Key Features for Sprint 1

Requirement 1.0: Database integration for persistent storage (MongoDB or PostgreSQL)

Feature Overview: Replace in-memory/local JSON persistence with a real database to ensure user data, saved recipes, and shared collections persist across server restarts and scale with usage.

Service Layer and Integration: Introduce a database layer used by services (users, saved recipes, collections) and ensure all reads/writes go through it.

Functionality:

- The application uses a database for persistence instead of JSON files.
- User accounts and saved recipes persist across server restarts.
- Shared collections persist with correct membership and recipe associations.

Requirement 1.1: Select the database and hosting approach

Feature Overview: Decide between MongoDB and PostgreSQL and document the rationale and hosting plan.

Structure: Choose DB type, hosting method (local dev + hosted env), and connection strategy (env vars, secrets, CI/CD config).

Functionality:

- Database choice is documented with reasoning (schema fit, team familiarity, hosting simplicity).
- Hosting approach is defined for dev + deployment (e.g., local container + managed cloud).
- Connection variables/config are standardized and reproducible across machines.

Requirement 1.2: Create database connection layer

Feature Overview: Implement a reusable connection module and ensure all services can access the DB reliably.

Structure: Add connection initialization, config loading, lifecycle handling (startup/shutdown), and health checks.

Functionality:

- Backend starts and connects to DB successfully in local and deployed environments.
- DB connection configuration is driven by environment variables.
- Failures to connect produce clear error logs and standardized API error responses.

Requirement 1.3: Migrate users and saved recipes into the database

Feature Overview: Move existing persisted data (in-memory/localStorage assumptions and/or JSON) into the DB so users don't lose progress.

Structure: Define migration script or one-time import flow, map legacy fields → DB schema, validate results.

Functionality:

- Existing users and saved recipes are stored in DB format.
- Migration prevents duplicate records and preserves identifiers where needed.
- Post-migration, all user/saved recipe operations read/write from DB only.

Requirement 2.0: Move collection storage from JSON to database

Feature Overview: Replace file-based JSON storage for collections with persistent DB storage.

Structure: Implement collection repository/DAO, update collection creation/join/edit flows to use DB.

Functionality:

- Creating, joining, updating, and deleting collections persists in DB.
- Collection membership remains correct after server restart.
- Shared collection data stays consistent across all members.

Requirement 2.1: Create database schema for shared collections

Feature Overview: Formalize how collections, membership, and recipe mappings are represented in the DB.

Structure: Define entities/tables/collections and relationships (membership, permissions, recipe references).

Functionality:

- Collections store: name, owner, created timestamp, etc.
- Membership supports multi-user access and role/permission fields (as needed).
- Recipes can belong to both personal saved lists and shared collections without conflict.

Requirement 2.2: Update collection service layer to use the DB schema

Feature Overview: Refactor service logic so collection operations use DB-backed queries rather than JSON operations.

Service Layer and Integration: Ensure service functions enforce authorization and maintain data integrity (no duplicate recipe entries, consistent membership).

Functionality:

- Adding/removing recipes updates DB state correctly.
- Duplicate prevention is enforced at the data layer and/or service layer.
- Authorization checks prevent non-members from editing collections.

Requirement 3.0: Basic API error handling improvements (standardization)

Feature Overview: Standardize how backend errors are returned so the frontend can display meaningful messages consistently.

Structure: Centralize error formatting, include HTTP status codes, error codes, and readable messages.

Functionality:

- Backend returns consistent JSON error shapes across endpoints.
- Errors include: status code, message, and a stable error identifier/code.
- Common failures (validation, not found, unauthorized, DB failure) are handled cleanly.

Requirement 3.1: Add standardized error responses across services

Feature Overview: Replace scattered `try/catch` responses with shared helpers/middleware so behavior is uniform.

Structure: Add error middleware (or shared response builder), map known errors to status codes.

Functionality:

- Similar failures produce identical response formats across endpoints.
- Logs include useful debugging info without leaking sensitive internals to the client.
- Unknown/unhandled errors still return a safe standardized response.

Requirement 3.2: Add frontend error display for API failures

Feature Overview: Ensure frontend communicates errors clearly (toast/banner/inline field error) rather than failing silently.

UI and Flow: Add a reusable error component or notification pattern and integrate into affected screens.

Functionality:

- Users see a clear error message when API requests fail.
- Validation vs server errors are differentiated (where applicable).
- Error states don’t break the UI (loading spinners stop, users can retry).