WEST UNIVERSITY OF TIMIŞOARA
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
BACHELOR STUDY PROGRAM: COMPUTER SCIENCE IN ENGLISH

BACHELOR THESIS

SUPERVISOR:
Asistent Dr. Florin Rosu

GRADUATE:
Serban Ples

TIMIŞOARA
2025

WEST UNIVERSITY OF TIMIŞOARA
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
BACHELOR STUDY PROGRAM: COMPUTER SCIENCE IN ENGLISH

Cloud Class

SUPERVISOR:
Asistent Dr. Florin Rosu

GRADUATE:
Serban Ples

TIMIŞOARA
2025

Abstract

University students today often juggle multiple tools for communication, collaboration, and managing resources, which leads to fragmented workflows and a lot of wasted time. In this thesis, I introduce *Cloud Class*, a browser-based platform I built to bring together real-time chat, group messaging, and centralized file sharing in one cohesive, student-friendly environment. I designed the system using a modular, microservice-based architecture, with each feature running as its own independently deployable service. This makes it easier to scale and adapt as needs evolve. The user interface focuses on simplicity and accessibility, so students can quickly form study groups, share ideas, and organize their documents all in one place. Throughout the development process, I focused on keeping data models clear, ensuring reliable communication between services, and writing solid tests to make sure that core features—like authentication, messaging, and file uploads—work smoothly together. Ultimately, *Cloud Class* aims to reduce context switching, streamline peer-to-peer interaction, and provide a flexible base for future features like calendar integration and video calling.

Contents

Chapter 1

Introduction

1.1 Motivation

In the modern educational environment, students face a lot of challenges that can impact their ability to maximize their academic potential. With the increasing demand of coursework, assignments, and exams, it is essential for students to have access to tools that offer them an easier learning process. One of the most significant obstacles students encounter is managing study materials. It is common for students to accumulate a lot of notes, articles, and resources, but often struggle to keep them organized and accessible when needed.

In addition to document management, collaboration plays a vital role in the learning experience. Peer discussions, group study sessions, and collaborative learning are essential components of academic success. However, the logistics of coordinating these activities, especially in today's digital age, can often prove to be a lot harder than expected. Scheduling study sessions, coordinating group discussions, and sharing feedback can become disorganized without an effective system in place.

Recognizing these challenges, the development of a web application aimed at improving the student experience has become increasingly important. This application serves as a one-stop solution for students to share learning documents, create chat groups for collaboration, and schedule study sessions. By providing a platform that integrates these essential tools, students getting the support they desperately need to enhance their productivity, engage in meaningful discussions, and collaborate more efficiently. This thesis explores the creation, functionality, and potential impact of this web application on student learning, aiming to provide an innovative solution that addresses the evolving needs of today's learners.

1.2 Objectives

The primary objective of this thesis is to design and develop a web application that addresses the key challenges students face in managing learning materials, collaborating with peers, and organizing study sessions. The objectives are as follows:

1. **Document Management**: To provide a user-friendly platform where students can easily upload, share, and organize learning documents such as lecture notes, textbooks, and articles. This functionality aims to ensure that

students have quick access to the materials they need for studying and can collaborate more effectively.

2. **Collaborative Features**: To create an interactive environment for students to communicate and collaborate through chat groups. These groups will allow students to engage in discussions, share ideas, clarify doubts, and foster a sense of community and teamwork, which is vital in the learning process.

3. **User Experience and Accessibility**: To ensure that the application is intuitive, easy to navigate, and accessible across different devices, providing a seamless experience for students. The user interface (UI) will be designed to accommodate a wide range of users, including those with limited technical expertise.

By achieving these objectives, the web application aims to enhance the overall learning experience for students, promoting better organization, communication, and academic success.

1.3 Contextualization

In the digital age, educational systems are rapidly evolving, influenced by advancements in technology and changes in student learning behavior. Traditional study methods, such as in-person lectures and physical textbooks, have become replaced by digital tools that allow for greater flexibility and accessibility. Despite the growing presence of online resources, students still face significant challenges in organizing and accessing learning materials, collaborating with peers, and effectively managing their time.

One key issue is the fragmentation of learning resources. With a variety of platforms and tools available for storing and sharing documents, students often struggle to find a centralized place to access and manage all their study materials. Whether it's lecture notes, reading materials, or research articles, organizing these documents can be a time-consuming task, leading to inefficiency and disorganization in students' study habits.

Moreover, the collaborative aspect of learning is often hindered by communication barriers. While students rely on chat platforms and messaging apps for group discussions, the lack of integration with academic tools makes it difficult to focus on study-related tasks. Similarly, coordinating group study sessions often requires juggling multiple apps and platforms, leading to missed opportunities for productive collaboration.

The contextualization of this research emphasizes the need for a unified platform where students can not only manage their learning materials but also interact with their peers and schedule study sessions efficiently. This chapter explores the academic and technological landscape that has led to the development of this web application, highlighting the gaps in existing tools and the opportunities for improvement.

1.4 Relevance

The relevance of this research is rooted in the increasing demand for educational technology that addresses the evolving needs of students in modern academic environments. As more students move toward online and hybrid learning models, the necessity for integrated platforms that streamline the learning process becomes more evident. This web application offers a solution by combining document sharing, peer collaboration, and study session management into a single, user-friendly interface.

This application is highly relevant for current educational contexts, where students are often balancing multiple courses, assignments, and extracurricular activities. By providing a platform that enables students to access and organize learning materials, collaborate effectively with peers, and manage their study schedules, the app can significantly enhance productivity and academic performance.

The relevance extends beyond individual academic success; this research also contributes to the broader conversation around educational equity and technology access. By creating a tool that is easy to use and accessible, the application has the potential to benefit a wide range of students, including those from diverse backgrounds and those who may not have access to advanced learning management systems or collaboration tools.

Furthermore, the growing role of educational technology in shaping the future of education makes this research particularly significant. As the landscape of education continues to evolve, tools like this web application can pave the way for more efficient, inclusive, and collaborative learning environments, promoting a deeper connection between students and their academic communities.

Chapter 2

Related Work

2.1 Existing Solutions

A number of established platforms address parts of the collaboration and organization problem, but none combine document sharing, chat, study scheduling and real-time video in a student-centric package:

Slack [1] Slack is widely adopted in professional and academic circles for team communication. Its channel-based model and rich integration ecosystem (e.g. file attachments, bots, external APIs) make it ideal for project-driven discussion. However, Slack's interface and feature set are geared toward corporate workflows; it lacks built-in study-session scheduling or academic resource management, and its cost structure can be prohibitive for student use.

Google Classroom [2] Google Classroom streamlines assignment distribution, grading, and resource sharing within educational institutions. It integrates seamlessly with Google Meet for video sessions and Google Drive for file storage. While strong on assignment workflows, Classroom is not designed for open-ended study groups or peer-to-peer chat outside of formal class rosters, and offers limited flexibility in customizing collaborative spaces.

Microsoft Teams [3] Microsoft Teams unifies chat, file sharing, and video conferencing under the Office 365 umbrella. Its "Teams" and "Channels" structure supports both class-level and small group discussions, and includes a built-in calendar. Despite its power, the learning curve and enterprise focus of Teams can overwhelm students, and it provides few features tailored specifically to student-driven study sessions.

NutriBiochem Mobile Application [4] The NutriBiochem app was crafted for undergraduate biochemistry and nutrition courses, offering interactive content and quizzes. Its design demonstrates how a domain-specific tool can deeply engage learners in a particular subject area. However, its tightly focused scope limits reuse across varied disciplines, and it omits real-time communication or scheduling capabilities common to collaborative study.

2.2 Gaps in Current Solutions

Although these platforms excel at discrete tasks—chat (Slack, Teams), course management (Google Classroom), or subject-focused learning (NutriBiochem)—several gaps remain:

First, none provide a unified, student-driven environment that seamlessly combines file management, ad hoc group chat, and on-demand study-session scheduling. Students often juggle multiple services (e.g. Slack for chat, Google Drive for documents, Calendly for scheduling), which introduces friction and context-switching overhead.

Second, current tools emphasize instructor-led workflows or corporate use cases; they lack features like flexible "study rooms," lightweight event RSVPs, and peer-to-peer resource tagging that empower students to take ownership of their collaborative learning.

Finally, integration with external calendars and real-time video is either missing or bolted on as a generic add-on, rather than designed around the rhythm of student study sessions and group learning.

2.3 Proposed Solution

In response to the limitations of existing tools, I have developed a web application focused on two core capabilities: file sharing and real-time chat. By combining these into a single, student-centric platform, the goal is to reduce the fragmentation and context-switching that students experience when juggling separate services.

- **File Upload & Repository:** Users can upload lecture notes, articles, and other learning materials directly into organized collections. Each file is stored securely, versioned, and can be tagged or searched by course, topic, or keyword. Peers in the same study group can browse and download shared resources without leaving the application.
- **Real-Time Chat Groups:** Ad-hoc chat rooms allow students to form topic-specific discussion channels on the fly. Messages are delivered instantly, with "seen" indicators and simple reaction support. The chat interface is designed for quick group Q&A, file previewing, and link sharing, all within the same context as the file repository.

The architecture is deliberately modular—each feature lives in its own microservice—so that future additions (for example, scheduling study sessions or integrating with external calendars) can be slotted in with minimal disruption.

Chapter 3

Architecture and Technologies

3.1 Architecture

The application uses a combination of Event Driven architecture and Request-Response architecture, making as much use as possible of Node JS's[5] built in capabilities for handling multiple operations on a single thread using concurrency and asynchronous programming.

The application consists of multiple microservices. These microservices implement communication patterns like *Request-Response* and *Publish/Subscribe*. All microservice communication is done using message queues like RabbitMQ and BullMQ. While RabbitMQ offers great support for both Request-Response communication and Publish/Subscribe communication, I chose to use BullMQ for Publish/Subscribe. BullMQ uses Redis to queue messages and process them. By using Redis, BullMQ offers an easy way to schedule the times when a message is executed and implement retry policies. On the other hand, RabbitMQ is built upon AMQP (Advanced Message Queuing Protocol)[6]. This makes it possbile to pass messsages across an internal network and register handlers for these messages.

Since this application is heavily focused on microservices, here is a short description for each one of them:

- **Webserver Service** - Single HTTP server, tasked with serving the client application and providing HTTP routes for fetching data.
- **Authentication Service** - Service handling all authentication logic. This server is only accessible via RabbitMQ queues.
- **Core Service** - Service handling all reads and writes to the database. This server is accessbile via RabbitMQ queues.
- **Authorization Service** - Service handling authorization across the whole application. The application implements a RBAC (Role Based Access Control). Each request goes through this service before reaching the Core service. Only accessbile via RabbitMQ.
- **Uploader Service** - Service handling all file uploads. Since uploads are usually resource intensive tasks and can fail due to numerous reasons, this is a processor service using BullMQ.

- **Mail Service** - Service handling all emails sent. Sending emails is also a resource intensive task especially if sending to multiple users at the same tim. This is a processor service using BullMQ.
- **Websocket Server** - Server exposing only a websocket connection. This server is handling all of the chat features inside of the application, listening for notifications sent by clients, invoking the Core service for different writes (saves, updates) to the database, and dispatching a notification to the intended client.
- **Notification Server** - Server exposing a route for handling SSE (Server Sent Events). This server is the one handling in-app notifications.

3.2 Technology Stack

The *Cloud Class* application is a fully browser-based system built end-to-end in TypeScript/JavaScript. Both client and server leverage modern, asynchronous architectures and microservices to ensure scalability, maintainability, and real-time interactivity. This section outlines the key technologies and frameworks that form the backbone of my solution.

3.2.1 Server Implementation

The backend of *Cloud Class* is implemented as a suite of modular microservices responsible for business logic, data persistence, and event handling. All services are written in TypeScript, compiled to JavaScript, and orchestrated via Docker.

- **TypeScript**
  - A statically typed superset of JavaScript that compiles to ES2019.
  - Enables interface-driven design, strong type checking, and advanced IDE support (e.g., IntelliSense).
  - Facilitates early detection of errors, consistent code formatting (TSLint / ESLint), and straightforward refactoring in large codebases.
- **Node.js**
  - Event-driven, non-blocking I/O runtime built on Chrome's V8 engine.
  - Ideal for handling high concurrency with minimal resource overhead.
- **NestJS**
  - Progressive Node.js framework leveraging decorators and dependency injection.
  - Promotes a modular architecture: controllers, providers, and gateways can be developed and tested independently.
  - Native support for REST, GraphQL, WebSockets, and microservice patterns (TCP, gRPC, Redis-based transport).

- **MongoDB**
  - NoSQL document database storing data in flexible BSON format.
  - Supports horizontal scaling via sharding, automatic replica sets for high availability, and ACID transactions across multiple documents (v4.0+).
  - Optimized for rapid development cycles, enabling schema evolution without downtime.
- **MinIO (S3-compatible)**
  - High-performance, Kubernetes-native object storage with Amazon S3 API compatibility.
  - Features erasure coding, bitrot protection, server-side encryption, and multi-tenant isolation.
- **Real-Time Communication**
  - *Socket.IO*: Bidirectional WebSocket library with automatic fallback to HTTP long-polling. Used for live chat, collaborative editing, and presence notifications.
  - *Server-Sent Events (SSE)*: Lightweight, one-way event streams for broadcasting state changes (e.g., file uploads progress, system alerts) with minimal overhead.
- **Logging & Analytics:**
  - *Elasticsearch & Kibana*: Distributed search and analytics engine powering full-text search, structured queries, and real-time log aggregation.
  - Logs are shipped via Filebeat into an ELK stack, enabling dashboards for performance monitoring, error tracking, and usage metrics.
- **Testing**
  - *Jest*: Zero-configuration JavaScript testing framework for unit and integration tests, featuring snapshot testing and code coverage reports.
- **Message Queues & Background Jobs:**
  - *RabbitMQ*: AMQP broker for reliable, complex routing of messages between microservices (e.g., task dispatch, order processing).
  - *BullMQ*: Redis-backed job queue for scheduled and repeatable tasks such as email delivery, report generation, and notification bursts.
  - Combined to balance high-throughput event streams (RabbitMQ) with lightweight, retry-capable background jobs (BullMQ).

3.2.2 Client Implementation

The frontend of *Cloud Class* is a single-page application delivering responsive, component-driven user experiences. Written in TypeScript and React, it communicates with backend microservices via REST, and real-time channels.

- **HTML5 & Semantic Markup**
  - Establishes the document structure with accessibility in mind (ARIA roles, landmarks).
- **CSS3 & Responsive Design**
  - Utilizes Flexbox, Grid layouts, and custom properties (CSS variables) for adaptable, maintainable styling.
- **React & TypeScript**
  - Component-based architecture with JSX syntax and a virtual DOM for efficient UI updates.
- **Real-Time Updates (Socket.IO Client)**
  - Establishes persistent WebSocket connections for instant notifications (e.g., new messages, collaborative whiteboard changes).
  - Automatic reconnection and event buffering ensure continuity during transient network issues.

Chapter 4

Application Description

4.1 Features

The main features of the application are: a steady authentication, a live-chat for allowing students to study in groups and a file-sharing system for easily sharing study notes.

4.1.1 Authentication

The most important part of any web based application is the authentication. While developing the authentication system for my application, I followed OAuth2[7] standards.

Registration: During registration, users submit their full name, email, password and a confirmation password. The system hashes the password with a strong algorithm (implemented using bcryptjs), creates a new user record marked as unverified, and generates a time-limited email verification token. The token is stored in the database and emailed, prompting users to confirm their address. (Figure 4.1)

Endpoint: /auth/register

Method: POST

Request Body: JSON Object containing fullname, email, password and confirmation password.

Response: JSON Object containing a success message if the request was successful

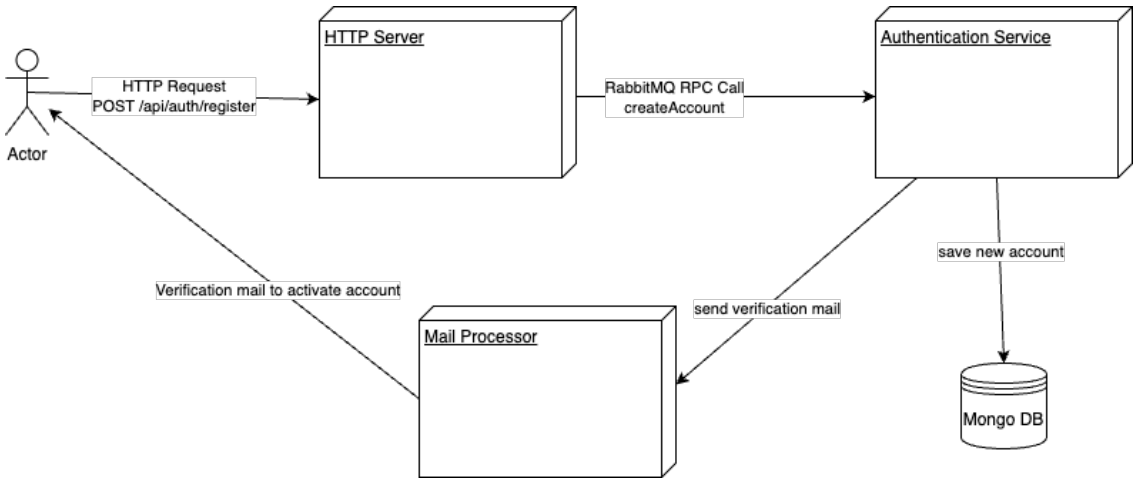

Figure 4.1: Register Flow

Verify Account: When users click the verification link, the system extracts the token and searches the database for a matching, unexpired record. If valid, it marks the user's account as verified, removes the token to prevent reuse, and returns a success response, enabling full access to login and other protected features.

Endpoint: /auth/verifyaccount

Method: GET

Query Parameters: verificationToken (string) -¿ verification token for certain account, recieved via email.

Response: JSON Object containing a success message if the request was successful

Login: On login, users submit credentials, which the system verifies against stored hashes. After confirming the account is verified, it generates a cryptographically secure session code, encrypts it, and stores it in the database under the user profile. The code is set in an HttpOnly, Secure cookie, establishing the authenticated session. (Figure 4.2)

Endpoint: /auth/login

Method: POST

Request Body: JSON Object containing email, and password.

Response: JSON Object containing a success message if the request was successful.

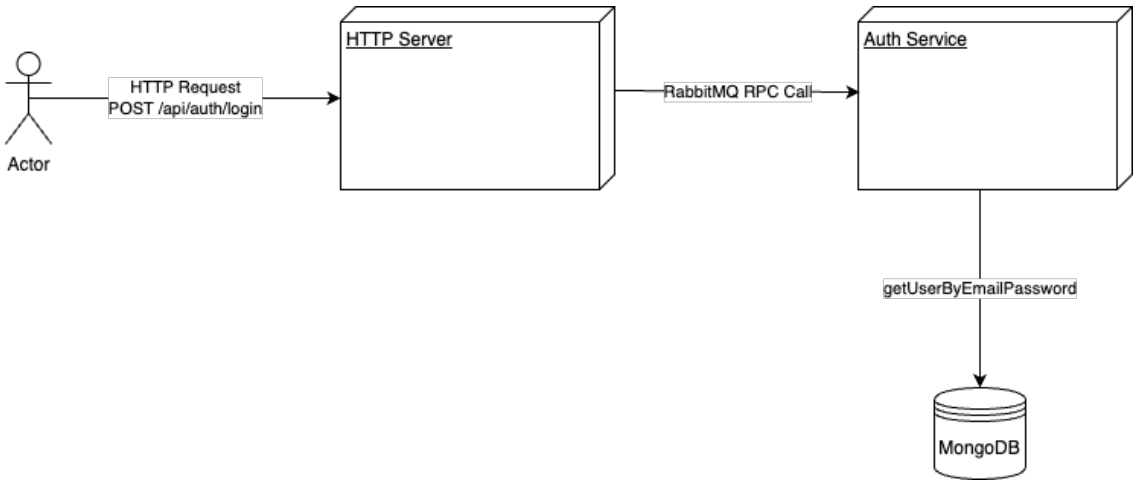

Figure 4.2: Login Flow

Reset Password: When a password reset is requested, users provide their email address. The system generates a short-lived, cryptographically random reset token, stores it, and emails a reset link. Upon visiting the link and submitting a new password, the system validates the token, updates the password hash and deletes the token.

Endpoint: /auth/reset-password

Method: POST

Request Body: JSON Object containing reset token, password and confirmation password.

Response: JSON Object containing a success message if the request was successful

Logout: For logout, the client sends a request to invalidate the session. The server locates the encrypted session code in the database, removes it, and instructs the browser to clear the authentication cookie by setting it with a past expiration. The user is then unauthenticated for any subsequent requests.

Endpoint: /auth/logout

Method: POST

Response: JSON Object containing a success message if the request was successful.

Whoami: For Whoami, the client sends a request to get the user holding the current session. The server locates the encrypted session code in the database, and returns the user data.

Endpoint: /auth/whoami

Method: Get

Response: JSON Object containing userId, role and email of the logged in user if the request was successful.

4.1.2 Live-chat

The main part of any peer to peer platform is a real-time chat. Since this application focuses on providing students a way to communicate with each other in order to make stuying togheter easier, a real-time chat is a core functionality that I had to implement.

The real-time chat is implemented using Web Sockets[8]. Even though Web Sockets are usually used for machine to machine communication, most applications that include a real-time chat like Microsoft Teams, Discord, Whatsapp use Web Sockets under to hood.

Web Sockets work by creating a bidirectional communication channel between clients and servers. Data is transported on this channel using messages. Each message must have a defined 'name' and maybe have some content.

In my implementation, I started by thinking about all of the messages that would be used inside of a chat application. After careful consideration, I ended up with 5 message types that my application must use:

- **Join Conversation** - Message sent from the client side, containing the user authentication token and a conversationId. The server then checks if the user is a participant of that conversation, and responds back with a message of 'joined-conversation' (Figure 4.3).

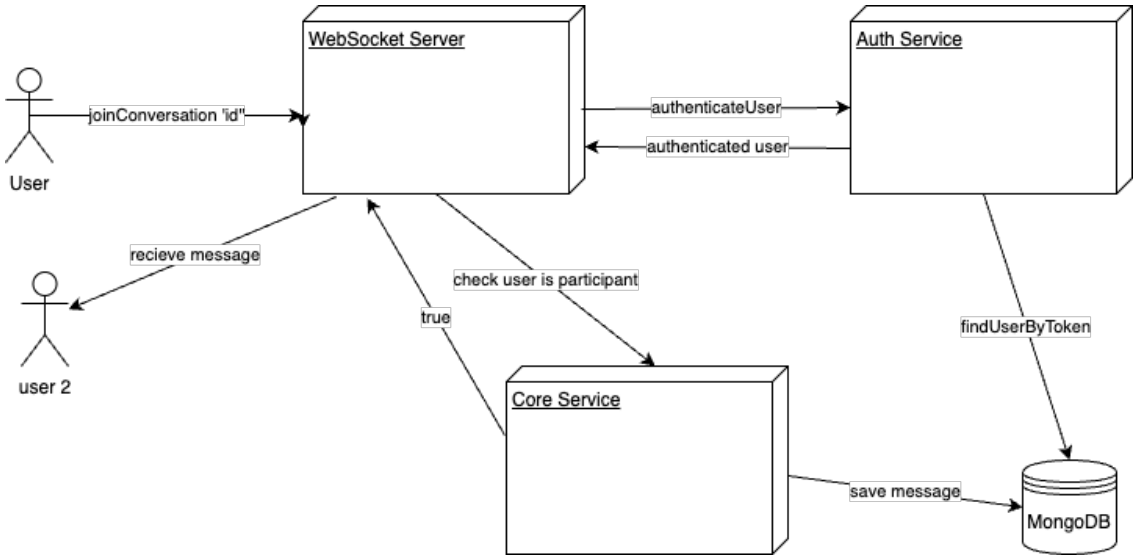

Figure 4.3: Join Conversation Flow

- **Leave Conversation** - This message is sent by the client side, when a user decides to leave a conversation. Leaving a conversation is done by either joining a different conversation, navigating to a different part of the application or closing the application. For this message, the server just removes the client from the conversation session.

- **Send Message** - This is the last and most importand message sent from the client application. Everytime the user creates a message and decides to send it, the application will send a message containing the user's authentication token, message content and conversationId. After creating a database entry with the message details, the server responds back with 'recieve-message', in order to alret the application that a new message appeared (Figure 4.4).

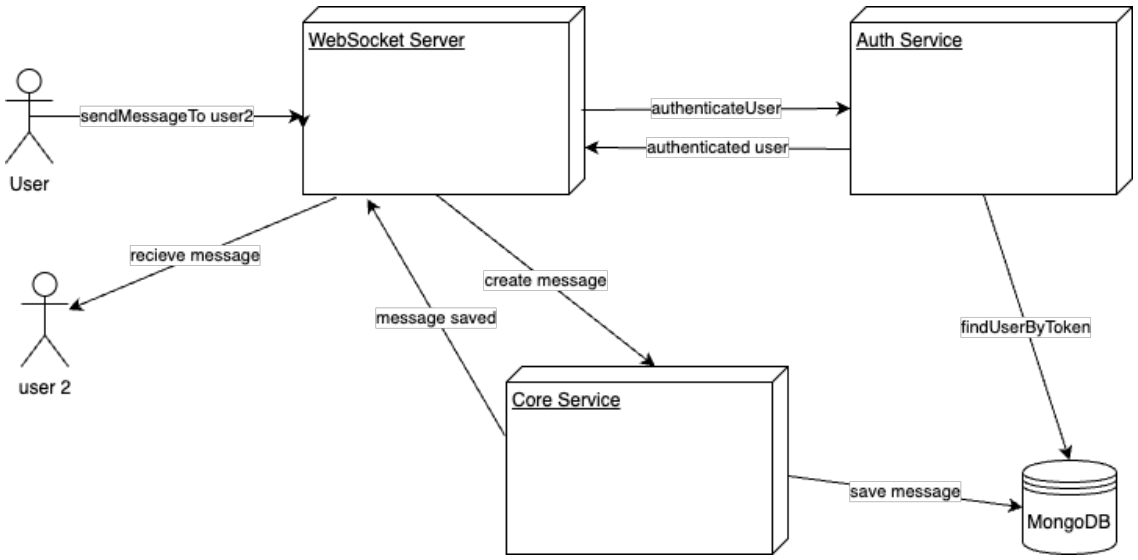

Figure 4.4: Send Message Flow

- **Recieve Message** - The server sends this message after creating a entry in the database for the message sent by a user. The content of this message is the full database entry created. The client application listens for this message and correctly displays the new chat.

- **Joined Conversation** - After a user joins a conversation, the server sends this message back to the client application. This notifies the application that it can let the user start sending messages and recieving messages.

4.1.3 File Uploader

In the early days of file exchange, teams set up FTP servers on static IPs, juggling usernames, anonymous logins, and fragile directory hierarchies. Uploads would time out, backups were manual, and there was no easy way to integrate with modern web apps. As requirements grew—from embedding user-generated images in chat apps to serving large datasets via HTTP—FTP simply couldn't keep up. Object storage emerged as the successor: a RESTful, scalable, self-healing store where applications can generate time-limited download links, automatically version files, and offload the burden of replication and durability. Today, replacing FTP with an S3-compatible store streamlines both developer workflows and end-user experience, turning clunky file drops into seamless, secure uploads and downloads.

S3-compatible object storage treats each file as an immutable "object" stored in a flat namespace of buckets. It exposes a simple HTTP API—PUT to upload, GET to download, DELETE to remove—so developers don't wrestle with filesystem mounts or NFS quirks. Objects can carry arbitrary metadata (key/value tags or headers), and platforms often offer built-in features like automatic replication across nodes, lifecycle policies (e.g. archive or delete old versions), and fine-grained access controls. Because it's designed for petabyte-scale data, S3-style stores provide high throughput for large payloads, seamless horizontal scaling, and durability guarantees that make them a natural fit for user uploads, backups, and any scenario where large binary blobs need reliable, cost-effective storage.

Traditional databases shine at structured records: rich querying, ACID transactions, indexing, and relationships across tables. However, stuffing multi-megabyte or multi-gigabyte files into a database can bloat its storage engine, slow down queries and backups, and drive up resource costs. In contrast, an S3-style object store offloads blob storage entirely, keeping the database lean for metadata lookups and relational joins. Object stores scale throughput linearly with added nodes, use pay-as-you-store pricing models, and excel at serving large downloads with built-in HTTP caching semantics. While a database might store a file's URL and metadata, the object store holds the bytes—each in its own durable, self-healing container.

In *Cloud Class*, the file uploader is a key component that allows users to share study materials, lecture notes, and other resources. It is designed to handle large files efficiently while ensuring reliability and scalability. The uploader integrates with an S3-compatible object storage service (MinIO) for storing files, and uses a message queue (BullMQ) to manage upload jobs asynchronously.

The file uploader is implemented as an asynchronous pipeline that delegates heavy I/O work to background workers. When a user submits a file, the client issues an HTTP POST to the webserver, which immediately enqueues a processing job and returns a 201 Created. A dedicated worker then uploads the file to object storage and notifies the persistence service to save metadata in the database, linking the file to the user's record. (Figure 4.5)

- **Client HTTP Request** The frontend sends the file as `multipart/form-data` to `/files/upload` and receives back a job identifier.

- **Webserver Endpoint** The HTTP handler enqueues a "process-file" job on the queue, packaging the raw file bytes along with user context.

- **Job Queue** Redis + BullMQ persist and manage jobs with retry policies, ensuring uploads survive transient failures.

- **Worker Process** A background worker consumes each job, reconstructs the file buffer, generates a unique storage key, and uploads the file to the S3-compatible store.

- **Object Storage** Minio stores the binary in a bucket and returns a URL or object key.

- **Persistence Service** Upon successful storage, an event is sent to the core service, which creates a MongoDB document for the file (URL, key, MIME type, size, uploader ID, timestamps) and adds a reference to the user's files array.

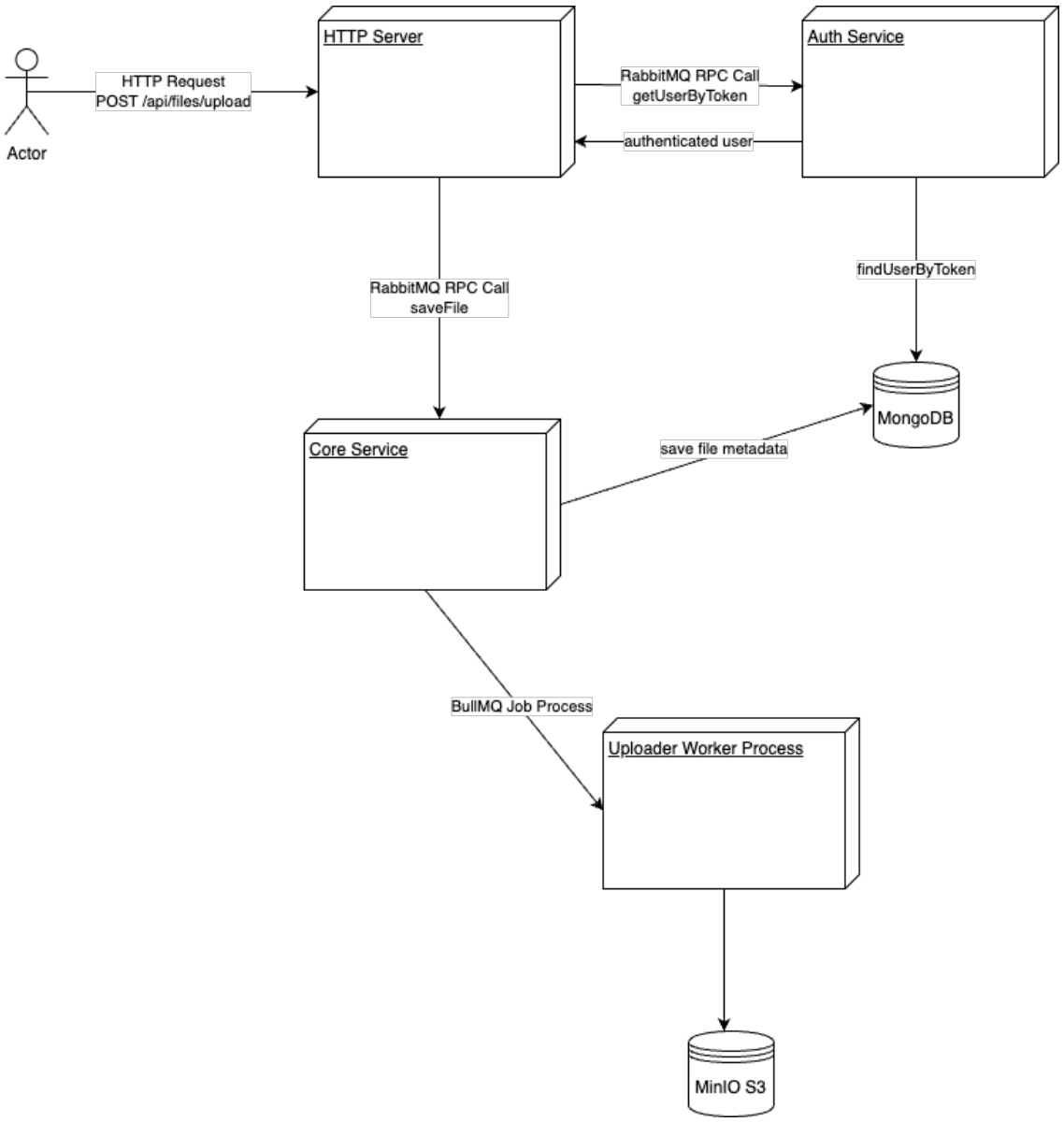

Figure 4.5: File Upload Flow

4.2 Interface

This is the user interface for chat feature, where users can send and receive messages in real-time (Figure 4.6).

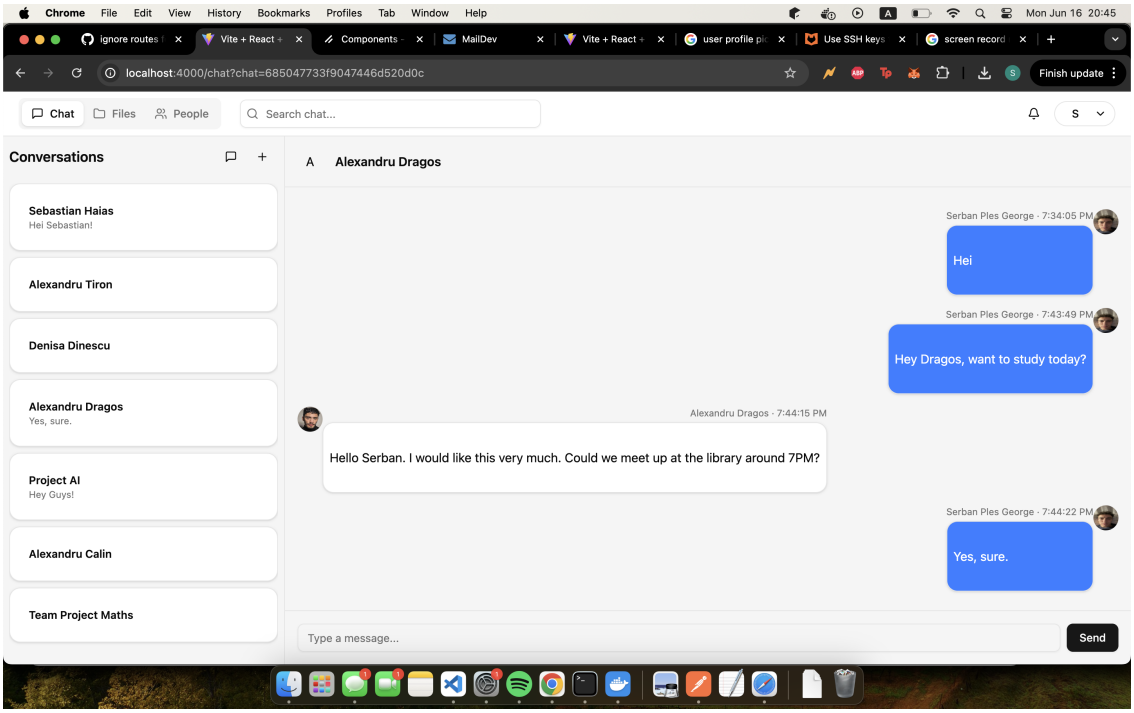

Figure 4.6: User interface for the chat feature

This is the user interface for displaying information about a study resource, from where users can download it (Figure 4.7).

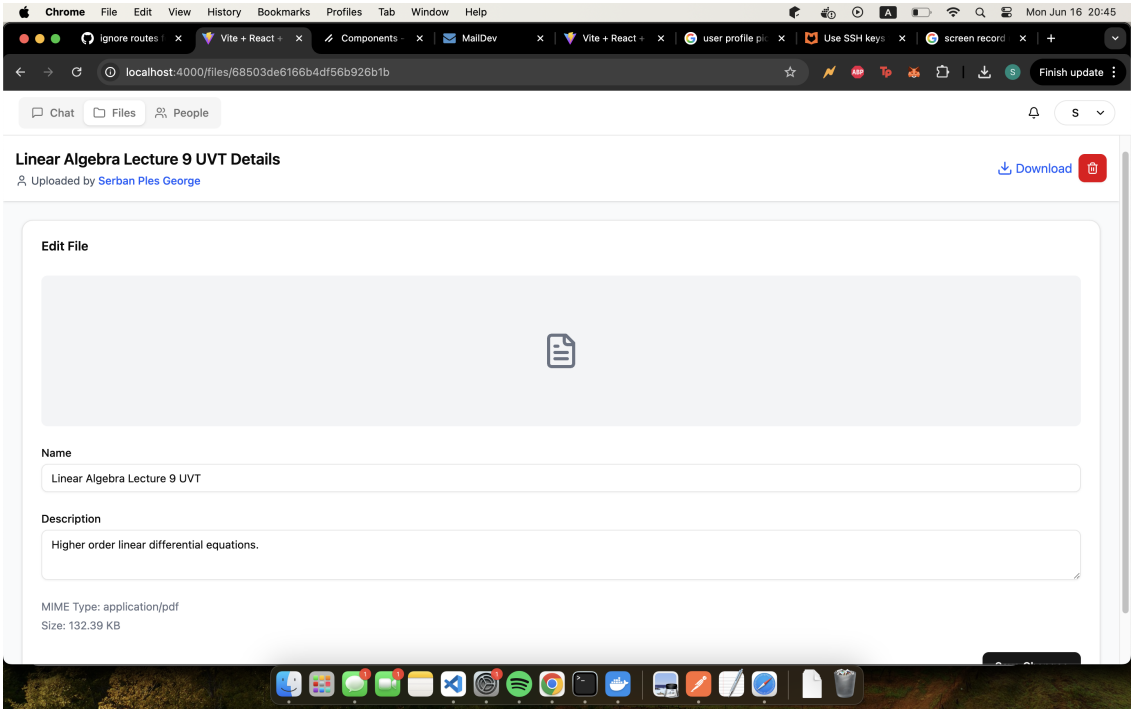

Figure 4.7: User interface for the file feature

This is the user interface for displaying profile information. In this interface, other users can see a specific user's uploaded files and additional data about them. (Figure 4.8).

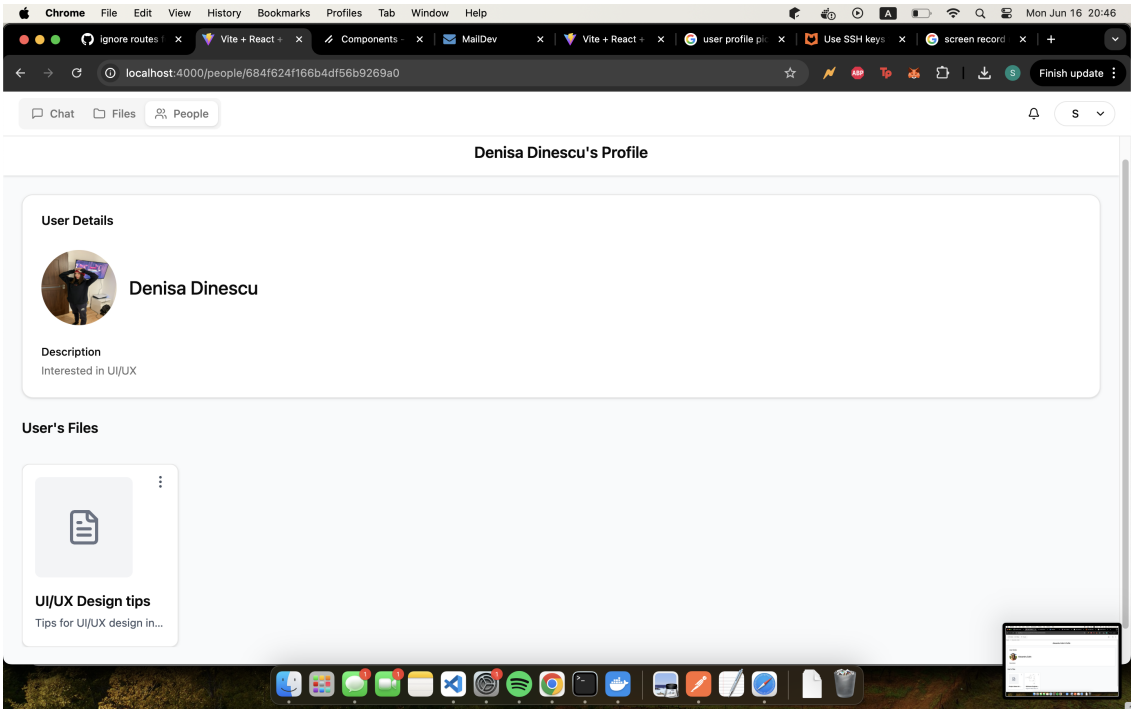

Figure 4.8: User interface for the profile feature

This is the user interface for uploading a file. Using this interface, the user can drag and drop a file inside of the marked area, edit the name, add a description, and upload the file directly to the server (Figure 4.9).

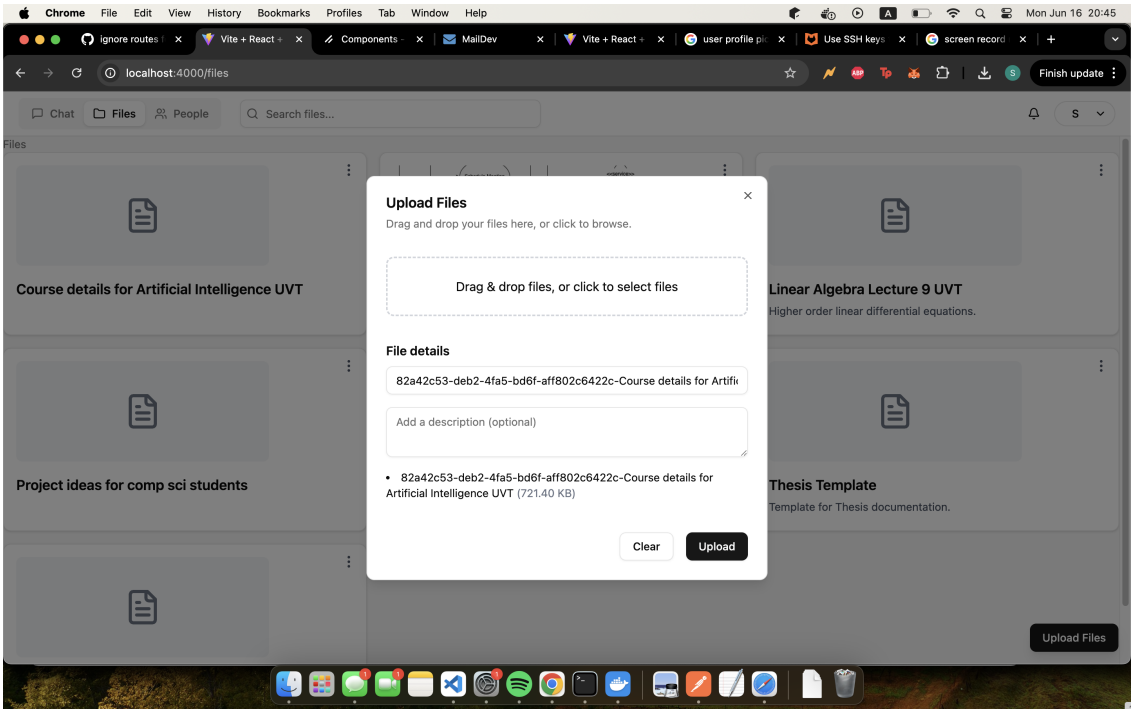

Figure 4.9: User interface for the upload feature

4.3 Database Structure

4.3.1 Overview

Relational databases have long been the default choice for structured data storage, but modern web applications often require far greater flexibility and horizontal scalability than traditional SQL systems can easily provide. NoSQL databases, in particular document-oriented stores, have emerged to meet these requirements by allowing each record to be a self-describing JSON-like document rather than rigid rows and columns. This schemaless approach enables rapid evolution of data models as application requirements change, without costly migrations or downtime.

MongoDB[9] is one of the most popular document-oriented NoSQL databases. It persists data as BSON (Binary JSON) documents, offering rich query capabilities, indexing on nested fields, and atomic operations on single documents. With built-in support for horizontal sharding and automatic replica sets, MongoDB can scale across many servers while providing high availability and fault tolerance. Moreover, its native JSON syntax for queries and updates makes the developer experience intuitive when working with JavaScript/TypeScript on both client and server.

To bridge the gap between the dynamic nature of MongoDB and the static typing guarantees of TypeScript, I used Mongoose as an Object Document Mapper (ODM). Mongoose allows strict schema definitions, validation rules enforcing, and hooking into lifecycle events (middleware) for tasks like hashing passwords or populating references. This combination of MongoDB‘s flexible document model with Mongoose‘s type-safe schema definitions ensures both agility during development and robustness in production.

4.3.2 Data Modeling

In a schemaless document store like MongoDB, documents can in principle contain any structure. However, imposing a schema at the application level provides consistency, validation and clarity. Data modeling involves defining, for each collection, the shape of its documents: field names, data types, default values, required properties and indexing strategies. This layer of structure enables the application to enforce invariants (e.g. “email must be a non-empty string” or “createdAt must always be present”), catch errors early, and generate optimized queries.

Schema options often include:

- **Automatic Timestamps:** Adding `createdAt` and `updatedAt` fields to every document to track its lifecycle.
- **Serialization Transforms:** Customizing how documents are converted to JSON—for example converting internal object IDs into strings and removing internal metadata fields.
- **Validation Rules:** Specifying which fields are required, their allowed types (String, Number, Boolean, Date, ObjectId, Array, etc.), and constraints such as unique or enum values.
- **Default Values:** Providing sensible defaults (e.g. an empty array for list fields, ‘false‘ for boolean flags) to simplify document creation.

Modeling relationships in a document database can be achieved in two main ways:

- **References:** Storing the identifier of a related document (typically an ObjectId) in a field. This keeps collections normalized and minimizes duplication, at the cost of additional lookups when joining data at query time.
- **Embedded Subdocuments:** Nesting related data directly within a parent document. This approach denormalizes data for faster reads and atomic updates of the entire structure, best suited for "owns" or "contains" relationships with bounded subdocument sizes.

Finally, careful use of indexes is critical for query performance and data integrity:

- **Unique Indexes:** Enforce uniqueness on fields such as email addresses or usernames to prevent duplicate accounts.
- **Compound Indexes:** Accelerate queries involving multiple fields (e.g. sorting messages by conversation and timestamp).
- **TTL Indexes:** Automatically expire documents after a given time, useful for short-lived tokens or audit logs.

By combining a clear schema definition with appropriate relationship modeling and indexing, the application gains both the flexibility of a document store and the reliability of a structured database.

4.3.3 Schema Definitions

Below I would like to present each schema with its MongoDB collection name and a concise description of the most important fields and relationships.

Account Schema (`accounts`)

Models user authentication, authorization, and credential management (Figure 4.10).

`email` **(*String*, required, unique)** Primary login identifier, enforced as unique to prevent duplicates.

`password` **(*String*, required)** Hashed credential; excluded from default query results for security.

`role` **(*String*, enum {USER, ADMIN})** Access level determining permissions within the application.

`isVerified` **(*Boolean*, default:** `false`**)** Indicates whether the user's email has been confirmed.

`accountVerificationToken`, `verificationTokenExpiration` Token and expiry timestamp for email verification workflows.

`passwordResetToken`, `resetTokenExpiration` Token and expiry timestamp for password recovery processes.

`profileId` (***ObjectId***) Reference linking account credentials to the corresponding user profile.

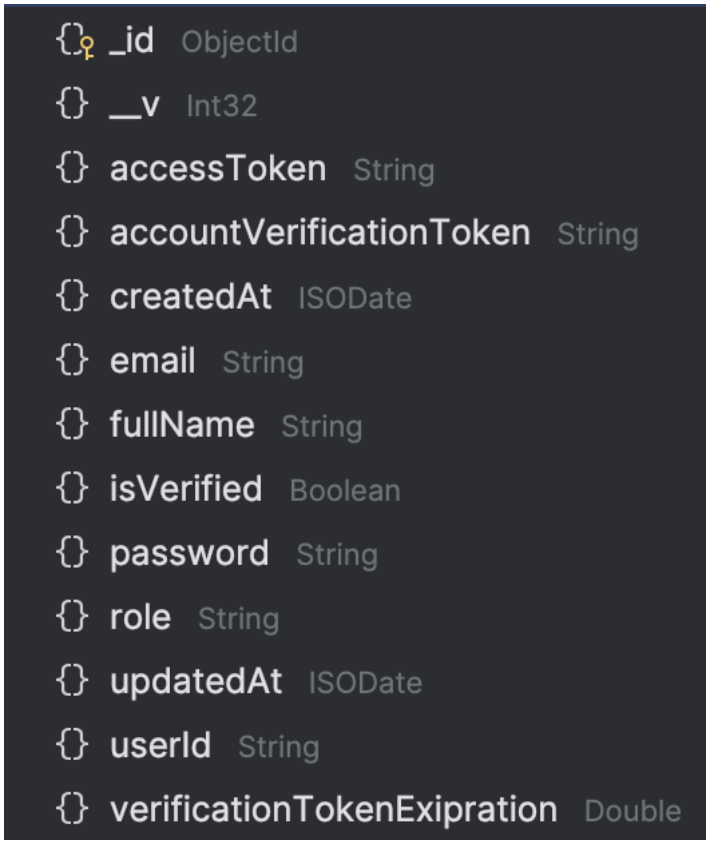

Figure 4.10: Account Schema

User Schema (`users`)

Represents user profiles. (Figure 4.11)

`email`, `fullName` (***String*, required, unique***[*]**) Basic identity fields; email is unique to enforce one profile per account.

`description` (***String***) Optional biography or profile summary.

[*]An index enforces email uniqueness at the database level.

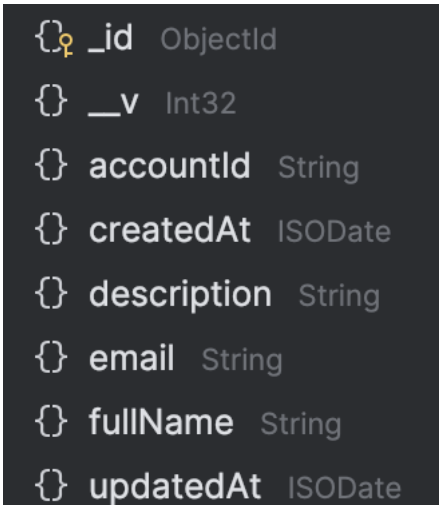

Figure 4.11: User Schema

Conversation Schema (`conversations`)

Defines chat groups and direct-message threads. (Figure 4.12)

`participants` (***[ObjectId]***, **required**) List of user IDs involved in the conversation.

`title`, `description` (***String***) Optional metadata for group chats or topic summaries.

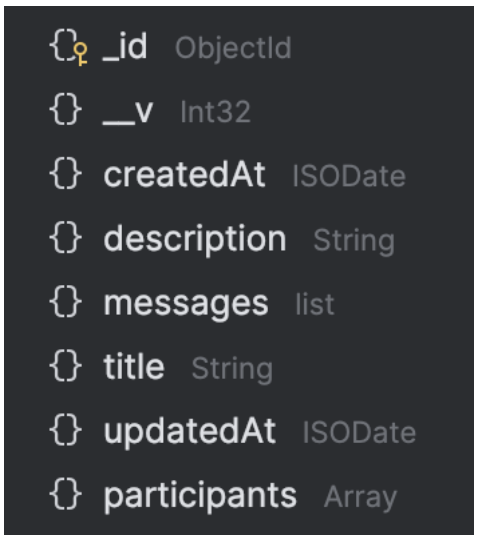

Figure 4.12: Conversation Schema

Message Schema (`messages`)

Stores individual chat messages within conversations. (Figure 4.13)

`createdBy` (***ObjectId***, **required**) Authoring user‘s ID.

`conversationId` (***ObjectId***, **required**) Parent conversation reference.

`text` (***String*, required**) Message content.

`seenBy` (***[ObjectId]*, default: `[]`**) IDs of users who have read the message.

`reactions` (***[String]*, default: `[]`**) Emoji or reaction identifiers attached by users.

`isDeleted`, `isEdited` (***Boolean*, default: `false`**) Flags for deletion and edit status.

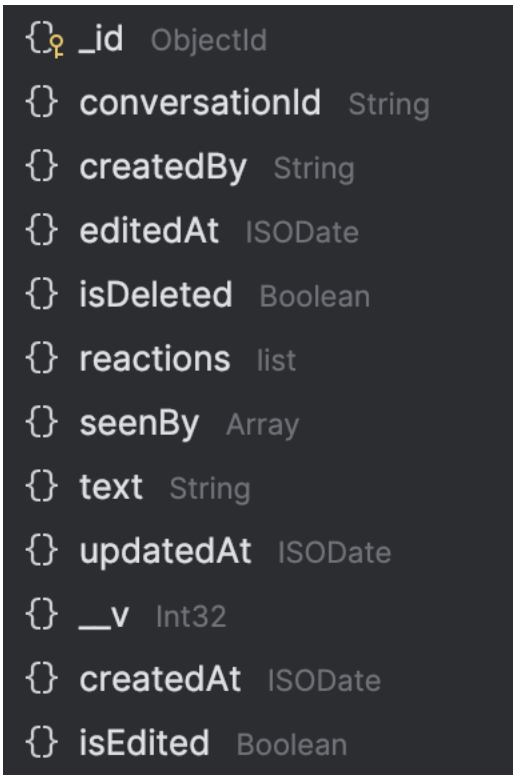

Figure 4.13: Message Schema

Notification Schema (`notifications`)

Captures system alerts and user-to-user notifications. (Figure 4.14)

`sendTo` (***[ObjectId]*, required**) Recipients of the notification.

`topic` (***String*, enum**) Notification category (e.g. new message, new reaction to message).

`data` (***Object***) Payload containing contextual information (e.g. IDs, message previews).

`isSeen` (***Boolean*, default: `false`**) Read/unread flag.

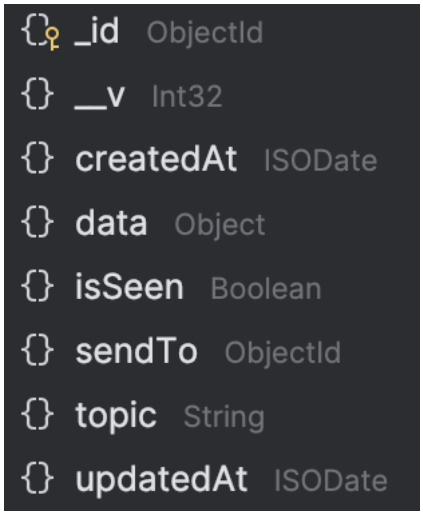

Figure 4.14: Notification Schema

File Metadata Schema (`files`)

Models additional data for uploaded files. (Figure 4.15)

`name` **(*String*, required)** Original file name as provided by the user.

`description` **(*String*)** Optional description or caption for the file.

`size` **(*Number*, required)** Size of the file in bytes.

`mimeType` **(*String*, required)** MIME type of the file (e.g., image/jpeg, application/pdf).

`fileURL` **(*String*, required)** URL pointing to the file in the S3-compatible object storage.

`uploadedBy` **(*ObjectId*, required)** Reference to the user who uploaded the file.

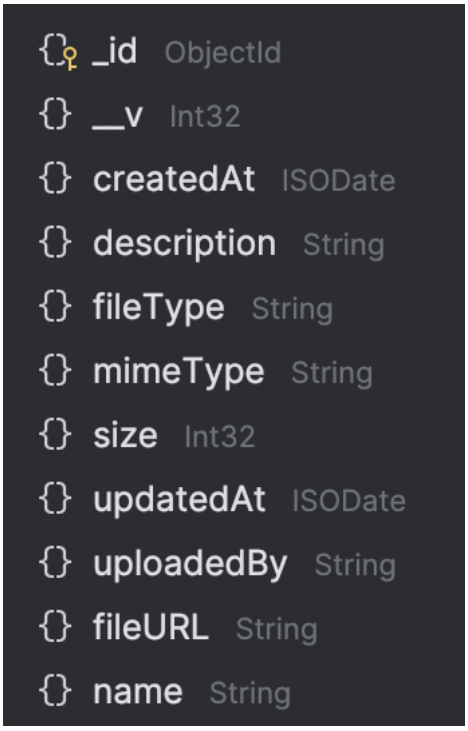

Figure 4.15: File Metadata Schema

4.3.4 Indexes and Performance

In order to ensure low-latency queries, I carefully defined several indexes across these different collections.

Indexes

- **Unique Indexes**
  - `accounts.email`: enforces one account per email address.
  - `users.email`: guarantees unique user profiles.
- **Single-Field Indexes**
  - `messages.conversationId`: accelerates retrieval of all messages in a conversation.
  - `notifications.sendTo`: speeds up lookup of unread notifications for a given user.
- **Compound Indexes**
  - `messages.conversationId, createdAt`: supports paginated fetches of recent messages by conversation.
  - `conversations.participants`: optimizes queries for all conversations a user participates in.
- **TTL Indexes**

  - `accounts.verificationTokenExpiration`: automatically removes stale verification tokens.
  - `accounts.resetTokenExpiration`: expires old password-reset tokens without manual cleanup.

4.4 Testing strategies

For validating the correct behaviour of my functionalities, I used Jest along with NestJS's built-in testing tools to write a solid set of unit tests. I focused mainly on the core business logic—things like authentication, file uploads, and message handling. Having these tests in place helps catch bugs early and keeps the codebase stable as new features are added.

4.4.1 Framework and Configuration

- **Jest**
  - Default test runner for NestJS projects; supports mocks, spies, snapshot testing, and coverage reporting out of the box.
  - Configured via `jest.config.ts` to transform TypeScript, collect coverage, and ignore compiled output directories.
- **NestJS Testing Module**
  - Uses `Test.createTestingModule()` to instantiate a lightweight application context.
  - Allows overriding providers with mocked implementations or in-memory substitutes (e.g. in-memory MongoDB) for isolation.

4.4.2 Unit Tests

I prioritized unit tests for core functionalities, ensuring that key features behave correctly under both normal and error conditions.

- **Authentication**
  - `signing up`: verifies password hashing, user creation, and error on duplicate email.
  - `validating users`: checks credential validation logic.
  - Edge cases: invalid credentials, expired tokens, and role-based access assertions.
- **File Uploads**
  - `upload file`: tests file-type validation, size limits, and interaction with MinIO/S3 API (mocked).
  - Error scenarios: unsupported formats, storage failures, and retry logic.

- **Messages**
  - `create message`: validates conversation existence and correct document shape.
  - `find by conversations`: tests pagination, sorting by timestamp, and filtering out deleted messages.
  - Mocking of Mongoose Model methods (`save()`, `find()`, `lean()`).

4.5 Scalability and Performance Considerations

One thing I really like about using a microservice architecture is that it makes it easy to manage and scale parts of the app separately. For this project, I broke things down into eight Dockerized services—including *Authentication*, *WebSocket Server*, *Uploader*, and *Notification*. Each one runs in its own container, so I can tweak resources like CPU and memory depending on what that service needs.

For example, if messaging traffic spikes, I can spin up more WebSocket containers without touching services like the Uploader or Mailer. And when file uploads slow down, I just scale down the Upload Server to save resources.

Even though I'm not using Kubernetes or autoscaling yet, having everything containerized gives me a lot of flexibility. It also helps with fault isolation and makes deploying updates a lot simpler, since I can restart or rebuild just one service without affecting the rest of the system.

- **Fault Isolation:** A failure or performance degradation in one service (e.g. a heavy file upload in the Uploader Server) does not directly impact the availability or responsiveness of other services.
- **Independent Deployment:** Each service can be updated, rolled back, or redeployed without requiring a full system outage. This reduces planned downtime and allows performance optimizations to be rolled out rapidly.
- **Resource Specialization:** Services with very different I/O patterns—such as compute-heavy data aggregation versus I/O-bound file transfers—can be placed on node pools with hardware tailored to their needs (e.g. SSD-backed nodes for storage services, high-CPU nodes for processing).

Finally, horizontal scaling and service isolation simplify capacity planning and permit near-linear performance gains: doubling the number of replicas for a stateless service generally doubles its throughput, up to the limits of downstream dependencies (such as the database). To maximize this benefit, this application also employs deployment best practices such as:

- **Caching Layers:** Introducing in-memory caches (e.g. Redis) at strategic service boundaries to offload read-heavy operations and reduce latency.
- **Asynchronous Processing:** Offloading non-critical tasks (notifications) or resource intensive tasks (sending emails, file uploads) to message queues and background workers, smoothing out traffic spikes and decoupling end-user interactions from longer-running processes.

Together, these architectural patterns ensure that *Cloud Class* can meet demanding performance requirements today while remaining flexible enough to grow with future user populations and feature sets.

Chapter 5

Future Work

This chapter outlines planned improvements and extensions to *Cloud Class*, focusing on achieving 100% integration test coverage, adding new features and third-party integrations, and enhancing deployment strategies for greater reliability and scalability.

5.1 Integration Testing

Integration tests go beyond individual modules to validate that my microservices and supporting infrastructure collaborate seamlessly. By running full end-to-end scenarios—from incoming HTTP or message-queue requests through background-job execution and database operations—I can uncover issues such as misrouted messages, data-format mismatches, or broken service contracts before they affect real users. Achieving 100% coverage in integration tests will also give me the confidence to refactor core components, evolve data schemas, and extend APIs without fear of unintended side effects.

To put this into practice, I will:

- Set up a **Docker-Compose-based test environment** that launches all eight microservices alongside MongoDB (configured as a replica set), Redis, MinIO, and both RabbitMQ and BullMQ brokers. Automated scripts will then invoke HTTP endpoints with SuperTest and publish to queues via Newman to exercise every part of the system.

- Introduce **consumer-driven contract testing** (for example, with Pact) so that each service explicitly verifies the schemas it expects from its peers, preventing subtle API drifts and ensuring compatibility across deployments.

- Use **in-memory database instances** (such as MongoDB Memory Server and an embedded Redis) together with ephemeral queue brokers to keep tests fast, isolated, and free from interference with production data.

- Integrate all integration suites into my **GitHub Actions pipeline**, organizing tests by service domain and running them in parallel. This approach will maintain quick feedback for development while enforcing strict coverage thresholds before merging changes.

5.2 New Features and Integrations

I plan to introduce two new capabilities—Google Calendar synchronization and a built-in video call feature. In a microservice architecture, this can be as straightforward as adding two dedicated services alongside the existing ones.

5.2.1 Study Session Feature

The Study Session service will let users create, join, and manage collaborative study meetings. It will provide endpoints such as `/study-sessions` for listing upcoming sessions and `/study-sessions/:id/join` for RSVP-style participation. Internally, it will store session details—topic, start and end times, participant list—and emit events whenever sessions are created, updated, or canceled. Other services (like Calendar and Video Call) will subscribe to these events to keep in sync.

5.2.2 Google Calendar Integration

The Calendar microservice will serve as a bridge between this application and the Google Calendar API. Its responsibilities include handling OAuth2 flows, synchronizing events, and delivering calendar data to other services.

First, I will define a narrow REST API within the new service for operations such as `GET /calendar/events`, `POST /calendar/events`, and `DELETE /calendar/events`. Internally, the service will manage user credentials and tokens in a secure store, refreshing them as needed via the Google OAuth2 endpoint. A background scheduler will poll or subscribe to calendar push notifications, ensuring that any changes in a user‘s Google Calendar are reflected in my system within seconds. When a user schedules a class the Calendar service will create or update the corresponding Google Calendar entry. Error handling and retry logic will ensure resilience against transient API failures.

5.2.3 Video Call Service

The Video Call microservice will manage session setup and signaling for real-time audio and video between users. To do this, I'll expose endpoints like `POST /calls/create`, `POST /calls/join`, and use a WebSocket or Socket.IO namespace to exchange things like SDP offers, ICE candidates, and call metadata.

Behind the scenes, this service will connect to a WebRTC media server (such as Janus or mediasoup) to handle NAT traversal (using STUN/TURN) and route the actual media streams. It will also check with the Auth microservice to make sure users are allowed to join or start calls. Call details—like participants, start/end times, and whether the call was recorded—will be stored in MongoDB for later use.

By keeping all video logic in its own microservice, I can scale the video part (like TURN servers) separately from other parts of the app like chat or file uploads.

Like the others, this microservice will be built with NestJS, TypeScript, and Docker.

5.3 Deployment

Ensuring the reliable operation for my application requires a considered deployment strategy composed of three key elements: database resilience, container orchestration, and traffic management.

5.3.1 MongoDB Replication

MongoDB will be deployed as a three-node replica set to provide data redundancy and automatic failover. In the event that the primary node becomes unavailable, a secondary node will assume the primary role without manual intervention, thereby minimizing downtime. Read preferences will be configured to direct reporting and analytical queries to secondary nodes, which helps to distribute read load and preserve primary performance. Additionally, the built-in change stream functionality will be leveraged to support real-time features—such as notifications and audit logging—while avoiding unnecessary complexity in the application layer.

5.3.2 Containerization and Orchestration

All services are packaged as Docker containers to ensure consistency across development, testing, and production environments. Two orchestration platforms will be evaluated:

Docker Swarm Offers a straightforward setup with native clustering, service replication, and rolling updates. Its simplicity makes it well suited for smaller teams or early-stage deployments.

Kubernetes Provides a more comprehensive feature set, including declarative resource definitions, automated scaling through the Horizontal Pod Autoscaler, and support for stateful workloads via StatefulSets. Its extensive ecosystem—encompassing service meshes, GitOps tools, and monitoring solutions—makes it ideal for large-scale, production-grade environments.

The choice between these platforms will be guided by the desired balance between operational simplicity and the need for advanced scalability and resilience features.

5.3.3 Load Balancing and Service Replication

An ingress controller (for example, Nginx[10] or Traefik[10]) will be deployed to manage external traffic. This component will perform TLS termination, path-based routing, and distribute incoming requests evenly across available service replicas. Health checks and readiness probes will ensure that only healthy instances receive traffic, and rolling-update or canary deployment strategies will minimize user impact during releases. Finally, autoscaling policies—based on metrics such as CPU utilization, memory consumption, and application-specific indicators—will enable the system to adjust its capacity dynamically, ensuring consistent performance while optimizing infrastructure costs.

Bibliography

[1] "What is Slack?" [Online]. Available: https://slack.com/help/articles/115004071768-What-is-Slack

[2] "What is Google Classroom." [Online]. Available: https://edu.google.com/workspace-for-education/products/classroom/

[3] "What is Microsoft Teams." [Online]. Available: https://support.microsoft.com/en-us/topic/what-is-microsoft-teams-3de4d369-0167-8def-b93b-0eb5286d7a29

[4] "NutriBiochem Application." [Online]. Available: https://iubmb.onlinelibrary.wiley.com/doi/full/10.1002/bmb.20771

[5] S. Tilkov and S. Vinoski, "Using javascript to build high-performance network programs," National Institute of Standards and Technology, IEEE Internet Computing, 2010. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/5617064

[6] A. Prajapati, "Amqp and beyond," in *2021 International Conference on Smart Applications, Communications and Networking (SmartNets)*, 2021, pp. 1–6.

[7] R. Bond, *Getting Started with OAuth 2.0*, O. Media, Ed. O'Reilly Media, 2014.

[8] X. S. Qigang Liu, "Research of web real-time communication based on web socket," Sydney Institute of Language and Commerce, ShangHai University, ShangHai, China, Tech. Rep., 2012. [Online]. Available: https://www.scirp.org/journal/paperinformation?paperid=25428

[9] A. Chauhan, "A review on various aspects of mongodb databases." http://www.ijert.org/, 2109.

[10] A. Johansson, "Http load balancing performance evaluation of haproxy, nginx, traefik and envoy with the round-robin algorithm," in *Dissertation*, 2022.