Dog Rescue Project: Understanding Our Database

Welcome to the Dog Rescue project! This document helps you understand how we store and manage all the important information for our application. Think of it as the blueprint for our digital filing cabinet.

Our project uses a type of database called **MongoDB**. It's a **NoSQL** database, which means it's a bit different from traditional databases you might have heard of (often called **SQL** or relational databases). We'll explain more about that later!

What Information Do We Store? (Our Collections)

In MongoDB, we organize data into "collections." You can think of a collection as a folder where we keep similar types of information. Our project has four main collections:

1. **`Dogs`**: This is where we keep all the details about each dog in our care.
    - *Examples*: Name, breed, age, photos, personality, and whether they are available for adoption.
2. **`RescueSubmissions`**: When someone finds a dog that needs help, they fill out a form. This collection stores all those submitted rescue requests.
    - *Examples*: Where the dog was found, a description of the dog, and the contact details of the person who found it.
3. **`Admins`**: This collection holds information about the people who have special access to manage the website and its data (the administrators).
    - *Examples*: Admin usernames, names, and email addresses.
4. **`Volunteers`**: People can apply to help us out! This collection stores applications from potential volunteers.
    - *Examples*: Applicant names, contact info, what kind of help they can offer (like dog walking or fostering), and when they are available.

How is Our Database (MongoDB/NoSQL) Different from SQL Databases?

You might be familiar with SQL databases (like MySQL, PostgreSQL, or SQL Server). These are also called **relational databases**. Our MongoDB database is a **NoSQL** database (specifically, a "document database"). Here's a simple breakdown of the key differences and why we might choose NoSQL for this project:

1. Structure of Data:

- **SQL (Relational) Databases:**
    - Think of data stored in tables with rows and columns, like a very organized spreadsheet.
    - Each row is a record, and each column is a specific piece of information (e.g., a `Dogs` table might have columns for `DogID`, `Name`, `Breed`, `Age`).
    - The structure (schema) is usually fixed. You define the columns and their data types (like number, text, date) beforehand. Changing it later can be complex.
    - Relationships between tables are clearly defined (e.g., a `Dog` might be linked to an `Owner` in a separate `Owners` table via an `OwnerID`).
- **MongoDB (NoSQL Document Database):**
    - Think of data stored in "documents" (which are like flexible, self-contained information packets). These documents are typically in a format called JSON (JavaScript Object Notation), which is very human-readable.
    - Each document in a collection can have a slightly different structure. For example, one dog document might have a `favoriteToy` field, while another might not. This is called a **flexible schema**.

- Instead of joining tables, related data can often be embedded directly within a document. For instance, a `Dog` document might contain an array of `medicalRecords` directly, rather than linking to a separate `MedicalRecords` table.

2. Scalability (Growing the Database):

- **SQL Databases:**
  - Often scaled "vertically," meaning you make the server more powerful (more RAM, faster CPU). This can get expensive.
  - Scaling "horizontally" (spreading data across multiple servers) can be more complex to set up and manage.
- **MongoDB (NoSQL):**
  - Designed to scale "horizontally" more easily. You can add more servers to distribute the data and workload, which can be more cost-effective for large amounts of data or high traffic.

3. Flexibility and Development Speed:

- **SQL Databases:**
  - The need to define a strict schema upfront can sometimes slow down initial development if requirements are changing rapidly.
- **MongoDB (NoSQL):**
  - The flexible schema is great for agile development where project requirements might evolve. You can add new fields or change data structures more easily without major database overhauls.
  - Often preferred for applications where data is diverse or unstructured.

Why Did We Choose MongoDB (NoSQL) for the Dog Rescue Project?

For a project like ours, MongoDB offers several advantages:

- **Flexibility for Dog Profiles:** Dogs have varied information. Some might have extensive medical histories, others might have detailed behavioral notes, and some might have come in as strays with little known background. MongoDB's flexible documents handle this variety well.
- **Ease of Development:** When adding new features (like tracking a dog's favorite toys or dietary restrictions), we can easily add new fields to the `Dog` documents without restructuring a rigid table.
- **Handling User Submissions:** Rescue submissions and volunteer applications can also vary. A NoSQL approach can simplify storing this diverse user-generated content.
- **Scalability (Future Growth):** If our rescue organization grows and we have many more dogs, users, and submissions, MongoDB is designed to scale out to handle the increased load.

In a Nutshell:

Feature	SQL Databases (Relational)	MongoDB (NoSQL Document Database)
Data Model	Tables with rows & columns (like spreadsheets)	Collections of JSON-like documents
Schema	Strict, defined upfront	Flexible, dynamic
Relationships	Via table joins (foreign keys)	Embedding data, or simple references
Scalability	Primarily vertical (bigger server)	Primarily horizontal (more servers)
Best For	Structured data, complex transactions, data integrity	Unstructured/semi-structured data, rapid development, scalability

Both SQL and NoSQL databases are powerful tools, and the best choice depends on the project's specific needs. For the Dog Rescue application, MongoDB provides the flexibility and scalability that benefits our development and future growth.

Detailed Look at Our Collections (The Schemas)

Below, we describe what specific pieces of information (called "fields") are stored in each document within our collections. This is defined in files called "models" (like `Dog.ts`, `RescueSubmission.ts`, etc.) in our project's code.

1. `Dogs` Collection (from `Dog.ts`)

This is where we store all the details about each dog.

```
// This is a simplified view of how a Dog's information is structured.
// The actual code uses 'mongoose' to define this structure for MongoDB.

interface IDog {
  name: string;          // The dog's name
  breed: string;         // The dog's breed (e.g., "Labrador Retriever", "Mixed")
  age: string;           // How old the dog is (e.g., "2 years", "6 months", "Adult")
  size: string;          // Size category (e.g., 'Small', 'Medium', 'Large')
  gender: string;        // 'Male' or 'Female'
  image: string;         // A web link (URL) to a picture of the dog
  description: string;   // A paragraph describing the dog's personality, history, etc.
  tags: string[];        // Keywords to help describe the dog (e.g., ["friendly", "good with kids"])
  rescueId?: string;     // If the dog came from a rescue submission, this is its ID (optional)
  status?: string;       // Current status (e.g., 'available', 'adopted', 'fostered')
  // createdAt, updatedAt: Dates automatically added to track when the dog's info was added or last changed.
}
```

Key Information Stored for Each Dog:

- `name`: (Text, Required) - The dog's name.
- `breed`: (Text, Required) - The dog's breed.
- `age`: (Text, Required) - The dog's age (e.g., "2 years", "Puppy").
- `size`: (Text, Required, Must be one of: `'Small'`, `'Medium'`, `'Large'`) - The general size of the dog.
- `gender`: (Text, Required, Must be one of: `'Male'`, `'Female'`) - The dog's gender.
- `image`: (Text, Required) - A web link to the dog's picture.
- `description`: (Text, Required) - A detailed story or notes about the dog.
- `tags`: (List of Text, Optional) - Short descriptive words (like "playful", "shy").
- `rescueId`: (Text, Optional) - If this dog was entered into the system via a rescue submission, this links to that submission.
- `status`: (Text, Optional, Must be one of: `'available'`, `'adopted'`, `'fostered'`, `'pending'`, Default: `'available'`) - The dog's current adoption status.
- `createdAt`, `updatedAt`: (Date/Time) These are automatically added to show when the dog's record was created and last updated.

---

2. `RescueSubmissions` Collection (from `RescueSubmission.ts`)

This collection stores forms submitted by people who have found a dog needing rescue.

```
// Simplified structure for a Rescue Submission
interface IRescueSubmission {
  name?: string;          // Name of the dog, if the finder knows it (optional)
  breed?: string;         // Breed of the dog, if known (optional)
  gender: string;        // 'Male', 'Female', or 'Unknown'
  age: string;           // Approximate age of the dog
  size: string;          // Approximate size
  location: string;      // Where the dog was found
  description: string;   // Details about the dog and the situation
  contactName: string;   // Name of the person who found the dog
  contactEmail: string;  // Email of the finder (must be a valid email format)
  contactPhone: string;  // Phone number of the finder
  imageUrls?: string[];   // Links to any photos the finder took (optional)
  status: string;        // Status of the submission (e.g., 'pending', 'rescued', Default: 'pending')
  submittedAt: Date;     // When the form was submitted (defaults to now)
  // createdAt, updatedAt: Dates automatically added for record tracking.
}
```

Key Information Stored for Each Rescue Submission:

- `name`: (Text, Optional) - The found dog's name, if known.
- `breed`: (Text, Optional) - The found dog's breed, if known.
- `gender`: (Text, Required, Must be one of: `'Male'`, `'Female'`, `'Unknown'`).
- `age`: (Text, Required) - Approximate age.
- `size`: (Text, Required) - Approximate size.
- `location`: (Text, Required) - Where the dog was found.
- `description`: (Text, Required) - Details about the dog and why it needs rescue.
- `contactName`: (Text, Required) - Finder's name.
- `contactEmail`: (Text, Required, Must be a valid email).
- `contactPhone`: (Text, Required) - Finder's phone number.
- `imageUrls`: (List of Text, Optional) - Links to photos of the dog.
- `status`: (Text, Required, Must be one of: `'pending'`, `'processing'`, `'rescued'`, `'closed'`, Default: `'pending'`).
- `submittedAt`: (Date/Time, Default: current time) - When the submission was made.
- `createdAt`, `updatedAt`: (Date/Time) Automatically added for tracking.

---

3. `Admins` Collection (from `Admin.ts`)

This stores information for users who can manage the website (administrators).

```
// Simplified structure for an Admin user
interface IAdmin {
  username: string;      // Unique username for login
  password_hashed: string; // The admin's password (it's stored securely, not as plain text!)
  name: string;          // Admin's full name
  email: string;         // Admin's email (must be unique and valid)
  role: string;          // Should always be 'admin'
  status: string;        // Account status (e.g., 'approved', 'pending')
  // createdAt, updatedAt: Dates automatically added.
}
```

Key Information Stored for Each Admin:

- `username`: (Text, Required, Must be unique) - The username an admin uses to log in.
- `password`: (Text, Required, Min. 8 characters) - The admin's password. *Important: This is stored in a "hashed" (scrambled) format for security, so no one can see the actual password.*
- `name`: (Text, Required) - The admin's full name.
- `email`: (Text, Required, Must be unique and a valid email).
- `role`: (Text, Default: `'admin'`) - Defines the user type.
- `status`: (Text, Default: `'approved'`, Must be one of: `'approved'`, `'pending'`) - Status of the admin account.
- `createdAt`, `updatedAt`: (Date/Time) Automatically added.
- *Security Note*: The system includes a secure way to check passwords without ever revealing the stored hashed password.

---

4. `Volunteers` Collection (from `Volunteer.ts`)

This collection holds applications from people wanting to volunteer.

```
// Simplified structure for a Volunteer application
interface IVolunteer {
  name: string;          // Applicant's full name
  email: string;         // Applicant's email (must be valid)
  phone: string;         // Applicant's phone number
  volunteerType: string; // What they want to help with (e.g., 'Dog Walker', 'Foster Parent')
  availability: string;  // When they are available (e.g., 'Weekends', 'Flexible')
  experience: string;    // Description of their experience with animals
  message?: string;      // An optional message from the applicant
  status: string;        // Application status (e.g., 'pending', 'approved', Default: 'pending')
  submittedAt: Date;     // When the application was submitted (defaults to now)
  // createdAt, updatedAt: Dates automatically added.
}
```

Key Information Stored for Each Volunteer Application:

- `name`: (Text, Required) - Applicant's name.
- `email`: (Text, Required, Must be a valid email).
- `phone`: (Text, Required) - Applicant's phone number.
- `volunteerType`: (Text, Required, Must be one of: `'Dog Walker'`, `'Foster Parent'`, `'Event Helper'`, `'Kennel Assistant'`, `'Other'`).
- `availability`: (Text, Required, Must be one of: `'Weekdays'`, `'Weekends'`, `'Evenings'`, `'Mornings'`, `'Flexible'`).
- `experience`: (Text, Required) - Details about their experience.
- `message`: (Text, Optional) - Any additional message.
- `status`: (Text, Required, Must be one of: `'pending'`, `'approved'`, `'rejected'`, Default: `'pending'`).
- `submittedAt`: (Date/Time, Default: current time) - When the application was sent.
- `createdAt`, `updatedAt`: (Date/Time) Automatically added.

This revised documentation should give everyone, especially beginners, a clearer understanding of our project's database! For the most precise technical details, developers can always refer to the actual model files in `backend/src/models/`.