Smart Kitchen Helper: SQL and NoSQL Database Design

Amit Christian

2024-08-04

# Introduction

This document provides a comprehensive overview of the SQL and NoSQL database design for the Smart Kitchen Helper project. It includes detailed schema diagrams, table definitions, MongoDB collections, and a thorough explanation of the constraints, business rules, and potential joins that govern the system. The goal is to ensure that the system is scalable, maintainable, and capable of efficiently managing the data required for the application’s features.

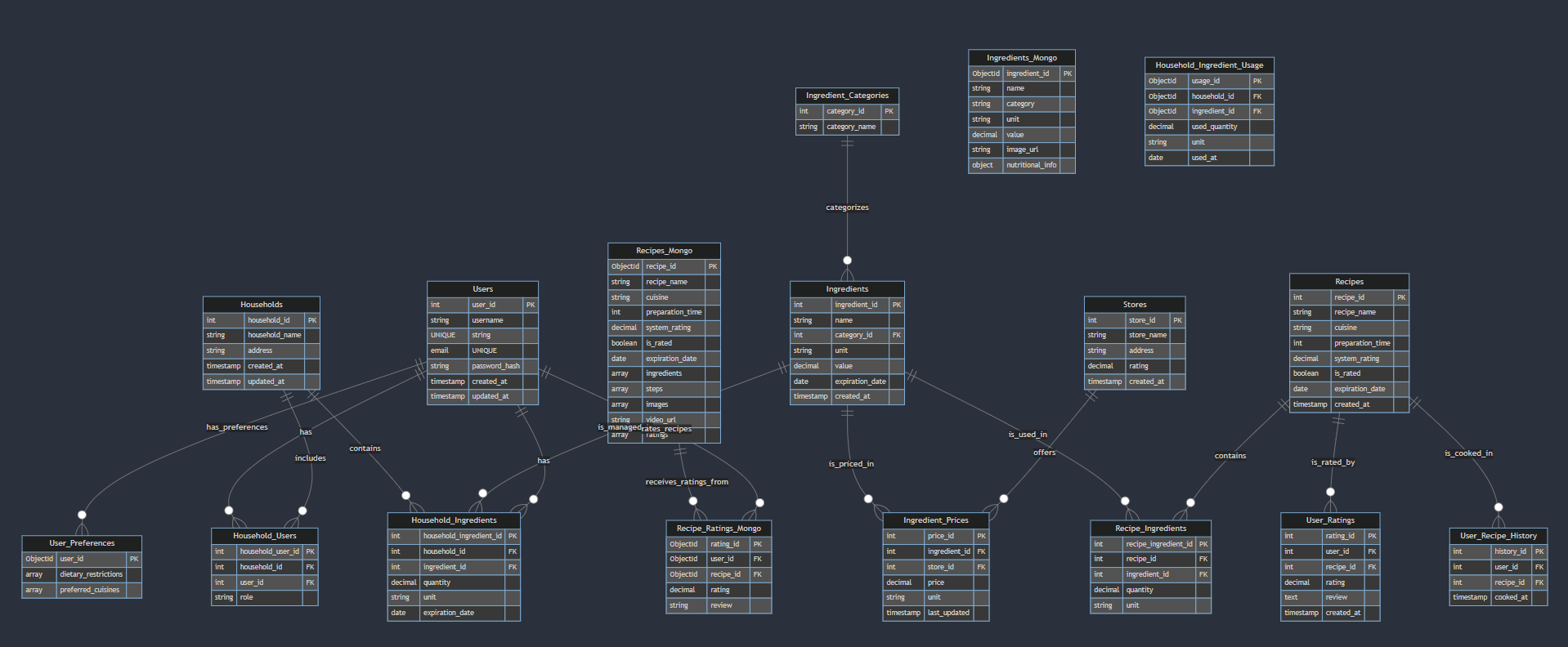
**Author:** Amit Christian  
**Course:** Cloud Computing, Durham College  
**Student ID:** 100937637

# SQL Database Design

## Conceptual Design

The SQL database is designed to handle structured data, such as user profiles, households, ingredients, recipes, and additional entities like stores, prices, and ingredient categories. Constraints and rules are implemented to enforce data integrity and business logic.

### ER Diagram



### Entities and Relationships

* **Users:** Manages user profiles, including personal details and authentication information.
* **Households:** Represents a group of users, typically members of a family or shared living space, that manages a collective inventory of ingredients.
* **Ingredients:** Contains the list of ingredients available in each household, categorized by type (e.g., veggie, dairy, meat) along with quantities, units, and expiration dates.
* **Recipes:** Stores basic information about recipes, including names, cuisines, preparation times, system ratings, and whether the recipe is rated by users.
* **Stores:** Manages information about stores where ingredients can be purchased, including address and ratings.
* **Ingredient\_Prices:** Tracks the price of ingredients at various stores, including units and last updated timestamps.
* **Recipe\_Ingredients:** A join table that connects recipes with their required ingredients.
* **Household\_Ingredients:** Manages the inventory of ingredients within each household, including quantities, units, and expiration dates.
* **User\_Recipe\_History:** Logs the history of recipes cooked by users, allowing them to track their culinary activities over time.
* **User\_Ratings:** Collects individual user ratings for recipes, including a text review.

## SQL Table Schemas with Constraints, Rules, and Logical Details

Below are the detailed table schemas for the SQL database, including constraints, business rules, and explanations of logical joins and null handling.

### 1. Users Table

CREATE TABLE Users (  
 user\_id INT PRIMARY KEY AUTO\_INCREMENT,  
 username VARCHAR(50) UNIQUE NOT NULL,  
 email VARCHAR(100) UNIQUE NOT NULL CHECK (email LIKE '%\_@\_\_%.\_\_%'),  
 password\_hash VARCHAR(255) NOT NULL,  
 created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,  
 updated\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP  
);

**Constraints and Rules:** - **Unique Constraints:** username and email must be unique. - **Email Format Validation:** Ensures that email addresses follow a valid format. - **Not Null:** Ensures that all required fields are filled.

### 2. Households Table

CREATE TABLE Households (  
 household\_id INT PRIMARY KEY AUTO\_INCREMENT,  
 household\_name VARCHAR(100) NOT NULL,  
 address VARCHAR(255),  
 created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,  
 updated\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP  
);

**Constraints and Rules:** - **Not Null:** The household\_name must be provided. - **Timestamp Defaults:** Automatically track when households are created and updated.

### 3. Household\_Users Table

CREATE TABLE Household\_Users (  
 household\_user\_id INT PRIMARY KEY AUTO\_INCREMENT,  
 household\_id INT,  
 user\_id INT,  
 role ENUM('owner', 'member') DEFAULT 'member',  
 FOREIGN KEY (household\_id) REFERENCES Households(household\_id) ON DELETE CASCADE,  
 FOREIGN KEY (user\_id) REFERENCES Users(user\_id) ON DELETE CASCADE  
);

**Constraints and Rules:** - **Foreign Keys:** Links users to households, with cascading deletes to maintain referential integrity. - **Role Constraint:** Limits the role to either ‘owner’ or ‘member’.

### 4. Ingredient\_Categories Table

CREATE TABLE Ingredient\_Categories (  
 category\_id INT PRIMARY KEY AUTO\_INCREMENT,  
 category\_name VARCHAR(50) UNIQUE NOT NULL  
);

**Constraints and Rules:** - **Unique Constraint:** Each category name must be unique. - **Not Null:** Category names must be provided.

### 5. Ingredients Table

CREATE TABLE Ingredients (  
 ingredient\_id INT PRIMARY KEY AUTO\_INCREMENT,  
 name VARCHAR(100) NOT NULL,  
 category\_id INT,  
 unit VARCHAR(50) NOT NULL,  
 value DECIMAL(10, 2) NOT NULL CHECK (value >= 0),  
 expiration\_date DATE CHECK (expiration\_date > CURRENT\_DATE),  
 created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,  
 FOREIGN KEY (category\_id) REFERENCES Ingredient\_Categories(category\_id) ON DELETE SET NULL  
);

**Constraints and Rules:** - **Foreign Key:** Links ingredients to their category, with ON DELETE SET NULL to maintain data even if a category is deleted. - **Not Null:** Ingredient names, units, and values must be provided. - **Value Validation:** Ensures that ingredient values (e.g., quantity, weight) are non-negative. - **Expiration Date Validation:** Ensures that expiration dates are in the future.

### 6. Stores Table

CREATE TABLE Stores (  
 store\_id INT PRIMARY KEY AUTO\_INCREMENT,  
 store\_name VARCHAR(100) NOT NULL,  
 address VARCHAR(255) NOT NULL,  
 rating DECIMAL(2, 1) CHECK (rating >= 0 AND rating <= 5),  
 created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP  
);

**Constraints and Rules:** - **Rating Validation:** Ensures that store ratings are within the 0 to 5 range. - **Not Null:** Store names and addresses must be provided.

### 7. Ingredient\_Prices Table

CREATE TABLE Ingredient\_Prices (  
 price\_id INT PRIMARY KEY AUTO\_INCREMENT,  
 ingredient\_id INT,  
 store\_id INT,  
 price DECIMAL(10, 2) NOT NULL CHECK (price >= 0),  
 unit VARCHAR(50) NOT NULL,  
 last\_updated TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,  
 FOREIGN KEY (ingredient\_id) REFERENCES Ingredients(ingredient\_id) ON DELETE CASCADE,  
 FOREIGN KEY (store\_id) REFERENCES Stores(store\_id) ON DELETE CASCADE  
);

**Constraints and Rules:** - **Price Validation:** Ensures that prices are non-negative. - **Foreign Keys:** Links ingredient prices to specific stores and ingredients, with cascading deletes for data integrity. - **Null Handling:** If a price is unavailable, the record is simply not added, avoiding the need for nulls.

### 8. Household\_Ingredients Table

CREATE TABLE Household\_Ingredients (  
 household\_ingredient\_id INT PRIMARY KEY AUTO\_INCREMENT,  
 household\_id INT,  
 ingredient\_id INT,  
 quantity DECIMAL(10, 2) NOT NULL CHECK (quantity >= 0),  
 unit VARCHAR(50) NOT NULL,  
 expiration\_date DATE CHECK (expiration\_date > CURRENT\_DATE),  
 FOREIGN KEY (household\_id) REFERENCES Households(household\_id) ON DELETE CASCADE,  
 FOREIGN KEY (ingredient\_id) REFERENCES Ingredients(ingredient\_id) ON DELETE CASCADE  
);

**Constraints and Rules:** - **Quantity Validation:** Ensures that ingredient quantities are non-negative. - **Expiration Date Validation:** Ensures that expiration dates are in the future. - **Foreign Keys:** Links household ingredients to specific households and ingredients, maintaining referential integrity.

### 9. Recipes Table

CREATE TABLE Recipes (  
 recipe\_id INT PRIMARY KEY AUTO\_INCREMENT,  
 recipe\_name VARCHAR(100) NOT NULL,  
 cuisine VARCHAR(50),  
 preparation\_time INT CHECK (preparation\_time >= 0),  
 system\_rating DECIMAL(2, 1) CHECK (system\_rating >= 0 AND system\_rating <= 5),  
 is\_rated BOOLEAN DEFAULT FALSE,  
 expiration\_date DATE CHECK (expiration\_date > CURRENT\_DATE),  
 created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP  
);

**Constraints and Rules:** - **Preparation Time Validation:** Ensures that preparation times are non-negative. - **System Rating Validation:** System ratings must be between 0 and 5. - **Expiration Date Validation:** Ensures that the recipe’s expiration date is in the future (e.g., for ingredients that might expire). - **Not Null:** Recipe names must be provided. - **Boolean Field:** The is\_rated field indicates whether the recipe has been rated by users.

### 10. Recipe\_Ingredients Table

CREATE TABLE Recipe\_Ingredients (  
 recipe\_ingredient\_id INT PRIMARY KEY AUTO\_INCREMENT,  
 recipe\_id INT,  
 ingredient\_id INT,  
 quantity DECIMAL(10, 2) NOT NULL CHECK (quantity >= 0),  
 unit VARCHAR(50) NOT NULL,  
 FOREIGN KEY (recipe\_id) REFERENCES Recipes(recipe\_id) ON DELETE CASCADE,  
 FOREIGN KEY (ingredient\_id) REFERENCES Ingredients(ingredient\_id) ON DELETE CASCADE  
);

**Constraints and Rules:** - **Quantity Validation:** Ensures that ingredient quantities in recipes are non

-negative. - **Foreign Keys:** Links recipe ingredients to specific recipes and ingredients, ensuring data consistency.

### 11. User\_Recipe\_History Table

CREATE TABLE User\_Recipe\_History (  
 history\_id INT PRIMARY KEY AUTO\_INCREMENT,  
 user\_id INT,  
 recipe\_id INT,  
 cooked\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,  
 FOREIGN KEY (user\_id) REFERENCES Users(user\_id) ON DELETE CASCADE,  
 FOREIGN KEY (recipe\_id) REFERENCES Recipes(recipe\_id) ON DELETE CASCADE  
);

**Constraints and Rules:** - **Foreign Keys:** Links user cooking history to specific users and recipes, maintaining referential integrity.

### 12. User\_Ratings Table

CREATE TABLE User\_Ratings (  
 rating\_id INT PRIMARY KEY AUTO\_INCREMENT,  
 user\_id INT,  
 recipe\_id INT,  
 rating DECIMAL(2, 1) CHECK (rating >= 0 AND rating <= 5),  
 review TEXT,  
 created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,  
 FOREIGN KEY (user\_id) REFERENCES Users(user\_id) ON DELETE CASCADE,  
 FOREIGN KEY (recipe\_id) REFERENCES Recipes(recipe\_id) ON DELETE CASCADE  
);

**Constraints and Rules:** - **Rating Validation:** Ensures that user ratings are between 0 and 5. - **Foreign Keys:** Links ratings to specific users and recipes, maintaining data integrity. - **Null Handling:** If a user does not leave a review, the review field can be null.

## Logical Joins and Null Handling

### 1. Viewing Household Ingredients

SELECT   
 h.household\_name,   
 i.name AS ingredient\_name,   
 hi.quantity,   
 hi.unit,   
 hi.expiration\_date  
FROM   
 Households h  
JOIN   
 Household\_Ingredients hi ON h.household\_id = hi.household\_id  
JOIN   
 Ingredients i ON hi.ingredient\_id = i.ingredient\_id  
WHERE   
 h.household\_id = ?;

**Explanation:**  
This query retrieves all ingredients in a specific household, including their quantities, units, and expiration dates. Null handling ensures that if an ingredient’s category or other optional data is not available, the query still returns valid results.

### 2. Fetching Store Prices for Ingredients

SELECT   
 s.store\_name,   
 i.name AS ingredient\_name,   
 ip.price,   
 ip.unit,   
 ip.last\_updated  
FROM   
 Stores s  
JOIN   
 Ingredient\_Prices ip ON s.store\_id = ip.store\_id  
JOIN   
 Ingredients i ON ip.ingredient\_id = i.ingredient\_id  
WHERE   
 i.ingredient\_id = ?;

**Explanation:**  
This query retrieves the prices of a specific ingredient across all stores. Null handling ensures that if a price is not available, the record is excluded from the result, avoiding the need for null values.

### 3. Total Ingredients in a Household

SELECT   
 COUNT(hi.ingredient\_id) AS total\_ingredients  
FROM   
 Household\_Ingredients hi  
WHERE   
 hi.household\_id = ?;

**Explanation:**  
This query returns the total number of different ingredients in a household. It handles nulls by excluding any non-existent ingredients from the count.

### 4. Average User Rating for a Recipe

SELECT   
 r.recipe\_name,   
 AVG(ur.rating) AS average\_rating  
FROM   
 Recipes r  
JOIN   
 User\_Ratings ur ON r.recipe\_id = ur.recipe\_id  
WHERE   
 r.recipe\_id = ?;

**Explanation:**  
This query calculates the average user rating for a specific recipe. It excludes null ratings, ensuring that only valid ratings contribute to the average.

# MongoDB (NoSQL) Database Design

## Conceptual Design

The NoSQL database, implemented using MongoDB, is used to store unstructured or semi-structured data such as detailed recipe instructions, ingredient images, and media URLs. MongoDB’s flexible schema allows for efficient management of this data type.

### Collections Overview

1. **recipes:** Stores detailed information about recipes, including preparation steps, images, and YouTube video URLs.
2. **ingredients:** Manages images and detailed information for ingredients.
3. **household\_ingredient\_usage:** Tracks the usage of ingredients within households, including historical usage data.
4. **recipe\_ratings:** Collects user ratings and reviews for recipes, allowing for community feedback and recipe improvement.
5. **user\_preferences:** Stores user-specific preferences and settings, such as dietary restrictions and preferred cuisines.

## MongoDB Collections with Constraints, Rules, and Logical Details

### 1. Recipes Collection

{  
 "recipe\_id": ObjectId("60c72b2f9b1e8a4d5f8e4d67"),  
 "recipe\_name": "Butter Chicken",  
 "cuisine": "Indian",  
 "preparation\_time": 45,  
 "system\_rating": 4.5,  
 "is\_rated": true,  
 "expiration\_date": ISODate("2023-12-31T00:00:00Z"),  
 "ingredients": [  
 {  
 "ingredient\_id": ObjectId("60c72b2f9b1e8a4d5f8e4d68"),  
 "name": "Chicken",  
 "quantity": 500,  
 "unit": "grams"  
 },  
 {  
 "ingredient\_id": ObjectId("60c72b2f9b1e8a4d5f8e4d69"),  
 "name": "Butter",  
 "quantity": 100,  
 "unit": "grams"  
 }  
 ],  
 "steps": [  
 "Marinate the chicken with spices and yogurt.",  
 "Cook the marinated chicken in butter until done.",  
 "Prepare the sauce and simmer with chicken."  
 ],  
 "images": [  
 "https://example.com/butter\_chicken\_step1.jpg",  
 "https://example.com/butter\_chicken\_step2.jpg"  
 ],  
 "video\_url": "https://youtube.com/example\_butter\_chicken",  
 "ratings": [  
 {  
 "user\_id": ObjectId("60c72b2f9b1e8a4d5f8e4d61"),  
 "rating": 4.5,  
 "review": "Delicious and easy to make!"  
 }  
 ]  
}

**Constraints and Rules:** - **Schema Flexibility:** MongoDB allows for dynamic schemas, so documents can vary within the same collection. - **Validation:** MongoDB can enforce JSON Schema validation for required fields and data types. - **Logical Expiration:** Recipes include an expiration date, which could be used to ensure ingredients are still valid or the recipe is still relevant.

### 2. Ingredients Collection

{  
 "ingredient\_id": ObjectId("60c72b2f9b1e8a4d5f8e4d68"),  
 "name": "Chicken",  
 "category": "Meat",  
 "unit": "grams",  
 "value": 500,  
 "image\_url": "https://example.com/chicken.jpg",  
 "nutritional\_info": {  
 "calories": 200,  
 "protein": "20g",  
 "fat": "10g",  
 "carbohydrates": "0g"  
 }  
}

**Constraints and Rules:** - **Category Validation:** Ingredients are categorized, which helps in filtering and querying. - **Unit and Value:** Each ingredient has a unit and a value, helping in accurate inventory management. - **Optional Fields:** MongoDB allows fields like nutritional\_info to be optional, providing flexibility.

### 3. Household Ingredient Usage Collection

{  
 "usage\_id": ObjectId("60c72b2f9b1e8a4d5f8e4d70"),  
 "household\_id": ObjectId("60c72b2f9b1e8a4d5f8e4d60"),  
 "ingredient\_id": ObjectId("60c72b2f9b1e8a4d5f8e4d68"),  
 "used\_quantity": 250,  
 "unit": "grams",  
 "used\_at": ISODate("2023-08-04T14:00:00Z")  
}

**Constraints and Rules:** - **Usage Tracking:** Tracks the quantity of ingredients used in households, allowing for detailed inventory management.

### 4. Recipe Ratings Collection

{  
 "rating\_id": ObjectId("60c72b2f9b1e8a4d5f8e4d71"),  
 "user\_id": ObjectId("60c72b2f9b1e8a4d5f8e4d61"),  
 "recipe\_id": ObjectId("60c72b2f9b1e8a4d5f8e4d67"),  
 "rating": 4.5,  
 "review": "This recipe was fantastic! The butter chicken turned out perfect."  
}

**Constraints and Rules:** - **Rating Validation:** Ratings should be within a 0 to 5 range, which can be enforced through application logic or validation.

### 5. User Preferences Collection

{  
 "user\_id": ObjectId("60c72b2f9b1e8a4d5f8e4d61"),  
 "dietary\_restrictions": ["Vegetarian", "Gluten-Free"],  
 "preferred\_cuisines": ["Indian", "Italian"]  
}

**Constraints and Rules:** - **Schema Flexibility:** Allows for varying preferences and settings per user. - **Array Fields:** Supports lists of preferences,

providing flexibility in how user data is stored and accessed.

# Summary

This document outlines the detailed design of the SQL and MongoDB databases for the Smart Kitchen Helper project, including constraints, business rules, and potential joins. The SQL database handles structured data essential for managing users, households, ingredients, recipes, and stores, while the MongoDB database efficiently manages unstructured data, such as images, video URLs, and detailed recipe steps.

Together, these databases provide a robust and flexible backend that supports the application’s diverse data requirements, ensuring scalability, maintainability, and a seamless user experience. The constraints, rules, and logical joins outlined ensure data integrity and enforce the necessary business logic throughout the system. ```

### Key Enhancements:

* **Ingredient Units and Values:** The SQL tables now include unit and value fields for ingredients, ensuring precise tracking of quantities and units across households, recipes, and stores.
* **Recipe Expiration:** Recipes now include an expiration date, allowing the system to validate whether the recipe is still valid based on ingredient availability and freshness.
* **System Ratings and User Ratings:** Recipes now have a system\_rating field and a boolean is\_rated field, alongside a separate User\_Ratings table, allowing for both system-generated and user-generated ratings.
* **Logical Joins:** Detailed examples of logical joins show how to retrieve data across related tables, handle nulls properly, and ensure that the system can provide accurate and comprehensive reports.
* **NoSQL Collections:** MongoDB collections include constraints and rules similar to SQL, but with the flexibility inherent in NoSQL databases, allowing for dynamic schema changes and efficient data storage.

This detailed design document provides a solid foundation for implementing the Smart Kitchen Helper project, ensuring that all data is well-organized, accessible, and logically managed. The system is designed to handle current requirements and future scalability needs, providing a robust backend for the application’s features.