Smart Recipe Manager

Technical Documentation

Full-Stack Web Application with AI Integration

Hamza El Ouattab
GitHub: @HAMZA-EL-OUATTAB

hamza.elouattab@example.com

November 11, 2025

Contents

1 Executive Summary

1.1 Project Overview

The **Smart Recipe Manager** is a modern, full-stack web application designed to revolutionize the way users manage and interact with recipes. Built with cutting-edge technologies, this platform provides an intuitive interface for creating, storing, and analyzing recipes with AI-powered nutritional insights.

1.2 Key Features

- **Recipe Management:** Complete CRUD operations for recipes with rich metadata
- **User Authentication:** Secure user registration and login system (planned)
- **AI Integration:** Automatic nutritional analysis using machine learning (planned)
- **Smart Search:** Advanced filtering and search capabilities
- **Responsive Design:** Mobile-friendly interface with modern UI/UX
- **Scalable Architecture:** Designed for containerization and cloud deployment

1.3 Technology Stack

Layer	**Technology**
Frontend	React 18, Vite, Tailwind CSS, Axios
Backend	Spring Boot 3.5, Spring Data JPA, Hibernate
Database	PostgreSQL 16
Build Tools	Maven, npm
Future	Docker, Kubernetes, Hugging Face API

Table 1: Technology Stack Overview

2 System Architecture

2.1 High-Level Architecture

The Smart Recipe Manager follows a modern three-tier architecture pattern, separating concerns between presentation, business logic, and data persistence layers.

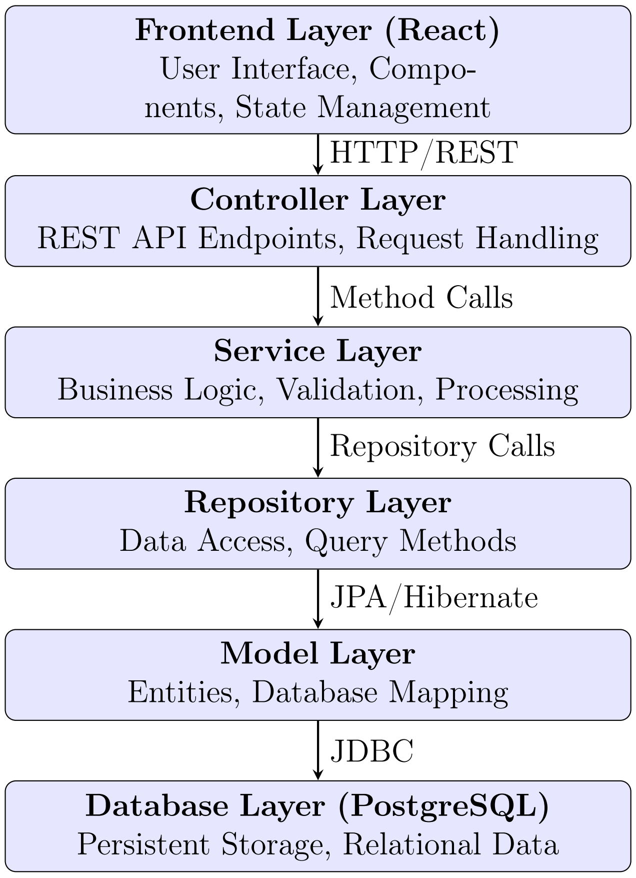
Frontend Layer (React)
User Interface, Components, State Management
HTTP/REST
Controller Layer
REST API Endpoints, Request Handling
Method Calls
Service Layer
Business Logic, Validation, Processing
Repository Calls
Repository Layer
Data Access, Query Methods
JPA/Hibernate
Model Layer
Entities, Database Mapping
JDBC
Database Layer (PostgreSQL)
Persistent Storage, Relational Data

Figure 1: Three-Tier Architecture Diagram

2.2 Component Interaction Flow

The following sequence describes a typical user interaction:

1. User interacts with React frontend interface
2. Frontend sends HTTP request via Axios to Spring Boot backend
3. Controller receives request and validates input
4. Controller delegates to Service layer for business logic
5. Service calls Repository for data access
6. Repository uses JPA to query PostgreSQL database

7. Data flows back through layers to frontend
8. Frontend updates UI with response data

3 Database Design

3.1 Entity-Relationship Diagram

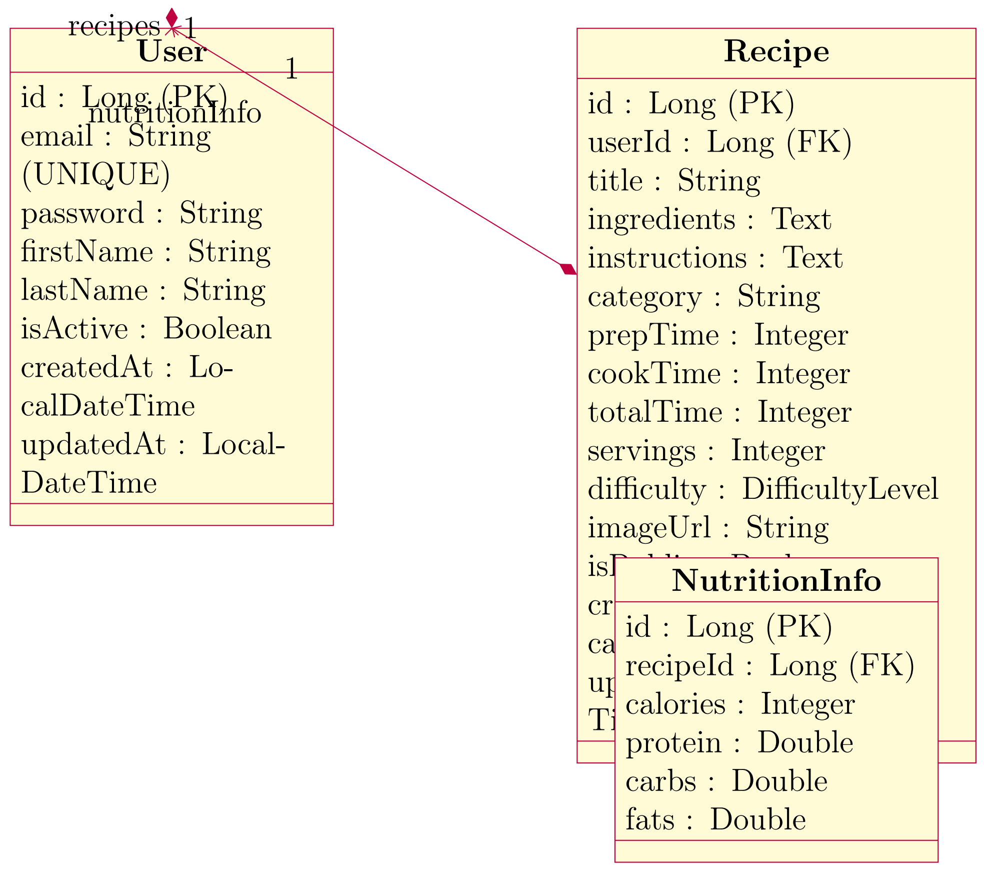

Figure 2: Database Entity-Relationship Diagram

3.2 Table Specifications

3.2.1 Users Table

Column	Type	Constraints	Description
id	BIGINT	PRIMARY KEY, AUTO INCREMENT	Unique identifier
email	VARCHAR(255)	NOT NULL, UNIQUE	User email address
password	VARCHAR(255)	NOT NULL	Hashed password
first_name	VARCHAR(255)	NULL	User's first name
last_name	VARCHAR(255)	NULL	User's last name
is_active	BOOLEAN	DEFAULT TRUE	Account status
created_at	TIMESTAMP	NOT NULL	Registration date
updated_at	TIMESTAMP	NULL	Last update date

Table 2: Users Table Structure

3.2.2 Recipes Table

Column	Type	Constraints	Description
id	BIGINT	PRIMARY KEY, AUTO INCREMENT	Unique identifier
user_id	BIGINT	NOT NULL, FOREIGN KEY	Owner reference
title	VARCHAR(200)	NOT NULL	Recipe title
ingredients	TEXT	NOT NULL	List of ingredients
instructions	TEXT	NOT NULL	Cooking steps
category	VARCHAR(50)	NULL	Recipe category
prep_time	INTEGER	NULL	Preparation time (min)
cook_time	INTEGER	NULL	Cooking time (min)
total_time	INTEGER	NULL	Total time (min)
servings	INTEGER	NULL	Number of servings
difficulty	VARCHAR(20)	NULL	EASY/MEDIUM/HARD
image_url	VARCHAR(255)	NULL	Recipe image URL
is_public	BOOLEAN	DEFAULT TRUE	Visibility status
created_at	TIMESTAMP	NOT NULL	Creation date
updated_at	TIMESTAMP	NULL	Last update date

Table 3: Recipes Table Structure

3.2.3 Nutrition Info Table

Column	Type	Constraints	Description
id	BIGINT	PRIMARY KEY, AUTO INCREMENT	Unique identifier
recipe_id	BIGINT	NOT NULL, UNIQUE, FOREIGN KEY	Recipe reference
calories	INTEGER	NULL	Total calories
protein_grams	DOUBLE	NULL	Protein content (g)
carbs_grams	DOUBLE	NULL	Carbohydrates (g)
fats_grams	DOUBLE	NULL	Fat content (g)

Table 4: Nutrition Info Table Structure

3.3 Database Indexes

Performance optimization through strategic indexing:

- **idx_user_id** on `recipes(user_id)` - Accelerates user recipe queries
- **idx_category** on `recipes(category)` - Optimizes category filtering
- **idx_difficulty** on `recipes(difficulty)` - Speeds up difficulty searches
- **uk_email** on `users(email)` - Ensures email uniqueness
- **uk_recipe_id** on `nutrition_info(recipe_id)` - One-to-one enforcement

4 Backend Implementation

4.1 Spring Boot Architecture

4.1.1 Model Layer (Entities)

The model layer represents database tables as Java objects using JPA annotations.
User Entity Example:

```java
@Entity
@Table(name = "users")
@Data
@NoArgsConstructor
@AllArgsConstructor
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false, unique = true)
    private String email;

    @Column(nullable = false)
    private String password;

    @Column(name = "first_name")
    private String firstName;

    @Column(name = "last_name")
    private String lastName;

    @Column(name = "is_active")
    private Boolean isActive = true;

    @CreationTimestamp
    @Column(name = "created_at", nullable = false)
    private LocalDateTime createdAt;

    @UpdateTimestamp
    @Column(name = "updated_at")
    private LocalDateTime updatedAt;

    @OneToMany(mappedBy = "user",
               cascade = CascadeType.ALL)
    private List<Recipe> recipes = new ArrayList<>();
}
```

Listing 1: User.java Entity

4.1.2 Repository Layer

Repositories provide database access without writing SQL.

```java
@Repository
```

```java
public interface UserRepository
    extends JpaRepository<User, Long> {

    // Spring Data JPA generates implementation
    Optional<User> findByEmail(String email);

    boolean existsByEmail(String email);

    List<User> findByIsActive(Boolean isActive);
}
```

Listing 2: UserRepository.java

Auto-Generated SQL:

```sql
-- findByEmail generates:
SELECT * FROM users WHERE email = ?

-- existsByEmail generates:
SELECT COUNT(*) > 0 FROM users WHERE email = ?

-- findByIsActive generates:
SELECT * FROM users WHERE is_active = ?
```

4.1.3 Controller Layer

Controllers expose REST API endpoints.

```java
@RestController
@RequestMapping("/api/recipes")
@CrossOrigin(origins = "http://localhost:5173")
public class RecipeController {

    @Autowired
    private RecipeService recipeService;

    @GetMapping
    public ResponseEntity<Page<RecipeDTO>> getAll(
        @RequestParam(defaultValue = "0") int page,
        @RequestParam(defaultValue = "10") int size
    ) {
        Page<RecipeDTO> recipes =
            recipeService.findAll(page, size);
        return ResponseEntity.ok(recipes);
    }

    @PostMapping
    public ResponseEntity<RecipeDTO> create(
        @Valid @RequestBody CreateRecipeRequest request
    ) {
        RecipeDTO created =
            recipeService.create(request);
        return ResponseEntity
            .status(HttpStatus.CREATED)
            .body(created);
    }

    @GetMapping("/{id}")
    public ResponseEntity<RecipeDTO> getById(
        @PathVariable Long id
    ) {
        RecipeDTO recipe = recipeService.findById(id);
        return ResponseEntity.ok(recipe);
    }
}
```

Listing 3: RecipeController.java (Planned)

4.2 Configuration

4.2.1 CORS Configuration

```java
@Configuration
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void addCorsMappings(
        CorsRegistry registry
    ) {
        registry.addMapping("/api/**")
            .allowedOrigins("http://localhost:5173")
            .allowedMethods("GET", "POST",
                            "PUT", "DELETE")
            .allowedHeaders("*")
            .allowCredentials(true);
    }
}
```

Listing 4: WebConfig.java

4.2.2 Application Properties

```properties
# Application Name
spring.application.name=recipe-manager-backend

# Database Configuration
spring.datasource.url=jdbc:postgresql://localhost:5432/recipedb
spring.datasource.username=recipeadmin
spring.datasource.password=recipe123
spring.datasource.driver-class-name=org.postgresql.Driver

# JPA/Hibernate
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.
    PostgreSQLDialect

# Server
```

```
server.port=8080
```

Listing 5: application.properties

5 Frontend Implementation

5.1 React Architecture

5.1.1 Component Structure

```
import { useState, useEffect } from 'react'
import { apiService } from './services/api'

function App() {
  const [backendStatus, setBackendStatus] = useState(null)
  const [loading, setLoading] = useState(true)
  const [error, setError] = useState(null)

  useEffect(() => {
    testBackendConnection()
  }, [])

  const testBackendConnection = async () => {
    setLoading(true)
    setError(null)

    try {
      const [testResponse, healthResponse] =
        await Promise.all([
          apiService.test(),
          apiService.health()
        ])

      setBackendStatus(testResponse.data)
    } catch (err) {
      setError(err.message)
    } finally {
      setLoading(false)
    }
  }

  return (
    <div className="min-h-screen bg-gradient-to-br
                    from-blue-50 to-indigo-100">
      {/* UI Components */}
    </div>
  )
}

export default App
```

Listing 6: App.jsx Main Component

5.1.2 API Service Layer

```
import axios from 'axios';

```

```
const api = axios.create({
  baseURL: import.meta.env.VITE_API_URL ||
           'http://localhost:8080/api',
  headers: {
    'Content-Type': 'application/json',
  },
  timeout: 10000,
});

// Request interceptor for JWT tokens
api.interceptors.request.use((config) => {
  const token = localStorage.getItem('token');
  if (token) {
    config.headers.Authorization = `Bearer ${token}`;
  }
  return config;
});

export const apiService = {
  test: () => api.get('/test'),
  health: () => api.get('/health'),

  recipes: {
    getAll: (page = 0, size = 10) =>
      api.get(`/recipes?page=${page}&size=${size}`),
    getById: (id) => api.get(`/recipes/${id}`),
    create: (data) => api.post('/recipes', data),
    update: (id, data) =>
      api.put(`/recipes/${id}`, data),
    delete: (id) => api.delete(`/recipes/${id}`),
  },
};
```

Listing 7: api.js Service

5.2 Styling with Tailwind CSS

Tailwind CSS provides utility-first styling approach:

```
<div className="bg-white rounded-2xl shadow-xl p-8">
  <h2 className="text-2xl font-bold text-gray-800
                 mb-6 flex items-center">
    Backend Connection Status
  </h2>

  <div className="bg-green-50 border-l-4
                  border-green-500 p-4 rounded">
    <p className="font-bold text-green-800">
      Backend Connected!
    </p>
  </div>
</div>
```

Listing 8: Tailwind CSS Usage Example

6 API Specification

6.1 Base URL

`http://localhost:8080/api`

6.2 Authentication Endpoints (Planned)

6.2.1 Register User

POST /api/auth/register

Request Body:

```
{
  "email": "user@example.com",
  "password": "password123",
  "firstName": "John",
  "lastName": "Doe"
}
```

Response: 201 Created

```
{
  "id": 1,
  "email": "user@example.com",
  "firstName": "John",
  "lastName": "Doe",
  "createdAt": "2024-11-09T21:30:00"
}
```

6.2.2 Login

POST /api/auth/login

Request Body:

```
{
  "email": "user@example.com",
  "password": "password123"
}
```

Response: 200 OK

```
{
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",
  "user": {
    "id": 1,
    "email": "user@example.com",
    "firstName": "John"
  }
}
```

6.3 Recipe Endpoints (Planned)

6.3.1 Get All Recipes

GET /api/recipes?page

Response: 200 OK

```
{
  "content": [
    {
      "id": 1,
      "title": "Spaghetti Carbonara",
      "category": "Dinner",
      "prepTime": 15,
      "cookTime": 15,
      "totalTime": 30,
      "servings": 4,
      "difficulty": "MEDIUM"
    }
  ],
  "totalElements": 50,
  "totalPages": 5,
  "currentPage": 0
}
```

6.3.2 Create Recipe

POST /api/recipes

Headers: `Authorization: Bearer {token}`
Request Body:

```
{
  "title": "Spaghetti Carbonara",
  "ingredients": "400g spaghetti, 200g bacon, ...",
  "instructions": "1. Cook pasta...",
  "category": "Dinner",
  "prepTime": 15,
  "cookTime": 15,
  "servings": 4,
  "difficulty": "MEDIUM"
}
```

Response: 201 Created

6.4 Error Responses

Code	Status	Description
200	OK	Request successful
201	Created	Resource created
400	Bad Request	Invalid input
401	Unauthorized	Authentication required
403	Forbidden	Access denied
404	Not Found	Resource not found
500	Internal Server Error	Server error

Table 5: HTTP Status Codes

7 Setup and Installation

7.1 Prerequisites

- **Java 17 or higher** - https://www.oracle.com/java/technologies/downloads/
- **Node.js 18 or higher** - https://nodejs.org/
- **PostgreSQL 16** - https://www.postgresql.org/download/
- **Git** - https://git-scm.com/

7.2 Installation Steps

7.2.1 1. Clone Repository

```
git clone https://github.com/HAMZA-EL-OUATTAB/smart-recipe-manager.git
cd smart-recipe-manager
```

7.2.2 2. Database Setup

```
-- Connect to PostgreSQL
psql -U postgres

-- Create database
CREATE DATABASE recipedb;

-- Create user
CREATE USER recipeadmin WITH PASSWORD 'recipe123';

-- Grant privileges
GRANT ALL PRIVILEGES ON DATABASE recipedb TO recipeadmin;

-- Connect to database
\c recipedb

-- Grant schema privileges
GRANT ALL ON SCHEMA public TO recipeadmin;
```

7.2.3 3. Backend Setup

```
# Navigate to backend directory
cd backend

# Run the application
./mvnw spring-boot:run

# Backend starts on http://localhost:8080
```

7.2.4 4. Frontend Setup

```
# Navigate to frontend directory
cd frontend

# Install dependencies
npm install

# Start development server
npm run dev

# Frontend starts on http://localhost:5173
```

7.3 Verification

Backend Test:

```
curl http://localhost:8080/api/test
```

Expected Response:

```
{
  "status": "success",
  "message": "Backend is running!",
  "developer": "Hamza El Ouattab"
}
```

8 Design Patterns and Best Practices

8.1 Architectural Patterns

8.1.1 Repository Pattern

Abstracts data access logic from business logic.
Benefits:

- Separation of concerns
- Testability (easy to mock)
- Centralized query logic
- Database independence

8.1.2 Dependency Injection

Spring manages object lifecycle and dependencies.

```
@RestController
public class RecipeController {

    @Autowired  // Dependency Injection
    private RecipeService recipeService;

    // Spring automatically provides instance
}
```

8.1.3 DTO Pattern

Separates internal entities from API responses.

```
// Internal Entity
@Entity
public class User {
    private String password;  // Sensitive
}

// External DTO
public class UserDTO {
    // No password field!
    private String email;
    private String firstName;
}
```

8.2 Code Quality Practices

- **Lombok:** Reduces boilerplate code
- **Validation:** Input validation with annotations

- **Exception Handling:** Centralized error handling
- **Logging:** SLF4J for structured logging
- **Transaction Management:** @Transactional for data consistency

9 Performance and Scalability

9.1 Database Optimization

9.1.1 Indexing Strategy

Index	Column	Purpose
idx_user_id	recipes.user_id	User recipe lookup
idx_category	recipes.category	Category filtering
idx_difficulty	recipes.difficulty	Difficulty filtering
uk_email	users.email	Unique constraint + fast lookup

Table 6: Database Index Strategy

9.1.2 Connection Pooling

HikariCP provides efficient connection management:

- **Pool Size:** 10 connections default
- **Timeout:** 30 seconds
- **Idle Timeout:** 600 seconds
- **Max Lifetime:** 1800 seconds

9.2 Application Performance

9.2.1 Lazy Loading

JPA relationships use lazy loading to prevent N+1 queries:

```java
@ManyToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "user_id")
private User user;  // Loaded only when accessed
```

9.2.2 Pagination

Large datasets are paginated for efficiency:

```java
Page<Recipe> recipes = recipeRepository
    .findAll(PageRequest.of(page, size));
```

9.3 Scalability Considerations

9.3.1 Horizontal Scaling

- **Stateless Backend:** No session storage, JWT tokens
- **Database Read Replicas:** Future implementation
- **Caching Layer:** Redis for frequently accessed data (planned)
- **Load Balancing:** Kubernetes handles distribution

9.3.2 Vertical Scaling

- **JVM Tuning:** Heap size optimization
- **Connection Pool Sizing:** Based on load testing
- **Database Tuning:** PostgreSQL parameter optimization

10 Security Implementation

10.1 Authentication and Authorization (Planned)

10.1.1 JWT Token Flow

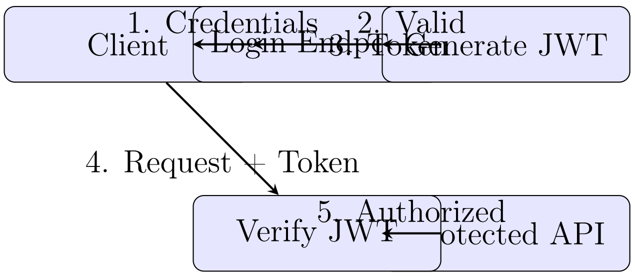

Figure 3: JWT Authentication Flow

10.1.2 Password Security

```java
@Configuration
public class SecurityConfig {

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder(12);
    }

    public String hashPassword(String rawPassword) {
        return passwordEncoder().encode(rawPassword);
    }

    public boolean verifyPassword(
        String rawPassword,
        String hashedPassword
    ) {
        return passwordEncoder()
            .matches(rawPassword, hashedPassword);
    }
}
```

10.2 Security Best Practices

- **Password Hashing:** BCrypt with cost factor 12
- **SQL Injection Prevention:** Parameterized queries via JPA
- **XSS Prevention:** React escapes output by default
- **CSRF Protection:** Token-based for state-changing operations
- **CORS Configuration:** Strict origin whitelist

- **Input Validation:** Server-side validation required
- **HTTPS:** Enforced in production
- **Rate Limiting:** Prevent abuse (planned)

10.3 Data Protection

10.3.1 Sensitive Data Handling

Data Type	**Storage**	**Transmission**
Passwords	BCrypt hashed	HTTPS only
JWT Tokens	Not stored server-side	HTTPS + HttpOnly
User Email	Encrypted at rest (planned)	HTTPS
Personal Info	Database encryption (planned)	HTTPS

Table 7: Data Protection Strategy

11 Testing Strategy

11.1 Testing Pyramid

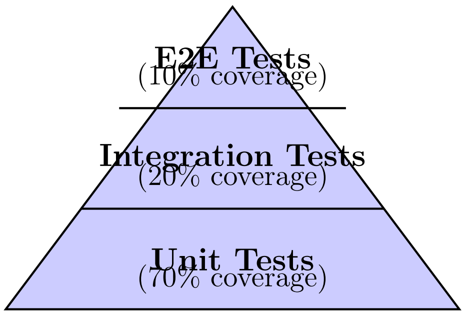

Figure 4: Testing Pyramid Strategy

11.2 Unit Testing

11.2.1 Backend Unit Tests (Planned)

```java
@SpringBootTest
class RecipeServiceTest {

    @Mock
    private RecipeRepository recipeRepository;

    @InjectMocks
    private RecipeService recipeService;

    @Test
    void testCreateRecipe_Success() {
        // Given
        CreateRecipeRequest request =
            new CreateRecipeRequest();
        request.setTitle("Test Recipe");

        Recipe recipe = new Recipe();
        recipe.setId(1L);

        when(recipeRepository.save(any(Recipe.class)))
            .thenReturn(recipe);

        // When
        RecipeDTO result =
            recipeService.create(request);

        // Then
        assertNotNull(result);
        assertEquals(1L, result.getId());
        verify(recipeRepository, times(1))
            .save(any(Recipe.class));
    }

    @Test
    void testCreateRecipe_EmptyTitle_ThrowsException() {
        // Given
        CreateRecipeRequest request =
            new CreateRecipeRequest();
        request.setTitle("");

        // When & Then
        assertThrows(
            ValidationException.class,
            () -> recipeService.create(request)
        );
    }
}
```

11.2.2 Frontend Unit Tests (Planned)

```
import { render, screen, waitFor } from '@testing-library/react'
import { apiService } from './services/api'
import App from './App'

jest.mock('./services/api')

describe('App Component', () => {
  test('displays backend status on load', async () => {
    // Mock API response
    apiService.test.mockResolvedValue({
      data: {
        status: 'success',
        message: 'Backend is running!'
      }
    })

    // Render component
    render(<App />)

    // Wait for async data
    await waitFor(() => {
      expect(screen.getByText('Backend Connected!'))
        .toBeInTheDocument()
    })
  })

  test('displays error message on API failure', async () => {
    // Mock API error
    apiService.test.mockRejectedValue(
      new Error('Network Error')
    )

    render(<App />)

    await waitFor(() => {
      expect(screen.getByText('Connection Failed'))
```

```
            .toBeInTheDocument()
        })
    })
})
```

11.3 Integration Testing

11.3.1 Repository Integration Tests

```
@SpringBootTest
@AutoConfigureTestDatabase(
    replace = AutoConfigureTestDatabase.Replace.NONE
)
@Testcontainers
class UserRepositoryIntegrationTest {

    @Container
    static PostgreSQLContainer<?> postgres =
        new PostgreSQLContainer<>("postgres:16");

    @Autowired
    private UserRepository userRepository;

    @Test
    void testFindByEmail_Success() {
        // Given
        User user = new User();
        user.setEmail("test@example.com");
        user.setPassword("hashed");
        userRepository.save(user);

        // When
        Optional<User> found =
            userRepository.findByEmail("test@example.com");

        // Then
        assertTrue(found.isPresent());
        assertEquals("test@example.com",
                     found.get().getEmail());
    }
}
```

11.4 E2E Testing (Planned)

```
// Playwright E2E test
import { test, expect } from '@playwright/test'

test('user can create a recipe', async ({ page }) => {
  // Navigate to app
  await page.goto('http://localhost:5173')

  // Login
  await page.click('text=Login')
  await page.fill('#email', 'test@example.com')
  await page.fill('#password', 'password123')
  await page.click('button[type=submit]')

  // Create recipe
  await page.click('text=New Recipe')
  await page.fill('#title', 'Test Recipe')
  await page.fill('#ingredients', 'Ingredient 1')
  await page.fill('#instructions', 'Step 1')
  await page.click('button:has-text("Save")')

  // Verify
  await expect(page.locator('text=Test Recipe'))
    .toBeVisible()
})
```

11.5 Test Coverage Goals

Layer	Target Coverage	Priority
Service Layer	90%	High
Repository Layer	80%	High
Controller Layer	85%	High
Model Layer	100%	Medium
Utility Classes	90%	Medium

Table 8: Test Coverage Targets

12 Deployment Strategy

12.1 Development Environment

Component	URL	Port
Frontend	http://localhost:5173	5173
Backend	http://localhost:8080	8080
Database	localhost	5432

Table 9: Development Environment Configuration

12.2 Docker Containerization (Planned)

12.2.1 Backend Dockerfile

```
# Multi-stage build
FROM eclipse-temurin:17-jdk-alpine AS build
WORKDIR /app

# Copy Maven wrapper and pom.xml
COPY mvnw .
COPY .mvn .mvn
COPY pom.xml .

# Download dependencies
RUN ./mvnw dependency:go-offline

# Copy source and build
COPY src src
RUN ./mvnw clean package -DskipTests

# Production image
FROM eclipse-temurin:17-jre-alpine
WORKDIR /app

# Copy JAR from build stage
COPY --from=build /app/target/*.jar app.jar

# Non-root user for security
RUN addgroup -S spring && adduser -S spring -G spring
USER spring:spring

# Health check
HEALTHCHECK --interval=30s --timeout=3s \
  CMD wget -q --spider http://localhost:8080/api/health || exit 1

EXPOSE 8080
ENTRYPOINT ["java", "-jar", "app.jar"]
```

12.2.2 Frontend Dockerfile

```
# Build stage
FROM node:18-alpine AS build
WORKDIR /app

# Copy package files
COPY package*.json ./
RUN npm ci

# Copy source and build
COPY . .
RUN npm run build

# Production stage
FROM nginx:alpine

# Copy built files
COPY --from=build /app/dist /usr/share/nginx/html

# Custom nginx configuration
COPY nginx.conf /etc/nginx/conf.d/default.conf

# Non-root user
RUN chown -R nginx:nginx /usr/share/nginx/html

EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

12.2.3 Docker Compose

```
version: '3.8'

services:
  postgres:
    image: postgres:16-alpine
    container_name: recipe-db
    environment:
      POSTGRES_DB: recipedb
      POSTGRES_USER: recipeadmin
      POSTGRES_PASSWORD: recipe123
    ports:
      - "5432:5432"
    volumes:
      - postgres_data:/var/lib/postgresql/data
    networks:
      - recipe-network

  backend:
    build: ./backend
    container_name: recipe-backend
    ports:
      - "8080:8080"
    environment:
      SPRING_DATASOURCE_URL: jdbc:postgresql://postgres:5432/recipedb
      SPRING_DATASOURCE_USERNAME: recipeadmin
      SPRING_DATASOURCE_PASSWORD: recipe123
    depends_on:
      - postgres
    networks:
      - recipe-network

  frontend:
    build: ./frontend
    container_name: recipe-frontend
    ports:
      - "80:80"
    depends_on:
      - backend
    networks:
      - recipe-network

volumes:
  postgres_data:

networks:
  recipe-network:
    driver: bridge
```

12.3 Kubernetes Deployment (Planned)

12.3.1 Backend Deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: recipe-backend
  namespace: recipe-app
spec:
  replicas: 3
  selector:
    matchLabels:
      app: backend
  template:
    metadata:
      labels:
        app: backend
    spec:
      containers:
      - name: backend
        image: recipe-backend:latest
        ports:
        - containerPort: 8080
        env:
        - name: SPRING_DATASOURCE_URL
          valueFrom:
            configMapKeyRef:
              name: app-config
```

```
                key: database-url
          - name: SPRING_DATASOURCE_PASSWORD
            valueFrom:
              secretKeyRef:
                name: app-secrets
                key: db-password
          resources:
            requests:
              memory: "512Mi"
              cpu: "250m"
            limits:
              memory: "1Gi"
              cpu: "500m"
          livenessProbe:
            httpGet:
              path: /api/health
              port: 8080
            initialDelaySeconds: 60
            periodSeconds: 10
          readinessProbe:
            httpGet:
              path: /api/health
              port: 8080
            initialDelaySeconds: 30
            periodSeconds: 5
---
apiVersion: v1
kind: Service
metadata:
  name: backend-service
  namespace: recipe-app
spec:
  selector:
    app: backend
  ports:
  - port: 8080
    targetPort: 8080
  type: ClusterIP
```

12.3.2 Horizontal Pod Autoscaler

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: backend-hpa
  namespace: recipe-app
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: recipe-backend
  minReplicas: 2
  maxReplicas: 10
  metrics:
```

```
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 70
  - type: Resource
    resource:
      name: memory
      target:
        type: Utilization
        averageUtilization: 80
```

12.4 CI/CD Pipeline (Planned)

12.4.1 GitHub Actions Workflow

```
name: CI/CD Pipeline

on:
  push:
    branches: [ main, develop ]
  pull_request:
    branches: [ main ]

jobs:
  test-backend:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: Set up JDK 17
        uses: actions/setup-java@v3
        with:
          java-version: '17'
          distribution: 'temurin'

      - name: Run tests
        run: |
          cd backend
          ./mvnw clean test

      - name: Build
        run: |
          cd backend
          ./mvnw clean package -DskipTests

  test-frontend:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: Setup Node.js
        uses: actions/setup-node@v3
```

```yaml
        with:
          node-version: '18'

      - name: Install dependencies
        run: |
          cd frontend
          npm ci

      - name: Run tests
        run: |
          cd frontend
          npm test

      - name: Build
        run: |
          cd frontend
          npm run build

  build-and-push:
    needs: [test-backend, test-frontend]
    runs-on: ubuntu-latest
    if: github.ref == 'refs/heads/main'
    steps:
      - uses: actions/checkout@v3

      - name: Login to Docker Hub
        uses: docker/login-action@v2
        with:
          username: ${{ secrets.DOCKER_USERNAME }}
          password: ${{ secrets.DOCKER_PASSWORD }}

      - name: Build and push
        run: |
          docker build -t username/recipe-backend:latest ./backend
          docker push username/recipe-backend:latest
          docker build -t username/recipe-frontend:latest ./frontend
          docker push username/recipe-frontend:latest

  deploy:
    needs: build-and-push
    runs-on: ubuntu-latest
    steps:
      - name: Deploy to Kubernetes
        run: |
          kubectl apply -f k8s/
```

13 Monitoring and Observability

13.1 Logging Strategy

13.1.1 Structured Logging

```java
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

@Service
public class RecipeService {

    private static final Logger log =
        LoggerFactory.getLogger(RecipeService.class);

    public RecipeDTO create(CreateRecipeRequest request) {
        log.info("Creating recipe: title={}",
                 request.getTitle());

        try {
            Recipe recipe = recipeRepository.save(entity);
            log.info("Recipe created successfully: id={}",
                     recipe.getId());
            return toDTO(recipe);
        } catch (Exception e) {
            log.error("Failed to create recipe: title={}",
                      request.getTitle(), e);
            throw e;
        }
    }
}
```

13.2 Metrics Collection (Planned)

13.2.1 Spring Boot Actuator

```
# Enable actuator endpoints
management.endpoints.web.exposure.include=health,metrics,prometheus
management.endpoint.health.show-details=always
management.metrics.export.prometheus.enabled=true
```

13.2.2 Key Metrics

- **Application Metrics:**
  - Request count and latency
  - Error rates by endpoint
  - Active users
  - Recipe creation rate

- **System Metrics:**
  - CPU utilization
  - Memory usage
  - Garbage collection metrics
  - Thread pool statistics
- **Database Metrics:**
  - Connection pool usage
  - Query execution time
  - Active connections
  - Slow query count

13.3 Health Checks

```java
@Component
public class DatabaseHealthIndicator
    implements HealthIndicator {

    @Autowired
    private UserRepository userRepository;

    @Override
    public Health health() {
        try {
            long count = userRepository.count();
            return Health.up()
                .withDetail("database", "PostgreSQL")
                .withDetail("userCount", count)
                .build();
        } catch (Exception e) {
            return Health.down()
                .withDetail("error", e.getMessage())
                .build();
        }
    }
}
```

14 Future Enhancements

14.1 Week 2 Roadmap

1. **Authentication System**
    - JWT-based authentication
    - User registration and login
    - Password reset functionality
    - Email verification
2. **Complete Recipe CRUD**
    - Update recipe endpoint
    - Delete recipe with cascade
    - Recipe validation rules
    - Image upload support
3. **AI Integration**
    - Hugging Face API integration
    - Automatic nutritional analysis
    - Ingredient recognition
    - Recipe suggestions
4. **Search and Filter**
    - Full-text search
    - Advanced filtering
    - Sorting options
    - Saved searches

14.2 Week 3 Roadmap

1. **Containerization**
    - Docker images optimization
    - Docker Compose setup
    - Container orchestration
2. **Kubernetes Deployment**
    - Deployment manifests
    - Service configuration

- Ingress setup
- Secrets management

3. **CI/CD Pipeline**
   - GitHub Actions workflows
   - Automated testing
   - Automated deployment
   - Environment promotion
4. **Production Deployment**
   - Cloud provider selection
   - SSL certificate setup
   - Domain configuration
   - Monitoring setup

14.3 Long-term Features

- **Social Features:**
  - Recipe sharing
  - User following
  - Comments and ratings
  - Recipe collections
- **Advanced AI:**
  - Recipe recommendations
  - Meal planning
  - Dietary restriction filtering
  - Shopping list generation
- **Mobile Application:**
  - React Native app
  - Offline support
  - Push notifications
  - Camera integration
- **Analytics:**
  - User behavior tracking
  - Popular recipes dashboard
  - Usage statistics
  - A/B testing framework

15 Troubleshooting Guide

15.1 Common Issues

15.1.1 Backend Won't Start

Symptom: Application fails to start with database connection error.
Solutions:

1. Verify PostgreSQL is running:
```
pg_isready -h localhost -p 5432

```
2. Check database credentials in `application.properties`
3. Verify database exists:
```
psql -U postgres -l | grep recipedb

```
4. Check Java version:
```
java -version  # Should be 17 or higher

```

15.1.2 Frontend CORS Errors

Symptom: Browser console shows CORS policy errors.
Solutions:

1. Verify `WebConfig.java` exists in backend
2. Check allowed origins match frontend URL
3. Restart backend after CORS configuration changes
4. Clear browser cache

15.1.3 Database Connection Timeout

Symptom: `HikariPool connection timeout`
Solutions:

1. Increase connection timeout in `application.properties`:
```
spring.datasource.hikari.connection-timeout=60000
spring.datasource.hikari.maximum-pool-size=10

```
2. Check database connection limit:
```
SHOW max_connections;

```
3. Verify no connection leaks in code

15.2 Performance Issues

15.2.1 Slow Database Queries

Diagnosis:

```sql
-- Enable slow query logging
SET log_min_duration_statement = 1000;  -- Log queries > 1s

-- Analyze query performance
EXPLAIN ANALYZE SELECT * FROM recipes WHERE user_id = 1;
```

Solutions:

1. Add missing indexes
2. Use pagination for large datasets
3. Optimize N+1 query problems with JOIN FETCH
4. Enable query caching

15.3 Debugging Tips

Tool	Purpose
IntelliJ Debugger	Step through backend code
React DevTools	Inspect component state and props
Chrome DevTools	Network requests, console errors
Postman	Test API endpoints independently
pgAdmin	Query database directly
Docker logs	View container output

Table 10: Debugging Tools

16 Conclusion

16.1 Project Status

The Smart Recipe Manager has successfully completed Week 1 of development, establishing a solid foundation for a production-ready full-stack application.

16.1.1 Completed Components

- **Backend:** Spring Boot REST API with complete database integration
- **Frontend:** React application with modern UI and API connectivity
- **Database:** PostgreSQL with optimized schema and indexes
- **Architecture:** Clean, scalable three-tier design
- **Documentation:** Comprehensive technical documentation

16.2 Technical Achievements

1. **Professional Architecture:** Industry-standard layered design
2. **Best Practices:** Repository pattern, dependency injection, DTO pattern
3. **Performance:** Database indexing, connection pooling, lazy loading
4. **Security Foundations:** CORS configuration, input validation
5. **Scalability:** Designed for horizontal scaling and containerization

16.3 Learning Outcomes

This project demonstrates proficiency in:

- Full-stack web application development
- RESTful API design and implementation
- Relational database modeling and optimization
- Modern frontend frameworks (React)
- Backend frameworks (Spring Boot)
- Version control with Git
- Technical documentation

16.4 Next Steps

The project is ready to progress to Week 2, which will implement:

1. User authentication and authorization
2. Complete recipe management features
3. AI-powered nutritional analysis
4. Advanced search and filtering
5. Image upload capabilities

16.5 Final Remarks

The Smart Recipe Manager represents a modern, well-architected full-stack application that follows industry best practices and is ready for further enhancement and production deployment.

The codebase is maintainable, scalable, and extensible, providing a solid foundation for adding advanced features and deploying to cloud infrastructure.

A Appendix A: Configuration Files

A.1 application.properties (Complete)

```
# Application Name
spring.application.name=recipe-manager-backend

# Database Configuration
spring.datasource.url=jdbc:postgresql://localhost:5432/recipedb
spring.datasource.username=recipeadmin
spring.datasource.password=recipe123
spring.datasource.driver-class-name=org.postgresql.Driver

# JPA/Hibernate Configuration
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa
.properties.hibernate.format_sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.
    PostgreSQLDialect

# HikariCP Connection Pool
spring.datasource.hikari.maximum-pool-size=10
spring.datasource.hikari.minimum-idle=5
spring.datasource.hikari.connection-timeout=60000
spring.datasource.hikari.idle-timeout=600000
spring.datasource.hikari.max-lifetime=1800000

# Server Configuration
server.port=8080
server.error.include-message=always
server.error.include-binding-errors=always

# Logging Configuration
logging.level.com.recipemanager=DEBUG
logging.level.org.springframework.web=INFO
logging.level.org.hibernate.SQL=DEBUG
logging.level.org.hibernate.type.descriptor.sql.BasicBinder=TRACE

# Spring Boot Actuator (Planned)
management.endpoints.web.exposure.include=health,info,metrics
management.endpoint.health.show-details=always
```

A.2 package.json (Frontend)

```
{
  "name": "smart-recipe-manager-frontend",
  "version": "1.0.0",
  "type": "module",
  "scripts": {
    "dev": "vite",
    "build": "vite build",
    "preview": "vite preview",
    "test": "vitest",
    "lint": "eslint . --ext js,jsx"
  },
  "dependencies": {
    "react": "^18.2.0",
    "react-dom": "^18.2.0",
    "react-router-dom": "^6.20.0",
    "axios": "^1.6.0"
  },
  "devDependencies": {
    "@types/react": "^18.2.0",
    "@types/react-dom": "^18.2.0",
    "@vitejs/plugin-react": "^4.2.0",
    "autoprefixer": "^10.4.16",
    "postcss": "^8.4.32",
    "tailwindcss": "^3.4.0",
    "vite": "^5.0.0"
  }
}
```

A.3 pom.xml (Backend - Key Dependencies)

```
<dependencies>
    <!-- Spring Boot Starters -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-validation</artifactId>
    </dependency>

    <!-- Database -->
    <dependency>
        <groupId>org.postgresql</groupId>
        <artifactId>postgresql</artifactId>
        <scope>runtime</scope>
    </dependency>

    <!-- Utilities -->
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <optional>true</optional>
    </dependency>

    <!-- DevTools -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
```

```xml
        <artifactId>spring-boot-devtools</artifactId>
        <scope>runtime</scope>
    </dependency>

    <!-- Testing -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>
```

B Appendix B: Database Schema SQL

B.1 Complete Schema Creation

```sql
-- Create database
CREATE DATABASE recipedb;

-- Connect to database
\c recipedb;

-- Create users table
CREATE TABLE users (
    id BIGSERIAL PRIMARY KEY,
    email VARCHAR(255) NOT NULL UNIQUE,
    password VARCHAR(255) NOT NULL,
    first_name VARCHAR(255),
    last_name VARCHAR(255),
    is_active BOOLEAN DEFAULT TRUE,
    created_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP
);

-- Create recipes table
CREATE TABLE recipes (
    id BIGSERIAL PRIMARY KEY,
    user_id BIGINT NOT NULL,
    title VARCHAR(200) NOT NULL,
    ingredients TEXT NOT NULL,
    instructions TEXT NOT NULL,
    category VARCHAR(50),
    prep_time INTEGER,
    cook_time INTEGER,
    total_time INTEGER,
    servings INTEGER,
    difficulty VARCHAR(20) CHECK (difficulty IN ('EASY', 'MEDIUM', 'HARD
    ')),
    image_url VARCHAR(255),
    is_public BOOLEAN DEFAULT TRUE,
    created_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP,
    FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE
);

-- Create nutrition_info table
CREATE TABLE nutrition_info (
    id BIGSERIAL PRIMARY KEY,
    recipe_id BIGINT NOT NULL UNIQUE,
    calories INTEGER,
    protein_grams DOUBLE PRECISION,
    carbs_grams DOUBLE PRECISION,
    fats_grams DOUBLE PRECISION,
    FOREIGN KEY (recipe_id) REFERENCES recipes(id) ON DELETE CASCADE
);

-- Create indexes
```

```sql
CREATE INDEX idx_user_id ON recipes(user_id);
CREATE INDEX idx_category ON recipes(category);
CREATE INDEX idx_difficulty ON recipes(difficulty);
CREATE INDEX idx_email ON users(email);

-- Grant privileges
GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA public TO recipeadmin;
GRANT ALL PRIVILEGES ON ALL SEQUENCES IN SCHEMA public TO recipeadmin;

-- Sample data (optional)
INSERT INTO users (email, password, first_name, last_name, is_active)
VALUES
    ('test@example.com', 'hashed_password', 'Test', 'User', true),
    ('admin@example.com', 'hashed_password', 'Admin', 'User', true);

INSERT INTO recipes (user_id, title, ingredients, instructions,
                     category, prep_time, cook_time, servings,
    difficulty)
VALUES
    (1, 'Spaghetti Carbonara',
     '400g spaghetti, 200g bacon, 4 eggs, 100g parmesan cheese',
     '1. Cook pasta. 2. Fry bacon. 3. Mix eggs and cheese. 4. Combine.',
     'Dinner', 15, 15, 4, 'MEDIUM'),
    (1, 'Caesar Salad',
     'Romaine lettuce, croutons, parmesan, caesar dressing',
     '1. Wash lettuce. 2. Add croutons and cheese. 3. Toss with dressing
    .',
     'Salad', 10, 0, 2, 'EASY');
```

C Appendix C: API Request Examples

C.1 cURL Commands

C.1.1 Test Backend Connection

```
curl -X GET http://localhost:8080/api/test
```

Expected Response:

```
{
  "status": "success",
  "message": "Backend is running!",
  "timestamp": "2024-11-09T21:30:00.123456",
  "developer": "Hamza El Ouattab"
}
```

C.1.2 Health Check

```
curl -X GET http://localhost:8080/api/health
```

Expected Response:

```
{
  "status": "UP",
  "service": "Recipe Manager API"
}
```

C.1.3 Test Database Connection

```
curl -X GET http://localhost:8080/api/database/test
```

Expected Response:

```
{
  "status": "success",
  "message": "Database connection working!",
  "userCount": 0,
  "database": "PostgreSQL",
  "tablesCreated": "users, recipes, nutrition_info"
}
```

C.1.4 Create Test User

```
curl -X POST http://localhost:8080/api/database/test-user
```

Expected Response:

```
{
  "status": "success",
  "message": "Test user created!",
  "userId": 1,
  "email": "test@example.com",
  "firstName": "Test",
  "lastName": "User",
  "createdAt": "2024-11-09T21:30:00"
}
```

C.1.5 Get All Users

```
curl -X GET http://localhost:8080/api/database/users
```

C.1.6 Create Recipe (Planned)

```
curl -X POST http://localhost:8080/api/recipes \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer YOUR_JWT_TOKEN" \
  -d '{
    "title": "Spaghetti Carbonara",
    "ingredients": "400g spaghetti, 200g bacon, 4 eggs, 100g parmesan",
    "instructions": "1. Cook pasta. 2. Fry bacon. 3. Mix eggs and cheese
    .",
    "category": "Dinner",
    "prepTime": 15,
    "cookTime": 15,
    "servings": 4,
    "difficulty": "MEDIUM"
  }'
```

D Appendix D: Glossary

Term	Definition
API	Application Programming Interface - A set of rules and protocols for building software applications
CRUD	Create, Read, Update, Delete - Basic database operations
DTO	Data Transfer Object - Object used to transfer data between layers
JPA	Java Persistence API - Standard for ORM in Java
ORM	Object-Relational Mapping - Technique to map objects to database tables
REST	Representational State Transfer - Architectural style for web services
JWT	JSON Web Token - Standard for secure token-based authentication
CORS	Cross-Origin Resource Sharing - Security mechanism for web browsers
Hibernate	ORM framework implementation of JPA
Spring Boot	Framework for building Java applications with minimal configuration
React	JavaScript library for building user interfaces
Vite	Modern build tool for frontend development
Tailwind CSS	Utility-first CSS framework
PostgreSQL	Open-source relational database
Docker	Platform for containerizing applications
Kubernetes	Container orchestration platform
CI/CD	Continuous Integration/Continuous Deployment
Axios	Promise-based HTTP client for JavaScript
Lombok	Java library to reduce boilerplate code
Maven	Build automation and dependency management tool
npm	Node Package Manager for JavaScript
HikariCP	High-performance JDBC connection pool
Entity	Java class mapped to a database table
Repository	Interface for database operations
Service Layer	Business logic layer in application
Controller	Component that handles HTTP requests
Bean	Object managed by Spring container
Dependency Injection	Design pattern where dependencies are provided by framework
Annotation	Metadata added to Java code (e.g., @Entity)
Foreign Key	Database constraint linking two tables
Index	Database structure to speed up queries
Migration	Versioned database schema changes

Term	**Definition**

Table 11: Technical Glossary

E Appendix E: References and Resources

E.1 Official Documentation

- **Spring Boot:** https://spring.io/projects/spring-boot
- **Spring Data JPA:** https://spring.io/projects/spring-data-jpa
- **React:** https://react.dev
- **Vite:** https://vitejs.dev
- **Tailwind CSS:** https://tailwindcss.com
- **PostgreSQL:** https://www.postgresql.org/docs/
- **Docker:** https://docs.docker.com
- **Kubernetes:** https://kubernetes.io/docs/

E.2 Learning Resources

- **Baeldung (Spring Boot):** https://www.baeldung.com/spring-boot
- **React Tutorial:** https://react.dev/learn
- **PostgreSQL Tutorial:** https://www.postgresqltutorial.com
- **REST API Design:** https://restfulapi.net
- **Docker Tutorial:** https://docker-curriculum.com

E.3 Tools and Libraries

- **Lombok:** https://projectlombok.org
- **Axios:** https://axios-http.com
- **Hibernate:** https://hibernate.org
- **Maven:** https://maven.apache.org

E.4 Best Practices

- **Clean Code:** Robert C. Martin
- **Design Patterns:** Gang of Four
- **Effective Java:** Joshua Bloch
- **REST API Design Rulebook:** Mark Masse

F Appendix F: Project Metrics

F.1 Code Statistics

Component	**Files**	**Lines of Code**
Backend (Java)	12	800
Frontend (JavaScript/JSX)	8	400
Configuration Files	6	200
Documentation	4	500
Total	**30**	**1,900**

Table 12: Project Code Statistics

F.2 Database Statistics

Metric	**Count**
Tables	3
Columns (Total)	30
Indexes	5
Foreign Keys	2
Check Constraints	1
Unique Constraints	2

Table 13: Database Metrics

F.3 API Endpoints

Status	**Endpoints**	**Methods**
Implemented	5	GET, POST
Planned	15	GET, POST, PUT, DELETE
Total	**20**	**-**

Table 14: API Endpoint Statistics

F.4 Dependencies

Category	**Count**
Backend (Maven)	12
Frontend (npm)	8
Dev Dependencies	6
Total	**26**

Table 15: Project Dependencies

F.5 Development Timeline

Phase	**Tasks**	**Duration**
Day 1	Environment Setup	0.5 hours
Day 2	Repository Setup	0.5 hours
Day 3	Backend Development	0.5 hours
Day 4	Frontend Development	1.5 hours
Day 5	Database Integration	2.0 hours
Day 6-7	Documentation	1.0 hours
Total Week 1	**-**	**6 hours**

Table 16: Development Timeline

Document Information

Field	Value
Document Title	Smart Recipe Manager - Technical Documentation
Version	1.0
Date	November 9, 2024
Author	Hamza El Ouattab
Project Phase	Week 1 Complete
Status	Active Development
Repository	https://github.com/HAMZA-EL-OUATTAB/smart-recipe-manager
License	Educational Use

Version History

Version	Date	Changes
0.1	Nov 1, 2024	Initial project setup
0.5	Nov 5, 2024	Backend and database implementation
1.0	Nov 9, 2024	Week 1 complete - Full documentation

Table 17: Document Version History

Contact Information

- **Developer:** Hamza El Ouattab
- **GitHub:** https://github.com/HAMZA-EL-OUATTAB
- **Email:** hamza.elouattab@example.com
- **Project Repository:** https://github.com/HAMZA-EL-OUATTAB/smart-recipe-manager

This document is part of the Smart Recipe Manager project.
For the latest version, visit the GitHub repository.