Victor Fatunse (2313618)

Recipe Database

Learning Project

Databases

2025

Kaakkois-Suomen
ammattikorkeakoulu

Table of contents

INTRODUCTION

In this project, I am going to be making a recipe management system that will have command line application that interacts with a database.

The instructions and steps for this project are highlighted in the project link. The updated version is also commited to the repository to reflect the changes in the project.

Most parts of the project require setting up a database, using the right relationships, junction tables and making sure that they all interoperate properly with each other. After setting that up, the database will be scaffolded into the project and the commandline app can interact with the database to provide recipes to the users.

The learning objective of this project is to understand more about databases and how to integrate them in day-to-day software / tools.

Link to classroom repository: https://github.com/xamk-mire-classroom/database-project-2025-vicking20

High level description

Schema Overview

To begin with I created the database manually in my db manager, the name of the database is recipe_db. Inside the database, I used the commands from the project located in the create.sql folder to make the tables into the database.

The purpose of this database schema is to manage and organize recipe-related data in a structured database. We can effectively store detailed data about:

- Recipes, preparation time, number of servings

- Ingredients and the quantity to use in each recipe

- Measurement units to standardize ingredient tracking

- Diet categories (eg. Dessert, vegetarian)

- Cooking instructions

- Relationships between recipes and ingredients, between recipes and categories.

Entity	Description
Recipe	Stores general information about the recipe
Ingredient	Stores individual ingredients
Category	Instructions for preparing the recipe
Unit	Measurement units (e.g grams, cups)
Step	Instructions for preparing the recipe

Relationships

A recipe can have many ingredients, and each ingredient can be used in many recipes (Recipes -> Many-to-many ->)

A recipe has many steps (Steps -> One-to-many)

Core Tables:

Recipe	RecipeID, Name, Description, PrepTime, CookTime, Servings
Ingredient	IngredientID, Name
Category	CategoryID, Name
Unit	UnitID, Name
Step	StepID, RecipeID, StepNumber, Instruction
RecipeIngredient	RecipeID, IngredientID, Quantity, UnitID
RecipeCategory	RecipeID, CategoryID

Normalization Strategies

1NF: All columns hold atomic values

2Nf: Non-key columns are fully functionally dependent on the primary key

3NF: No transitive dependencies

Constraints to apply

Primary Key (PK) -> Available on all base tables

Foreign Key (FK) -> Available for example on Recipe Ingredients, Recipe Category, Step

Unique -> Available on fields like Category.Name, Ingredient.Name

Not Null -> On essential columns (Recipe.Name, Step.Instruction)

Diagram

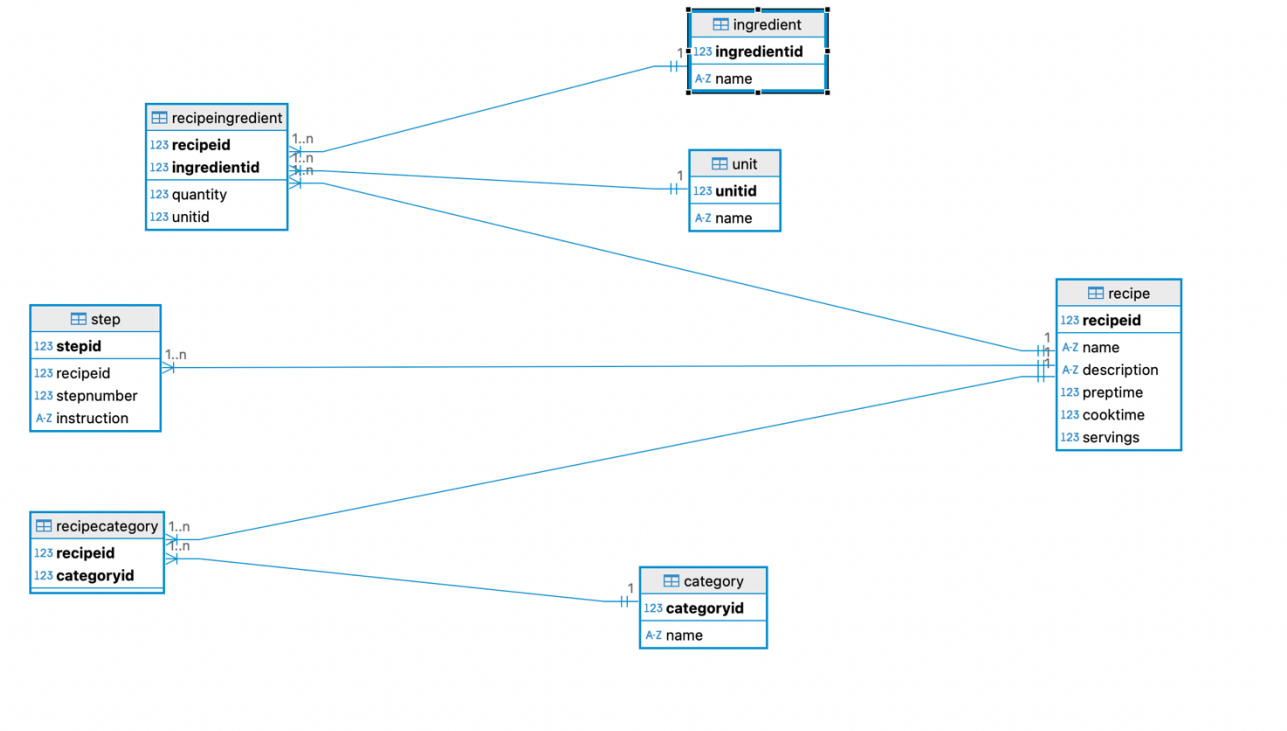

Figure 1: ER diagram of recipe_db

Entities and relationships

List of entities

1. Recipe : RecipeID(PK), Name, Description, PrepTime, CookTime, Servings

2. Ingredient : IngredientID(PK), Name

3. Unit : UnitID(PK), Name

4. Step : StepID(PK), RecipeID(FK), StepNumber, Instruction

5. Category : CategoryID(PK), Name

6. RecipeIngredient(junction table) : RecipeID(FK, PK), IngredientID(FK,PK), Quantity, UnitID(FK)

7. RecipeCategory(junction table) : RecipeID(FK, PK), CategoryID(FK, PK)

Relationship Description

Recipe - Step (One-to-many(1:N)): A single recipe can have many steps. Step references RecipeID as a foreign key.

Recipe - Ingredient (Many-to-Many(M:N)): A recipe uses many ingredients, and an ingredient can be used in many recipes. This is modeled using the RecipeIngredient table.

Recipe - category (Many-to-Many(M:N)): A recipe can belong to multiple categories (e.g vegan, dinner), and each category includes multiple recipes. This is handled with the RecipeCategory junction table.

Unit - RecipeIngredient (One-to-many(1:N)): A unit (e.g grams, teaspoons) can be used by many recipe-ingredient entries.

Normalization and constraints

Normalization levels

This schema is designed to satisfy third normal form 3NF where there are no transitive dependencies; all non-key attributes are only dependent on the key. This ensures data integrity, minimizes redundancy and improves scalability.

Constraints

Primary Keys: Uniquely identify records in each table.

Foreign Keys: Maintain referential integrity between related tables.

NOT NULL: Enforced on required fields like Recipe.Name or Step.Instruction.

UNIQUE: Applied where necessary to prevent duplicates (e.g., Ingredient.Name and Category.Name).

Composite Primary Keys: Used in RecipeIngredient and RecipeCategory to avoid duplicate entries.

Design choices and rationale

Reasonings

I used junction tables (RecipeIngredient, RecipeCategory) to handle many-to-many relationships.

A separate Unit table was included to normalize units instead of storing them as plain text (e.g., “grams”, “ml”)—this improves consistency and reduces typos.

Steps are stored in a separate table with a StepNumber to maintain order and allow editing of individual instructions.

Alternatives considered

I considered storing ingredients and quantities directly in the Recipe table, but this would violate normalization and make querying more complex.

Another alternative was embedding categories as a comma-separated list in Recipe, but this violates 1NF and makes filtering by category inefficient.

Database Implementation

Sample Data

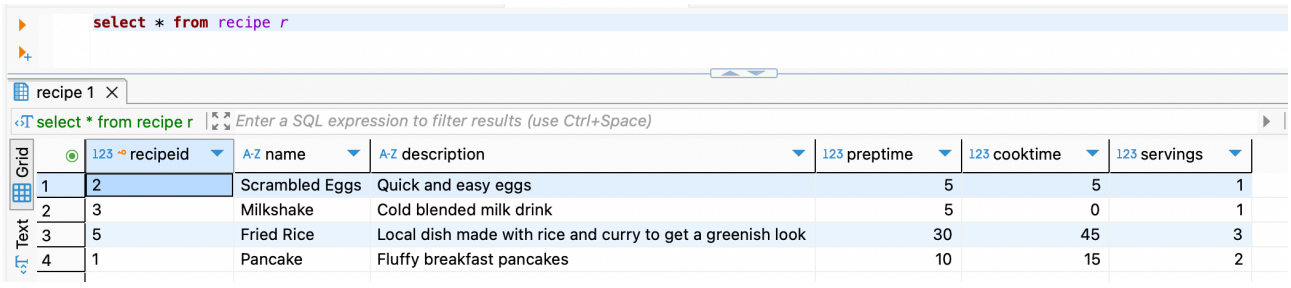

Figure 2: Sample recipe data there are 4 recipes inputted into the database

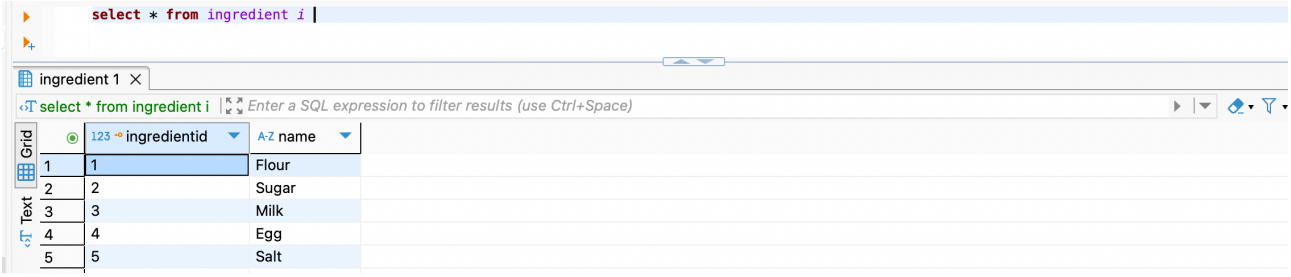

Figure 3: Sample data for ingredients. We have 5 ingredients inputted into the table

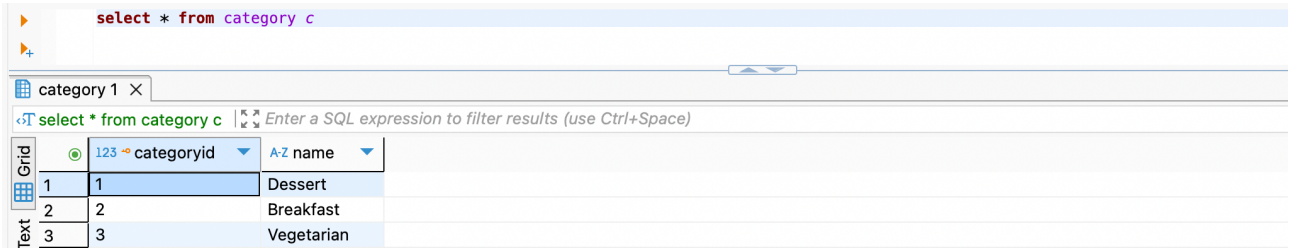

Figure 4: Sample data for categories

Key fields and data types

Table	Important Fields	Notes
Recipe	RecipeID(PK, serial), Name(Text)	Core entity that stores metadata about each recipe
Ingredient	IngredientID(PK), Name(Text)	Ingredeient names are unique and can be used in many recipes
Unit	UnitID(PK), Name(Text)	Defines measurement units (e.g grams)
Step	StepID(PK), RecipeID(FK), Instruction	Ordered steps by recipe using StepNumber for definition
Category	CategoryID(PK), Name(Text)	Recipe categories
RecipeIngredient	Composes of RecipeID + IngredientID	Junction table storing Quantity and UnitID
RecipeCategory	Composes of RecipeID + CategoryID	Junction table to implement many to many models for recipes and categories

Constraints used

'Primary' and 'foreign' keys used to ensure data integrity

'Not Null' used for constraints on essential fields (e.g names, step instructions)

'Unique' constraints on Ingredient.Name and Category.Name

Composite 'Primary' keys in RecipeIngredient and RecipeCategory prevent duplicates

Insert Statements

SQL-queries > Insert.sql

```sql
-- Insert Units
INSERT INTO Unit (Name) VALUES ('grams'), ('ml'), ('tbsp');

-- Insert Ingredients
INSERT INTO Ingredient (Name) VALUES
('Flour'),
('Sugar'),
('Milk'),
('Egg'),
('Salt');

-- Insert Categories
INSERT INTO Category (Name) VALUES
('Dessert'),
('Breakfast'),
('Vegetarian');

-- Insert Recipes
INSERT INTO Recipe (Name, Description, PrepTime, CookTime, Servings) VALUES
('Pancakes', 'Fluffy breakfast pancakes', 10, 15, 2),
('Scrambled Eggs', 'Quick and easy eggs', 5, 5, 1),
('Milkshake', 'Cold blended milk drink', 5, 0, 1);

-- Insert Steps
-- Pancakes
INSERT INTO Step (RecipeID, StepNumber, Instruction) VALUES
(1, 1, 'Mix flour, sugar, and salt in a bowl.'),
(1, 2, 'Add milk and egg, then whisk until smooth.'),
(1, 3, 'Heat a pan and pour in batter. Cook until golden.');

-- Scrambled Eggs
INSERT INTO Step (RecipeID, StepNumber, Instruction) VALUES
(2, 1, 'Crack egg into a bowl and beat.'),
(2, 2, 'Pour into hot pan and stir gently.');

-- Milkshake
INSERT INTO Step (RecipeID, StepNumber, Instruction) VALUES
(3, 1, 'Blend milk, sugar, and ice until smooth.');

-- Insert RecipeIngredient
-- Pancakes
INSERT INTO RecipeIngredient (RecipeID, IngredientID, Quantity, UnitID) VALUES
(1, 1, 200, 1),  -- Flour, grams
(1, 2, 50, 1),   -- Sugar, grams
(1, 3, 300, 2),  -- Milk, ml
(1, 4, 1, 3);    -- Egg, tbsp

-- Scrambled Eggs
INSERT INTO RecipeIngredient (RecipeID, IngredientID, Quantity, UnitID) VALUES
```

Figure 5: Sample of insert commands used to populate the table

Sample data summary

The SQL-queries/Insert.sql file contains all INSERT INTO statements

Table - Rows Inserted

Recipe - 3 recipes

Ingredient - 5 ingredients

Unit - 3 units

Step - 7 steps

Category - 3 categories

RecipeIngredient - 6 entries

RecipeCategory - 4 entries

Each recipe has:

2+ Ingredients

Ordered preparation steps

One or more categories

Validation & Testing

Basic queries

```
select * from recipe
```

recipe 1

select * from recipe | Enter a SQL expression to filter results (use Ctrl+Space)

	recipeid	name	description	preptime	cooktime	servings
1	2	Scrambled Eggs	Quick and easy eggs	5	5	1
2	3	Milkshake	Cold blended milk drink	5	0	1
3	5	Fried Rice	Local dish made with rice and curry to get a greenish look	30	45	3
4	1	Pancake	Fluffy breakfast pancakes	10	15	2

Figure 6: basic query to show all recipes

```
SELECT r.Name AS Recipe, s.StepNumber, s.Instruction
FROM Recipe r
JOIN Step s ON r.RecipeID = s.RecipeID
ORDER BY r.Name, s.StepNumber;
```

recipe(+) 1

SELECT r.Name AS Recipe, s.StepNumber, s. | Enter a SQL expression to filter results (use Ctrl+Space)

	recipe	stepnumber	instruction
1	Milkshake	1	Blend milk, sugar, and ice until smooth.
2	Pancake	1	Mix flour, sugar, and salt in a bowl.
3	Pancake	2	Add milk and egg, then whisk until smooth.
4	Pancake	3	Heat a pan and pour in batter. Cook until golden.
5	Scrambled Eggs	1	Crack egg into a bowl and beat.
6	Scrambled Eggs	2	Pour into hot pan and stir gently.

Figure 7: Query to show recipe, stepnumber and instruction

First query shows all data in recipe table, we have 3 items, and they have their primary key which is the recipe id, their name, description, prep time, cook time and how many servings.

The second query shows recipes, the steps involved, the instructions to prepare from each step.

Connection Strings

For the database connection strings, it was done with appsettings.json, and initialized in the program.cs

```
namespace RecipeDatabaseApp
{
    internal class Program
    {
        static async Task Main(string[] args)
        {
            //Build configuration to read app setitngs json
            var config = new ConfigurationBuilder()
                .SetBasePath(Directory.GetCurrentDirectory())
                .AddJsonFile("appsettings.json", optional: false, reloadOnChange: true)
                .Build();
            var connectionString = config.GetConnectionString("DefaultConnection");

            var options = new DbContextOptionsBuilder<RecipeDbContext>()
                .UseNpgsql(connectionString)
                .Options;

            //Database context initialization
            using var dbContext = new RecipeDbContext(options);

            //run menu
            await RunMenu(dbContext);
            // 1. Initialize your database context here
            // e.g using var dbContext = new WebStoreContext();

            // 2. Pass the context to the RunMenu method to run a simple menu loop
            //RunMenu(dbContext);
        }
```

Figure 8: initializing db connection string

DBContext

This DbContext connects to a PostgreSQL database (recipe_db) and manages entities involved in a recipe application: Recipe, Ingredient, Step, Category, Unit, and junction entities like RecipeIngredient and RecipeCategory.

Entity Mappings

Recipe

- Table: recipe
- Primary Key: recipeid
- Has many:
  - Steps
  - RecipeIngredients
  - Categories (many-to-many via recipecategory)

Category

- Table: category
- Primary Key: categoryid
- Unique: name

- Many-to-many with Recipes via recipecategory

Ingredient

- Table: ingredient
- Primary Key: ingredientid
- Unique: name
- Related to Recipes via recipeingredient (many-to-many)

Unit

- Table: unit
- Primary Key: unitid
- Unique: name
- Used in RecipeIngredient to describe the quantity

Step

- Table: step
- Primary Key: stepid
- Belongs to one Recipe (via recipeid)
- Includes stepnumber and instruction

RecipeIngredient (junction table)

- Table: recipeingredient
- Composite Primary Key: { recipeid, ingredientid }
- Contains: quantity, unitid
- Foreign Keys:
  - recipeid → recipe
  - ingredientid → ingredient
  - unitid → unit

RecipeCategory (junction table)

- Table: recipecategory
- Composite Primary Key: { recipeid, categoryid }
- Many-to-many mapping using EF's UsingEntity<Dictionary<string, object>>

Implemented Features

CRUD operations

Create: Users can add new recipes to the database

```
internal async Task AddNewRecipe()
{
    Console.WriteLine("Write recipe name: ");
    string name = Console.ReadLine();

    Console.WriteLine("Write recipe description: ");
    string description = Console.ReadLine();

    Console.WriteLine("Write the preparation time (in minutes): ");
    int preptime = int.Parse(Console.ReadLine());

    Console.WriteLine("Write the cook time (in minutes): ");
    int cooktime = int.Parse(Console.ReadLine());

    Console.WriteLine("Write how many servings: ");
    int servings = int.Parse(Console.ReadLine());

    var recipe = new Recipe
    {
        Name = name,
        Description = description,
        Preptime = preptime,
        Cooktime = cooktime,
        Servings = servings

    };
    await _dbContext.Set<Recipe>().AddAsync(recipe);
    await _dbContext.SaveChangesAsync();

    Console.WriteLine("Recipe added successfully!");
}
```

Figure 9: create operations from addnewrecipe function

```
=== Recipe Database App ===
1. List All Recipes
2. Add New Recipe
3. Update Recipe
4. Delete Recipe
5. Fetch Recipes by Category
6. Search Recipes by Ingredients
7. Add Category to Recipe
8. Remove Category from Recipe
9. Extra option
0. Exit
Select an option: 2
Write recipe name:
Fried Rice
Write recipe description:
Local dish made with rice and curry to get a greenish look
Write the preparation time (in minutes):
30
Write the cook time (in minutes):
45
Write how many servings:
3
Recipe added successfully!

============== ************ ==============


=== Recipe Database App ===
1. List All Recipes
2. Add New Recipe
3. Update Recipe
4. Delete Recipe
5. Fetch Recipes by Category
6. Search Recipes by Ingredients
7. Add Category to Recipe
8. Remove Category from Recipe
9. Extra option
0. Exit
Select an option:
```

Figure 10: Adding new recipe to database

Read: At the launch of the console application, read operations are executed from the readallrecipes function.

```
public async Task ListAllRecipes()
{
    var recipes = await _dbContext.Set<Recipe>().ToListAsync();
    if (recipes.Count == 0)
    {
        Console.WriteLine("There were no recipes found!");
        return;
    }
    // Print out all Recipes
    Console.WriteLine("Recipe lists");
    foreach (var recipe in recipes)
    {
        Console.WriteLine($"ID: {recipe.Recipeid}, Name: {recipe.Name}, Description: {recipe.Description}");
    }
}
```

Figure 11: Listallrecipes function

```
=== Recipe Database App ===
1. List All Recipes
2. Add New Recipe
3. Update Recipe
4. Delete Recipe
5. Fetch Recipes by Category
6. Search Recipes by Ingredients
7. Add Category to Recipe
8. Remove Category from Recipe
9. Extra option
0. Exit
Select an option: 1
Recipe lists
ID: 2, Name: Scrambled Eggs, Description: Quick and easy eggs
ID: 3, Name: Milkshake, Description: Cold blended milk drink
ID: 5, Name: Fried Rice, Description: Local dish made with rice and curry to get a greenish look
ID: 1, Name: Pancake, Description: Fluffy breakfast pancakes
```

Figure 12: List all recipes in console app

Update: Recipes names can be updated from the commandline

```
internal async Task UpdateRecipe()
{
    await ListAllRecipes();

    Console.WriteLine("Select Recipe to update by its ID (number): ");
    if (!int.TryParse(Console.ReadLine(), out int recipeId))
    {
        Console.WriteLine("Invalid input. Please enter a valid numeric ID.");
        return;
    }

    var recipe = await _dbContext.Set<Recipe>().FirstOrDefaultAsync(r => r.Recipeid == recipeId);
    if (recipe == null)
    {
        Console.WriteLine($"No recipe found with ID {recipeId}.");
        return;
    }

    Console.WriteLine("\nSelect the number for the option you want to update: ");
    Console.WriteLine("1. Name");
    Console.WriteLine("2. Description");
    Console.WriteLine("3. Prep Time");
    Console.WriteLine("4. Cook Time");
    Console.WriteLine("5. Servings");
    Console.WriteLine("Choose: ");

    var choice = Console.ReadLine();

    switch (choice)
    {
        case "1":
            Console.Write("Enter new name: ");
            recipe.Name = Console.ReadLine();
            break;

        case "2":
            Console.Write("Enter new description: ");
            recipe.Description = Console.ReadLine();
```

Figure 13: updaterecipe function

```
=== Recipe Database App ===
1. List All Recipes
2. Add New Recipe
3. Update Recipe
4. Delete Recipe
5. Fetch Recipes by Category
6. Search Recipes by Ingredients
7. Add Category to Recipe
8. Remove Category from Recipe
9. Extra option
0. Exit
Select an option: 3
Recipe lists
ID: 1, Name: Pancakes, Description: Fluffy breakfast pancakes
ID: 2, Name: Scrambled Eggs, Description: Quick and easy eggs
ID: 3, Name: Milkshake, Description: Cold blended milk drink
ID: 5, Name: Fried Rice, Description: Local dish made with rice and curry to get a greenish look
Select Recipe to delte by its ID (number):
1

Select the number for the option you want to update:
1. Name
2. Description
3. Prep Time
4. Cook Time
5. Servings
Choose:
1
Enter new name: Pancake
Recipe updated successfully.

============== ************ ==============


=== Recipe Database App ===
1. List All Recipes
2. Add New Recipe
3. Update Recipe
4. Delete Recipe
5. Fetch Recipes by Category
6. Search Recipes by Ingredients
7. Add Category to Recipe
8. Remove Category from Recipe
9. Extra option
0. Exit
Select an option:
```

Figure 14: Updating recipe name from console

Delete: Recipes can be safely removed from databse

```
internal async Task DeleteRecipe()
{
    await ListAllRecipes();

    Console.WriteLine("Select Recipe to delete by its ID (number): ");
    if (!int.TryParse(Console.ReadLine(), out int recipeId))
    {
        Console.WriteLine("Invalid input. Please enter a valid numeric ID.");
        return;
    }

    var recipe = await _dbContext.Set<Recipe>()
        .Include(r => r.Steps)
        .FirstOrDefaultAsync(r => r.Recipeid == recipeId);

    if (recipe == null)
    {
        Console.WriteLine($"No recipe found with ID {recipeId}.");
        return;
    }

    _dbContext.Set<Recipe>().Remove(recipe);
    await _dbContext.SaveChangesAsync();
    Console.WriteLine($"Recipe with ID {recipeId} has been deleted.");

}
```

Figure 15: Deleterecipe function

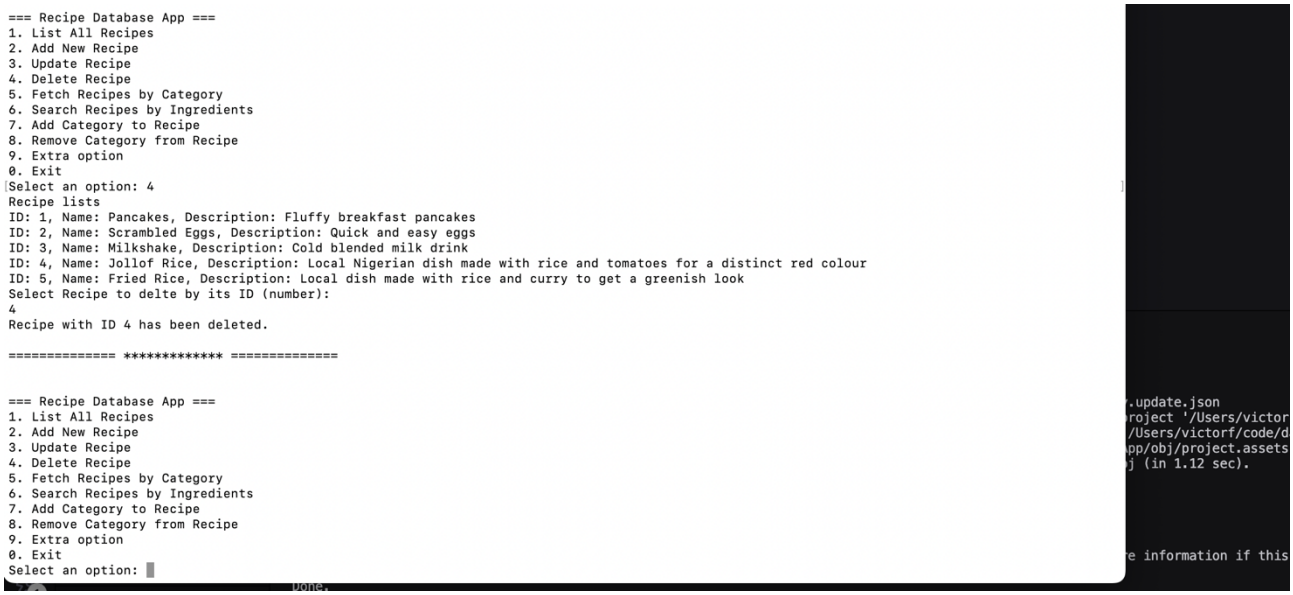

Figure 16: Recipe deletion in console app

Advanced queries and methods

Search for recipes by ingredients

```
internal async Task SearchRecipeByIngredients()
{
    Console.WriteLine("Enter ingredient names separated by commas (e.g., egg, milk, sugar):");
    string? input = Console.ReadLine();

    if (string.IsNullOrWhiteSpace(input))
    {
        Console.WriteLine("No ingredients provided.");
        return;
    }

    var ingredientNames = input.Split(',')
                               .Select(i => i.Trim().ToLower())
                               .Where(i => !string.IsNullOrEmpty(i))
                               .ToList();

    if (ingredientNames.Count == 0)
    {
        Console.WriteLine("No valid ingredients entered");
        return;
    }

    //To find matching ingredient ids
    var ingredientIds = await _dbContext.Set<Ingredient>()
        .Where(i => ingredientNames.Contains(i.Name.ToLower()))
        .Select(i => i.Ingredientid)
        .ToListAsync();

    if (ingredientIds.Count != ingredientNames.Count)
    {
        Console.WriteLine("Some ingredients were not found in the database.");
        return;
    }

    // Find recipes that contain ALL of the specified ingredients
    var recipes = await _dbContext.Set<Recipe>()
        .Where(r => ingredientIds.All(id =>
            r.Recipeingredients.Any(ri => ri.Ingredientid == id)))
        .ToListAsync();

    if (recipes.Count == 0)
    {
        Console.WriteLine("No recipes found with all the specified ingredients.");
        return;
    }
```

Figure 17: search recipe by ingredients function

Figure 18: search recipe by ingredient in console app

Challenges & lessons learned

Obstacles faced

EF Core Configuration: Setting up relationships in OnModelCreating was initially challenging, especially many-to-many associations between Recipes and Categories.

Junction Table Handling: Managing the recipecategory and recipeingredient tables required careful mapping to avoid duplication and ensure referential integrity.

LINQ Complexity: Writing LINQ queries to enforce conditions like “recipes must contain *all* specified ingredients” took extra research and trial/error.

Entity Linking: Ensuring ingredients, units, and steps were properly linked to recipes required both logical checks and proper navigation property usage.

Database Seeding: Manually populating initial data was tedious but essential for testing search and category features.

Key Takeaways

Learned how to model real-world relational data using EF Core and PostgreSQL effectively.

Gained confidence in navigating complex entity relationships, especially many-to-many mappings.

Improved skills in writing efficient LINQ queries and understanding EF Core behaviors.

Understood the importance of database constraints, indexing, and query performance in practical apps.

Learned how to separate concerns and structure a clean repository-style class for logic, even in a console app.

Conclusion

Project summary

This project resulted in a fully functioning .NET console application integrated with a PostgreSQL database. It allows users to:

- Add, update, and delete recipes.
- Search recipes by category or ingredients.
- Associate ingredients and categories with recipes.
- Maintain normalized, relational data with proper linking and querying.

The app showcases practical database interaction, LINQ proficiency, and understanding of EF Core mappings in a non-trivial real-world domain.

Future enhancements

Build a rating and comment system for community feedback.

Develop a GUI (desktop or web frontend) to improve usability.