CS2102 Database Systems
Project Report

Course Management System - *CosMaS*

National University of Singapore, School of Computing

November 9, 2019

Group 50

	Student Number	Responsibilities
Na Nazhou	A0164627Y	Users, Semesters, Modules and forum accesses, UI
Liu Guangyuan	A0187558J	Forums, threads, replies, logging, permissions
Liu Zimu	A0188044B	Courses, course requests and course memberships
Wang Xueting	A0187429N	Group and group memberships

GitHub repo: https://github.com/Na-Nazhou/cs2102-project

Table of Contents

1. Introduction

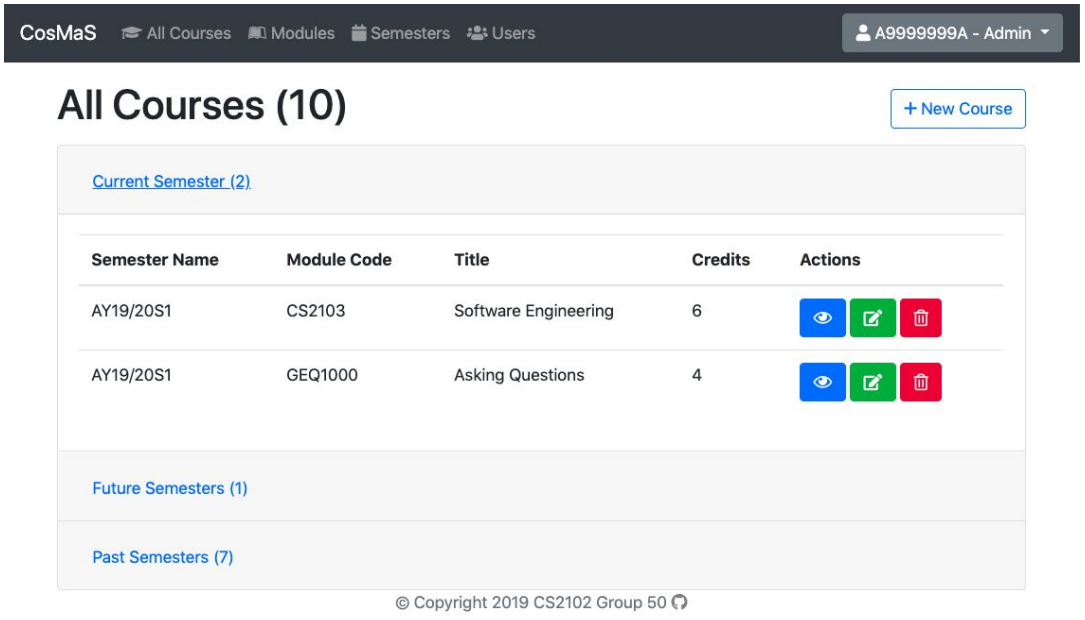
1.1. Requirements

CosMaS is a course management system that allows **professors** to manage a **course** that he / she is **teaching**. **Students** may **request** to enter a course. Professors may categorize students in the course into **groups** such as the different tutorial groups. Professors may create **forums** where only certain groups or collection of groups in the course can read or write. However, the professor may read, write, or delete **threads** and **replies** in the forum. Additionally, professors may add **teaching assistants** to manage groups. Teaching assistants cannot create groups or forums but can read, write, or delete entries in the forum. Each **user** must have an account.

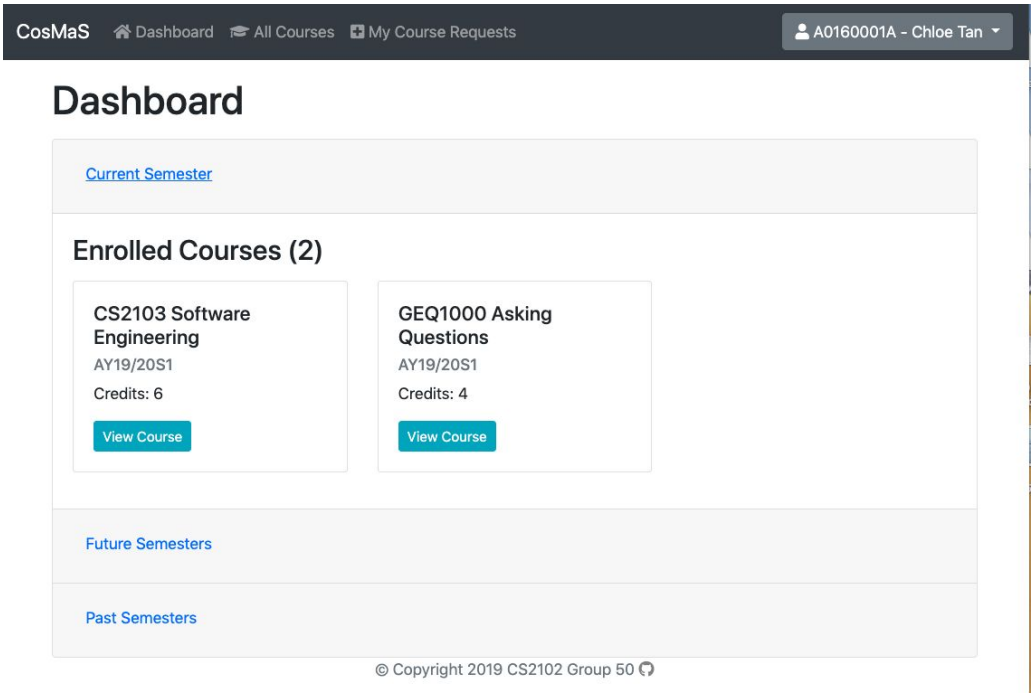
1.2. Functionalities

1.2.1. User management

In *CosMaS*, generally there are two types of user, which are **admin user** and **normal user**. Admin user accounts can only be created through directly adding to the database. Admin users can access more pages such as the list of users, semesters, modules compared to normal users. A normal user account can be easily set up with a 9-character user ID (following the standard of NUS matriculation number) and a password. The raw password will be encrypted and only the encrypted password will then be safely saved into our database. After logging in, admin users will be redirected to the following page which consists of a list of all the courses which are grouped according to the semesters:



However, for normal users, they will be redirected to their dashboard which contains their own information such as the courses they have been enrolled, teaching or tutoring:

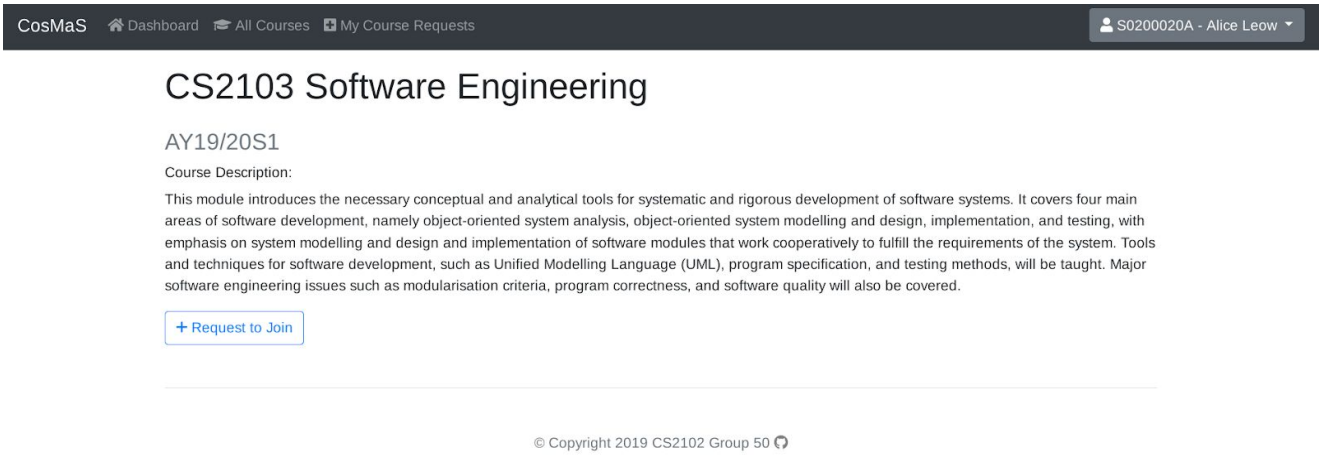


1.2.2. Semesters, modules, and courses

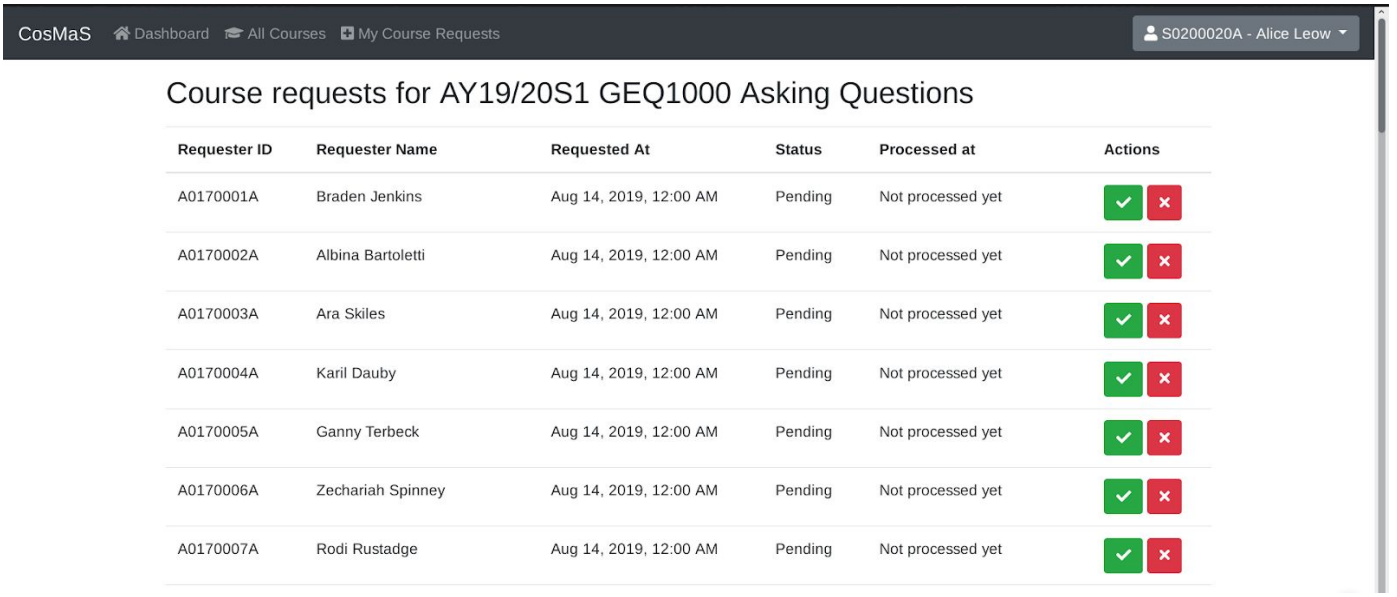
Before the start of each semester, the admin has to set up the new **semester** and specify the start time and end time of the semester. The admin is also in charge of managing **modules** and **courses** and only the admin has the right to create, update, destroy these entities. The difference between module and course is that a module can be offered in each semester as a course. For example, there is only one module CS2102, but it can be offered in multiple semesters as independent courses.

1.2.3. Course and group memberships

Each user in *CosMaS* can be enrolled in any number of existing courses. A user can apply for a **course request** as a student to enter a course that is offered in the current semester or will be offered in future semesters by clicking on the `request to join` button on the course page.

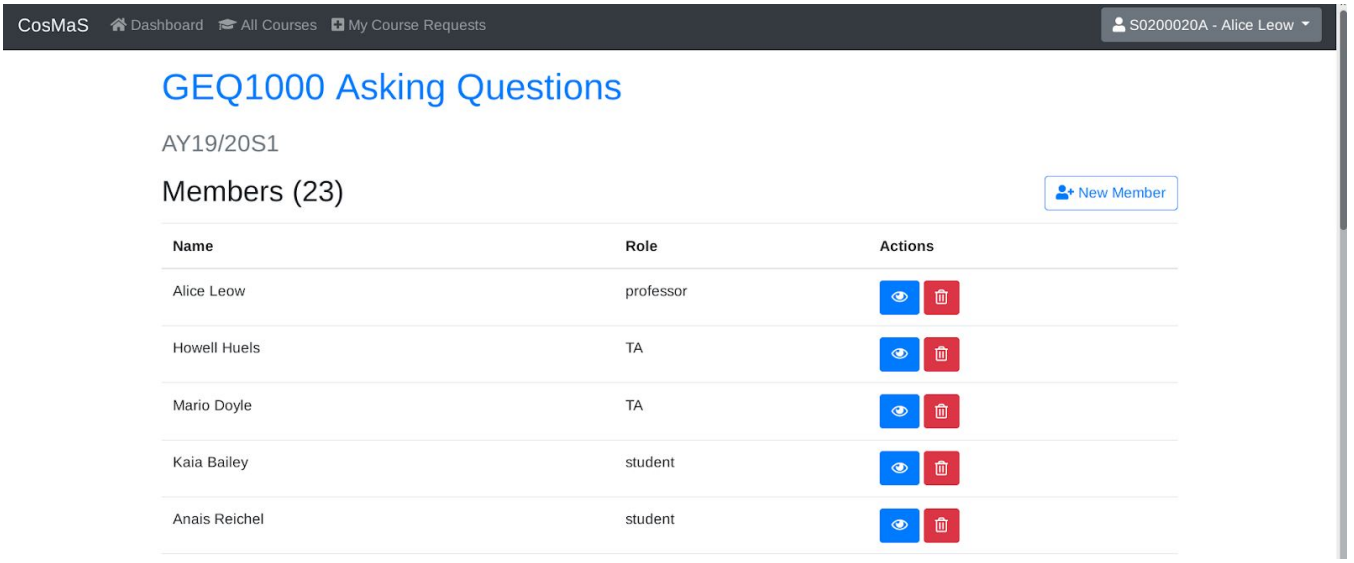


Admins or professors of that course will then be able to approve or reject these requests on the following page. Upon approval, the requester will be automatically added to the course as a student.

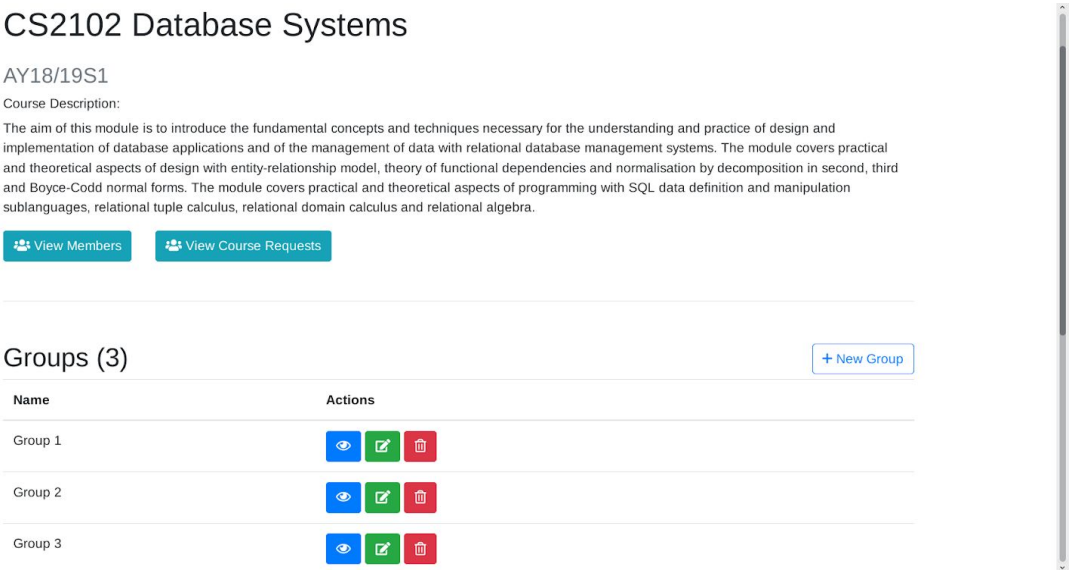


As for professors, only admin can assign a user as the professor of a course. The assigned professor can then assign other users as teaching assistants of his/her course. These teaching assistants can also be further categorized into different groups.

A list of all members involved in a course will be shown in the **course membership** page. All professors, teaching assistants and students will be displayed in this page. Admin is able to view a member's dashboard which contains all the relevant course information of that member by clicking the `view` button. Admins can also delete any member from this course if necessary. Professor can only delete teaching assistants or students from this course i.e. they cannot delete themselves or other professors in the course.

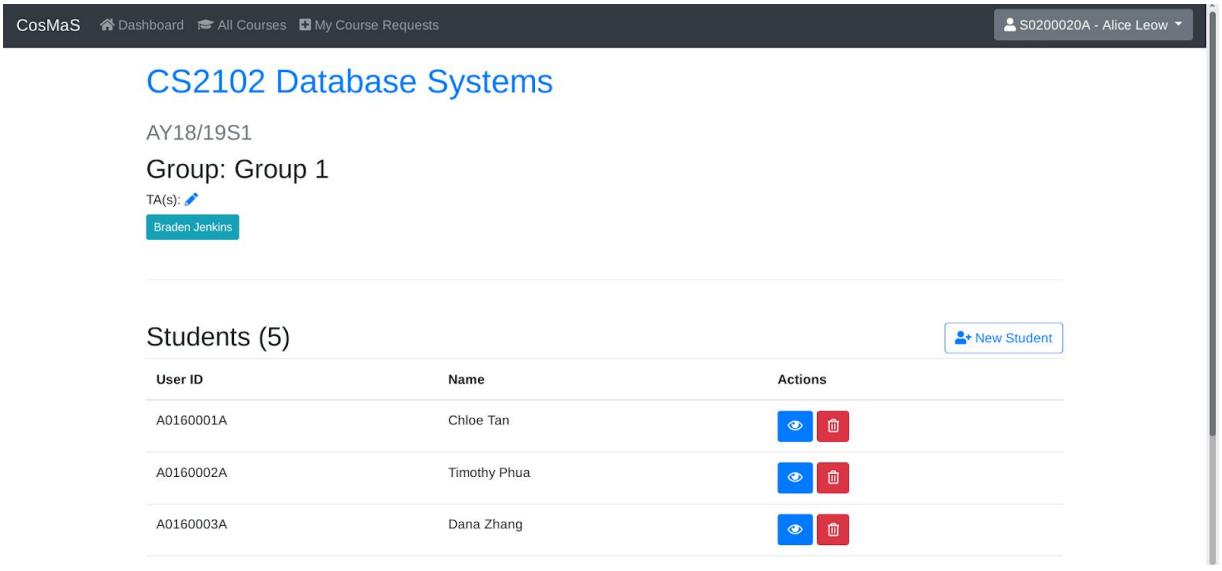


Each course can have zero or more groups for students. A list of existing groups within each course is shown on each individual course page. Admin or professor can add a new group to this course with a specified group name. They can also view, edit or delete the group. The following diagram shows the course page which consists of a list of groups within the course CS2102 AY18/19S1.



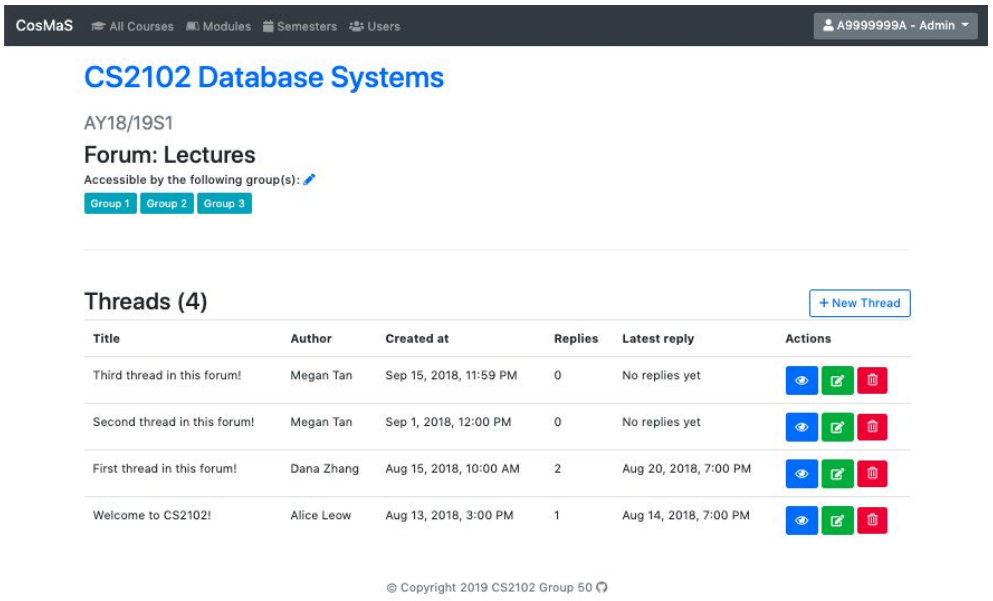
Group memberships are made up of teaching assistants and students from this course. A list of teaching assistants and students in each group will be shown in the individual group page. Teaching assistants are able to view all the groups in their courses, and students in a group are able to view all other members in the group. Admins or professors can add or remove teaching assistants to or from a particular group, and add students enrolled in this course into a particular group. They can also delete a student from the group.

Admins can view all the student's and teaching assistants' dashboard from a group, while a student can only view his or her own dashboard.



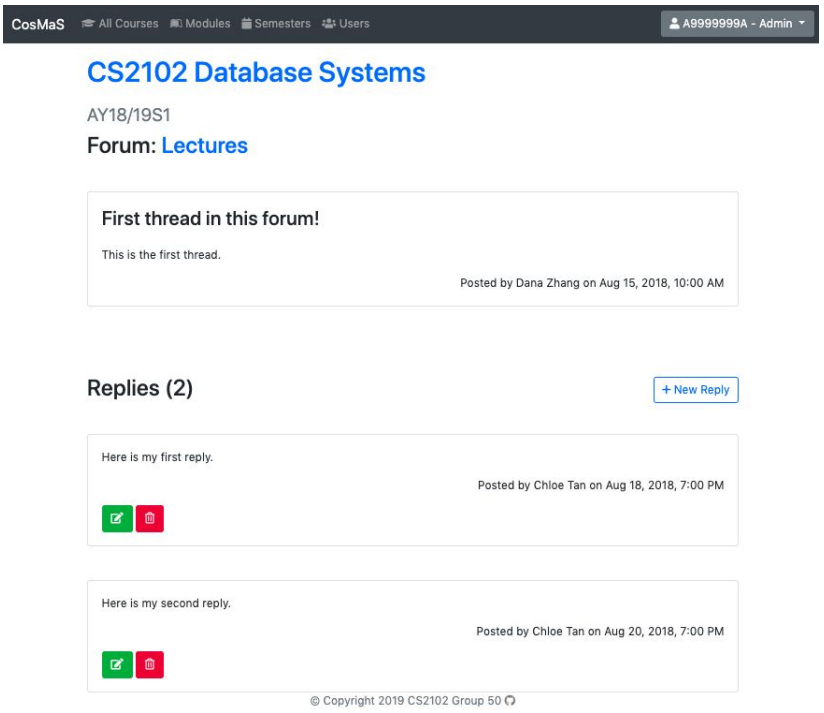
1.2.4. Forums, threads, and replies

Each course in *CosMas* can have zero or more **forums**. Forums are platforms for discussion and exchange among students. Forums can also serve as a platform for students to communicate with the course instructors (professors and teaching assistants).
A forum consists of multiple **threads**. Each thread is usually for a particular topic. For example, forum named “Tutorials” can have a thread named “Need help with q2 of tutorial 3”. Forum members can respond to a thread by posting **replies**.



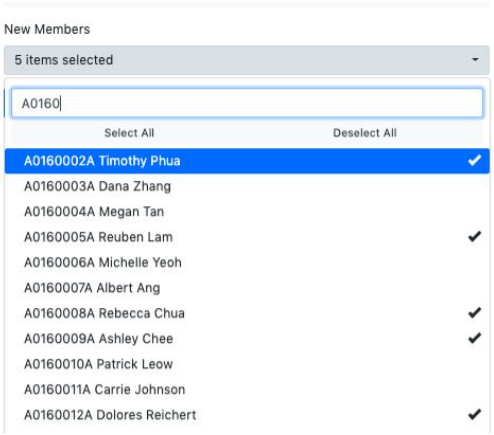
Only professors can create forums. The professor him/herself has full access to write, read, or delete threads and replies in the forum. Access to forums for other members of the course are granted on a group basis. Only groups granted access by the professor can read posts in a particular forum and create new entries in it. Any number of groups can be granted access to any number of forums.

The professors can also assign teaching assistants to help him/her manage the forum. Teaching assistants also have full read, write, and delete access to the threads and replies.



1.3. Interesting Features and Functionalities

The most interesting and important aspect of CosMaS is that it supports a solid and complex permission management system. Users with different permissions are only allowed to access the authorised pages. They might also see different content even on the same page. Since the concept of role is associated with the course and group, the same user can have different access levels in different courses or groups depending on the role he has. This makes the system really flexible as compared to how traditional permission architecture manages roles by directly tying the role to the user. The permission controls are implemented at both the frontend and the backend. This means



Another user-friendly feature of our application is that we support batch insert and update for adding users to a group or course. We understand that adding one user at a time can be time consuming especially when you have a large amount of users to process. Batch insert and update is powered through PostgreSQL functions. As for the front-end, we also makes use of BootStrap Select to support live search so that users do not need to scroll over a long list of users to locate the correct user to add.

```
CREATE OR REPLACE FUNCTION add_course_members(
    character varying[],
    course_memberships.role%TYPE,
    semesters.name%TYPE,
```

```
    modules.module_code%TYPE)
    RETURNS void AS $$
    BEGIN
        FOR i in 1 .. COALESCE(array_length($1, 1), 0)
        LOOP
            INSERT INTO course_memberships (user_id, role,
semester_name, module_code) VALUES($1[i], $2, $3, $4);
        END LOOP;
    END
    $$ LANGUAGE plpgsql;
```

1.4. Interesting Queries

1.4.1. Computes all the requests to a course. The result will include the requester's name as `requester_name`, the status of the request as one of 'pending', 'approved', or 'rejected'. The results are computed by joining the `course_requests` table and the `users` table.

```
SELECT CR.semester_name, CR.module_code, CR.requester_id,
CR.requested_at, CR.is_approved, CR.closed_at,
U.name AS requester_name,
COALESCE (
    CASE WHEN CR.closed_at IS NULL THEN 'Pending' ELSE NULL END,
    CASE WHEN is_approved THEN 'Approved' ELSE 'Rejected' END)
AS status
FROM course_requests CR
    LEFT JOIN users U
    ON CR.requester_id = U.id
    WHERE semester_name=$1 AND module_code=$2
    ORDER BY closed_at DESC, requested_at ASC
```

1.4.2. Computes the information of all the threads in a forum. Besides the information present in the `threads` table, it also includes the following:

- Author name, which is obtained by performing a left join of the threads table with the `users` table
- Replies count, which is obtained by performing an inner join with the `replies` table and grouping by the attributes in the primary key of `threads`.
- Latest reply time, which is also obtained similarly to replies count, except `MAX` aggregate function is used on the `created_at` column of `replies`.

The results are sorted by their **last activity** time. For threads without replies, their activity time is their time they are created. For threads with replies, their activity time is the time their last reply was posted.

```
SELECT t.semester_name AS semester_name, t.module_code AS
module_code,
```

```
    t.forum_title AS forum_title, u.name AS author_name,
    t.created_at AS created_at, t.title AS title, t.content AS
content,
    COUNT(r.created_at) AS replies_count,
    MAX(r.created_at) AS latest_reply_time
FROM threads t
LEFT JOIN users u
    ON t.author_id=u.id
LEFT JOIN replies r
    ON t.semester_name=r.semester_name AND
t.module_code=r.module_code
        AND t.forum_title=r.forum_title AND
t.created_at=r.thread_created_at
    WHERE t.semester_name=$1 AND t.module_code=$2 AND
t.forum_title=$3
    GROUP BY(t.semester_name, t.module_code, t.forum_title,
t.created_at, u.name)
    ORDER BY GREATEST(MAX(r.created_at), t.created_at) DESC
```

1.4.3. Find the 5 most diligent students of the current semester. Student A is considered to be more diligent than student B if one of the following conditions are met:

- A has higher total credits from courses taken in current semester than B.
- A and B have the same total credits from courses taken in current semester, and A takes more courses in current semester than B.
- A and B have the same total credits from courses taken in current semester and takes the same number of course in the current semester, and ID of A is alphabetically smaller than that of B.

The result contains the ID, name, number of courses taken and total credits of the 5 most diligent students, sorted by total credits, followed by number of courses, followed by ID. Students who do not take any course in current semester is also included in the result for completeness of the query.

```
WITH StudentInfo AS (
   SELECT CM.user_id id, COUNT(*) numCourses,
       SUM(C.credits) totalCredits
   FROM course_memberships CM, courses C
   WHERE CM.semester_name = C.semester_name
       AND CM.module_code = C.module_cod AND C.semester_name = $1
       AND CM.role='student'
   GROUP BY CM.user_id
   UNION
   SELECT id, 0, 0 FROM users
   WHERE id NOT IN
       (SELECT user_id FROM course_memberships WHERE
role='student'))
SELECT U.id, U.name, numCourses num_courses, totalCredits
total_credits
```

```
FROM StudentInfo S, Users U
WHERE S.id = U.id
ORDER BY numCourses DESC, totalCredits DESC, id
LIMIT 5
```

1.5. Constraints

All the constraints are broken down into following components for greater readability. Some of the constraints are captured by our ER diagram, which some have to be captured by triggers.

1.5.1. Semesters - Courses - Modules

- A module can be offered in zero or many semesters as different “courses”.
- A semester can offer zero or many courses.
- A module can be offered at most once in the same semester.

1.5.2. Users - Course_Memberships - Courses

- A user can be a member of zero or many courses.
- A course can have zero or many users as members.
- Within the same course, a user can have exactly one role of either professor, teaching assistant or student. To restrict the roles that user can take, an enum is used to represent the three roles.
- Admin users cannot be in any course.

1.5.3. Users - Course_Requests - Courses

- A user can request to enter zero or many courses.
- A course can be requested by zero or many users.
- A user cannot send more than one request for the same course.
- A user cannot request to enter a course offered in a passed semester. This constraint is captured by a trigger on course_requests table.
- A user cannot request to enter a course if he is already a member of that course. This constraint is captured by a trigger on course_requests table.
- Admin users cannot request to enter any course.

1.5.4. Groups - Within - Courses (Groups is a weak entity set of Courses)

- A course can have zero or many groups.
- Each group belongs to exactly one course.

1.5.5. Users - Group_Memberships - Groups

- A group can have zero or many users as members.
- A user can be a member of zero or many groups, with the following restrictions:
  - A user’s must have a role in each group he/she belongs to. (captured by NOT NULL table constraint)
  - A user’s role in this group must be either TA, or student.
  - If a user u has role r in group g and g belongs to course c, then u must have a course membership in course c with the same role r. This is captured by a trigger on Group_Memberships.

    - Professor cannot be assigned to groups. (implied by the previous two constraints)
    - A user cannot have different roles in the same group. (captured by the PK of Group_Memberships)

1.5.6. Forums - Within - Courses (Forums is a weak entity set of Courses)

- A course can have zero or many forums.
- Each forum belongs to exactly one course.

1.5.7. Groups - Accesses - Forums

- A group can access zero or many forums.
- A forum can be accessed by zero or many groups.
- A group can only be granted access to a forum if both the group and the forum belongs to the same course.

1.5.8. Forums - Contains - Threads (Threads is a weak entity of Forums)

- A forum can have zero or many threads.
- Each thread belongs to exactly one forum.

1.5.9. Replies - Under - Threads (Replies is a weak entity of Threads)

- A thread can have zero or many replies.
- Each reply belongs to exactly one thread.

1.5.10. Users - Authors - Threads, Users - Posts - Replies

- A user can author zero or many threads.
- A thread is authored by exactly one user.

- A user can post zero or many replies.
- A reply is posted by exactly one user.

- When a user posts a thread/reply, he/she must be authorised to write to the forum in the access table and this is captured using a trigger on Threads and Replies table
    - The user must be a member of the course that the forum belongs to
    - If the user is a student in the course, check further if any groups he belongs to have access to the forum
- Additionally, the foreign key to Users' table is nullable. This accounts for the situation where the author's account is deleted.

2. ER model

The following is our ER diagram which captures the majority of the constraints mentioned in Section 1.5. Our ER diagram can also be broken down into smaller components in a similar way we did in section 1.5. There is also another copy of the ER diagram in our source code.

3. Relational Schema

```
CREATE TABLE Users (
    id              CHAR(9) PRIMARY KEY,
    name            VARCHAR(255) NOT NULL,
    is_admin        BOOLEAN NOT NULL DEFAULT FALSE,
    password_digest VARCHAR(255) NOT NULL
);

CREATE TABLE Semesters (
    name            VARCHAR(50) PRIMARY KEY,
    start_time      TIMESTAMP NOT NULL,
    end_time        TIMESTAMP NOT NULL,
    CHECK(end_time > start_time)
);

CREATE TABLE Modules (
    module_code VARCHAR(10) PRIMARY KEY
);

CREATE TABLE Courses (
    semester_name   VARCHAR(50) REFERENCES Semesters(name),
    module_code     VARCHAR(10) REFERENCES Modules(module_code),
    title           VARCHAR(100) NOT NULL,
    description     TEXT NOT NULL,
    credits         INTEGER NOT NULL,
    PRIMARY KEY(semester_name, module_code)
);

CREATE TABLE Course_Requests (
    requester_id    CHAR(9) REFERENCES Users(id) ON DELETE CASCADE NOT
NULL,
    requested_at    TIMESTAMP NOT NULL,
    semester_name   VARCHAR(50),
    module_code     VARCHAR(10),
    is_approved     BOOLEAN NOT NULL DEFAULT FALSE,
    closed_at       TIMESTAMP DEFAULT NULL,
    FOREIGN KEY(semester_name, module_code) REFERENCES Courses ON DELETE
CASCADE,
    PRIMARY KEY(semester_name, module_code, requester_id)
);

CREATE TYPE COURSE_ROLE AS ENUM ('student', 'TA', 'professor');
CREATE TABLE Course_Memberships (
    role            COURSE_ROLE NOT NULL DEFAULT 'student',
    semester_name   VARCHAR(50),
    module_code     VARCHAR(10),
    user_id         CHAR(9) REFERENCES Users(id) ON DELETE CASCADE NOT
NULL,
```

```
    FOREIGN KEY(semester_name, module_code) REFERENCES Courses ON DELETE
CASCADE,
    PRIMARY KEY(semester_name, module_code, user_id)
);

CREATE TABLE Groups (
    name            VARCHAR(100),
    semester_name   VARCHAR(50),
    module_code     VARCHAR(10),
    FOREIGN KEY(semester_name, module_code) REFERENCES Courses ON DELETE
CASCADE,
    PRIMARY KEY(semester_name, module_code, name)
);

CREATE TYPE GROUP_ROLE AS ENUM ('student', 'TA');
CREATE TABLE Group_Memberships (
    role            GROUP_ROLE NOT NULL DEFAULT 'student',
    semester_name   VARCHAR(50),
    module_code     VARCHAR(10),
    group_name      VARCHAR(100),
    user_id         CHAR(9) REFERENCES Users(id) ON DELETE CASCADE NOT
NULL,
    FOREIGN KEY(semester_name, module_code, group_name) REFERENCES Groups
ON DELETE CASCADE,
    PRIMARY KEY(semester_name, module_code, group_name, user_id)
);

CREATE TABLE Forums (
    title           VARCHAR(255),
    semester_name   VARCHAR(50),
    module_code     VARCHAR(10),
    FOREIGN KEY(semester_name, module_code) REFERENCES Courses ON DELETE
CASCADE,
    PRIMARY KEY(semester_name, module_code, title)
);

CREATE TABLE Accesses (
    semester_name     VARCHAR(50),
    module_code       VARCHAR(10),
    group_name        VARCHAR(100),
    forum_title       VARCHAR(255),
    FOREIGN KEY(semester_name, module_code, group_name) REFERENCES Groups
ON UPDATE CASCADE ON DELETE CASCADE,
    FOREIGN KEY(semester_name, module_code, forum_title) REFERENCES Forums
ON UPDATE CASCADE ON DELETE CASCADE,
    PRIMARY KEY(semester_name, module_code, group_name, forum_title)
);

CREATE TABLE Threads (
    created_at      TIMESTAMP,
    title           VARCHAR(255) NOT NULL,
```

```
    content         TEXT NOT NULL,
    author_id       CHAR(9) REFERENCES Users(id) ON DELETE SET NULL,
    semester_name   VARCHAR(50),
    module_code     VARCHAR(10),
    forum_title     VARCHAR(255),
    FOREIGN KEY(semester_name, module_code, forum_title) REFERENCES Forums
ON DELETE CASCADE,
    PRIMARY KEY(semester_name, module_code, forum_title, created_at)
);

CREATE TABLE Replies (
    created_at        TIMESTAMP,
    content           TEXT NOT NULL,
    author_id         CHAR(9) REFERENCES Users(id) ON DELETE SET NULL,
    semester_name     VARCHAR(50),
    module_code       VARCHAR(10),
    forum_title       VARCHAR(255),
    thread_created_at TIMESTAMP,
    FOREIGN KEY(semester_name, module_code, forum_title, thread_created_at)
REFERENCES Threads ON DELETE CASCADE,
    PRIMARY KEY(semester_name, module_code, forum_title, thread_created_at,
created_at),
    CHECK (thread_created_at < created_at)
);
```

All database tables are in **BCNF** form since left-hand side of all non-trivial functional dependencies in the relation schema are superkeys. All functional dependencies in the minimal cover is enforced using primary key constraints in the database. There are no additional triggers to violate this property of the schema. Hence, No attributes are uniquely identified by any set of attributes that are not superkeys.

4. Non-trivial Constraints

4.1. Semesters cannot overlap in time. i.e. The following is not allowed: semester A's start date is earlier than semester B's end date and semester A's end date is later than semester B's start date.

```
CREATE OR REPLACE FUNCTION validate_semester() RETURNS TRIGGER AS $$
    BEGIN
    IF TG_OP = 'INSERT' THEN
        IF EXISTS (
            SELECT 1
            FROM semesters
            WHERE start_time <= NEW.end_time
                AND end_time >= NEW.start_time) THEN
            RAISE EXCEPTION 'Semesters cannot have overlapped time
intervals';
        END IF;
    END IF;
```

```
    IF TG_OP = 'UPDATE' THEN
        IF EXISTS (
            SELECT 1
            FROM semesters
            WHERE start_time != OLD.start_time
                AND end_time != OLD.end_time
                AND start_time <= NEW.end_time
                AND end_time >= NEW.start_time) THEN
            RAISE EXCEPTION 'Semesters cannot have overlapped time
intervals';
        END IF;
    END IF;

    RETURN NEW;
    END;
$$ LANGUAGE plpgsql;

DROP TRIGGER IF EXISTS validate_semester ON semesters;
CREATE TRIGGER validate_semester
BEFORE INSERT OR UPDATE ON semesters
FOR EACH ROW EXECUTE PROCEDURE
validate_semester();
```

4.2. A user cannot request to join courses in the past semesters.

```
CREATE OR REPLACE FUNCTION check_request_to_past_courses()
RETURNS TRIGGER AS $$
BEGIN
    IF TG_OP = 'INSERT' OR TG_OP = 'UPDATE' AND OLD.is_approved = false AND
NEW.is_approved = true THEN
        IF LOCALTIMESTAMP(0) > (SELECT end_time FROM semesters WHERE
name=NEW.semester_name) THEN
            RAISE EXCEPTION 'Requesting or approving request to join past
courses';
        END IF;
    END IF;
    RETURN NEW;
END; $$
LANGUAGE plpgsql;

DROP TRIGGER IF EXISTS reject_requests_to_past_courses ON course_requests;
CREATE TRIGGER reject_requests_to_past_courses
BEFORE INSERT OR UPDATE ON course_requests
FOR EACH ROW EXECUTE PROCEDURE
check_request_to_past_courses();
```

4.3. A user cannot request to join courses that he/she is in, either as a student, teaching assistant or professor. A user cannot submit multiple requests to the same course.

```
CREATE OR REPLACE FUNCTION check_eligibility_to_request()
```

```
RETURNS TRIGGER AS $$
BEGIN
    IF TG_OP = 'UPDATE' AND OLD.requester_id=NEW.requester_id AND
OLD.semester_name=NEW.semester_name AND OLD.module_code=NEW.module_code
THEN
        RETURN NEW;
    END IF;

    IF EXISTS (
        SELECT 1 FROM course_requests
        WHERE semester_name=NEW.semester_name
        AND module_code=NEW.module_code
        AND requester_id=NEW.requester_id
    ) THEN
        RAISE EXCEPTION 'Cannot have duplicate requests to the same
course';
    END IF;

    IF EXISTS (
        SELECT 1 FROM course_memberships
        WHERE semester_name=NEW.semester_name
        AND module_code=NEW.module_code
        AND user_id=NEW.requester_id
    ) THEN
        RAISE EXCEPTION 'Cannot request to join a course that the user is
already in';
    END IF;
    RETURN NEW;
END; $$
LANGUAGE plpgsql;

DROP TRIGGER IF EXISTS reject_illegible_requests ON course_requests;
CREATE TRIGGER reject_illegible_requests
BEFORE INSERT ON course_requests
FOR EACH ROW EXECUTE PROCEDURE
check_eligibility_to_request();
```

5. Software tools/Frameworks

Below is a list of tools and packages that we used in our CosMaS:

- Node.js, express
- PostgresQL and node-postgres
- dotenv
- Date-fns
- passport, bcrypt
- method-override
- ejs, ejs-locals, connect-flash
- Bootstrap 4, bootstrap-select
- npm, nodemon, eslint, prettier

6. Difficulties and lessons learned

Throughout the team project, we have encountered several technical difficulties.

JavaScript is not a nice language to work with. The typing system is extremely fragile and its async-ness often brings bugs that are very hard to find for those who are unfamiliar with the language. However, the tight timeline means it is also impossible to use TypeScript to avoid some of the problems of JavaScript.
For our team, 97% of the time was spent coding the web backend and frontend instead of on database and design. This is also due to the lack of web development experience among our team. Although the guide files are helpful, we still spent a lot of time figuring out additional tools and technologies such as AJAX, asynchronous functions, etc.

While all access control can be implemented at the backend, it is hard to bring the same level of strictness to the frontend. Passing the access control information from the backend to the frontend either complicates the database queries a lot, or brings about N+1 problems.

Using weak entity sets and not using serial IDs make the tables store too much redundant data. They also make it very difficult to change the database schema design halfway into the development cycle. However, from here we also learned that carefully designing the database schema before delve into coding is extremely important.

Beside all these challenges, we are still very proud that we have completed our product, and here are some key takeaways from this project:

Having a solid database design is really crucial in developing any system. It builds the foundation of the system. It really affects the later stage of the development a lot.

Moreover, we need to plan ahead with better time and project management. It would be better if we are able to familiarize with the necessary technical aspects as much as possible before doing the project itself. This would help to improve the efficiency to a great extent.