FINAL PROJECT MILESTONE 3

Ujwal Chandrashekar **Group 2**

Shreyas Pogal Naveen

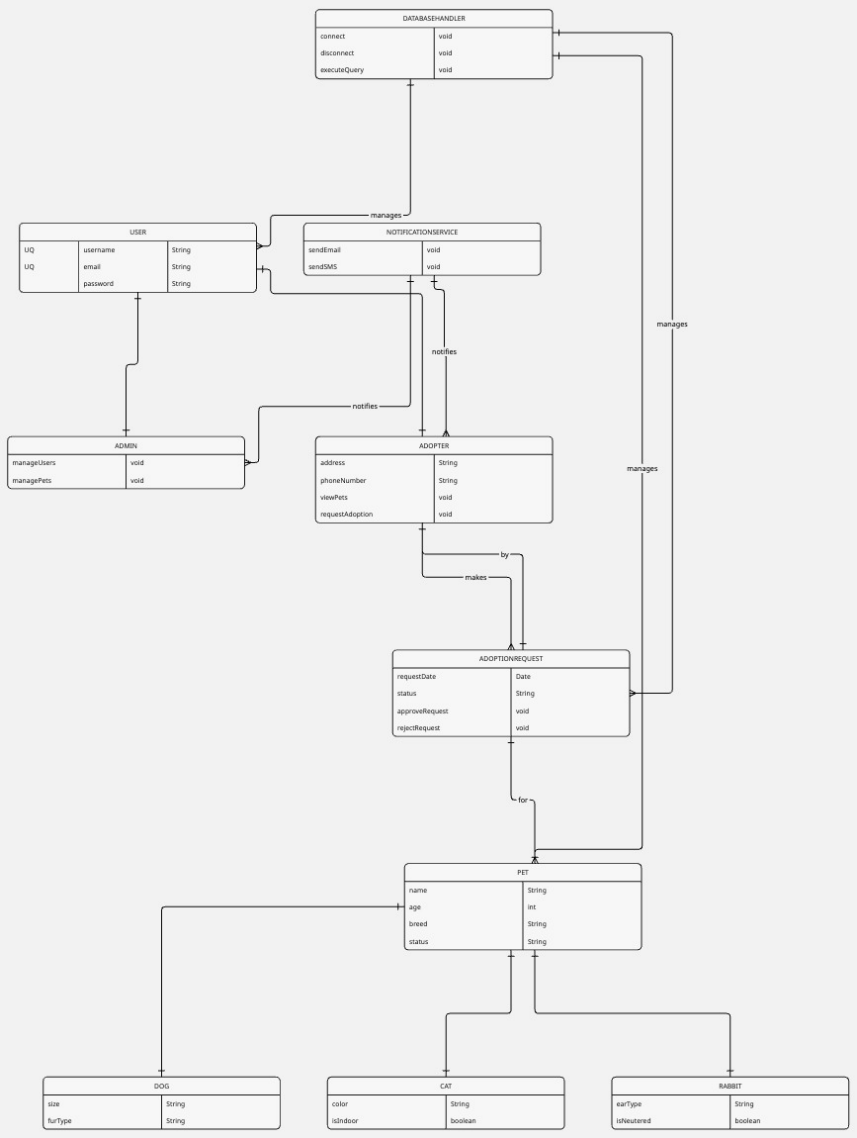
Project Topic: Pet Adoption Management System

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.

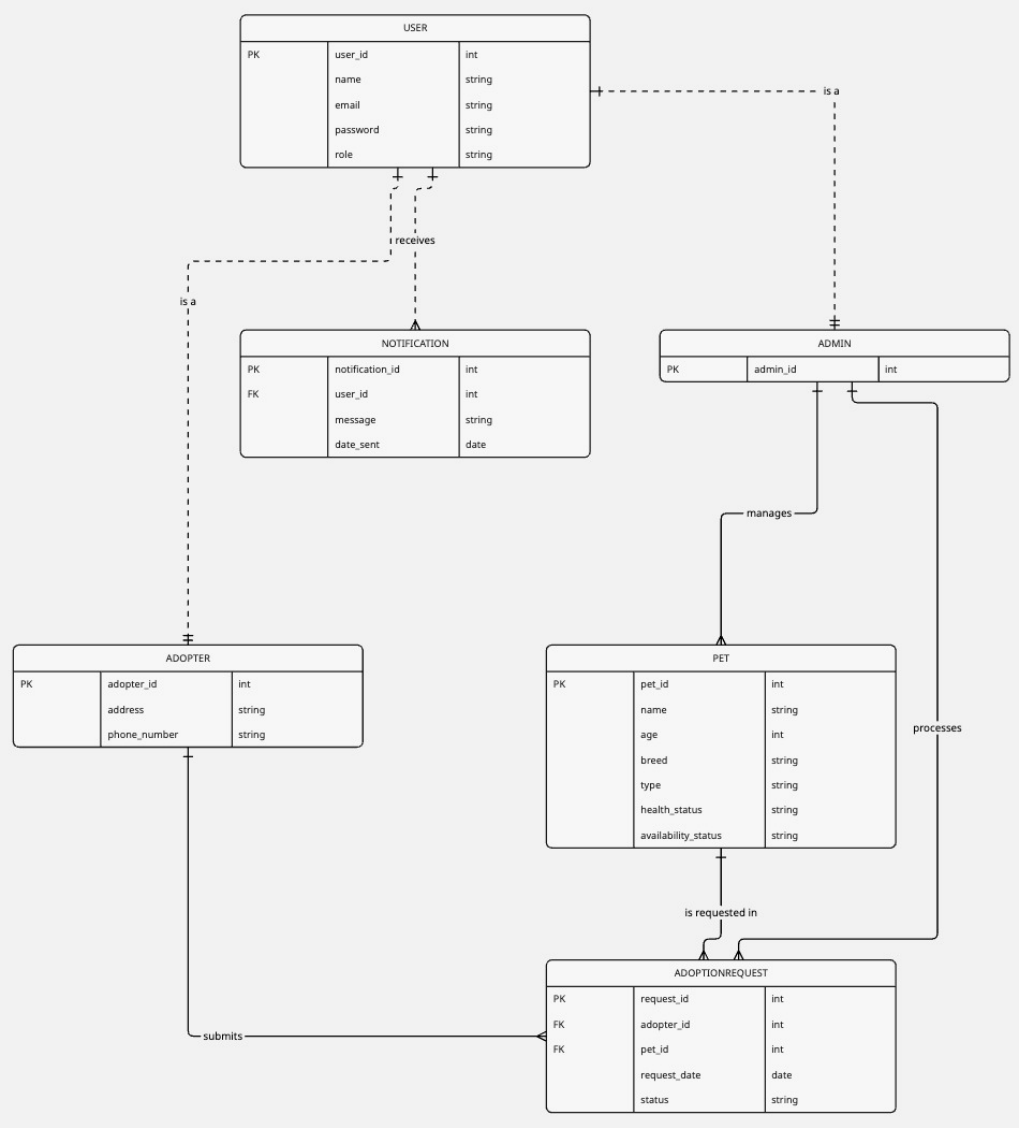
UML Diagram

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 with Spring Boot
2. Frontend: React.js, HTML, CSS
3. Logging : CSV files for listing's CRUD updates
4. Database: MongoDB
5. IDE: Eclipse
6. Testing: Manual testing via console, UI and controller endpoints

Functionalities Implemented:

1. User Registration and Login:
    a. Role-based registration for Admin and Adopter.
    b. Login validation using AuthController.java.

2. Pet Management (Admin)
    a. Admin can add or remove pets (Dog, Cat, Rabbit) with unique attributes.
    b. Pets stored in listings column in MongoDB.
    c. Pet listings displayed for browsing.

3. Adoption Request Flow:
    a. Adopters submit adoption requests by clicking "Request".
    b. Admins can approve or reject requests.
    c. All CRUD operations on requests are logged in Listings_Log.CSV file.
    d. Denied adoption requests are re-listed on User Dashboards.

4. Notification System:
    a. Users receive request decisions on their dashboard.

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:

```java
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);
}
```

```
public class Admin extends User {
    public Admin(String username, String password) {
        super(username, password, "Admin");
    }
}
```

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:

```
public class Pet {
    public String getDetails() {
        return "General Pet";
    }
}

public class Dog extends Pet {
    @Override
    public String getDetails() {
        return "Dog-specific details";
    }
}
```

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

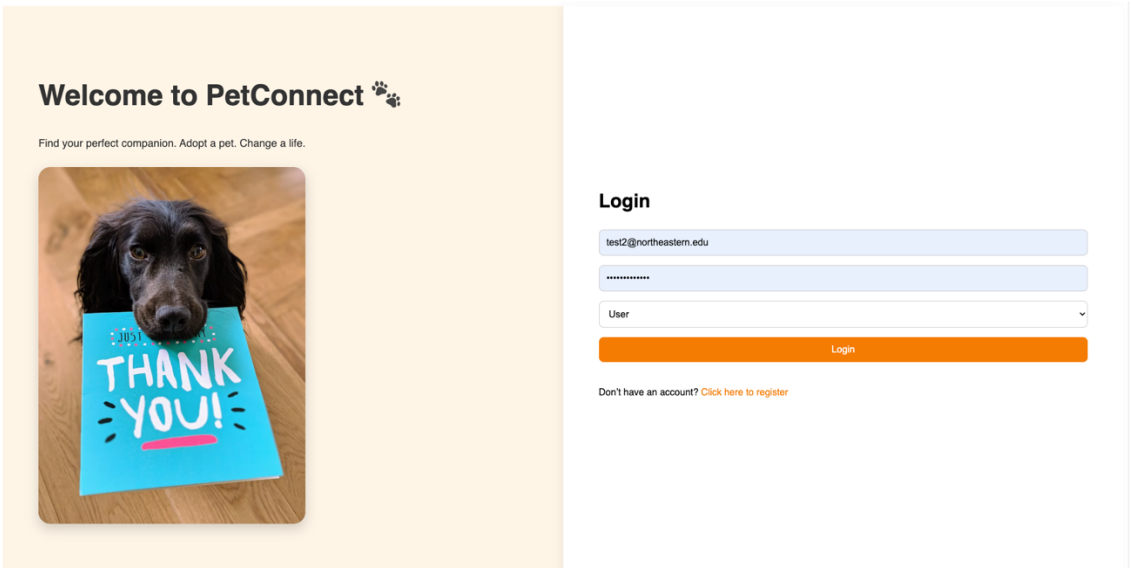
Different endpoints across ItemController, ListingController, and AuthController return or process different objects while following a uniform controller structure.

Functionalities to be Implemented by the End of Milestone 3:

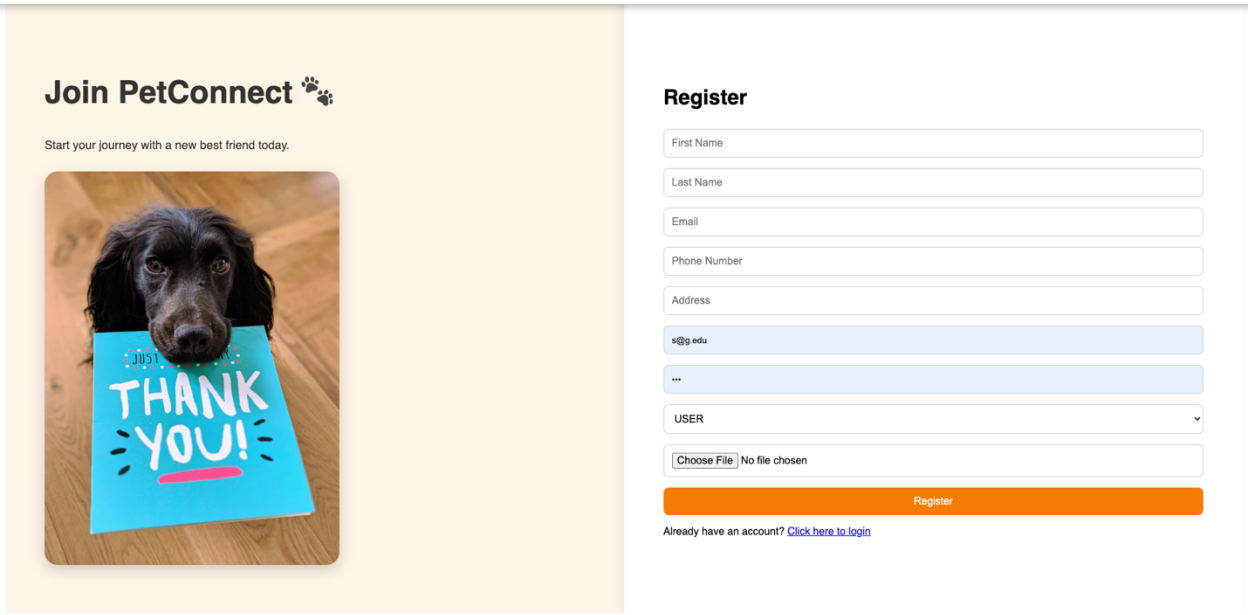
1. Add React for login, registration, and dashboard
2. Modular Dashboard System
3. Persistent Session Handling
4. Robust Exception Handling
5. Input validation and graceful error management

Screenshots:

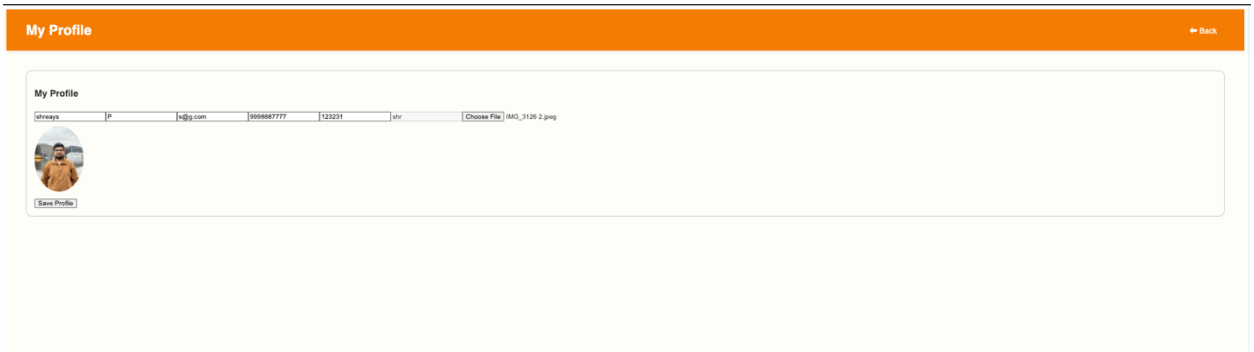
Login Page:



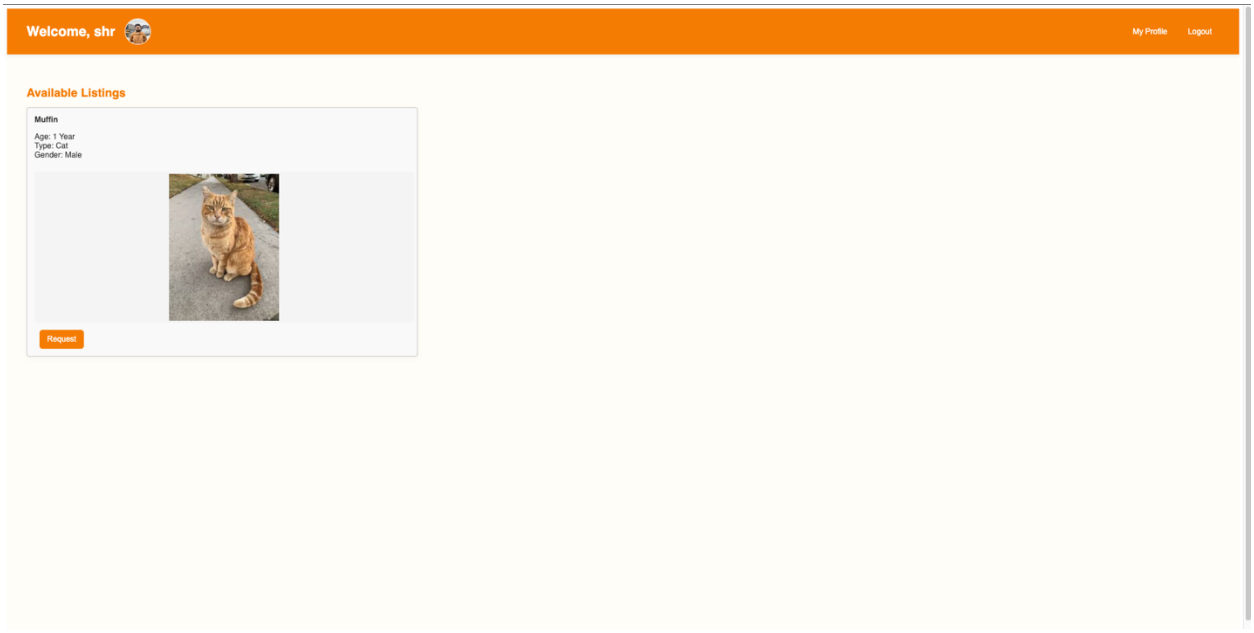
Sign-UpPage:



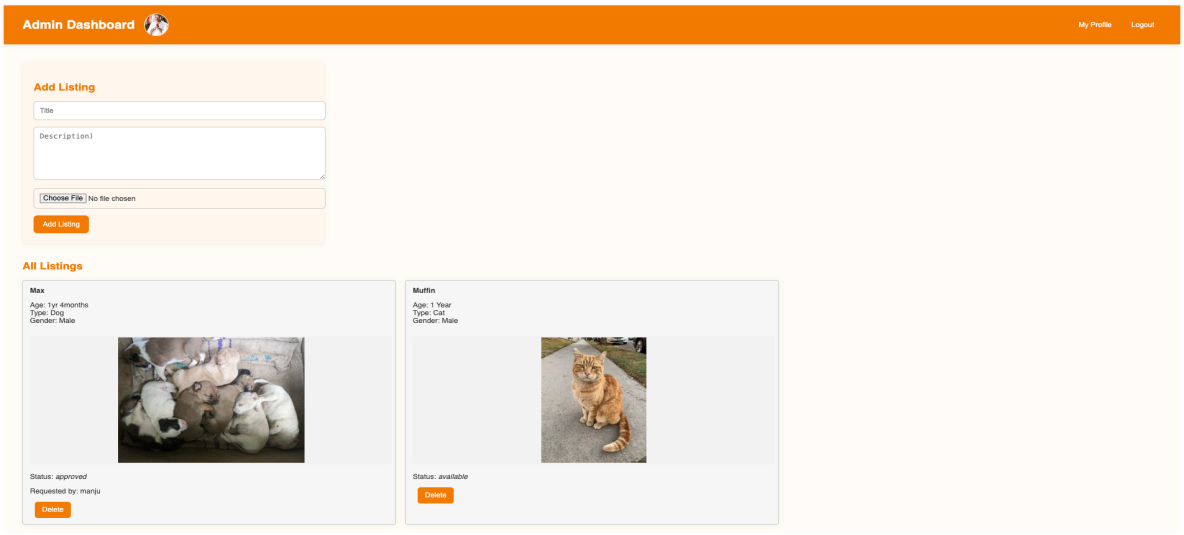
UserProfile:



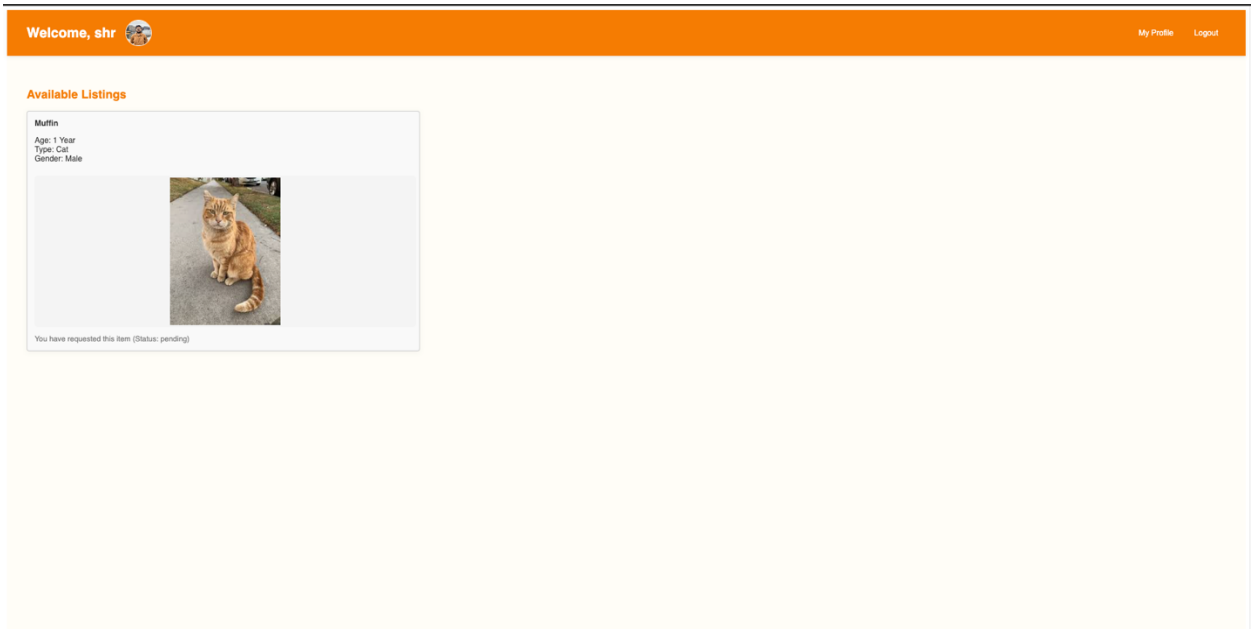
Listings:



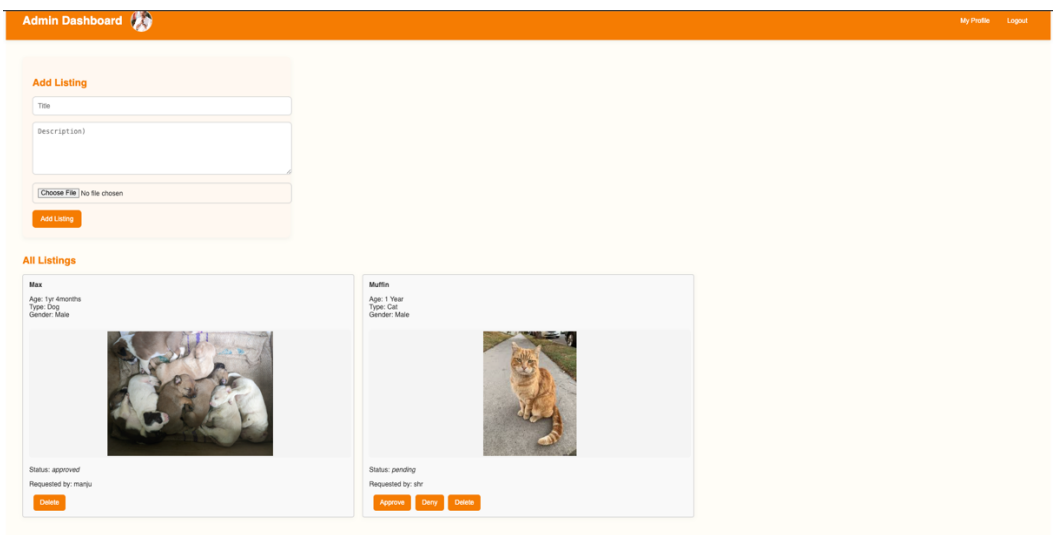
Admin:



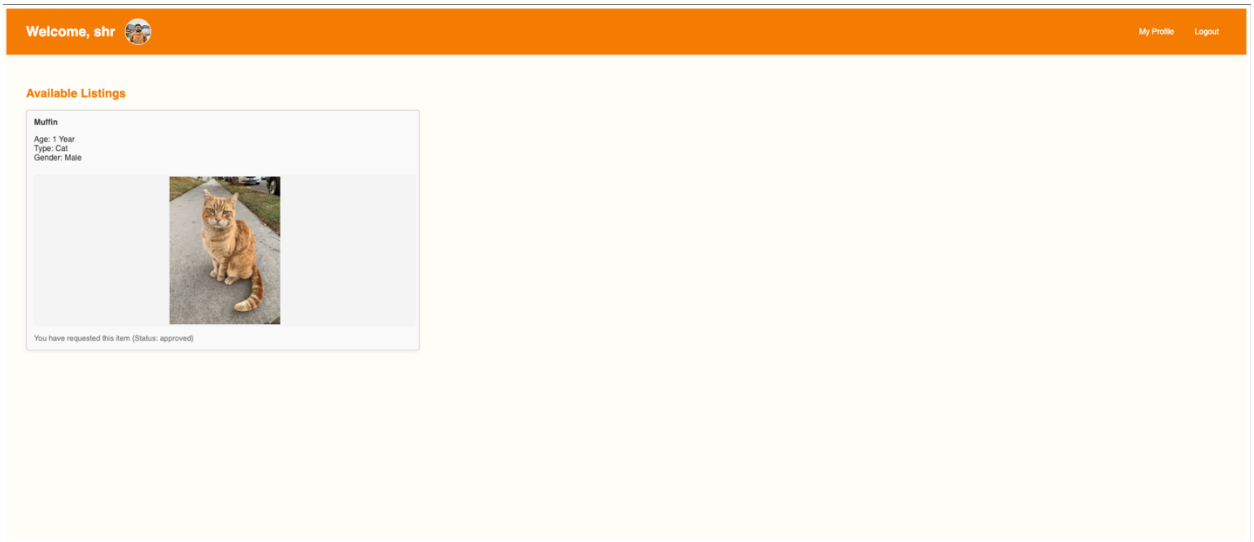
UserRequest:



Admin Approval:



User Approved:



Contributions

1. Ujwal Chandrashekar – Backend development, User authentication
2. Shreyas Pogal Naveen – Frontend development, React.JS
3. Ujwal Chandrashekar – Adoption Logic, workflow, Data Handling
4. Shreyas Pogal Naveen – Database connection Setup, Testing and debugging

Documentation Update:

Approach to Implementation

For Milestone 3, our primary objective was to complete the end-to-end functionality of the Pet Adoption Management System, including the integration of a graphical user interface using React.js, and finalizing all adoption-related features. We continued using a modular Java application design and organized the project into distinct Model, Controller, and Service components. For data persistence, we used MongoDB to manage users, pets, and adoption requests, while also maintaining flexibility for future MongoDB integration.

Each domain entity—such as User, Pet, and AdoptionRequest—was modeled as a separate class with encapsulated fields and accessor methods. The controllers were responsible for handling business logic related to login, registration, pet creation, adoption requests,. Logic related to file operations was abstracted out to utility classes to maintain separation of concerns.

Maintained layered structure:

a. Model: User, Listing, Item, Enum (UserRole, ListingStatus), BaseEntity (shared fields)
b. Controller: Auth (Login/Register/Profile), Listing (CRUD, request, approval), Item (basic request flow)
c. Service/Utility: ListingService (business logic), ListingCSVLogger (CSV logging), Exportable (CSV row interface)
d. Data persistence via MongoDB (collections: users, listings, items) and CSV file (logs/listings_log.csv)
e. Implemented Exportable interface for modular CSV row formatting (used in ListingCSVLogger)
f. Added input validation & exception handling (e.g., missing fields, logging errors, user not found)
g. Search/filter logic handled via service (get listings per user, show denied-to-public flow)
h. Applied OOP principles:
    1. Inheritance: User, Listing, Item extend BaseEntity (id, timestamps)
    2. Polymorphism: Listing implements Exportable, used in logger polymorphically
    3. Encapsulation: All models use private fields with public getters/setters
    4. Interface: Exportable used for flexible CSV serialization

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