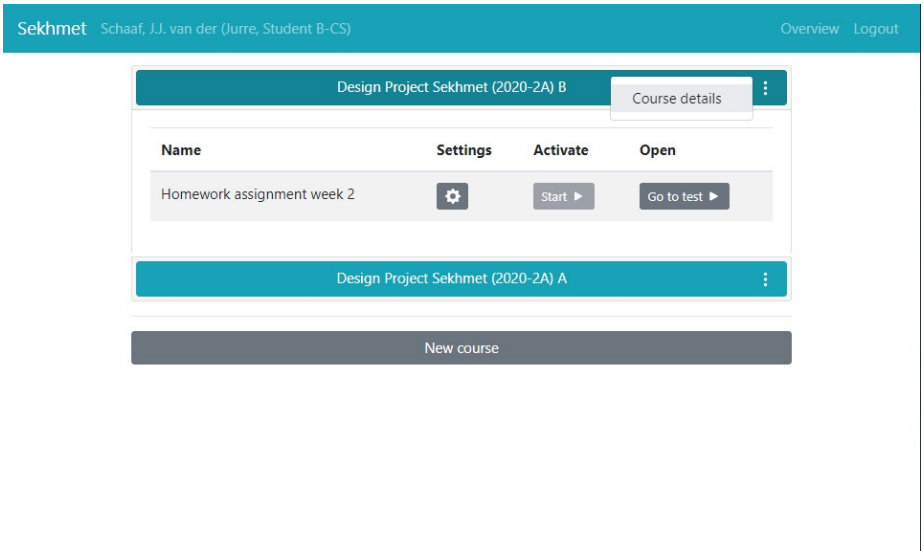

Figure 5.20: Courses overview page for a supervisor

The course overview shows every course in a list. A course can be expanded to also show the list of tests. For employees of the university, this overview has shortcuts to go to the test settings, to open the test, and to go the test chat. For students, only the latter one is available.

For employees, there is also a button to import new courses (see the bottom of figure 5.20). If the user already has set a Canvas API token, the user can import a course with tests (figure 5.22), otherwise, the user is asked to set an API token first (figure 5.21).

Figure 5.21: Import course from canvas when no API token has been set yet.

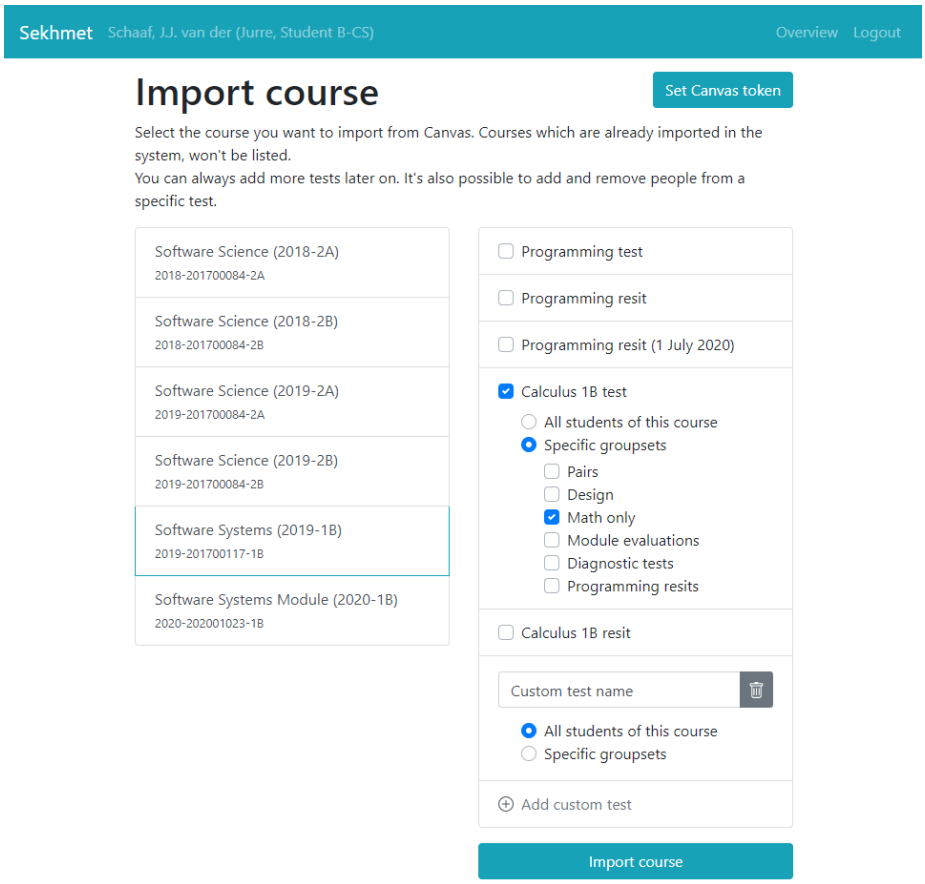

Figure 5.22: Import course from canvas.

On the import canvas page, on the left, a list of available courses is displayed. On selection of one, on the right all available assignments on canvas are displayed, that can be imported as tests. An extra option is to add a custom test if the user wishes to import a test that is not in canvas.

Per test, the user can select which students to enroll in the test. There is an option to enroll all users of the Canvas course or enroll specific user sets only. For instance, if you have a student set of resit students, you could only enroll them in a resit exam.

5.5 Security

Authentication
Authentication in Sekhmet is done by utilizing the Single Sign-On of the University of Twente. Using this service, students and employees can log in using their University of Twente credentials. The Single Sign-On is implemented using OpenID Connect (OIDC), which is a layer on top of OAuth2. OpenID Connect "allows Clients to verify the identity of the End-User based on the authentication performed by an Authorization Server, as well as to obtain basic profile information about the End-User" (from "OpenID Connect", n.d.). In our application, the End-User is the student or employee, our application is the Client and the University of Twente provides the Authorization Server, the place where credentials and user information are stored. After the user has clicked the "Login with UT credentials" button, they get redirected to the login form managed by the University of Twente. They provide their credentials there, so outside our application. Once the University of Twente has verified the filled in credentials, a user is redirected back to our application, along with a unique token. Our system can use this token to verify if the sign-in has succeeded and to retrieve some basic user information; user ID (student or employee ID), name and email address. Sekhmet now knows who logged in and is able to create a session for this user. For this implementation, we've used the build-in functionality of Spring Security. Because Spring Security is used and tested by many others, we can assume most bugs have already been found and resolved, and therefore is the best way to ensure a secure application to the users.

Authorization
The server checks with every request who the user is and what they can do. This is done according to their respective roles (see section 5.5). This information can be found in the database, so for every request, the database is accessed once, twice or thrice, depending on the complexity of the role. The hierarchy is as follows: system admin - module coordinator - employee - teacher - student. A system admin can do anything, module coordinator can do anything within their module, employees can do anything within a test they are assigned to and non-employees (Teaching assistants) can do basic operations within a test: starting the test, stopping the test, answering questions. A student can only ask questions and read the announcements. A visual representation is shown in figure 5.23.

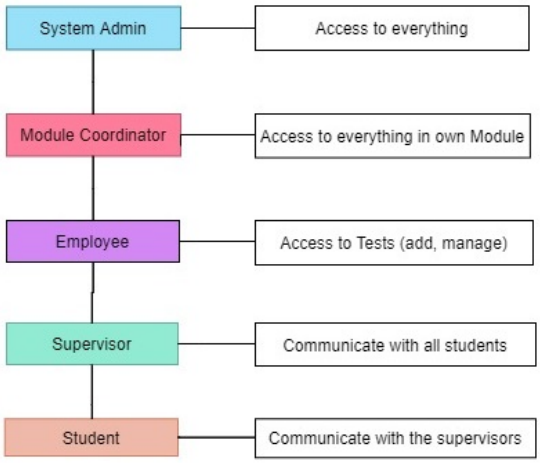

Figure 5.23: Hierarchy of roles

Roles
The different roles are as described in chapter 2. Whether the user is a teacher or student in a course depends on the enrollment, and whether or not they are a system admin or employee depends on the information in the users table, which is inserted upon sign in using the SSO information of the University of Twente. The reason we decided not to use Spring's built-in role system is that the roles are somewhat dynamic. This would mean that roles should be updated, and then the built-in system loses its advantage because then persistence needs to be verified. In contrast, using the database is still fast enough, completely dynamic and has persistence built into it.

Encryption
All traffic with our application is TLS-encrypted. In the current production environment, this is done by the Nginx reverse-proxy server[15] that runs in front of our application referring any and all HTTP requests to their HTTPS counterparts, denying the connection if this is not possible. If not running on an already secure environment, our project has the functionality to do this referring itself as well.

SQL injection
As mentioned, our application uses the Hibernate ORM and JPA framework. This reduces the use of direct database queries inside of our application. JPA converts the tables of the database into abstract objects, this way data of the database is accessed as if they are a Java object. Meanwhile, the retrieval of these Java objects is provided by Hibernate with minimal use of database

[15]Nginx is not part of our project, just part of the production environment. For more on Nginx: `https://nginx.org/en/`

queries. In addition, although our application limits the use of database queries, no user inputs (in form of String especially) are applied to the process of manipulating the database. Thus, SQL injections in our application (such as through the messaging system) is not possible.

Session cookies
To remember the currently signed-in user, session cookies are used. These cookies store a random string of characters, which in the backend can be mapped to a signed-in user. Because the user information is not contained in the cookie, there's no vulnerability to this. These cookies are sent along with every request for the domain Sekhmet is running on. This however does raise a possibility to do cross-site scripting attacks. More about how these attacks are blocked in our system is in the paragraph "Cross-site scripting (XSS)" below.

Cross-site scripting (XSS)
To protect the application against cross-site scripting attacks, the build-in CSRF protection of the Spring Framework has been used. The whole project, except for the endpoint for posting errors (see more in section 5.6), is secured by this CSRF protection. This means that for all POST, PUT and DELETE endpoints it is required to send along a valid CSRF token. This token, generated by the backend, is injected into the HTML of the frontend when it is served from the server. Only when this token is valid, the request is executed.

Spoofing In any chat application a serious security concern is users spoofing messages, fooling security by claiming to be someone else when sending a message. In our system this is not possible, since for routing and saving message purposes the sender is extracted from the http session, not the data within the message objects. This does leave the technical ability open to send a message to someone you're already allowed to talk to, in a chat you're already allowed to write in, but under a name you don't own.

5.6 Error handling

Front-end

The front-end has a general error handling that can display error messages to the user, and log error messages to the server. The former is used to let the user know something went wrong (figure 5.24), the latter writes the error and additional information to a log file to help maintainers of the system fix the errors.

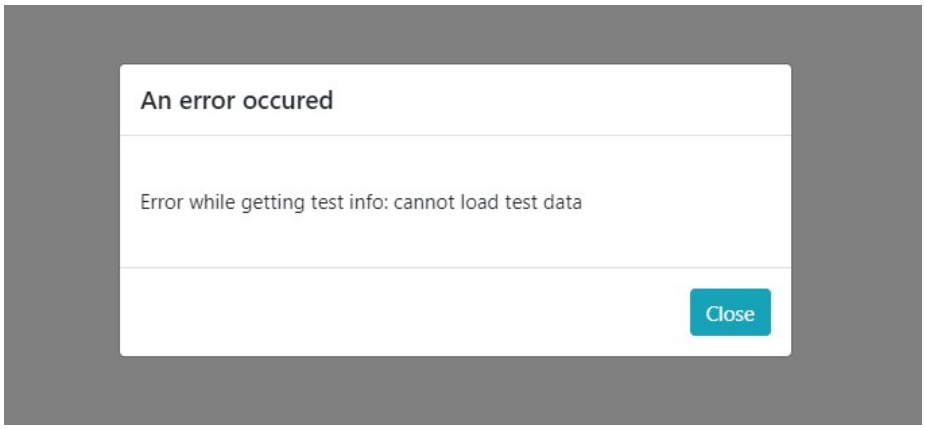

Figure 5.24: Error pop-up for a failed test load

On every HTTP request, the response is checked to be valid. If the response is not valid, for instance, because it doesn't contain the needed objects or the response code indicates an error response, an error is thrown.

Checking for errors is also done when WebSocket messages are received. For instance, if a chat message is received that has missing fields, or indicates that it belongs to a chat you're not part of, that is an erroneous situation.

At last, the error pop-ups are also used for the nack system, as explained in section 5.6.

Back-end

At every request, there could be an error. The server gives an appropriate response for each type of error in the form of an HTTP response with useful code and body. For most responses, the front-end can handle it without having the need to show it to the customer, but there are a few that have to be shown.

One of the possibilities is that the client is not authorized to access or update some information. This is checked before anything else and returns a 403 ERROR if the user does not have the correct authentication levels.
It is possible that a user gives an invalid JSON object as the body in a PUT or POST request.
When interacting with a test it is easy to do something invalid, like opening an opened test, closing a non-opened test, etc. These requests all throw conflict error, which is fed back to the user by means of an information popup.
Lastly, the most difficult to handle, the 500 ERROR. This one is thrown at every exception that is not thrown by an expected source. These cannot be handled by the front-end, because it is an unknown error. These are quite rare, luckily.

These errors are all not supposed to happen when the UI is used, so they are not specifically handled in the front-end: the user just gets to see an overlay with the error message. However, this response will only occur when the user ignores the UI and tries to access the endpoints themselves, so we don't see it as problematic that these errors are handled in a rudimentary way.

Nack

Apart from the endpoints, the server also works to connect clients using WebSockets. Where the endpoints return error responses, the client does not really await a response from the WebSocket. If a message is not stored in the database due to an error, the server sends the affected client a nack (non-acknowledgement) saying that the persistence did not work. Similarly, when the intended receiver does not receive the message, it also sends a nack warning to the front end. If the sender is a teacher and receiver is a student, it sends the nack if the student does not receive it (is offline), and if the sender is a student (then the receiver is by design a teacher) then the student gets a nack if no teachers received that message. Both these types of nack are handled in the same way in the front-end: the message in question is coloured red and an exclamation mark is added (figure 5.25), and the user gets a popup saying the receiver did not get the message (with either the student's name or with "teachers"). This allows the user to take appropriate action.

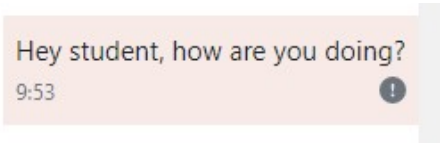

Figure 5.25: Indication that a problem occurred with a chat message

Chapter 6

Ethical considerations and security

In our project, there are two main ethical considerations: security: making sure our application does not put users in danger of their computers or accounts being compromised; and the balance between privacy and authorization: in order to verify that users claiming to be teachers are teachers and students are students.

6.1 Security

Cross-site scripting A fundamental part of our project is displaying text entered by other users. This technically leaves the possibility open for cross-site scripting. A cross-site scripting attack is virtually unbounded in its potential damage, as the victim's entire system could be compromised. Making protecting users from potential cross-site-scripting attacks an ethical must.

Database security In the current implementation of our system, courses, enrollments, and in general, all Canvas info is retrieved by a teacher's Canvas API token. Meaning we have the same permissions as that teacher has, including write access to grades and student submissions. These permissions would be seamlessly transferred to any other person where they acquire that API token. As a result, there is an ethical requirement to avoid database breaches in order to protect the validity of educational administration.

6.2 Authorization

A core requirement of our application is to disallow communications between students, but do allow communication between supervisors and students. To this end, supervisors and students have to be kept strictly separate. A potential

alternate design to achieve this might be by setting up a separate supervisor- and student-rooms with different join-codes. This would rely on supervisors to communicate their access code between each other without leaking it to the outside or any student. We, however, decided that the ethical implications of acquiring student's names, enrollments, and student numbers without their knowledge was worth the increased rigidity of the authorization, by hooking into the University of Twente Single Sign-On system.

Data ownership The first method of approaching this issue is through the lens of data ownership. Where a student's name, enrollment and student number are all data owned by the student (the point of view taken by the GDPR Council, 2016), and we are effectively forcing teachers to give us their students' data in order to use our application. This would be a massive ethical problem if our application was some publicly available program any person may sign up for giving us their, and their students' data. Except that our project is not publicly available, and entirely useless by design outside of the scope of our university. That leads us to the second method of approach.

Onymous context The second method of approaching this issue is in questioning whether the context calls for anonymity. Marx, 1999 described a number of rationales that might be applicable in favor of, and against anonymity in a given context. Partially because the context of test-taking already features identifiability as a crucial aspect, far more of the rationales for identifiability apply in our context than against it. Not to mention the fact that most other similar online tools used by the university feature similar breaches of privacy.

Chapter 7

Testing

There were four kinds of tests planned for the system: internal (automated) system tests, usability (HCI) tests, rigidity tests and exploratory (comparing) tests.

7.1 Internal system tests

Setup
The automated (internal) tests try out two things: correctness of behavior and correctness of security. Correctness of behavior is whether or not you can open an already opened test, or when an incorrect JSON in a request is handled correctly. It tries to access endpoints with all kinds of different correct and incorrect input and checks the error handling, the responses, and the state of the database.

Security tests also test for correctness of behavior, but specifically whether or not people get access to everything and only what they should have access to. This is done by requesting the method to which an API link is bound, and then seeing what happens when dummy users with preset roles try to access them. By accessing every endpoint with all possible roles, it shows that the system is secure, assuming that the roles are sent correctly to the endpoint (it only handles authorization of authenticated users, for authentication, there is the Single Sign-On supplied by the University of Twente).

The major advantage of setting up these tests early is that it becomes quickly apparent if a bug fix or update changed the behavior of the system.

Results
Due to the complexity of the simplifications Spring and Hibernate make, it was quite difficult to automate the testing. Firstly, Spring uses an @Autowired system to automatically instantiate a connection to the database. Unfortunately,

this proved to only work for selecting data, not updating or deleting. Because it is Spring, instead of giving an error, it just would not send a delete or update statement. So calling an API endpoint would work, because it was not a test, and using Hibernate within a test would not work. It took quite some time to figure that out.

Unfortunately, getting the tests working took the first 4 weeks (it was low priority, so the back-end got top priority). This did mean that the tests were applied later than we hoped, but after those weeks we did check the back-end after most updates. And it did find quite some behavior issues.

To be fair, a few edge cases still slipped through the net and were found during other tests, which were mostly due to unexpected database content. For instance: when an expected field, like assigned teachers to test, was empty, the JSON formatting did not go correctly. This did indicate that a server-side measure was required: the only correct JSON was to leave the server. The reason these cases were not caught by automated tests is that the database needs to have a specific setup in order for it to crash.

7.2 Usability tests

Setup
One of the requirements clearly states the user experience must be intuitive. To determine what potential users think of certain features and the UI/UX of the product, usability tests are conducted.

In these remotely conducted usability tests, a few potential users will be asked to use the application. Certain tasks will be given, which are representative of the use case of the type of user. The users are asked to share their screen with the team, in order to keep track of how the goals are achieved. In the case of students, as a preparation, these users will have already been added to an active test in the system, in the case the users are students. The task given to the student is rather imprecise. This way, a real scenario when a student has a question during an exam, where there's in principle no possibility to ask anyone how the system works. In the case of a teacher or coordinator, these people are asked to import a course and test, with teachers, TAs and students, and make it fully ready as an exam actually starts at a specific time. This way, the full procedure teachers or coordinators have to follow to set up the system properly is simulated.

Although students may not be the most important users, because they use a rather limited part of the system, most of the users are however students. Besides, students use the system during their exams, which is not a desirable moment to search in an interface for the right place to ask a question. This is why usability tests will be focused mainly on the student's interface. As the number of different and specific elements in the interface is rather limited, especially for students, not many different users are needed to find most issues in the

system. According to Virzi, 1992, "80% of the usability problems are detected with four or five subjects", thus these test is conducted on five people, namely a few students during a peer feedback session, as well as the two supervisors.

Results
All the usability tests were, in the end, run on 5 students (not including developers), 1 teaching assistant, 2 teachers (the supervisors), the developers themselves, and some friends and family that wanted to see the system. While we did want to test on groups of hundreds, we feel that the tests we ran already gave us some good improvements. The main improvements are listed below.

- **Problem:** the menu was a bit unclear because users that tried to get the overview of tests got redirected to the current page because they only had one test active.
  Solution: we hid the sidebar if a user had no need of the overview (because they only had 1 test)
- **Problem:** the menu options were somewhat unclear: they did not know if options were specific for the current test or global, like the settings and people page.
  Solution: we moved the global pages to the top bar and the test specific pages to the sidebar, and that sidebar would only appear if there was a need for it
- **Problem:** it was unclear that the system was working, like when opening a test or importing a lot of information. This easily takes 20 seconds, and the user had no feedback that it was working.
  Solution: now when those buttons are clicked, they are visibly disabled, then there is some feedback using a simple loading wheel animation and a message that it can take a while.
- **Problem:** a remark on the list of chats in the teacher view. If there were 100 students in the test, it would show 100 conversations.
  Solution: We hide empty conversations, clearing up the view tremendously
- **Problem:** the student does not know who answered their question. However, the endpoint still sent that information, the front-end just did not display it. If they went into the browser console, however, this information was readily available.
  Solution: we don't consider this a security issue, and not even a privacy issue, since with the old system you would immediately know who answered your question anyway. Either way, it was not what we wanted, so that was fixed.
- **Problem:** the buttons that could not be used by a specific user would not even be visible. This was sound advice, and we had already implemented

this in part, but apparently, we overlooked a couple of buttons.

- **Problem:** there was no way of knowing whether or not a message was received.
  Solution: we implemented a nack system to be certain that a message was received by a (or the) receiver. More on the nack in chapter 5.6.

The tests did show some positive points. The login screen and redirect were quite clear. Also, the interaction with the chat system worked very well. People easily understood how to use chatting, where to send responses, which chats are unread and how to edit or delete a message. The announcement also worked quite well, although someone did make a point that only the last announcement was shown on the main page, we decided it would clutter the screen too much if we included multiple announcements, especially because the previous announcements are also still available. Teachers also considered it good that they could easily see that a student was being helped and who was helping them. They also found the way of making announcements sufficient.

7.3 Rigidity tests

Setup
The setup of the rigidity (or full system) tests were such that a lot of users just use our system and we collect data on that (bugs, delays, etc). This shows whether or not the prototype we let them test is actually a (minimal) viable product. We had set up this test during some regular exams of Technical Computer Science. The first test would have had 50 participants, the second one would involve 400 participants. We also got the opportunity for a third exam to test our system on.

For both these tests it suffices to let the participants use the system in a real scenario. This will give rise to any potential problems in the system. All errors occurring are logged, both the front-end and back-end errors, so we don't need interaction with the students during the test. This way the testing is the least disturbing, and participants can focus on their exam.

Results
Unfortunately, we could not complete the rigidity tests, due to the system being too unstable at the time of the exams. However, we did ask some students to test the system when it was stable, and they did address some issues with the behavior. So although it was fewer users than we had hoped, it still surfaced some bugs that could be fixed.

These tests also introduced a new problem in the program: database access. For database queries, we used the JPA package. Unfortunately, the way we set

up the ORM was quite badly optimized. So it introduced us to the N+1 problem: having to create a query for every single object that is retrieved to retrieve its children. This was not as apparent earlier, because testing was done with only ten or twenty people in the database, so it was still super fast. However, when there were 500 people in the database, it slowed down enormously. This was solved by replacing the automated queries that we used thus far with custom queries that would still only retrieve the necessary information but would do so in one single request.

7.4 Exploratory tests

Setup
At the start of the project, we wondered whether a peer-to-peer network system would be viable or even desirable for communications. This would really depend on the number of simultaneous exams and amount of people per exam. If there are a lot of simultaneous exams, it might relieve the stress off the server if the messaging was handled locally, and the delays would probably decrease. If there were tests with lots of users, this would most likely become infeasible.

We set up a test that would stress the system hard, using 50 Chromebooks that constantly send messages to an off-site server. Then we would measure the delays if people were using peer-to-peer networking, or normal client-server-client messaging. The Chromebooks were set up in a potential exam room at the University of Twente.

Results
Unfortunately, due to the routing system within that room at the University of Twente, we could not use our server as a surrogate Domain Name System, so we could not set up peer-to-peer communications. However, because the test with standard client-server-client communications went so well with such small delays (20 ms - 1000 ms under enormous strain) that we decided client-server-client communication would suffice. Since the server was able to handle those approximately 300 messages per second, the real usage would be way friendlier. Even with 10.000 questions in a single test, this would still only come down to 3 per second (if the test takes an hour).

7.5 Discussion

The automated tests give a good indication of how well the certain parts of the system behave. The system seems secure as long as the authentication is secure, as we tested all endpoints with all possible roles and it returned exactly what was expected.

It would have been nice to have more input from different potential users. If we

could have had our prototype stable for one of the exams, it could have given input from a hundred or more students. This would most likely have had an impact on our UI design, and might even have surfaced hidden bugs. Nevertheless, we do feel that it was a good choice to deploy a version during an actual exam only if we were 100% certain it was stable enough to not hinder students during a real exam.

The choice of networking system, peer-to-peer or client-server-client, has not given any problems. The tests showed that the server was put under quite a bit of strain due to unoptimized queries, which would affect client-server-client harder, but after that was fixed, we had no problems whatsoever with people getting notable delays when chatting.

There were also no problems found with any code insertion attacks like SQL injection or XSS, thanks to the character escaping and the prepared statements in the back-end.

Chapter 8

Reflection

8.1 Group reflection

At the start of this project, we failed to oversee the required complexity of the intended product a bit. That is why some design choices proved somewhat overengineered, whereas the complexity of other fields was a bit underestimated. What follows is a reflection of the most influential design choices, our hindsight opinion on them, and how they could have been prevented or improved where applicable. It also includes a reflection on the process and organization.

8.1.1 Technologies

There are a number of technologies we decided to use, or specifically not use in our project. In the following section, we will discuss the impact this had on our project as a whole.

Spring
Spring has quite some features built into it. If It was clear to us how to use those features, this process would have gone nice and smoothly. Unfortunately this was not the case.

Firstly, the Spring Security, which should have made authentication easier, did not explain well what it did. So to create your own specific use case is quite a nightmare. Due to the lack of documentation and the sparse occurrences of good examples online, this took a lot of energy.

Secondly the integration with Hibernate through JPA. The problem here was that we just wanted a database with UTF8 encoding, but there are so many possible solutions online of which most don't work for our exact setup, it takes a lot of effort to separate correct from incorrect. This was especially the case when the solution did work locally but did not work on the

Similar to the debacle with STOMP & SockJS discussed later, an example of the clash between our project group and spring happened when first serving web pages. In the "default" spring-boot environment adding `@getMapping("/some/path")` in front of a function returning a string, spring will at the given mapping serve the html page found in resourcestemplates with the filename corresponding to the returned string, file extension not included. This is very convenient, but also deeply magical. When first looking into serving pages we wasted a full day going through tutorials and code-bases in the hopes of finding exactly what section of the program was responsible for making this link (ultimately discovering it was Thymeleaf, when we removed it. We did not actually use the templating feature).

The advantage of of the framework is that it makes it very easy to start up the server itself. There is no issue with threading, endpoints, TomCat configuration and run-on-startup classes. Unfortunately, this element is somewhat veiled by the lack of documentation. So debugging this is really not an option. The @AutoWired and @Component annotations worked most of the time, but whenever they did not, it was a try-all-and-error trajectory up until the point it worked.

Hibernate
Hibernate as ORM was not a bad choice. However, there were some quirks that Hibernate had that really delayed our progress. We will mention a few here.

Hibernate unit tests will allow you to create and update the database, but won't allow you to delete for some reason. Not knowing this for a (too) long time made the testing of deletion a complete nightmare due to inconsistencies in the automated test and a manual test.

Hibernate needs to handle all its relationships, and it is very easy to overlook some. This is not a problem per se, but it does make it quite difficult for a beginner to grasp all correct concepts and make the right cascading, dereferencing, and updating choices. This also took some time to get right, because wrong relationships are not checked upon compilation. So to find these errors, we had to gain more knowledge, then comb through all entities and update them accordingly. This was eventually completed.

Hibernate also has some trouble with loading nested objects from the database in a compact manner, which we will go into in paragraph 8.1.2:Database design.

Whether or not Hibernate ultimately saved us time or cost us, is hard to say. It should be mentioned however that in one of the final weeks we were seriously discussing ripping out hibernate entirely, and re-writing all database interaction by hand in raw SQL. After going to bed angry and confused, the next day a

new Hibernate feature was discovered however and the problem in question was solved.

Vue.js
The reason we wanted a front-end framework was to speed up development and make it easier to create a great user experience. Despite the fact that learning a new framework also requires an investment in time, we do think that using Vue was a great choice and that it did everything we expected from it. There was enough documentation and the community support was great as well.

Peer to peer
Peer-to-peer communication (p2p) was a mistake. Not a mistake in hindsight either, we should have built the some one the bench-marking by the end of week one. Early on some people suggested that p2p might not be necessary, just a flashy feature that we were hyped about. And while it was true that the fears of p2p's effects on security and authorization were unfounded, the unexpected presence of a NAT increased the expected set-up cost of p2p significantly. Enough to make us reconsider whether it would be worth it. All-and-all we lost the better half of a work week in man-hours on p2p, 3 days to set it up + 1.5 days for the improved prototypes and benchmark code.

WebSockets: not using STOMP& SockJS
When using WebSockets within the spring framework it is often all but assumed you also use the STOMP[1] protocol and SockJS[2] WebSockets. After some research, the conclusion was reached that neither of these technologies would pose any advantages over self-defined protocol/plain-text via standard browser WebSockets. Avoiding these two technologies has a significant impact on maintenance, as such are listed here. The lack of documentation of spring in favor of STOMP+SockJS-tutorials ultimately meant the decision to not use these technologies cost about two working days in man-hours, as it became very difficult to find people with similar problems using a similar solution. In the end, it is hard to gauge whether avoiding STOMP and SockJS was worth it. On the one hand, we did end up implementing NACKs (non-acknowledgment messages) ourselves and discuss here that ACKs might have come in helpful, both of which are part of STOMP. On the other hand, the maxims of KISS (Keep It Simple, Stupid), or Occam's razor as applied to software development would suggest that choosing to avoid these technologies was correct.

8.1.2 Design

There are a number of aspects of our design that posed a hindrance or would have helped. We will discuss these in the following section.

[1] `https://stomp.github.io/`
[2] `https://github.com/sockjs`

Database design
We created the basic database structure quite early on in the project. This design was fine, but a little more complex than necessary due to the unknowns still in the design, like whether or not we would use p2p. But when we chose particular designs, the database was extended accordingly, but never reworked. This increased the complexity somewhat, but it still felt handleable. However, what we probably should have done is recreate the entire database structure to match the design better. We updated primary keys to be compound, we let the front-end create keys, etc. This was in hindsight probably unnecessary and made both the front-end and the server needlessly complex. However, that change would increase the database size and therefore the carbon footprint.

We could also have used an automatically generated person ID to increase privacy: The ID that we store of people is the same as their Canvas id and makes them recognizable. It would have increased privacy somewhat if we had only sent (randomly) generated ids to the front-end. However, since that information is only available for people that are logged in, and those people could access that information in Canvas anyway, it was not a major concern of ours.

Back-end structure
The back-end had a few minor design alterations, but no big reworks, even though that was probably necessary.

Due to our unfamiliarity with Hibernate, we could not let the Gson library (which transforms objects into their textual JSON representation) transform our entities, because the library did not know where to stop with nested objects. We then made the mistake to create a way more complex entity-to-string system that used string concatenation. This made the back-end contain some bugs, so it delayed our progress somewhat. In the end, we decided to still use the Gson library, but use its Json builder to create objects instead of letting Gson try to automatically transform an entity to Json or using string concatenation. This improved the reliability tremendously. It was a step that we should have taken way earlier.

Another wrong design choice was to let the front-end generate message IDs, as mentioned in paragraph Database design. This was still set up when we did not know if all communication would go through the server, but when it turned out it would, we should have let the database create the ID's, which would allow us to send acknowledgments to the front-end, which would have improved database complexity, server complexity, and error handling.

A very unexpected problem that surfaced late in the project was with database interaction and transactions. Hibernate suffered from the $N+1$ problem, where for a group of objects, the database would be queried for the children of each object separately, and in that object had children, that would be queried sepa-

rately, etc. This would create an exceptionally large amount of singular database queries, which slowed down our server enormously. This only started to occur when we imported real tests from canvas, with 500 users. Because we had used about 10 users in our database up until then, the $N + 1$ problem was less apparent. To solve this, we defined a lot of queries ourselves instead of letting Hibernate define them automatically, and used SQL Joins to reduce the number of queries from $1 + 500 + 500 * 2$ to just 1 more complex query. ($1 + 500 + 500 * 2$ is an example of database grabbing 1 test with 500 enrollments in 1 query, then 500 queries to get the users that are part of that enrollment, then $500 * 2$ queries to get the conversations and the messages within there). Unfortunately Insert statements could not be done in a single command, but that was handled by creating batches and using batch inserts. This still created the same amount of queries, but only 1/50th of the server requests (for batch size 50).

Something that was mentioned in the meetings was loading (large parts of) the database into cache. It would have sped up the server, but we did not dare to allocate our limited resources to this task due to the deadline coming very close. It would solve the delays when inserting, and retrieving would lose the connection delay.

Error handling
More towards the end of the project, we decided to beef up the error handling. Up until now, the only errors that were handled were with API calls, and not with the WebSocket communication. If we had thought of this issue earlier, we would probably have been able to rework the database, back-end, and front-end to create an ACK system (acknowledgment for every message stored/forwarded/handled), so that the client would always know the status of their messages. However, due to the time constraints we settled for a NACK system (notify user only when something throws an error or is not received by target). This is still a massive improvement on the fire-and-forget we had up until then, and improved usability enormously. It is not as robust as ACK systems, but it's a close second. It can't fix the errors, but the client will know there was a problem.

We should probably have designed the error handling schema earlier in the project so that we could refine the possible fixes the front-end or back-end could apply.

8.1.3 Teamwork & Organisation

In the following section, we will discuss a number of factors in the teamwork and our organization of the project.

Task division
In the early stages of the project, most of the task was divided based on the architecture of the system which are the back-end, front-end, and database. This

seemed to not cause any problem at first, however as the project goes on we found out that some of the features for the web application are dependant across the architecture types. As an example, in our system, the front-end requires the back-end to serve data in the form of JSON, however since different people worked on the front-end and the back-end at times the JSON form expected by the front-end was not what the back-end actually serve. Furthermore, changes made on one part of the system (e.g. front-end) require more than one person to work on it while keeping each other informed on the changes on their part. Implementing this practice for the whole duration of the project without any mistakes turns out to be difficult and many errors in the system are a result of this. Together with the observation that the more feature-oriented code required much less review, leads us into thinking that perhaps it would have been wise to divide the task differently based on the features. Although we could not really say firmly that this is better, as many features are intertwined, especially in areas such as the database.

Planning
The planning phase was one of the successful parts of the project. During the planning, we were able to extensively think and discuss what would be best for the web application, such as the functionalities of the web application, how we would implement them, and also when we should have a functionality ready. We were planning on the things that will lead to successfully create of our web application and we can say that this went well as we ultimately have a working web application. However, an aspect that we seemed to have overlooked was planning on preventing the things that will disrupt our planning. Along the process of the project, we faced issues like the difference in protocols (such as the endpoints list) for features that need the functionality of both the back-end and the front-end, and this does disrupt our progress. Though we eventually did specify the protocol, it could have been done earlier in the planning stage.

Documentation
Because we had open and nearly constant communication within our group, one trap we fell into was a failure to document many smaller, granular decisions that are crucial for functional interaction between practically disconnected pieces of software. As a result of this, when testing started we found that a large amount of the protocols for back-end-front-end communication were mismatched, causing a large number of avoidable issues. On a similar note, the failure to keep to what documentation we did have, in the design mock-ups we made before we started development. Since a number of new feature suggestions we got in early demonstrations were already accounted for in those mock-ups.

Testing Testing was an aspect that we also kind of overlooked as well. We did plan a test procedure mainly for the last part of the project, but we did not plan to test for in-between changes and new feature implementations. If testing

were done as early as possible in the development phase, we could perhaps fix bugs and faulty parts of the system as they show up along the way instead of waiting for them to pile up and fixing them at the end.

A related issue our supervisors are very familiar with is our repeated self-inflected demo-effect. Where we would find some minor issue during a practice round for an upcoming demonstration, would fix it quickly before the demonstration started, and then end up in the demonstration itself with our pants down and a program that didn't work. A possible solution to this would have been a staging server, we however were not in a situation where we constantly needed to have a running version ready. What we needed was the self-control to not show off the absolute shiniest, best, latest version.

While we did not have a formal staging server, we did have local testing with a separate selection of databases. Here we ran into the issue that our testing and production databases were not exact copies. Most notably is that our test databases did not check constraints, likely due to a difference in the database engine we overlooked during setup. As well as the selection of users in the test database being far smaller, leading us to only notice a major performance issue once we tried for our first full-system-tests.

Speaking of full-system-tests, we did not do any, but not for lack of trying. We had a number of tests scheduled, but for each and everyone we did not manage to have a satisfactorily stable program ready in time, once even going so far as to do initial testing of a new version inside of the room where the test was to be held, shuffling our way out as the students started shuffling in. Theoretically, we should have had an old, stable prototype kicking around as a proof-of-concept for exactly these situations, though this would have required a development philosophy of building out from an older prototype, rather than building large individual pieces and connecting them. Notable though, our last test for which we had a version that was merely buggy, but theoretically usable the test fell through due to a major bug in another experimental piece of software making the teacher unwilling to expose their students to more tests at that time.

Lastly, still somewhat related to the concept of testing is that in this project we learned the value of code reviewing. There are often simpler, far more maintainable solutions to problems that you are not seeing. And a code review helps both to manage the degree to which a piece of code is "your handwriting", as well as to keep at least one other person informed in detail of what you are doing.

Team communication

The team communication worked very well regarding organization. The 2-daily meetings made sure we were all on the same page regarding the planning, and it ensured we all knew what was being worked on, by who and how long it

would take (approximately). The only thing that could have been improved was the communication on the structure of stuff. Because we split up the work on front-end and back-end between different people, we should have had more collaboration on formatting and required information, and especially writing these things down. In written documentation it is way easier to spot differences than in verbal communication. In the end, this communication went better, but especially in the beginning it went a bit sluggishly. We do feel that communication in the team, and also with the supervisors, went in an open, friendly and safe way.

Communications with LISA
Communication with the LISA department of the University of Twente was required to set up the Canvas integration and the Single Sign-On. At first, we had good communication with LISA about integrating Canvas. We got all the information within a short period of time and could start working on the implementation. We were able to ask questions quite concrete which led to an efficient communication. However, later on, it turned out that we missed something in our communication with them. We started on the test environment of Canvas to test and implement our solution. When we wanted to upgrade to the production environment of Canvas, it turned out that we had to fulfil some requirement which we couldn't meet, and therefore we had to implement an alternative approach (more about this in section 5.2). These were: it needs to be hosted on a LISA maintained server and there had to be an specific maintainer who is responsible for maintaining, bug fixing and enhancing the application. We did not expect this (partly because the upgrade to production environment of the Single Sign On did not require anything special), but also because we did not communicate about this. If we would have known earlier (so if we asked about it), we probably could have managed something so that this could be used. Communication with LISA about Single Sign On was somewhat rough and time-consuming, but in the end not that bad. By preparing everything we could before we had the actual information (like registration IDs and secrets), it didn't led to big delays in the development of this part of the system.

Iteration of design
With the design iterations, we overlooked some options by being too focused on a specific choice. For example, when we discontinued the p2p system, we should have taken a step back and seen how that choice could make our entire design different. Instead, we focused on whether or a change was needed in order for the system to work, and the answer there was no.

Another example of being too focused on a specific problem or choice is with online users. In the current system, only the back-end knows who is online, and the front-end just knows who could be online. If we made that more dynamic, we could have changed the structure of the back-end on error handling, we could have reduced the amount of data that had to be appended for a lot of requests,

just because the front-end already knows the address book. If we had zoomed out and reconsidered the entire synchronization and state between back-end and front-end, we could have made the interaction simpler.

The problem with not zooming out and reconsidering design choices is that it's a downward spiral: fixing one design issue creates the need for another design choice, and it becomes more and more complex. The only thing needed to break through that cycle is a reconsideration of the entire design, but unfortunately, none of us realized this issue until we were done with the system and had run out of time.

The positive
There were a number of things that went right in this project, however, mostly relating to the dynamic of the project group. During one of our very first meetings, we discussed what we wanted from the project. And even though this group was assembled from the "If you're having trouble finding a group, mail me"-list, we unanimously agreed that our aim for this project was "to build something we're proud of, and then we'll see where the grade ends up". This sense of "being on the same page" managed to persist for almost the entirety of the project. Besides just generally being able to have fun with each other, when it became clear our project was struggling we all agreed to work 14+ hour days and through weekends, as well as to the day of rest after that deadline had passed. Everyone was open to questions and suggestions, and the discussion was lively, but never heated. We helped each other, learned things from each other, and when motivation started dropping, propped each other up to salvage from this project what we could.

8.2 Individual reflections

8.2.1 Ben

When we first moved from open design discussions to proper development, I took the task of setting up the real-time communication. I initially started work on peer-to-peer communication, research mainly as I was entirely unfamiliar with webrtc and had kept my exposure to Javascript to a minimum. After a few days, I realised webrtc was dependant on WebSockets, so my focus shifted, and for WebSockets we needed to serve web-pages so with the help of Dion we set that system up as well; this was by the end of week 2 of development. After moving these features from proof-of-concepts to usable prototypes I set up the production environment, so that everything would be ready for the exploratory test in week 4. After that catastrophe, I gave myself until the next Wednesday morning meeting to get p2p to work. On Monday I did some benchmark tests for the WebSockets, and came to the conclusion that p2p was overkill. Instead, I began working on permission checks at each endpoint, and having the WebSocket intercept and handle important messages rather than just blindly

forwarding them. After that was done, I would try to help in debugging, as I was the only one with access to the production environment and database. As increasingly the only unfinished parts of the project were found in the front-end, the next task I really "took on" was the reflection report.

P2p To mutate a phrase: I was so preoccupied with the fact that I could, I didn't stop to think if I had to. As discussed in the group reflection, p2p was a mistake. Peer-to-peer just appealed to me, and honestly, to this day I believe that Sekhmet with p2p as designed would be a better piece of software than Sekhmet without. It would be cliche to claim that our project was done-in by ambition, so let's call it a more base desire for an unusual eye-catching technical feature instead.

Tunnel-vision Especially early on in a project, it is quite easy to develop a kind of tunnel-vision. My networking code initially was build to be entirely generic, imported as a library in other parts of the project. As a result, it did not matter to me what was happening in the rest of the project. I stopped paying attention to design discussions in our meetings, giving updates and answering questions when called but contributing very little. When I re-surfaced with my module and instructions on how to use it, the rest of the project was still foreign to me, even though it was already about a third done. This effect is somewhat unavoidable, but I do regret mentally checking out of the meetings. I was one of the bigger advocates for keeping up-to-date semi-standardized descriptions of the interfaces between parts of the project, -especially ones that fail on run-time, such as REST endpoints and intercepted web-socket messages- and early in the bug-fixing phase a lot of issues were caused these interactions being mis-matched.

Agreeing to spring True, this was a group decision, but it is one I also endorsed. Having forgotten how Java is second only to C++ in how notorious it is for featuring massive ornate frameworks, I also argued: "Well spring and Django do the same thing, we're not particularly familiar with either, how hard can it be?". Alternatively, when it took me two days to find a tutorial on how to set up web-sockets without sockJs or STOMP (since having looked into it, I decided we needed neither), I could very much see it coming that the focus on boilerplate-ridden tutorials over formal documentation was going to be a problem. Instead I did nothing. The amount of time lost on spring is immeasurable, since the time investment of "figuring out how to get spring to do a thing":"building that thing" was shockingly close to 10:1.

Silent refusal to touch the front-end To be fair, I did touch the front-end once. I implemented logging of abnormal web-socket closes to our error endpoint. Nevertheless, it took having quite literally nothing else to do at that moment, and after seeing the toothpicks, chewing gum and wet paper towels that hold up any interactive web-page I did proceed to touch the front-end

exclusively for debugging communication. Now while I can't honestly say I personally believe my life would be better had I worked on the front-end more, I do believe we should have had another pair of eyes on the front-end from day one. And since I was weirdly floating from minor odd-job to minor odd-job for half a week, I should have spend that time critically dissecting the front-end instead.

8.2.2 Dion

Spring During the first weeks of deciding to use the Spring framework, I have no problem with it whatsoever as I am relatively new to Spring as well as to the previously chosen framework Django. Simply said I have no preference over the framework. As we go along the project it started to show up how complicated it is to learn spring (as someone who has never touched it before). Documentation on classes, Java annotations, etc was difficult to find which then resulted in me investing time to only figure out how a part of Spring works, leaving me less time on actually creating the functionality of the back-end.

Task Division At first the task division was divided based on the structure of the system (front-end, back-end, database, etc) and I was working in the back-end which I really liked. As the project progress, it turns out that most of the part of the project are included in the back-end such as implementing the database and connecting the system to canvas and others which at the beginning was not assigned to me. Thus, in the end I mainly work on creating the endpoints of the system. I feel like I could have and should have done more, especially in the end of the project where countless bug kept appearing in the front-end. This leads me to think maybe I should have tried to divide my focus into helping the front end as well somewhere in the middle of the project since Martijn is the only person fully working on it, in which ultimately, I did not do.

Project Overall, I really like the project. Creating a chat system is interesting and I learned a lot during especially the technicalities we used in our web application. It was really nice meeting and getting to know Ben, Jurre and Martijn during the project (I already knew Jan before the project). Working with everyone was a fun experience.

8.2.3 Jan

I feel like quite a few things could have gone better over the course of this module. I will list a few possible improvements for me, though there are more than I can get into here.

Firstly, the ordering. I think our planning was very good, but we overlooked something. It would have been better to first determine the endpoints that were needed, and only then implement them. This would probably have sped up the process , since then there would not have been double work or reworking the

endpoints. That was my mistake, because I just started coding the endpoints and just getting the data from the database. Most of these endpoints were correct, but some were later deprecated or added.

Another thing I should have focussed on is testing. I wanted to have the endpoints ready for the front end, but the front end could (and already did) work with dummy data. So I could have focussed more on creating system tests, and solving all the problems associated with Hibernate and JPA testing. Then the front end would have had to run for one more week on dummy data, which would probably have been possible, and the back-end could be tested with every update if the behaviour was still correct. This could probably have saved time in the long run, but I focussed too much on finishing the endpoints. The problems with the tests themselves, where Hibernate does not want to delete values in database would still have stirred, but then they could have been solved two weeks earlier, and everything would have been a bit smoother.

Another issue I should have solved was about front-end. I could not get Vue to compile consistently and kept forgetting the build steps, so I just tested all endpoints only on server side. I should have just built the front end everytime I wanted to test what I created, because in the end we found out that there is a JSON format exception in the frontend (for actual invalid JSON) that slipped through the backend (back-end did not see it as invalid). This, and other bugs like these, could have been prevented by me if I just used front-end for testing the implementation.

On the subject of front-end, I do think we should have divided the task better, because I think the front-end coders had more work than the back-end coders. If we had divided those tasks, we would have had a high fidelity prototype earlier in the process, which could have helped in development. So next time I do a project with multiple people, I will make sure to put more resources to front-end until a high fidelity prototype is obtained. Then we could have built that out into the final product step by step, and on each step there would be a working prototype.

On the plus side on the organizational part, I do believe that the two-daily meetups were really helpful to see what went on and how people were progressing. This really helped with the oversight. I believed that it could have been better if we implemented some more Scrum features in those meetings, specifically having partial deliverables and working in sprints instead of one big trajectory.

8.2.4 Jurre

As my main task was 'authentication and integration', I worked mainly on the integration with Canvas (import courses) and the authentication with the Single Sign On of the UT. I really liked this task, because for me this was something

I was not yet experienced with. The good thing is that I indeed learned how these techniques (particularly OAuth2) work. Although, while I was working on this and didn't fully mastered the Spring Framework, which I worked with to implement this, I got demotivated to work on it, and also a bit on the project in general. I think this could have been prevented by better diving into this specific framework, as it turns out different frameworks don't have to share a lot of similarities. This demotivation however didn't take took long, and once I understood the framework slightly more, thing came together and worked.

Learning the framework This leads to the second point: invest time in learning a new framework. Although I've always learned new frameworks and techniques by just making something with it, but I believe it would have been totally worth it to invest some days in watching YouTube tutorials and reading blogs to get an introduction of the basics of the framework. I'm still not really convinced/satisfied by the Spring Framework and if I would redo this, I would pick another language/framework combination.

Design choices Altough we talked about design choices, I think we all made our own assumptions on these designs, which turned out not to be the same. Sometimes in some details, but important details. I think that is one of the reasons (among others) why different parts of the project didn't come together as we expected them to be. We should have worked those designs out, and 'present' it to each other. If needed, this can be changed, but more important, we're all on the same page. That has gone wrong several times in this project.

Making basal design changes In my opinion, we should have made some substantial changes in our design somewhere halfway the project, of even after that. We've discovered things didn't work many times. Over and over again, we just looked at fixing the bugs in the code, but we never went back to the drawing board to implement another approach. Of course, at first glance this looks like a larger task to deal with than just fixing that one bug, but in the end it's probably not. Finding and fixing bugs over and over again takes lots of time, which could partly be resolved by redesigning some fundamental parts of the system, making it easier to write the code and thus is less error-prone. This again, will lead to a better product.

Feeling about the project To conclude, I'm quite sad about the feeling I finish this project with. I looked forward to this project as part of my studies and started really motivated. I was really happy with the choice of project within the team. As the system was bugging all the time and demos were failing, that changed drastically. In the end, I'm honestly looking back on the worst part of my studies (hitherto). I hope I can change that feeling somewhat in the upcoming weeks, as I will probably get some more rest and time to do other private projects, but it will definitely not meet the expectations I had before starting the quartile. The people were great to work with, though.

8.2.5 Martijn

Task division For this project, I mainly contributed to the front-end. I really liked focusing one one part, such that you can really master it and know everything about it. On the other hand, it also brought in some difficulties for me from time to time. As I didn't work on the back-end, I also didn't know where to solve small mistakes there. Everything I wanted from the back-end, required a dependency on someone who was working on the back-end. This required some good communication (as described in the next section), as well as great planning. In previous projects I was more used to splitting tasks on features, instead of splitting tasks on architecture. The former means that you can do the whole process yourself: designing, developing the back-end, developing the front-end, and do full testing. Now every time I wanted to do a full test on certain features, I was dependent on the back-end. If something was wrong, we had to collaboratively go back to the design phase, implement front-end and then wait for the back-end to be re-implemented (or the other way around, depending who finished first), then test, and maybe do this multiple times. Especially in the beginning of the project, some of my shortcomings on communication and planning came forward due to this approach.

Communication In the beginning of the project, we discussed a lot of design choices with the whole project group, which was great. Then at the start of the implementation phase, I slowly found out that for most things we only discussed the general ideas. For some things we discussed the implementation in detail, but we did not write down a lot. This caused me to make quite some assumptions, that were not shared across all project members. Most of these assumptions were about very small details, but still costs a lot of time to change. Some examples: we have discussed how chat messages look like at a JSON structure, the back-end implemented it, and the front-end implemented it, and when both parties finished we started testing. Things where not working, because the back-end assumed camelCase for property names, and the front-end assumed snake_case for property names. Or I expected the list of courses from an endpoint to be an object index by key (because I was working with that idea for a week already), and the back-end served the list as an array (which makes a lot of sense if someone asks you just to 'return all courses'). Those things are not hard to fix, but it still takes quite some time to find out what exactly is the issue, solve it, and then test again.
If I would have written down the exact structure of data I was expecting from the back-end, other group members could have used that as a reference, and even could give their opinion on my thoughts. It would have improved the quality of some design choices, and would have saved quite a lot of time. The same holds for the endpoints, if I created a list of all endpoints needed, with the structure I'd like, we could discuss that and make sure everyone is one the same page. In the end we did these things, but we should have done them way earlier in the project.

Motivation During some parts of the project, I was having some motivational problems. They were largely caused by the current situation where we have to work from home all day, live in a lock-down, and I was have some personal issues as well. I found it really hard to set myself to work and to keep focused. I found out that when I was video calling while working, my focus was a lot better. Where we started of with three (online) project sessions a week, we ended with video calling everyday, sometimes even in the evenings and weekends if some tasks had to be finished. This greatly improved my focus, motivation, and productivity. It was always fun with the project group, and I'm really grateful that it helped me through this, at least for me, really difficult times.

Chapter 9

Conclusion

9.1 Project recap

The project in itself is an interesting project. During the whole duration of the project, we were able to communicate properly and achieved most of what was planned in the beginning. Deadlines, mostly the ones that were established by ourselves in regards to implementations and features, were also met successfully. Our initial aim for this project is to build something that we are proud of and is usable to the University of Twente. As parts of the system still need work, the system as it is delivered is not directly usable by the University of Twente. Since the end result does not meet all of our expectations, we are not mostly proud of the project. However, the process of the project itself is a valuable experience in which we learned a lot.

9.2 Delivered system

The system as it is delivered can be considered as a minimum viable product, with the notion that more extensive testing should be done to verify that the system is stable. Therefore, the system as delivered is not ready to be used at the University of Twente as it is, yet. Depending on the result of stability testing, reconsideration of some design choices (described in 8.1.2) might be necessary to come to a stable system. All features required for a standard exam scenario are present. The system provides a base to further extend on.

9.3 Future

Besides a few more rounds of bug-fixes, if we were to keep working on this project first-and-foremost would be to schedule and perform a full system test, since all opportunities to do so during development fell through. A lot of minor deign decisions during the project were made in haste, and given proper time and care

the database complexity and data formats of messages should be given another pass. Afterwards, the system by its current spec would have been finished, and work could start on additional features, such as screenshots & image transfer, private teacher chats and live typing.

Bibliography

Council, E. (2016). on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation). *Official Journal of the European Union*.

Marx, G. (1999). What's in a Name? Some Reflections on the Sociology of Anonymity. *Information Society*, *15*. https://doi.org/10.1080/019722499128565

Openid connect. (n.d.). Retrieved April 14, 2021, from https://openid.net/connect/

Virzi, R. (1992). Refining the Test Phase of Usability Evaluation: How Many Subjects Is Enough? *Human Factors: The Journal of the Human Factors and Ergonomics Society*, *34*. https://doi.org/10.1177/001872089203400407