Application description and ambition

Our ambition for this project, which we have dubbed Ice, was to create a complement to the Fire system, where students could register themselves to courses created by admins, then easily find others who are registered to the same courses and form groups with them. This forming of groups can either be done manually, with users creating groups and having others join them, or done in a matchmaking manner where users are presented with a list of all other ungrouped students in a given course, ranked by how closely matched they are in skill level in the given course. This skill level is determined by a self-assessment quiz posted by the course administrator.
We also had the ambition to allow administrators to post news/updates about the course and other practical features, but unfortunately we didn't have enough time to implement these use cases.

Application structure overview

The application is built in two main modules: a client side application and a server with a REST API for the client to make calls to.

The client is built to be as thin as possible, containing little logic besides what is necessary to display the information fetched from the server. The application is built using AngularJs for templating and control purposes. For handling the look of the application, we only used basic HTML and CSS. The application is built as a single page, with a sidebar which is used to navigate the site. The AngularJS ng-show directive along with environment variables are used to control which links are displayed to the user, depending on whether they are logged in or not and whether or not the user is an administrator. For example, if the user logs in with a student account, they will not be shown the link to the course creation/deletion page.
The index.html page contains the main ng-switch directive, which is what loads the different views, or partials, of the application. By setting the displayPartial environment variable to different values, the ng-switch directive will load different partials. Each partial has its own AngularJS controller, which makes the necessary calls to the server to fetch information to be displayed on the page and sets the variables of the partial to appropriate values.
The client application heavily relies on sessionStorage to pass information between different partials. For example, when loading the partial that displays a users profile, the application sets "desiredProfile" to the username of the user whose profile is to be displayed.

As mentioned in the previous paragraph, the server is where the logic of the application resides. The server is written using Node Express as the main framework, with several packages to assist. Communication with the database is done via Sequelize. As we had difficulty with making Sequelize actually generate tables, the tables and triggers of the database are written in separate sql-files from the Sequelize model descriptions.
Communication between the client application is achieved by Ajax calls. Naturally, a lot of routes in the servers API require authentication to be used. This is done by JWT-tokens, the payload of which contain the users username, access level and an expiration time. Every

time that the application sends a request to the server, this token is included and, if the authentication is successful, the server updates the expiration time of the token and returns a new token for the client application to use. In the client side application, this token is stored in session storage as “token”.
Initially, we planned to use the Passport module for authentication. However, partway into implementation of the server we realised that this module didn’t fit our needs very well. Passport is still used in the application, but this is mainly due to time constraints not allowing us to refactor the code.

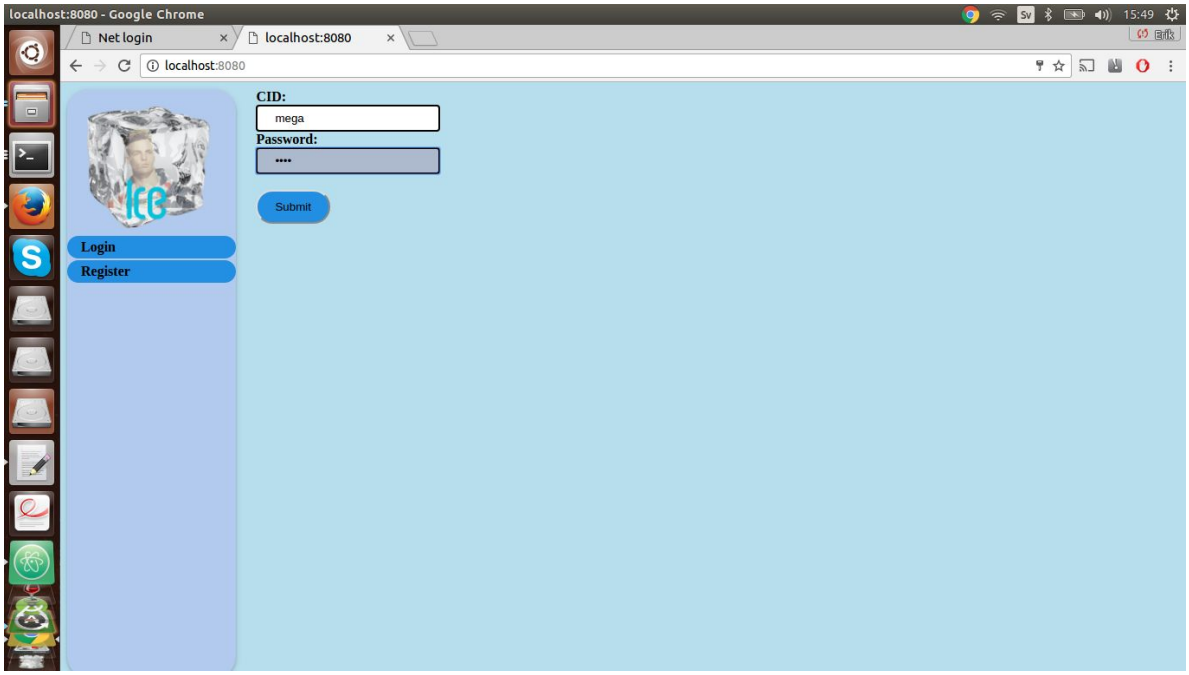
Application file structure:

- clientv2(dubbed such due to a previous failed attempt at writing a client page)
    - partials: Contains the html code for each partial, largely self-explanatory
        - adm-courses.html
        - course.html
        - courseReg.html
        - createCourse.html
        - default.html
        - groupJoin.html
        - groupManagement.html
        - login.html
        - profile.html
        - quiz.html
        - registerCourse.html
        - register.html
        - removeCourse
    - scripts: Contains the JavaScript code for every page
        - index.js: Implements the controller for each partial , as well as utility methods such as setDisplayPartial, which is used in several of the other script files. index.js also contains the logic for doing the initial loading of the application, which includes checking if the session already contains a token, as well as contacting the server to see if this token is still valid.
        - admCourses.js
        - constants.js
        - course.js
        - courseReg.js
        - createCourse.js
        - groupJoin.js
        - groupManagement.js:
        - login.js
        - profile.js
        - quiz.js
        - registerCourse.js
        - register.js

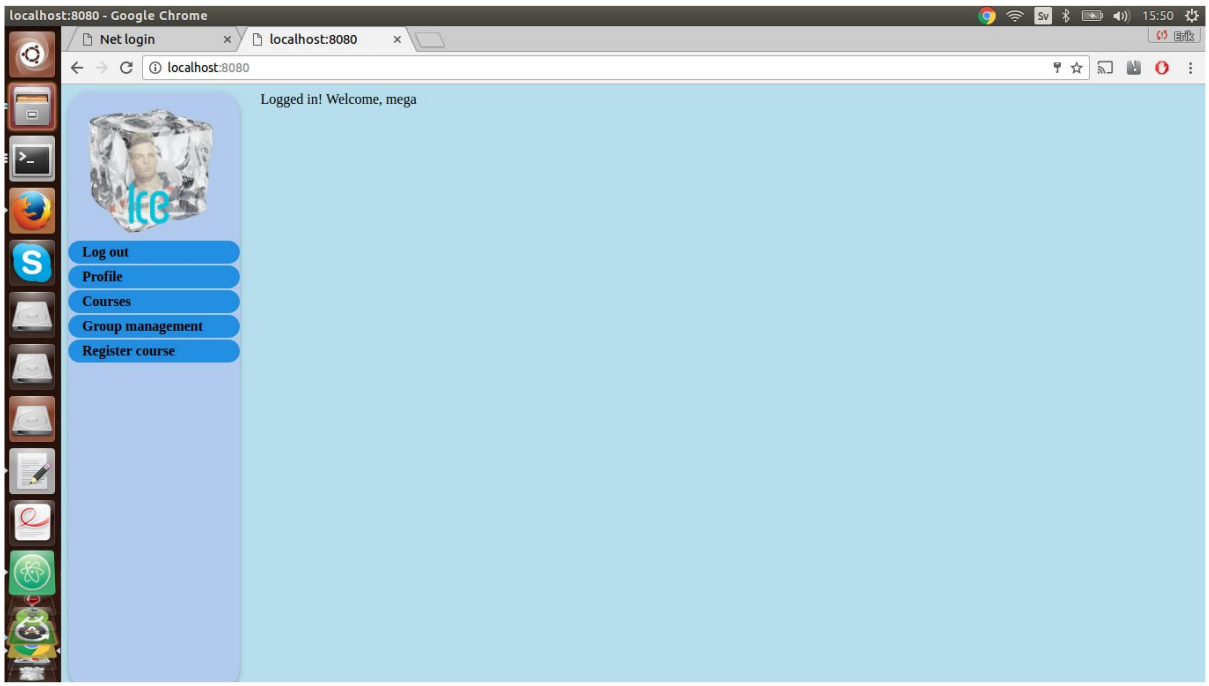
        - removeCourse.js
    - index.html: The main page of the application. Contains the sidebar, as well as the ng-switch directive for displaying every partial.
    - css
    - img: Contains the excellently glorious logo for Ice
- server
    - auth: Contains files for authenticating and registering users
        - authStrats: Contains logic for registering & logging in users
        - isLoggedIn: Contains logic for verifying & updating the expiration times of JWT tokens
    - db
        - ice_orm: Contains the definition of the Sequelize object used in the application
    - models: Contains model definitions for Sequelize. Largely self-explanatory.
    - resources
        - sql: This folder contains the sql-files for creating the databases tables, triggers and views. There is also a file for inputting some dummy data, though the users in this dummy data can not log in.
        - constants.js: Contains constant variables for the application. In this version, this consists simply of the encryption key for JWT tokens.
    - routes: Contains files defining routes for the application:
        - admin.js: Contains routes for managing administrative functions. Currently contains routes for creating new courses, removing existing ones and checking whether a given user is an administrator.
        - course.js: Contains routes for registering students to courses, fetching information about groups within the course, fetching information about students registered to a course that haven't joined a group within the course as well as of course fetching information about the course itself.
        - group.js:
        - quiz.js: Contains routes for managing the self-evaluation quizes for courses. Currently contains a route for posting the score of a given user on a given course and a route for getting the questions and the weight of each question for a quiz.
        - student.js: Contains routes for fetching profile information for students.
        - login.js: Contains a single route which uses auth/authStrats to verify the given credentials of a user, then generates and sends a JWT-token to the client if their credentials are succesfully authenticated.
        - register.js: Contains a single route for registering a new user.
        - user.js: Contains routes to fetch/manipulate information regarding a given user, such as fetching all groups that the user belongs to, or updating their personal information and password.
    - test

Use-case example: View profile of another user

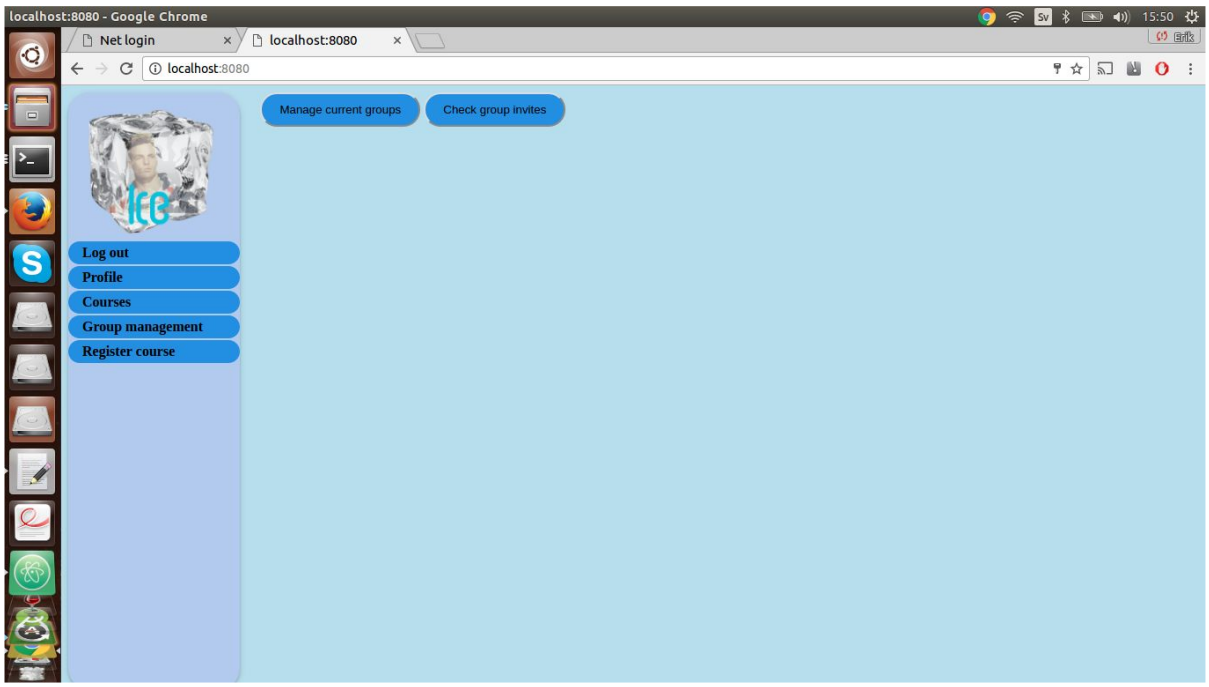
To view the profile of another user, the user first needs to log in to the application.



In the login view, the user writes in their username and password. This data is then sent to the server (in plaintext for the moment, unfortunately) as a POST request on the /login route and the client waits for a response. When the server receives a request on this route, it queries the database (via Sequelize) to see if there is a user with the given username. If there is, the server hashes the username and password together using bcrypt and compares this with the stored hash. If the hashes match up, the server authenticates the user, generates a JWT token and sends this back in the response.



Upon receiving a positive response, the client switches to the above view and stores the given token in session storage for later use. Now, since the user wishes to view the profile of one of their group members, the user clicks on the group management link. This simply changes the values of the displayPartial environment variable to display the below view.

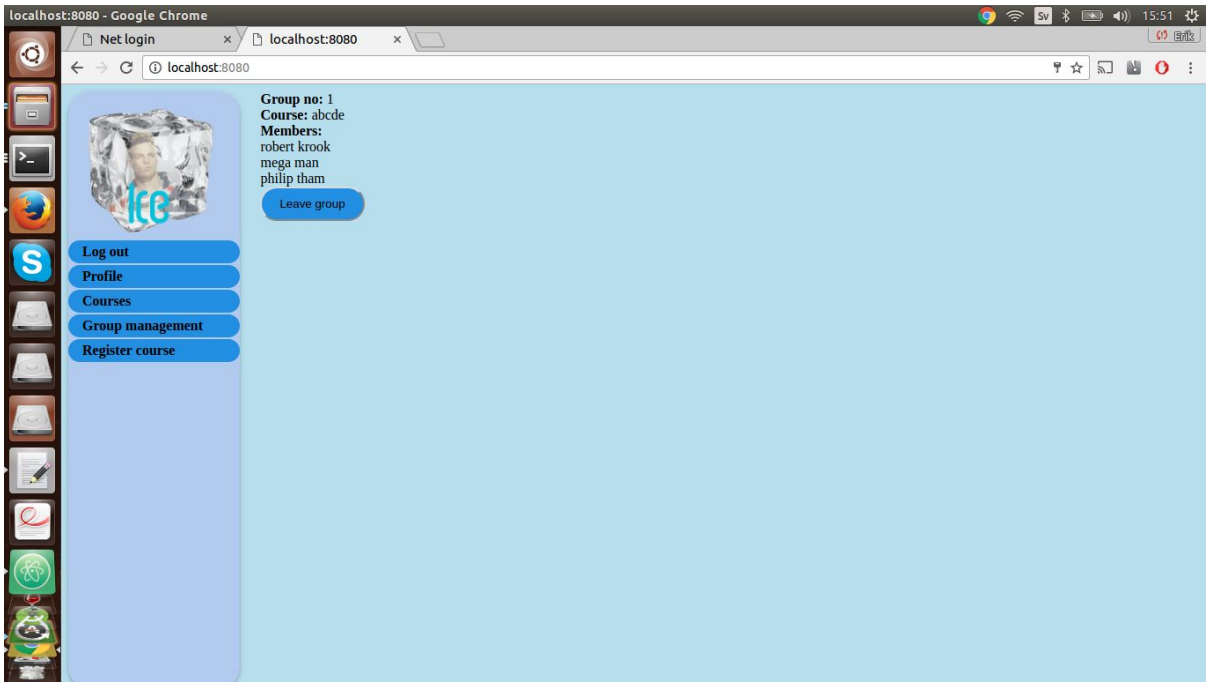


Here, the user clicks on the “Manage current groups” button to see a list of all groups that they are currently a member of. This will send a GET request to the server on the user/groups/ route. The application will include the previously gained token in the request under the “Authorization” header.

When the server receives this request, the first thing it does is decode the given token to see if

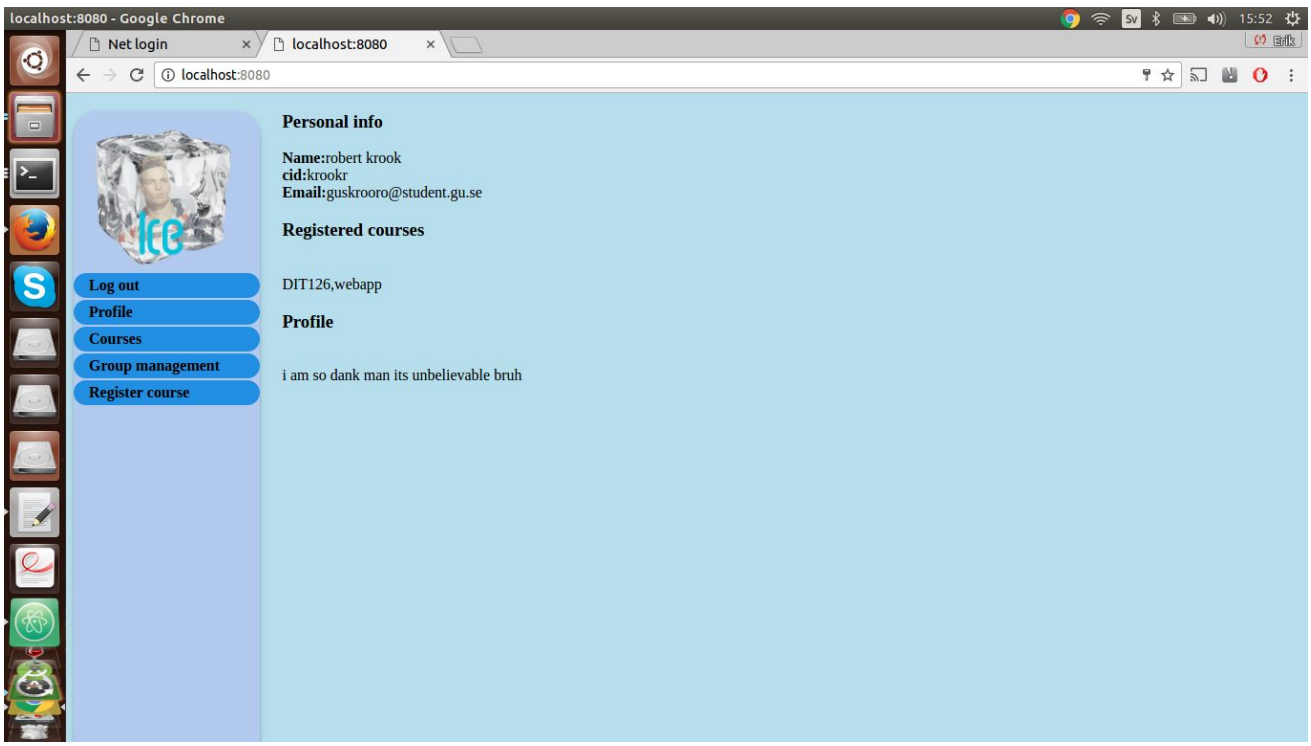
1) It is a valid token
2) It has not expired

If both of the above are true, the server goes on to process the request. Otherwise, errors are sent in a response to the client. When processing the request, the server will decode the given JWT token to get the username of the user sending the request. This is then used to query the Groups table, to find the group number and course of every group that the user is a member of. After this, the GroupMembers view is queried to fetch information on the members of every group that the user belongs to. After compiling this information, the server sends it to the client in its response.



Once the client receives information on all groups, it will display the group number, course and members of each group, as well as a button that allows the user to leave each group. The user may then click on the name of any group member to display their profile. When this is done, the application will write the username of the desired user to “desiredProfile” in session storage, then load the profile partial.

When loading the profile partial, the AngularJS controller of this partial will run. This controller will make several calls the server to fetch information about the users personal info, as well as which courses the user is registered to, then display this information as shown below.



Testing
Tests have been written to verify that the rest-API answers in an appropriate way to requests.

The framework Mocha has been used to define these tests. Mocha allows for writing tests that are very easy to follow and understand. Someone who didn't write the tests can follow what's happening and what's being tested very easily.

Each test performs a HTTP-request on the server. The response is analyzed and compared with expected values.

If you wish to run the tests, enter the /server/test/ subdirectory and run 'npm test' to run all tests, or 'mocha -filename' to run tests from a single file.
In order for the tests to work properly, a user has to run initiate_database.sql (found in /server/resources/sql subdirectory) on the mysql-server. This file sets up the database in a 'clean' state. The need for this action arises because the tests try to create users, and these are statically written in the tests. It will try to create users that already exists (if the testsuite has been run before) and fail a test that would otherwise pass.