FoodBlog Website — Project Documentation

Author: Naveen Dornala

Version: 1.0

Date: 2025-08-09

1. Project Overview

This document provides a comprehensive manual for the FoodBlog Website project. The goal of the project is to build a responsive, secure, and performant web application that allows users to view, create, and share recipes, comment on posts, and interact with the community. This manual contains architecture, installation steps, API specification, frontend structure, database schema, testing, deployment, and operational guidance.

1.1 Goals & Core Features

Core features include: - User authentication (signup, login, password reset) - User profiles with avatars and bio - CRUD operations for recipes (create, read, update, delete) - Search and filter by cuisine, ingredients, tags - Commenting and likes for recipes - Image upload and management (optimised) - Responsive UI (mobile-first) - Admin panel for moderation - SEO-friendly routes and metadata

2. Technology Stack

Recommended stack (example): - Frontend: HTML5, CSS3 (Flexbox/Grid), JavaScript (ES6+), React.js (optional) or Vanilla JS - Styling: SASS or Tailwind CSS (recommended for speed) - Backend: Node.js with Express.js - Database: MongoDB (Atlas or self-hosted) / PostgreSQL alternative - Authentication: JWT (JSON Web Tokens) + bcrypt for password hashing - File Storage: AWS S3 for images, with CloudFront CDN - DevOps: GitHub Actions for CI, Docker for containerization, and deployment on Heroku/Render/AWS EC2 - Monitoring: Sentry for errors, Prometheus/Grafana for metrics (optional)

3. System Architecture

The system follows a classic client-server architecture: - Client (browser or mobile) -> interacts with RESTful API endpoints - Backend (Express) -> handles authentication, business logic, and data access - Database (MongoDB) -> stores users, posts, comments, tags, and media references - Object storage (S3) -> stores image files; backend stores URL references in DB - CDN -> serves cached static assets and images for low latency A diagram (not included) would show these components and flow of requests.

4. Database Schema (Example - MongoDB)

Below are example collections and essential fields. Use indexes on frequently queried fields (slug, tags, createdAt).

Users collection (users): - _id: ObjectId - name: string - email: string (unique, indexed) - passwordHash: string - avatarUrl: string - bio: string - role: enum ('user','admin') - createdAt: Date - updatedAt: Date

Recipes collection (recipes): - _id: ObjectId - authorId: ObjectId (ref users) - title: string - slug: string (unique, URL-friendly) - content: string (HTML or Markdown) - ingredients: [string] - steps: [string] - tags: [string] - cuisine: string - prepTime: number (minutes) - cookTime: number (minutes) - servings: number - coverImageUrl: string - images: [string] - likesCount: number - commentsCount: number - createdAt, updatedAt

Comments collection (comments): - _id: ObjectId - postId: ObjectId (ref recipes) - userId: ObjectId (ref users) - content: string - createdAt

5. API Endpoints (Representative)

Authentication: - POST /api/auth/register - Register new user (name,email,password) - POST /api/auth/login - Login (email,password) -> returns JWT - POST /api/auth/forgot - Request password reset - POST /api/auth/reset - Reset password with token

Users & Profiles: - GET /api/users/:id - Get user profile - PUT /api/users/:id - Update profile (auth required)

Recipes: - GET /api/recipes - List recipes (filters: tag, cuisine, search, page) - GET /api/recipes/:slug - Get recipe by slug - POST /api/recipes - Create new recipe (auth required, multipart/form-data for images) - PUT /api/recipes/:id - Update recipe (auth, ownership) - DELETE /api/recipes/:id - Delete recipe (auth, ownership)

Comments & Likes: - POST /api/recipes/:id/comments - Add comment - GET /api/recipes/:id/comments - List comments - POST /api/recipes/:id/like - Toggle like

6. Frontend Structure

Suggested folder structure (React): /src /components Header.jsx, Footer.jsx, RecipeCard.jsx, CommentBox.jsx /pages Home.jsx, Recipe.jsx, Profile.jsx, Admin.jsx /services api.js (axios instances) /styles main.scss Use semantic HTML, ARIA attributes for accessibility, and lazy-load images using loading='lazy'.

7. Security Best Practices

Hash passwords with bcrypt (salt rounds >=10). Use HTTPS everywhere; enforce HSTS in production. Validate and sanitize inputs server-side to prevent XSS/SQL injection. Use CORS properly and only allow trusted origins. Rate-limit authentication endpoints to prevent brute-force attacks. Store JWTs in HttpOnly cookies (or use secure storage with proper XSRF protections).

8. Testing Strategy

Unit tests: Jest for backend logic, React Testing Library for frontend UI. Integration tests: Supertest for API endpoints. E2E tests: Playwright / Cypress to simulate user flows (signup, create post, comment). CI: Run tests on every PR using GitHub Actions.

9. Deployment & DevOps

Containerize app with Docker; use multi-stage builds to keep images small. Use environment variables for secrets (do not commit .env). CI/CD: GitHub Actions workflow to build, test, and deploy to Heroku/Render/AWS/Netlify. Use a managed database (MongoDB Atlas) for production; setup automated backups. Monitor logs and errors with Sentry; set up alerts for downtime.

10. Performance & SEO

Optimize images: serve WebP, use responsive srcset, compress images. Minify and bundle JS/CSS; enable long-term caching headers for static assets. Use SSR (Next.js) or prerendering for critical pages if SEO is important. Add meta tags, Open Graph tags, and structured data (JSON-LD) for recipe schema.

11. Local Setup (Quick Start)

1. Clone repo: git clone 2. Install dependencies: npm install 3. Setup .env with DB_URI, JWT_SECRET, AWS_S3 credentials, PORT 4. Run dev server: npm run dev 5. Build & run production: npm run build && npm start

12. Useful Code Snippets

Password hashing (Node.js / bcrypt): const bcrypt = require('bcrypt'); const hash = await bcrypt.hash(password, 12);

Express middleware: error handler example: function errorHandler(err, req, res, next) { console.error(err.stack); res.status(500).json({ error: 'Internal server error' }); }

Sample recipe POST route (Express): app.post('/api/recipes', authMiddleware, upload.single('image'), async (req, res) => { const { title, ingredients, steps } = req.body; // validate, save image to S3, create recipe in DB });

13. Appendices & Recommendations

Consider user analytics (Google Analytics or Plausible) to track engagement. Add social login (Google, Facebook) to simplify onboarding. For large-scale, introduce pagination and caching layers (Redis). Keep documentation updated with architecture changes and API revisions.

Credits & Disclaimer

This document was generated to help structure your FoodBlog project and provide an advanced manual. Please review and adapt any code snippets to your project's specific requirements. Ensure you secure API keys and follow best practices for production deployments.