Planning Document

Technology Stack

- Backend Framework: Flask
- ORM: SQLAlchemy
- Serialization/Validation: Marshmallow
- Database: PostgreSQL

Overview

The goal is to develop a web api system that supports a pet adoption managemnt platform. The API will allow users to view available pets, complete adoption and create and update users/details. Core functionality will include pet listing and tracking of adoption history.

Validation and sanitisation will be handled using Marshmallow schemas to enforce types, required fields, and clean inputs before persisting to the database

ORM functionality will be handled using SQLAlchemy to manage database relationships using python models.

Database Schema

Users Table

- id (Primary Key)
- name
- email

Pets Table

- id (Primary Key)
- name
- species
- breed
- age
- description

- status

Adoptions Table

- id (Primary Key)
- user_id (Foreign Key → Users.id)
- pet_id (Foreign Key → Pets.id)
- adoption_date

Pet owners Table

- id (Primary Key)
- user_id (Foreign Key → Users.id)
- pet_id (Foreign Key → Pets.id)

- **Defined Relationships**
- A user may adopt multiple pets
- A pet can be owned by multiple owners

Database Normalisation (3NF)

The schema adheres to **Third Normal Form (3NF)**:

- All attributes are atomic (1NF) -> e.g breed is a single value and not a list.
- Every non-key attribute is fully dependent on the primary key (2NF) -> e.g., email depends on user_id, not on name.
- No transitive dependencies (3NF) -> e.g No names stored in adoptions table as they exist in the users and pets table and can be joined when needed

How the Database structure supports full CRUD functionality

The database structure is designed to support full CRUD functionality across all key entities:

- Users: Can be created during registration, updated with personal or address details, and deleted if needed. The users table enables direct user management.

- Pets: List all pets (read), add a new pet to the system (create), updated info (patch, put), or deleted(DELETE). The status field allows filtering pets that are available for adoption.
- Adoptions: Each adoption is recorded with user_id and pet_id, supporting creation of historical records. Adoptions can also be viewed or updated if needed
- PetOwners: To be removed based on feedback and replaced with one -> many relationship between pets and owners(users)

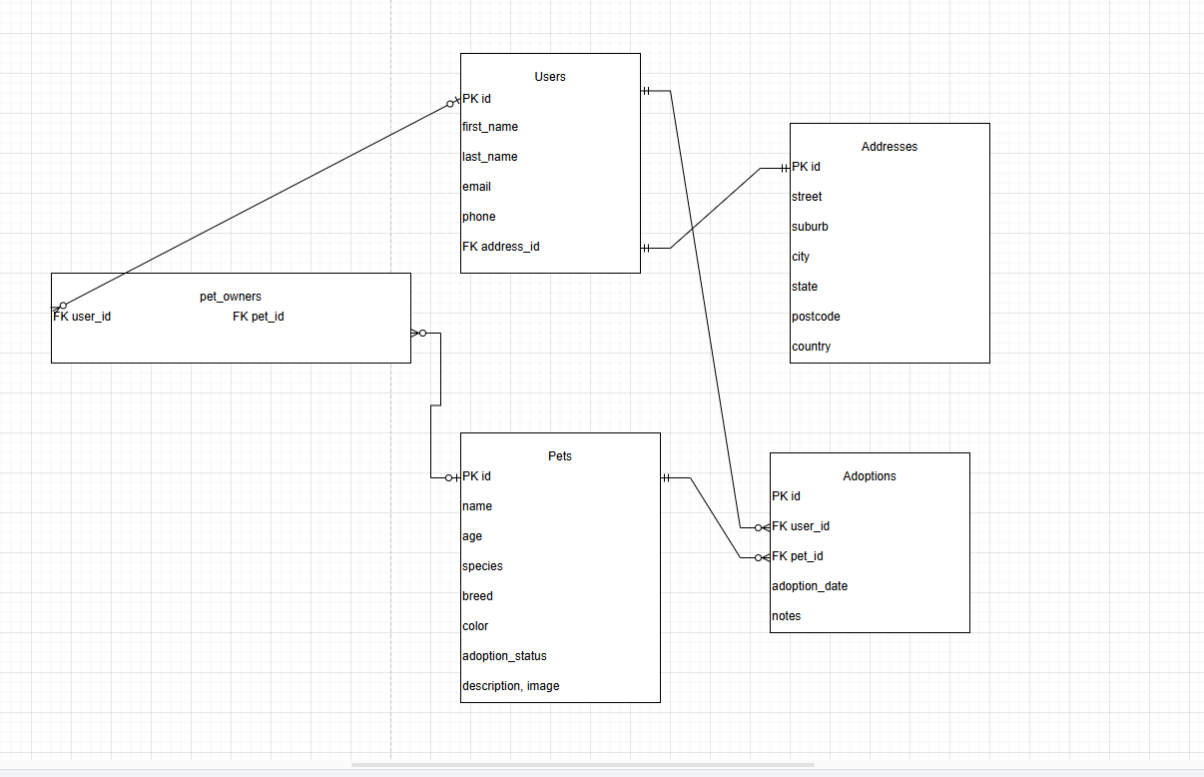
Chosen Database Technology: PostgreSQL

PostgreSQL was chosen for its reliability and robust support for relational data. It offers enforcement of data integrity through constraints and foreign keys, making it useful for managing relationships between users, pets, and adoptions. Additionally, it integrates seamlessly with SQLAlchemy. Additionally, it provides an ability to scale as the user base or pet database grows.

Simple Comparison: PostgreSQL vs MySQL

Feature	PostgreSQL	MySQL
Type	Relational (more advanced features)	Relational (basic features)
Data Checks	Very strong rules to keep data correct	Good rules, but more flexible
Data Safety	Always safe and consistent	Also safe and consistent
SQL Rules	Follows SQL rules very closely	Follows most SQL rules
Extra Features	Can add custom types and use advanced tools	Fewer extra tools
Speed	Better for complex tasks	Faster for simple tasks
Best Use	Better for strict complex apps like pet adoption	Good for smaller or simpler apps or websites

ERD Diagram



Feedback & Iteration Log

Planning Phase

- The ERD had both pet_owners and adoptions tables, which created confusion about their roles.
  Action: Removed the pet_owners table and added a user_id field to the pets table to show current ownership. Kept adoptions to track adoption history.
- The pets table included optional fields like image and description, which weren't always necessary.
  Action: Left these fields as optional. Sometimes a pet will be listed before full details are available.
- The field 'status' was unclear in meaning.
  Action: Renamed status to is_adopted and changed the type to Boolean for clarity.
- PostgreSQL was suggested as a good choice.
  Action: Kept PostgreSQL and added more reasoning to the documentation (e.g. strong foreign key support and good for relational data).

Planning Phase – Feedback Round 2

- Repetition was noticed between pet_owners and adoptions.
  Action: Already handled in round 1. Ownership is now shown using user_id in the pets table only.
- Field atomicity was questioned for items like breed and status.
  Action: Reviewed the database and confirmed all fields are atomic. breed is a single value, and status was already updated to is_adopted.
- More info was needed on how CRUD operations are supported.
  Action: Wrote a short explanation in the documentation showing how Create, Read, Update, and Delete are implemented across users, pets, addresses, and adoptions.

Final Feedback – After Initial Build

- The adoption logic didn't stop a pet from being adopted multiple times. It only checked if the same user already owned the pet.
  Action: Fixed this by updating the POST /adoptions endpoint to check if the pet is already marked as adopted (is_adopted = True) or already linked to a user. Now blocks adoption in both cases.

Final Changes Made Based on Feedback

- Removed pet_owners table
- Renamed status to is_adopted (Boolean)
- Updated adoption logic to prevent multiple owners
- Expanded PostgreSQL justification
- Added explanation of CRUD functionality