Project Ideation: Student Commute Optimizer

1. Introduction

The "Student Commute Optimizer" is a full-stack web application designed to facilitate carpooling and route-sharing among students. The primary goal is to provide a platform where students can find others traveling along similar routes, connect with them, and coordinate shared rides. This not only helps in reducing travel costs but also contributes to a greener environment. The application will feature an interactive map, real-time chat, and a robust backend to handle user matching and communication.

2. My Thought Process (The Game Plan)

Alright, let's break this down. The project is a 'Student Commute Optimizer'. First thought: it's a carpooling app for students. The core idea is to connect people on similar routes. The biggest constraint is the 90-minute time limit. That means I have to be smart about what I build and what I leave out. Speed is everything.

So, what's the absolute minimum viable product (MVP)?

1. A way for a student to sign up. Anonymity is key, as the prompt says, so a unique username is a must.
2. A map where they can drop pins for their home and their college.
3. A way to see other students who are going roughly the same way.
4. A way to start a conversation with those potential matches.

That's it. Anything else is a 'nice-to-have' for later.

Now, how do I build this fast?

- **Architecture:** Forget microservices. That's overkill and a time sink. A simple, monolithic Node.js/Express backend with a React frontend is the way to go. One codebase, easy to manage, quick to get running.
- **The 'Matching' Logic:** My initial thought might be some complex algorithm that analyzes the entire route polyline for overlaps. But in 90 minutes? No chance. The trade-off is to simplify. What if I just check for proximity? If another student's start point is within, say, a 1-kilometer radius of my start point, AND their end point is within a 1km radius of my end point, they're a match. It's not perfect, but it's a fantastic starting point and way faster to code. I can use MongoDB's geospatial queries for this, which are super efficient.
- **Authentication:** Social logins like Google or Facebook are slick, but they require setting up API keys and callbacks. Too much hassle. A classic username/password system is

straightforward and gets the job done. I'll just need to remember to hash the passwords.
- **Tech Choices:** React with Vite for the frontend is a no-brainer; it's incredibly fast to spin up. On the backend, Node.js with Express is my go-to for building REST APIs quickly. For the database, MongoDB is perfect because its flexible schema won't slow me down if I need to make small changes. And for the real-time chat, Socket.IO is the industry standard and integrates beautifully with the Node/React stack.

So, the game plan is clear: build a simple monolith, use a proximity-based matching system, and stick to a familiar, fast tech stack. Focus on the core user journey: sign up, set route, see matches, and chat. Nail these, and I'll have a working prototype.

3. Technology Stack

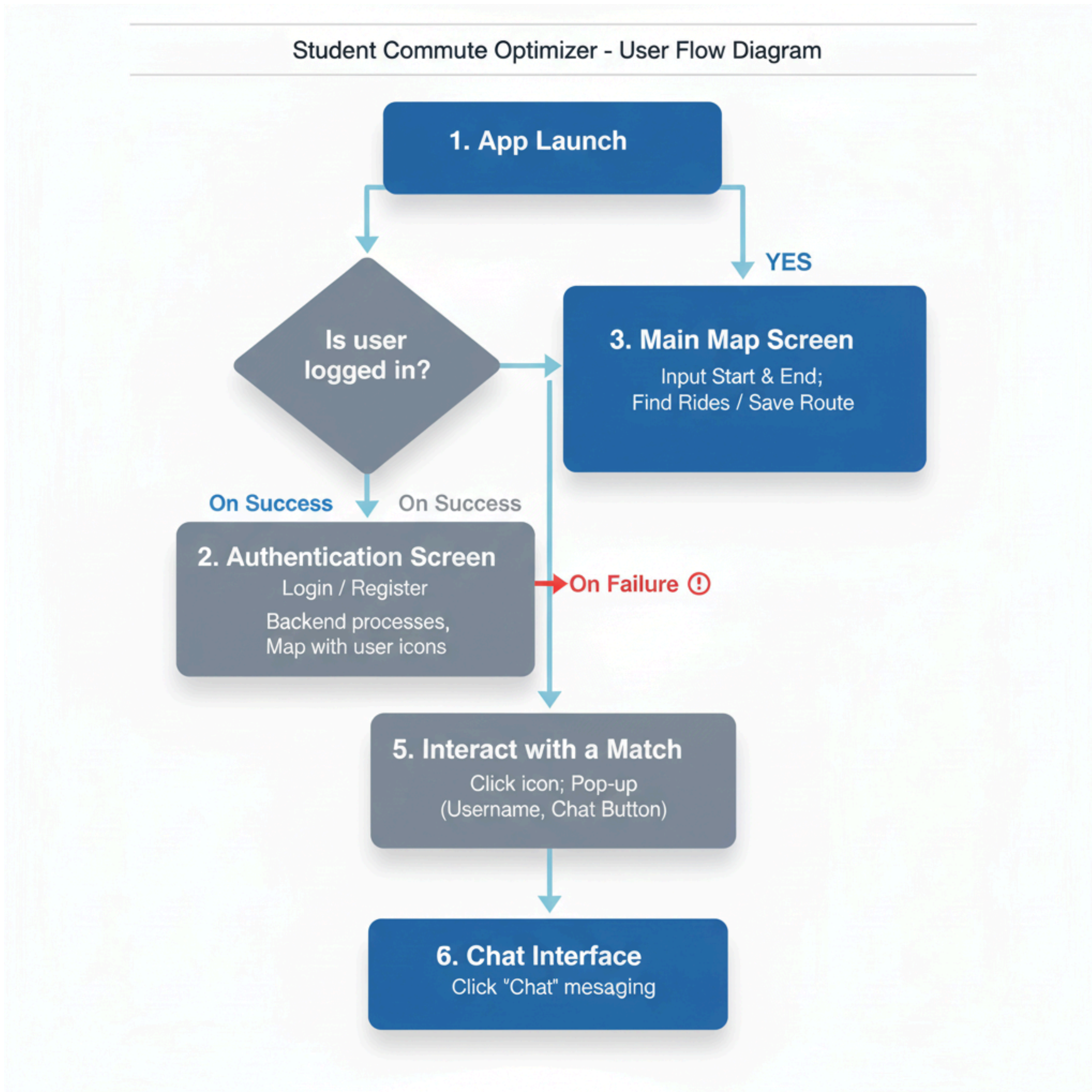
- **Frontend:** React with Vite for a fast development environment.
- **Mapping:** Leaflet.js with OpenStreetMap for a free and easy-to-use map solution.
- **Backend:** Node.js with Express.js for its speed and simplicity in creating REST APIs.
- **Database:** MongoDB, a NoSQL database that is flexible and works well with JavaScript-based stacks.
- **Real-time Communication:** Socket.IO for enabling real-time chat between users.

4. High-Level Diagrams

a. System Architecture

- **User's Browser (Client):** This is where the student interacts with our React application.
- **React SPA (Frontend):** Our single-page application that handles all the UI/UX. It talks to the backend via two channels:
  - **REST API (HTTP):** For actions like logging in, registering, and saving routes.
  - **WebSocket:** For real-time communication, specifically for the chat feature.
- **Node.js Server (Backend):** The brain of the operation. It runs on a server and contains:
  - **Express.js API:** Handles the HTTP requests from the frontend.
  - **Socket.IO:** Manages the real-time WebSocket connections for chat.
  - **Business Logic:** Contains the logic for matching users.
- **MongoDB (Database):** Our database where we store all persistent data like user accounts, their routes, and chat histories.

b. User Flow Diagram



This diagram illustrates the step-by-step journey a student takes when using the app, including key decisions and actions.

- **1. App Launch:** User opens the application.
    - **Decision:** Is the user already logged in?
        - **YES:** Proceed directly to the **Main Map Screen (Step 3)**.
        - **NO:** Proceed to the **Authentication Screen (Step 2)**.
- **2. Authentication Screen:** The user is prompted to log in or register.
    - **Action:** User enters credentials and submits the form.
    - **On Success:** Proceed to the **Main Map Screen (Step 3)**.
    - **On Failure:** Display an error message on the same screen.
- **3. Main Map Screen:** The user's primary interface.

- **Action:** User inputs their "Start" and "End" locations in the provided fields.
- **Action:** User clicks a "Find Rides" or "Save Route" button.

- **4. View Matches:** The system processes the route and displays results.
  - The app sends the user's route to the backend.
  - The backend finds matching users and returns them.
  - The map is populated with icons representing other students on similar routes.
- **5. Interact with a Match:** The user decides to connect with someone.
  - **Action:** User clicks on a student's icon on the map.
  - A pop-up appears showing the anonymous username and a "Chat" button.
- **6. Initiate Chat:** The user starts a conversation.
  - **Action:** User clicks the "Chat" button.
  - This action opens the **Chat Interface (Step 7)**.
- **7. Chat Interface:** The user communicates with the potential match.
  - A chat modal or new screen opens.
  - Users can send and receive messages in real-time to coordinate their commute.

5. Database Schema

We'll need three main collections:

users collection:

```
{
  "_id": "ObjectId",
  "username": "String (unique)",
  "password": "String (hashed)",
  "createdAt": "Timestamp"
}
```

routes collection:

```
{
  "_id": "ObjectId",
  "userId": "ObjectId (ref to User)",
  "startLocation": {
    "type": "Point",
    "coordinates": ["Longitude", "Latitude"]
  },
  "endLocation": {
    "type": "Point",
    "coordinates": ["Longitude", "Latitude"]
```

```
  },
  "createdAt": "Timestamp"
}
```

chats collection:

```
{
  "_id": "ObjectId",
  "participants": ["ObjectId (ref to User)", "ObjectId (ref to User)"],
  "messages": [
    {
      "senderId": "ObjectId (ref to User)",
      "message": "String",
      "timestamp": "Timestamp"
    }
  ]
}
```

6. API Design

Endpoint	Method	Description	Request Body	Response
/api/auth/register	POST	Register a new user	{ "username", "password" }	{ "token" }
/api/auth/login	POST	Login a user	{ "username", "password" }	{ "token" }
/api/routes	POST	Create a new route	{ "startLocation", "endLocation" }	{ "message": "Route saved" }
/api/routes/nearby	GET	Get nearby users	Query Params: lat, long	[{ "user", "route" }]

/api/chats	POST	Start a new chat	{ "recipientId" }	{ "chatId" }
/api/chats/:id	GET	Get chat history		[{ "messages" }]

7. Pseudo Code for Core Logic

Finding Nearby Users

This will be a backend function.

```
FUNCTION findNearbyUsers(currentUser, maxDistance):
  userRoute = GET userRoute from database WHERE userId = currentUser.id

  // Find users with start location near the current user's start location
  nearbyStartUsers = GET users from routes_collection WHERE startLocation is WITHIN
maxDistance of userRoute.startLocation

  // Find users with end location near the current user's end location
  nearbyEndUsers = GET users from routes_collection WHERE endLocation is WITHIN
maxDistance of userRoute.endLocation

  // Find the intersection of the two sets
  matchingUsers = INTERSECTION(nearbyStartUsers, nearbyEndUsers)

  RETURN matchingUsers
```

Real-time Chat with Socket.IO

Server-side:

```
// Set up Socket.IO server
io.on('connection', (socket) => {
  console.log('a user connected');

  // Join a room based on the chat ID
  socket.on('join_chat', (chatId) => {
    socket.join(chatId);
  });
```

```
  // Listen for new messages
  socket.on('send_message', (data) => {
    // Save message to database
    saveMessageToDb(data.chatId, data.message);

    // Broadcast the message to the other user in the room
    socket.to(data.chatId).emit('receive_message', data.message);
  });

  socket.on('disconnect', () => {
    console.log('user disconnected');
  });
});
```

Client-side (React):

```
// In your Chat component
useEffect(() => {
  // Connect to the Socket.IO server
  const socket = io("http://localhost:3000");

  // Join the chat room
  socket.emit('join_chat', chatId);

  // Listen for incoming messages
  socket.on('receive_message', (message) => {
    setMessages([...messages, message]);
  });

  // Clean up on component unmount
  return () => socket.disconnect();
}, [chatId]);

const sendMessage = () => {
  socket.emit('send_message', { chatId, message: newMessage });
  setMessages([...messages, newMessage]);
  setNewMessage("");
};
```