SMART STUDENT HANDBOOK

Title	Name	Surname	Student No
Mr	Takudzwa	Magunda	u22599160
Mr	Reinhard	Pretorius	u22509578
Mr	Mpumelelo	Njamela	u23534932
Mr	Tanaka	Ndhlovu	u22610792
Mr	Junior	Motsepe	u22598473

Software Requirements Specification (SRS)

1 Introduction

Our vision is to create an advanced application that makes it easy for students to create, share, and find notes. The Smart Student application is designed to enhance digital note-taking and academic collaboration for students. It enables users to create and manage notebooks, which can be organized into folders for easy access. Each notebook can be divided into sections, and sections further split into individual pages, offering a structured and personalized way to capture and categorize notes.

Users can write notes from a text editor, unlocking a wide range of formatting options such as headings, bullet lists, tables, images, and other multimedia elements. To improve productivity, the application features a context-aware AI Smart Assist toolbar that appears when a notebook is opened. This assistant suggests relevant notebooks based on content similarity, allowing users to merge, duplicate, or clone them into their workspace seamlessly. The application can also change text to speech for one to listen in when they are doing something that needs their attention.

In addition, users can provide feedback by liking notebooks or providing comments on them, helping highlight high-quality content. They can also add events on the app calendar to keep track of their schedule.

The Figma prototype below illustrates our initial concept for the Smart Student Handbook interface. While still in early development and subject to change, it provides a clear foundation and preview of the intended final product.

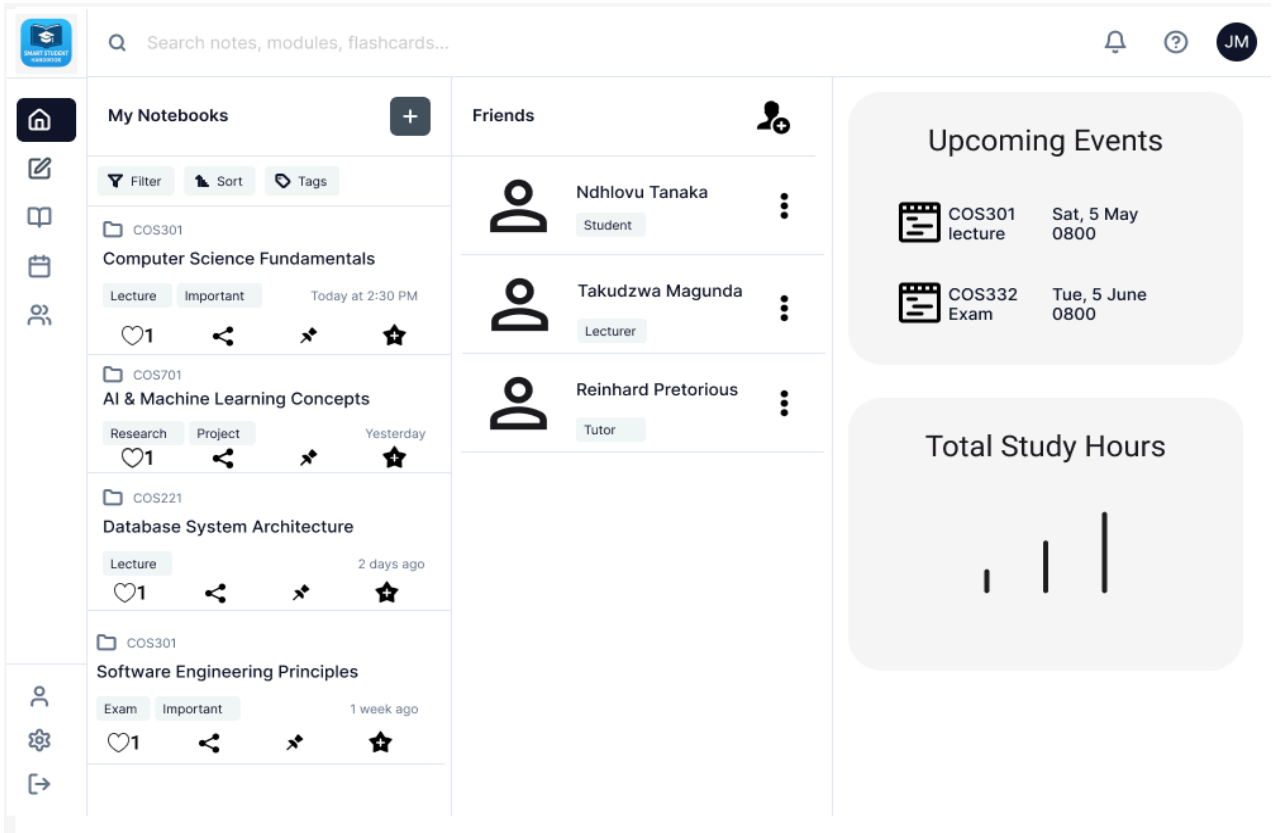


Figure 1: Envisioned Smart Student Handbook User Interface Prototype

2 User Characteristics

The primary users of the Smart Student Handbook are university students who rely on their gadgets(laptops, cellphones, tablets) as their main tool for academic engagement. These users must have an internet connection at first to sign up, but can then use the application in offline mode later.

2.1 USER

- A person who can operate a computer or smartphone
- A person who wants to create notebooks in order to take and organize notes
- A person who wants to collaborate with other students on the same notebook
- A person who wants to improve/make their notes by using other students' notes
- A person who wants to share their notes with other users
- A person who wants to keep track of their times on the app calendar
- A person who wants to create study groups with their friends
- A person who wants to search for classmates and manage friend requests (send, cancel, accept, decline)
- A person who wants to remove a friend when the friendship is over.
- A person who wants to create a public or private organisation (study group) for students.
- A person who wants to join or leave a public organisation whenever they like
- A person who, as an organisation admin, wants to add members to a private organisation
- A person who wants to favourite or unfavourite an organisation for quick access
- A person who wants to decide whether shared notes are read-only or editable by others
- A person who wants to view or edit notes shared with them, depending on those permissions
- A person who wants all friendships, notes and organisations to sync automatically when they go back online

- A person who prefers a choice between light mode and dark mode while studying

3 User Stories

3.1 USER STORY 1
As a user, I want to be able to sign up for an account and securely log in so that I can access and manage my notebooks and other features.

3.2 USER STORY 2
As a user, I want to be able to view and edit my profile details, such as my username and profile picture, so I can personalize my account to reflect my identity.

3.3 USER STORY 3
As a user, I want to be able to create notebooks and type my notes in a text formatter so I can style my notes how I want.

3.4 USER STORY 4
As a user, I want to organize my notebooks into folders (e.g., by year or module) so I can easily find them without scrolling endlessly.

3.5 USER STORY 5
As a user, I want to be able to take notes even when I am offline, so that I can continue studying during load shedding or without data, and have my notes sync when I'm back online.

3.6 USER STORY 6
As a user, I want the application to suggest relevant notebooks or snippets based on what I’m typing, so I can easily access useful and related content.

3.7 USER STORY 7
As a user, I want to rate notebooks or specific snippets and view ratings from others, so I can prioritize high-quality and trusted content.

3.8 USER STORY 8
As a user, I want to collaborate on notebooks with my friends by sharing or co-editing, so we can divide work and improve our notes together.

3.9 USER STORY 9
As a user, I want to search for notebooks related to my topics of interest, so I can explore diverse perspectives and information.

3.10 USER STORY 10
As a user, I want to use a calendar to view upcoming events, add lectures, and schedule study sessions, so I can manage my academic time effectively.

3.11 USER STORY 11
As a user, I want to add friends, view their shared notebooks, and collaborate with them, so I can learn and grow in a community environment.

3.12 USER STORY 12
As a user, I want to track the number of hours I spend studying or working, so I can monitor my productivity and improve time management.

3.13 USER STORY 13
As a user, I want to choose whether someone has **read-only** or **write** rights when I share a note, so I can keep control of my work.

3.14 USER STORY 14
As a user, I want to open notes that have been shared with me and if I have write permission,; can edit them together with the owner, so we can collaborate.

3.15 USER STORY 15
As a user, I want to search for classmates by name and send them a friend request, so I can build my academic network easily.

3.16 USER STORY 16
As a user, I want to cancel a pending friend request if I change my mind, so I don’t spam other students.

3.17 USER STORY 17
As a user, I want to accept or decline incoming friend requests, so I can control who is on my friends list.

3.18 USER STORY 18
As a user, I want to end (unfriend) an existing friendship, so I can clean up my contacts when necessary.

3.19 USER STORY 19
As a user, I want to create an organisation and choose whether it is **public** or **private**, so I can run an open study club or a closed project group.

3.20 USER STORY 20
As a user, I want to join any public organisation whenever I like, so I can take part in new study communities without waiting for an invite.

3.21 USER STORY 21
As a user, I want to leave an organisation at any time, so I’m not stuck in groups that are no longer relevant to me.

3.22 USER STORY 22
As an organisation admin, I want to add members to my private organisation after it is created, so I can grow the group without making it public.

3.23 USER STORY 23

As a user, I want to favourite an organisation, so it shows up at the top of my list for quick access.

3.24 USER STORY 24

As a user, I want to unfavourite an organisation, so I can keep my favourites list tidy.

3.25 USER STORY 25

As a user, I want to view the full details of a private organisation only if I’m a member, so private project or study group information stays confidential.

4 USE CASES

1.

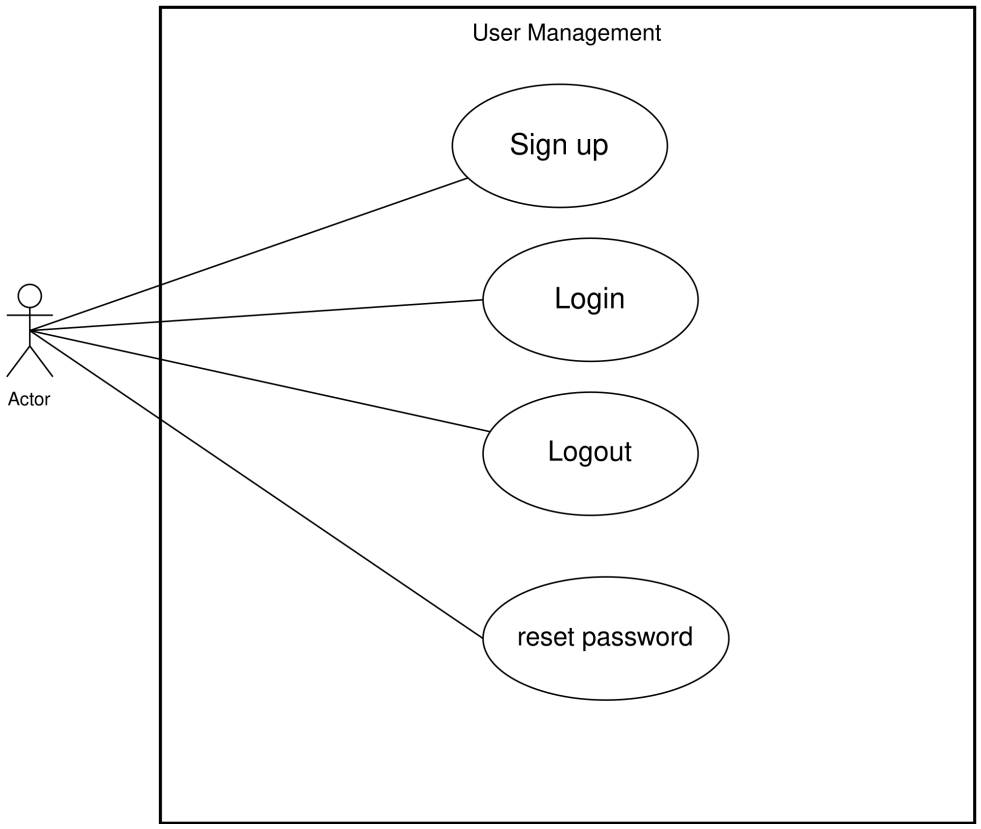


Figure 2: User Management Use Case Diagram

User Management Service Contracts:

Sign Up

- What It Does: Registers a new user in the system.
- Inputs: Username, email, password, role (e.g., "Actor").
- Outputs: Success message or error (e.g., "Email already exists").
- Interacts with: Database (to store user details)

Login

- What It Does: Authenticates a user and grants access to the system.
- Inputs: Email/username, password.
- Outputs: Authentication token or error (e.g., "Invalid credentials").
- Interacts With: Database (to verify credentials).’

Logout

- What It Does: Terminates the user's active session.
- Inputs: Authentication token.
- Outputs: Success message.
- Interacts with: Session management service.

Reset Password

- What It Does: Allows users to reset their password via email.
- Inputs: Email address.
- Outputs: Password reset link (sent to email) or error.
- Interacts with: Email service, database.

2.

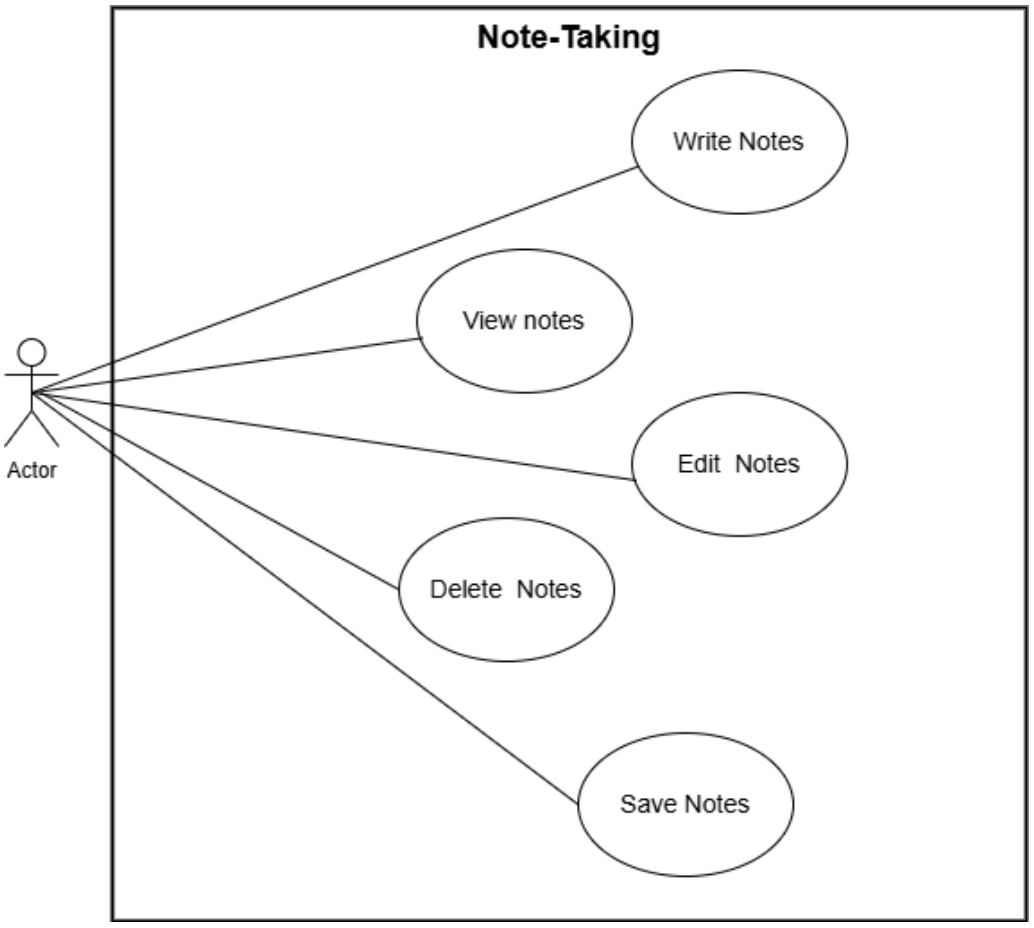


Figure 3: Note-Taking Use Case Diagram

Note-Taking Service Contracts

Write Notes

- What It Does: Creates a new note.
- Inputs: User ID, note title, content.
- Outputs: Note ID or error.
- Interacts with: Database (to save notes).

View Notes

- What It Does: Retrieves notes for a user.
- Inputs: User ID, optional filters (e.g., date).
- Outputs: List of notes or error.
- Interacts with: Database (to fetch notes).

Edit Notes

- What It Does: Updates an existing note.
- Inputs: Note ID, updated content.

- Outputs: Success message or error.
- Interacts With: Database.
-

Delete Notes

- What It Does: Removes a note permanently.
- Inputs: Note ID.
- Outputs: Success message or error.
- Interacts With: Database.

Save Notes

- What It Does: Auto-saves note changes.
- Inputs: Note ID, partial/full content.
- Outputs: Success message or error.
- Interacts With: Database.

3.

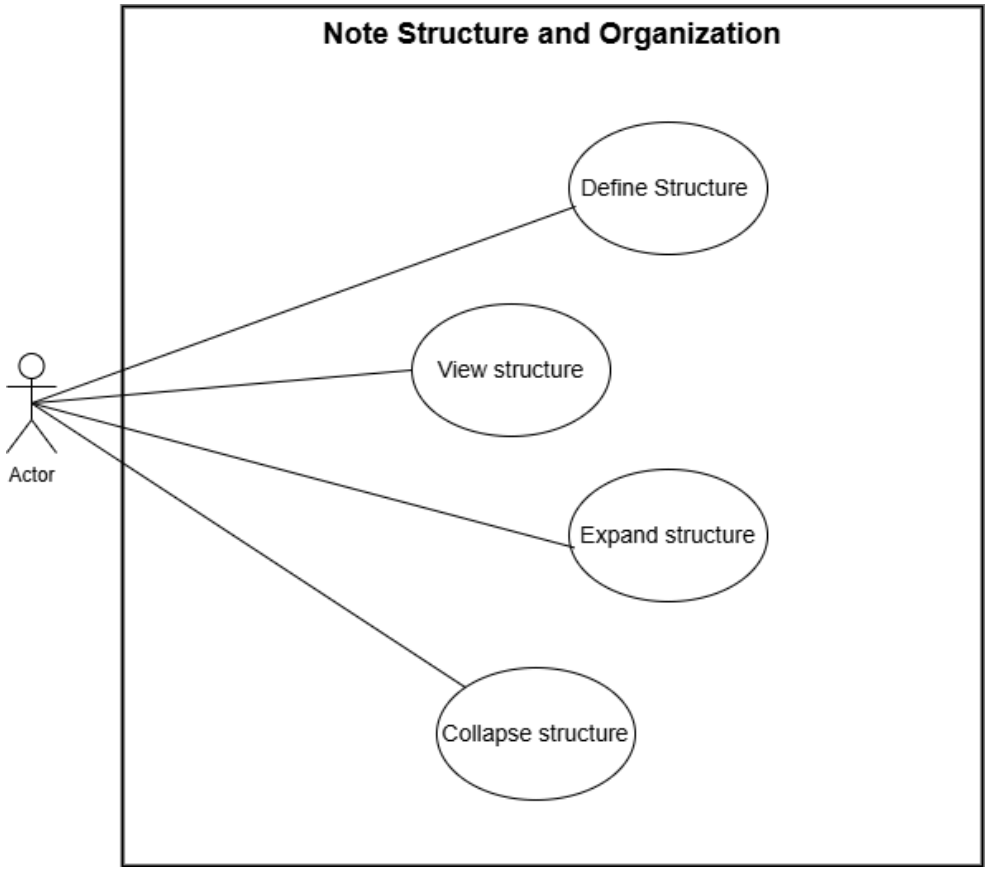


Figure 4: Note Structure and Organisation Use Case Diagram

Note Structure and Organisation Service Contracts

Define Structure

- What It Does: Creates a hierarchical structure for notes (e.g., folders).
- Inputs: User ID, structure name, parent ID (optional).
- Outputs: Structure ID or error.
- Interacts With: Database.

View Structure

- What It Does: Displays the note hierarchy.
- Inputs: User ID.
- Outputs: Tree-like structure of notes or error.
- Interacts With: Database.

Expand/Collapse Structure

- What It Does: Toggles visibility of substructures.
- Inputs: Structure ID, action ("expand" or "collapse").
- Outputs: Updated UI state.
- Interacts With: Frontend components.

4.

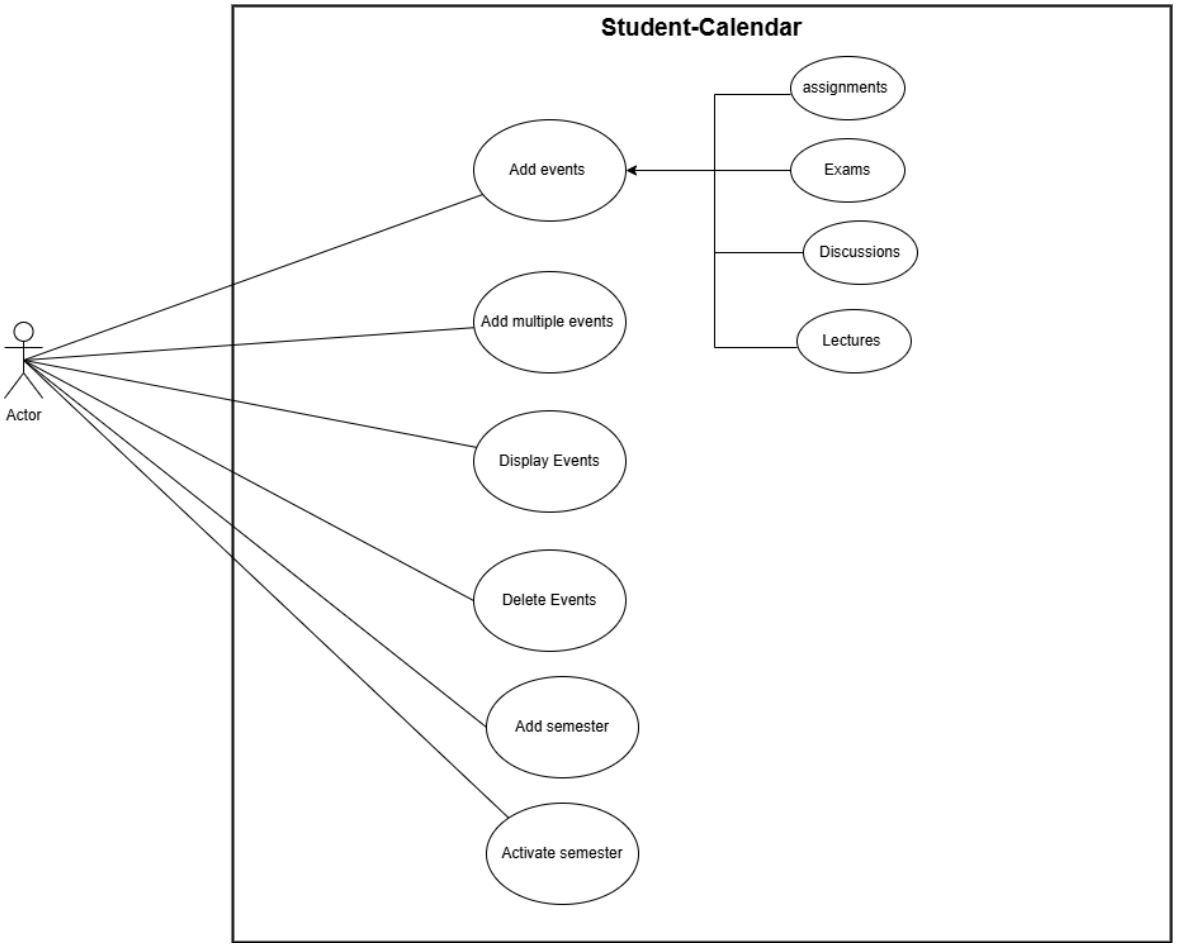


Figure 5: Student-Calendar Use Case Diagram

Student-Calendar Service Contracts

Add Events

- What It Does: Schedules a new event (e.g., lecture).
- Inputs: User ID, event title, date/time, type (e.g., "Exam").
- Outputs: Event ID or error.
- Interacts With: Database.

Display Events

- What It Does: Shows events for a given timeframe.
- Inputs: User ID, date range.
- Outputs: List of events or error.
- Interacts With: Database.

Delete Events

- What It Does: Removes an event.
- Inputs: Event ID.
- Outputs: Success message or error.
- Interacts With: Database.

Add/Advise Semester

- What It Does: Creates or updates semester plans.
- Inputs: User ID, semester name, courses.
- Outputs: Semester ID or error.
- Interacts With: Database.

5

Figure 6: Profile-Management Use Case Diagram

Profile-Management Service Contracts

View/Edit Details

- What It Does: Displays or updates user profile data.
- Inputs: User ID, updated fields (e.g., bio).
- Outputs: Profile data or error.
- Interacts With: Database.

View Study Hours/Badges

- What It Does: Shows analytics or achievements.
- Input: User ID.
- Outputs: Study hours/badges list or error.
- Interacts With: Analytics service.

View/Remove Friends

- What It Does: Manages friend connections.
- Inputs: User ID, friend ID (for removal).
- Outputs: Friend list or success message.
- Interacts With: Social graph service.

Change Password

- What It Does: Updates user password.
- Inputs: User ID, old password, new password.
- Outputs: Success message or error.
- Interacts With: Database.

6.

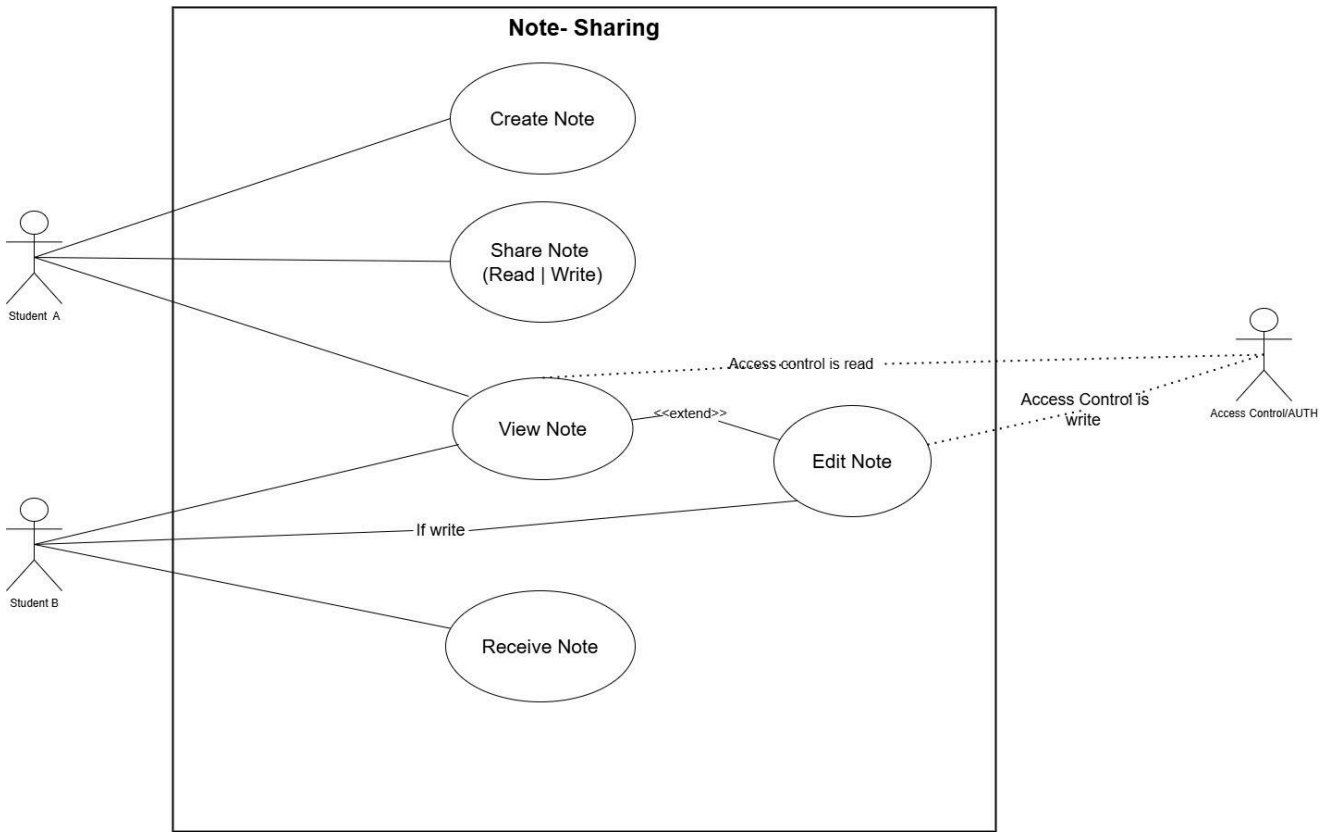


Figure 7 : Note-sharing use case diagram

<u>Note Sharing Service Contracts</u>
Create Note

- What It Does: Saves a new note under the owner's account.
- Inputs: Owner ID, title, content.
- Outputs: Note ID or error.
- Interacts With: Database.

Share Note

- What It Does: Grants *read* or *write* access to another student.
- Inputs: Owner ID, Note ID, Target-User ID, permission (read | write).
- Outputs: Success message, Access control entry created, target can view the shared note.
- Interacts With: Database.

Receive/View Note

- What It Does: Target can view the shared note.
- Inputs: Target-User ID, Note ID.
- Outputs: User can view the note or error.
- Interacts With: Database.

Edit Note

- What It Does: Saves changes when permission is *write*.
- Inputs:Editor ID, Note ID, updated content.
- Outputs: “Saved” confirmation or “read-only” error.
- Interacts With: Database.

7

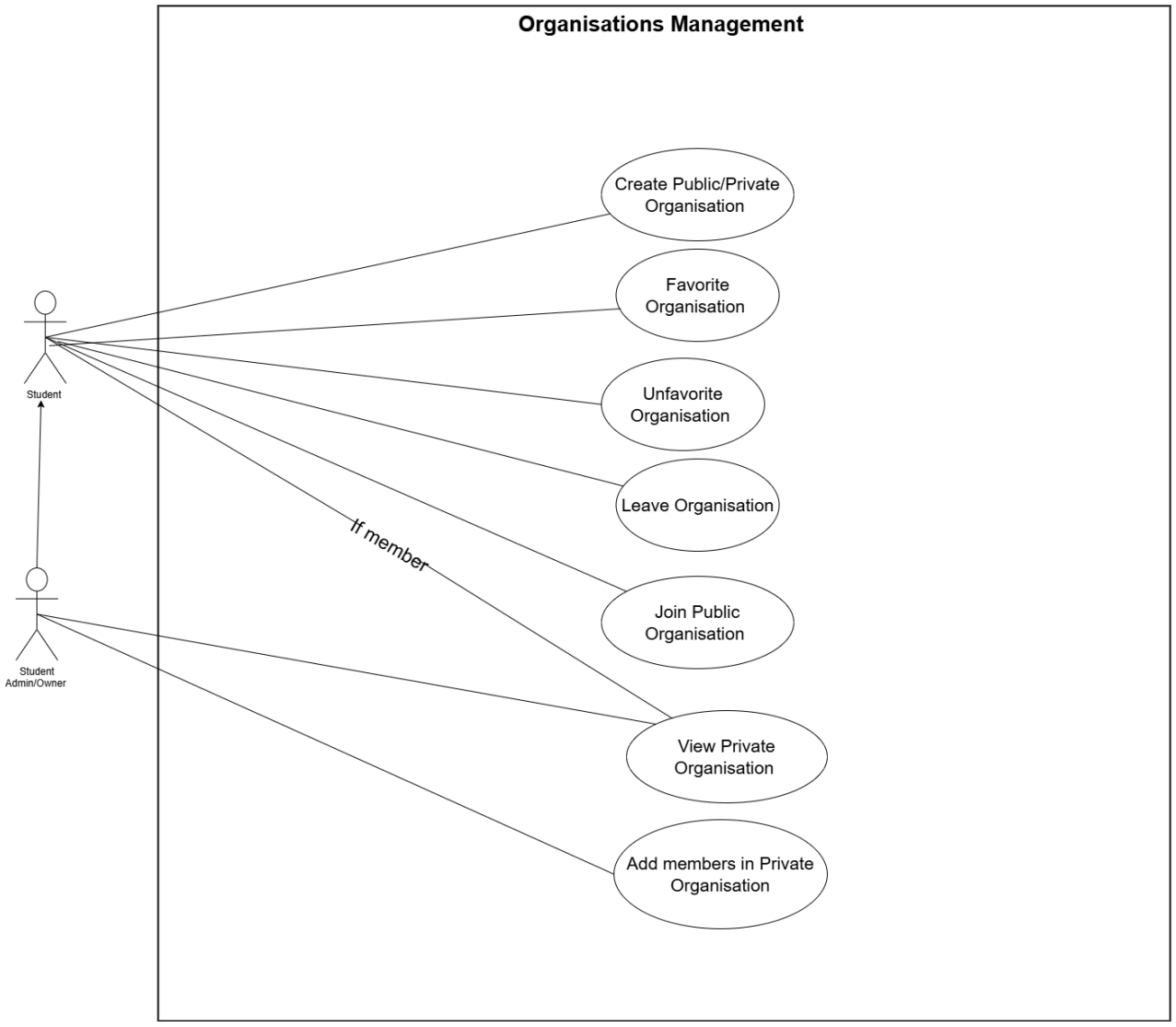


Figure 8: Organisations management use case diagram

Organisations Management Service Contracts
Create Organisation

- What It Does: Creates a new public or private organisation; creator of private organisation becomes admin.
- Inputs: Owner ID, Name,organisation icon ,description ,visibility (public | private).
- Outputs: Organisation ID or error ("name taken").
- Interacts With: Database (organisation collection).

Add Member (private organisation)

- What It Does: Admin adds a student to a private organisation.
- Inputs: Admin ID, Org ID, New-Member ID.
- Outputs: Success or "not admin" error; member notified.
- Interacts With: Database.

Join Public Organisation

- What It Does: Allows any student to join a public organisation.
- Inputs:User ID, Org ID.
- Outputs: Success.
- Interacts With: Database.

Leave Organisation

- What It Does: Removes the student from the member list.
- Inputs:User ID, Org ID.
- Outputs:Success message.
- Interacts With: Database.

Favourite Organisation

- What It Does: Pins the organisation for quick access.
- Inputs:User ID, Org ID.
- Output: Favourite badge confirmation and heart turns red.
- Interacts With: Database.

Unfavourite Organisation

- What It Does: Removes the pin.
- Inputs:User ID, Org ID.
- Output:heart turn white, favourite badge is removed on card.
- Interacts With: Database.

View Private Organisation

- What It Does: Displays org details only if the user is a member.
- Inputs:User ID, Org ID.
- Output:the organisation card.
- Interacts With: Database.

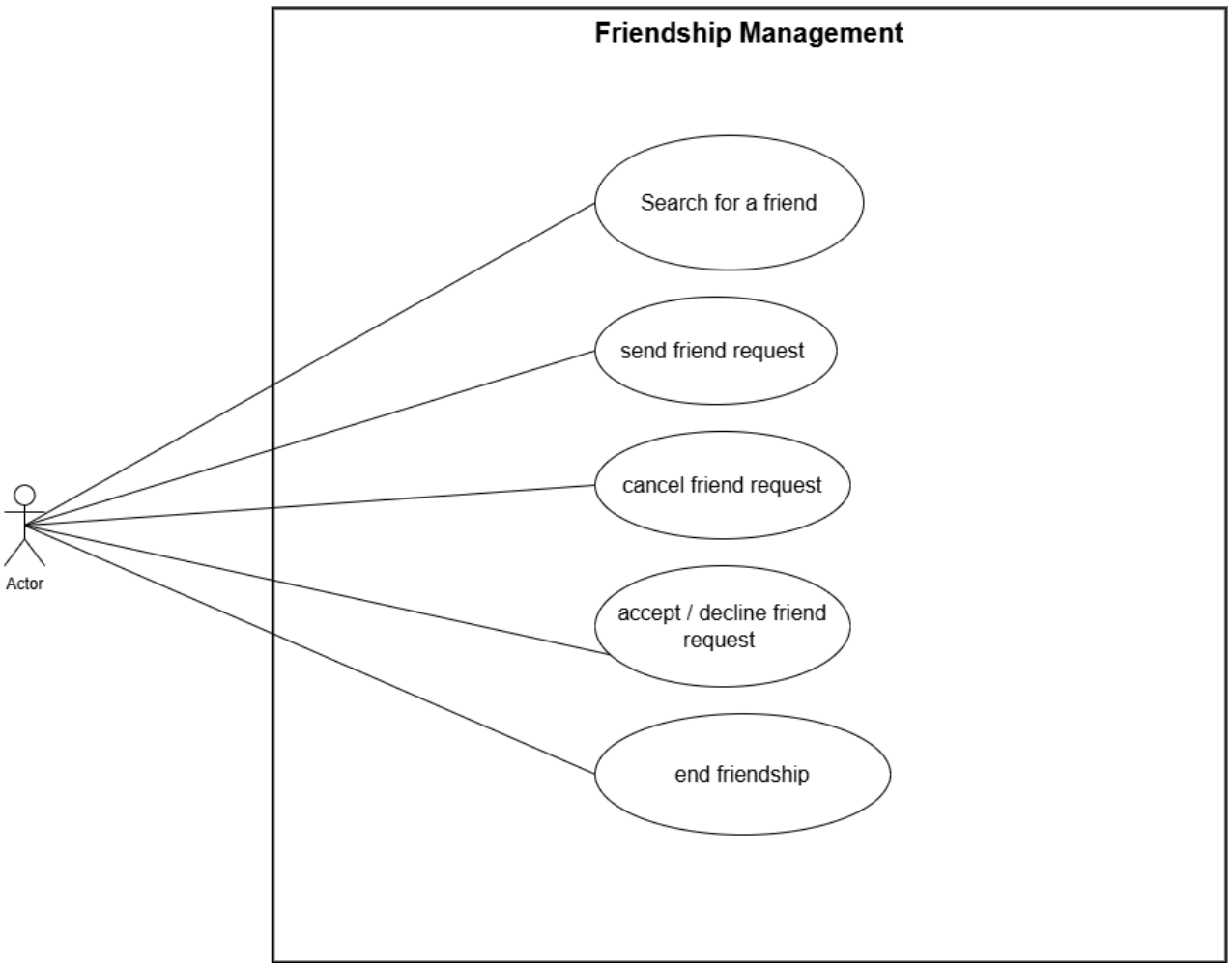


Figure 9 : friendship management use case diagram

Friendship Management Service Contracts
Search for Friend

- What It Does: Finds other students by name or email.
- Inputs: Current user ID, search text.
- Outputs: List of matching names or "no matches".
- Interacts With: Database.

Send Friend Request

- What It Does: Sends a pending friend request.
- Inputs: Sender ID, Receiver ID.
- Outputs: Request ID or error ("already friends"/"request sent").
- Interacts With: Database.

Cancel Friend Request

- What It Does: Withdraws a still-pending request.
- Inputs: Request ID, Sender ID.
- Outputs:request removed from receiver's list.
- Interacts With: Database.

Accept / Decline Friend Request

- What It Does: Lets the receiver accept or reject a request.
- Inputs:Request ID, Receiver ID, action (accept | decline).
- Outputs: friendship column and request column updated.
- Interacts With: Database.

End Friendship

- What It Does: Removes an existing friendship.
- Inputs:User ID, Friend ID.
- Outputs:Success message or error.
- Interacts With: Database.

4 FUNCTIONAL REQUIREMENTS:

- **R1:** The system should allow users to create accounts and sign in
- **R2:** The system should allow users to manage their Profile
  - **R2.1** The system should allow the user to update their personal information(name, surname, profile picture)
  - **R2.2** The system should allow a user to change their password
- **R3:** The system should allow users to create notebooks and notes
  - **R3.1:** The system should allow a user to create a notebook
  - **R3.2:** The system should allow a user to create folders inside a notebook
  - **R3.3:** The system should allow a user to create notes inside folders
- **R4:** The system should allow a user to share notes
- **R5:** The system should allow a user to search and see shared notes
  - **R5.1:** The system should allow a user to clone shared notes
  - **R5.2:** The system should allow a user to like shared notes
- **R6:** The system should allow a user to collaborate with friends on a notebook
- **R7:** The system should allow a user to make and view notes offline
- **R8:** The system should allow a user to access the app calendar
  - **R8.1:** The system should allow the user to see all upcoming events on the calendar
  - **R8.2:** The system should allow the user to add an event to the calendar
- **R9:** The system should allow users to share notes and enforce access-control on the shared notes.
  - **R9.1:** The system should allow a note owner to grant read-only or write permission to each invited user.
  - **R9.2:** The system should prevent users who have only read permission from editing the note.
  - **R9.3:** The system should display the current permission level (read or write) to every user viewing a shared note.
- **R10:** The system should allow users to manage friendships.
  - **R10.1:** The system should allow a user to search for other students by name.
  - **R10.2:** The system should allow a user to send a friend request to another user.
  - **R10.3:** The system should allow a user to cancel a pending friend request.
  - **R10.4:** The system should allow a user to accept or decline an incoming friend request.
  - **R10.5:** The system should allow a user to remove (unfriend) an existing friend.
  - **R10.6:** The system should display an up-to-date list of the friends list.
- **R11:** The system should allow users to create and participate in organisations (study groups or project groups).
  - **R11.1:** The system should allow a user to search for other students by name.
  - **R11.2:** The system should allow any user to join a public organisation without an invitation.
  - **R11.3:** The system should allow a user to leave an organisation at any time.

- **R11.4:** The system should allow an organisation admin to add members to a private organisation after it has been created.
- **R11.5:** The system should allow a user to favourite or unfavourite an organisation for quick access.
- **R11.6:** The system should restrict access to the details of a private organisation to its members only.

5 Architectural Requirements

The Smart Student Handbook is a progressive-web application that keeps students on top of timetables, coursework, study groups and study buddies all in one place. A Next.js front-end provides an app-like experience while Firebase Cloud Functions (TypeScript) power the back-end services. This document records the architecture design strategy, quality goals and the patterns.

5.1Architecture Design Strategy

5.1.1 Decomposition

For the Smart Student Handbook we follow a decomposition strategy that is based on quality-driven design and enforced by test-first development. The back-end codebase is already split into Firebase Function modules. Organisations and Calendar and friends API route handlers. Each functionality is divided in its own folder, a group member can test and change functionality for one area without touching the rest, this matches our sprint rhythm. Every structural choice is traced to a quality requirement.

5.2 Architectural Strategies

Three architectural styles shape the solution. First, a layered structure separates presentation from application and service/API routes and finally the Data tier (Realtime Database). Second, we apply service-oriented decomposition inside the API layer: each cloud function set owns exactly one business domain and the Realtime-Database subtree that belongs to it, so scaling or replacing one domain never affects another. Third, we use an event-driven strategy that links the services. These three styles together supply the loose coupling, parallel development and low latency that we need for the handbook.

5.3Architectural Design and Patterns

A browser loads server-rendered Next.js pages that hydrate into React components. Those components call Api routes/firebase functions. Inside the function module a small repository performs the Realtime-Database read or write, publishes an optional Pub/Sub message and returns a JSON transfer object. This flow embodies several patterns: we have the edge route that acts as a **Facade** shielding the UI from Firebase details; each domain function is a repository around its own data; Pub/Sub implements the **Observer** pattern; the Admin SDK is initialised exactly once per container, following the **Singleton** pattern. We keep a very small duplicate index called a pointer tree. When a student joins an organisation we add the full record under organisations and also drop a tiny flag under userOrganizations/uid/orgId. That extra flag costs almost nothing to write but it lets us list all organisations for a student with one quick read instead of searching the whole collection. All of this data together with the Cloud Functions that use it lives in us-central1 so everything runs in the same region and stays fast.

5.4 Architectural Constraints

Several factors dictate our architecture. The project now runs on the Blaze plan, but we still treat cost like a student budget setting billing alarms and designing for the free-tier thresholds we grew up with. All code is written in TypeScript—the language the whole team knows. We can't use virtual machines, so we depend entirely on serverless services: Vercel hosts the Next.js front-end and Firebase Cloud Functions run the back-end. Every resource (Functions, Realtime DB, Storage) lives in us-central1. We keep data in one region and avoid cross-zone latency charges. Because we perform multiple demos, the system is sliced into independent vertical features that can ship to production without touching the rest of the stack.

5.5 Technology Choices

Front End

We compared the following frameworks:
-Angular
- React (with Next.js 13)
-Vue (with Nuxt 3).

Angular offers an all in one too chain and strong enterprise patterns, but its has a heavier bundle size and steep learning curve make iteration slower for a small student team. Vue/Nuxt provides an elegant syntax and built-in SSR, but we all don't have production Vue experience. **React with Next .js 13** was the clear fit: everyone on the team already writes React, Next.js supplies hybrid SSR/ISR out of the box, and Vercel's serverless hosting keeps first-paint latency low even on unreliable campus Wi-Fi. React therefore balances skill familiarity, performance, and our serverless constraint better than the alternatives.

Styling / UI Kit

Options considered were:
-ShadCN + Tailwind
- Chakra UI and Material-UI.

ShadCN with Tailwind won because it delivers a tiny, tree-shakable CSS payload and stays fully type-safe in our all-TypeScript codebase. Chakra and MUI works for far larger bundles and would have pushed us over our performance budget.

Back-End

We compared three server-side options:

-Firebase Cloud Functions v2
-Cloud Run
-AWS Lambda

Firebase Cloud Functions inherit Firebase Auth context automatically, scale to zero without Docker maintenance and work inside our blaze account , this is perfect for our student workload. Cloud Run gives full container freedom but would force us to build and patch images, manage secrets by hand, and handle cold-start tuning

ourselves.

AWS Lambda has a lot of features but it lives in a different cloud, which would introduce cross-cloud latency to Firebase data and duplicate IAM and monitoring setups. Cloud Functions meet our serverless implementation with the least operational effort and Firebase integration, they are the optimal choice for Smart Student Handbook.

Database

We evaluated:
- Firebase Realtime Database, Firestore
- Supabase Postgres.

Realtime DB offers millisecond latency, WebSocket updates and the lowest cost for write, this is critical for collaborative timetable edits for example. Firestore adds richer querying but higher read fees, while Postgres would break the no-server rule. We choose the firebase realtime Database.

5.6 Quality Requirements

Q1 Performance

1.1 Routine notebook actions (create, read, update, delete) should complete in under one second, even under heavy load.

1.2 Synchronizing data across devices must finish within five seconds by batching updates efficiently.

1.3 More complex database queries must return results in under ten seconds by leveraging optimized, indexed queries.

Q2 Scalability

2.1 The platform must handle a minimum of 55,000 concurrent active users, each can have multiple notebooks, and the system should not slow down.

2.2 It should support seamless scaling both by upgrading server resources and by adding additional instances to meet demand spikes

2.3 Auto-scaling mechanisms must dynamically adjust capacity as the number of users increases or decreases.

Q3 Storage

3.1 Our system must handle ever-increasing volumes of notes and attachments without large data costs or hurting performance.

3.2 Before saving, all user-uploaded images and large files will be automatically compressed to reduce storage use and speed up access.

3.4 We reserve at least **10 to 15 %** of our total database space to be able to absorb unexpected spikes in data.

Q4 Availability

Q4.1 Annual downtime must stay under 2.2 hours and should not increase by more than 8.5% each year

Q4.2 During planned maintenance, at least 80 % of services must continue running so students can still access their notes.

Q5 Reliability

5.1 The application should run stably without unhandled crashes; any error must trigger a captured report with context, allowing us to quickly fix the error.

5.2 User notebooks must be protected from data loss—automatic version backups and rollback procedures should aid in this.

5.3 Service deployments must eliminate single points of failure through redundancy.

Q6 Security

6.1 Access controls must ensure that only authenticated and authorised users can read or write a given notebook unless given access through collaboration.

6.2 User data should be encrypted to ensure that users' anonymity is kept.

6.3 Mechanisms to reset all user passwords if a breach is required.

5.7 Architectural Patterns

Layered Architecture:

- **Presentation Layer:** React/Next.js frontend for user interaction and Shadcn for structured components.
- **Application Layer:** Spring Boot/NestJS backend for business logic
- **Data Layer:** Firebase for dynamic structure of notes
- **AI Layer:** Lightweight LLM integration for smart assist features

Microservices Pattern:

- Authentication Service: User management and security
- Note Management Service: Core note operations
- AI Processing Service: Machine learning and recommendations
- Collaboration Service: Real-time sharing and editing
- Search Service: Advanced search and filtering capabilities

5.8 General Constraints

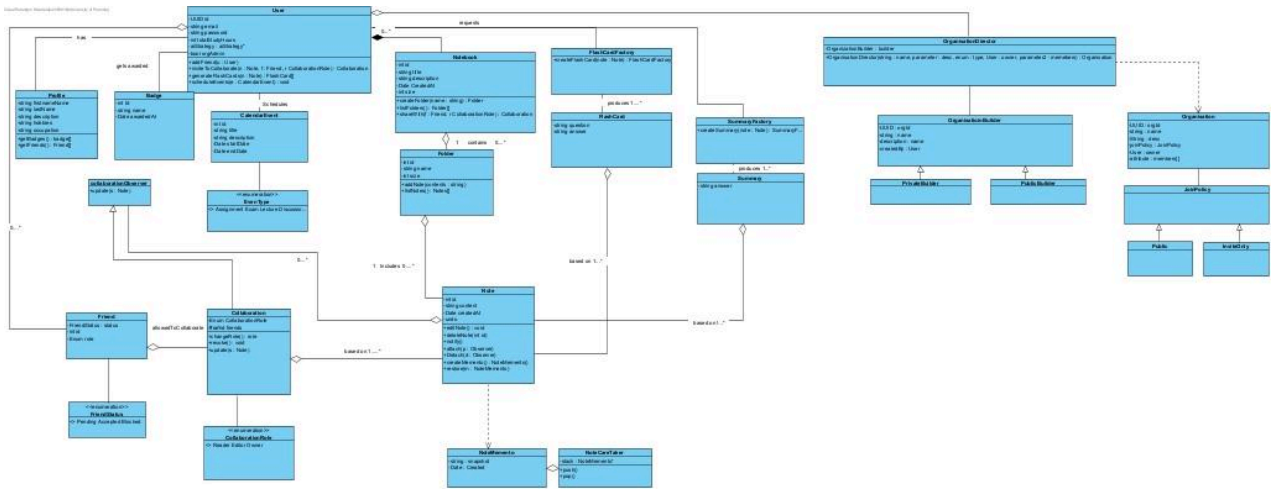
Technical Constraints:

- Must integrate with existing university authentication systems
- AI processing must remain within budget constraints using free-tier services
- Mobile responsiveness is required for cross-device compatibility
- Offline functionality is limited to read-only operations initially

Business Constraints:

- Development timeline: 12 weeks (academic semester)
- Budget limitations: Utilize free-tier cloud services where possible
- Team expertise: Student-level experience requiring learning curve consideration
- Regulatory compliance: Must adhere to educational data privacy requirements

6. Updated Domain Model



DomainModel.jpg

7. Design Patterns

1.Composite (nested notebooks and folders)
Notebook, Folder, and Note all implement a common Component interface so operations such as size calculation, sharing or exporting work exactly the same whether a student selects a single note or an entire notebook. The whole structure forms one uniform tree, eliminating type checks and duplicate logic.

2.Observer (real-time collaboration updates)
A Note plays the role of subject. When the note changes, it calls notify(), which immediately pushes updates to collaborators without any hard-wired dependencies inside the editing code.

3.Memento (safe undo and redo)
Before a note is modified, it captures its current state in a

NoteMemento object and pushes that snapshot onto a stack managed by a caretaker. Later, restore(memento) rolls the content back while still keeping the internals of Note encapsulated. This gives users unlimited, trustworthy undo and redo.

4.Factory Method(generating study aids)
A shared ContentFactory interface hides the concrete creation of study aids like FlashCard or Summary. When we want to introduce new study aids such as mind-maps, we simply provide another factory class. The user interface and service layers continue to work unchanged.

5.State
The Friend class stores its current FriendStatus, which can be Pending, Accepted, or Blocked. Behaviour changes internally based on that status.

8. Technology Requirements

Frontend Technologies

- **Framework:** Next.js with React for server-side rendering and SEO optimization
- **Language:** TypeScript for type safety and maintainable code
- **Styling:** TailwindCSS for rapid, responsive UI development
- **Component Library**: Shadcn for prebuilt, customizable UI components
- **State Management:** Redux Toolkit for complex state management

Backend Technologies

- **Framework:** Spring Boot (Java) or NestJS (TypeScript) for robust API development
- **API Design:** RESTful API with OpenAPI documentation
- **Authentication:** JWT tokens with a refresh token mechanism
- **Real-time:** WebSocket for collaborative features

Database Technologies

- **Primary Database:** Firebase for dynamic structured notes
- **File Storage:** Firebase Storage

AI/ML Technologies

- **ML Framework:** Hugging Face Transformers for pre-trained models
- **Model:** DistilBERT for lightweight NLP processing

- **Processing:** Python-based AI service for recommendation generation
- **Integration:** REST API for AI service communication

Cloud Infrastructure

- **Platform:** AWS for scalable cloud infrastructure
- **Compute:** EC2 instances with auto-scaling capabilities
- **Serverless:** Lambda functions for event-driven processing
- **CDN:** CloudFront for global content delivery

Development Tools

- **IDE:** Visual Studio Code with relevant extensions
- **Containerization:** Docker for consistent development environments
- **Version Control:** Git with GitHub for code management
- **CI/CD:** GitHub Actions for automated testing and deployment
- **Testing:** Jest for unit testing, Cypress for end-to-end testing
- **Design:** Figma for UI/UX design and prototyping

Monitoring and Analytics

- **Application Monitoring:** AWS CloudWatch for system metrics

- **Error Tracking:** Sentry for error monitoring and debugging
- **Analytics:** Custom analytics dashboard for user engagement tracking
- **Performance:** Lighthouse for web performance optimization

9. Service Contracts

Technology Stack

- Frontend
  Next.js / TypeScript 4
  ShadCN UI / Tailwind CSS
  Recharts

- Backend
  Firebase Realtime Database
  Firebase Authentication
  Firebase Cloud Functions — TypeScript (Node.js runtime)
  Express.js (REST gateway, written in TypeScript)

- Testing
  Jest

  React Testing Library
  Cypress

- Data Storage
  Firebase Realtime Database – NoSQL
  Firebase Cloud Storage (for media)

10.Deployment model

local dev

- run `npm run dev` to view the Next .js front-end on localhost
- start Firebase emulators with `firebase emulators:start` (functions, auth, database)
- set `NEXT_PUBLIC_FIREBASE_EMULATOR=true` so the browser sends requests to the emulators

preview per pull request (2025 workflow)

- GitHub Actions installs deps, runs Jest and Playwright
- if tests pass, the job runs `next build && next export` to generate static files
- the CLI posts the unique preview URL on the PR; channel deletes itself after 7 days

production

- merge to `main` triggers another GitHub Actions job
- job builds the site and executes `firebase deploy --only hosting,functions --project prod`
- static site goes to `studenthandbook-a215a.web.app`
- Cloud Functions v2 roll out in `us-central1`, next to Realtime DB

rollback

- `firebase hosting:rollback <version>` restores an earlier UI release in seconds
- `firebase functions:rollback --to <id>` brings back a previous function version

monitoring

- Cloud Logging streams function logs; errors surface in the Firebase console
- Cloud Monitoring budget alert emails the team if Blaze spend crosses the set cap

why it works for us

- one cloud (Firebase) keeps the stack simple, no VMs to manage
- preview channels let lecturers test every sprint without touching prod
- us-central1 co-locates Functions and Realtime DB, so latency stays low and billing clear