Review Document for Pet Name Generator

Review: Functional & Non-Functional Requirements

Format

The requirements are well-structured in a clear tabular format with consistent ID naming (FR001-FR005, NFR001-NFR005). Each requirement includes a descriptive name and detailed description.

Clarity

Both functional and non-functional requirements are written clearly and unambiguously. The functional requirements use "shall" statements appropriately, defining specific system behaviors. Non-functional requirements include measurable criteria (e.g., "within 2 seconds", "95% uptime", "1,000 unique pet names").

Completeness

The functional requirements cover the core features well, but some gaps exist. Missing: data persistence details, error handling beyond invalid animal types, how popularity metrics are calculated/updated, and what happens when the database is empty. Non-functional requirements lack coverage of security (data privacy, SQL injection protection), maintainability, scalability beyond data storage (concurrent users?), and accessibility standards.

Review: User Stories

Format

User stories follow a consistent structure with clear "As a/I want to/so that" statements and well-organized acceptance criteria using Given-When-Then format. Scenario numbering and bullet formatting are consistent throughout.

Acceptance Criteria Clarity

The scenarios are comprehensive and testable. They cover happy paths, edge cases, and error conditions effectively. Each scenario is specific about preconditions, actions, and expected outcomes. The use of "And" statements properly chains multiple conditions.

Alignment with Functional Requirements:

Strong alignment overall. US001 maps to FR001 (Name Generation), US002 to FR002 (Animal Type Filtering), US003 to FR003 (Batch Generation), and US004 to FR004 (Generation History).

However, FR005 (Popular Names Display) has no corresponding user story—this is a gap that should be addressed with a fifth user story describing how users view popular/trending names.

Initial Database Schema

Table: pet_names

- id (Primary Key, Serial/Auto-increment)
- name (VARCHAR, NOT NULL)
- animal_type (VARCHAR, e.g., 'Dog', 'Cat', 'Bird', 'Fish', 'Hamster', 'Rabbit')
- origin (VARCHAR, optional - cultural background)
- meaning (TEXT, optional - name meaning/description)
- created_at (TIMESTAMP, default NOW())

Table: generation_history

- id (Primary Key, Serial/Auto-increment)
- pet_name_id (Foreign Key → pet_names.id)
- generated_at (TIMESTAMP, default NOW())
- session_id (VARCHAR, optional - to track user sessions)

Table: name_popularity

- id (Primary Key, Serial/Auto-increment)
- pet_name_id (Foreign Key → pet_names.id, UNIQUE)
- generation_count (INTEGER, default 0)
- last_generated_at (TIMESTAMP)

Example API Endpoints

Frontend Routes (Express serving static files):

- GET / - Home page with name generator interface
- GET /recent - Recent names page
- GET /popular - Popular names page

Backend API Endpoints:

Name Generation

POST /api/names/generate
Request body: { "animalType": "Dog", "count": 1 }
Response: { "names": ["Buddy"], "success": true }

POST /api/names/generate-batch
Request body: { "animalType": "Cat", "count": 10 }
Response: { "names": ["Whiskers", "Shadow", ...], "success": true }

Recent Names

GET /api/names/recent?limit=10
Response: { "names": [{id, name, animalType, generatedAt}, ...], "success": true }

Popular Names

GET /api/names/popular?limit=10
Response: { "names": [{id, name, generationCount, animalType}, ...], "success": true }

Animal Types

GET /api/animal-types
Response: { "types": ["Dog", "Cat", "Bird", "Fish", "Hamster", "Rabbit"], "success": true }