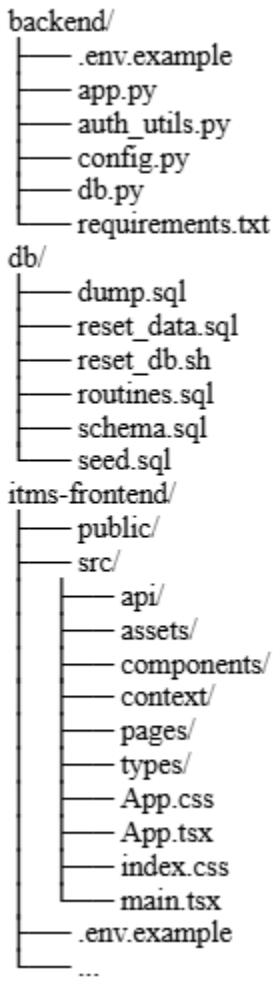
ITMS - Issue Tracking Management System

Danyal Ahmed

1) Project Creation - Running it on your own machine

The root directory should roughly match this structure:



The backend/ directory will contain all necessary files to run the API layer.

The db/ directory should contain sql scripts, including `dump.sql`, which will create a starting point for testing.

The itms-frontend/ directory contains the React application for the frontend, with build and testing handled by Vite.

Basic requirements

Please ensure you have the following installed prior to attempting to run this project:

- MySQL Server version 8.0 or later
- MySQL Workbench (not necessary but very helpful)
- Python 3.9 or later
- Node.js (LTS preferred and npm (npm will be bundled with Node)
- A modern web browser (anything that supports the fetch API, ES modules, and cookie-based sessions)
- Access to localhost on your machine

Database Setup

Ensure you have MySQL Server running. The easiest way to set up the database is to run the `dump.sql` file found in the db/ directory. This should create a database called ‘itms’ with 8 tables, as well as the associated triggers, constraints, stored procedures, and basic seed data to get started.

Should the dump.sql script not work for some reason, you can also run the following scripts in order:

- schema.sql
- routines.sql
- seed.sql

If at any point you would like to reset the database, you may run the `reset_db.sh` script in the db/ directory, which assumes you have the `mysql` shell command installed and added to your PATH.

API Layer Configuration

Unless you never use Python, you will likely want to create a virtual environment from which to run the application. **If you do not want to use a virtual environment, or have another preferred method (such as Conda), you can skip the venv setup.**

Run the following commands to create a virtual environment:

```
cd backend
python -m venv myenv
```

Then activate your virtual environment:

macOS/Linux:

```
source myenv/bin/activate
```

Windows:

```
myenv/Scripts/activate
```

If you're using Git Bash on Windows, you'll want to follow the macOS/Linux command here. IDEs like VSCode and similar often have streamlined options for creating virtual environments that might be easier than the steps above.

Whether you are using a virtual environment or not, you will want to install the Python dependencies using the following command:

```
pip install -r requirements.txt
```

API .env Setup

Crucially, the frontend and backend have both provided `.env.example` files for use by the associated applications. In /backend/, you can use `cp .env.example .env` to copy the example file into a proper .env file. It is worthwhile to understand what each field does:

```
FLASK_ENV=development
FLASK_DEBUG=1
SECRET_KEY=change-me-for-real-use

DB_HOST=localhost
DB_PORT=3306
DB_NAME=itms
DB_USER=itms_user
DB_PASSWORD=[itms_password]

FRONTEND_ORIGIN: "http://localhost:5173"
```

You should set DB_PORT to the port MySQL Server is running on (3306 by default). DB_NAME will be the name of the database (itms if you used dump.sql). DB_USER should be your connection username - this is usually `root` for many users. DB_PASSWORD should be changed to your password for MySQL Server - **the api will not be able to connect to the database without this configured**.

You may want to change FRONTEND_ORIGIN later, if port :5173 does not work on your machine for some reason. SECRET_KEY can be kept as it is, although in a real-world scenario, it should be a long, secure, randomized string for use in cookie authentication.

Once the .env file is configured, you can run `python app.py` to start the app. You should see a message saying "Running on http://127.0.0.1:8000". With FLASK_DEBUG set to 1, you can see request information as it arrives from the frontend, or a testing framework like Postman.

Now you should be able to run the following to start the API server locally

```
python app.py
```

Now you can move on to setting up the frontend. **Make sure to complete the frontend setup in a new terminal**, so that by the end you have both the API layer and the frontend Vite server running at the same time.

Frontend Setup

PLEASE ENSURE THERE IS NO EXISTING node_modules FOLDER IN YOUR DIRECTORY. Remove it if still present.

To start the frontend, you can run the following commands:

```
cd itms-frontend
cp .env.example .env
npm install
```

This should create a .env file for the front end, and install all necessary node modules to run the frontend on your machine. Once your packages have finished installing, run the following command:

```
npm run dev
```

This will start the local Vite Dev Server, and output a url like `http://localhost:5173`. You should now be able to type that address into your browser's address bar and see the ITMS Login page.

A Note on Ports and CORS

By default, this application is set up with the following port assignments:

- MySQL: 3306 (default)
- API: 8000
- Frontend: 5173

This is built to work out of the box but if, for whatever reason, these ports are already occupied by your machine, you can change them in the following ways:

MySQL Server - This is the trickiest to reconfigure, and requires changing config files associated with MySQL Server. I recommend checking official documentation for guidance.

API - The API's port is defined at the bottom of app.py in a line reading `app.run(host="0.0.0.0", port=8000, debug=app.config["DEBUG"].`

If you would like to change this to another port, ensure that you also change the .env file found at /itms-frontend/.env to match with:

```
VITE_API_BASE_URL=http://localhost:<new_port>
```

Frontend - There is a very low chance of port collisions here, but if you'd like to change it, add the following to your vite.config.ts file:

```
export default defineConfig({
 // ...some configs
server: { port: <new_port>, },
});
```

Ensure that you change the associated port in /backend/.env:

```
FRONTEND_ORIGIN="http://localhost:<new_port>"
```

to ensure that your API layer allows requests from the frontend (CORS).

Project Use

You should now have a running backend (app.py) and frontend (Vite server). You can now access http://localhost:5173 (or whatever other port you may have chosen), and immediately get pushed to the login screen. The following accounts are provided by dump.sql:

Role	Username	Password
LEAD	lead1	leadpass
DEVELOPER	dev1	devpass
VIEWER	viewer1	viewpass

You should, at this stage, also be able to register your own user and start creating and editing projects. Feel free to:

- View projects
- View membership
- Create new projects
- Create/edit issues
- Assign and remove project members
- Apply labels
- Add and delete comments
- Watch history updates to issues over time

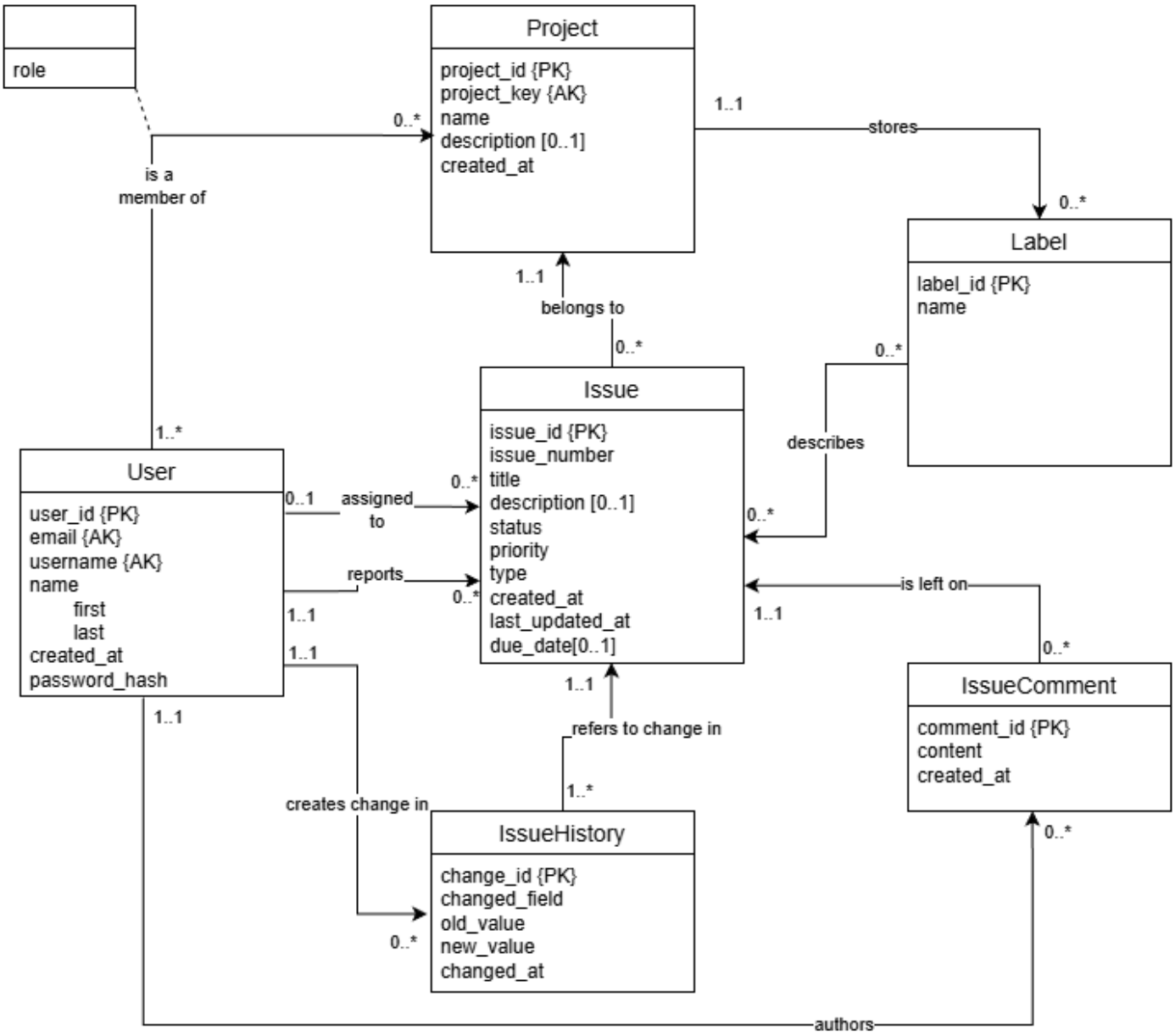
2) Technical Specifications & Description

The Issue Tracking Management System (ITMS) is a full-stack web application, providing authenticated, role-based project and issue management. The system consists of three primary layers:

1) MySQL 8.0 relational database
2) Flask (Python) backend API
3) React/Vite (JavaScript/TypeScript) frontend

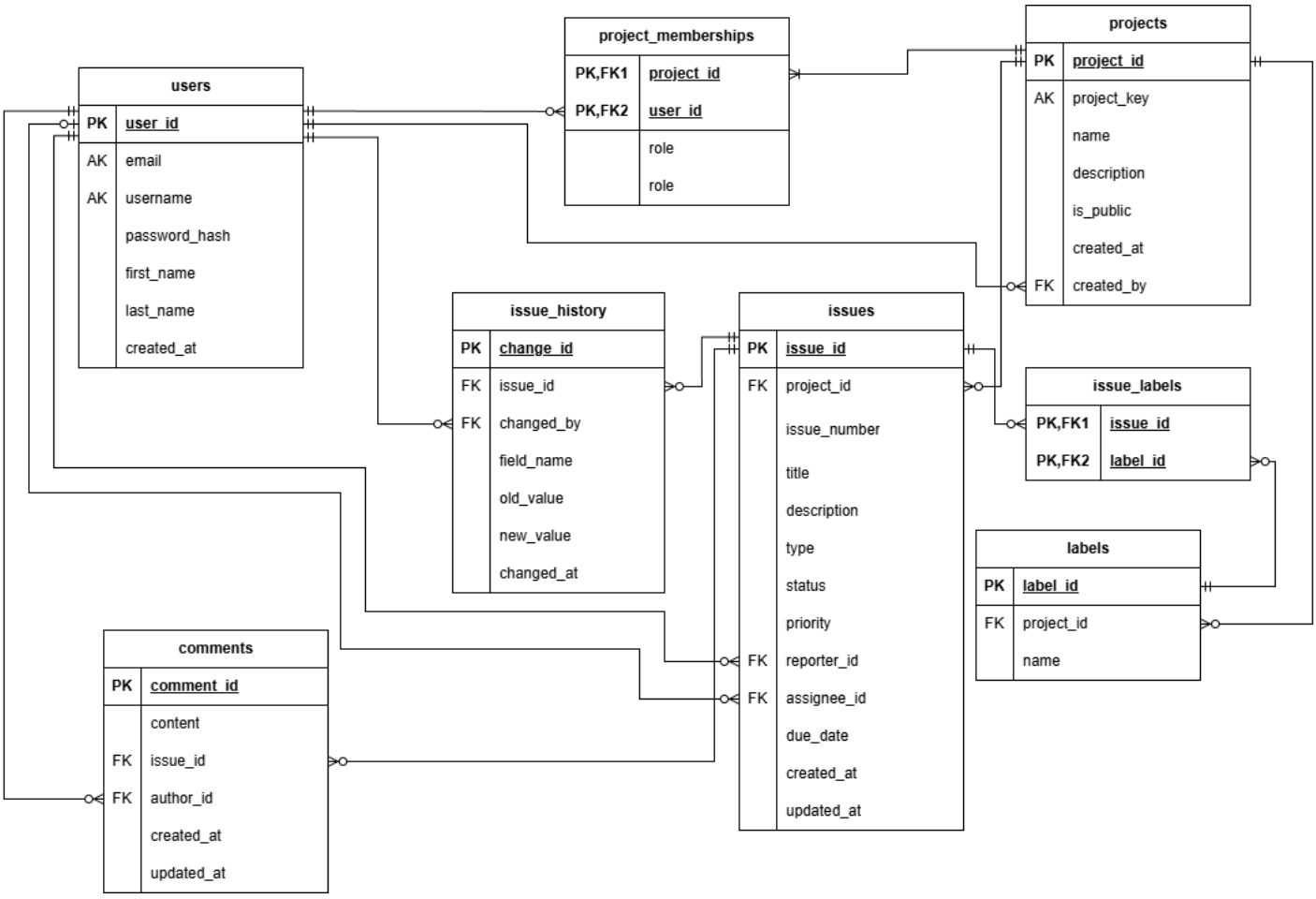
User authentication is handled by layer 2 (API), via HTTP session cookies that handle server-side sessions. In this way, permissions and authorization are both handled at the backend, and cannot be bypassed from the frontend application, even with browser-based alterations to API requests.

3) Conceptual Design

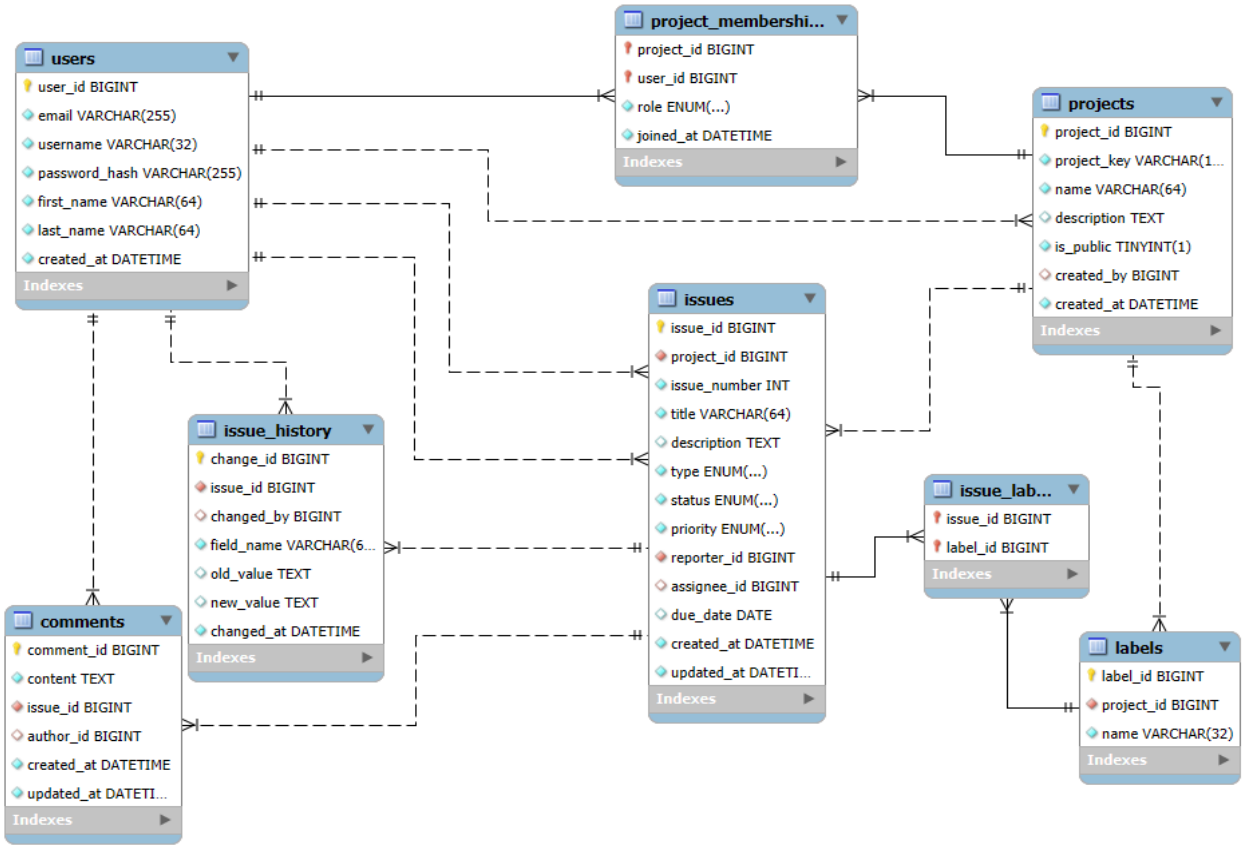


This ERD describes the high-level functional components of the system, including basic intended behavior, interactions, and desired data association.

4) Logical Design



This logical design outlines the schema used in the project, as well as multiplicities expected for each relationship. Notably, this includes foreign key relationships, as well as many-to-many tables such as project_memberships, which tracks which users are members of which projects. Care has been taken to align all indicators with their respective foreign keys.



This is the MySQL Workbench's reverse engineered logical design diagram. Notable, it shows the multiplicities of extant relationships, rather than all possible logical relationships (or lacks thereof). Therefore, it seems to omit zero-to-many relationship pointers in favor of stricter one-to-many relationship indicators for all relations.

5) User Flow

a) Actors and Roles

Users can be exactly one of the following:

- **Anonymous User:** Not logged in. Can only see the login and registration pages
- **Authenticated User:** Logged-in user. Can see all public projects as well as any private projects that they are a member of. Can edit their own username and first/last names provided there are no collisions with other existing accounts.

Users have the following relationship with any given project:

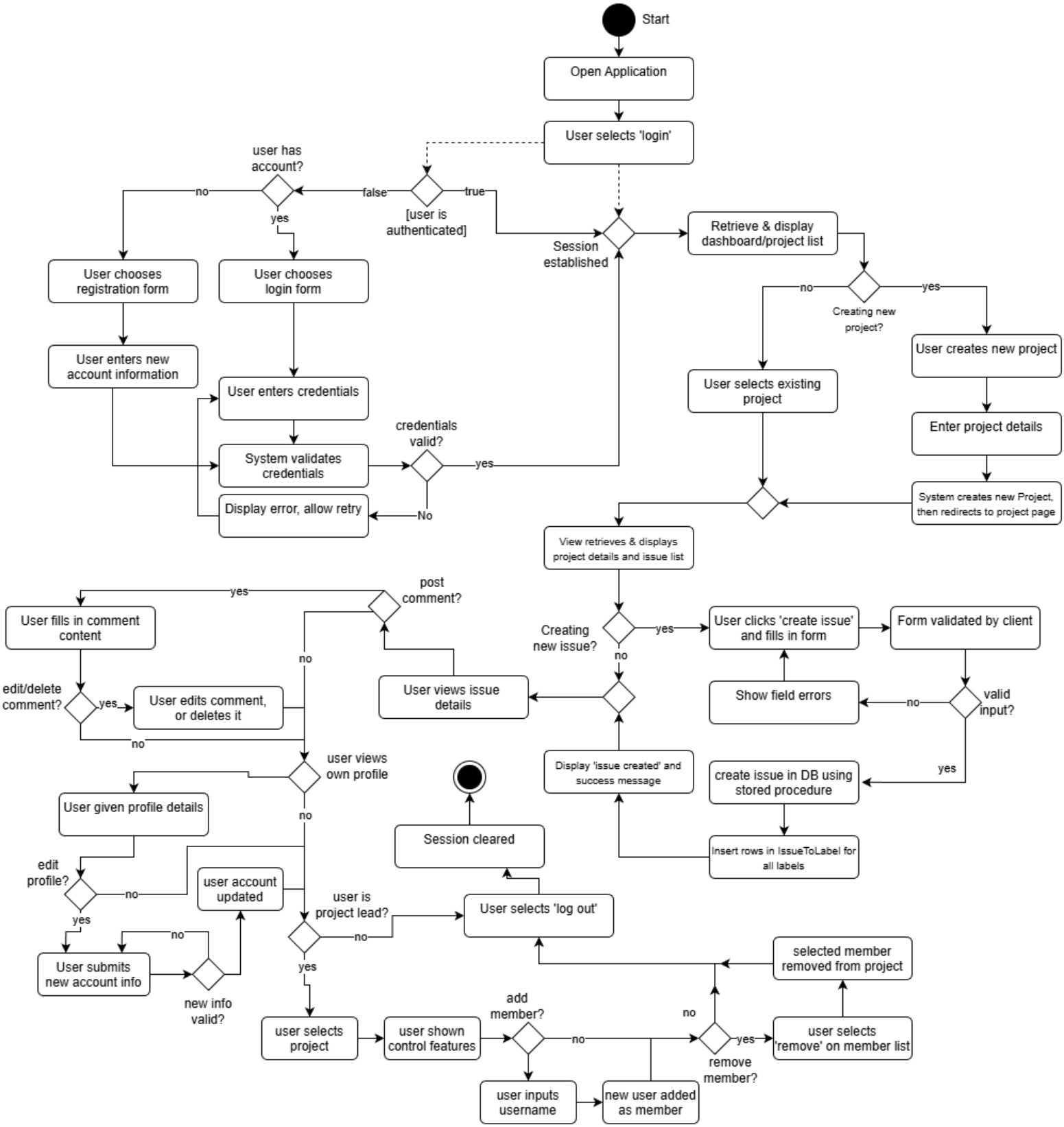
- **Non-Member Viewer:** a non-member that is viewing a public project. Can comment on issues and edit their comments, but cannot modify projects or issues under them
- **Member Viewer:** A member viewer that can view a private project. Can comment on issues and edit their comments, but cannot modify projects or issues under them.
- **Developer:** Can create issues and modify issues to which they are assigned. Full access to view all aspects of a project.
- **Lead:** Full project control. Can add or remove members, adjust roles, add or remove labels, create and edit issues, delete comments, and delete the entire project.

b) Global Overview

At a high level, a user flow proceeds as:

- Open application in browser. Frontend sends GET /me to check for an existing session
- If not authenticated, user logs in or registers a new account, and is then authenticated
- Once authenticated, user is directed to the Projects page and shown all projects visible to them
- User can select a project to view its details (issues, members, labels, visibility)
- Within a project, a user may:
    - View issues and their associated labels
    - Create/edit issues if their role allows
    - Add their own comments to issues
    - Edit their own comments
    - Delete comments if they are a lead or the author of that comment
    - View issue history
- Users may navigate to other projects
- Users can view their own profiles, and update their username, first name, and/or last name
- Users can log out, clearing their session and returning them to the login screen

A fuller activity diagram has been added to the following page. Please note that while thorough, it is not fully exhaustive of all user story orderings.


Start
Open Application
User selects 'login'
user has account?
no
yes
false
[user is authenticated]
true
Session established
Retrieve & display dashboard/project list
User chooses registration form
User chooses login form
User enters new account information
User enters credentials
System validates credentials
credentials valid?
yes
No
Display error, allow retry
Creating new project?
no
yes
User selects existing project
User creates new project
Enter project details
System creates new Project, then redirects to project page
View retrieves & displays project details and issue list
Creating new issue?
yes
no
User clicks 'create issue' and fills in form
Form validated by client
valid input?
no
yes
Show field errors
create issue in DB using stored procedure
Insert rows in IssueToLabel for all labels
Display 'issue created' and success message
User views issue details
post comment?
yes
no
User fills in comment content
edit/delete comment?
yes
no
User edits comment, or deletes it
user views own profile
yes
no
User given profile details
edit profile?
yes
no
user account updated
User submits new account info
new info valid?
no
yes
user is project lead?
yes
no
user selects project
user shown control features
add member?
yes
no
user inputs username
new user added as member
remove member?
yes
no
user selects 'remove' on member list
selected member removed from project
User selects 'log out'
Session cleared

6) Lessons Learned

This project was a great exercise in full-stack development. While I would not consider the application remotely finished, I'm quite happy with the experience I gained, alongside the opportunity I have to continue iterating upon it.

One of the biggest takeaways I gained was that of planning an application from start to finish, and the challenges that come with that. While my API had a staggering 31 endpoints (not counting those used for testing), I still found myself wanting new endpoints to improve the user experience on the frontend. One example is the ability to pull all issues to which a user has been assigned. This functionality doesn't currently exist, and implementing it on the frontend without updating the API would require recursive calls to the database, which I think is untenable for performance.

I also severely underestimated how much time would be eaten up by wiring different layers together. The API had to be defined before finalizing the database design, which was entirely dependent on user stories that interact with the frontend - a complete order reversal. Making frontend development sensible meant predefining all of the API calls and their request/response types, which was simple but tedious. While I have experience with frontend and full-stack development, my experience does not usually extend to writing the entirety of the API-database relationship myself (I usually use tools like an S3 bucket or a lambda function to firebase). Having to keep everything matching, type-safe, and compliant took more stamina than I anticipated.

If I had to choose to completely rework something, I'd have spent less time worrying about database triggers/procedures, and more time focusing on API shape and data expression. I found myself annoyingly limited by the data the API layer was returning, and while relying on procedures and triggers for data retrieval seemed like a good idea at first, I threw that idea away quickly when I realized the sheer scale and frequency of data accesses I'd need. There's a reason development has quickly moved to ORMs and predefined schemas.

Right now, the flexibility for data manipulation on the frontend is severely lacking. Users still can't delete their accounts, remove themselves from projects unless they're a lead, or retrieve information on other users easily. I think a user directory could be useful, and I want to expand the API layer considerably to accommodate for the number of features a user might want in an application like this. While nothing is working incorrectly, there's a *lot* of functionality left on the table that I look forward to sweeping up.

I genuinely think this is among the more useful projects I've undertaken in the past year, and I look forward to finding a way to integrate it into my website as a scalable, interactive demo.

7) Future Work

My primary goal at the moment is to work this into my personal website. My website uses React + Next.js, with a small API layer through next going into a Google Cloud Run server (GC's analog for AWS' Lambda). I'll likely rewrite a lot of the frontend code to more closely align with the theming and visual language of my own site - I have a background in animation that I think would compliment this project quite nicely.

Given that this was a class assignment, I left the niceties rather slim. Vite is very lightweight, but the functionality of Next, the ability to use Bun, and more heavy SSR (server-side rendering) to complement an increase in data retrieval would be a godsend. In terms of features, I plan to fully implement user control over data, including self-removal from projects, better issue management flows, and the ability to view other users' profiles to make it easier to find people to add to your project. I would love to add groups later, so that users can search for people in their own network and find them more easily. A stretch goal of mine would be to add email functionality to alert users to changes in their assignment, issue status, and project membership.

I would likely keep and expand the schema I came up with, as I think it does quite a good job of performing the tasks at hand. I'd move hosting from a self-hosted MySQL Server to something like PlanetScale or SupaBase (if I wanted to migrate to PostGres). The API layer could be moved to Vercel alongside the rest of the frontend in Next.js, which would simplify communication (types, interfaces, etc), as well as let me leverage SSR and edge computing on Vercel's network. I was shocked at how enjoyable this project was, and look forward to iterating on it over the next few months!