**Project Documentation**

Cook book : your virtual kitchen assistant

# 1. Introduction

* **Project Title:** Cook Book your virtual assistant
* **Team ID** :NM2025TMID29894
* **Team Leader:** SARANYA S
* **LEADER MAIL ID :** [202400954@sigc.edu](mailto:202400954@sigc.edu)
* **Team Members:**
* SARU LATHA A – [202400689@sigc.edu](mailto:202400689@sigc.edu)
* RINOSHLINE A – [202400108@sigc.edu](mailto:202400108@sigc.edu)
* PREETHI N – [202400892@sigc.edu](mailto:202400892@sigc.edu)

# 2. Project Overview

Purpose: The primary purpose of the Virtual Assistant Cookbook project is to develop an easily accessible, dynamic, and practical guide to creating, managing, and optimizing virtual assistants. This project aims to serve both technical developers and non-technical users who are interested in building or using AI-powered assistants for personal, business, or educational purposes.

* **Features:**

\* **Project Posting and Bidding:**

* Users can post projects and place bids, enabling easy collaboration between clients and service providers.

\* **Secure Chat System:**

* Built-in encrypted messaging for real-time communication between users, ensuring privacy and security.

\* **Feedback and Review System:**

* Clients and service providers can leave ratings and reviews, fostering transparency and trust.

\* **Admin Control Panel:**

* Admins can manage users, monitor activities, and control platform settings from a centralized dashboard.

# 3. Architecture

-> Frontend:

* **User Interface (UI):**
  + A responsive web/mobile interface where users can post projects, bid, chat, and leave feedback.
  + Built with frameworks like React.js, Angular, or Vue.js for dynamic rendering.
* **Backend:**
* **API Layer:**
  + RESTful APIs to handle requests between the frontend and the server (Node.js, Django, or Laravel).
  + Manages authentication, project posting, bidding, and feedback submission.
* **Real-Time Communication:**
  + WebSocket-based service for the secure chat system (Socket.io, Firebase).
  + Ensures instant communication between users.
* **Database:**
* **Relational Database (SQL):**
  + Stores user data, project information, bids, reviews, and feedback (MySQL, PostgreSQL).
  + Structured tables for projects, bids, user profiles, etc.

# 4. Setup Instructions

* **Prerequisites:**

Before you begin, ensure that the following tools and services are installed and set up:

* + Node.js : It is required to run the backend server and manage dependencies.

\* **Installation:**

* + Download and install Node.js from here.
* Verify installation:
  + node -v
  + npm -v

 MongoDB

It is used to store recipe data, user profiles, and other application data.

 **Installation:**

* Install MongoDB by following the official MongoDB setup guide.
* Alternatively, you can use a cloud service like [MongoDB Atlas](https://www.mongodb.com/cloud/atlas).
* After installation, start the MongoDB service:

sudo systemctl start mongodb

sudo systemctl enable mongodb

 Git is necessary for version control and managing your source code.

 **Installation:**

* Download Git from [here](https://git-scm.com/).
* Verify installation:
  + git --version
  + React.js

 React.js is used to build the frontend of your cookbook application.

 **Installation:**

* If you don't have create-react-app installed, use the following:
  + npm install -g create-react-app
  + Express.js **–** Mongoose **–** Visual Studio Code

npm install -g create-react-app

-Create a new React project (or clone your existing repo):

create-react-app cookbook-app

Install necessary dependencies by navigating to the project directory:

cd cookbook-app

npm install

* **Installation Steps:**
* Clone the Repository  
  Clone the project repository to your local machine:

git clone <repository-url>

* Install Client Dependencies  
  Navigate to the client directory and install the frontend dependencies:

cd client

npm install

* Install Server Dependencies  
  Navigate to the server directory and install the backend dependencies:

cd ../server

npm install

# Start the Server and Client

* Start the Backend Server:  
  From the server directory:

npm start

* Start the Frontend (React App):  
  From the client directory:

npm start

# 5. Folder Structure

# SB-Works/

# │

# ├── client/ # React frontend

# │ ├── components/ # Reusable UI components (e.g., buttons, forms)

# │ └── pages/ # React components representing different pages/views

# │

# └── server/ # Node.js backend

# ├── routes/ # API route handlers (e.g., /api/recipes, /api/users)

# ├── models/ # Database models (e.g., Recipe, User schema)

# └── controllers/ # Business logic for routes (e.g., createRecipe, getRecipes)

# ->folder details

**Folder Details:**

* **client/**: Contains all the frontend code.
  + **components/**: Holds reusable components (buttons, form fields, etc.).
  + **pages/**: Contains React components for different pages like Home, Recipe List, Recipe Detail, etc.
* **server/**: Contains the backend code (API and business logic).
  + **routes/**: Defines API endpoints and links them to controller functions.
  + **models/**: Defines the database schemas for MongoDB (e.g., Recipe model, User model).
  + **controllers/**: Contains the logic to handle requests (e.g., adding a new recipe, fetching recipes).

# 6. Running the Application

**Frontend:**

1. Navigate to the client directory:

cd client

2. Start the React frontend:

npm start

**Backend:**

1. Navigate to the server directory:

cd server

1. Start the Node.js backend:

npm start

**Access the Application:**

1.Once both the frontend and backend are running, open your browser and go to:

http://localhost:3000

# 7. API Documentation

**User**

1. **POST** /api/user/register
   * **Description**: Register a new user.
   * **Request Body**:
   * {
   * "username": "user123",
   * "email": "user@example.com",
   * "password": "securepassword"

}

* + **Response**:
  + {
  + "message": "User successfully registered",
  + "userId": "12345"
  + }

1. **POST** /api/user/login
   * **Description**: Login an existing user and get an authentication token.
   * **Request Body**:
   * {
   * "email": "user@example.com",
   * "password": "securepassword"
   * }
   * **Response**:
   * {
   * "message": "Login successful",
   * "token": "jwt-token-here"
   * }

**Projects**

1. **POST** /api/projects/create
   * **Description**: Create a new project.
   * **Request Body**:
   * {
   * "title": "Cookbook Website",
   * "description": "A platform to share and discover recipes.",
   * "budget": 5000,
   * "category": "Web Development"
   * }
   * **Response**:
   * {
   * "message": "Project created successfully",
   * "projectId": "7890"
   * }
2. **GET** /api/projects/:id
   * **Description**: Get details of a specific project by ID.
   * **Response**:
   * {
   * "projectId": "7890",
   * "title": "Cookbook Website",
   * "description": "A platform to share and discover recipes.",
   * "budget": 5000,
   * "category": "Web Development"
   * }

**Applications**

1. **POST** /api/apply
   * **Description**: Apply for a project.
   * **Request Body**:
   * {
   * "projectId": "7890",
   * "userId": "12345",
   * "coverLetter": "I am the best candidate for this project because..."
   * }
   * **Response**:
   * {
   * "message": "Application submitted successfully"
   * }

**Chats**

1. **POST** /api/chat/send
   * **Description**: Send a message in the chat system.
   * **Request Body**:
   * {
   * "senderId": "12345",
   * "receiverId": "67890",
   * "message": "Hello, I'm interested in your project!"
   * }
   * **Response**:
   * {
   * "message": "Message sent successfully"
   * }
2. **GET** /api/chat/:userId
   * **Description**: Get all chats for a specific user.
   * **Response**:
   * {
   * "chats": [
   * {
   * "chatId": "1",
   * "userId": "12345",
   * "messages": [
   * {
   * "sender": "12345",
   * "message": "Hello, are you available to discuss the project?"
   * },
   * {
   * "sender": "67890",
   * "message": "Yes, let's chat!"
   * }
   * ]
   * }
   * ]

}

# 8.AUTHENTICATION:

# JWT-based Authentication

Description: The application uses JSON Web Tokens (JWT) to secure login and protect private routes.

Flow:

User Login: When a user logs in, the server validates their credentials.

JWT Generation: Upon successful login, a JWT token is generated and sent to the client.

Token Storage: The client stores the token (usually in local Storage or session Storage for web applications).

Authenticated Requests: For future requests to protected routes, the client includes the token in the Authorization header as a Bearer token.

Example of Login Request:

POST /api/user/login

Request Body:

{

"email": "user@example.com",

"password": "secure password"

}

Response:

{

"message": "Login successful",

"token": "jwt-token-here"

}

Middleware for Protecting Private Routes

Description: Middleware functions are used to protect private routes by verifying the JWT token in the Authorization header. Only authenticated users with a valid token can access these routes.

JWT Middleware Logic:

The server checks the Authorization header for a Bearer token.

If the token is missing or invalid, the request is rejected with a 401 Unauthorized response.

If the token is valid, the user’s identity is decoded and attached to the request object (req.user), allowing the route handler to access user data.

Example Middleware Implementation (Node.js/Express):

const jwt = require('json web token');

const authenticate Token = (req, res, next) => {

const token = req.header('Authorization')?.split(' ')[1]; // Extract the token from Bearer token

if (!token) {

return res.status(401).json({ message: "Access denied. No token provided." });

}

try {

const decoded = jwt.verify(token, process.env.JWT\_SECRET); // Verify the token

req.user = decoded; // Attach the decoded user data to the request object

next(); // Continue to the next middleware or route handler

} catch (err) {

return res.status(400).json({ message: "Invalid token." });

}

};

module.exports = authenticate Token;

Protecting Routes:  
To protect a route, simply use the authenticate Token middleware before your route handler:

const express = require('express');

const router = express.Router();

const authenticate Token = require('./middleware/authenticate Token');

router.get('/protected-route', authenticate Token, (req, res) => {

res.json({ message: "You have access to this protected route.", user: req.user });

});

module.exports = router;

# 

# 9. User Interface

**Landing Page**

* **Purpose**: The landing page serves as the entry point for the users (both freelancers and clients). It provides a brief overview of the platform, key features, and actions to guide the user into signing up or logging in.
* **Key Components**:
  1. **Navigation Bar**:
     + Logo
     + Links to Sign Up, Login, About, and Contact
  2. **Hero Section**:
     + Eye-catching banner with a call-to-action (CTA) like "Start your freelance journey" or "Find your next project."
  3. **Features Section**:
     + Short descriptions of what the platform offers (e.g., "Post Projects", "Apply for Jobs", "Secure Payments").
  4. **How It Works**:
     + A brief explanation of the platform's process (posting a job → applying for a job → working with clients).
  5. **Testimonials/Reviews** (Optional):
     + Quotes or ratings from happy freelancers or clients, building trust and credibility.
  6. **Footer**:
     + Contact info, links to privacy policy, terms of service, and social media.
* **Example Layout**:
  1. **Hero Section**: Image/Illustration + CTA button
  2. **Features Section**: Icons with short descriptions
  3. **Footer**: Basic links to legal and social pages

**Freelancer Dashboard**

* **Purpose**: The freelancer dashboard is the personal workspace for freelancers where they can manage their profile, applications, projects, and communication with clients.
* **Key Components**:
  1. **Profile Overview**:
     + Display user info like name, profile picture, rating, and short bio.
  2. **My Projects**:
     + List of projects the freelancer has posted or is currently working on, with status updates (e.g., "In Progress", "Completed", etc.).
     + Option to create a new project or edit existing ones.
  3. **Applications**:
     + View applied projects and track the status of applications (Accepted, Pending, Rejected).
     + A button to apply for new projects.
  4. **Messages/Chat**:
     + Real-time chat functionality with clients and other freelancers.
     + Display recent conversations with clients and quick access to ongoing chats.
  5. **Notifications**:
     + Display alerts or updates regarding new project invitations, application status changes, or new messages.
  6. **Earnings and Stats**:
     + Overview of earnings, including paid projects, current balance, and payment history.
  7. **Settings**:
     + Profile settings, notification preferences, account management, etc.
* **Example Layout**:
  1. **Top Navigation Bar**: Dashboard link, Profile link, Settings, Logout
  2. **Left Sidebar**: Quick links to "My Projects", "Applications", "Messages", "Notifications"
  3. **Main Content Area**: Sections like "My Projects", "Applications", "Chat", and "Earnings" with cards or lists

Top of Form

Bottom of Form

# Testing

**1. Unit Testing**

**Purpose**: Test individual components and backend functions.

**Tools**: Jest, React Testing Library (Frontend), Mocha/Chai (Backend)

**Example**: Test that recipe details render correctly on the UI.

**2. Integration Testing**

**Purpose**: Test how frontend and backend work together.

**Tools**: Cypress (Frontend), Super test (Backend)

**Example**: Test that submitting a recipe form stores it in the database and displays on the UI.

**3. API Testing**

**Purpose**: Test API endpoints for CRUD operations.

**Tools**: Super test, Postman

**Example**: Ensure POST /api/recipes/create creates a new recipe.

**4. UI/UX Testing**

**Purpose**: Ensure a responsive, user-friendly UI.

**Tools**: Cypress, Lighthouse

**Example**: Check if the mobile view displays the navigation correctly.

**5. Performance Testing**

**Purpose**: Test app performance under load.

**Tools**: Lighthouse, JMeter

**Example**: Measure page load time and API response time.

# Run Tests:

**Frontend**: npm test

**Backend**: npm test

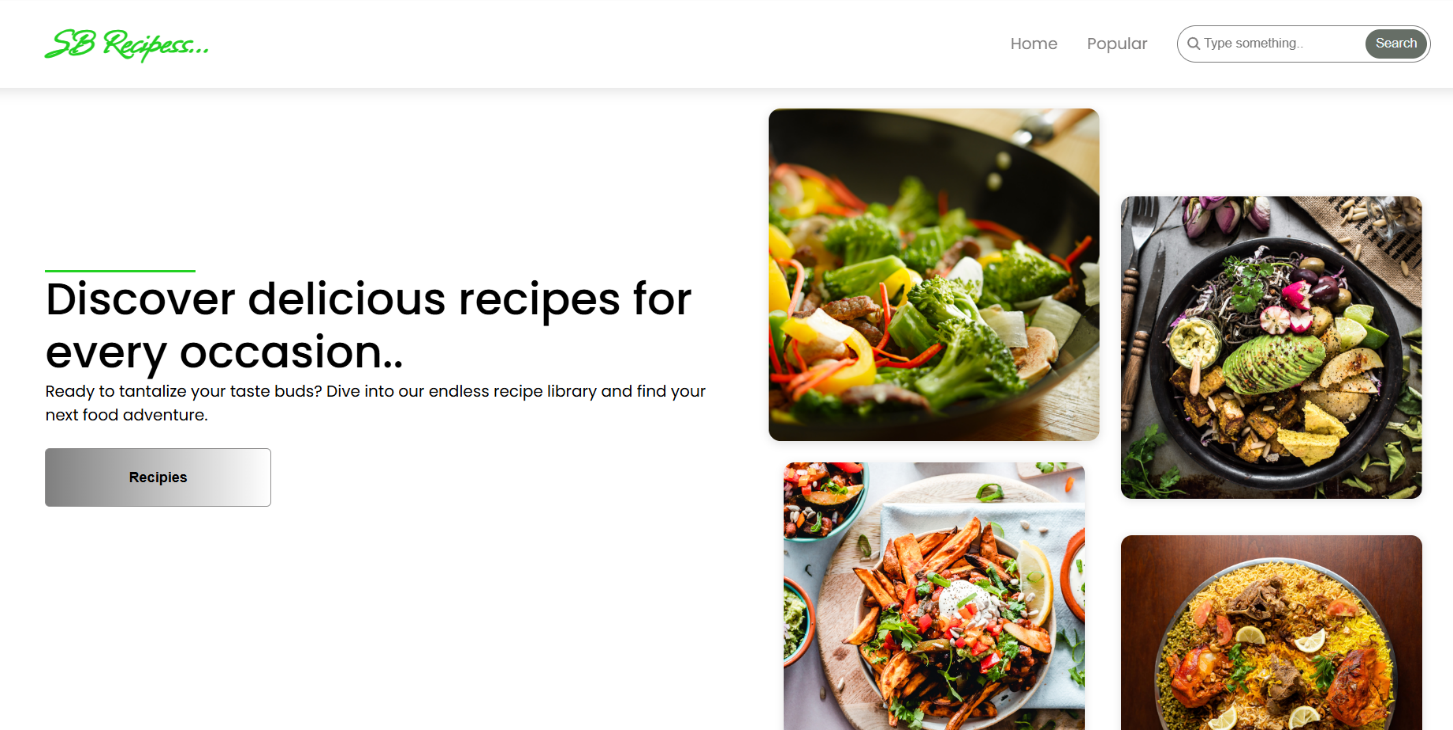
**E2E (Cypress)**: npx cypress open

1. **Known Issues**
2. **Search Functionality**:
   * **Issue**: Search results may not always be accurate due to partial matches in the recipe database.
   * **Solution**: We are improving the search algorithm for better accuracy.
3. **Responsive Layout**:
   * **Issue**: Some mobile devices might not display the recipe card grid properly on smaller screens.
   * **Solution**: UI improvements are in progress to ensure a consistent layout across all devices.
4. **Recipe Image Upload**:
   * **Issue**: Occasionally, images fail to upload or display in the recipe details.
   * **Solution**: We are optimizing image upload functionality and server storage.
5. **User Authentication**:
   * **Issue**: Users may experience occasional delays when logging in or logging out due to session management.
   * **Solution**: Session handling improvements are being worked on.
6. **Performance**:
   * **Issue**: Loading times for recipe lists can be slower when there are a large number of entries.
   * **Solution**: Pagination and lazy-loading features are being implemented to improve performance.

# Future Enhancements

* **Advanced Search Filters**: Add filters for cuisine, diet, and cooking time.
* **User Ratings & Reviews**: Allow users to rate and review recipes.
* **Recipe Categories**: Organize recipes by categories (e.g., Desserts, Quick Meals).
* **Meal Planner**: Help users plan meals for the week.
* **Shopping List**: Generate shopping lists based on selected recipes.
* **Social Sharing**: Let users share recipes on social media.
* **Push Notifications**: Notify users about new recipes and updates.
* **Multilingual Support**: Support multiple languages for international users.

# 13.ScreenShot



# 14.Demo video link

<https://drive.google.com/file/d/1Kwfxot5BWq7VB30rzea14qHAXxEUzv34/view?usp=drive_link>

# THANK YOU!!!