Commute Buddy

System Design Document

Sprint 3

A commute matching platform for York University students

EECS 3311 – Software Design

Fall 2025

Team Members:

Ashraf (Product Owner) • Shaun • Ali • Sathmi

Version 3.0 — December 1, 2025

Table of Contents

1. Overview

Commute Buddy is a web application built for York University students who want to find commute partners. The idea is simple: students who take similar routes or have overlapping schedules can connect, coordinate, and travel together. This makes commuting safer, more social, and sometimes even cheaper when splitting costs.

The platform handles everything from user registration (restricted to York email addresses) to real-time messaging between matched students. We built it using the MERN stack—MongoDB for data storage, Express and Node.js on the backend, and React on the frontend.

What's New in Sprint 3

This sprint focused heavily on communication features. The major additions include:

- **Real-time messaging** – Users can now chat directly with their commute buddies. Messages appear instantly without refreshing the page, thanks to Socket.io.
- **Push notifications** – When someone sends you a request or message, you get a notification immediately. There's a bell icon in the navbar that shows unread counts.
- **Better filtering** – You can now filter potential matches by faculty, program, and availability window, not just routes and times.
- **Session improvements** – We added rate limiting on login attempts, automatic logout after an hour of inactivity, and silent token refresh so active users don't get logged out unexpectedly.

2. System Architecture

We went with a three-tier architecture, which is pretty standard for web apps like this. The presentation layer (React frontend) handles what users see and interact with. The business logic layer (Express/Node backend) processes requests and enforces rules. The data layer (MongoDB) stores everything persistently.

What makes our setup a bit different is the real-time layer we added for Sprint 3. Socket.io runs alongside the REST API on the same server, handling WebSocket connections for instant messaging and notifications. This means we're not purely request-response anymore—the server can push updates to clients whenever something relevant happens.

Architecture reference: This follows the layered architecture pattern described in Fowler's *Patterns of Enterprise Application Architecture*, extended with WebSocket support as documented at socket.io/docs.

2.1 Architecture Diagram

The diagram below shows how the different parts of the system connect:

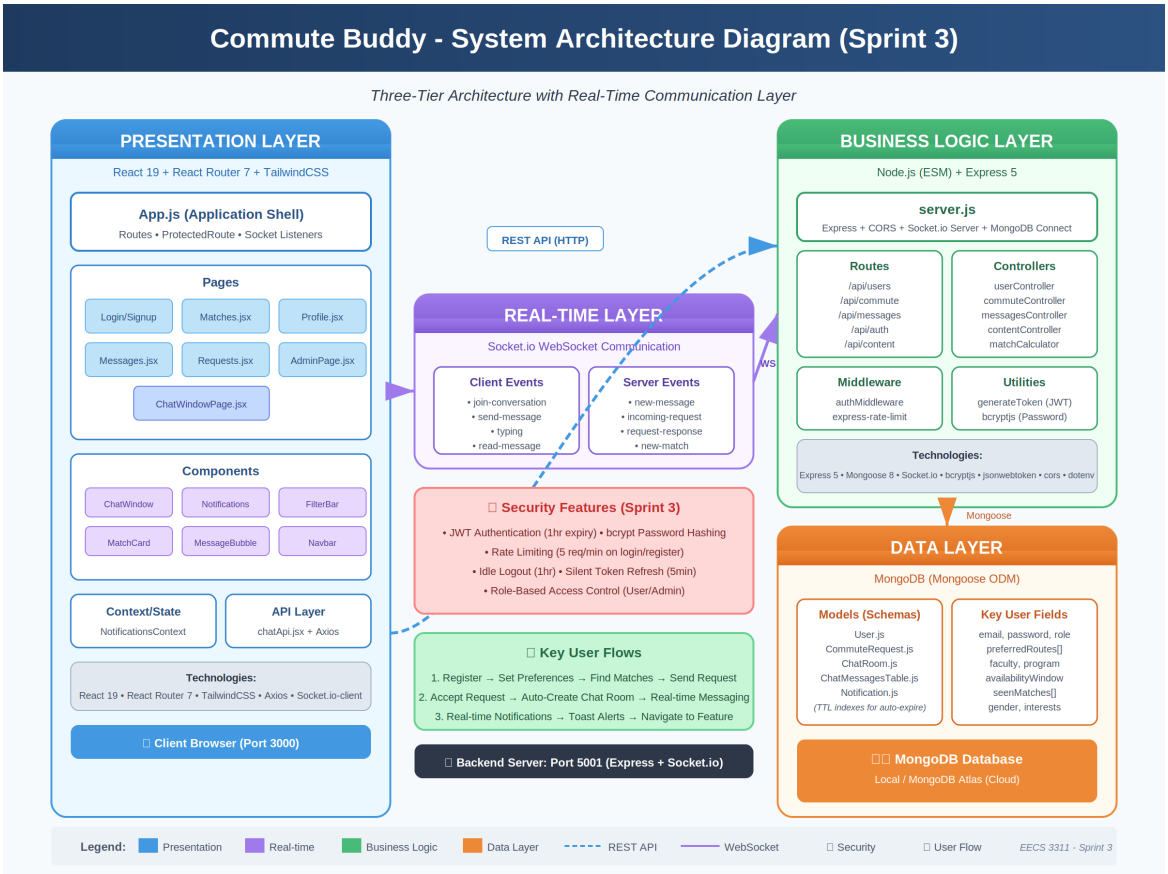

Figure 1: System architecture overview

The left side shows the React frontend running in the browser. It communicates with the backend through two channels: REST API calls (for things like fetching user profiles or submitting forms) and WebSocket connections (for real-time features). The backend sits in the middle, handling business logic and talking to MongoDB on the right.

2.2 Technology Stack

Here's what we used for each layer:

Layer	Technologies Used
Frontend	React 19, React Router 7, TailwindCSS, Axios, Socket.io-client
Backend	Node.js with ES modules, Express 5, Mongoose 8, Socket.io server, bcryptjs for password hashing, jsonwebtoken for auth, express-rate-limit
Database	MongoDB (we used a local instance for development, Atlas for production). TTL indexes handle automatic expiration of old requests.

3. CRC Cards

These cards describe the main classes in our system, what they're responsible for, and which other classes they work with. We have six primary classes that handle the core functionality.

User

User	
Responsibilities	**Collaborators**
Stores identity info (email, hashed password, role)	CommuteRequest
Keeps commute preferences (routes, times, start area)	ChatRoom
Tracks profile details like faculty and program	ChatMessage
Maintains list of already-seen matches	MatchCalculator

CommuteRequest

CommuteRequest	
Responsibilities	**Collaborators**
Links sender and receiver users	User (as sender)
Tracks status: pending, approved, or declined	User (as receiver)
Holds optional message from sender	ChatRoom (created on accept)
Auto-expires after 24 hours if not answered	Notification

ChatRoom

ChatRoom	
Responsibilities	**Collaborators**
Represents a conversation between two users	User (participant 1)
Stores both user IDs (sorted for uniqueness)	User (participant 2)
Maps to a Socket.io room for real-time delivery	ChatMessage

ChatMessage

ChatMessage	
Responsibilities	**Collaborators**
Stores the actual message text	ChatRoom
References who sent it	User (sender)
Belongs to a specific ChatRoom	Notification
Records timestamp for ordering	

Notification

Notification	
Responsibilities	**Collaborators**
Categorizes by type (match, request, message)	User (recipient)
Tracks which user should see it	CommuteRequest
Manages read/unread state	ChatMessage
Gets pushed via Socket.io instantly	

MatchCalculator

MatchCalculator (Utility)	
Responsibilities	**Collaborators**
Computes match percentage between users	User (current)
Weighs different factors (routes, times, area)	User (candidates)
Applies filters for faculty, gender, etc.	CommuteRequest
Excludes users with existing requests	

4. Data Models

Our MongoDB collections mirror the CRC classes above. Here are the key fields for each:

User Schema

email, password (hashed), name, role (user/admin), preferredRoutes[], startArea, commuteWindow, transportMode, faculty, program, gender, interests[], availabilityWindow, seenMatches[], profileImage

CommuteRequest Schema

sender (ref), receiver (ref), status (pending/approved/declined), message, expiresAt (TTL index for 24h auto-deletion), timestamps

ChatRoom Schema

user1Id (ref), user2Id (ref) – both sorted so the same pair always produces the same room ID

ChatMessage Schema

chatRoomId (ref, indexed), sender (ref), text, createdAt

5. API Endpoints

The backend exposes REST endpoints grouped by resource. All protected routes require a valid JWT in the Authorization header.

User Routes (/api/users)

Endpoint	Method	What it does
/register	POST	Creates account, returns JWT. Rate limited.
/login	POST	Authenticates, returns JWT with 1hr expiry.
/profile	GET/PUT	Read or update your profile.
/preferences	GET/PUT	Manage commute preferences.

Messaging Routes (/api/messages) – New in Sprint 3

Endpoint	Method	What it does
/open-or-create/:friendId	POST	Gets existing chat room or creates one.
/my-chats	GET	Lists all your chat rooms.
/:chatRoomId/messages	GET	Fetches message history for a room.
/:chatRoomId/send	POST	Sends a message, notifies recipient via socket.

6. Security

Security was a focus area this sprint, especially around session management:

- **Authentication** uses JWT tokens that expire after one hour. We store them in localStorage on the client side.
- **Rate limiting** is active on the login and register endpoints—5 requests per minute per IP. This helps prevent brute force attacks.
- **Idle logout** kicks in after an hour of inactivity. The frontend tracks mouse/keyboard activity and clears the session if the user goes idle.
- **Silent refresh** happens every 5 minutes for active users. This way, if you're actively using the app, you won't suddenly get logged out.
- **Password hashing** is done with bcrypt. We never store plain text passwords.
- **Email validation** only allows @yorku.ca and @my.yorku.ca addresses, enforced both client-side and server-side.

7. Setup Instructions

To run the application locally:

1. Clone the repository

2. Run **`npm install`** in both `setup/backend` and `setup/frontend`

3. Create a `.env` file in setup/backend with your `MONGO_URI` and `JWT_SECRET`

4. Start the backend: **`npm start`** from setup/backend (runs on port 5001)

5. Start the frontend: **`npm start`** from setup/frontend (runs on port 3000)

The frontend will automatically proxy API requests to the backend. Make sure MongoDB is running (locally or use Atlas connection string).

— End of Document —