UNIVERSITY OF DHAKA

Department of Computer Science and Engineering

CSE-3216: Software Design Pattern

Petfy: Connecting loving pets with caring homes!

Progress Tracking - 1

Submitted By:

Ahaj Mahhin Faiak

Roll No : 01

Tahsin Ahmed

Roll No : 03

Md. Atikur Rahman Hridoy

Roll No : 19

Mahafuzul Islam Shawon

Roll No : 35

Submitted On :

November 24, 2024

Submitted To :

Dr. Ismat Rahman

Redwan Ahmed Rizvee

Contents

1 Introduction

1.1 Purpose of the System

Petfy: Connecting loving pets with caring homes! is an innovative online platform designed to simplify and enhance the pet adoption process. By bridging the gap between pet shelters, rescues, and individuals looking to adopt, Petfy creates a seamless experience that benefits both pets and potential owners. With a user-friendly interface and advanced search capabilities, it ensures pets find loving homes quickly and efficiently.

2 Use of Design Patterns

2.1 Frontend

1. Component Design Pattern
Purpose:
Enables the creation of reusable, modular, and isolated UI components.
Implementation:
React components divide the UI into distinct, reusable elements.
Component:
Examples include ‘Navbar.jsx‘, ‘Card.jsx‘, and ‘ProfileHeader.jsx‘.
Shared Features:

- **Encapsulation:** Manages its own logic and presentation.
- **Reusability:** Used across multiple sections of the app.

Example Usage:

- **Page:** ‘ProfilePage.jsx‘
  Work: Displays user profiles using ‘ProfileHeader‘ and cards for additional information.
- **Page:** ‘HomePage.jsx‘
  Work: Renders the ‘Navbar‘ and ‘Card‘ components to showcase highlights or featured items.

Benefits:

- Simplifies development.
- Promotes consistency and reduces code duplication.

2. Routing Pattern
Purpose:
Manages navigation between different application views.
Implementation:
Routes are defined using ‘react-router-dom‘.
Component:
Examples include ‘App.jsx‘ and ‘Navbar.jsx‘.
Shared Features:

- **Dynamic Rendering:** Renders components based on URL paths.
- **Nested Routes:** Handles child routes dynamically.

Example Usage:

- **Page:** ‘App.jsx‘
  Work: Defines all application routes, such as ‘/login‘ for the login page and ‘/profile‘ for the user profile.

- **Page:** ‘LogIn.jsx‘
  Work: Handles user authentication and redirects upon successful login.

Benefits:

- Simplifies navigation logic.

- Improves performance by dynamically loading views.

3. Observer Pattern
Purpose:
Enables components to react to state changes dynamically.
Implementation:
Managed using React hooks (‘useState‘, ‘useEffect‘) or a global state (e.g., Context API).
Component:
Examples include ‘Navbar.jsx‘ and ‘ProfileHeader.jsx‘.
Shared Features:

- **State Management:** Centralized state updates.

- **Dynamic UI Updates:** Re-renders based on state changes.

Example Usage:

- **Page:** ‘Navbar.jsx‘
  Work: Updates buttons (e.g., Log In/Log Out) based on the login state.

- **Page:** ‘ProfileHeader.jsx‘
  Work: Reacts to user data updates and displays current profile information dynamically.

Benefits:

- Improves user experience with real-time responsiveness.

- Reduces manual DOM manipulation.

2.2 Backend

1. Model-View-Controller (MVC)

Overview:

The MVC pattern remains a core design pattern, separating the concerns of handling user input, business logic, and data management into three distinct layers.

Key Features:

- Separation of concerns between the controller, model, and view (for APIs, the view is typically the response sent back to the user).
- Facilitates modular development and easy maintenance.
- Each component can be tested independently.

Core Components:

- **Controller:**
  - PetController.java: Handles HTTP requests related to pet operations like /add, /get, etc. Also integrates with services and manages response generation.
  - UserController.java (implied by UserService usage): Likely handles user-related operations such as registration, login, and profile management.
- **Model:**
  - Pet.java, User.java, and AdoptionRequest.java: These classes define the application's data structure.
  - SecurityConfig.java: Manages the security configurations but isn't strictly part of MVC in the traditional sense; it is more of a cross-cutting concern.
- **Service:**
  - PetService.java: Handles the pet-related business logic.
  - UserService.java: Manages user operations, such as authentication, registration, and profile updates.
- **Security Configuration:**
  - SecurityConfig.java: Configures Spring Security settings like authentication providers, password encoding, and permissions.

Functional Highlights:

- **Routing:** Endpoints such as /add for pet creation are mapped to controller methods.
- **User Management:** The UserService is involved in user-related operations like authentication and authorization.
- **Security:** The SecurityConfig class manages security-related features like authentication providers and user details service.

Advantages:

- Separation of concerns enhances maintainability and extensibility.

- Security management is centralized in SecurityConfig rather than spread across controllers.

2. Service Layer Pattern
Overview:
The Service Layer Pattern manages all business logic in a dedicated service layer. This is especially useful for handling complex operations like user authentication or pet management.
Key Features:

- Centralized business logic and service orchestration.
- Services provide abstraction over business logic and can be reused across multiple controllers.

Core Components:

- **Service Classes:**
  - PetService.java: Encapsulates business logic related to pet operations like creating, updating, or deleting pets.
  - UserService.java: Handles business logic related to user authentication, registration, and password management.
- **Security-Related Service:**
  - SecurityConfig.java: Configures Spring Security's authentication manager and related services.

Functional Highlights:

- **User Authentication:** UserService interacts with authentication logic (possibly invoking custom logic or leveraging Spring Security).
- **Role-based Access Control:** SecurityConfig defines role-based authorization, ensuring that only authorized users can access certain resources (e.g., admin-only endpoints).
- **Business Logic Separation:** Pet-related operations are centralized in PetService, and user-related logic is managed by UserService.

Advantages:

- Testability: Services can be independently tested, particularly for user and pet management.
- Maintainability: Each service handles a specific part of the business logic, making the codebase cleaner.

3. Repository Pattern
Overview:
The Repository Pattern abstracts all database interactions, allowing other layers of the application (services, controllers) to focus on business logic without worrying about how data is retrieved or stored.
Key Features:

- Provides a centralized interface for data access.

- Makes it easier to implement different data access strategies (e.g., switching from JPA to MongoDB).

Core Components:

- **Repository Interfaces:**
  - PetRepo.java: Handles CRUD operations for pet-related data.
  - AnimalRepo.java, CategoryRepo.java: Manage interactions with other domain objects.
  - User Repository (implied by UserService): Manages interactions with the User entity.

Functional Highlights:

- **CRUD Operations:** Abstracts the complexities of interacting with the database (e.g., findById(), save(), delete()) in the repository layer.

- **Pagination/Sorting:** Can manage complex data retrieval tasks, such as fetching paginated pet data or user records.

Advantages:

- Code Reusability: Repositories abstract the data layer, making it reusable across services.

- Flexibility: The underlying data source can be swapped without affecting the higher layers (controllers, services).

4. Singleton Pattern (SecurityContext)
Overview:
The Singleton Pattern ensures that only a single instance of a class is created and shared across the application.
Key Features:

- Ensures one instance of AuthenticationManager and AuthenticationProvider throughout the application.

- The security configuration guarantees that components like password encoders or authentication providers are not instantiated multiple times.

Core Components:

- SecurityConfig.java: Ensures that AuthenticationProvider and AuthenticationManager are singletons in the Spring container by using @Bean.

Functional Highlights:

- **Authentication Management:** AuthenticationManager is used across the application to authenticate users.

- **Password Encoding:** Ensures that password encoding is consistent throughout the application.

Advantages:

- Efficiency: Reduces resource consumption by reusing instances.

- Centralized Control: Ensures that authentication-related components are handled by a single, shared instance, simplifying configuration and management.

3 Conclusion

In this project, a variety of well-established design patterns have been utilized to ensure a clean, maintainable, and scalable architecture.

On the frontend, the **Component Design Pattern** facilitates modularity by breaking the UI into reusable components, leading to better code organization and a more consistent user interface. The **Routing Pattern** simplifies navigation between different application views, enhancing the user experience by enabling dynamic view rendering. Additionally, the **Observer Pattern** enables real-time UI updates based on state changes, which provides a responsive and interactive experience for the users.

On the backend, the **Model-View-Controller (MVC)** pattern ensures a clear separation of concerns between the controller, business logic, and data management, which enhances maintainability and extensibility. The **Service Layer Pattern** centralizes business logic, making it easier to manage complex operations such as user authentication and pet management. The **Repository Pattern** abstracts data access, promoting flexibility and code reusability, while the **Singleton Pattern** ensures that key authentication components are efficiently managed as single instances throughout the application.

By applying these design patterns, the system benefits from improved organization, flexibility, and scalability, ensuring that both the frontend and backend are well-structured and easy to maintain as the application evolves.