FINAL PROJECT MILESTONE 2

Ujwal Chandrashekar **Group 2**

Shreyas Pogal Naveen

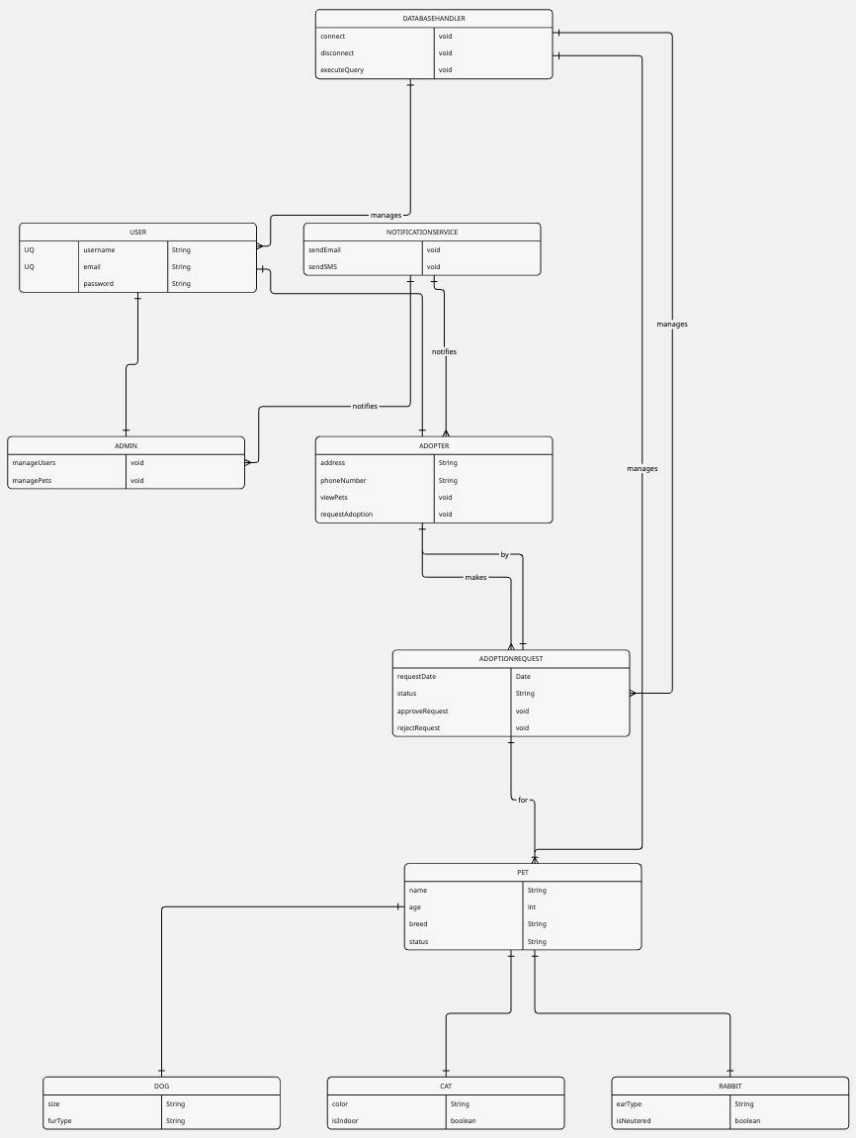
Project Topic: Pet Adoption Management System

Problem Statement

Many animal shelters face challenges in efficiently managing pet adoption processes. Adopters often find it difficult to search for pets, while shelter staff struggle to keep track of pet availability, adoption requests, and user details. This project aims to create a Pet Adoption Management System that allows shelters to add and manage pet listings and users to search for pets and apply for adoption online. The system will simplify the process through a user-friendly interface, streamline the approval workflow for staff, and improve adoption success rates by automating and organizing the adoption process.

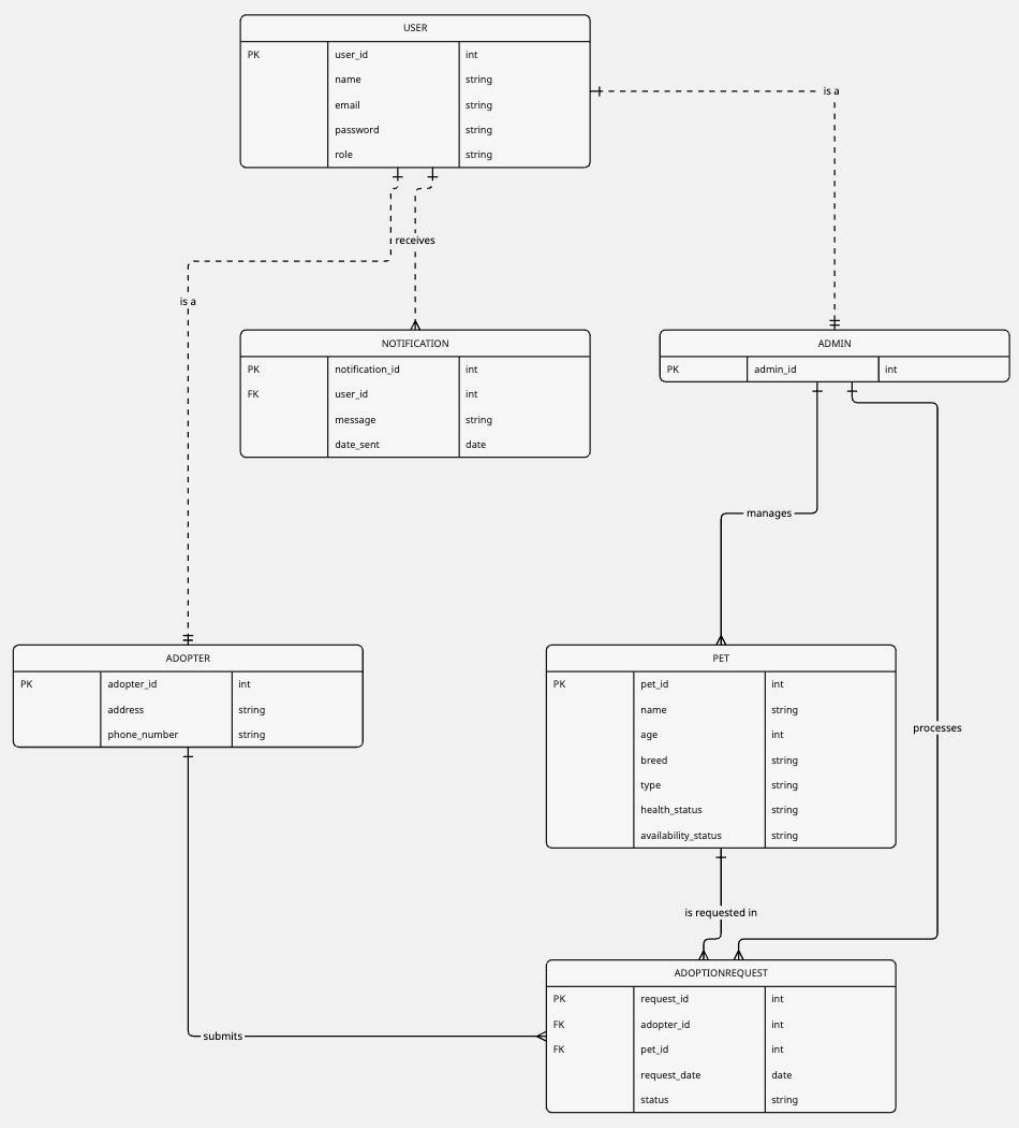
UML Diagram

The UML Class Diagram models the object-oriented structure of the Pet Adoption Management System. It outlines the key classes such as User, Admin, Adopter, Pet, AdoptionRequest, NotificationService, and DatabaseHandler. The diagram highlights inheritance relationships (e.g., Admin and Adopter are specialized forms of User) and clearly defines class attributes and methods. It also showcases class interactions such as how Adopters submit adoption requests, Admins manage pets, and NotificationService notifies users. This diagram emphasizes the use of core OOP principles like abstraction, inheritance, encapsulation, and method definition in line with Java development practices.



ER Diagram:

The Entity-Relationship (ER) Diagram represents the database structure of the Pet Adoption Management System. It includes entities such as User, Admin, Adopter, Pet, AdoptionRequest, and Notification, along with their primary keys, attributes, and foreign key relationships. The diagram captures how users receive notifications, adopters submit adoption requests, and admins manage and process pets. The use of "is-a" relationships denotes inheritance, while labeled associations like submits, manages, and processes define clear relational flows. This ER diagram provides a normalized view of how data is organized and related across the system.



Tech Stack

1. Backend: Java
2. Frontend: HTML, CSS
3. Data Handling: CSV files for storing pet and user data
4. Database: MongoDB
5. IDE: Eclipse
6. Testing: Manual testing via the Main Class and CLI outputs

Functionalities Implemented for Milestone 2:

1. User Registration and Login:
    a. Users can register as Admin or Adopter.
    b. Credentials are stored and verified using user_data.csv.
2. Add and Display Pets:
    a. Admins can add pets (Dog, Cat, Rabbit) with unique attributes.
    b. Pets are stored in pet_data.csv and can be listed or viewed.
3. Adoption Request Flow:
    a. Adopters can submit adoption requests by entering a pet ID.
    b. Admins can view and process requests (approve/reject).
    c. Requests are logged in adoption_requests.csv.
4. Notification System:
    a. Basic notification implemented through console messages after request decisions.

Object-Oriented Concepts Implemented:

1. Abstraction:
   The User, Item, and Listing classes act as **data models** that abstract real-world entities. Each class holds relevant properties and behavior related to its role in the system.

   Example:

```
public class User {
    private Long id;
    private String username;
    private String password;
    private String role;

    // Getters and Setters omitted for brevity
}
```

2. Inheritance:
   While classical inheritance is not deeply used in this milestone, the use of **interface extension** and Spring Boot's JpaRepository inheritance supports database operations through abstraction.

   Example:

```
public interface UserRepository extends JpaRepository<User, Long> {
    User findByUsername(String username);
}
```

   This is a form of **interface-based inheritance**, allowing reuse of Spring's repository methods for entities.

3. Encapsulation:
   All model classes use private fields with public getters and setters, protecting direct access to internal state.

   Example:

```
public class Listing {
    private Long id;
    private String title;
    private String description;
    private String status;

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    // Other getters/setters...
}
```

4. Polymorphism:
   Polymorphism is demonstrated in the controller layer by how endpoints invoke services

that return different model types (Item, Listing, etc.). The same controller method can return different data objects dynamically based on path or parameters.

Example:

```
@GetMapping("/items")
public List<Item> getAllItems() {
    return itemRepository.findAll();
}
```

Different endpoints across ItemController, ListingController, and AuthController return or process different objects while following a uniform controller structure.

5. Composition:
   Composition is used where one entity embeds another. For instance, Listing contains Item references logically (in code and business logic if not directly embedded in the class), and the ItemController coordinates both entities together. Even without explicit object references in model fields, the logical composition is evident in controller workflows.

6. Interface:
   All repository interfaces use Spring Data JPA's interface system, making it easy to decouple implementation from definition.

   Example:

```
public interface ItemRepository extends JpaRepository<Item, Long> {
}
```

   Spring implements this interface automatically at runtime. This is **interface-based programming**, promoting loose coupling.

Functionalities to be Implemented by the End of Milestone 3:

- GUI Interface using Java Swing or improved HTML interface.
- Implement NotificationService interface to modularize and simulate various alert types.
- Add persistent login session handling.
- Implement robust exception handling and user input validation.
- Enhance the search feature to support combined filters and partial matches.

Contributions

1. Ujwal Chandrashekar – Backend development (User authentication, Adoption process)
2. Shreyas Pogal Naveen – Frontend development (HTML, CSS, Validations)

3. Ujwal Chandrashekar – Data Handling
4. Shreyas Pogal Naveen – Database connection Setup, Testing and debugging

Documentation Update:

Approach to Implementation

For Milestone 2, our focus was to build out the backend functionality for user registration/login, listing management, and item management using a layered Java backend built with Spring Boot. We used MongoDB (or file-based data initially) for persistence and followed a modular approach to organize our logic into three primary layers: Model, Repository, and Controller.

Each domain entity—such as User, Listing, and Item—is represented as a Java class with clearly encapsulated fields. The controllers expose RESTful endpoints for handling HTTP requests, while repository interfaces handle database operations. We emphasized clean code, object-oriented design, and REST principles.

Functionalities and Code Snippets

1. **1. User Registration & Login (Authentication)**
    - Implemented in AuthController.java
    - Stores and fetches users via UserRepository
    - Login works by checking the existence of a username/password match

```java
@RestController
@RequestMapping("/auth")
public class AuthController {

    @Autowired
    private UserRepository userRepository;

    @PostMapping("/register")
    public User registerUser(@RequestBody User user) {
        return userRepository.save(user);
    }

    @PostMapping("/login")
    public User loginUser(@RequestBody User user) {
        User existingUser = userRepository.findByUsername(user.getUsername());
        if (existingUser != null && existingUser.getPassword().equals(user.getPassword())) {
            return existingUser;
        }
        return null;
    }
}
```

2. User Data Handling (Model)
   - All fields are encapsulated and exposed via getters/setters
   - Role-based access can be extended using the role field

```java
public class User {
    private Long id;
    private String username;
    private String password;
    private String role;

    // Getters and setters...
}
```

3. Item Management
   - Implemented in ItemController.java
   - Provides endpoints to retrieve and create item entries
   - Uses ItemRepository for data access

```java
@RestController
@RequestMapping("/items")
public class ItemController {

    @Autowired
    private ItemRepository itemRepository;

    @GetMapping
    public List<Item> getAllItems() {
        return itemRepository.findAll();
    }

    @PostMapping
    public Item createItem(@RequestBody Item item) {
        return itemRepository.save(item);
    }
}
```

4. Listing Management
   - Implemented in ListingController.java
   - Handles retrieval and creation of listings
   - Similar structure to item management for consistency

```java
@RestController
@RequestMapping("/listings")
public class ListingController {

    @Autowired
    private ListingRepository listingRepository;

    @GetMapping
    public List<Listing> getAllListings() {
        return listingRepository.findAll();
    }

    @PostMapping
    public Listing createListing(@RequestBody Listing listing) {
        return listingRepository.save(listing);
    }
}
```

5. Repository Interfaces
   - Defined for each model: User, Item, and Listing
   - Inherit from Spring's JpaRepository to provide CRUD operations without boilerplate

```java
public interface UserRepository extends JpaRepository<User, Long> {
    User findByUsername(String username);
}
```

```java
public interface ListingRepository extends JpaRepository<Listing, Long> {
}
```

```java
public interface ItemRepository extends JpaRepository<Item, Long> {
}
```

6. Backend Design Pattern
   - We followed a Model–Repository–Controller pattern:
   - Model – Holds data structure (e.g., User.java, Item.java, Listing.java)
   - Repository – Interfaces for DB operations (e.g., UserRepository)
   - Controller – Exposes REST endpoints to frontend (e.g., AuthController, ItemController)
   - This separation of concerns makes the codebase modular, testable, and scalable.

GitHub Repository: https://github.com/CSYE6200-Object-Oriented-Design-Spr25/csye6200-spring-2025-final-project-final_project_group_2