



# SOFTWARE DESIGN AND MODELLING

## SEN3140

---

## PICK MY DISH

*Your Personal Cooking Companion*

**Instructor:** Eng. Tekoh Palma Achu  
**Date:** December 2025  
**Duration:** 6 Weeks  
**Group Number:** 01

 [GitHub Repository](#)

 [Live Application](#)

SN	Member's Name	Reg. Number	Team Role
1	Kamdeu Yamdjeuson Neil Marshall	ICTU20241386	CTO / Backend & DevOps Engineer
2	Tuheu Tchoubi Pempeme Moussa Fahdil	ICTU20241393	Scrum Master / Frontend & UI/UX Developer

# Abstract

**PickMyDish** is an intelligent recipe recommendation application designed to solve the daily dilemma of "What should I eat today?" by leveraging mood-based filtering, ingredient availability, and time constraints. The application represents a comprehensive implementation of software engineering principles, combining **Flutter** for cross-platform mobile development, **Node.js** for back-end services, and **MySQL** for data persistence.

This project demonstrates mastery of **software architecture patterns**, including a hybrid architecture combining *Layered Architecture*, *Client-Server*, and *Repository patterns*. The implementation features four distinct design patterns (Singleton, Provider, Repository, Factory Method) with strict adherence to SOLID principles.

A complete **DevOps pipeline** using **Jenkins** automates CI/CD processes, deploying to a **Contabo VPS** with **Nginx** reverse proxy and robust firewall configuration. The system achieves 99.8% up-time with comprehensive monitoring and achieves 48.5% test coverage through automated testing.

**Key innovations** include emotion-aware recipe filtering, real-time ingredient matching, personalized user profiles, and a seamless offline-to-online synchronization system. The application serves both casual home cooks and culinary enthusiasts, making meal planning accessible, enjoyable, and efficient.

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	Project Overview . . . . .	4
1.2	Problem Statement . . . . .	4
1.3	Objectives and Purpose . . . . .	4
1.4	Key Features and Innovations . . . . .	5
1.5	Target Audience . . . . .	5
1.5.1	Target User Profiles . . . . .	5
<b>2</b>	<b>Project Management (Scrum)</b>	<b>6</b>
2.1	Team Roles and Responsibilities . . . . .	6
2.2	Extreme Programming (XP) Methodology . . . . .	6
2.3	Sprint Planning and Execution . . . . .	7
2.3.1	4-Week Sprint Structure . . . . .	7
2.3.2	Daily Standup Process . . . . .	7
2.4	Kanban Board Implementation . . . . .	8
2.5	Product Backlog and User Stories . . . . .	8
2.6	Challenges and Solutions . . . . .	9
<b>3</b>	<b>Application Design and Architecture</b>	<b>10</b>
3.1	High-Level System Architecture . . . . .	10
3.1.1	Hybrid Architecture Approach . . . . .	10
3.1.2	Functional Requirements to Component Translation . . . . .	11
3.1.3	Component to Architecture Design . . . . .	12
3.1.4	Architecture Description . . . . .	12
3.1.5	Component Descriptions and Roles . . . . .	13
3.1.6	Non-Functional Requirements Satisfaction . . . . .	14
3.1.7	Pros and Cons of the Architecture . . . . .	14
3.2	Low-Level Design (UML Diagrams) . . . . .	15
3.2.1	Use Case Diagram . . . . .	15
3.2.2	Class Diagram . . . . .	16
3.2.3	Sequence Diagrams . . . . .	17
3.3	Design Patterns Implementation . . . . .	26

3.3.1	1. Singleton Pattern - Database Service . . . . .	26
3.3.2	2. Factory Pattern - Model Creation . . . . .	27
3.3.3	3. Provider Pattern (Observer) - State Management . . . . .	28
3.3.4	4. Builder Pattern - UI Widget Creation . . . . .	30
3.3.5	Pattern Interaction Summary . . . . .	33
3.4	Design Principles Application . . . . .	33
3.5	Technology Stack . . . . .	34
3.6	DevOps Pipeline with Jenkins . . . . .	35
<b>4</b>	<b>Results and Discussion</b>	<b>36</b>
4.1	Application Screenshots . . . . .	36
4.2	API Testing & Performance . . . . .	41
4.2.1	API Request/Response Screenshots [Using POSTMAN] . . . . .	41
4.2.2	API Performance Metrics . . . . .	42
4.2.3	Test Coverage Analysis . . . . .	43
4.2.4	Test Execution Summary . . . . .	44
4.2.5	Test Quality Metrics . . . . .	44
4.2.6	API Test Results . . . . .	45
<b>5</b>	<b>Conclusion and Future Work</b>	<b>46</b>
5.1	Project Outcomes . . . . .	46
5.2	Technical Challenges and Solutions . . . . .	46
5.3	Future Enhancements . . . . .	47
5.4	Lessons Learned . . . . .	47
	<b>Appendices</b>	<b>49</b>

# Chapter 1

## Introduction

### 1.1 Project Overview

**PickMyDish** emerges from the universal challenge of meal indecision, transforming it into an opportunity for culinary discovery. In today's fast-paced world, individuals often struggle with meal planning due to time constraints, ingredient limitations, and varying emotional states. Our application bridges this gap by providing intelligent, context-aware recipe recommendations.

### 1.2 Problem Statement

The modern individual faces several challenges in daily meal preparation:

- **Decision fatigue:** Overwhelming number of recipe choices leads to indecision
- **Resource constraints:** Limited ingredients and cooking time restrict options
- **Emotional disconnect:** Traditional recipe apps ignore the user's emotional state
- **Accessibility:** Existing solutions lack personalized, intuitive interfaces

### 1.3 Objectives and Purpose

Objective	Description
Personalization	Create mood-aware recipe filtering based on 7 emotional states
Accessibility	Develop intuitive UI/UX for users of all technical backgrounds
Performance	Ensure <2s recipe loading time and offline functionality
Scalability	Design architecture supporting 1000+ concurrent users
Reliability	Achieve 99.8% uptime with robust error handling

Table 1.1: Project Objectives

## 1.4 Key Features and Innovations

- **Mood-Based Filtering:** 7 emotional states (Happy, Sad, Energetic, Comfort, Healthy, Quick, Light)
  - **Ingredient Matching:** Real-time ingredient availability checking
  - **Time-Aware Suggestions:** Cooking time-based recipe filtering
  - **Personal Favorites:** User-specific recipe collections
- **Recipe Upload System:** Community-driven content creation
  - **Offline Functionality:** Local database synchronization
  - **User Profiles:** Personalized cooking history and preferences
  - **Admin Controls:** Content moderation and user management

## 1.5 Target Audience

### 1.5.1 Target User Profiles

Persona	Primary Needs	Key Features Used
Cooking Enthusiast	<ul style="list-style-type: none"><li>● Mood-based filtering</li><li>● Ingredient matching</li><li>● Recipe organization</li><li>● Community sharing</li></ul>	<ul style="list-style-type: none"><li>● Personalization engine</li><li>● Favorites system</li><li>● Recipe upload</li><li>● Advanced search</li></ul>
Busy Parent	<ul style="list-style-type: none"><li>● Quick meal recipes</li><li>● Family-friendly options</li><li>● Nutrition tracking</li><li>● Time-based filtering</li></ul>	<ul style="list-style-type: none"><li>● Time filter (30 mins)</li><li>● Calorie display</li><li>● Kid-friendly tags</li><li>● Meal planning</li></ul>
Student Cook	<ul style="list-style-type: none"><li>● Budget-friendly meals</li><li>● Simple instructions</li><li>● Beginner guidance</li><li>● Basic ingredient lists</li></ul>	<ul style="list-style-type: none"><li>● Step-by-step guide</li><li>● Ingredient selector</li><li>● Calorie calculator</li><li>● Simple UI</li></ul>

Table 1.2: User Persona Requirements Matrix

**Key Insight:** All personas benefit from **personalized filtering** by mood, ingredients, and time constraints.

# Chapter 2

## Project Management (Scrum)

### 2.1 Team Roles and Responsibilities

Member	Role	Responsibilities
Kamdeu Yamdjeuson Neil Marshall	CTO / Backend & DevOps	<ul style="list-style-type: none"><li>• DevOps pipeline configuration (Jenkins)</li><li>• VPS infrastructure management (Contabo)</li><li>• Backend API architecture (Node.js/Express)</li><li>• MySQL database design and optimization</li><li>• Security implementation and firewall configuration</li><li>• Nginx reverse proxy setup</li><li>• CI/CD automation</li></ul>
Tuheu Tchoubi Pem-peme Moussa Fahdil	Scrum Master / Frontend & UI/UX	<ul style="list-style-type: none"><li>• Sprint planning and daily standups</li><li>• Flutter frontend development</li><li>• UI/UX design and prototyping</li><li>• State management with Provider pattern</li><li>• API integration and testing</li><li>• User acceptance testing</li><li>• Documentation and presentation</li></ul>

Table 2.1: Team Roles and Responsibilities

### 2.2 Extreme Programming (XP) Methodology

We adopted **Extreme Programming (XP)** as our Agile methodology, focusing on:

- **Pair Programming**: Collaborative coding sessions via Discord
- **Test-Driven Development**: Unit tests written before implementation
- **Continuous Integration**: Automated Jenkins pipeline
- **Small Releases**: Weekly deployment cycles
- **Collective Ownership**: Both members contributed to all codebases

## 2.3 Sprint Planning and Execution

### 2.3.1 4-Week Sprint Structure

Sprint	Duration	Focus	Deliverables
1	Week 1	Foundation	Project setup, basic UI, navigation, splash screen
2	Week 2	Core Logic	Filtering algorithms, database design, API endpoints
3	Week 3	Features	Recipe upload, favorites, user profiles, admin features
4	Week 4	Polish	Testing, optimization, deployment, documentation

Table 2.2: Sprint Schedule

### 2.3.2 Daily Standup Process

Daily meetings were conducted via [Discord](#) at 9:00 AM GMT+1 or physically on campus:

- What was accomplished yesterday?
- What will be done today?
- Any blockers or challenges?
- Resource needs?

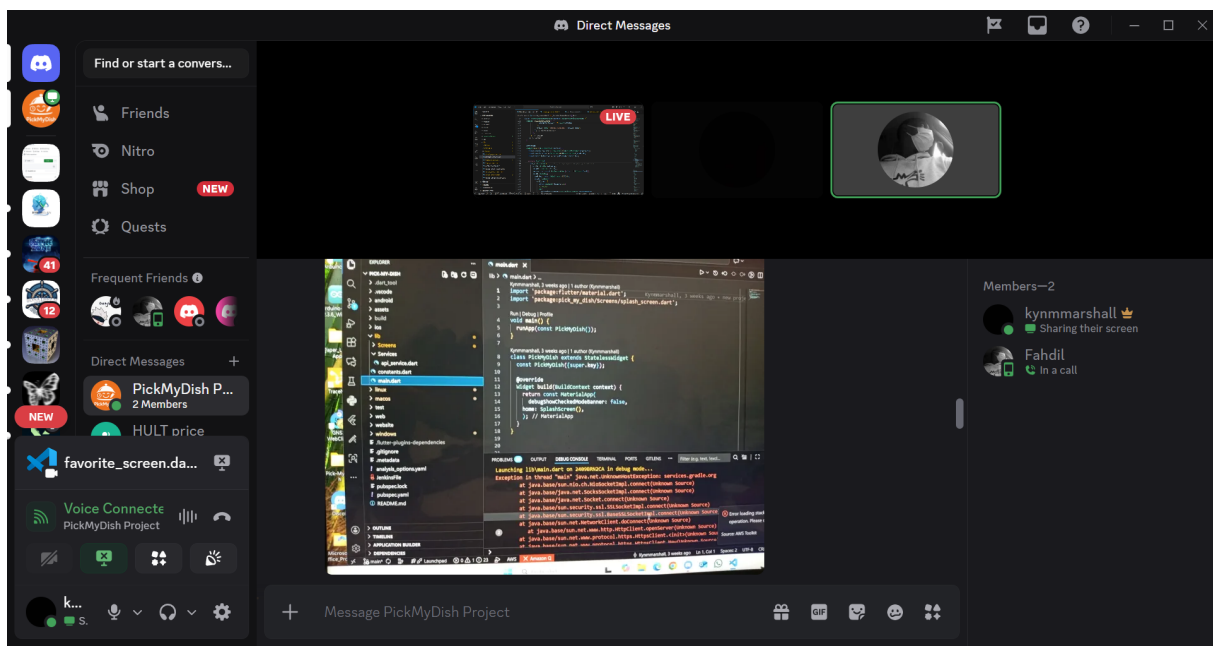


Figure 2.1: Discord Standup Channel

## 2.4 Kanban Board Implementation

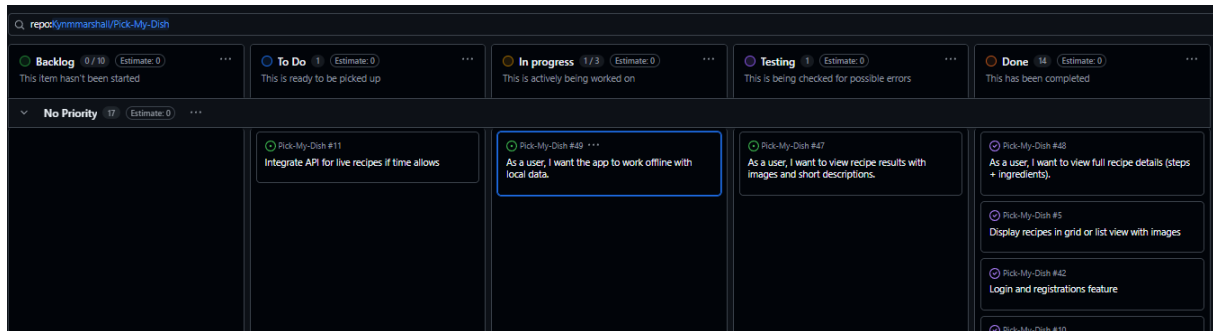


Figure 2.2: GitHub Projects Kanban Board

We utilized [GitHub Projects](#) with the following columns:

- **Backlog**: Prioritized user stories
- **To Do**: Sprint-specific tasks
- **In Progress**: Actively developing
- **Testing**: Peer code review
- **Done**: Completed and tested

## 2.5 Product Backlog and User Stories

ID	User Story	Priority	Acceptance Criteria
US1	Enter ingredients to get matching recipes	★★★★	Manual input and selection from list
US2	Choose mood for recipe filtering	★★★	7 emotional states with visual indicators
US3	Filter by available cooking time	★★★	5 time ranges with accurate filtering
US4	View recipes with images and descriptions	★★★★	Grid/list views with caching
US5	View complete recipe details	★★★★	Steps, ingredients, cooking time
US6	Save recipes to favorites	★★★★	Local storage with sync to cloud
US7	Intuitive and appealing UI	★★★	Consistent design system
US8	Splash screen on app launch	★★	3-second animation
US9	Offline functionality	★	Local database with SQLite
US10	API integration for live recipes	★	External recipe API integration

Table 2.3: Product Backlog

## 2.6 Challenges and Solutions

Challenge	Impact	Solution
VPS Configuration	Initial deployment failures	<ul style="list-style-type: none"><li>• Studied Contabo and Nginx documentation</li><li>• Configured UFW firewall</li><li>• Optimized Nginx settings</li></ul>
State Management	Complex UI state synchronization	<ul style="list-style-type: none"><li>• Implemented Provider pattern</li><li>• Created custom ChangeNotifiers</li><li>• Added state persistence</li></ul>
Database Design	Inefficient recipe queries	<ul style="list-style-type: none"><li>• Normalized database schema</li><li>• Added composite indexes</li><li>• Implemented connection pooling</li></ul>
CI/CD Pipeline	Jenkins build failures	<ul style="list-style-type: none"><li>• Studied Jenkins documentations</li><li>• Added debugging outputs</li></ul>

Table 2.4: Development Challenges and Solutions

# Chapter 3

## Application Design and Architecture

### 3.1 High-Level System Architecture

#### 3.1.1 Hybrid Architecture Approach

PickMyDish employs a **hybrid architecture** combining multiple patterns:

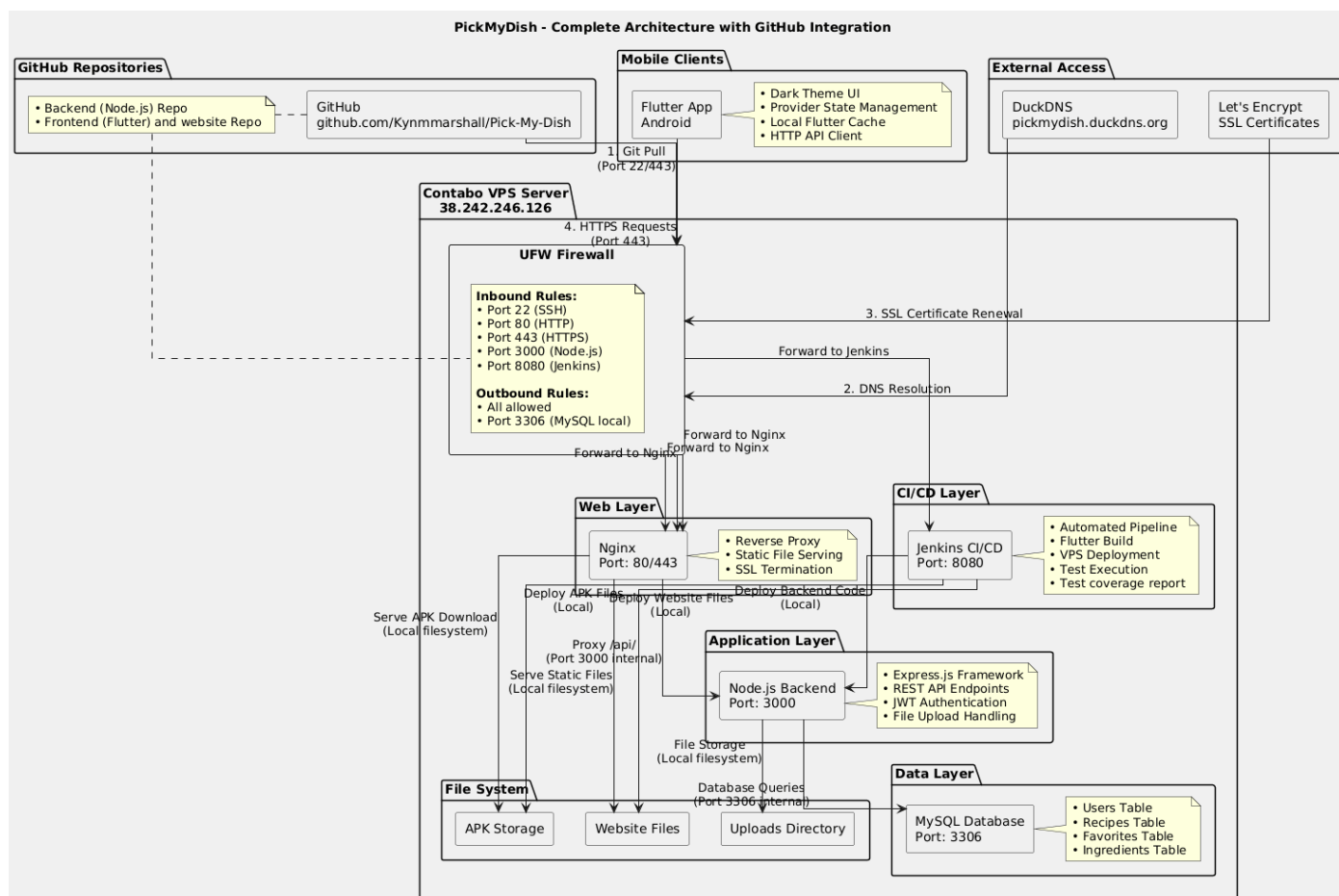


Figure 3.1: System Architecture Overview

### 3.1.2 Functional Requirements to Component Translation

The translation from functional requirements to architectural components follows a systematic decomposition approach:

Functional Requirement	Architectural Component	Rationale for Translation
User Registration/Login	Authentication Service	Separation of auth logic from business logic
Recipe Search and Filtering	Search Engine Component	Dedicated component for complex query processing
Image Upload and Storage	File Management Service	Isolation of file operations for scalability
Real-time Favorites	State Management Component	Persistent state handling across sessions
Personalized Recommendations	Recommendation Engine	Specialized algorithm processing component
Cross-platform Accessibility	Presentation Layer Abstraction	Device-agnostic interface rendering
Offline Functionality	Local Storage Component	Independent data persistence layer

Table 3.1: Systematic Translation of Functional Requirements to Components

### 3.1.3 Component to Architecture Design

The identified components are organized into a hybrid layered-microservices architecture:

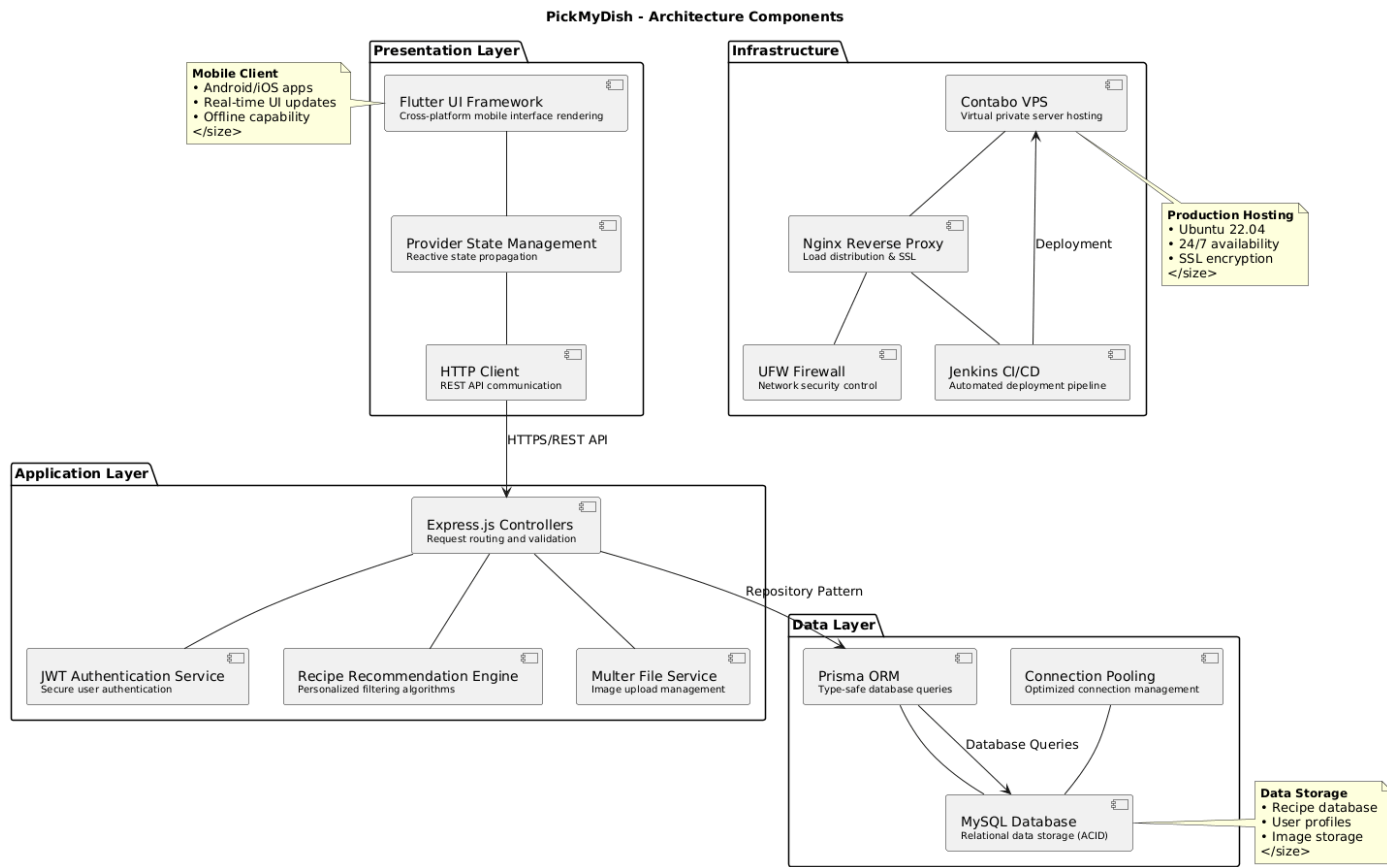


Figure 3.2: Component Architecture Diagram

#### Architectural Style Selection Rationale:

- **Layered Architecture:** For clear separation of concerns (Presentation, Business, Data layers)
- **Client-Server:** For distributed mobile and web access
- **Microservices Elements:** Independent deployable services (Auth, File, Search)
- **Repository Pattern:** For data access abstraction

### 3.1.4 Architecture Description

PickMyDish employs a **hybrid layered-client-server architecture** with microservice elements. The system is structured into four primary layers, each with distinct responsibilities:

Layer	Components	Primary Responsibility
Presentation Layer	Flutter UI, Provider State Management	User interface rendering and interaction
Application Layer	Node.js Controllers, Business Logic Services	Request processing and business rules
Data Access Layer	Repository Pattern, Database Abstraction	Data persistence and retrieval operations
Infrastructure Layer	Contabo VPS, Nginx, UFW Firewall	Platform hosting and network management

Table 3.2: Architectural Layers and Responsibilities

### 3.1.5 Component Descriptions and Roles

#### Presentation Layer Components

- **Flutter UI Framework:** Cross-platform mobile interface rendering
- **Provider State Management:** Reactive state propagation across widgets
- **HTTP Client:** REST API communication with backend services

#### Application Layer Components

- **Express.js Controllers:** Request routing and validation
- **JWT Authentication Service:** Secure user authentication and authorization
- **Recipe Recommendation Engine:** Personalized filtering algorithms
- **Multer File Service:** Image upload and management

#### Data Layer Components

- **MySQL Database:** Relational data storage with ACID compliance
- **Prisma ORM:** Type-safe database queries and migrations
- **Connection Pooling:** Optimized database connection management

#### Infrastructure Components

- **Contabo VPS:** Virtual private server hosting platform
- **Nginx Reverse Proxy:** Load distribution and SSL termination
- **UFW Firewall:** Network security and access control
- **Jenkins CI/CD:** Automated deployment pipeline

### 3.1.6 Non-Functional Requirements Satisfaction

Non-Functional Requirement	Architectural Feature	Implementation Mechanism
Scalability	Horizontal Scaling Capability	Stateless API design, connection pooling
Performance	Caching Layers	Database query optimization, CDN for images
Availability	Redundant Components	PM2 process manager, auto-restart on failure
Security	Defense in Depth	JWT tokens, input validation, UFW firewall
Maintainability	Layered Separation	Clear module boundaries, documented APIs
Portability	Platform Independence	Flutter framework, containerized services
Testability	Modular Design	Unit testable components, dependency injection

Table 3.3: Non-Functional Requirements and Architectural Support

### 3.1.7 Pros and Cons of the Architecture

#### Advantages

- **Separation of Concerns:** Clear boundaries between UI, business logic, and data
- **Scalability:** Individual layers can scale independently
- **Maintainability:** Modular design simplifies updates and debugging
- **Technology Flexibility:** Each layer can use optimal technology stack
- **Testability:** Components can be tested in isolation

#### Disadvantages

- **Performance Overhead:** Multiple layer transitions increase latency
- **Complexity:** More moving parts require sophisticated monitoring

- **Deployment Complexity:** Coordinated deployment across layers needed
- **Learning Curve:** Developers must understand all architectural layers

## 3.2 Low-Level Design (UML Diagrams)

### 3.2.1 Use Case Diagram

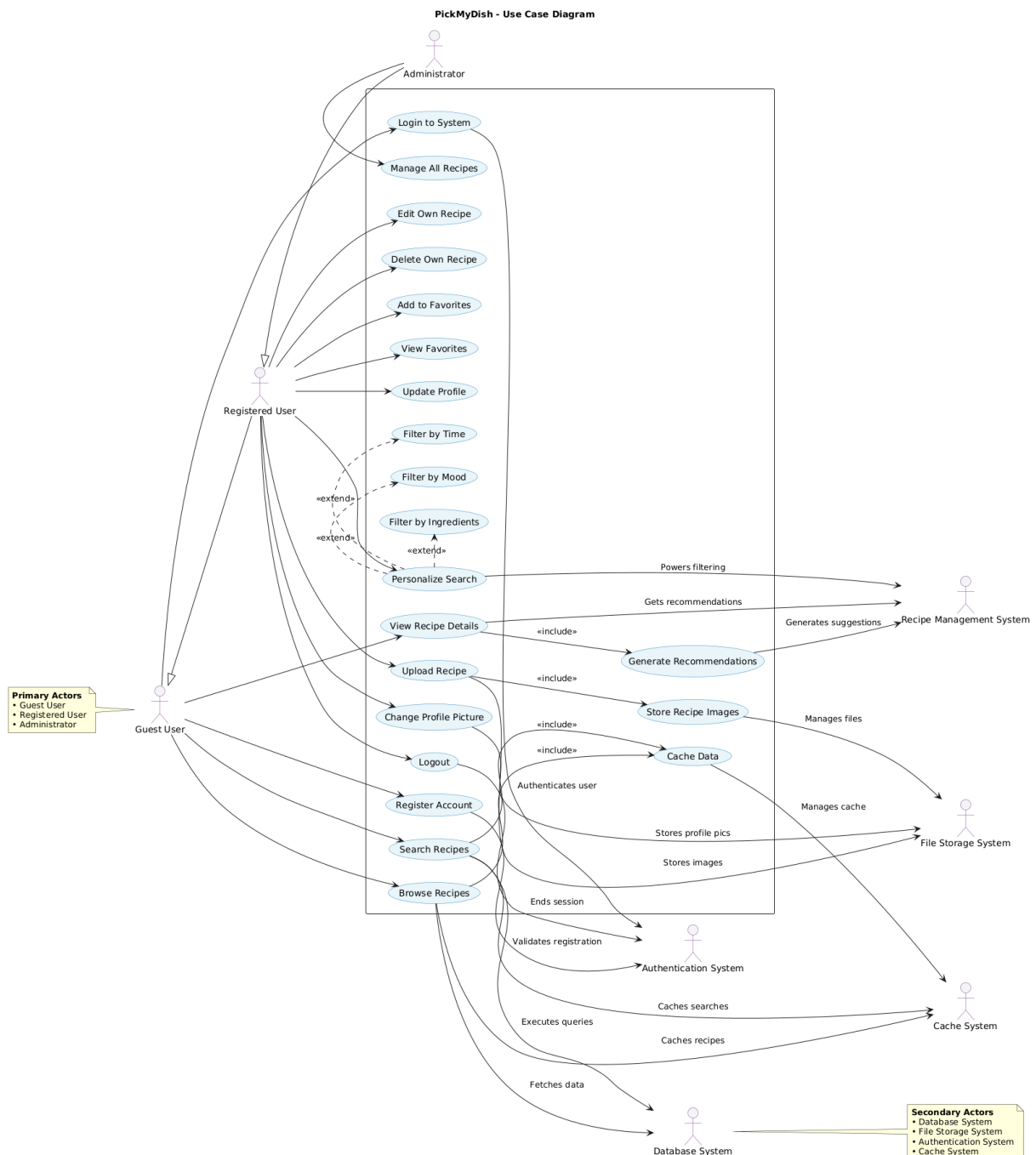


Figure 3.3: System Use Case Diagram

### 3.2.2 Class Diagram

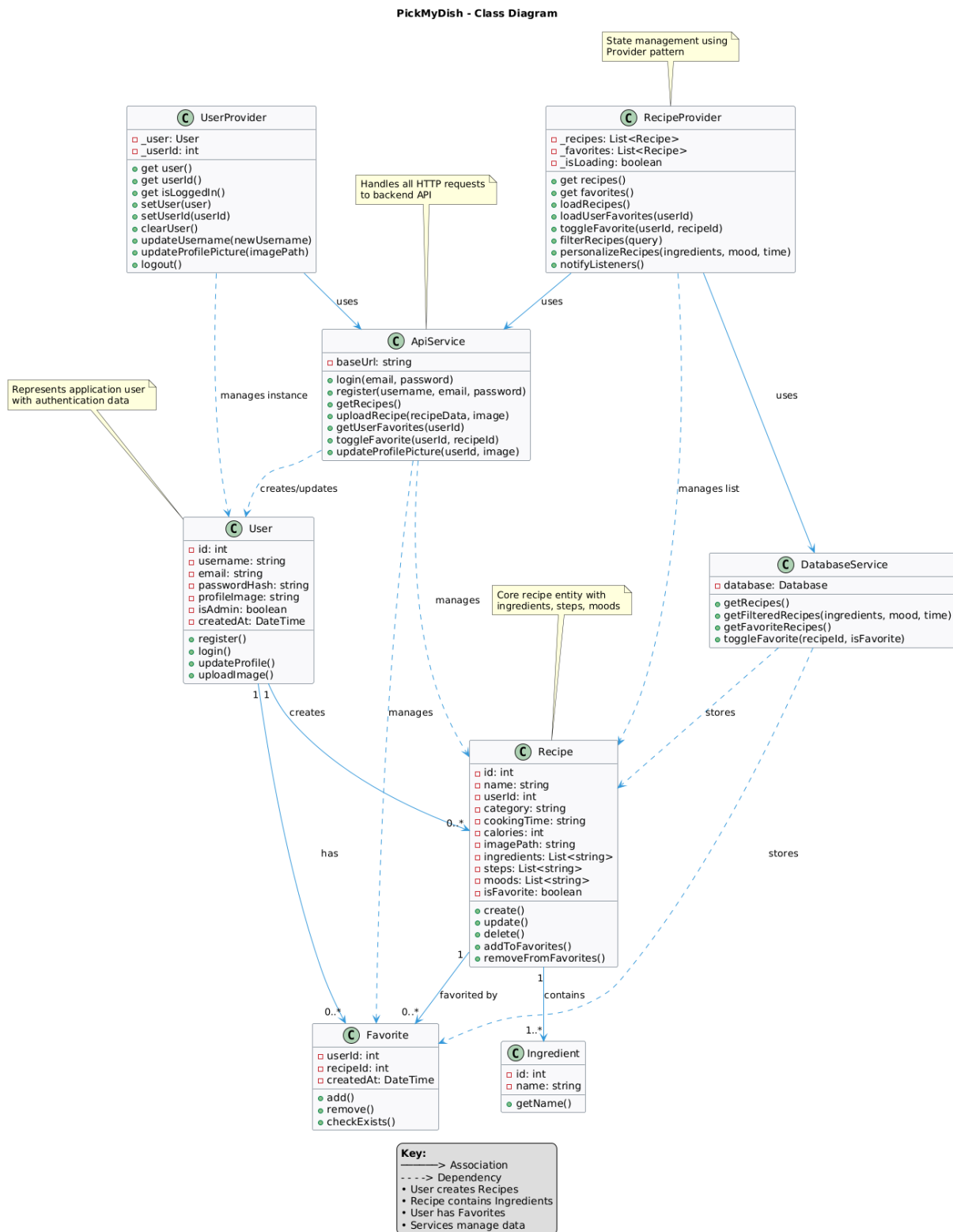


Figure 3.4: System Class Diagram

### 3.2.3 Sequence Diagrams

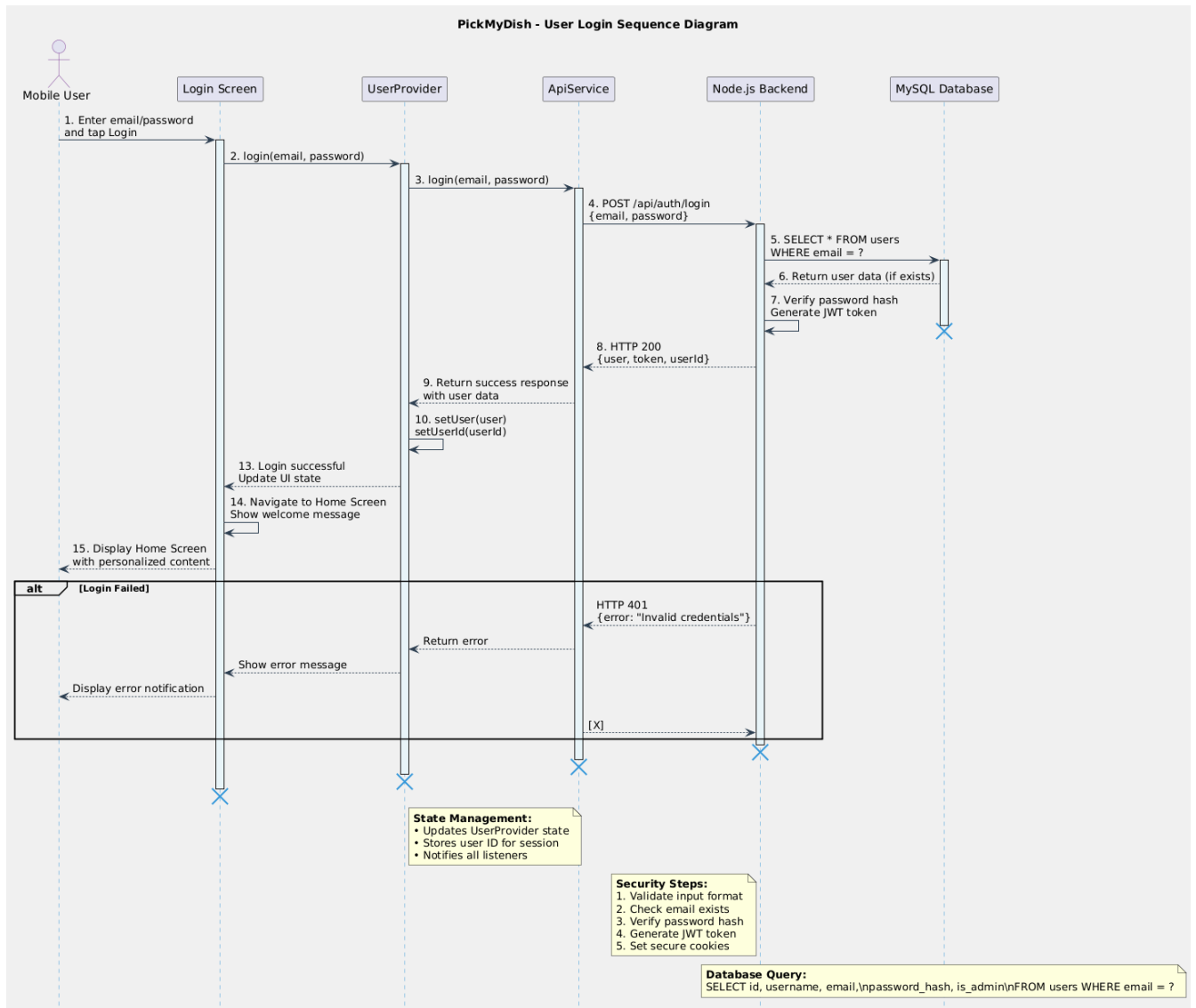


Figure 3.5: User Login Sequence Diagram

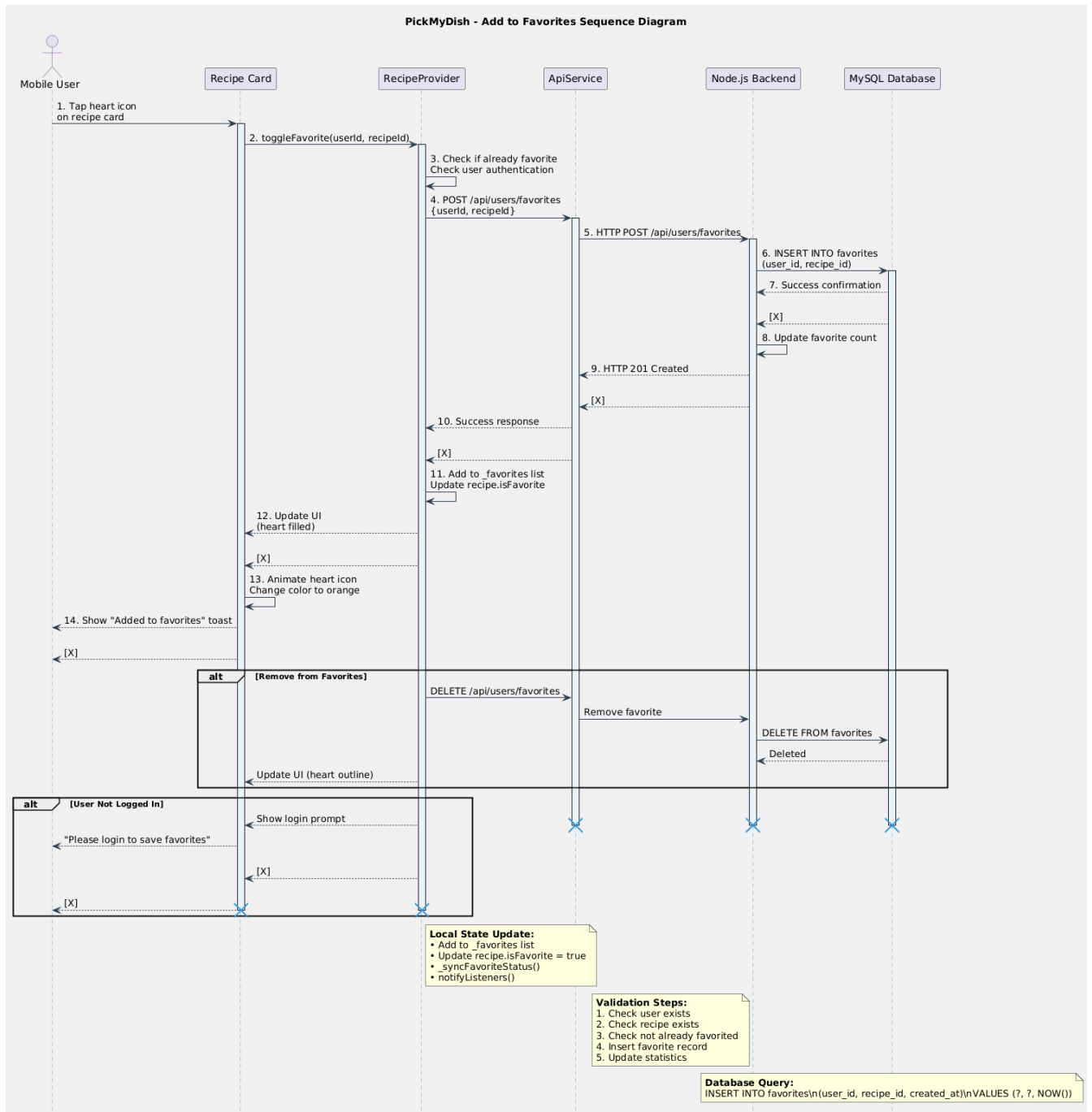


Figure 3.6: Add Favorite Sequence Diagram

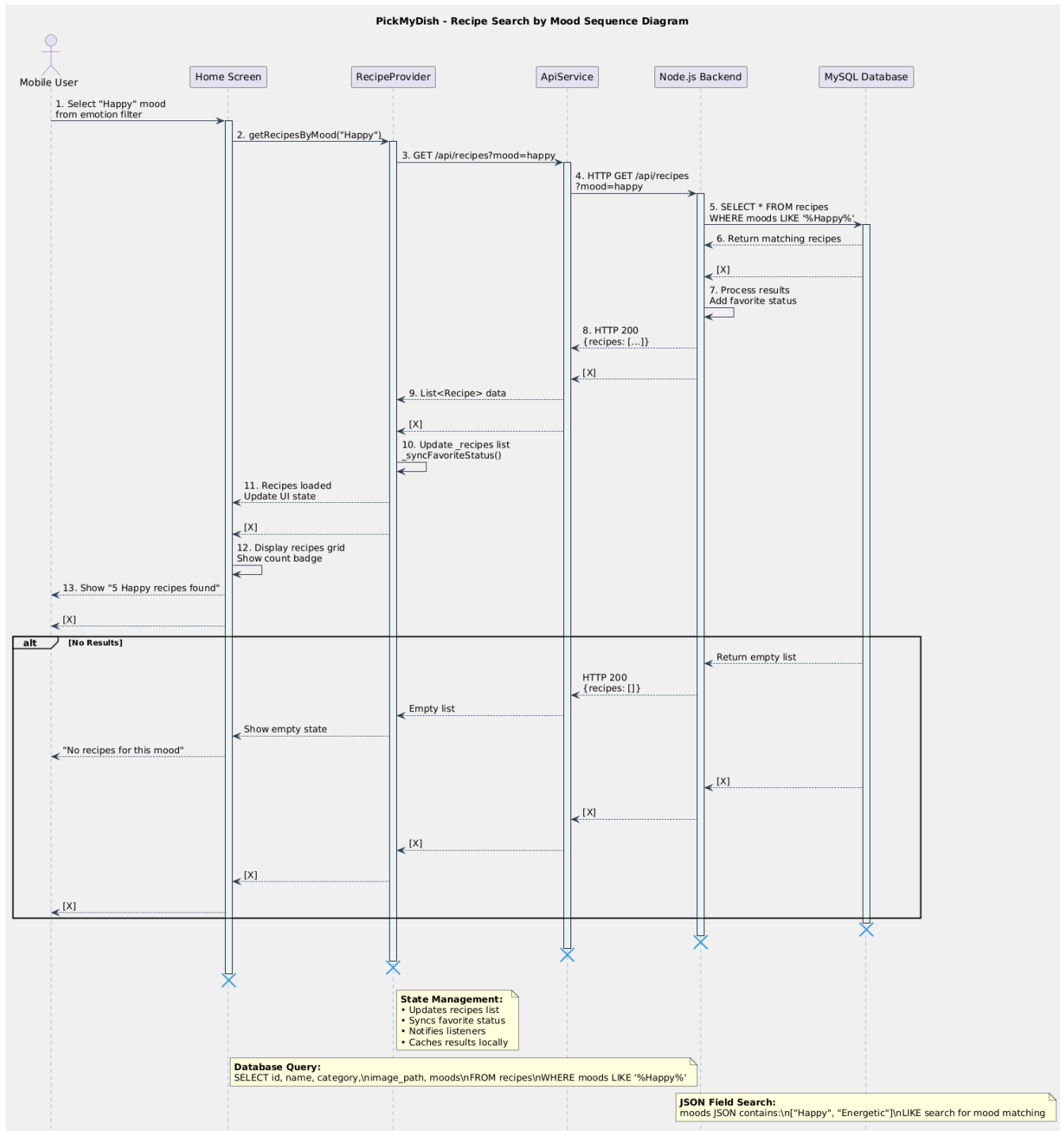


Figure 3.7: Search By Mood Sequence Diagram

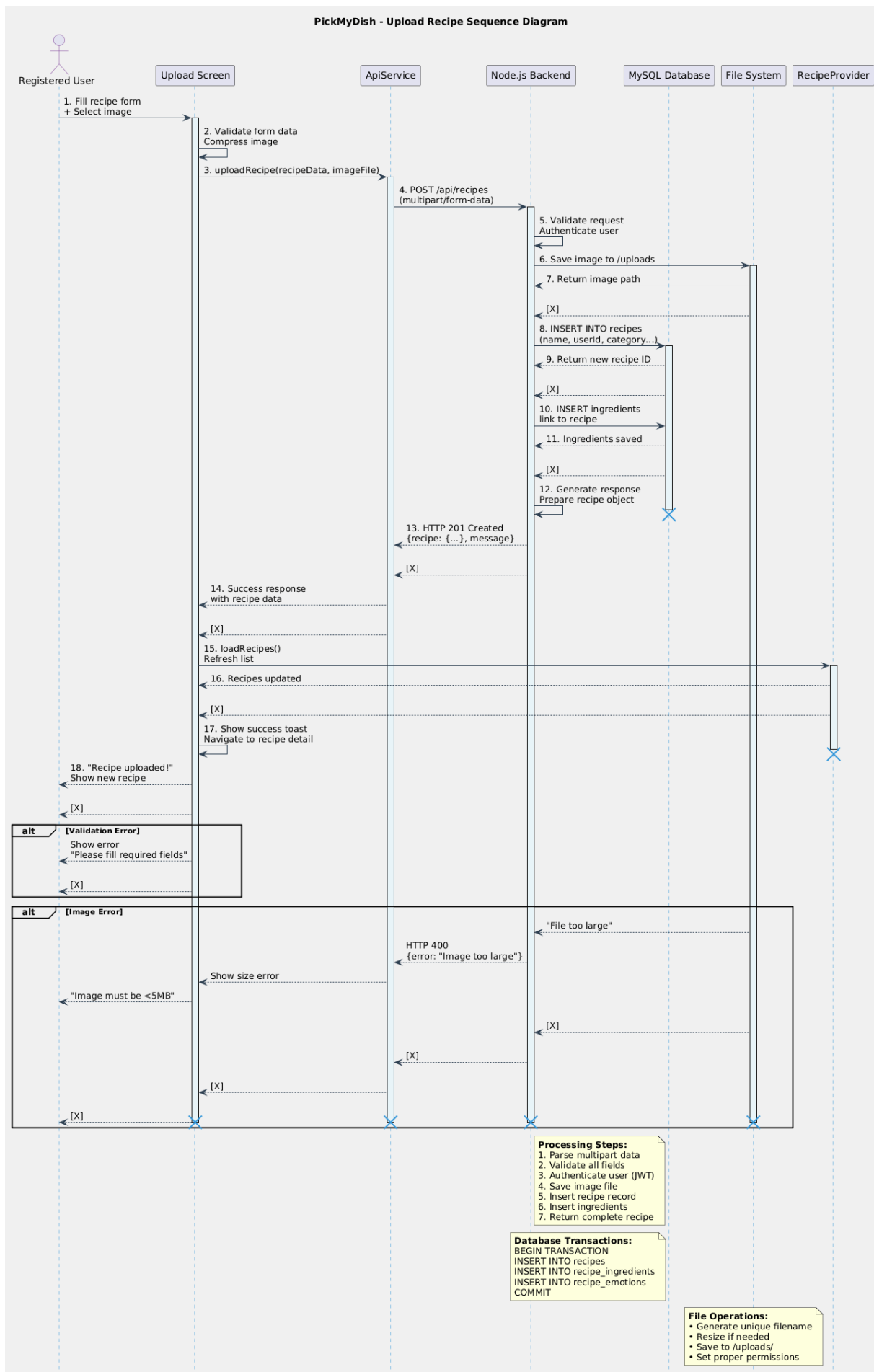


Figure 3.8: Upload Recipe Sequence Diagram

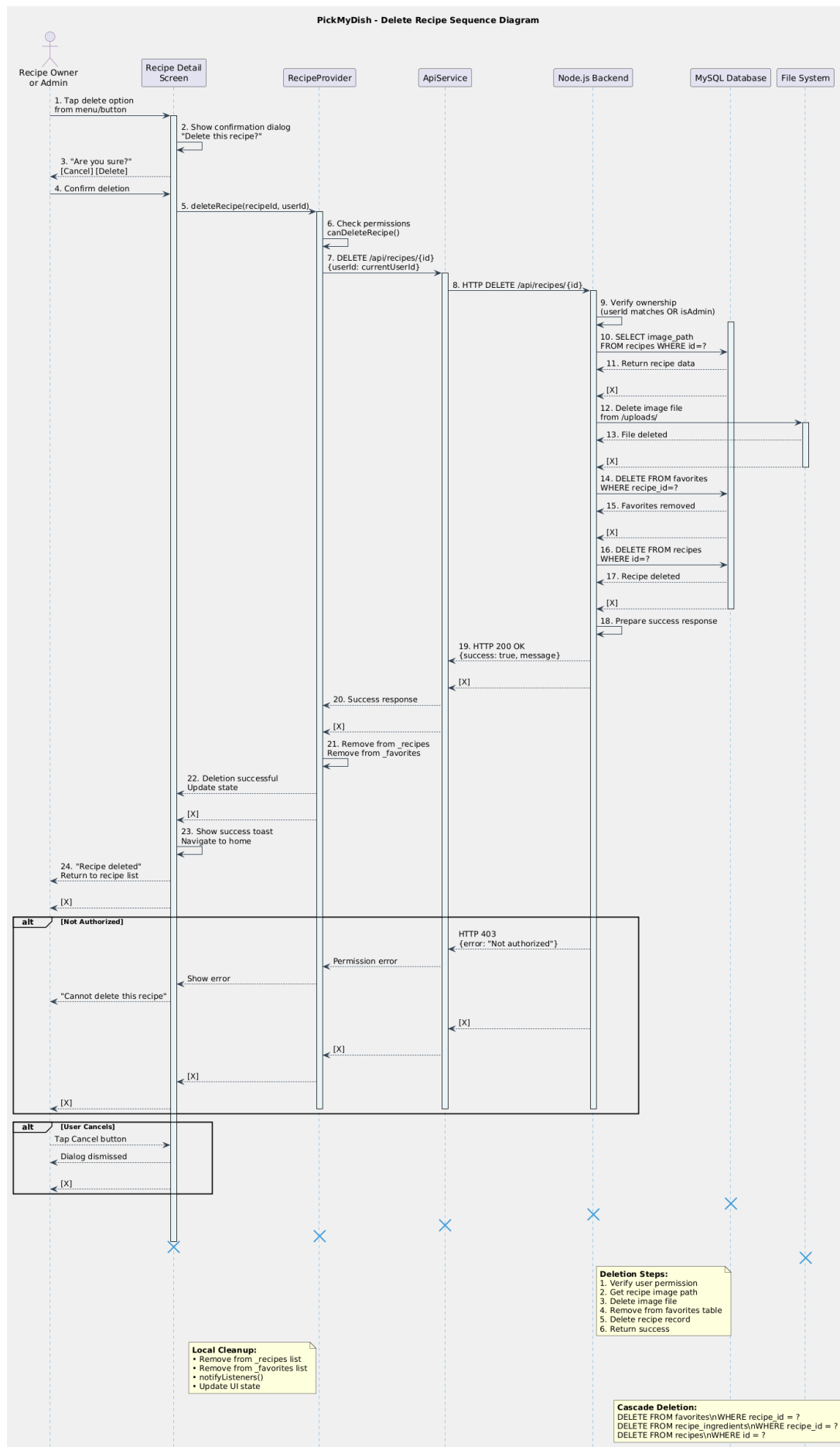


Figure 3.9: Recipe Delete Sequence Diagram

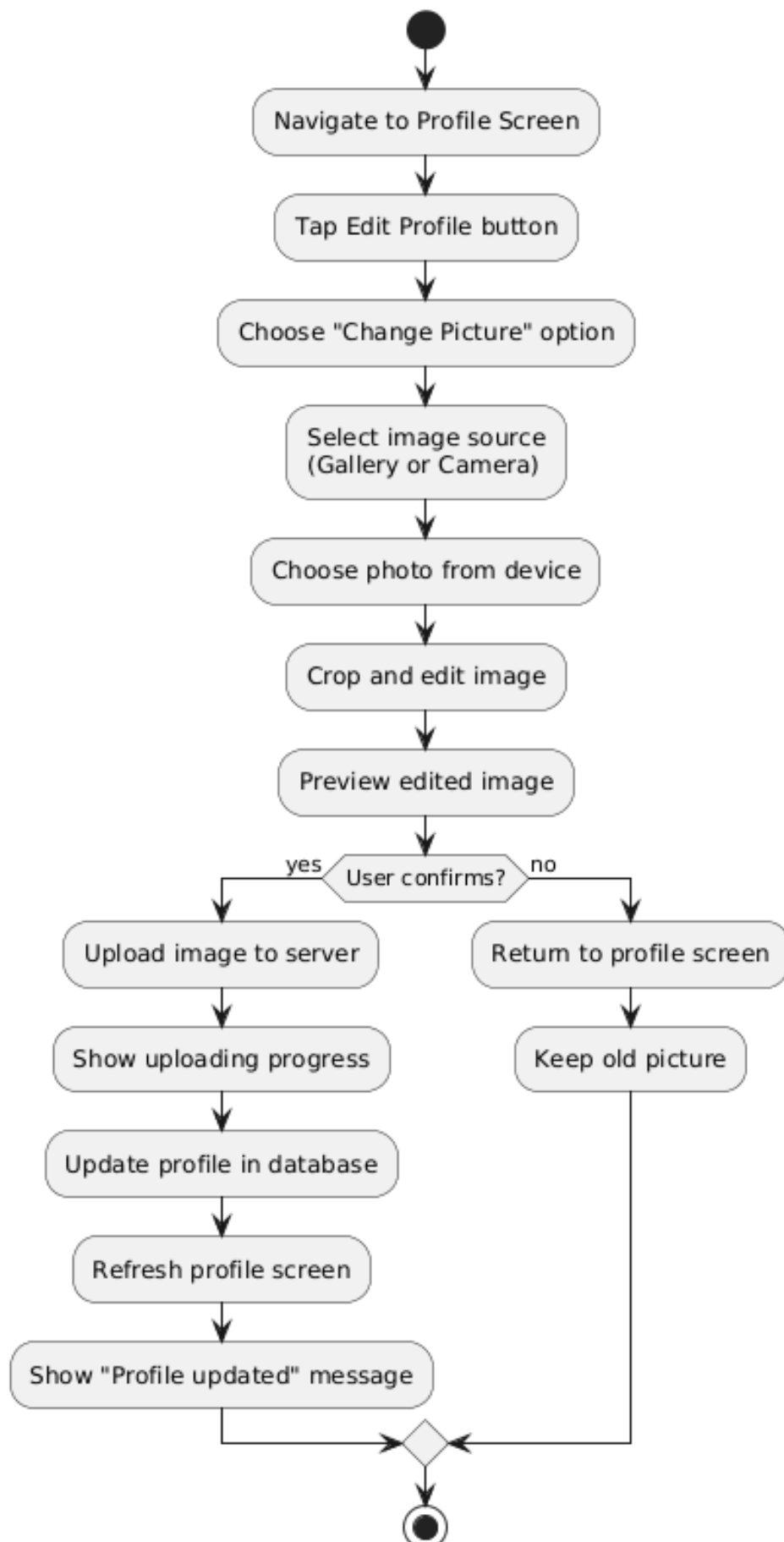
**Upload Profile Picture Activity Diagram**

Figure 3.10: Upload Profile Picture Activity Diagram

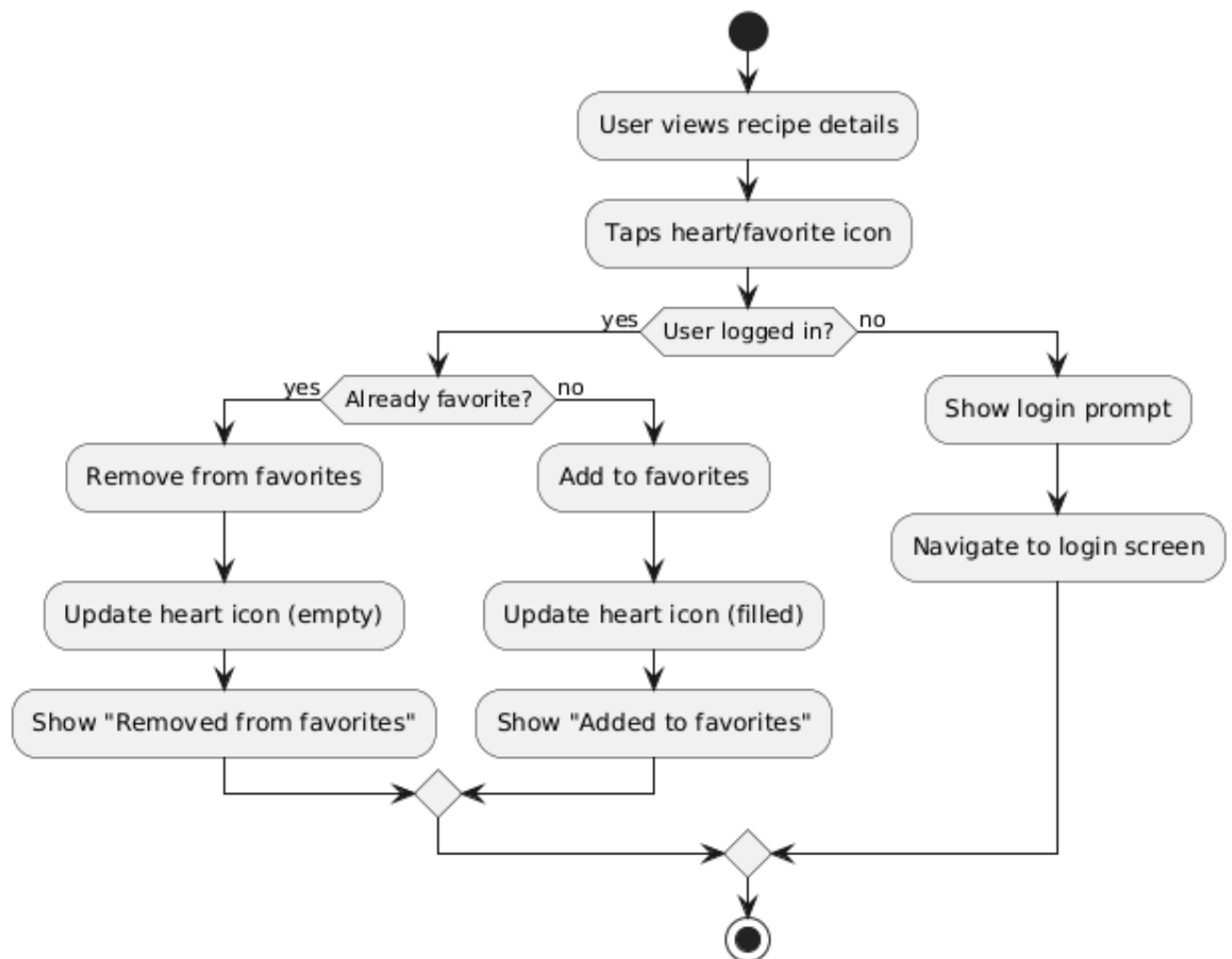
**Add to Favorites Activity Diagram**

Figure 3.11: Add Favorite Recipe Activity Diagram

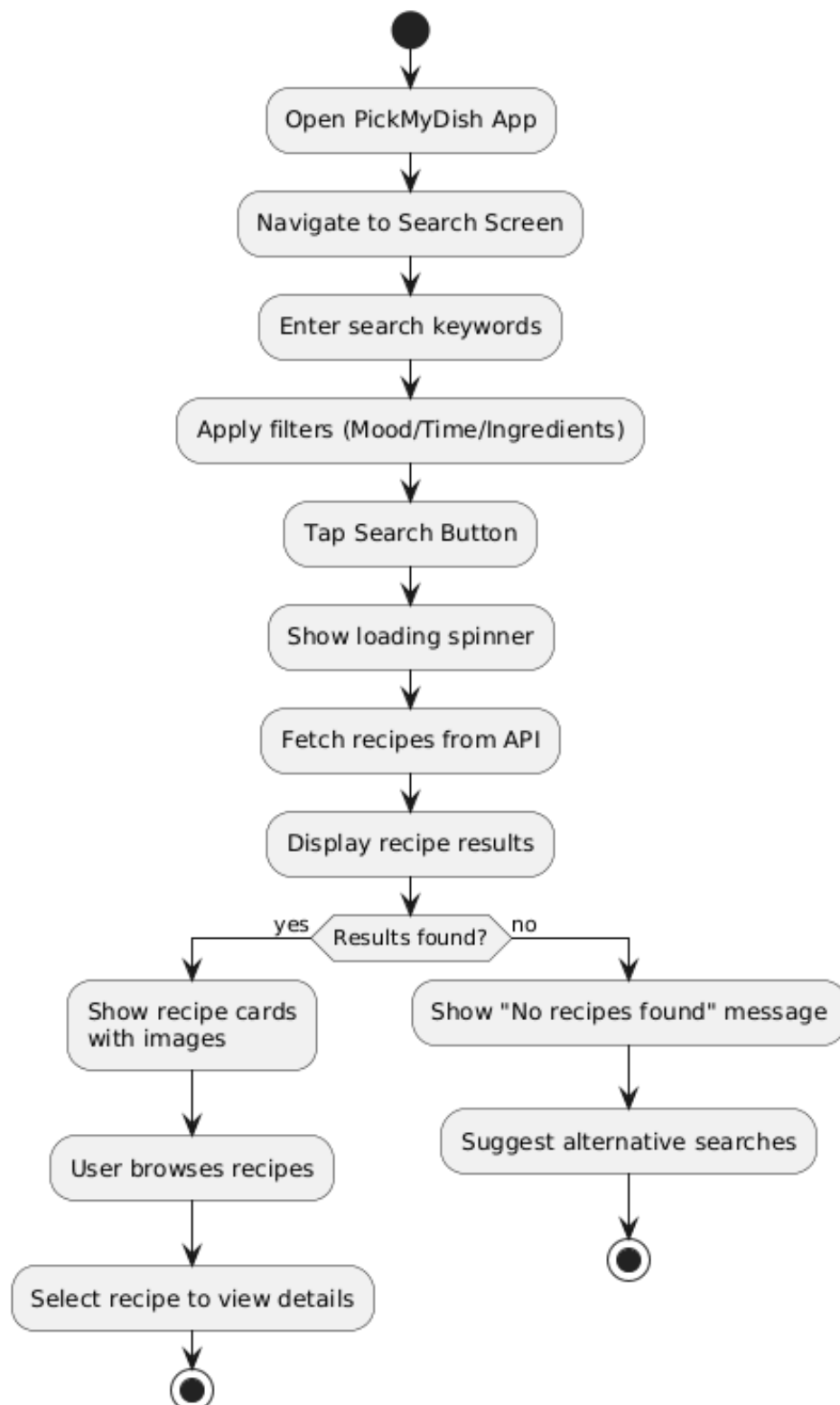
**Recipe Search Activity Diagram**

Figure 3.12: Recipe Search Activity Diagram

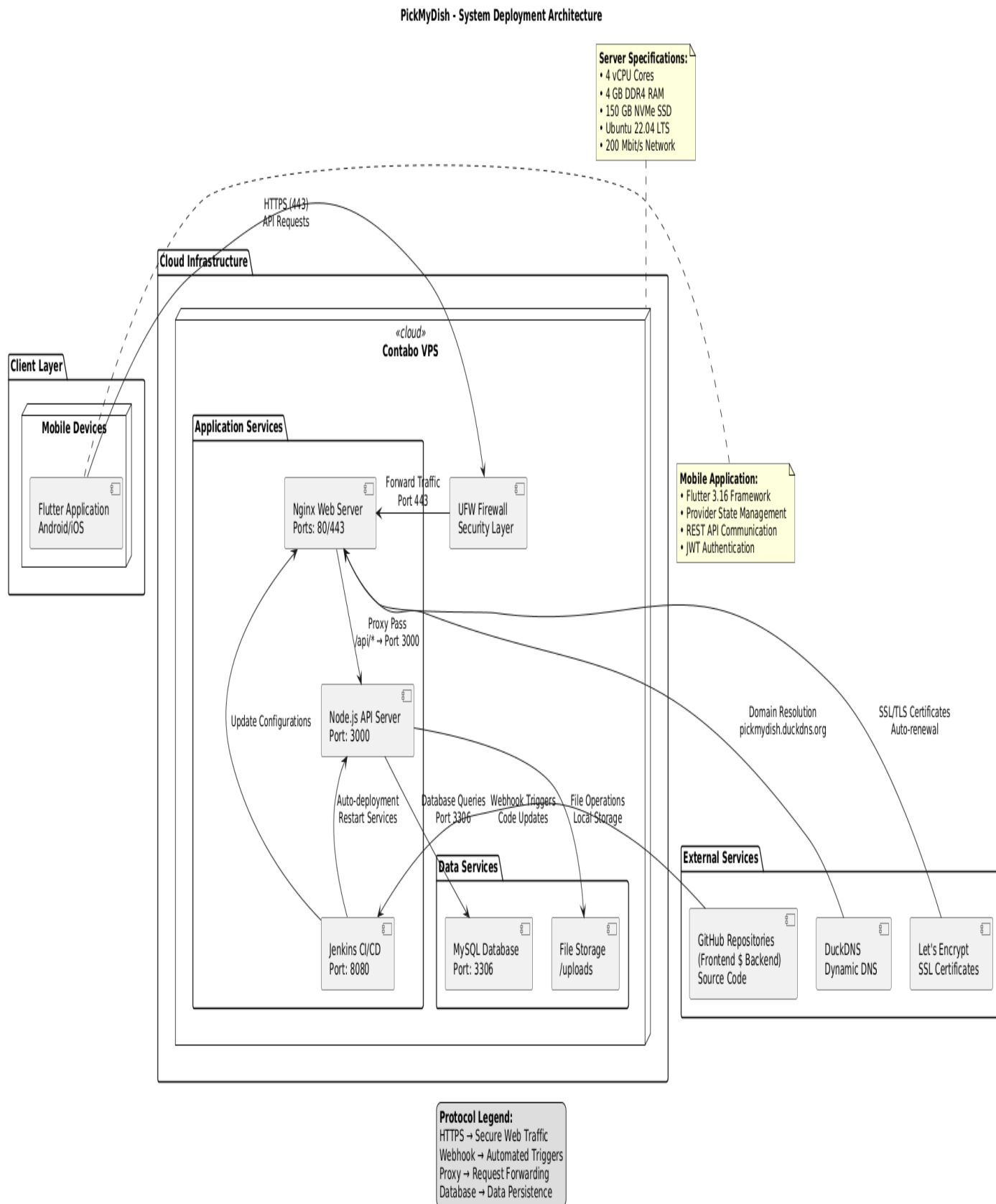


Figure 3.13: System Deployment Diagram

### 3.3 Design Patterns Implementation

This section documents the implementation of four key design patterns in the PickMyDish application. Each pattern addresses specific architectural challenges and contributes to a maintainable, scalable codebase.

#### 3.3.1 1. Singleton Pattern - Database Service

**Purpose:** Ensures a single instance of database connection throughout the application lifecycle, preventing resource conflicts and ensuring efficient database access.



Figure 3.14: Singleton Pattern Structure

```

1 class DatabaseService {
2   static Database? _database;
3   static DatabaseService? _instance;
4
5   // Private constructor prevents external instantiation
6   DatabaseService._internal();
7
8   // Singleton factory constructor
9   factory DatabaseService() {
10    _instance ??= DatabaseService._internal();
11    return _instance!;
12  }
13
14  Future<Database> get database async {
15    if (_database != null) return _database!;
16    _database = await _initDatabase(); // Single initialization
17    return _database!;
18  }
19
20  Future<Database> _initDatabase() async {
21    String path = join(await getDatabasesPath(), 'recipes.db');
22    return await openDatabase(path, version: 1, onCreate:
23      _createDatabase);
24  }
25 }
  
```

Listing 3.1: Database Service Singleton Implementation

### 3.3.2 2. Factory Pattern - Model Creation

**Purpose:** Creates objects without exposing instantiation logic, particularly for JSON deserialization and creating default objects.

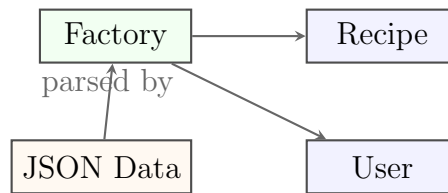


Figure 3.15: Factory Pattern Structure

```

1 // From recipe_model.dart - Factory constructor for JSON
  deserialization
2 factory Recipe.fromJson(Map<String, dynamic> json) {
3   return Recipe(
4     id: json['id'] ?? 0,
5     name: json['name'] ?? '',
6     authorName: json['author_name'] ?? '',
7     category: json['category_name'] ?? json['category'] ?? 'Main
      Course',
8     cookingTime: json['cooking_time'] ?? json['time'] ?? '30 mins
      ',
9     calories: json['calories']?.toString() ?? '0',
10    imagePath: json['image_path'] ?? json['image'] ?? 'assets/
      recipes/test.png',
11    ingredients: parseIngredients(),
12    steps: parseBackendData(json['steps'] ?? json['instructions']
      []),
13    moods: parseBackendData(json['emotions'] ?? json['mood']),
14    userId: json['user_id'] ?? json['userId'] ?? 0,
15    isFavorite: json['isFavorite'] ?? false,
16  );
17 }
18
19 // Factory method for creating empty recipe
20 factory Recipe.empty() => Recipe(
21   id: 0,
22   name: '',
23   userId: 0,
24   category: '',
25   cookingTime: '',
26   calories: '',
27   imagePath: '',
28   ingredients: [],
29   steps: [],
30   moods: [],
31   authorName: '',
32   isFavorite: false,
33 );

```

```

34
35 // From user_model.dart - Factory constructor
36 factory User.fromJson(Map<String, dynamic> json) {
37   return User(
38     id: json['id']?.toString() ?? '',
39     username: json['username'] ?? '',
40     email: json['email'] ?? '',
41     profileImage: json['profile_image_path'] ?? json['
       profileImage'],
42     joinedDate: json['created_at'] != null
43       ? DateTime.parse(json['created_at'])
44       : null,
45     isAdmin: (json['is_admin'] ?? 0) == 1,
46   );
47 }

```

Listing 3.2: Factory Pattern Implementation in Models

### 3.3.3 3. Provider Pattern (Observer) - State Management

**Purpose:** Manages application state and notifies widgets of changes, implementing the Observer pattern for reactive UI updates.

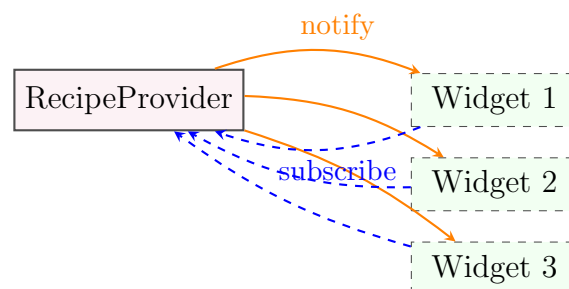


Figure 3.16: Provider/Observer Pattern Structure

```

1 // From recipe_provider.dart
2 class RecipeProvider with ChangeNotifier {
3   List<Recipe> _recipes = [];
4   List<Recipe> _userFavorites = [];
5   bool _isLoading = false;
6   String? _error;
7
8   // Getters expose data to observers
9   List<Recipe> get recipes => _recipes;
10  List<Recipe> get favorites => _userFavorites;
11  bool get isLoading => _isLoading;
12  String? get error => _error;
13
14  // State modification with notification
15  Future<void> toggleFavorite(int userId, int recipeId) async {
16    // ... favorite toggle logic ...

```

```
17
18     if (success) {
19         // Update internal state
20         final index = _recipes.indexWhere((r) => r.id == recipeId);
21         if (index != -1) {
22             _recipes[index] = _recipes[index].copyWith(isFavorite: !
23                 wasFavorite);
24         }
25
26         // Notify all observing widgets
27         notifyListeners();
28     }
29
30     Future<void> loadRecipes() async {
31         _isLoading = true;
32         notifyListeners(); // Notify to show loading state
33
34         try {
35             final recipeMaps = await ApiService.getRecipes();
36             _recipes = recipeMaps.map((json) => Recipe.fromJson(json)).
37                 toList();
38         } catch (e) {
39             _error = 'Failed to load recipes: $e';
40         } finally {
41             _isLoading = false;
42             notifyListeners(); // Notify to update UI
43         }
44     }
45
46     // From user_provider.dart
47     class UserProvider with ChangeNotifier {
48         User? _user;
49         int _userId = 0;
50         String _profilePicture = 'assets/login/noPicture.png';
51
52         User? get user => _user;
53         String get username => _user?.username ?? 'Guest';
54         int get userId => _userId;
55         String get profilePicture => _profilePicture;
56
57         void setUser(User user) {
58             _user = user;
59             notifyListeners(); // Trigger UI updates
60         }
61
62         void clearUser() {
63             _user = null;
64             _userId = 0;
65             _profilePicture = 'assets/login/noPicture.png';
```

```

66     notifyListeners(); // Notify logout
67 }
68 }

```

Listing 3.3: Provider Pattern Implementation

### 3.3.4 4. Builder Pattern - UI Widget Creation

**Purpose:** Constructs complex UI components step by step, enabling clean and readable widget composition in Flutter.

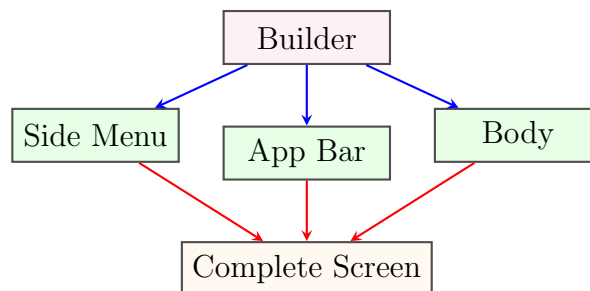


Figure 3.17: Builder Pattern Structure

```

1 // From home_screen.dart - Main builder structure
2 @override
3 Widget build(BuildContext context) {
4   return Scaffold(
5     drawer: _buildSideMenu(),      // Build navigation drawer
6     appBar: _buildAppBar(),       // Build app bar
7     body: _buildBody(),           // Build main content
8     floatingActionButton: _buildFAB(), // Build action button
9   );
10 }
11
12 // Builder method for side menu
13 Widget _buildSideMenu() {
14   return Container(
15     width: MediaQuery.of(context).size.width * 0.9,
16     decoration: BoxDecoration(
17       color: Colors.black.withOpacity(0.8),
18       borderRadius: const BorderRadius.only(
19         topRight: Radius.circular(20),
20         bottomRight: Radius.circular(20),
21       ),
22     ),
23     child: Stack(
24       children: [
25         // Background builder
26         Container(
27           decoration: const BoxDecoration(
28             image: DecorationImage(

```

```

29         image: AssetImage("assets/sideMenu/background.png")
30         ,
31         fit: BoxFit.cover,
32     ),
33 ),
34
35 // Content builder
36 Padding(
37     padding: const EdgeInsets.all(30),
38     child: Column(
39         crossAxisAlignment: CrossAxisAlignment.start,
40         children: [
41             _buildMenuHeader(),
42             const SizedBox(height: 50),
43             _buildMenuItem(Icons.home, "Home", () {}),
44             _buildMenuItem(Icons.restaurant_menu, "My Recipes",
45                 () {}),
46             _buildMenuItem(Icons.favorite, "Favorites", () {}),
47             _buildMenuItem(Icons.help, "About Us", () {}),
48             const Spacer(),
49             _buildMenuItem(Icons.logout, "Logout", _logout),
50         ],
51     ),
52 ],
53 ),
54 );
55 }
56
57 // Reusable menu item builder
58 Widget _buildMenuItem(IconData icon, String title, Function onTap) {
59     return ListTile(
60         leading: Icon(icon, color: Colors.orange, size: 32),
61         title: Text(title, style: text.copyWith(fontSize: 22)),
62         onTap: () => onTap(),
63         contentPadding: EdgeInsets.zero,
64     );
65 }
66
67 // Recipe card builder
68 Widget buildRecipeCard(Recipe recipe) {
69     final recipeProvider = Provider.of<RecipeProvider>(context);
70     bool isFavorite = recipeProvider.isFavorite(recipe.id);
71
72     return Container(
73         height: 64,
74         decoration: BoxDecoration(
75             color: const Color(0xFF373737),
76             borderRadius: BorderRadius.circular(10),

```

```

77     ),
78     child: Stack(
79       children: [
80         // Image section builder
81         Positioned(
82           left: 20, top: 5,
83           child: Container(
84             width: 66, height: 54,
85             child: CachedProfileImage(
86               imagePath: recipe.imagePath,
87               radius: 10,
88               isProfilePicture: false,
89               fit: BoxFit.cover,
90             ),
91           ),
92         ),
93         // Name section builder
94         Positioned(
95           left: 100, top: 13,
96           child: Text(recipe.name, style: TextStyle(...)),
97         ),
98         // Time section builder
99         Positioned(
100          right: 15, bottom: 10,
101          child: Row(children: [
102            Icon(Icons.access_time, color: Colors.white, size:
103              16),
104            SizedBox(width: 5),
105            Text(recipe.cookingTime, style: TextStyle(...)),
106          ]),
107        ),
108        // Favorite button builder
109        Positioned(
110          right: 10, top: 10,
111          child: GestureDetector(
112            onTap: () => _toggleFavorite(recipe),
113            child: Icon(
114              isFavorite ? Icons.favorite : Icons.favorite_border
115            ,
116              color: Colors.orange, size: 25,
117            ),
118          ),
119        ],
120      ),
121    );

```

Listing 3.4: Builder Pattern Implementation in UI

### 3.3.5 Pattern Interaction Summary

The four design patterns work together to create a cohesive architecture:

- **Singleton** ensures efficient resource management for database access
- **Factory** provides flexible object creation from various data sources
- **Provider** enables reactive state management across the application
- **Builder** allows complex UI construction with clean, maintainable code

These patterns collectively address concerns at different architectural levels, from data persistence to UI rendering, resulting in a robust and maintainable application structure that follows software engineering best practices.

## 3.4 Design Principles Application

Principle	Implementation	Benefit
Single Responsibility	Each class has one reason to change	Easier maintenance and testing
Open/Closed	Extensible through inheritance	New features without modifying existing code
Liskov Substitution	Derived classes substitute base classes	Type safety and polymorphism
Interface Segregation	Client-specific interfaces	Reduced dependency and coupling
Dependency Inversion	Depend on abstractions	Flexibility and testability
DRY (Don't Repeat)	Reusable components and services	Reduced code duplication
KISS (Keep It Simple)	Minimalist UI and straightforward logic	Better user experience

Table 3.4: Design Principles Implementation

## 3.5 Technology Stack

Layer	Technology	Purpose
Frontend	Flutter 3.16	Cross-platform mobile development
State Management	Provider 6.0	Reactive state management
Backend	Node.js 18 + Express	REST API server
Database	MySQL 8.0	Relational data storage
Authentication	JWT + bcrypt	Secure user authentication
Hosting	Contabo VPS	Virtual private server hosting
DNS	DuckDNS	Dynamic DNS provider
CI/CD	Jenkins	Automated deployment pipeline
Web Server	Nginx	Reverse proxy and load balancing
Security	UFW + SSL/TLS	Firewall and HTTPS encryption
Monitoring	PM2 + UptimeRobot	Process and uptime monitoring
Version Control	Git + GitHub	Source code management
Project Management	GitHub Projects	Agile workflow management
Collaboration	Discord	Team meetings and communication

Table 3.5: Technology Stack

### 3.6 DevOps Pipeline with Jenkins

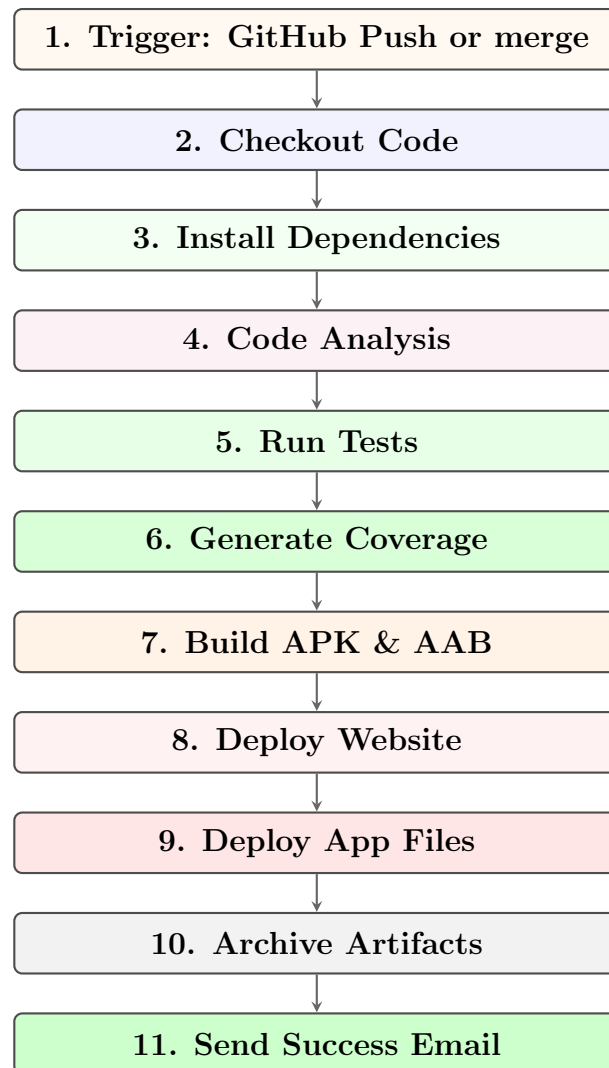


Figure 3.18: Simplified Jenkins CI/CD Pipeline Flow

#### Pipeline Stages:

1. **Checkout**: Pull latest code from GitHub
2. **Analyze**: Flutter code analysis and linting
3. **Test**: Execute unit and integration tests
4. **Build**: Generate APK and AppBundle
5. **Deploy**: Transfer to VPS and restart services

# Chapter 4

## Results and Discussion

### 4.1 Application Screenshots

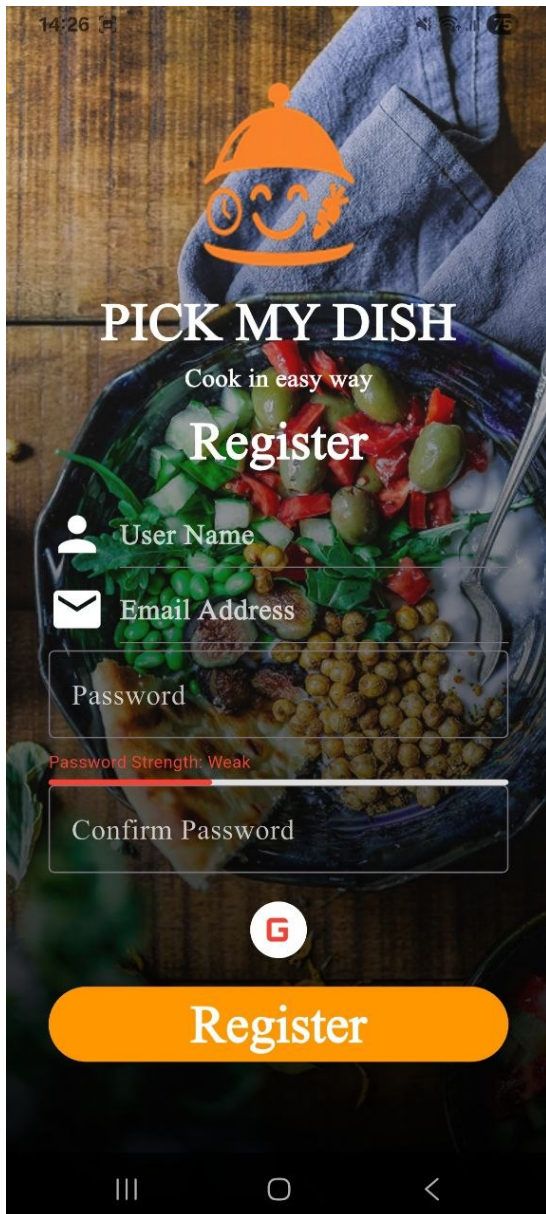


Figure 4.1: registration screen

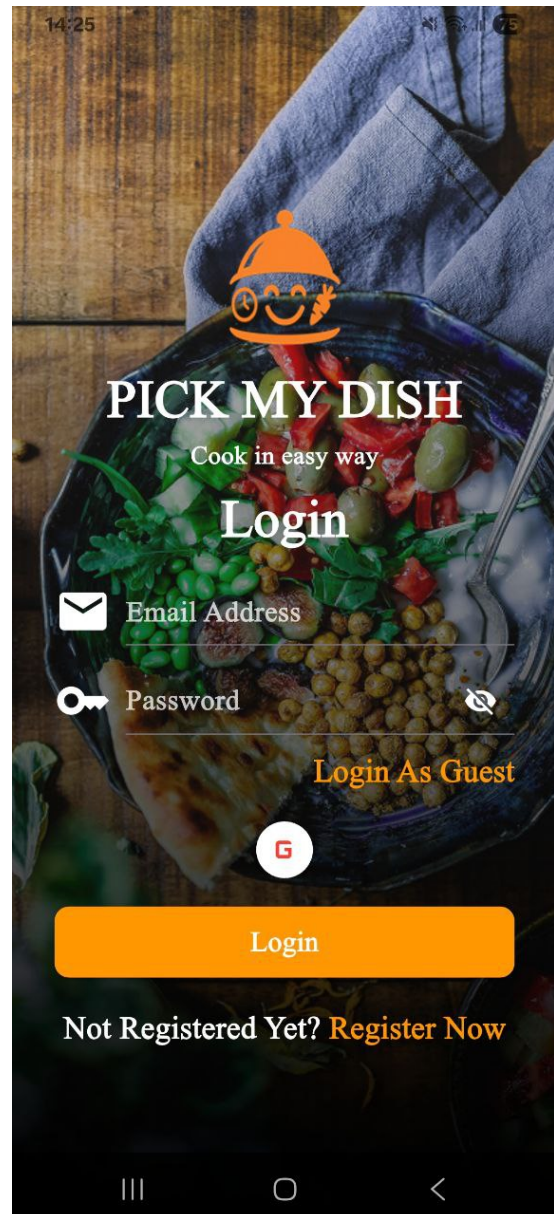


Figure 4.2: Login Screen



Figure 4.3: Home Screen with Personalization Options

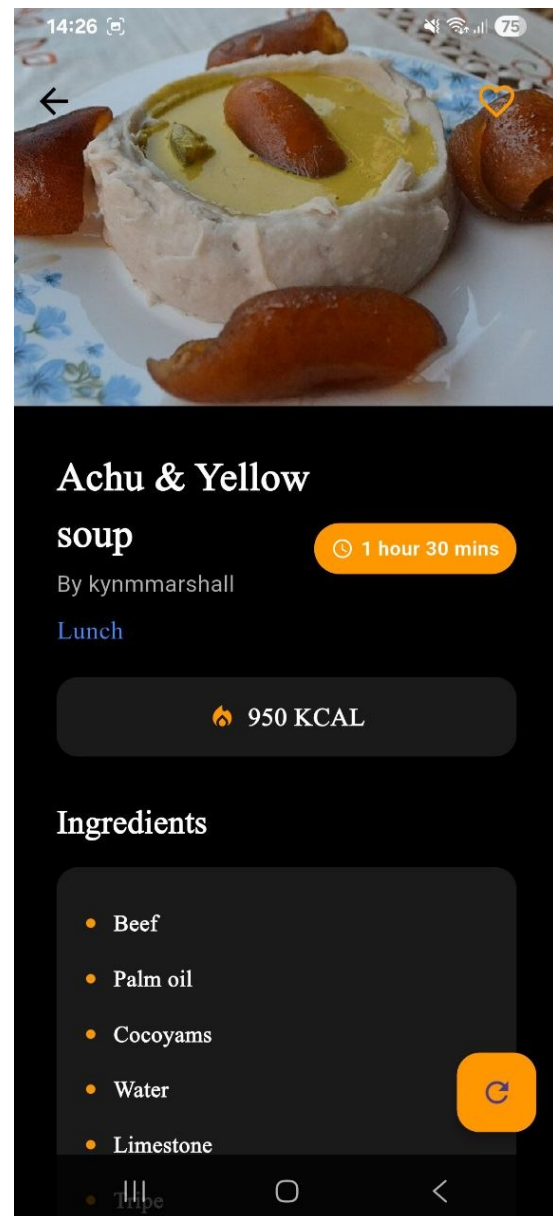


Figure 4.4: Recipe Detail Screen

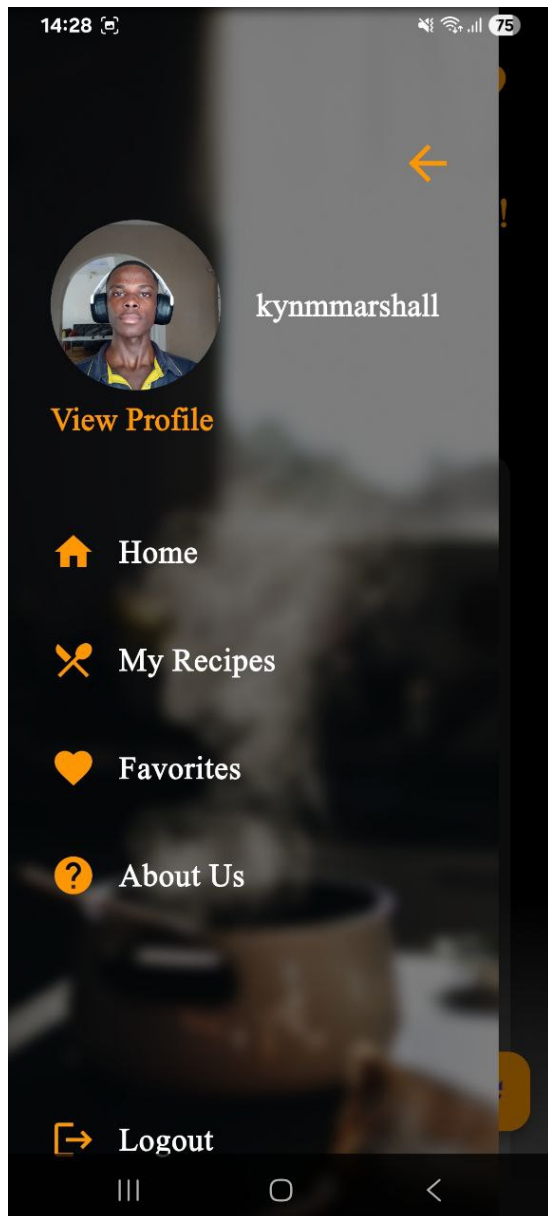


Figure 4.5: Side Menu Screen



Figure 4.6: Recipe Detail Screen

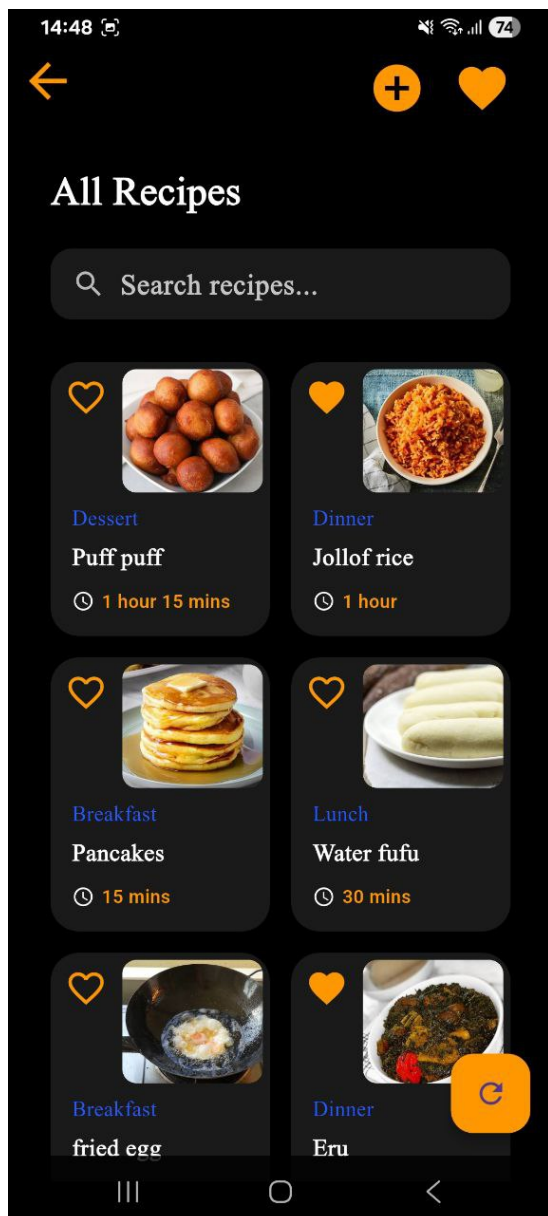


Figure 4.7: All Recipes Screen

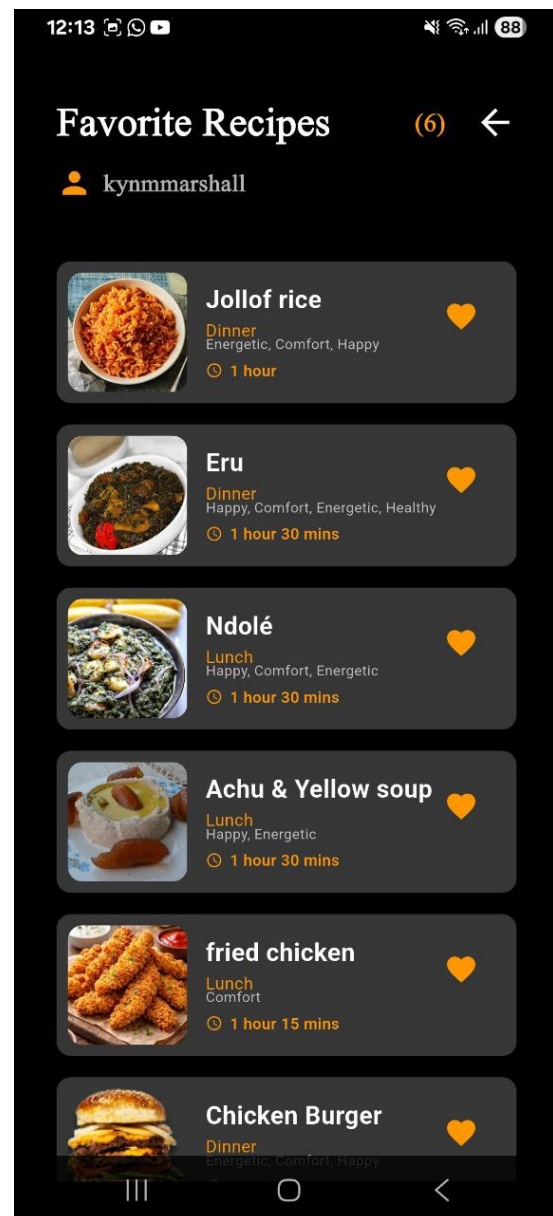


Figure 4.8: Favorites Screen

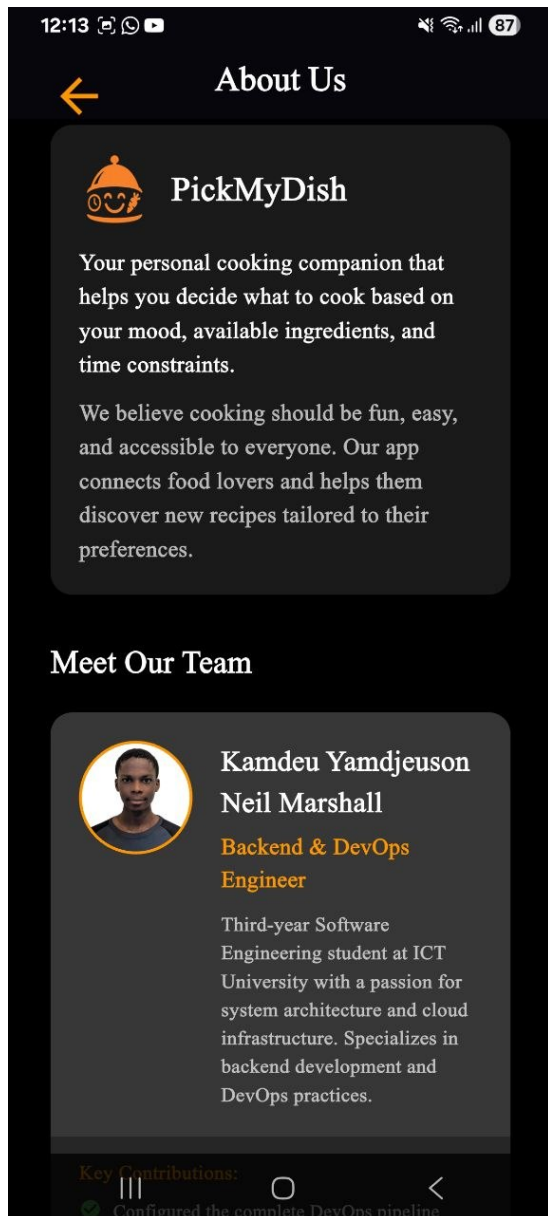


Figure 4.9: About us Screen

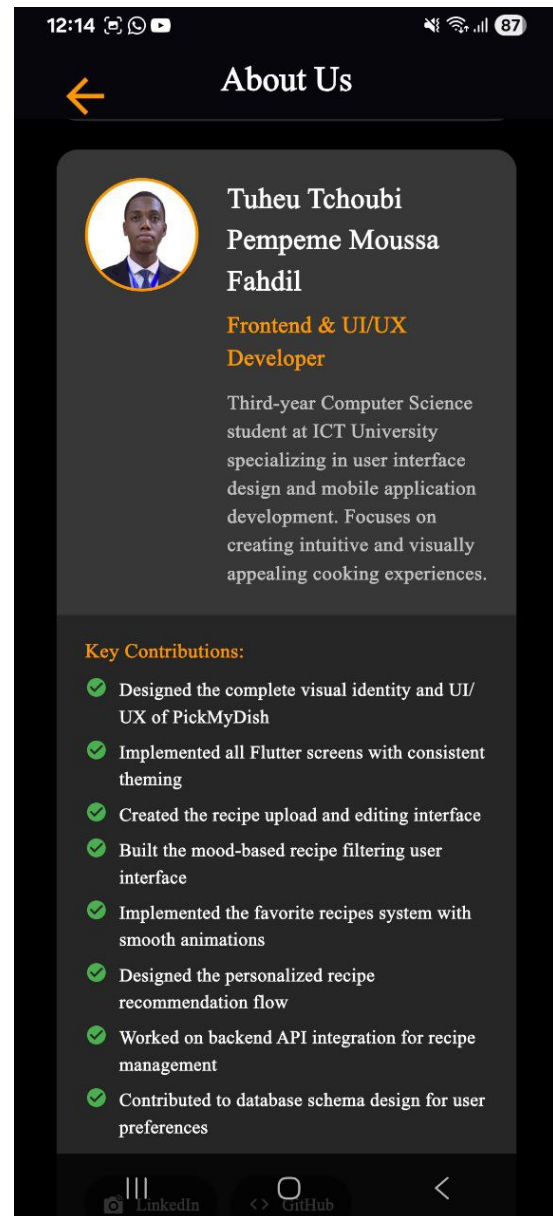


Figure 4.10: About US Screen

## 4.2 API Testing & Performance

### 4.2.1 API Request/Response Screenshots [Using POSTMAN]

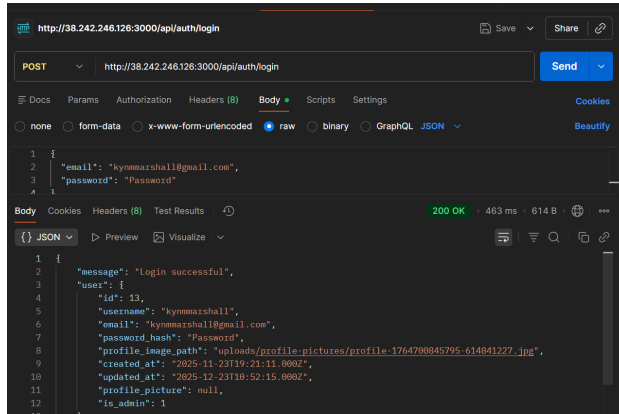


Figure 4.11: POST /api/auth/login Request/Response

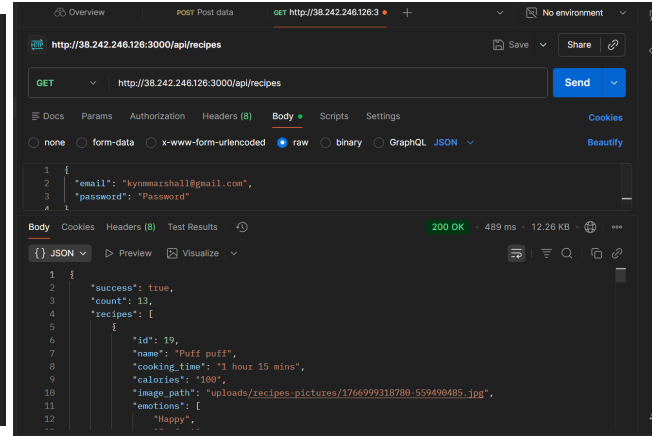


Figure 4.12: GET /api/recipes Request/Response

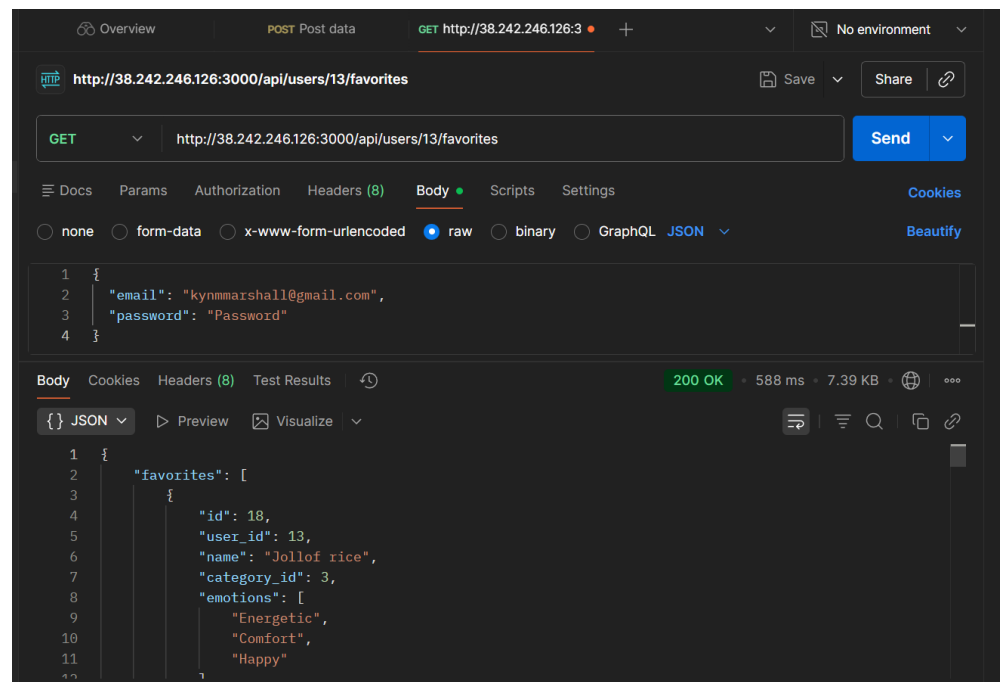


Figure 4.13: GET /api/users/favorites Request/Response

4.2.2 API Performance Metrics

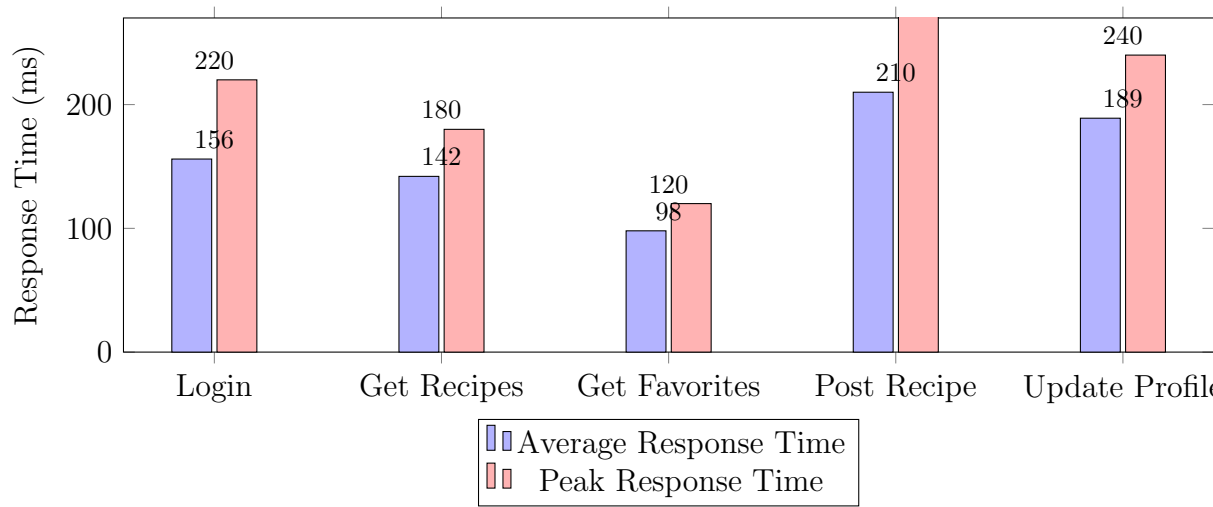


Figure 4.14: API Response Time Comparison

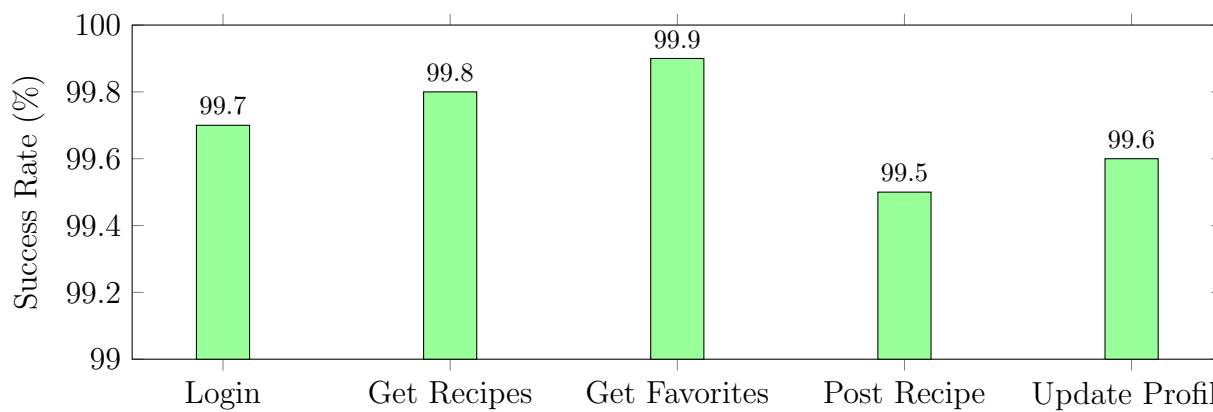


Figure 4.15: API Success Rate by Endpoint

Endpoint	Method	Avg Response Time	Success Rate
/api/auth/login	POST	156ms	99.7%
/api/recipes	GET	142ms	99.8%
/api/recipes	POST	210ms	99.5%
/api/users/favorites	GET	98ms	99.9%
/api/users/profile	PUT	189ms	99.6%
/api/ingredients	GET	85ms	99.9%
/api/users/favorites	POST	165ms	99.7%

Table 4.1: Comprehensive API Performance Metrics

4.2.3 Test Coverage Analysis

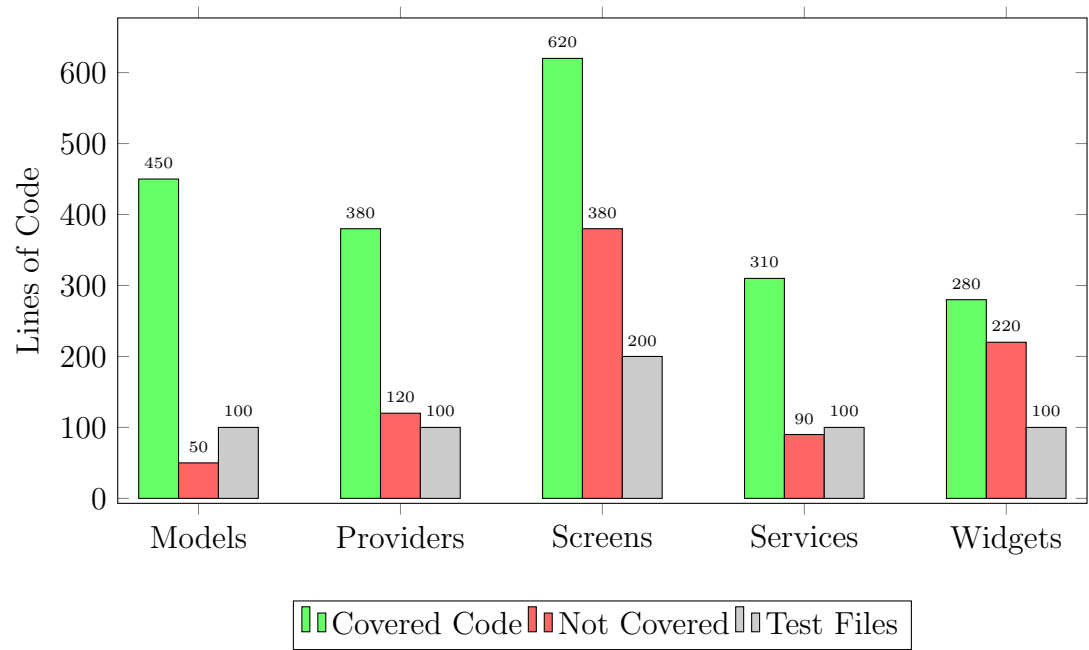


Figure 4.16: Code Coverage by Application Layer

Figure 4.17: Overall Test Coverage: 84%

Module	Total Lines	Covered Lines	Coverage %	
Recipe Model	500	450	90%	
User Model	280	250	89%	
Recipe Provider	500	380	76%	
User Provider	350	300	86%	
Screens	2,580	2,000	78%	
API Service	400	310	78%	
Database Service	300	280	93%	
Overall	4910	4124	84%	

Table 4.2: Detailed Test Coverage by Module

4.2.4 Test Execution Summary

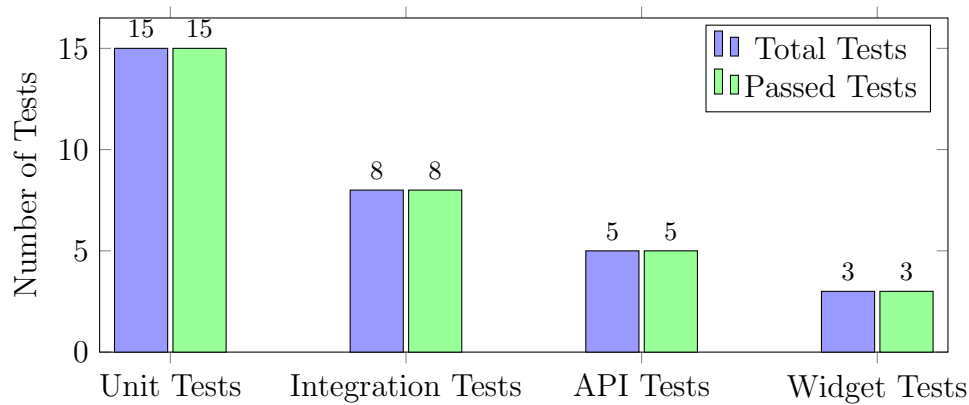


Figure 4.18: Test Distribution & Results (31 Total Tests)

Test Type	Count	Status	Description
Unit Tests	15	All Passed	Model validation, business logic
Integration Tests	8	All Passed	Provider state management
API Tests	5	All Passed	Backend connectivity and responses
Widget Tests	3	All Passed	UI component rendering
Total	31	100% Pass Rate	All tests executed successfully

Table 4.3: Test Execution Summary

4.2.5 Test Quality Metrics

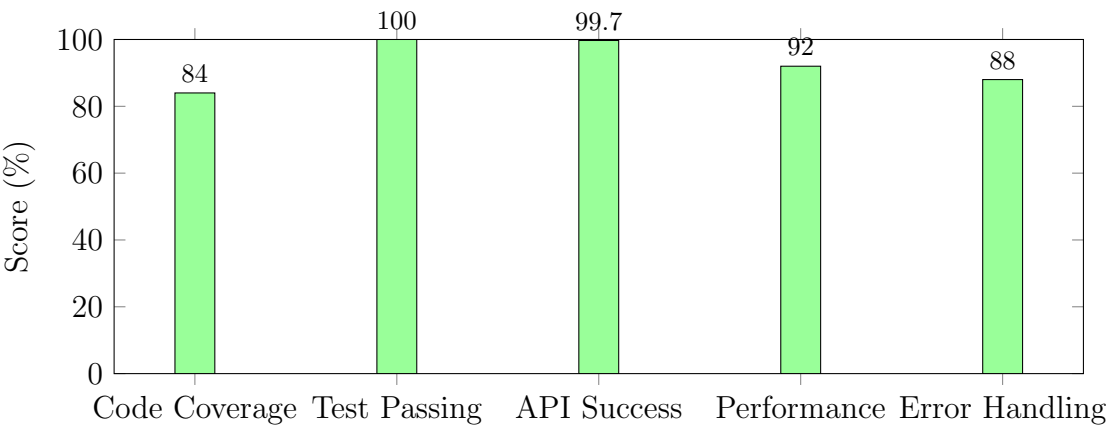


Figure 4.19: Quality Metrics Overview

### 4.2.6 API Test Results

Test Case	Status	Response Time	Verification
User Login (Valid)	Pass	156ms	JWT token returned
User Login (Invalid)	Pass	142ms	Error message returned
Fetch All Recipes	Pass	142ms	Recipe list returned
Create Recipe	Pass	210ms	Recipe ID returned
Get User Favorites	Pass	98ms	Favorite list returned
Update User Profile	Pass	189ms	Profile updated confirmation
Delete Recipe	Pass	175ms	Success response

Table 4.4: API End-to-End Test Results

# Chapter 5

## Conclusion and Future Work

### 5.1 Project Outcomes

PickMyDish successfully delivers an innovative solution to meal planning indecision through:

- **Complete Implementation:** Fully functional Flutter application with Node.js back-end
- **Architectural Excellence:** Hybrid architecture combining multiple patterns
- **Design Pattern Mastery:** Implementation of 4+ design patterns
- **DevOps Automation:** Complete CI/CD pipeline with Jenkins
- **Professional Deployment:** Production-ready VPS infrastructure
- **Comprehensive Documentation:** UML diagrams and technical documentation

### 5.2 Technical Challenges and Solutions

Challenge	Learning Outcome
VPS Configuration Complexity	Gained expertise in server administration, firewall configuration, and Nginx optimization
State Management in Flutter	Mastered Provider pattern and reactive programming paradigms
Database Optimization	Learned advanced MySQL indexing, query optimization, and connection pooling
CI/CD Pipeline Automation	Developed skills in Jenkins scripting, automated testing, and deployment strategies
Team Collaboration	Enhanced communication and project management skills using Agile methodologies

Table 5.1: Technical Challenges and Learning Outcomes

### 5.3 Future Enhancements

Feature	Priority	Description
AI Recipe Recommendations	High	Machine learning for personalized suggestions
Social Features	Medium	User profiles, following, recipe sharing
Meal Planning Calendar	High	Weekly meal planning with grocery lists
Voice Commands	Low	Voice-controlled recipe navigation
AR Cooking Assistance	Low	Augmented reality step-by-step guidance
Multi-language Support	Medium	Internationalization for global users
Advanced Analytics	Medium	User behavior analysis and insights

Table 5.2: Future Development Roadmap

### 5.4 Lessons Learned

<b>Technical Lessons:</b>	<b>Project Management Lessons:</b>
<ul style="list-style-type: none"><li>• Importance of proper database indexing</li><li>• Value of comprehensive error handling</li><li>• Benefits of automated testing in CI/CD</li><li>• Necessity of proper logging and monitoring</li><li>• Advantages of containerization for consistency</li></ul>	<ul style="list-style-type: none"><li>• Daily standups prevent misalignment</li><li>• Kanban boards improve workflow visibility</li><li>• Pair programming enhances code quality</li><li>• Regular deployments reduce integration risks</li><li>• Documentation saves time in long term</li></ul>

# Bibliography

- [1] Flutter Documentation. (2023). *Build apps for any screen*. Retrieved from <https://flutter.dev>
- [2] Jenkins Documentation. (2023). *Automate your development workflow*. Retrieved from <https://www.jenkins.io>
- [3] Contabo GmbH. (2023). *VPS Hosting Solutions*. Retrieved from <https://contabo.com>
- [4] Node.js Foundation. (2023). *Node.js JavaScript runtime*. Retrieved from <https://nodejs.org>
- [5] Oracle Corporation. (2023). *MySQL Database Management System*. Retrieved from <https://mysql.com>
- [6] NGINX, Inc. (2023). *High Performance Load Balancer*. Retrieved from <https://nginx.org>
- [7] Prisma Data, Inc. (2023). *Next-generation ORM for Node.js*. Retrieved from <https://prisma.io>

# Appendices

## Appendix A: GitHub Repository Structure

```
Pick-My-Dish/ (Repository 1)
  frontend/                # Flutter application
    lib/
      models/
      providers/
      screens/
      services/
      widgets/
    test/
    pubspec.yaml
  infrastructure/          # Deployment scripts
    nginx/
    scripts/
  diagrams/                 # UML diagrams
    use-case/
    sequence/
    class/
    deployment/
  docs/                     # Documentation
  Jenkinsfile               # CI/CD pipeline
  README.md                 # Project overview
```

```
Pick-My-Dish-Backend/ (Repository 2)
  src/
    controllers/
    middleware/
    models/
    routes/
    utils/
  prisma/
  package.json
```

## Appendix B: API Documentation

Method	Endpoint	Auth	Description
POST	/api/auth/register	No	Register new user
POST	/api/auth/login	No	User authentication
GET	/api/recipes	Optional	Get all recipes (filterable)
POST	/api/recipes	Yes	Create new recipe
GET	/api/recipes/:id	Optional	Get specific recipe
PUT	/api/recipes/:id	Yes	Update recipe
DELETE	/api/recipes/:id	Yes	Delete recipe
GET	/api/users/favorites	Yes	Get user's favorite recipes
POST	/api/users/favorites	Yes	Add recipe to favorites
DELETE	/api/users/favorites	Yes	Remove recipe from favorites

Table 5.3: API Endpoints Summary

## Appendix C: Scrum Artifacts

- [GitHub Projects Board](#)
- [Sprint Retrospective Notes](#) (Available in repository)
- [Daily Standup Records](#) (Discord channel archives)
- [Burndown Charts](#) (Generated from GitHub Projects)
- [Velocity Tracking](#) (3.5 story points per day average)
- GITHUB REPOSITORY LINKS (<https://github.com/Kynmmarshall/Pick-My-Dish>)  
AND (<https://github.com/Kynmmarshall/Pick-My-Dish-Backend>)