**DESIGN DOCUMENT ONLINE PET ADOPTION SYSTEM**

A Thesis Project Presented to the Faculty of Datamex College of Saint Adeline, Inc.

In Partial Fulfillment of the Requirements for the

Degree of Bachelor of Science in Information Technology

**By:**

**Chavas, Mary Shelireen D.**

**INTRODUCTION**

This document presents the design proposal for the Online Pet Adoption System, which aims to provide an efficient and accessible platform for facilitating pet adoption. It discusses the system’s objectives, essential features, and design considerations to ensure functionality, usability, and reliability. Furthermore, it serves as a reference for developers, researchers, and stakeholders in guiding the system’s development process.

The purpose of this document is to present the design proposal for the Online Pet Adoption System. It provides a comprehensive overview of the system’s objectives, features, and design considerations, serving as a reference for developers, designers, and stakeholders throughout the development lifecycle.

**Overview of the Software System Being Designed**

The Online Pet Adoption System is a web-based platform that connects animal shelters and rescue organizations with potential adopters. The system enables users to browse available pets, view detailed profiles, and submit adoption applications online. Administrators and shelter staff can manage pet listings, review and process applications, and communicate with adopters. The system aims to simplify the adoption process, increase pet visibility, and promote responsible pet ownership.

**Scope of the Design Document**

This design document covers the following aspects of the Online Pet Adoption System:

User interface and user experience (UI/UX) design

System architecture and main components

Database structure and data flow

User roles and permissions0

Key features and functional requirements

Design considerations for security, scalability, and accessibility

**SYSTEM ARCHITECTURE**

**Overview of the System Architecture**

The Online Pet Adoption System is designed as a web-based, client-server application. It consists of several high-level components that interact to provide a seamless experience for both end-users and administrators.

**High-Level Components and Their Interactions**

**Frontend (Client)**

Provides the user interface for adopters, administrators, and shelter staff.

Built using HTML, CSS, and JavaScript, rendered in the user’s web browser.

Communicates with the backend via HTTP requests (RESTful APIs).

**Backend (Server)**

Handles business logic, data processing, and security.

Exposes RESTful API endpoints for the frontend to interact with.

Manages authentication, authorization, and user sessions.

**Database**

Stores persistent data such as user accounts, pet profiles, adoption applications, and status updates.

Interacts with the backend for data retrieval and storage.

**Static Assets**

Includes images, stylesheets, and JavaScript files served to the client.

**DATABASE DESIGN**

**Entity-Relationship Diagram (ERD)**

Below is a textual representation of the ERD for the Online Pet Adoption System. If you’d like, I can also generate a visual diagram.

**Entities**

**User**

**Pet**

**Adoption Application**

**Pet Image** (for multiple images per pet)

**Relationships**

A **User** can submit multiple **Adoption Applications**.

A **Pet** can have multiple **Adoption Applications**.

A **Pet** can have multiple **Pet Images**.

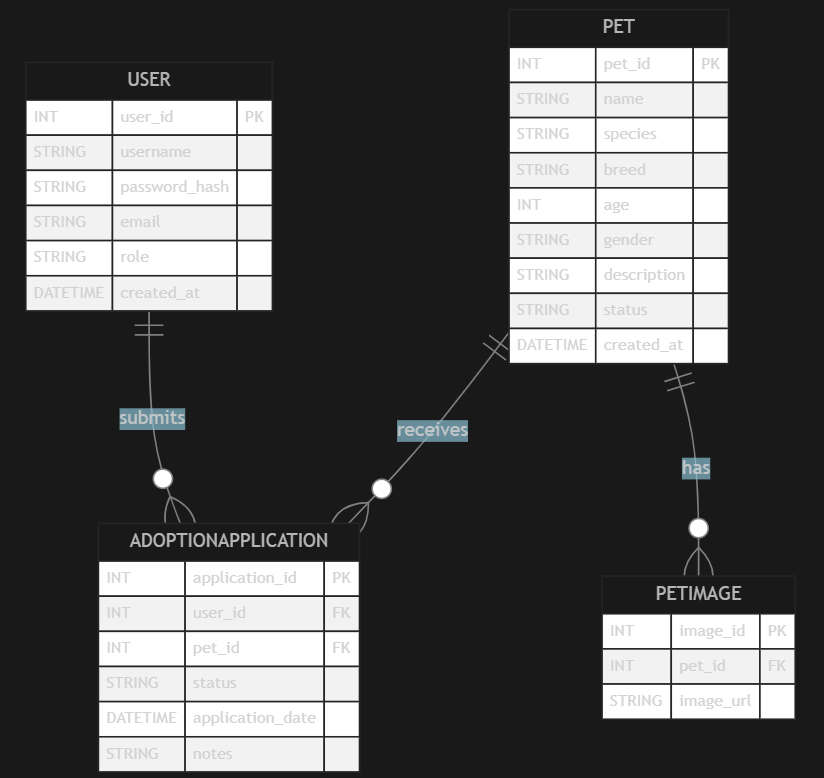
|  |  |  |
| --- | --- | --- |
| **Field** | **Type** | **Description** |
| user\_id | INTEGER PK | Unique identifier |
| username | TEXT | User’s login name |
| password\_hash | TEXT | Hashed password |
| email | TEXT | User’s email address |
| role | TEXT | ‘adopter’, ‘admin’, ‘shelter’ |
| created\_at | DATETIME | Account creation timestamp |
| pet\_id | INTEGER PK | Unique identifier |
| name | TEXT | Pet’s name |
| species | TEXT | Species (dog, cat, etc.) |
| age | INTEGER | Age in years |
| gender | TEXT | Gender |
| description | TEXT | Description/bio |
| status | TEXT | ‘available’, ‘adopted’, etc. |
| created\_at | DATETIME | Listing creation timestamp |

**Data Normalization Techniques Used**

**1st Normal Form (1NF)** All tables have atomic values; no repeating groups or arrays.

**2nd Normal Form (2NF):** All non-key attributes are fully functionally dependent on the primary key.

**3rd Normal Form (3NF)** No transitive dependencies; all fields depend only on the primary key.

**Foreign Keys** Used to maintain referential integrity between tables (e.g., user id in Adoption Application references User).

**COMPONENT DESIGN**

**Description of Key System Components/Modules**

1. **Frontend (User Interface)**

Provides web pages for adopters, administrators, and shelter staff.

Handles user input, forms validation, and displays data from the backend.

Technologies: HTML, CSS, JavaScript (optionally a framework like React or Vue).

1. **Backend (Application Server)**

Implements business logic, authentication, and authorization.

Processes requests from the frontend and interacts with the database.

Technologies: Python (Flask/Django), Node.js, etc.

1. **Database**

Stores persistent data: users, pets, applications, images.

Ensures data integrity and supports queries from the backend.

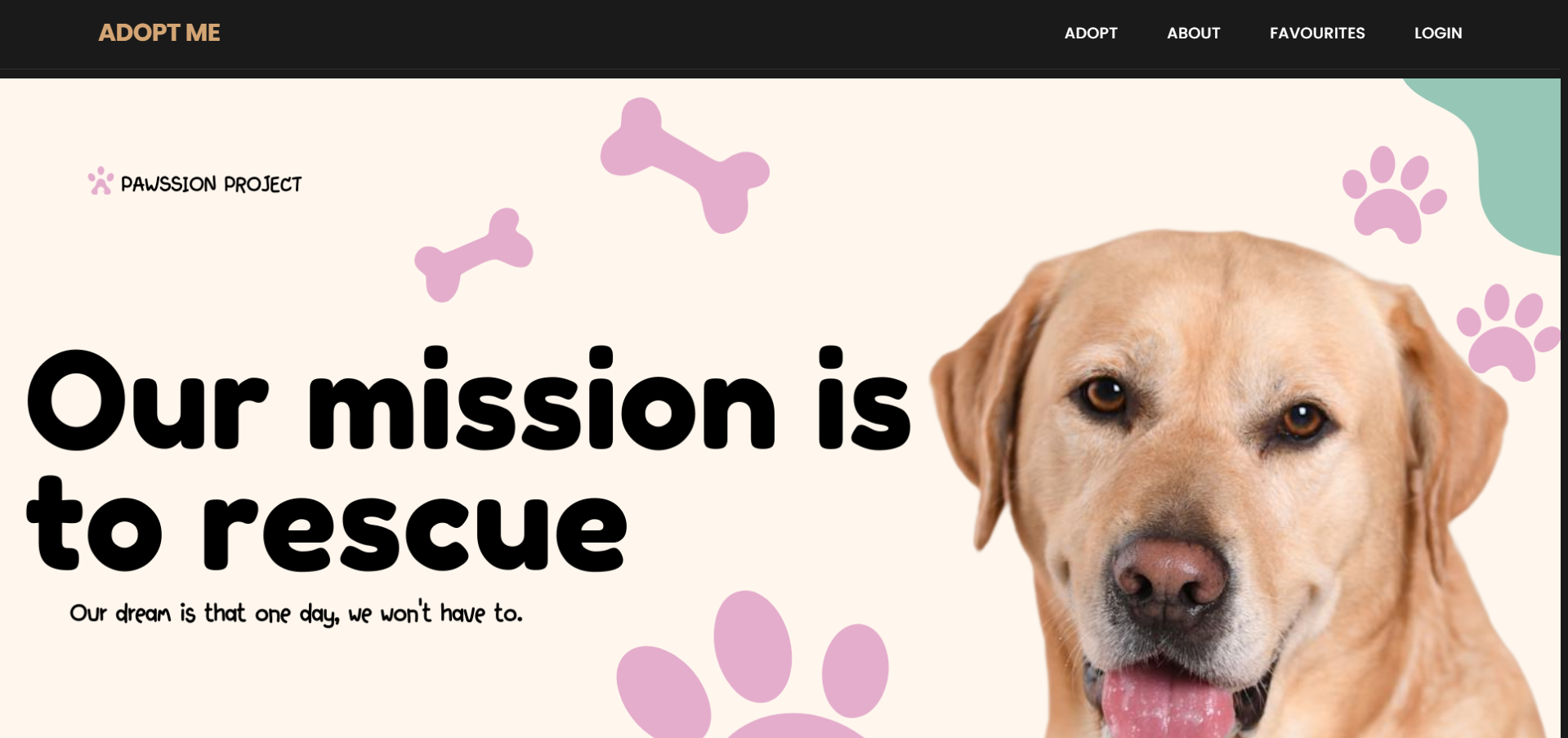
Technologies: SQLite, PostgreSQL, MySQL, etc.

1. **Static File Server**

Serves images, CSS, and JavaScript files to the frontend.

May be integrated with the backend or handled by a CDN.

**Interface Specifications for Each Component**

****

**Figure 1. Main Dashboard**

**1. Frontend ↔ Backend (API Endpoints)**

**User Authentication**

POST /api/login — User login

POST /api/register — User registration

POST /api/logout — User logout

**Pet Management**

GET /api/pets — List all pets

GET /api/pets/<pet\_id> — Get pet details

POST /api/pets — Add new pet (admin)

PUT /api/pets/<pet\_id> — Edit pet (admin)

DELETE /api/pets/<pet\_id> — Remove pet (admin)

**Adoption Applications**

POST /api/applications — Submit application

GET /api/applications — List user’s applications

GET /api/applications/<application\_id> — Application details

PUT /api/applications/<application\_id> — Update status (admin)

**2. Backend ↔ Database (Function Signatures)**

get\_user\_by\_id (user\_id) → User

create\_pet (pet\_data) → Pet

get\_all\_pets() → List [Pet]

submit\_application (user\_id, pet\_id, notes) → Application

get\_applications\_by\_user (user\_id) → List [Application]

update\_application\_status (application\_id, status) → None

**3. Static File Server**

Serves files via URLs like /static/images/<filename>, /static/css/<filename>, etc.

**Dependency Management and Interaction Between Components**

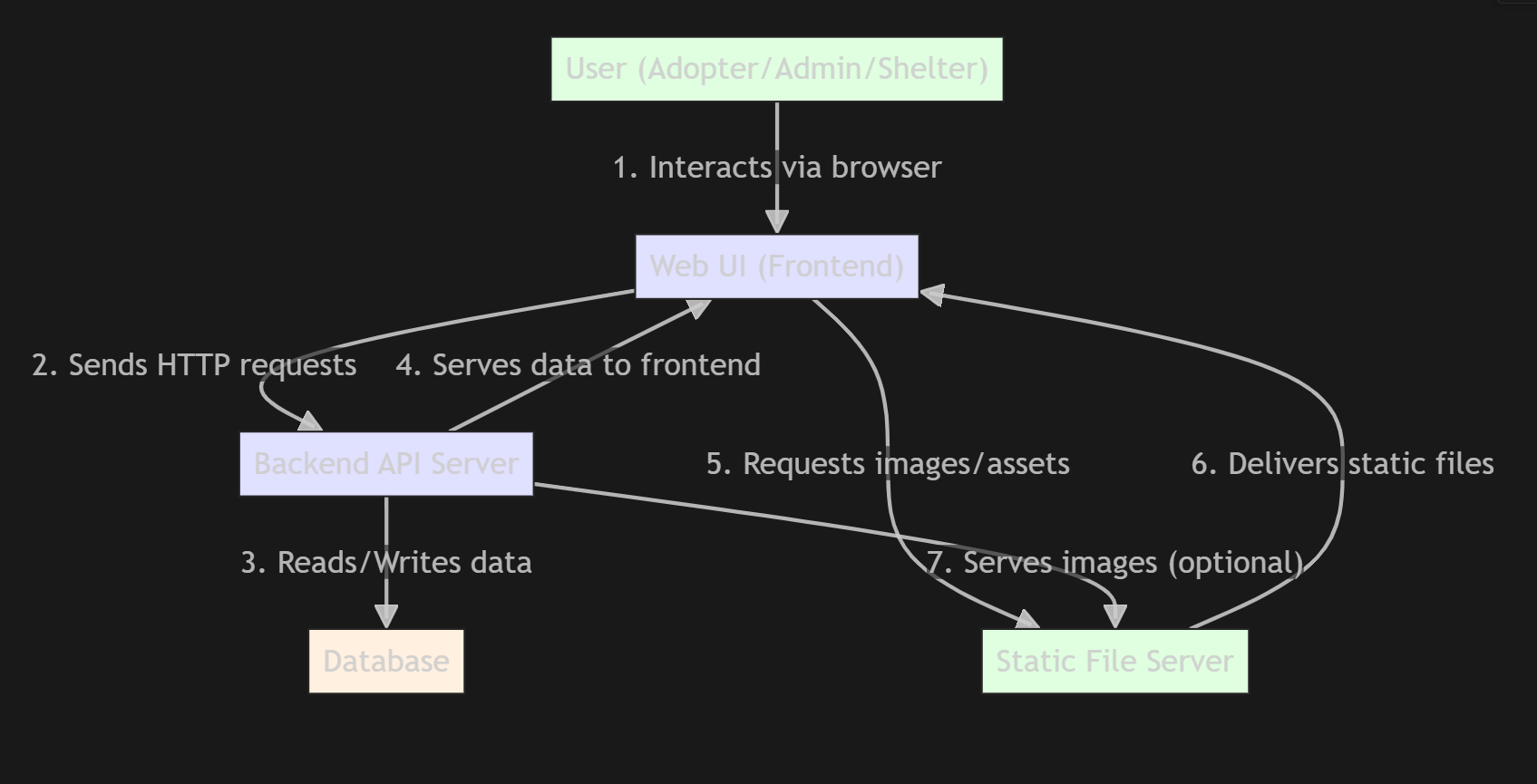
**Frontend** depends on the **Backend** for data and business logic via RESTful APIs.

**Backend** depends on the **Database** for persistent storage and retrieval of data.

**Static File Server** is used by the **Frontend** to load assets (images, CSS, JS).

**Backend** may also handle static file serving if not using a separate CDN.

All components interact over standard protocols (HTTP/HTTPS).



**Data Sources**

Users (Adopters, Admins, Shelter Staff)

Static File Server (for images, CSS, JS)

**Processing Logic**

Web UI (Frontend) handles user input and displays data

Backend API Server processes request, applies business logic, and manages authentication

**Data Destinations:**

Database (stores users, pets, applications, etc.)

Web UI (displays results to users)

Static File Server (delivers images/assets to the frontend)

**Security Design**

**Overview of Security Requirements and Considerations**

The Online Pet Adoption System must ensure the confidentiality, integrity, and availability of user and pet data. Security is critical to protect sensitive information (such as user credentials and adoption applications) and to prevent unauthorized access or data breaches The system should comply with best practices for web application security.

**Key security requirements**

Only authorized users can access sensitive features (admin functions, user data).

User credentials and personal information must be protected at all times.

All data exchanges between client and server must be secure.

The system must be resilient against common web vulnerabilities (SQL injection, XSS, CSRF).

**AUTHENTICATION AND AUTHORIZATION MECHANISMS**

**Authentication**

Users must register and log in with a unique username/email and password.

Passwords are never stored in plain text; they are hashed using a secure algorithm (bcrypt, Argon2).

Session management is implemented using secure cookies or tokens (JWT).

Optional: Multi-factor authentication (MFA) for admin accounts.

**Authorization**

Role-based access control (RBAC) is enforced:

Regular users (adopters) can browse pets and submit applications.

Admins and shelter staff can manage pet listings and review applications.

Access to sensitive endpoints is restricted based on user roles.

Backend checks all permissions before processing requests.

**DATA ENCRYPTION AND PROTECTION MEASURES**

**In Transit**

All communication between client and server is secured using HTTPS (SSL/TLS).

Sensitive data (login credentials) is always transmitted over encrypted channels.

**At Rest:**

Passwords are stored as salted hashes.

Sensitive user data is protected in the database with appropriate access controls.

Database backups are encrypted and securely stored.

**Other Protection Measures**

Input validation and sanitization to prevent SQL injection and XSS attacks.

CSRF tokens to protect against cross-site request forgery.

Rate limiting and account lockout mechanisms to prevent brute-force attacks.

Regular security audits and updates to dependencies.

**Performance Design**

**Performance Requirements and Objectives**

The system should provide a responsive user experience, with page loads and API responses typically under 2 seconds.

The system must support multiple concurrent users, including peak loads during adoption events or promotions.

Database queries and data retrieval should be optimized to minimize latency.

Static assets (images, CSS, JS) should load quickly, even on slower connections.

**Strategies for Optimizing System Performance**

**Caching**

* Use server-side caching for frequently accessed data (pet listings, images).
* Implement client-side caching via HTTP cache headers for static assets.
* Optionally, use a distributed cache (Redis) for session data and popular queries.

**Database Optimization**

* Use proper indexing on frequently queried fields ( pet status, user ID).
* Optimize SQL queries and avoid N+1 query problems.
* Regularly analyze and tune database performance.

**Static Asset Optimization:**

* Compress images and use modern formats (WebP).
* Minify CSS and JavaScript files.
* Serve static assets via a Content Delivery Network (CDN) for global reach.

**Load Balancing and Scalability:**

* Deploy the backend behind a load balancer to distribute traffic across multiple servers (if needed).
* Design the system to be horizontally scalable (add more servers as demand grows).

**Asynchronous Processing:**

Offload long-running or resource-intensive tasks (image processing, email notifications) to background jobs.

**Performance Testing Plan**

**Load Testing**

Simulate multiple concurrent users performing typical actions (browsing pets, submitting applications).

Use tools like JMeter, Locust, or k6 to measure system throughput and response times.

**Stress Testing**

Test system behavior under extreme load to identify breaking points and bottlenecks.

**Profiling and Monitoring**

Profile backend code to identify slow functions or queries.

Monitor server and database performance in real time (CPU, memory, disk, network).

**Regression Testing**

Ensure that performance optimizations do not break existing functionality.

**Continuous Improvement**

Regularly review performance metrics and optimize as needed based on real-world usage.

**ERROR HANDLING AND LOGGING**

**Error Handling Mechanisms and Strategies**

**Centralized Error Handling:**

All errors in the backend are caught and handled by a centralized error handler. This ensures consistent error responses and prevents application crashes.

**Graceful Degradation:**

The system provides user-friendly error messages and fallback options when possible, so users are not left with blank screens or cryptic errors.

**Input Validation:**

All user inputs are validated both on the frontend and backend to prevent invalid data from causing errors or security vulnerabilities.

**Exception Handling**

Try-except blocks are used in backend code to catch and handle exceptions. Specific exceptions (database errors, authentication failures) are handled with appropriate responses.

**Frontend Error Display**

The frontend displays clear, actionable error messages to users (“Invalid login credentials”, “Pet not found”, “Application failed, please try again”).

**Logging Requirements and Specifications**

**Log Levels**

**INFO** General application events (user logins, successful operations).

**WARNING** Suspicious or unexpected events (multiple failed logins, deprecated API usage).

**ERROR** Application errors that require attention (database failures, unhandled exceptions).

**CRITICAL** Severe errors causing system outages or data loss.

**Log Content**

Logs should include timestamps, user IDs (if applicable), request details, error messages, and stack traces for debugging.

**Log Storage**

Logs are written to files on the server and/or sent to a centralized logging service.

Sensitive information ( passwords, personal data) is never logged.

**Log Rotation and Retention:**

Implement log rotation to prevent disk space issues.

Retain logs for a defined period (30-90 days) for auditing and troubleshooting.

**Error Codes and Messages**

**Standardized Error Codes**

The backend returns standardized error codes and messages in API responses. Example format:

{

"error code": "AUTH\_INVALID\_CREDENTIALS",

"message": "Invalid username or password."

}

**Common Error Codes:**

* AUTH\_INVALID\_CREDENTIALS: Invalid login details.
* AUTH\_UNAUTHORIZED: User not authorized for this action.
* NOT\_FOUND: Requested resource does not exist.

VALIDATION\_ERROR: Input validation failed.

SERVER\_ERROR: Internal server error.

**User-Facing Messages**

Error messages shown to users are clear, concise, and do not expose sensitive system details.

**THIRD-PARTY INTEGRATIONS**

**List of Third-Party Services or APIs Integrated into the System**

Depending on your implementation, the Online Pet Adoption System may integrate with the following third-party services:

**Email Service Providers** (SendGrid, Mail gun, SMTP)

**Cloud Storage/CDN** (AWS S3, Cloudinary, Google Cloud Storage)

**Authentication Providers** (optional, e.g., Google, Facebook OAuth)

**Analytics Tools** (Google Analytics)

**CAPTCHA Services** (Google reCAPTCHA)

**SMS Notification Services** (optional, Twilio)

**Description of Integration Points and Data Exchange Formats**

**Email Service Providers**

**Integration Point:** Backend triggers emails for account registration, password reset, and application status updates.

**Data Exchange Format:** JSON or SMTP protocol; includes recipient email, subject, and message body.

**Cloud Storage/CDN**

**Integration Point:** Pet images and other static assets are uploaded to and served from cloud storage.

**Data Exchange Format:** HTTP(S) requests with file uploads (multipart/form-data); URLs returned for asset access.

**Authentication Providers (OAuth)**

**Integration Point:** Users can log in using third-party accounts (Google, Facebook).

**Data Exchange Format:** OAuth 2.0 protocol; exchange of access tokens and user profile data in JSON.

**Analytics Tools**

**Integration Point:** Frontend includes tracking scripts to monitor user interactions and site traffic.

**Data Exchange Format:** JavaScript snippets; data sent as HTTP requests to analytics servers.

**CAPTCHA SERVICES**

**Integration Point:** Used on registration and application forms to prevent spam and bots.

**Data Exchange Format:** Frontend displays CAPTCHA widget; backend verifies user response via API (JSON).

**SMS Notification Services**

**Integration Point** (Optional) Send SMS notifications for important updates (application status).

**Data Exchange Format:** REST API calls with recipient phone number and message content (JSON).

**DEPLOYMENT PLAN**

**Overview of the Deployment Process**

1. **Preparation**

Ensure all code is tested and passes quality checks.

Update documentation and configuration files as needed.

1. **Build and Package**

Package the application code, static assets, and dependencies.

Build Docker images (if using containerization).

1. **Deployment**

Deploy the backend application to a web server or cloud platform (AWS, Heroku, Digital Ocean).

Deploy the database to a managed service or secure server.

Upload static assets (images, CSS, JS) to a CDN or static file server.

Apply database migrations if schema changes are present.

1. **Post-Deployment**

Run smoke tests to verify deployment success.

Monitor logs and system health.

Roll back if critical issues are detected.

**MAINTENANCE AND SUPPORT**

**Guidelines for System Maintenance and Support**

**Regular Monitoring**

Continuously monitor system health, performance, and security using automated tools and manual checks.

**Documentation**

Maintain up-to-date documentation for system architecture, deployment, configuration, and troubleshooting.

**User Support**

Provide a helpdesk or support contact (email, ticketing system) for users to report issues or request assistance.

**Preventive Maintenance**

Schedule regular reviews of logs, database integrity, and system resource usage to detect and address potential issues early.

**PROCEDURES FOR HANDLING SOFTWARE UPDATES, PATCHES, AND BUG FIXES**

**Version Control**

All changes (updates, patches, bug fixes) are tracked in a Git repository with clear commit messages and version tags.

**Testing**

Apply updates and bug fixes in a staging environment first. Run automated and manual tests to ensure stability.

**Deployment**

Use automated deployment tools (CI/CD pipelines) to roll out updates to production. Schedule deployments during low-traffic periods when possible.

**Rollback**

Maintain backup and rollback procedures to quickly restore the previous stable version if an update causes issues.

**Patch Management**

Regularly check for and apply security patches to all software dependencies and the underlying operating system.

**ESCALATION PROCESS FOR RESOLVING ISSUES**

**Issue Triage**

Classify issues by severity (critical, major, minor) and impact on users.

**First-Line Support**

Support staff or system administrators handle common issues and user requests.

**Escalation**

If an issue cannot be resolved at the first level, escalate to the development team or relevant technical experts.

**Incident Management:**

For critical incidents (esystem outages, data breaches), follow an incident response plan:

Notify stakeholders immediately.

Assign a response team to investigate and resolve the issue.

Communicate status updates to affected users.

Document the incident and implement measures to prevent recurrence.