Project 1 - Collaborative AI-Powered Ideation & Project Management Platform

1.0 Project Overview

- **Problem Statement:** Develop an advanced platform for teams to collaboratively brainstorm ideas, plan projects, and manage tasks with real-time interaction and AI assistance for enhanced productivity.
- **Use Case:** Build a sophisticated tool that facilitates team collaboration, idea generation, and project execution, leveraging AI for intelligent suggestions and real-time updates for dynamic work environments.
- **Technology Stack:** MERN (MongoDB, Express.js, React.js, Node.js) with Socket.IO for real-time features and integration with third-party AI APIs.

2.0 Key Modules & Detailed Features

2.1 AI-Powered Idea Generation & Brainstorming Module

- **Description:** Users input prompts or keywords, and an integrated AI (e.g., OpenAI's GPT, Google Gemini API) generates creative ideas or breaks down complex problems. These ideas can then be saved, categorized, and refined collaboratively.
- **Implementation Notes:**
  - **Backend:** Node.js API handles prompt submission to the AI service and stores AI responses in MongoDB.

Conceptual Snippet (Backend - Express Route):

JavaScript

```
// routes/aiRoutes.js

router.post('/generate-ideas', authMiddleware, async (req, res) => {

  const { prompt, projectId } = req.body;

  try {

    // Placeholder for actual AI API call (e.g., OpenAI, Google Gemini)

    const aiResult = await aiService.generateText(prompt);

    const newIdea = new Idea({

      text: aiResult.generated_text,

      projectId: projectId,

      createdBy: req.user.id,

      // ... other fields
```

```
        });
        await newIdea.save();
        res.status(201).json(newIdea);
    } catch (error) {
        console.error("AI Idea Generation Error:", error);
        res.status(500).json({ message: "Failed to generate ideas." });
    }
});
```

Frontend: React components provide input forms, display AI results, and manage UI for saving/categorizing ideas.

Conceptual Snippet (Frontend - React Component):

JavaScript

```
// components/IdeaGenerator.js
import React, { useState } from 'react';
import axios from 'axios';

function IdeaGenerator({ projectId }) {
    const [prompt, setPrompt] = useState('');
    const [ideas, setIdeas] = useState([]);

    const handleSubmit = async () => {
        try {
            const response = await axios.post('/api/ai/generate-ideas', { prompt, projectId });
            setIdeas([...ideas, response.data]); // Add new idea to list
        } catch (error) {
            console.error("Error fetching ideas:", error);
        }
    };

    return (
        <div>
          <textarea value={prompt} onChange={(e) => setPrompt(e.target.value)} />
          <button onClick={handleSubmit}>Generate Ideas</button>
          <div>
            {ideas.map(idea => (
              <div key={idea._id}>{idea.text}</div> // Display AI generated ideas
            ))}
          </div>
        </div>
    );
}
```

2.2 Real-time Collaborative Whiteboard/Mind Map (Socket.IO)

- **Description:** A shared canvas where multiple users can simultaneously draw, add sticky notes, create shapes, and build mind maps. Changes are instantly visible to all participants in the same session.
- **Implementation Notes:**
    - **Backend:** Socket.IO manages real-time bidirectional communication, handles rooms, and broadcasts drawing/state updates.

Conceptual Snippet (Backend - Socket.IO Integration):

JavaScript

```
// server.js (excerpt)
const io = require('socket.io')(server); // 'server' is your http/https server

io.on('connection', (socket) => {
    console.log('A user connected:', socket.id);

    socket.on('joinRoom', (roomId, callback) => {
        socket.join(roomId);
        console.log(`${socket.id} joined room ${roomId}`);
        // Optionally, send initial whiteboard state from DB
        callback({ status: 'joined', roomId: roomId });
    });

    socket.on('drawing', (data) => {
        // data might contain drawing commands, coordinates, color etc.
        socket.to(data.roomId).emit('drawing', data); // Broadcast to others in the same room
    });

    socket.on('addShape', (data) => {
        socket.to(data.roomId).emit('addShape', data);
    });

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

Frontend: React component utilizing a canvas library (e.g., Konva.js, Fabric.js) or custom canvas implementation. Emits and listens for Socket.IO events for drawing data.

Conceptual Snippet (Frontend - React with Socket.IO):

JavaScript

```
// components/Whiteboard.js
import React, { useEffect, useRef } from 'react';
import io from 'socket.io-client';

const socket = io('http://localhost:5000'); // Connect to your backend Socket.IO

function Whiteboard({ roomId }) {
    const canvasRef = useRef(null);

    useEffect(() => {
        socket.emit('joinRoom', roomId);
        socket.on('drawing', (data) => {
          // Logic to draw on canvas based on received data
          console.log('Received drawing data:', data);
        });

        socket.on('addShape', (data) => {
          // Logic to add shape to canvas
          console.log('Received shape data:', data);
        });

        return () => {
          socket.off('drawing');
          socket.off('addShape');
          // socket.emit('leaveRoom', roomId); // Optional: if implementing leave room
        };
      }, [roomId]);

      const handleMouseDown = (e) => {
        // Logic to start drawing, emit 'drawing' event to server
        socket.emit('drawing', { roomId, /* drawing data */ });
      };

      return <canvas ref={canvasRef} onMouseDown={handleMouseDown} /* ... other events */ />;
    }
```

2.3 Dynamic Kanban-style Project Boards with Task Management

- **Description:** Boards with customizable columns (e.g., To Do, In Progress, Done) where tasks can be created, assigned, prioritized, and moved via drag-and-drop. Each task can have details, deadlines, and assigned users.
- **Implementation Notes:**

- **Backend:** RESTful APIs manage Task CRUD operations, associating tasks with projects/users.
- **Frontend:** React components use a drag-and-drop library (e.g., react-beautiful-dnd, react-dnd) with state management (Redux/Context API) for task data.

Conceptual Snippet (Frontend - Drag and Drop):

JavaScript

```
// components/KanbanBoard.js
import React, { useState, useEffect } from 'react';
import { DragDropContext, Droppable, Draggable } from 'react-beautiful-dnd';
import axios from 'axios';

function KanbanBoard({ projectId }) {
  const [columns, setColumns] = useState({}); // State for Kanban columns and tasks

  useEffect(() => {
    const fetchTasks = async () => {
      const response = await axios.get(`/api/projects/${projectId}/tasks`);
      // Process tasks into column structure
      setColumns(processTasksIntoColumns(response.data));
    };
    fetchTasks();
  }, [projectId]);

  const onDragEnd = async (result) => {
    const { source, destination, draggableId } = result;
    if (!destination) return;
    if (source.droppableId === destination.droppableId && source.index === destination.index) return;

    // Logic to update column state locally
    const newColumns = updateColumnsOnDragEnd(columns, source, destination, draggableId);
    setColumns(newColumns);
    // Update backend
    await axios.put(`/api/tasks/${draggableId}`, {
      status: destination.droppableId, // Update task status
      position: destination.index // Update task position
    });
  };

  return (
    <DragDropContext onDragEnd={onDragEnd}>
      {Object.entries(columns).map(([columnId, column]) => (
        <Droppable key={columnId} droppableId={columnId}>
          {(provided) => (
            <div ref={provided.innerRef} {...provided.droppableProps}>
              <h3>{column.title}</h3>
              {column.tasks.map((task, index) => (
                <Draggable key={task._id} draggableId={task._id} index={index}>
                  {(provided) => (
                    <div
                      ref={provided.innerRef}
                      {...provided.draggableProps}
                      {...provided.dragHandleProps}
                    >
                      {task.title}
                    </div>
                  )}
                </Draggable>
              ))}
              {provided.placeholder}
            </div>
```

```
                )}
              </Droppable>
            ))}
          </DragDropContext>
  );
}
```

2.4 Advanced User & Role-Based Access Control (RBAC)

- **Description:** Defines different roles (Admin, Project Manager, Team Member, Guest) with varying permissions for accessing, creating, editing, and deleting content across projects and features.
- **Implementation Notes:**

Backend: JWT for authentication. Middleware functions check user roles and permissions before allowing API access.

Conceptual Snippet (Backend - RBAC Middleware):

JavaScript

```
// middleware/authMiddleware.js
const jwt = require('jsonwebtoken');
const User = require('../models/User'); // Assuming User model has 'role' field

const protect = async (req, res, next) => {
  let token;
  if (req.headers.authorization && req.headers.authorization.startsWith('Bearer')) {
    try {
      token = req.headers.authorization.split(' ')[1];
      const decoded = jwt.verify(token, process.env.JWT_SECRET);
      req.user = await User.findById(decoded.id).select('-password');
      next();
    } catch (error) {
      res.status(401).json({ message: 'Not authorized, token failed' });
    }
  }
```

```
    if (!token) {
        res.status(401).json({ message: 'Not authorized, no token' });
    }
};


const authorizeRoles = (...roles) => {
    return (req, res, next) => {
        if (!roles.includes(req.user.role)) {
            return res.status(403).json({ message: `User role ${req.user.role} is not authorized to access this route` });
        }
        next();
    };
};
module.exports = { protect, authorizeRoles };


// Example usage in a route:
// router.post('/admin/users', protect, authorizeRoles('admin'), createUser);
```

Frontend: Conditional rendering of UI elements based on the logged-in user's role and permissions.

2.5 Integrated Real-time Chat & Commenting

- **Description:** Real-time chat within project rooms and threaded comments on individual tasks or idea cards.
- **Implementation Notes:**
    - **Backend:** Socket.IO for real-time chat messages. RESTful APIs store threaded comments in MongoDB.
    - **Frontend:** React components for chat interfaces and comment sections, utilizing Socket.IO for instant updates.

2.6 Version Control for Collaborative Documents/Notes

- **Description:** Allows users to create and edit shared text documents or notes within a project. The system tracks changes, enabling users to view previous versions and potentially revert.
- **Implementation Notes:**

Backend: Stores document content and tracks revisions in MongoDB (e.g., an array of historical versions). RESTful APIs manage document CRUD and version retrieval.

Conceptual Snippet (Backend - Document Model):

JavaScript

```
// models/Document.js

const mongoose = require('mongoose');


const DocumentSchema = new mongoose.Schema({
  projectId: { type: mongoose.Schema.Types.ObjectId, ref: 'Project', required: true },
  title: { type: String, required: true },
  currentContent: { type: String, default: '' },
  versions: [{
    content: { type: String, required: true },
    timestamp: { type: Date, default: Date.now },
    editedBy: { type: mongoose.Schema.Types.ObjectId, ref: 'User' }
  }],
  createdBy: { type: mongoose.Schema.Types.ObjectId, ref: 'User', required: true },
  createdAt: { type: Date, default: Date.now },
  updatedAt: { type: Date, default: Date.now }
});


// Pre-save hook to add new version on update
DocumentSchema.pre('save', function(next) {
  if (this.isModified('currentContent') && this.currentContent !== undefined) {
    this.versions.push({
      content: this.currentContent,
      editedBy: this.updatedBy || this.createdBy // 'updatedBy' would be set on put request
    });
    this.updatedAt = Date.now();
  }
```

```
    next();
});


module.exports = mongoose.model('Document', DocumentSchema);
```

Frontend: A rich text editor (e.g., Quill, Draft.js) for editing. UI displays version history and offers restoration of previous versions.

2.7 Analytics Dashboard with AI-driven Insights

- **Description:** Visualizes team progress, task completion rates, and identifies bottlenecks. AI can analyze collaboration patterns and suggest improvements.
- **Implementation Notes:**
    - **Backend:** Aggregates data from MongoDB (tasks, user activity). AI potentially analyzes patterns to generate insights.
    - **Frontend:** Charting libraries (e.g., Chart.js, Recharts) display data visually. React components design the dashboard layout.

3.0 API Endpoints (Conceptual)

(Note: BASE_URL refers to your deployed backend API URL, e.g., https://api.yourproject.com/v1)

3.1 Authentication & User Management

- POST /auth/register - User/Team Registration
- POST /auth/login - User/Team Login
- GET /auth/me - Get Current User Profile
- GET /users/:id - Get User Details (for profiles)
- PUT /users/:id - Update User Profile
- PUT /users/:id/role - Update User Role (Admin only)
- GET /admin/users - Get all users (Admin only)

3.2 Projects

- POST /projects - Create New Project
- GET /projects - Get All Projects (accessible to user)
- GET /projects/:id - Get Project Details
- PUT /projects/:id - Update Project Details
- DELETE /projects/:id - Delete Project
- POST /projects/:id/members - Add Member to Project

- DELETE /projects/:id/members/:memberId - Remove Member from Project

3.3 Tasks

- POST /projects/:projectId/tasks - Create New Task
- GET /projects/:projectId/tasks - Get All Tasks for a Project
- GET /tasks/:id - Get Task Details
- PUT /tasks/:id - Update Task (e.g., status, assignment, details)
- DELETE /tasks/:id - Delete Task

3.4 AI Ideation

- POST /ai/generate-ideas - Generate ideas from AI
- POST /ideas - Save a generated idea
- GET /projects/:projectId/ideas - Get ideas for a specific project
- PUT /ideas/:id - Update an idea (e.g., refine, categorize)

3.5 Whiteboard & Real-time (Socket.IO Events)

- connection, disconnect
- joinRoom: Client joins a specific whiteboard room
- drawing: Client sends drawing data (broadcast to room)
- addShape, addText, addStickyNote: Client sends data for adding objects
- updateObject, deleteObject: Client sends data for modifying/deleting objects
- canvasState: Initial canvas state transfer on room join

3.6 Chat & Comments

- GET /projects/:projectId/chat/messages - Get chat history for a project
- POST /tasks/:taskId/comments - Add a comment to a task
- GET /tasks/:taskId/comments - Get comments for a task
- *(Socket.IO Events for Real-time Chat)*: sendMessage, receiveMessage, typing

3.7 Documents & Version Control

- POST /projects/:projectId/documents - Create New Document
- GET /projects/:projectId/documents - Get All Documents for a project
- GET /documents/:id - Get Document Content
- PUT /documents/:id - Update Document Content (creates new version)
- GET /documents/:id/versions - Get Document Version History

- POST /documents/:id/versions/:versionId/restore - Restore a previous version

3.8 Analytics (Admin/PM specific)

- GET /analytics/projects/:projectId/progress - Project Progress Metrics
- GET /analytics/team/:teamId/activity - Team Activity Metrics
- GET /analytics/ai-insights/:projectId - AI-generated insights for project

4.0 Week-wise Development Plan

Week	Backend (Node.js + Express)	Frontend (React.js)
Week 1	JWT Auth (Team Roles), User & Project APIs, AI Integration Setup	React Setup, Auth UI, Collaborative Workspace UI
Week 2	Real-time APIs (Socket.IO for Whiteboard), Idea Generation API, Task CRUD APIs	Collaborative Whiteboard, AI Idea Integration, Basic Task UI
Mid Project Review	**Real-time Collaboration & AI Integration initiated**	**Core Collaborative UI functional**
Week 3	Task Status APIs, Commenting APIs, Version Control APIs	Task Management UI, Chat UI, Document Version History
Week 4	Analytics & Insights APIs, Admin APIs, API Cleanup & Testing	Analytics Dashboard, Admin Panel, UI Polish & Responsiveness
Final Project Review	**All modules functional: AI Integration, Real-time Collaboration, Project Management, Admin Panel**	**Fully responsive frontend: Collaborative Workspace, Dashboards**

5.0 Development Steps

1. **Project Setup & Base MERN:** Initialize MERN stack, configure MongoDB connection, set up basic folder structure for client and server.
2. **Authentication & Authorization:** Implement JWT token generation/verification. Create user, admin, project manager, and team member roles. Develop Express middleware for protected routes and role-based access control.
3. **Core Project & Task Management:** Build MongoDB schemas and Express APIs for creating, reading, updating, and deleting projects and tasks. Implement associated React components for displaying and managing these entities.
4. **Real-time Whiteboard:** Integrate Socket.IO on the backend to handle real-time drawing data. On the frontend, integrate a canvas library (e.g., Konva.js) and emit/listen for Socket.IO events for collaborative drawing. Implement saving and loading whiteboard states.
5. **AI Integration for Ideation:** Set up a secure connection from the Node.js backend to a chosen AI API (e.g., OpenAI, Google Gemini). Create Express routes to send prompts and receive AI-

generated ideas. Develop React components to display AI suggestions and allow users to save/categorize them.

6. **Kanban Board with Drag-and-Drop:** Utilize a React drag-and-drop library (e.g., react-beautiful-dnd) to build the Kanban board UI. Ensure seamless communication with the backend APIs for updating task statuses and positions.
7. **Real-time Chat & Comments:** Implement Socket.IO for instant messaging within project rooms. For threaded comments on tasks, create RESTful APIs and corresponding React components.
8. **Document Version Control:** Design MongoDB schemas to store multiple versions of documents. Implement backend logic to create new versions on each significant save. Develop frontend UI to view historical versions and a restore functionality.
9. **Admin Panel & Analytics:** Create separate backend routes and frontend components for the admin dashboard. Develop APIs to fetch aggregated project data and user activity. Integrate charting libraries (e.g., Chart.js) to visualize data.
10. **Refinement, Testing & Security:** Implement thorough validation on all API inputs. Conduct unit, integration, and end-to-end testing. Address potential security vulnerabilities (e.g., XSS, CSRF). Optimize database queries and frontend rendering for performance.
11. **Deployment:** Prepare the application for deployment. Deploy the React frontend to a static hosting service (e.g., Vercel, Netlify) and the Node.js backend/MongoDB to a cloud provider (e.g., Render, AWS EC2, DigitalOcean). Configure environment variables securely.

Project 2 - Personalized Health & Wellness Companion with Biometric Integration

1.0 Project Overview

- **Problem Statement:** Create a personalized health and wellness platform that integrates biometric data to provide AI-driven, customized recommendations for fitness, nutrition, and mental well-being.
- **Use Case:** Develop a comprehensive digital health companion that intelligently adapts to user's health data, offering tailored plans and insights to achieve personal wellness goals, akin to a smart personal trainer and nutritionist.
- **Technology Stack:** MERN (MongoDB, Express.js, React.js, Node.js) with potential integration with third-party fitness APIs and AI recommendation engines.

2.0 Key Modules & Detailed Features

2.1 User Registration & Health Profile Management

- **Description:** Secure user authentication with comprehensive profile creation including demographics (age, gender), physical attributes (height, weight, activity level), health goals, dietary preferences, and allergies.

- **Implementation Notes:**
    - **Backend:** JWT for authentication, extensive MongoDB schema for user profiles.
    - **Frontend:** Multi-step registration forms for detailed profile input.

2.2 Biometric Data Input & Integration (manual/API-based)

- **Description:** Allows users to manually log various biometric data points (e.g., heart rate, sleep duration, blood pressure, weight, steps). *Advanced*: Integration with third-party fitness tracker APIs (e.g., Fitbit, Google Fit, Apple HealthKit) for automatic data synchronization.
- **Implementation Notes:**

Backend: APIs to receive and store diverse biometric data over time. For API integration, manage OAuth flows and parse data from external services.

Conceptual Snippet (Backend - Biometric Data Model):

JavaScript

```
// models/BiometricData.js

const mongoose = require('mongoose');


const BiometricDataSchema = new mongoose.Schema({

  userId: { type: mongoose.Schema.Types.ObjectId, ref: 'User', required: true },

  type: { type: String, enum: ['weight', 'heart_rate', 'sleep_duration', 'blood_pressure', 'steps'], required:
true },

  value: { type: Number, required: true },

  unit: { type: String }, // e.g., 'kg', 'bpm', 'hours'

  date: { type: Date, default: Date.now }

});


module.exports = mongoose.model('BiometricData', BiometricDataSchema);


// routes/biometricRoutes.js

router.post('/', protect, async (req, res) => {

  const { type, value, unit } = req.body;

  try {

    const newData = new BiometricData({
            userId: req.user.id,
            type, value, unit
        });
        await newData.save();
        res.status(201).json(newData);
    } catch (error) {
        res.status(500).json({ message: "Failed to log biometric data." });
    }
});
```

Frontend: User-friendly forms for manual data input. UI for connecting to and authorizing external fitness trackers.

2.3 AI-Powered Recommendation Engine (Workouts, Meal Plans, Mindfulness)

- **Description:** Based on user profile, goals, and biometric data, the AI generates personalized workout routines (with exercises, sets, reps, video links), customized meal plans (recipes, calorie counts), and tailored mindfulness exercises. This system should continuously learn and adapt recommendations over time.
- **Implementation Notes:**

Backend: Integrates with an AI API (specialized health AI or fine-tuned general AI). Contains logic to process user data and query AI for recommendations. Stores generated plans.

Conceptual Snippet (Backend - Recommendation Service):

JavaScript

```
// services/recommendationService.js
const axios = require('axios'); // For calling AI API

async function getPersonalizedPlan(userId) {
    // Fetch user profile and recent biometric data
    const userProfile = await User.findById(userId);
    const recentBiometrics = await BiometricData.find({ userId }).sort({ date: -1 }).limit(7);

    // Construct a detailed prompt for the AI
    const prompt = `Generate a personalized 7-day workout plan for a ${userProfile.age}-year-old
${userProfile.gender} weighing ${userProfile.weight}kg, aiming for ${userProfile.goals.join(', ')}.
Recent heart rate: ${recentBiometrics[0]?.value}bpm. Include diverse exercises, sets, reps, and consider
${userProfile.activityLevel} activity.`;


  try {
     const aiResponse = await axios.post('https://api.ai-provider.com/generate', { prompt });
     // Parse AI response into structured plan
     const plan = parseAiPlan(aiResponse.data.text);
     return plan;
  } catch (error) {
     console.error("AI Recommendation Error:", error);
     throw new Error("Could not generate personalized plan.");
  }
}
module.exports = { getPersonalizedPlan };
```

Frontend: Clearly displays generated plans with interactive elements (e.g., mark workout complete, view recipe details).

2.4 Interactive Health Dashboards & Progress Tracking

- **Description:** Visual dashboards presenting trends in biometric data over time (e.g., charts for weight, heart rate, sleep patterns), displaying progress towards defined goals, and summarizing completed activities.
- **Implementation Notes:**
  - **Backend:** APIs to query and aggregate historical biometric data efficiently.
  - **Frontend:** Utilizes charting libraries (e.g., Chart.js, Recharts) to render dynamic graphs. React components are designed for progress indicators and activity summaries.

Conceptual Snippet (Frontend - Charting):

JavaScript

```
// components/WeightChart.js
import React, { useEffect, useState } from 'react';
import { Line } from 'react-chartjs-2';
import axios from 'axios';
import { Chart as ChartJS, CategoryScale, LinearScale, PointElement, LineElement, Title, Tooltip,
Legend } from 'chart.js';
```

```
ChartJS.register(CategoryScale, LinearScale, PointElement, LineElement, Title, Tooltip, Legend);

function WeightChart() {
  const [chartData, setChartData] = useState({ labels: [], datasets: [] });

  useEffect(() => {
    const fetchWeightData = async () => {
      const response = await axios.get('/api/biometrics/weight');
      const dates = response.data.map(d => new Date(d.date).toLocaleDateString());
      const weights = response.data.map(d => d.value);

      setChartData({
        labels: dates,
        datasets: [{
          label: 'Weight (kg)',
          data: weights,
          borderColor: 'rgb(75, 192, 192)',
          tension: 0.1
        }]
      });
    };
    fetchWeightData();
  }, []);

  return <Line data={chartData} options={{ responsive: true, plugins: { legend: { position: 'top' }, title: { display: true, text: 'Weight Progress' } } }} />;
}
```

2.5 Community & Expert Network (Optional: Booking/Video Conferencing)

- **Description:** Users can connect with friends, share progress optionally, and join interest-based groups. Additionally, a marketplace or directory allows users to find and book sessions with certified health experts (trainers, nutritionists, therapists).
- **Implementation Notes:**
    - **Backend:** APIs for friend connections, group management, expert profiles, and a robust booking system.
    - **Frontend:** Features include a social feed, group chat functionalities, and an expert booking interface with potential integration for calendar/scheduling APIs (e.g., Calendly API) and video conferencing APIs (e.g., Daily.co, Zoom SDK).

2.6 Gamification & Streak Tracking System

- **Description:** Awards badges, points, or virtual rewards for achieving health milestones (e.g., 30 days of consistent workouts, reaching a specific weight goal). Tracks streaks for consistent healthy habits to encourage sustained engagement.
- **Implementation Notes:**

Backend: Logic to track user activity, calculate points/badges, and manage consecutive streaks.

Conceptual Snippet (Backend - Streak Logic):

JavaScript

```
// services/gamificationService.js

async function updateWorkoutStreak(userId) {
  const today = new Date();
  today.setHours(0, 0, 0, 0); // Normalize date to start of day

  const yesterday = new Date(today);
  yesterday.setDate(yesterday.getDate() - 1);

  const user = await User.findById(userId);
  if (!user) return;

  const lastWorkout = await WorkoutLog.findOne({ userId }).sort({ date: -1 });

  if (lastWorkout && new Date(lastWorkout.date).toDateString() === yesterday.toDateString()) {
    user.currentWorkoutStreak += 1;
  } else if (!lastWorkout || new Date(lastWorkout.date).toDateString() !== today.toDateString()) {
    // Start new streak if no workout today or last one was not yesterday
    user.currentWorkoutStreak = 1;
  }
  // If workout was already logged today, streak remains same or is updated by other means
  await user.save();
}
```

  - **Frontend:** UI prominently displays earned badges, accumulated points, and current streaks.

2.7 Smart Notifications & Reminders

- **Description:** AI-driven reminders for workouts, meal times, water intake, or mindfulness breaks, intelligently adjusting based on the user's daily schedule and current activity levels.
- **Implementation Notes:**

Backend: Node.js cron jobs or scheduled tasks send push notifications or emails. Logic to personalize notification timing.

Frontend: UI for users to customize notification preferences.

3.0 API Endpoints (Conceptual)

(Note: BASE_URL refers to your deployed backend API URL, e.g., https://api.yourproject.com/v1)

3.1 Authentication & User Profiles

- POST /auth/register - User Registration
- POST /auth/login - User Login
- GET /auth/me - Get Current User Profile
- GET /users/:id - Get Public User Profile (e.g., for community)
- PUT /users/profile - Update Current User's Health Profile
- GET /admin/users - Get all users (Admin only)

3.2 Biometric Data

- POST /biometrics - Log New Biometric Data (e.g., POST /biometrics/weight, POST /biometrics/sleep)
- GET /biometrics/:type - Get Historical Biometric Data by Type (e.g., /biometrics/weight)
- GET /biometrics/summary - Get Summary of Recent Biometric Data
- POST /integrations/fitbit/callback - OAuth Callback for Fitbit (example)

3.3 Recommendations

- GET /recommendations/workouts - Get Personalized Workout Plan
- GET /recommendations/meals - Get Personalized Meal Plan
- GET /recommendations/mindfulness - Get Personalized Mindfulness Exercises
- POST /recommendations/feedback - Provide Feedback on Recommendations

3.4 Goals & Progress

- POST /goals - Set a New Health Goal
- GET /goals - Get User's Active Goals
- PUT /goals/:id - Update Goal Progress/Status
- GET /progress/dashboard - Get Data for Health Dashboard Charts

3.5 Community & Experts

- POST /community/posts - Create Community Post
- GET /community/feed - Get Community Feed
- POST /experts/search - Search for Experts (trainers, nutritionists)
- GET /experts/:id - Get Expert Profile
- POST /experts/:id/book - Book a Session with an Expert
- GET /groups - Get available groups
- POST /groups/:id/join - Join a group

3.6 Gamification

- GET /gamification/badges - Get User's Earned Badges
- GET /gamification/streaks - Get User's Current Streaks
- GET /gamification/leaderboard - Get Leaderboard (optional)

3.7 Notifications

- POST /notifications/settings - Update Notification Preferences
- GET /notifications/scheduled - Get Scheduled Reminders

4.0 Week-wise Development Plan

Week	Backend (Node.js + Express)	Frontend (React.js)
Week 1	Auth APIs, User Profile & Goal APIs, Biometric Data APIs	React Setup, Auth UI, Profile Setup, Data Input UI

Week	Backend (Node.js + Express)	Frontend (React.js)
Week 2	AI Recommendation Engine Integration, Workout/Meal Plan APIs	Dashboard UI (Charts), Recommendation Display, Progress Tracking
Mid Project Review	**Auth, Profile, Biometric Data & Basic Recommendations integrated**	**Core Dashboard & Data Input functional**
Week 3	Community/Expert APIs, Gamification APIs, Notification APIs	Community/Expert UI, Gamification UI, Notification Settings
Week 4	API Cleanup & Testing, Deployment Readiness	Responsive UI Fixes, UI Polish, Testing
Final Project Review	**All modules functional: Personalized Recommendations, Data Tracking, Community, Gamification**	**Complete responsive frontend: Dashboards, Profile, Community**

5.0 Development Steps

1. **Core User & Profile Setup:** Initialize MERN stack, database, and basic folder structure. Implement JWT authentication and secure user registration. Create a comprehensive MongoDB schema for user profiles, including health goals and preferences.
2. **Biometric Data Logging:** Develop Express APIs to handle the submission and storage of various biometric data points (weight, sleep, heart rate, etc.) to MongoDB. Build intuitive React forms and input fields for users to manually log this data.
3. **Basic Recommendation Engine (Rule-based):** Start with a foundational recommendation system on the backend, using rule-based logic to provide initial fitness or nutrition suggestions based on the user's static profile data (e.g., activity level, goals).
4. **Interactive Dashboards:** Develop backend APIs to query and aggregate historical biometric and activity data. On the frontend, integrate charting libraries (e.g., Chart.js, Recharts) to visualize this data in interactive graphs and progress charts on a user dashboard.
5. **AI Integration for Recommendations:** Integrate a third-party AI API (e.g., specialized health AI, or a general AI fine-tuned with health data) into your Node.js backend. Refine the backend logic to process dynamic user biometric data and send intelligent prompts to the AI for highly personalized workout plans, meal plans, and mindfulness exercises. Implement UI to display and interact with these AI-generated plans.
6. **Gamification & Notifications:** Develop backend logic to track user achievements, award points/badges for milestones, and manage activity streaks. Implement a notification system (e.g., Node.js cron jobs, web push API) to send personalized reminders for workouts, meals, or mindfulness sessions based on user preferences.
7. **Community & Expert Network (Optional):** Build social features allowing users to connect with friends and join health-related groups. For the expert network, develop APIs for expert

profiles, a search mechanism, and potentially integrate a booking system (e.g., Calendly API) and a video conferencing SDK (e.g., Daily.co) for consultations.

8. **Refinement, Testing & Security:** Conduct thorough unit, integration, and end-to-end testing across the entire application. Optimize backend API performance and database queries. Implement robust security measures, including data privacy and secure handling of sensitive health information. Ensure the application is fully responsive across devices.

9. **Deployment:** Prepare the MERN application for production deployment. Deploy the React frontend to a static hosting service (e.g., Vercel, Netlify) and the Node.js backend along with the MongoDB database to a scalable cloud provider (e.g., Render, AWS, DigitalOcean), ensuring all environment variables are securely configured.

Project 3 - Decentralized & Community-Driven News Aggregator/Fact-Checker Platform

1.0 Project Overview

- **Problem Statement:** Build a community-driven news aggregation platform focused on combating misinformation through collaborative fact-checking, source reliability scoring, and transparent content moderation.
- **Use Case:** Create a unique news consumption experience where users actively participate in verifying information, promoting credible sources, and flagging misinformation, fostering a more trustworthy news environment.
- **Technology Stack:** MERN (MongoDB, Express.js, React.js, Node.js) with potential for real-time updates via WebSockets and advanced text processing.

2.0 Key Modules & Detailed Features

2.1 User & Content Submission (Articles, Links)

- **Description:** Registered users can submit news articles by providing URLs, headlines, and summaries. Articles are then categorized (e.g., politics, technology, health) and tagged for discoverability.
- **Implementation Notes:**

Backend: APIs handle article submission, including logic for URL parsing (e.g., using node-readability or similar to extract core content). MongoDB schema stores article data.

Conceptual Snippet (Backend - Article Submission):

JavaScript

// models/Article.js

const mongoose = require('mongoose');

```
const ArticleSchema = new mongoose.Schema({
    title: { type: String, required: true },
    url: { type: String, required: true, unique: true },
    summary: { type: String },
    fullContent: { type: String }, // Extracted content
    category: { type: String, required: true },
    tags: [String],
    submittedBy: { type: mongoose.Schema.Types.ObjectId, ref: 'User', required: true },
    createdAt: { type: Date, default: Date.now },
    upvotes: { type: Number, default: 0 },
    downvotes: { type: Number, default: 0 },
    // ... other fields like 'status' (pending, approved, debunked)
});
module.exports = mongoose.model('Article', ArticleSchema);

// routes/articleRoutes.js
router.post('/', protect, async (req, res) => {
    const { url, title, summary, category, tags } = req.body;
    try {
        // Logic to fetch and parse full content from URL (e.g., using a library)
        const fullContent = await parseUrlContent(url); // Placeholder
        const newArticle = new Article({
            url, title, summary, fullContent, category, tags,
            submittedBy: req.user.id
        });
        await newArticle.save();
        res.status(201).json(newArticle);
    } catch (error) {
        res.status(500).json({ message: "Failed to submit article." });
    }
});
```

Frontend: Provides a form for submitting articles and displays parsed previews for user verification.

2.2 Community Upvoting/Downvoting & Categorization

- **Description:** Users can upvote or downvote submitted articles and individual fact-checks/annotations. Articles are dynamically ranked based on community consensus, influencing their visibility.
- **Implementation Notes:**

Backend: APIs manage voting (ensuring unique votes per user per item). Updates article and annotation scores in MongoDB.

Conceptual Snippet (Backend - Voting Logic):

JavaScript

```
// routes/voteRoutes.js
router.post('/articles/:id/vote', protect, async (req, res) => {
  const { id } = req.params;
  const { type } = req.body; // 'upvote' or 'downvote'
  try {
    const article = await Article.findById(id);
    if (!article) return res.status(404).json({ message: "Article not found" });

    // Prevent duplicate votes from the same user
    // (Requires a 'voters' array on the Article model or a separate Vote model)
    const existingVote = await Vote.findOne({ userId: req.user.id, targetId: id, targetType: 'article' });
    if (existingVote) return res.status(400).json({ message: "Already voted." });

    if (type === 'upvote') {
      article.upvotes += 1;
    } else if (type === 'downvote') {
      article.downvotes += 1;
    }
    await article.save();
    await new Vote({ userId: req.user.id, targetId: id, targetType: 'article', type }).save();
```

```
        res.status(200).json(article);
    } catch (error) {
        res.status(500).json({ message: "Voting failed." });
    }
});
```

Frontend: Provides upvote/downvote buttons and prominently displays current vote counts.

2.3 Collaborative Fact-Checking & Annotation System

- **Description:** Users can highlight specific claims directly within an article and submit supporting or refuting evidence (e.g., links to reputable studies, official reports, credible news sources). Other users can then vote on the credibility of these submitted annotations.
- **Implementation Notes:**

Backend: APIs for creating annotations linked to specific text ranges within an article, storing associated evidence URLs. MongoDB schema manages annotations.

Conceptual Snippet (Backend - Annotation Model):

JavaScript

```
// models/Annotation.js
const mongoose = require('mongoose');

const AnnotationSchema = new mongoose.Schema({
    articleId: { type: mongoose.Schema.Types.ObjectId, ref: 'Article', required: true },
    highlightedText: { type: String, required: true },
    startIndex: { type: Number, required: true }, // For accurate highlighting
    endIndex: { type: Number, required: true },   // For accurate highlighting
    claim: { type: String, required: true },
    evidenceUrl: { type: String }, // URL to supporting/refuting evidence
    submittedBy: { type: mongoose.Schema.Types.ObjectId, ref: 'User', required: true },
    createdAt: { type: Date, default: Date.now },
    credibilityVotes: { type: Number, default: 0 }, // Sum of credible/not-credible votes
    // ... other fields
});
module.exports = mongoose.model('Annotation', AnnotationSchema);
```

Frontend: Utilizes a rich text editor or a custom solution to enable text highlighting. Provides UI elements for submitting evidence and voting on annotation credibility.

Conceptual Snippet (Frontend - Text Highlighting/Annotation):

JavaScript

```
// components/ArticleReader.js

import React, { useState } from 'react';

import axios from 'axios';


function ArticleReader({ article }) {

  const [selectedText, setSelectedText] = useState(null);

  const [showAnnotationForm, setShowAnnotationForm] = useState(false);

  const [evidenceUrl, setEvidenceUrl] = useState('');

  const [claimText, setClaimText] = useState('');


  const handleMouseUp = () => {

    const selection = window.getSelection();

    if (selection.toString().length > 0) {

      setSelectedText({

        text: selection.toString(),

        startIndex: selection.anchorOffset,

        endIndex: selection.focusOffset,

        // You might need more complex range handling for real HTML content

      });

      setShowAnnotationForm(true);

    } else {

      setShowAnnotationForm(false);

      setSelectedText(null);

    }

  };
```

```
const handleSubmitAnnotation = async () => {
  if (selectedText && claimText && evidenceUrl) {
    try {
      await axios.post(`/api/articles/${article._id}/annotations`, {
        highlightedText: selectedText.text,
        startIndex: selectedText.startIndex,
        endIndex: selectedText.endIndex,
        claim: claimText,
        evidenceUrl: evidenceUrl
      });
      alert('Annotation submitted!');
      setShowAnnotationForm(false);
      setClaimText('');
      setEvidenceUrl('');
      // Refresh annotations for the article
    } catch (error) {
      console.error("Error submitting annotation:", error);
    }
  }
};

return (
  <div onMouseUp={handleMouseUp}>
    <h2>{article.title}</h2>
    <p>{article.fullContent}</p> {/* Render full content */}

    {showAnnotationForm && (
      <div className="annotation-form">
        <h4>Annotate selected text: "{selectedText?.text}"</h4>
```

```
          <textarea placeholder="Your claim about this text" value={claimText} onChange={(e) =>
setClaimText(e.target.value)} />
          <input type="text" placeholder="URL to evidence" value={evidenceUrl} onChange={(e) =>
setEvidenceUrl(e.target.value)} />
          <button onClick={handleSubmitAnnotation}>Submit Annotation</button>
          <button onClick={() => setShowAnnotationForm(false)}>Cancel</button>
        </div>
      )}
      {/* Display existing annotations below or within the text */}
    </div>
  );
}
```

2.4 Source Reliability Scoring & Transparency Metrics

- **Description:** The platform tracks and assigns a historical accuracy/bias score to different news sources based on the collective outcome of community fact-checks (e.g., if multiple claims from a particular source are consistently debunked, its reliability score decreases).
- **Implementation Notes:**
  - **Backend:** Implements logic to calculate and continuously update source scores based on annotation votes and article debunking statuses. Stores and manages source-specific data.
  - **Frontend:** Displays calculated reliability scores prominently alongside articles from the respective sources, enhancing transparency for users.

2.5 Topic-Based Subscriptions & Real-time Alerts

- **Description:** Users can subscribe to specific news categories or keywords of interest and receive real-time notifications for newly submitted articles or critical fact-checks related to their chosen topics.
- **Implementation Notes:**
  - **Backend:** APIs manage user subscriptions. Socket.IO can be used for real-time alerts or a Web Push API for browser notifications can be implemented.
  - **Frontend:** Provides a user interface for managing subscriptions and displays incoming real-time alerts.

2.6 User Reputation System (Gamified)

- **Description:** Users earn reputation points for valuable contributions to the platform, such as submitting high-quality, accurate articles, providing well-researched fact-checks, and engaging in constructive discussions. A higher reputation can grant users more influence in voting or moderation tasks.

- **Implementation Notes:**
    - **Backend:** Implements logic to assign and track reputation points based on user actions and the community's validation of their contributions.
    - **Frontend:** Displays user reputation prominently on their profiles and alongside their contributions (e.g., comments, annotations).

2.7 Admin & Moderation Tools

- **Description:** A dedicated admin panel provides comprehensive tools to oversee content submissions, review articles or annotations flagged by the community, manage user accounts, and resolve community disputes.
- **Implementation Notes:**
    - **Backend:** Develops admin-specific APIs for moderation actions, including content approval/rejection, user suspension, and dispute resolution.
    - **Frontend:** A dedicated admin dashboard UI provides the necessary tools for content review queues, user management interfaces, and detailed analytics for platform oversight.

3.0 API Endpoints (Conceptual)

(Note: BASE_URL refers to your deployed backend API URL, e.g., https://api.yourproject.com/v1)

3.1 Authentication & User Management

- POST /auth/register - User Registration
- POST /auth/login - User Login
- GET /auth/me - Get Current User Profile
- GET /users/:id/reputation - Get User Reputation
- GET /admin/users - Get all users (Admin only)
- PUT /admin/users/:id/status - Update User Status (Admin only)

3.2 Articles & Content Submission

- POST /articles - Submit a New News Article
- GET /articles - Get All Articles (News Feed)
- GET /articles/:id - Get Specific Article Details
- PUT /articles/:id/status - Update Article Status (e.g., pending review, approved, debunked - Admin/Moderator)
- GET /categories - Get all available categories
- GET /tags - Get all available tags

3.3 Voting

- POST /articles/:id/vote - Vote on an Article (up/down)
- POST /annotations/:id/vote - Vote on an Annotation (credible/not credible)

3.4 Fact-Checking & Annotations

- POST /articles/:articleId/annotations - Create New Annotation for a Claim in an Article
- GET /articles/:articleId/annotations - Get All Annotations for an Article
- GET /annotations/:id - Get Specific Annotation Details
- PUT /annotations/:id - Update Annotation (e.g., status - Moderator)
- POST /annotations/:id/evidence - Add Evidence to an Annotation

3.5 Source Reliability

- GET /sources - Get List of All Tracked News Sources
- GET /sources/:id - Get Details & Reliability Score for a Specific Source
- GET /sources/:id/articles - Get Articles from a Specific Source

3.6 Subscriptions & Notifications

- POST /subscriptions - Subscribe to a Topic/Keyword
- GET /subscriptions - Get User's Active Subscriptions
- DELETE /subscriptions/:id - Unsubscribe from a Topic/Keyword
- *(Socket.IO Events for Real-time Alerts)*: newArticleAlert, factCheckUpdate

3.7 Admin & Moderation

- GET /admin/articles/pending - Get Articles Pending Review
- GET /admin/annotations/flagged - Get Flagged Annotations
- POST /admin/moderate/article/:id - Moderate Article (approve/reject/flag)
- POST /admin/moderate/annotation/:id - Moderate Annotation
- GET /admin/analytics/overview - Get Overall Platform Statistics

4.0 Week-wise Development Plan

Week	Backend (Node.js + Express)	Frontend (React.js)
Week 1	Auth APIs, News Article CRUD APIs, Category APIs	React Setup, Auth UI, News Feed Display, Article Submission
Week 2	Voting APIs (Articles/Comments), Annotation APIs, Fact-Check Submission APIs	Article Detail View, Voting UI, Annotation UI, Commenting
Mid	**Auth, Article Management, and Core**	**News Feed, Article View, and Basic**

Week	Backend (Node.js + Express)	Frontend (React.js)
Project Review	**Voting/Annotation APIs implemented**	**Interaction UI connected to APIs**
Week 3	Source Scoring Logic, Subscription APIs, Reputation System APIs	Fact-Check Review UI, Source Reliability Display, Subscription Management
Week 4	Admin/Moderation APIs, Real-time Notification Setup, API Testing	Admin Panel UI, Notification UI, Responsive UI Polish
Final Project Review	**All modules functional: Content Submission, Fact-Checking, Source Scoring, Community Management, Admin Controls**	**Fully responsive frontend: News Feed, Fact-Checking Interface, Admin/User Dashboards**

5.0 Development Steps

1. **Project Setup & Authentication:** Initialize the MERN stack development environment. Implement JWT-based authentication for user and administrator roles, including secure password hashing.
2. **Article Submission & News Feed:** Develop MongoDB schemas and Express APIs for submitting and retrieving news articles (including fields for URL, title, content, categories, tags). Build the React frontend for a dynamic news feed and a form for users to submit new articles.
3. **Basic Voting System:** Implement backend APIs to allow users to upvote or downvote articles. Integrate this functionality into the frontend with corresponding UI elements to display vote counts.
4. **Annotation & Fact-Checking Core:** Create backend APIs for users to highlight specific text within articles and submit annotations with supporting or refuting evidence URLs. Develop sophisticated React components that allow for text selection and display annotations within the article view.
5. **Community Validation of Annotations:** Extend the voting system to annotations, enabling other users to vote on the credibility of submitted fact-checks. Update annotation credibility scores based on these votes.
6. **Source Reliability Scoring:** Implement a backend algorithm to calculate and continuously update a reliability score for each news source based on the collective outcomes of fact-checks associated with its articles. Display these scores transparently on the frontend.
7. **Topic Subscriptions & Real-time Alerts:** Develop APIs for users to subscribe to specific news categories or keywords. Implement a real-time notification system (e.g., using Socket.IO for in-app alerts or Web Push API for browser notifications) to alert users about relevant new articles or critical fact-checks.
8. **User Reputation System:** Design and implement a backend system to award reputation points to users based on the quality and community validation of their contributions (e.g., highly upvoted articles, accurate fact-checks). Display reputation on user profiles.

9. **Admin & Moderation Tools:** Create a dedicated admin panel with specific backend APIs for moderation tasks. This includes reviewing submitted articles, managing flagged annotations, approving/rejecting content, and user management functionalities.
10. **Refinement, Testing & Security:** Conduct rigorous unit, integration, and end-to-end testing. Implement robust data validation and error handling across all API endpoints. Focus heavily on security measures to prevent spam, manipulation, and ensure data integrity. Optimize performance for responsive interaction.
11. **Deployment:** Prepare the complete MERN stack application for live deployment. Host the React frontend on a static hosting service (e.g., Vercel, Netlify) and the Node.js backend with MongoDB on a scalable cloud platform (e.g., Render, AWS, DigitalOcean), ensuring all environment variables are securely configured.

Common Features Across All Projects

- **Secure JWT-based Authentication:** All projects will feature robust user authentication using JSON Web Tokens (JWT), supporting multiple user roles (e.g., User, Admin, Provider/Chef/Artisan).
- **MongoDB Schema Design and Data Modeling:** Each project requires careful design of MongoDB schemas to efficiently store and retrieve complex, interconnected data.
- **RESTful API Development using Express.js:** The backend of all projects will be built with Node.js and Express.js, exposing well-structured RESTful APIs for client-server communication.
- **Protected Routes and Component-based React Architecture:** Frontend React applications will consume these APIs, implementing protected routes based on user authentication and role. A component-driven architecture ensures modularity and reusability.
- **Razorpay/Stripe Payment Gateway Integration:** For projects involving transactions, secure payment gateway integration (either Razorpay or Stripe) will be implemented to handle online payments.
- **Mobile-Responsive, Component-Driven UI Design:** All projects prioritize a mobile-first approach, ensuring responsive user interfaces that adapt to various screen sizes. UI development will follow a component-driven methodology.
- **Admin Dashboards for Centralized Management:** Each project will include a dedicated admin panel, providing centralized control and monitoring capabilities for managing users, content, and system configurations.
- **Full-Stack Deployment:** The entire MERN stack application, including both the frontend (React) and backend (Node.js/Express with MongoDB), will be deployed to a live environment.