ChefMate Application Documentation

# Application Overview

## Overview of ChefMate's functionality and purpose.

# **Application Overview**

# ChefMate is a user-friendly application designed to simplify the process of finding and managing recipes based on user preferences and available resources. It integrates with OpenAI to provide personalized, dynamic, and easy-to-use functionality. Below is a detailed explanation of the app’s purpose and capabilities.

# **Purpose**

# ChefMate serves as a digital cooking assistant, catering to users who:

# 1. **Need Inspiration**: Users often struggle with deciding what to cook based on the ingredients they have at hand.

# 2. **Save Time**: The app eliminates the need for manual recipe searches by providing tailored suggestions.

# 3. **Want Convenience**: By streamlining recipe selection, saving, and sharing, ChefMate enhances the cooking experience.

# **Core Functionalities**

# 1. **Dynamic Recipe Search**:

# - Users can input groceries, meal type, cooking time, and number of diners.

# - The app fetches personalized recipes using OpenAI’s AI capabilities.

# 2. **Recipe Management**:

# - Users can view, save, and manage their favorite recipes.

# - Favorites are easily accessible in the Favorites section.

# 3. **Detailed Recipe Instructions**:

# - Each recipe includes the number of diners, cooking time, ingredients, and step-by-step instructions.

# - Recipes can be shared directly from the app.

# 4. **Profile Customization**:

# - Users can upload and manage their profile images, providing a personalized touch.

# 5. **Onboarding and Information**:

# - New users are guided through the app’s features, ensuring ease of use from the start.

# 6. **User Authentication**:

# - Secure phone-based login ensures personalized user experiences and data protection.

# **Key Highlights**

# 1. **Personalization**:

# - Recipes are tailored based on user preferences and available groceries.

# - The app supports multiple meal types, including vegan, vegetarian, meat, and dairy options.

# 2. **Accessibility**:

# - The interface is intuitive and visually appealing, making it easy for users of all ages to navigate.

# - Text-to-speech functionality ensures accessibility for users who prefer audio-based interactions.

# 3. **Efficiency**:

# - The app reduces the time and effort needed to find suitable recipes.

# - With favorites and sharing options, it enables users to organize and disseminate their favorite recipes effortlessly.

# 4. **Integration with AI**:

# - OpenAI powers the app’s core functionalities, ensuring accurate and creative recipe suggestions.

# - Validation, recipe generation, and detailed instructions are handled intelligently.

# **How ChefMate Transforms Cooking**

# ChefMate aims to transform the way people interact with recipes by:

# - **Simplifying the Search**: No more scrolling through endless recipes online. ChefMate offers tailored options in seconds.

# - **Promoting Smart Cooking**: By using what users already have, the app reduces waste and encourages creativity in the kitchen.

# - **Enabling Sharing and Discovery**: Users can share recipes with friends and family, promoting a sense of community.

# ChefMate is designed to bring joy and efficiency to cooking, making it not only a functional tool but also a delightful experience for all users.

# Core Application Features

## Recipe Search

The **Recipe Search** functionality is the heart of ChefMate, enabling users to discover recipes that match their preferences and available ingredients. This section explains the purpose, flow, and key components involved.

### - Purpose and functionality.

The Recipe Search feature provides:

**Personalized Recipe Suggestions**: Tailored to the user's input, including meal type, cooking time, number of diners, and groceries.

**Dynamic Input Handling**: Users can add groceries interactively and select meal preferences via dropdowns.

**Integration with AI**: The OpenAI API generates recipes dynamically based on user-provided data.

### - Flow: Steps from user input to recipe display.

1. **User Input**:
   * The user selects:
     + **Meal Type** (e.g., breakfast, lunch, dinner) using a dropdown menu managed by MealTypeSpinnerHandler.
     + **Cooking Time** (e.g., short, medium, long) using CookTimeSpinnerHandler.
     + **Number of Diners** through a numeric input field.
   * The user dynamically adds groceries to a list displayed in a FlexboxLayout.
2. **Validation and Processing**:
   * Inputs are packaged into a RecipeRequest object.
   * Groceries are validated against the selected meal type using OpenAI's validation prompt.
3. **Recipe Search**:
   * Upon successful validation, a request is sent to OpenAI to fetch recipe options.
4. **Display Recipe Titles**:
   * The API response, containing up to four recipe titles, is parsed.
   * Titles are displayed in RecipeOptionsActivity, allowing users to select one for detailed instructions.

### Validation Prompt

**Purpose**: To ensure that the provided groceries align with the selected meal type. This step reduces the likelihood of irrelevant or infeasible recipes being generated.

**Prompt Structure**:

You are a logic-based validator tasked with checking if the provided groceries match the meal type.

Meal type: 'MEAT'.

Groceries: ['chicken', 'tomato', 'cheese'].

IMPORTANT: Ignore all cultural, religious, or dietary laws (e.g., meat and dairy restrictions).

Return JSON: {'valid': '', 'reason': ''}.

**Key Components**:

1. **Meal Type**: The selected category for the recipe (e.g., MEAT, DAIRY).
2. **Groceries**: The list of user-provided ingredients.
3. **Rules**:
   * Focus only on technical feasibility.
   * Exclude any cultural or dietary considerations.

**Expected Response**:

* **Valid Input**:

json

{"valid": "true", "reason": ""}

* **Invalid Input**:

json

{"valid": "false", "reason": "Cheese is not a valid ingredient for a MEAT meal type."}

**Usage in Code**:

String validationPrompt = RecipePrompts.createValidationGroceriesForMealTypePrompt(

MealType.valueOf(mealTypeItem.getId()),

groceryList.toString()

);

callOpenAI(validationPrompt);

### - Key classes

**2.4.1. HomeActivity**

* **Role**: Acts as the starting point for the recipe search.
* **Responsibilities**:
  + Collect user input for meal preferences, cooking time, number of diners, and groceries.
  + Validate user inputs and interact with the OpenAI API.
* **Important Methods**:
  + createRecipeRequest(): Constructs a RecipeRequest object with the collected user inputs.
  + validateInputs(): Ensures all inputs are meaningful before processing.

**Example Code**:

if (validateInputs()) {

RecipeRequest recipeRequest = new RecipeRequest(

"", // Recipe ID (not used in this context)

Integer.parseInt(dinersInput.getText().toString()), // Number of diners

MealType.valueOf(mealTypeItem.getId()), // Meal type enum

CookTime.valueOf(cookTimeItem.getId()), // Cooking time enum

groceryList.toString() // List of groceries as a string

);

callOpenAI(RecipePrompts.createGetRecipeOptionsPrompt(

recipeRequest.getDiners(),

recipeRequest.getTime(),

recipeRequest.getGroceries(),

null // No previous options

));

}

**2.4.2. RecipeRequest**

* **Role**: Encapsulates user input data for API requests.
* **Properties**:
  + id: An identifier for the recipe (not used in the current context).
  + diners: Number of diners for the recipe.
  + mealType: The selected meal type (MealType enum).
  + time: The selected cooking time (CookTime enum).
  + groceries: A comma-separated string of groceries.

**2.4.3. OpenAI Integration**

* OpenAI prompts are created in RecipePrompts:
  + Validation Prompt: Ensures groceries align with the meal type.
  + Recipe Options Prompt: Requests up to four recipe titles.
* The prompts are sent using OpenAiService and responses are parsed.

## Recipe Options

The **Recipe Options** functionality is central to ChefMate’s user experience, allowing users to view tailored recipe suggestions based on their inputs. This section provides a detailed explanation of the purpose, flow, interaction with OpenAI, and the process for displaying recipe titles.

### - Purpose.

The Recipe Options feature:

1. **Displays Suggestions**: Shows up to four recipes matching the user's inputs, making it easy to choose relevant options.
2. **Provides Flexibility**: Users can reload recipes or proceed to detailed instructions for any selected recipe.
3. **Enhances Accessibility**: Includes text-to-speech functionality to read out recipe titles, improving accessibility for all users.

### Flow: Steps from Input Validation to Recipe Display

1. **Input Validation**:
   * Validated user inputs from HomeActivity are packaged into a RecipeRequest object.
2. **API Request**:
   * A prompt is created using RecipePrompts.createGetRecipeOptionsPrompt to fetch recipe options from OpenAI.
3. **Response Handling**:
   * The API response, a JSON array of recipe titles, is parsed into a list.
4. **Displaying Results**:
   * Recipe titles are dynamically displayed in RecipeOptionsActivity.
   * Users can select a recipe for detailed instructions or reload new options.

### - Interaction with OpenAI for fetching recipes.

**Prompt Structure**:

* **Purpose**: To request up to four recipe suggestions based on user inputs.
* **Prompt Example**:

Suggest four recipes for 2 diners using the following details:

- Groceries: ['eggs', 'tomato'].

- Cooking time: 10-30 minutes.

Return JSON as an array: ['Recipe 1', 'Recipe 2', 'Recipe 3', 'Recipe 4'].

**Key Elements**:

1. **Diners**: Number of people the recipe should serve.
2. **Cooking Time**: Range specified by the user (short, medium, or long).
3. **Groceries**: List of ingredients available for the recipe.

**Expected Response**:

* A JSON array of recipe titles:

["Omelette", "Tomato Soup", "Egg Salad", "Shakshuka"]

**Request Code**:

String prompt = RecipePrompts.createGetRecipeOptionsPrompt(

recipeRequest.getDiners(),

recipeRequest.getTime(),

recipeRequest.getGroceries(),

null // No previous options to exclude

);

callOpenAI(prompt);

**Response Parsing**:

* The OpenAI response is cleaned and parsed into a list of titles:

JSONArray recipesJsonArray = new JSONArray(cleanedResponse);

List<String> recipeTitles = new ArrayList<>();

for (int i = 0; i < recipesJsonArray.length(); i++) {

recipeTitles.add(recipesJsonArray.getString(i));

}

### - Displaying recipe titles.

**Activity**: RecipeOptionsActivity

1. **Receiving Data**:
   * Recipe titles are passed to RecipeOptionsActivity as a list.
   * The data is received through an Intent:

List<String> recipeTitles = getIntent().getStringArrayListExtra("recipeOptions");

1. **Dynamic UI Update**:
   * Titles are displayed in TextView components:

title1.setText(recipeTitles.get(0));

title2.setText(recipeTitles.get(1));

title3.setText(recipeTitles.get(2));

title4.setText(recipeTitles.get(3));

1. **Interactivity**:
   * Users can:
     + **Select a Recipe**: Clicking a title navigates to RecipeActivity for detailed instructions.
     + **Reload Options**: A button triggers a new API call to fetch fresh suggestions.
     + **Hear Titles**: Text-to-speech functionality reads out the recipe titles.

**Example Code for Text-to-Speech**:

SpeechService.speak(recipeTitles.get(0)); // Reads the first recipe title

### Key Classes and Methods

**RecipeOptionsActivity**:

Responsibilities:

* Fetch recipe titles using OpenAI.
* Display titles dynamically in the UI.
* Provide interactivity for selecting or reloading recipes.

**SpeechService:**

* Purpose: Converts text to audio for accessibility.
* Usage:

SpeechService.speak("Omelette");

**RecipePrompts:**

* Method: createGetRecipeOptionsPrompt
* Purpose: Constructs a prompt to fetch recipes based on user inputs.

**OpenAiService:**

* Method: callOpenAI
* Purpose: Sends the prompt to OpenAI and processes the response.

## Recipe Details

The **Recipe Details** functionality provides users with in-depth information about a selected recipe, including ingredients, preparation instructions, and additional details. This section explains its purpose, flow, and key steps from recipe selection to displaying detailed information.

### - Purpose and detailed explanation.

The Recipe Details feature:

1. **Displays Comprehensive Information**:
   * Includes the recipe title, number of diners, cooking time, ingredients list, and step-by-step instructions.
2. **Enables Interactivity**:
   * Users can save the recipe to their favorites or share it with others via the app’s built-in sharing options.
3. **Enhances Accessibility**:
   * Text-to-speech functionality allows users to hear the recipe details, making it accessible to a wider audience.

**Key Highlights**:

* Recipes are fetched dynamically from OpenAI based on the selected title.
* Users receive a complete, structured view of the recipe that they can act upon immediately.

### - Flow from selecting a recipe to displaying full details.

1. **Recipe Selection**:
   * Users select a recipe title from the RecipeOptionsActivity.
   * The selected title and associated RecipeRequest object are passed to RecipeActivity via an Intent:

Intent intent = new Intent(this, RecipeActivity.class);

intent.putExtra("selectedTitle", selectedTitle);

intent.putExtra("recipeRequest", recipeRequest);

startActivity(intent);

1. **Fetching Recipe Details**:
   * In RecipeActivity, a prompt is created to fetch detailed recipe data from OpenAI:

String prompt = RecipePrompts.createGetRecipePrompt(

selectedTitle, // Recipe title selected by the user

recipeRequest.getDiners(), // Number of diners

recipeRequest.getTime(), // Cooking time

recipeRequest.getGroceries() // List of groceries

);

callOpenAI(prompt);

* + **Prompt Structure**:

Create a recipe titled 'Omelette' for 3 diners with cooking time 10-20 minutes using ['eggs', 'cheese'].

Return JSON: {

'title': '',

'diners': '',

'groceries': [],

'time': '',

'instructions': []

}.

1. **Response Handling**:
   * The JSON response is cleaned and parsed into structured data:

JSONObject recipeJson = new JSONObject(cleanedResponse);

String title = recipeJson.getString("title");

int diners = recipeJson.getInt("diners");

JSONArray groceries = recipeJson.getJSONArray("groceries");

JSONArray instructions = recipeJson.getJSONArray("instructions");

* + The parsed data is used to create a Recipe object:

Recipe detailedRecipe = new Recipe(

title,

diners,

recipeRequest.getMealType(),

recipeJson.getInt("time"),

groceries.toString(),

instructions.toString()

);

1. **Displaying Recipe Details**:
   * The Recipe object is used to populate the UI elements in RecipeActivity:

titleView.setText(detailedRecipe.getTitle());

dinersView.setText(String.valueOf(detailedRecipe.getDiners()));

cookingTimeView.setText(detailedRecipe.getTime() + " minutes");

ingredientsView.setText(detailedRecipe.getGroceries());

instructionsView.setText(detailedRecipe.getInstructions());

1. **User Interaction**:
   * **Saving to Favorites**:
     + Clicking the "Favorite" button saves the recipe to the favorites list.

favoriteButton.setOnClickListener(view -> saveToFavorites(detailedRecipe));

* + **Sharing the Recipe**:
    - The "Share" button enables sharing via messaging apps or social media.

shareButton.setOnClickListener(view -> shareRecipe(detailedRecipe));

1. **Text-to-Speech**:
   * Users can hear the recipe details using SpeechService:

SpeechService.speak(detailedRecipe.getInstructions());

### Key Classes and Methods

**RecipeActivity**:

* **Role**: Manages fetching and displaying recipe details.
* **Responsibilities**:
  + Receives user selection and RecipeRequest.
  + Fetches detailed recipe information from OpenAI.
  + Populates the UI with detailed recipe data.
  + Handles user actions like saving and sharing recipes.

**RecipePrompts**:

* **Method**: createGetRecipePrompt
* **Purpose**: Constructs a detailed recipe prompt based on user inputs.

**Recipe**:

* **Properties**:
  + title: The recipe title.
  + diners: Number of diners.
  + mealType: Type of meal (from RecipeRequest).
  + time: Cooking time in minutes.
  + groceries: List of ingredients.
  + instructions: Step-by-step preparation guide.

**SpeechService**:

* **Purpose**: Provides text-to-speech functionality for recipe details.

## Favorites Management

The **Favorites Management** functionality allows users to save, view, and manage their favorite recipes within ChefMate. This section explains its purpose and the features that enable seamless interaction with saved recipes.

### Purpose

1. **Convenient Access**: Users can quickly access their favorite recipes without repeating searches.
2. **Organization**: Enables users to build a personalized collection of preferred recipes.
3. **Sharing and Editing**: Provides options to share or remove recipes from the favorites list.

**Features for Managing Saved Recipes**

1. **Adding Recipes to Favorites**:
   * Users can mark a recipe as a favorite directly from the RecipeActivity by clicking the "Favorite" button.
   * The recipe is saved to the favorites list in the FavoriteActivity.

favoriteButton.setOnClickListener(view -> saveToFavorites(currentRecipe));

1. **Viewing Saved Recipes**:
   * The FavoriteActivity displays all saved recipes in a FlexboxLayout.
   * Each recipe is represented by:
     + **Title**: Displayed in a TextView.
     + **Image**: An associated recipe image, if available.
     + **Actions**: Buttons for sharing and deleting the recipe.

**Layout Example**:

<LinearLayout

android:layout\_width="120dp"

android:layout\_height="wrap\_content"

android:orientation="vertical"

android:gravity="center"

android:background="@drawable/favorite\_border">

<TextView android:layout\_width="match\_parent" android:layout\_height="40dp" android:text="Recipe Title"/>

<ImageView android:layout\_width="120dp" android:layout\_height="120dp" android:src="@drawable/recipe\_image"/>

<LinearLayout android:layout\_width="match\_parent" android:layout\_height="wrap\_content" android:orientation="horizontal" android:gravity="center">

<ImageButton android:id="@+id/deleteButton" android:layout\_width="50dp" android:layout\_height="50dp" app:srcCompat="@android:drawable/ic\_menu\_delete" />

<ImageButton android:id="@+id/shareButton" android:layout\_width="50dp" android:layout\_height="50dp" app:srcCompat="@android:drawable/ic\_menu\_share" />

</LinearLayout>

</LinearLayout>

1. **Sharing Recipes**:
   * A "Share" button lets users share a recipe’s details via messaging apps, email, or social media.
   * The recipe details are converted into a shareable format.

shareButton.setOnClickListener(view -> {

String recipeDetails = currentRecipe.getTitle() + "\n" + currentRecipe.getGroceries() + "\n" + currentRecipe.getInstructions();

Intent shareIntent = new Intent(Intent.ACTION\_SEND);

shareIntent.setType("text/plain");

shareIntent.putExtra(Intent.EXTRA\_TEXT, recipeDetails);

startActivity(Intent.createChooser(shareIntent, "Share Recipe"));

});

1. **Deleting Recipes**:
   * Users can remove a recipe from their favorites by clicking the "Delete" button.
   * The app updates the saved recipes list and refreshes the UI.

deleteButton.setOnClickListener(view -> {

removeFromFavorites(recipeId);

favoriteLayout.removeView(view); // Removes the UI element

});

1. **Persistent Storage**:
   * Favorite recipes are stored persistently (e.g., using a database or local storage) to ensure they are available even after the app is restarted.
2. **Dynamic UI**:
   * Recipes in FavoriteActivity are dynamically displayed using a FlexboxLayout, allowing for flexible arrangements.

**Example Code for Dynamic UI**:

for (Recipe recipe : favoriteRecipes) {

GroceryItemView itemView = new GroceryItemView(this, recipe.getTitle(), null);

favoriteLayout.addView(itemView);

}

### Key Classes and Methods

**FavoriteActivity**:

* **Role**: Displays the list of favorite recipes and manages user interactions.
* **Responsibilities**:
  + Dynamically load and display saved recipes.
  + Handle user actions like sharing and deleting recipes.

**Recipe**:

* **Properties**:
  + id: Unique identifier for the recipe.
  + title: Recipe title.
  + groceries: Ingredients list.
  + instructions: Preparation steps.

**FavoriteManager** (if implemented as a helper class):

* **Role**: Manages storage and retrieval of favorite recipes.
* **Methods**:
  + saveToFavorites(Recipe recipe): Saves a recipe to favorites.
  + getFavorites(): Retrieves all saved recipes.
  + removeFromFavorites(String recipeId): Removes a recipe from favorites.

**User Experience**

The Favorites Management feature enhances the ChefMate app by:

1. Providing a **central hub** for saved recipes.
2. Allowing users to easily share and organize their favorite meals.
3. Ensuring recipes remain accessible through persistent storage.

## User Profile

The **User Profile** functionality allows users to personalize their experience in ChefMate by uploading and managing their profile images. This section explains the purpose and the steps involved in handling profile image uploads and management.

### Purpose

The User Profile feature provides:

1. **Personalization**:
   * Adds a personal touch by enabling users to upload their profile pictures.
   * Helps users distinguish their profiles in shared or multi-user environments.
2. **Improved User Engagement**:
   * Makes the app feel more tailored to the individual.
3. **Flexibility**:
   * Users can upload images from their gallery or capture new ones with their device’s camera.

### Steps to Upload and Manage Profile Images

1. **Accessing the Profile Page**:
   * Users navigate to ProfileActivity, which displays:
     + Current profile picture (or a default placeholder).
     + An upload button to add or update their profile image.

**UI Example (profile\_activity.xml)**:

<ImageView

android:id="@+id/profileImage"

android:layout\_width="150dp"

android:layout\_height="150dp"

app:srcCompat="@drawable/default\_profile\_icon" />

<Button

android:id="@+id/profile\_image\_upload\_button"

android:layout\_width="110dp"

android:layout\_height="wrap\_content"

android:text="Upload Image" />

1. **Handling Image Uploads**:
   * When the user clicks the "Upload Image" button, they can choose:
     + **Take a Photo**: Opens the device camera.
     + **Choose from Gallery**: Allows selection of an existing image.

**Code Example for Upload Options**:

profileImageUploadButton.setOnClickListener(view -> {

String[] options = {"Take Photo", "Choose from Gallery"};

AlertDialog.Builder builder = new AlertDialog.Builder(this);

builder.setTitle("Select Option")

.setItems(options, (dialog, which) -> {

if (which == 0) {

openCamera();

} else {

openGallery();

}

})

.show();

});

1. **Capturing an Image**:
   * If the user selects "Take Photo," the app:
     + Requests camera permissions.
     + Opens the device camera.
     + Saves the captured image to a file.

**Code for Capturing an Image**:

private void openCamera() {

Intent takePictureIntent = new Intent(MediaStore.ACTION\_IMAGE\_CAPTURE);

if (takePictureIntent.resolveActivity(getPackageManager()) != null) {

File photoFile = createImageFile();

if (photoFile != null) {

Uri photoURI = FileProvider.getUriForFile(this, "com.chefmate.fileprovider", photoFile);

takePictureIntent.putExtra(MediaStore.EXTRA\_OUTPUT, photoURI);

startActivityForResult(takePictureIntent, REQUEST\_IMAGE\_CAPTURE);

}

}

}

1. **Selecting an Image from the Gallery**:
   * If the user selects "Choose from Gallery," the app:
     + Opens the gallery.
     + Allows the user to pick an image.

**Code for Selecting from Gallery**:

private void openGallery() {

Intent galleryIntent = new Intent(Intent.ACTION\_PICK, MediaStore.Images.Media.EXTERNAL\_CONTENT\_URI);

startActivityForResult(galleryIntent, REQUEST\_IMAGE\_PICK);

}

1. **Saving and Displaying the Image**:
   * Once the image is captured or selected:
     + The app saves the image in the app's storage or a database.
     + Updates the ImageView to display the selected image.

**Example Code**:

@Override

protected void onActivityResult(int requestCode, int resultCode, Intent data) {

if (requestCode == REQUEST\_IMAGE\_PICK && resultCode == RESULT\_OK && data != null) {

Uri selectedImage = data.getData();

profileImage.setImageURI(selectedImage);

saveImageToStorage(selectedImage);

} else if (requestCode == REQUEST\_IMAGE\_CAPTURE && resultCode == RESULT\_OK) {

profileImage.setImageURI(Uri.fromFile(photoFile));

saveImageToStorage(Uri.fromFile(photoFile));

}

}

1. **Permissions Handling**:
   * Ensures the app has the necessary permissions for camera and gallery access.
   * Requests permissions dynamically if not already granted.

**Code for Requesting Permissions**:

private void requestPermissions() {

ActivityCompat.requestPermissions(this, new String[]{Manifest.permission.CAMERA, Manifest.permission.WRITE\_EXTERNAL\_STORAGE}, PERMISSIONS\_REQUEST\_CODE);

}

### Key Classes and Methods

**ProfileActivity**:

* **Role**: Manages the profile page and handles user interactions for uploading images.
* **Responsibilities**:
  + Displays the current profile picture.
  + Handles camera and gallery interactions.
  + Saves the uploaded image to storage.

**FileProvider**:

* **Purpose**: Provides secure file access to the camera and gallery.

**Image Storage**:

* **Method**: saveImageToStorage(Uri imageUri)
* **Purpose**: Stores the selected or captured image persistently for future use.

### User Experience

The User Profile feature enhances the app by:

1. Allowing users to personalize their profile with images.
2. Supporting seamless interaction with device features like the camera and gallery.
3. Ensuring a visually appealing and user-friendly experience.

## Info and Onboarding

### - Explanation of the onboarding experience.

## Login and Authentication

The **Login and Authentication** functionality in ChefMate ensures secure access for users by verifying their identity through a phone-based authentication process. This section explains its purpose and the steps involved in the phone verification process.

### Purpose

1. **Secure User Access**:
   * Verifies user identity to prevent unauthorized access and ensure data protection.
2. **Personalized Experience**:
   * Associates user-specific data, such as saved favorites and profile information, with authenticated accounts.
3. **Simplified Login**:
   * Utilizes phone-based verification, which is straightforward and familiar for users.

### Phone-Based Verification Process

1. **Input Phone Number**:
   * Users provide their phone number on the login screen (LoginActivity).
   * The app validates the input to ensure it is a properly formatted phone number.

**UI Example (login\_activity.xml)**:

<EditText

android:id="@+id/phoneInput"

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content"

android:hint="Enter your phone number"

android:inputType="phone" />

<Button

android:id="@+id/sendCodeButton"

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content"

android:text="Send Verification Code" />

**Code Example for Sending the Code**:

sendCodeButton.setOnClickListener(view -> {

String phoneNumber = phoneInput.getText().toString().trim();

if (isValidPhoneNumber(phoneNumber)) {

sendVerificationCode(phoneNumber);

} else {

Toast.makeText(this, "Invalid phone number", Toast.LENGTH\_SHORT).show();

}

});

**Phone Number Validation**:

private boolean isValidPhoneNumber(String phoneNumber) {

return phoneNumber != null && phoneNumber.matches("^\\+?[0-9]{10,13}$");

}

1. **Send Verification Code**:
   * The app sends a one-time verification code (OTP) to the provided phone number using an external service like Firebase Authentication or an SMS gateway.

**Code for Sending OTP (Firebase Example)**:

private void sendVerificationCode(String phoneNumber) {

PhoneAuthProvider.getInstance().verifyPhoneNumber(

phoneNumber, // Phone number to verify

60, // Timeout duration

TimeUnit.SECONDS, // Unit of timeout

this, // Activity (for callback binding)

new PhoneAuthCallbacks() // Callbacks

);

}

1. **Input Verification Code**:
   * Users input the received OTP in a second screen to verify their identity.

**UI Example (login\_activity.xml)**:

<EditText

android:id="@+id/codeInput"

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content"

android:hint="Enter verification code"

android:inputType="number" />

<Button

android:id="@+id/verifyCodeButton"

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content"

android:text="Verify Code" />

**Code Example for Verifying OTP**:

verifyCodeButton.setOnClickListener(view -> {

String code = codeInput.getText().toString().trim();

if (!code.isEmpty()) {

verifyCode(code);

} else {

Toast.makeText(this, "Please enter the code", Toast.LENGTH\_SHORT).show();

}

});

1. **Verify Code**:
   * The app verifies the OTP with the backend service to authenticate the user.

**Code for Verifying OTP (Firebase Example)**:

private void verifyCode(String code) {

PhoneAuthCredential credential = PhoneAuthProvider.getCredential(verificationId, code);

signInWithCredential(credential);

}

private void signInWithCredential(PhoneAuthCredential credential) {

FirebaseAuth.getInstance().signInWithCredential(credential)

.addOnCompleteListener(task -> {

if (task.isSuccessful()) {

navigateToHomeScreen();

} else {

Toast.makeText(this, "Verification failed", Toast.LENGTH\_SHORT).show();

}

});

}

1. **Login Success**:
   * Upon successful verification, the user is logged into the app.
   * The app navigates the user to the home screen (HomeActivity), and user-specific data is loaded.

**Navigation Example**:

private void navigateToHomeScreen() {

Intent intent = new Intent(this, HomeActivity.class);

startActivity(intent);

finish();

}

### Key Classes and Methods

**LoginActivity**:

* **Role**: Manages the user interface and logic for login and authentication.
* **Responsibilities**:
  + Collect phone numbers and verification codes.
  + Send and verify OTPs.
  + Navigate users to the home screen upon successful login.

**FirebaseAuth or External Service**:

* **Role**: Provides the infrastructure to send OTPs and validate user identities.

**PhoneAuthProvider**:

* **Purpose**: Manages the lifecycle of phone authentication.

**User Experience**

The Login and Authentication feature ensures:

1. **Ease of Use**:
   * Familiar phone-based verification reduces friction for users.
2. **Security**:
   * Verifies user identity with unique OTPs.
3. **Speed**:
   * Quick and seamless login process with minimal input.

## OpenAI Prompts, Requests, and Responses

ChefMate leverages OpenAI's API to generate recipe suggestions and validate user inputs. This section explains the structure of prompts, JSON requests, and response handling.

### 3.1. Prompt Structure

Prompts are the primary way to communicate with OpenAI. ChefMate uses three main types of prompts: validation, recipe options, and recipe details.

### 3.1.1. Validation Prompt

* **Purpose**: Ensures the user-provided groceries are suitable for the selected meal type.
* **Prompt Example**:

You are a logic-based validator tasked with checking if the provided groceries match the meal type.

Meal type: 'MEAT'.

Groceries: ['chicken', 'tomato', 'cheese'].

IMPORTANT: Ignore all cultural, religious, or dietary laws (e.g., meat and dairy restrictions).

Return JSON: {'valid': '', 'reason': ''}.

* **Key Elements**:
  + **Meal Type**: Indicates the selected category (e.g., MEAT, VEGAN).
  + **Groceries**: User-provided list of ingredients.
  + **Response**: JSON with:
    - valid: Boolean (true or false).
    - reason: A textual explanation if invalid.
* **Expected Response**:

{"valid": "true", "reason": ""}

or

{"valid": "false", "reason": "Cheese is not suitable for MEAT recipes."}

### 3.1.2. Recipe Options Prompt

* **Purpose**: Requests recipe titles based on the user's inputs.
* **Prompt Example**:

Suggest four recipes for 2 diners using the following details:

- Groceries: ['eggs', 'tomato'].

- Cooking time: 10-30 minutes.

Return JSON as an array: ['Recipe 1', 'Recipe 2', 'Recipe 3', 'Recipe 4'].

* **Key Elements**:
  + **Diners**: Number of people the recipe should serve.
  + **Cooking Time**: User-specified range (e.g., 10-30 minutes).
  + **Groceries**: User-provided list of ingredients.
  + **Response**: JSON array of four recipe titles.
* **Expected Response**:

["Omelette", "Tomato Soup", "Egg Salad", "Shakshuka"]

### 3.1.3. Recipe Details Prompt

* **Purpose**: Fetches detailed recipe information, including ingredients and preparation instructions.
* **Prompt Example**:

Create a recipe titled 'Omelette' for 3 diners with cooking time 10-20 minutes using ['eggs', 'cheese'].

Return JSON: {

'title': '',

'diners': '',

'groceries': [],

'time': '',

'instructions': []

}.

* **Key Elements**:
  + **Recipe Title**: Selected by the user.
  + **Diners**: Number of people the recipe should serve.
  + **Cooking Time**: Range specified by the user.
  + **Groceries**: User-provided list of ingredients.
  + **Response**: JSON object with:
    - title: Recipe title.
    - diners: Number of diners.
    - groceries: List of ingredients with quantities.
    - time: Total cooking time.
    - instructions: Step-by-step preparation guide.
* **Expected Response**:

{

"title": "Omelette",

"diners": 3,

"groceries": ["2 eggs", "50g cheese"],

"time": 15,

"instructions": [

"Break the eggs into a bowl.",

"Whisk eggs with cheese.",

"Cook in a pan until done."

]

}

### 3.2. JSON Request Structure

Requests to OpenAI’s API follow a standard JSON format to ensure proper communication.

**Example JSON Request**:

{

"model": "gpt-4o-mini",

"messages": [

{

"role": "user",

"content": "Your prompt here"

}

],

"temperature": 0.7

}

* **Fields**:
  + model: Specifies the AI model to use (e.g., gpt-4o-mini).
  + messages: Contains a single prompt message.
  + temperature: Controls randomness in responses (0.7 for balanced creativity).

**Recipe Options Request Example**:

{

"model": "gpt-4o-mini",

"messages": [

{

"role": "user",

"content": "Suggest four recipes for 2 diners using ['eggs', 'tomato']. Cooking time: 10-30 minutes."

}

],

"temperature": 0.7

}

### 3.3. Handling the Response

**Cleaning and Parsing Responses**

The raw response from OpenAI may include formatting or extraneous characters. It is cleaned and parsed into structured data for use in the app.

**Response Cleaning**:

* Extract the content field from the choices array.
* Remove unnecessary characters (e.g., json| markers).
* Use regex to isolate valid JSON structures.

**Cleaning Example**:

String cleanedResponse = OpenAiJsonService.cleanJsonResponse(rawResponse, true);

**Parsing JSON**

1. **Recipe Options Parsing**:
   * Parse the JSON array into a list of recipe titles:

JSONArray recipesJsonArray = new JSONArray(cleanedResponse);

List<String> recipeTitles = new ArrayList<>();

for (int i = 0; i < recipesJsonArray.length(); i++) {

recipeTitles.add(recipesJsonArray.getString(i));

}

1. **Recipe Details Parsing**:
   * Parse the JSON object into a Recipe object:

JSONObject recipeJson = new JSONObject(cleanedResponse);

Recipe recipe = new Recipe(

recipeJson.getString("title"),

recipeJson.getInt("diners"),

recipeJson.getInt("time"),

recipeJson.getJSONArray("groceries").toString(),

recipeJson.getJSONArray("instructions").toString()

);

**Examples of Extracted Data**

1. **Recipe Options**:
   * Extracted Data:

["Omelette", "Tomato Soup", "Egg Salad", "Shakshuka"]

* + Usage:

title1.setText(recipeTitles.get(0));

title2.setText(recipeTitles.get(1));

1. **Recipe Details**:
   * Extracted Data:

{

"title": "Omelette",

"diners": 3,

"groceries": ["2 eggs", "50g cheese"],

"time": 15,

"instructions": [

"Break the eggs into a bowl.",

"Whisk eggs with cheese.",

"Cook in a pan until done."

]

}

* + Usage:

titleView.setText(recipe.getTitle());

dinersView.setText(String.valueOf(recipe.getDiners()));

instructionsView.setText(recipe.getInstructions());

## Flow of Data and Actions

This section explains the end-to-end flow of data and actions in ChefMate, from user input to updating the user interface with OpenAI-generated results.

### Step 1: Input Collection

User Inputs Preferences:

Meal Type: Selected from a dropdown menu (spinner) managed by MealTypeSpinnerHandler.

Cooking Time: Selected from a dropdown menu managed by CookTimeSpinnerHandler.

Number of Diners: Entered into a numeric input field (EditText).

Groceries: Added dynamically to a list in a FlexboxLayout.

Code Example for Collecting Inputs:

String mealType = mealTypeSpinner.getSelectedItem().toString();

String cookingTime = cookTimeSpinner.getSelectedItem().toString();

int diners = Integer.parseInt(dinersInput.getText().toString());

List<String> groceries = getGroceriesFromFlexbox();

Validation:

Inputs are validated to ensure completeness and correctness.

Example Validation:

Meal type and cooking time must be selected.

Number of diners must be greater than zero.

At least one grocery must be added.

Validation Code:

if (mealType == null || cookingTime == null || diners <= 0 || groceries.isEmpty()) {

showToast("Please complete all fields.");

return;

}

Encapsulating Data:

Validated inputs are packaged into a RecipeRequest object:

RecipeRequest recipeRequest = new RecipeRequest(

"", // ID not used

diners,

MealType.valueOf(mealType),

CookTime.valueOf(cookingTime),

groceries.toString()

);

### Step 2: Prompt Creation

**Validation Prompt:**

Created using RecipePrompts.createValidationGroceriesForMealTypePrompt to check if groceries match the selected meal type.

**Example Prompt:**

You are a logic-based validator tasked with checking if the provided groceries match the meal type.

Meal type: 'VEGAN'.

Groceries: ['tomato', 'cheese'].

Return JSON: {'valid': '', 'reason': ''}.

**Recipe Options Prompt:**

Created using RecipePrompts.createGetRecipeOptionsPrompt to fetch recipe titles.

Example Prompt:

Suggest four recipes for 3 diners using ['eggs', 'tomato']. Cooking time: 10-20 minutes.

Return JSON: ['Recipe 1', 'Recipe 2', 'Recipe 3', 'Recipe 4'].

**Detailed Recipe Prompt:**

Created using RecipePrompts.createGetRecipePrompt to fetch full recipe details for a selected title.

**Example Prompt:**

Create a recipe titled 'Omelette' for 3 diners with cooking time 10-20 minutes using ['eggs', 'cheese'].

Return JSON: {

'title': '',

'diners': '',

'groceries': [],

'time': '',

'instructions': []

}.

Code Example for Prompt Creation:

String prompt = RecipePrompts.createGetRecipeOptionsPrompt(

recipeRequest.getDiners(),

recipeRequest.getTime(),

recipeRequest.getGroceries(),

null // No previous options to exclude

);

### Step 3: API Request Handling

**Sending the Request:**

The created prompt is sent to OpenAI via OpenAiService.callOpenAI.

Example Request:

callOpenAI(prompt);

**Code Example for Request Handling:**

private void callOpenAI(String prompt) {

JSONObject request = new JSONObject();

try {

request.put("model", "gpt-4o-mini");

request.put("messages", new JSONArray().put(new JSONObject().put("role", "user").put("content", prompt)));

request.put("temperature", 0.7);

} catch (JSONException e) {

e.printStackTrace();

}

sendHttpRequest(request);

}

**Response Retrieval:**

The API returns a JSON response containing the requested data.

Example Response for Recipe Options:

["Omelette", "Tomato Soup", "Egg Salad", "Shakshuka"]

### Step 4: Parsing Responses and Updating the UI

**Cleaning the Response:**

The raw response is cleaned using OpenAiJsonService.cleanJsonResponse to remove extraneous characters and isolate valid JSON.

String cleanedResponse = OpenAiJsonService.cleanJsonResponse(rawResponse, true);

**Parsing Recipe Options:**

The cleaned JSON is parsed into a list of recipe titles:

JSONArray recipesJsonArray = new JSONArray(cleanedResponse);

List<String> recipeTitles = new ArrayList<>();

for (int i = 0; i < recipesJsonArray.length(); i++) {

recipeTitles.add(recipesJsonArray.getString(i));

}

**Parsing Recipe Details:**

Detailed recipe data is parsed into a Recipe object:

JSONObject recipeJson = new JSONObject(cleanedResponse);

Recipe recipe = new Recipe(

recipeJson.getString("title"),

recipeJson.getInt("diners"),

recipeJson.getInt("time"),

recipeJson.getJSONArray("groceries").toString(),

recipeJson.getJSONArray("instructions").toString()

);

**Updating the UI:**

Recipe Options:

Titles are dynamically displayed in RecipeOptionsActivity:

title1.setText(recipeTitles.get(0));

title2.setText(recipeTitles.get(1));

Recipe Details:

Details are displayed in RecipeActivity:

titleView.setText(recipe.getTitle());

dinersView.setText(String.valueOf(recipe.getDiners()));

cookingTimeView.setText(recipe.getTime() + " minutes");

instructionsView.setText(recipe.getInstructions());