Application Documentation

"LetHimCook - RecipeFinder"

The aim of this project is to build a recipe-finding web application that can detect ingredients from an image and provide relevant recipes based on those ingredients.

Application Overview

The application consists of two main components: a frontend user interface built with React and a backend server implemented using Node.js and Express. The frontend enables users to upload an image of ingredients, while the backend processes the image, extracts the ingredient list using the Google Vision API, and retrieves recipes based on the detected ingredients using the Spoonacular API.

Key Features:

- Image-based ingredient detection using Google Vision API.
- Recipe retrieval based on detected ingredients through Spoonacular API.
- Responsive web design for seamless user experience on different devices.
- Cloud-based deployment ensuring global accessibility and scalability.

Architecture of the Application

The architecture of the application follows a modern web development approach that separates concerns between the frontend and backend:

- **Frontend**:
  - **Technology Stack**: React (JavaScript, HTML, CSS)
  - **Deployment Platform**: Vercel
  - **Core Components**:
    - **Image Upload**: Allows users to upload an image of ingredients.
    - **Recipes List**: Displays a list of relevant recipes retrieved from the Spoonacular API.
    - **User Interactivity**: Dynamic elements such as a scroll arrow, search results, and error handling.

- **Backend**:
  - **Technology Stack**: Node.js, Express
  - **Deployment Platform**: Heroku
  - **Core Functions**:
    - **Image Processing**: Receives the image, sends it to the Google Vision API, processes the returned text, and cleans the detected ingredients.
    - **Recipe Retrieval**: Communicates with the Spoonacular API, sending the cleaned ingredient list to retrieve matching recipes.
    - **API Endpoints**:
      - /detect-ingredients: Processes the uploaded image and returns detected ingredients.
      - /recipes: Fetches recipes based on the list of ingredients.

- **External APIs**:
  - **Google Vision API**: Used to detect ingredients in uploaded images.
  - **Spoonacular API**: Used to retrieve recipes based on the detected ingredients.

Technologies Used

The following technologies were used to develop and deploy the application:

- **React**: For the frontend interface.
- **Node.js & Express**: For the backend API.
- **Google Vision API**: For image recognition and ingredient extraction.
- **Spoonacular API**: For fetching recipes based on the ingredients.
- **Multer**: For handling image uploads in the backend.
- **Axios**: For making HTTP requests from both the frontend and backend.
- **Heroku**: For deploying the backend.
- **Vercel**: For deploying the frontend.

Development Process

The project was built incrementally, starting with setting up both frontend and backend environments locally, followed by API integration and cloud deployment.

Setting Up the Frontend

The frontend was built using React, focusing on a simple user interface where users could upload an image of their ingredients. Key components include:

RecipesList: Displays the recipes returned from the backend.

ImageIngredientDetection: Handles the image upload and sends it to the backend for processing. The frontend was continuously tested locally before being deployed to Vercel, ensuring smooth API communication with the backend.

Setting Up the Backend

The backend was built using Node.js and Express. It features two main routes:

/detect-ingredients: This route handles image uploads, processes the image using the Google Vision API, and returns the detected ingredients.

/recipes: This route takes the detected ingredients and fetches relevant recipes from the Spoonacular API.

Integration with APIs

Google Vision API was integrated to detect text (ingredients) from images uploaded by users.

Spoonacular API was integrated to fetch recipes based on the ingredients provided by the Google Vision API.

Cloud Deployment

The frontend was deployed on Vercel, and the backend was deployed on Heroku. During deployment, certain configurations such as binding to a dynamic port for Heroku and ensuring cross-origin requests (CORS) were handled.

Testing

The application was tested locally using Postman for API requests. The deployed version was then tested to ensure communication between Vercel (frontend) and Heroku (backend).

Challenges Encountered

Localhost vs Production URLs

Initially, the frontend was hardcoded to use the localhost URL for making API requests. This had to be updated to the Heroku URL after deployment to ensure the frontend could reach the backend in a production environment.

Handling Environment Variables

Ensuring that sensitive keys like API keys were not exposed required setting up environment variables both locally and on the cloud platforms (Heroku for backend).

Port Binding on Heroku

One issue faced during deployment on Heroku was related to binding the backend server to the dynamic port provided by Heroku. This was resolved by modifying the backend to use process.env.PORT.

CORS Issues

Cross-origin requests from the frontend (deployed on Vercel) to the backend (deployed on Heroku) caused CORS issues. This was handled by enabling CORS in the backend using the cors package.

Conclusion

The challenges encountered during the development process, particularly during deployment, provided valuable insights into cloud hosting and API integration. Learning to handle issues such as environment configuration, CORS, dynamic port binding, and API key management taught valuable lessons about building scalable, production-ready applications. These insights are transferable to future projects, particularly in how to ensure smooth deployment pipelines, secure handling of sensitive data (like API keys), and managing third-party dependencies.

This project underscores the importance of understanding not only how to build an application but also how to deploy it efficiently and securely on cloud platforms. The separation of frontend and backend components, combined with the integration of powerful external APIs, provides a scalable solution that is accessible to users anywhere in the world. The application architecture ensures that it can be easily extended in the future, making it a strong foundation for further development.