Technical Documentation: Data Management

1. Database Schema Design

The application will store all recipe-related data in an Azure SQL Database. The updated schema follows industry best practices, using normalized tables to reduce redundancy, maintain data integrity, and facilitate efficient querying.

1.1 Ingredient Reference Table

Purpose: To store unique, standardized ingredient names and details.

- Fields:
  - IngredientRefID (Primary Key): Unique identifier for each ingredient.
  - IngredientName: Standardized name of the ingredient (e.g., 'brown onion').
  - IngredientCategory (Optional): Category of the ingredient (e.g., 'Onion').
  - NormalizedIngredientName: Normalized name for consistency checks (e.g., all lowercase, stripped of spaces).
  - UnitType (Optional): Default unit of measurement (e.g., 'grams', 'ml').

1.2 Recipes Table

Purpose: To store basic information about each recipe.

- Fields:
  - RecipeID (Primary Key): Unique identifier for each recipe.
  - RecipeName: Name of the recipe.
  - Description: Brief description of the recipe.
  - DateCreated: Timestamp of when the recipe was created.
  - DateModified: Timestamp of when the recipe was last modified.
  - ServingSize: Number of servings.
  - Tags: Tags for dietary preferences, allergens, etc.

1.3 RecipeIngredients Table (Junction Table)

Purpose: To manage the many-to-many relationship between recipes and ingredients.

- Fields:
  - RecipeIngredientID (Primary Key): Unique identifier for each record.
  - RecipeID (Foreign Key): Reference to the associated recipe.
  - IngredientRefID (Foreign Key): Reference to the associated ingredient from the Ingredient Reference Table.
  - Quantity: Amount of the ingredient needed for the recipe.
  - Unit: Unit of measurement (e.g., grams, cups).

1.4 Instructions Table

Purpose: To store detailed step-by-step instructions for each recipe.

- Fields:

- InstructionID (Primary Key): Unique identifier for each instruction step.
- RecipeID (Foreign Key): Reference to the associated recipe.
- StepNumber: Order of the step in the recipe.
- InstructionText: Text of the instruction.
- StepType: Type of instruction (e.g., Prep, Cooking, Presentation).

1.5 Images Table

Purpose: To store references to recipe images.

- Fields:

  - ImageID (Primary Key): Unique identifier for each image.
  - RecipeID (Foreign Key): Reference to the associated recipe.
  - ImageURL: URL where the image is stored in Azure Blob Storage.
  - DateUploaded: Timestamp of when the image was uploaded.

1.6 Indexing and Optimization:

- Although the current database size and usage do not require advanced indexing strategies, the design allows for future scalability. Indexing can be introduced to optimise query performance as the number of entries grows—potentially up to 500 or more. This will include considerations for Big O notation and efficient searching algorithms.

1.7 Data Types and Constraints:

- The database schema is designed to maintain data integrity and minimise duplication. Upon entry, ingredients and recipes are standardised to facilitate fuzzy matching. Constraints, such as unique constraints on RecipeName and referential integrity constraints between foreign keys, are enforced to ensure consistency and data quality.

1.8 Security Considerations:

- All data transfers between the application and the database are secured using HTTPS to ensure data privacy and prevent unauthorised access. Azure provides secure storage by default, and additional security measures, such as encryption at rest, are considered unnecessary at this stage due to the app's lack of personal data storage.

2. Ingredient Management

The application employs a standardised approach for managing ingredients with fuzzy matching to reduce duplication and ensure data consistency.

2.1 Ingredient Standardization

All ingredient entries are automatically standardised upon input by converting text to lowercase, trimming whitespace, and removing special characters. This ensures consistency in the Ingredient Reference Table.

2.2 Fuzzy Matching with User Prompt

After standardisation, the app performs a fuzzy matching check against existing entries in the Ingredient Reference Table. If a similar entry is found within a defined threshold, the user is prompted to confirm or correct the entry. If no match is found, the ingredient is added directly. If an exact match is found, the ingredient is added without prompting.

3. API Integration for Nutritional Analysis

The Recipe Book Application's backend uses **Azure Functions** to provide a serverless architecture for handling API requests. This setup allows the application to scale as needed while managing costs efficiently. The API layer will expose RESTful endpoints to perform CRUD operations on recipe data, enable advanced searching and filtering, and integrate with the Spoonacular API for additional functionality.

3.1 API Design and Endpoints

The Recipe Book Application's backend, built using **Azure Functions**, exposes RESTful API endpoints to manage recipes, perform CRUD operations, enable advanced searching and filtering, and support lazy loading. These endpoints minimise data transfer, optimise performance, and support scalability.

3.1.1 CRUD Operations for Recipes

The following endpoints manage the core CRUD (Create, Read, Update, Delete) operations for recipe data in the Azure SQL Database:

- **POST /recipes**: Create a new recipe.
  - **Request Format**: This format accepts a JSON payload containing recipe details such as name, description, ingredients, instructions, and tags.
  - **Response**: Returns the created recipe object with a unique RecipeID.

- **GET /recipes/{id}**: Retrieve a specific recipe by its ID.
  - **Request Format**: Requires a path parameter for the recipe ID.
  - **Response**: Returns the recipe details, including ingredients, instructions, and associated images. Supports lazy loading by initially loading only metadata and fetching detailed information on demand.
- **PUT /recipes/{id}**: Update an existing recipe.
  - **Request Format**: Requires a path parameter for the recipe ID and a JSON payload with updated recipe details.
  - **Response**: Returns the updated recipe object.
- **DELETE /recipes/{id}**: Delete a recipe.
  - **Request Format**: Requires a path parameter for the recipe ID.
  - **Response**: Returns a success message or an error if the recipe is not found.

3.1.2 Search and Filter Recipes

To optimise data fetching and improve user experience, the following endpoints are designed to support **pagination** and **lazy loading**:

- **GET /recipes/search?query={searchTerm}&limit={limit}&offset={offset}**: Search for recipes by name or description.
  - **Request Format**: Accepts a query parameter for searchTerm and optional parameters for limit and offset to support pagination.
  - **Response**: This function returns a paginated list of recipes matching the search criteria, along with metadata such as total results, limit, and offset.
- **GET /recipes/filter?tag={tag}&limit={limit}&offset={offset}**: Filter recipes by a specific tag.
  - **Request Format**: Accepts a query parameter for tag and optional parameters for limit and offset for pagination.
  - **Response**: Returns a paginated list of recipes that have the specified tag.

3.1.3 Manage Recipe Tags

To manage and utilise tags for filtering recipes, the following endpoint is available:

- **GET /recipes/tags**: Retrieve all unique tags from the recipes.
    - **Response**: Returns a list of unique tags for filtering and categorising recipes.

3.2 Spoonacular API Integration

To enhance the functionality of the Recipe Book Application, the Spoonacular API will be integrated to provide nutritional analysis for recipes:

- **Endpoint**: /recipes/parseIngredients
    - **Method**: POST
    - **Request Format**: A JSON payload containing a list of ingredients with their quantities and units.
        - **Example Payload**:
          { "ingredients": [{"name": "chicken breast", "quantity": 200, "unit": "grams"}, {"name": "olive oil", "quantity": 2, "unit": "tablespoons"}] }
    - **Response Handling**:
        - Parse the JSON response to extract nutritional information (e.g., calories, fat, sugar).
        - Store the calculated values in the **Dietary Information Table**.

- **Error Handling**:
    - Implement retry logic with exponential backoff for Spoonacular API call failures.
    - Log errors with detailed information for debugging, as per the error handling strategy defined below.

3.3 Authentication and Security

- ### 3.3.1 Single User Setup:
    - Since the application is intended for personal use, there is no need for user authentication. However, all API communication will be secured using HTTPS to protect data during transfer.
    - The application will not include API keys in any public repository, ensuring security. Future versions may include a guide on how to set up the application and manage API keys for personal use.

3.4 Error Handling and Logging

The API integration will adhere to the error handling and logging strategy outlined in the existing technical documentation:

- **3.4.1 Error Handling:**
  - Comprehensive error handling will be implemented for all data operations, including API calls and database access.
  - Use transaction management to ensure data integrity during CRUD operations. Transactions will roll back on failure to maintain consistency.
  - Specific error scenarios like network issues, API rate limits, and data inconsistencies will be managed with appropriate responses and retries.
- **3.4.2 Logging Strategy:**
  - Utilise a logging framework such as Serilog or NLog to capture errors, warnings, and critical system events.
  - Logs will be stored in an Azure log service or local files, making them accessible for monitoring and debugging.

3.5 Performance Optimization

- **3.5.1 On-Demand Data Fetching:**
  - The app will pull individual recipes or small subsets of recipes from the database as needed rather than loading all data on startup.
  - This approach minimises initial load times and reduces unnecessary data transfer, making the app more responsive and scalable as the number of recipes grows.
- **3.5.2 In-Memory Caching:**
  - Recently viewed recipes will be cached in memory for quick access. Cached data will persist for the app session and be discarded upon closure.
  - A caching mechanism will ensure efficient memory usage by removing older, less frequently accessed data from memory.
- **3.5.3 Lazy Loading for Lists:**
  - When displaying lists of recipes (e.g., search results, tag filters), only a subset of recipes will be loaded initially, with additional recipes loaded as needed.
  - This will enhance performance, mainly when dealing with large datasets, by reducing the data fetched and displayed simultaneously.
- **3.5.4 Network Efficiency:**
  - All API calls and data transfers will use HTTPS to ensure secure communication, with optimisations for reducing data payload sizes wherever possible.

4. Data Synchronization Strategy

On application startup, check for updates and download only the metadata (e.g., recipe names, IDs) necessary for initial navigation. The full details of recipes, including ingredients and instructions, will be fetched from the Azure SQL Database only when a user selects a specific recipe. Images and large data will be loaded lazily to minimise data transfer and storage costs.

5. Data Backup and Recovery

Backup Configuration: Use Azure SQL's automatic backup functionality to create weekly backups.

Retention Policy: For the last five weeks, maintain a rolling backup, automatically deleting older backups to manage storage.

Backup Exclusions: Configure backups to exclude image files, focusing only on critical recipe data to save storage space and costs.

Development Milestones and Timeline

The development of the Recipe Book Application will follow a phased approach aimed at building a proof of concept (POC) to validate the chosen technology stack. The primary goal is to develop a basic, functional application with a GUI for mobile (Android) and PC (Windows 11) using .NET MAUI that connects to an Azure backend. The POC will demonstrate the ability to store and retrieve a single pre-made recipe from an Azure SQL Database. Subsequent phases will focus on refining and expanding the functionality based on the validated POC.

Key Phases:

- **Planning and Setup:**
    - Establish the development environment using .NET MAUI within Visual Studio Code for cross-platform development targeting Android and Windows 11.
    - Install the necessary tools for mobile app testing on a PC, including an Android Emulator and an Android Debug Bridge (ADB).
    - Set up the Azure backend, including configuring an Azure SQL Database for storing recipe data.
    - Define a basic schema for the database to manage recipe data, including tables for recipes, ingredients, instructions, and images.
- **Backend Development (Phase 1):**
    - Develop the initial backend using Azure Functions to handle API requests and manage data storage in the Azure SQL Database.
    - Implement basic CRUD (Create, Read, Update, Delete) operations for managing recipe data. Critical API endpoints include:
        - `POST /recipes`: Create a new recipe.
        - `GET /recipes/{id}`: Retrieve a specific recipe by its ID.
        - `PUT /recipes/{id}`: Update an existing recipe.
        - `DELETE /recipes/{id}`: Delete a recipe.
    - Test the API endpoints locally using Postman to ensure they correctly interact with the Azure SQL Database, validate the efficiency of lazy loading,

caching, and on-demand fetching strategies, and perform the necessary CRUD operations on a pre-made recipe

- **Rudimentary Frontend Development (Phase 1):**
    - Develop a GUI for mobile (Android) and PC (Windows 11) using .NET MAUI that supports lazy loading and on-demand data fetching. The GUI will initially display recipe metadata (e.g., titles) and will fetch full recipe details, including ingredients and instructions, only when selected by the user. Cached recipes will be kept in memory during the session and discarded upon closure to optimize performance.
    - Set up and configure the Android Emulator within Visual Studio Code to rapidly test and iterate the mobile version.
- **Backend Refinement and Integration (Phase 2):**
    - Refine the backend API to support more advanced operations, such as handling additional recipe data and managing device synchronisation.
    - Integrate the Spoonacular API to retrieve calorie and nutrient calculations. Implement additional API endpoints to support this feature.
    - Conduct integration tests to ensure smooth communication between the custom backend API and the Spoonacular API.
- **Frontend Enhancement (Phase 2):**
    - Enhance the GUI design for both platforms using .NET MAUI, focusing on user-friendliness and visual appeal.
    - Improve features such as responsive design, accessibility options (e.g., dark mode), and a more polished user interface.
    - Conduct user interface testing using tools like xUnit for .NET MAUI.
- **Testing and Validation:**
    - Perform comprehensive integration testing to ensure all components (frontend, backend, Azure, APIs) work together seamlessly.
    - Conducted unit tests for critical functionalities using xUnit for .NET MAUI and validated API operations using Postman.
    - I will use the Android emulator to test the mobile application on the PC to streamline development and avoid constantly rebuilding the program on my device.
- **Deployment and Review:**
    - Deploy the POC to local devices, ensuring the desktop (Windows 11) and mobile (Android) versions are functional.
    - Review the application's performance, functionality, and user experience to identify any areas for improvement or optimisation.

6. Error Handling and Logging

- ### 6.1 Error Handling:
  - Implement comprehensive error handling for all data operations (e.g., database access API calls).
  - Use transaction management to ensure data integrity during CRUD operations. Rollback transactions on failure.
  - Handle specific error scenarios like network issues, authentication failures, and data inconsistencies.
- ### 6.2 Logging Strategy:
  - Utilise a logging framework such as Serilog or NLog to capture errors, warnings, and critical system events.
  - Configure log storage to an Azure log service or local files for easier access and monitoring.

7. Image Compression and Storage Feasibility

7.1 Overview

This section explores the feasibility of storing 200 recipe images and text data within a 5GB storage limit using JPEG XL compression. The goal is to ensure the application remains scalable while balancing image quality and storage efficiency. Similar to recipes, image data will be loaded lazily on demand to minimise initial data transfer and optimise performance. Once loaded, images may be temporarily cached in memory during the session to prevent repeated fetching and reduce data transfer.

7.2 Image Compression Analysis

7.2.1 Average File Size for Source Images

A standard photo taken by the Google Pixel 5 camera (12MP, 4000 x 3000 pixels) in JPEG format is between 3MB and 6MB. The size variation depends on the image's complexity, colour range, and level of detail.

7.2.2 JPEG XL Compression Efficiency

JPEG XL is a modern image format offering lossless and lossy compression, providing better compression ratios than older formats like JPEG and WebP.

7.2.3 Compression Estimates:

- Lossless Compression:
    - Typically achieves a 20-40% reduction in file size.
    - Estimated size per image after compression: 1.8MB to 4.8 MB.
- Lossy Compression:
    - Achieves a more significant 60-80% reduction while maintaining good visual quality.
    - Estimated size per image after compression: 0.6MB to 2.4 MB.

7.3 Total Storage Requirement Calculations

7.3.1 Storage Requirements for Images

- For 200 images:
    - Lossless Compression:
        - Minimum Estimate: 200 x 1.8MB = 360MB
        - Maximum Estimate: 200 x 4.8MB = 960MB
    - Lossy Compression:
        - Minimum Estimate: 200 x 0.6MB = 120MB
        - Maximum Estimate: 200 x 2.4MB = 480MB

7.3.2 Storage Requirements for Text Data

Text data per recipe (title, ingredients, steps, tags, etc.) is estimated at around 10KB.

For 200 recipes: Total text data: 200 x 10KB = 2MB

7.3.3 Combined Storage Estimates

- Lossless Compression:
    - Total Minimum: 362MB (360MB images + 2MB text)
    - Total Maximum: 962MB (960MB images + 2MB text)
- Lossy Compression:
    - Total Minimum: 122MB (120MB images + 2MB text)
    - Total Maximum: 482MB (480MB images + 2MB text)

7.4 Conclusion and Recommendations

- Feasibility:
    - Both lossless and lossy JPEG XL compression methods allow storing 200 images and associated recipe data within the 5GB storage limit.
- Recommended Strategy:
    - Lossless Compression is advisable if image quality is a priority, as it maintains full image fidelity while offering significant storage savings.
    - If maximising storage capacity is more critical, Lossy Compression should be considered, as it provides a higher compression ratio while retaining acceptable visual quality.

7.5 Next Steps

- Implement the JPEG XL compression using available libraries (libjxl or tools like cjxl) in the chosen development environment.
- Conduct testing with sample images to verify compression effectiveness and fine-tune parameters for optimal quality and storage balance.

7.6 Implementation Notes

7.6.1 Libraries and Tools:

- JPEG XL Libraries: Utilize libjxl for cross-platform compatibility.
- Alternative Tools: Consider ImageMagick, which supports JPEG XL, for additional image processing capabilities.

7.6.2 Considerations for Color Data:

- Higher compression rates may impact colour fidelity; testing should include diverse images to evaluate acceptable quality levels.

7.6.3 Fallback Options:

- If JPEG XL support proves inadequate or compatibility issues arise, WebP remains a viable fallback option with broader platform support.

8. Possible Error Scenarios

A detailed listing of potential error scenarios affecting the project, database, and client-side interactions has been added for reference and future planning.

8.1 Database Connection Errors

- Timeout: The application fails to connect with the Azure SQL Database within the specified time limit.
- Network Issues: Loss of network connectivity between the app and the database server could cause connection failures.
- Authentication Failures: Incorrect or expired credentials when connecting to the Azure SQL Database.

8.2 Data Retrieval Errors

- Data Not Found: Attempting to retrieve a recipe or other data that does not exist in the database.
- Data Format Errors: There is a mismatch between the expected data format (e.g., JSON, XML) and the format returned by the database.
- Invalid Query Syntax: Malformed SQL queries sent to the database, causing query execution to fail.

8.3 Data Update and Insert Errors

- Duplicate Entry: Attempting to insert data that violates unique constraints (e.g., adding a recipe with a duplicate name).
- Constraint Violations: Violating database constraints such as foreign keys, unique or check constraints.
- Transaction Failures: Errors occurring during a transaction (e.g., inserting multiple records where one fails), leading to incomplete or partial updates.

8.4 Data Synchronization Issues

- Conflicts between the local data on the device and the data in the cloud database during synchronisation.
- Partial Data Transfer: Incomplete data transfer due to network issues or interruptions during synchronisation.

8.5 API Errors

- Rate Limit Exceeded: When the predefined rate limit for API requests is exceeded, the server temporarily rejects further requests.
- Invalid API Requests: Malformed requests sent to the API (e.g., missing parameters, incorrect HTTP methods).
- Server-Side Errors: 500-series errors (e.g., Internal Server Error, Service Unavailable) due to backend issues or unexpected server failures.

8.6 Security-Related Errors

- Unauthorised Access: Unauthorized attempts to access the API or database due to incorrect or missing authentication.
- Data Leakage: Potential exposure of sensitive data if encryption or HTTPS is not correctly implemented.

8.7 Storage and Cost Management Errors

- Exceeded Storage Quota: Exceeding the 5GB storage limit in Azure, potentially incurring additional costs.
- Early Deletion Fees: Deleting data before the minimum storage duration leads to additional charges.

8.8 Client-Side Application Errors

- UI Errors: Errors in rendering the user interface, such as missing elements, unresponsive buttons, or display issues.
- App Crashes: Application crashes due to unhandled exceptions, memory leaks, or excessive resource consumption.
- Compatibility Issues: Incompatibility with specific devices or operating system versions, leading to unexpected behaviour.

8.9 Logging and Monitoring Failures

- Logging Failures: Errors in the logging framework, such as failure to write logs to a file or database.
- Insufficient Error Data: Missing or incomplete log information makes debugging difficult.