REAL ESTATE MANAGEMENT SYSTEM - PROJECT REPORT

Student Information

- Name: Vikas Meneni
- Course: Enterprise Software Design
- Date: December 13, 2025

Contents

1. Project Summary

Project Name: RealEstate Pro

Technology Stack: Spring Boot, Hibernate/JPA, Thymeleaf, MySQL

RealEstate Pro is a comprehensive web-based property management platform designed to streamline the real estate transaction process. Unlike traditional real estate systems, this solution connects multiple stakeholders—property owners, real estate agents, buyers, renters, and administrators—on a unified platform with role-specific capabilities.

The application addresses the complexity of property management by implementing a multi-tier approval workflow where properties flow from creation through administrative review before becoming visible to potential buyers and renters. Built on a robust Spring Boot backend with MySQL database integration, the platform ensures data integrity, secure communication, and efficient property discovery through advanced search and filtering capabilities.

The system features a sophisticated favorites management system, viewing request coordination, and application processing workflows that facilitate seamless interaction between all parties involved in property transactions.

2. User Roles

The system is designed with specific roles to ensure appropriate access control and security:

- **Administrator (ADMIN):** Superusers responsible for platform oversight, property approval management, and user account administration.
- **Property Owner (OWNER):** Users who list properties for sale or rent. They can manage their listings, respond to viewing requests, and process rental/purchase applications.
- **Real Estate Agent (AGENT):** Professional intermediaries who work on behalf of property owners, buyers, and renters, managing the complete transaction lifecycle.
- **Buyer (BUYER):** Registered users searching for properties to purchase. They can browse listings, request viewings, and submit purchase applications.
- **Renter (RENTER):** Registered users searching for properties to rent. They possess the same capabilities as buyers but focus on rental transactions.

3. Functional Capabilities & Tasks

Task / Feature	Admin	Owner	Agent	Buyer	Renter
User Management					
Activate/Deactivate Users	✓	×	×	×	×
Manage User Permissions	✓	×	×	×	×
Property Discovery					
Browse Properties	✓	✓	✓	✓	✓
Search by Location	✓	✓	✓	✓	✓
Filter by Type/Price	✓	✓	✓	✓	✓
View Property Details	✓	✓	✓	✓	✓
Favorites System					
Add to Favorites	✓	✓	✓	✓	✓

Remove from Favorites	✓	✓	✓	✓	✓
View Favorites List	✓	✓	✓	✓	✓
Property Management					
Create Property Listing	×	✓	✓	×	×
Edit Own Listings	×	✓	✓	×	×
Delete Own Listings	×	✓	✓	×	×
View Property Status	×	✓	✓	×	×
Admin Approval					
Approve Properties	✓	×	×	×	×
Reject Properties	✓	×	×	×	×
View Pending Properties	✓	×	×	×	×
Viewing Requests					
Submit Viewing Request	×	×	✓	✓	✓
Approve/Disapprove Requests	×	✓	✓	×	×
View Request Status	×	×	✓	✓	✓
Applications					
Submit Application	×	×	✓	✓	✓
Approve/Disapprove Applications	×	✓	✓	×	×
Track Application Status	×	×	✓	✓	✓

4. Key Features Overview

4.1 Smart Property Search Dashboard

The property browsing interface features dynamic filtering capabilities:

- **Location Search:** Text-based location filtering with partial matching
- **Property Type Filter:** House, Apartment, Condo, Land options
- **Price Range:** Adjustable minimum and maximum budget constraints
- **Combined Filtering:** All filters work simultaneously for precise results

4.2 Property Lifecycle Management

A comprehensive approval workflow system:

- **Creation:** Owners/Agents submit property details
- **Admin Review:** Properties enter PENDING status

- **Approval/Rejection:** Admin decision determines visibility
- **Status Tracking:** Real-time status updates (PENDING/APPROVED/REJECTED)

4.3 Favorites Collection System

Universal bookmarking functionality available to all users:

- Save properties of interest for future reference
- Dedicated favorites page with full property details
- Quick add/remove toggle on property cards
- User-specific collections with persistent storage

4.4 Viewing Request Coordination

Interactive scheduling system for property tours:

- Buyers/Renters submit requests with preferred date and time
- Agents can submit requests on behalf of clients
- Owners/Agents approve or disapprove requests
- Status tracking with PENDING/APPROVED/REJECTED states

4.5 Application Processing System

Formal application workflow for property transactions:

- Detailed application forms (income, employment, message)
- Owner/Agent review and decision-making interface
- Status tracking throughout the approval process
- Historical record of all applications

5. Technical Implementation

5.1 Architecture: MVC Pattern

The project follows the standard Model-View-Controller architecture:

View (Thymeleaf) → Controller → Service → Repository → Database

5.2 Core Components

5.2.1 Controllers

- **AuthController:** Handles user registration, login, logout, and session management
- **AdminController:** Manages property approvals and user account activation/deactivation
- **OwnerController:** Property CRUD operations, viewing request and application management
- **BuyerController:** Property browsing, search, favorites, viewing requests, applications
- **RenterController:** Identical functionality to BuyerController for rental-focused users
- **AgentController:** Comprehensive access to manage properties and interactions for all parties

5.2.2 Services

- **UserService:** Business logic for user operations, authentication, and account management
- **PropertyService:** Handles property operations, search filtering, and status management
- **ViewingService:** Manages viewing request workflows and status updates
- **ApplicationService:** Processes rental/purchase applications and approvals

5.2.3 Repositories

All repositories utilize Spring Data JPA (no traditional DAO implementation):

- **UserRepository:** User account data access
- **PropertyRepository:** Property listing queries and filtering
- **ViewingRequestRepository:** Viewing request data management
- **ApplicationRepository:** Application data persistence

5.2.4 Models

- **User:** User accounts with UserRole enum (ADMIN/OWNER/AGENT/BUYER/RENTER)
- **Property:** Property listings with PropertyStatus enum (PENDING/APPROVED/REJECTED)

- **ViewingRequest:** Viewing appointments with RequestStatus enum
- **Application:** Rental/purchase applications with ApplicationStatus enum

6. Database Design

6.1 Primary Tables

- **users:** User accounts with credentials, roles, and active status
- **properties:** Property listings including rent, location, type, and status
- **viewing_requests:** Viewing appointment requests with dates and messages
- **applications:** Rental/purchase applications with financial details
- **favorites:** User-property bookmarking join table

6.2 Relationships

- **User → Property:** One-to-Many (One owner can list multiple properties)
- **User → ViewingRequest:** One-to-Many (One user can make multiple requests)
- **User → Application:** One-to-Many (One user can submit multiple applications)
- **Property → ViewingRequest:** One-to-Many (One property receives many requests)
- **Property → Application:** One-to-Many (One property receives many applications)
- **User → Favorite → Property:** Many-to-Many through favorites table

7. Security Implementation

7.1 Authentication Mechanism

- **Session-Based Authentication:** User credentials stored securely in HttpSession
- **Role Storage:** User role persisted in session for access control checks
- **Automatic Redirection:** Role-based routing after successful login

7.2 Role-Based Access Control (RBAC)

- **Controller-Level Checks:** Each method verifies user role before execution
- **Ownership Verification:** Users can only modify their own resources
- **Admin Override:** Administrators have elevated permissions across the platform

7.3 Data Validation

- **Multi-Layer Validation:**
  - **Frontend:** HTML5 patterns for immediate user feedback
  - **JavaScript:** Client-side validation before form submission
  - **Backend:** Spring validation annotations and custom business logic
- **Validation Rules:**
  - Email: Gmail-only validation with pattern matching
  - Phone: 10-15 digits, numeric characters only
  - Names: Alphabetic characters, 2-50 character length
  - Password: Minimum 6 characters

7.4 SQL Injection Protection

- JPA/Hibernate parameterized queries sanitize all database inputs
- Repository pattern abstracts direct SQL execution

8. Workflows

8.1 Property Approval Workflow

Owner/Agent creates property → Status: PENDING → Admin reviews listing →
Admin decision → Status: APPROVED or REJECTED → Visibility updated

Steps:

1. Owner/Agent submits property with complete details
2. Property enters system with PENDING status
3. Property appears in admin's pending queue
4. Admin reviews and makes approval decision
5. Status updates to APPROVED (visible to buyers/renters) or REJECTED (hidden)

8.2 Viewing Request Workflow

Buyer/Renter/Agent submits request → Status: PENDING → Owner/Agent reviews →
Owner/Agent decision → Status: APPROVED or DISAPPROVED → User notified

Steps:

1. User selects property and preferred viewing date/time
2. Request created with PENDING status
3. Owner/Agent receives notification of incoming request
4. Owner/Agent reviews requester information
5. Decision updates status to APPROVED or DISAPPROVED
6. Both parties can view updated status

8.3 Application Workflow

Buyer/Renter/Agent submits application → Status: PENDING → Owner/Agent reviews → Owner/Agent decision → Status: ACCEPTED or REJECTED → Transaction proceeds

Steps:

1. User completes application form (income, employment, message)
2. Application submitted with PENDING status
3. Owner/Agent reviews financial qualifications
4. Decision updates status to ACCEPTED or REJECTED
5. Applicant tracks status in personal dashboard

8.4 User Management Workflow

Admin accesses user management → Selects user account →
Activate/Deactivate action → User access granted or revoked

Steps:

1. Admin views list of all registered users
2. Admin identifies problematic or inactive accounts
3. Admin toggles account active status
4. Deactivated users cannot log in until reactivated

9. Conclusion

RealEstate Pro successfully delivers a comprehensive property management solution that addresses the complex needs of the real estate market. By implementing a robust MVC architecture with Spring Boot and Hibernate/JPA, the platform demonstrates enterprise-level software development principles including dependency injection, ORM mapping, and layered application design.

The system's key achievement lies in its multi-role architecture that accommodates five distinct user types, each with tailored capabilities and appropriate access controls. The implementation of a three-tier property approval workflow ensures content quality while the favourites system, viewing request coordination, and application processing features facilitate seamless interaction between all stakeholders.

The real estate agent role provides unique versatility, enabling professionals to represent multiple parties simultaneously managing property listings for owners while assisting buyers and renters with property discovery and application submission. This intermediary capability reflects real-world real estate transactions and adds significant value to the platform.

Technical implementation leverages Spring Data JPA repositories to eliminate traditional DAO complexity, while multi-layer validation ensures data integrity across frontend and backend systems. Session-based authentication with role verification provides foundational security, with clear pathways for enhancement through Spring Security framework integration in future iterations.

The platform provides a scalable foundation for real estate management that can be extended with additional features such as photo galleries, integrated messaging systems, and advanced analytics. The current implementation successfully demonstrates proficiency in enterprise software development, database design, and full-stack web application architecture.

10. Appendix: Service Layer Implementation

10.1 UserService.java

The UserService class handles all business logic related to user operations, including registration, authentication, and account management with activation/deactivation capabilities.

java

```
package com.realestate.management.service;

import com.realestate.management.dao.UserDao;
import com.realestate.management.model.User;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import java.time.LocalDateTime;
import java.util.List;

@Service
public class UserService {

  private final UserDao userDao;

  @Autowired
  public UserService(UserDao userDao) {
    this.userDao = userDao;
  }

  // Register a new user
  public User registerUser(User user) {
    // Check if email already exists
    if (userDao.emailExists(user.getEmail())) {
      throw new RuntimeException("Email already registered");
    }

    // Set timestamps
    user.setCreatedAt(LocalDateTime.now());
    user.setUpdatedAt(LocalDateTime.now());
    user.setActive(true);

    // Save user
```

```
    userDao.save(user);
    return user;
}

// Login user
public User loginUser(String email, String password) {
    User user = userDao.findByEmail(email);

    if (user == null) {
        throw new RuntimeException("User not found");
    }

    if (!user.getPassword().equals(password)) {
        throw new RuntimeException("Invalid password");
    }

    if (!user.isActive()) {
        throw new RuntimeException("Account is deactivated");
    }

    return user;
}

// Get user by ID
public User getUserById(Long id) {
    return userDao.findById(id);
}

// Get user by email
public User getUserByEmail(String email) {
    return userDao.findByEmail(email);
}

// Get all users
public List<User> getAllUsers() {
    return userDao.findAll();
}

// Get users by role
public List<User> getUsersByRole(User.UserRole role) {
    return userDao.findByRole(role);
```

```
    }

    // Update user
    public User updateUser(User user) {
        user.setUpdatedAt(LocalDateTime.now());
        userDao.update(user);
        return user;
    }

    // Deactivate user
    public void deactivateUser(Long id) {
        User user = userDao.findById(id);
        if (user != null) {
            user.setActive(false);
            user.setUpdatedAt(LocalDateTime.now());
            userDao.update(user);
        }
    }

    // Activate user
    public void activateUser(Long id) {
        User user = userDao.findById(id);
        if (user != null) {
            user.setActive(true);
            user.setUpdatedAt(LocalDateTime.now());
            userDao.update(user);
        }
    }

    // Check if email exists
    public boolean emailExists(String email) {
        return userDao.emailExists(email);
    }
}
```

Listing 1: UserService Implementation

10.2 PropertyService.java

The PropertyService class manages property-related business logic including CRUD operations, status management, search functionality, and filtering capabilities.

java

```
package com.realestate.management.service;

import com.realestate.management.dao.PropertyDao;
import com.realestate.management.model.Property;
import com.realestate.management.model.User;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import java.math.BigDecimal;
import java.time.LocalDateTime;
import java.util.List;

@Service
public class PropertyService {

  private final PropertyDao propertyDao;

  @Autowired
  public PropertyService(PropertyDao propertyDao) {
    this.propertyDao = propertyDao;
  }

  // Create a new property
  public Property createProperty(Property property, User owner) {
    property.setOwner(owner);
    property.setCreatedAt(LocalDateTime.now());
    property.setUpdatedAt(LocalDateTime.now());
    property.setStatus(Property.PropertyStatus.PENDING);
    propertyDao.save(property);
    return property;
  }

  // Update a property
  public Property updateProperty(Property property) {
    property.setUpdatedAt(LocalDateTime.now());
    propertyDao.update(property);
```

```
    return property;
}

// Delete a property
public void deleteProperty(Long id) {
    Property property = propertyDao.findById(id);
    if (property != null) {
        propertyDao.delete(property);
    }
}

// Get property by ID
public Property getPropertyById(Long id) {
    return propertyDao.findById(id);
}

// Get all properties
public List<Property> getAllProperties() {
    return propertyDao.findAll();
}

// Get all approved properties (for public listing)
public List<Property> getAllApprovedProperties() {
    return propertyDao.findAllApproved();
}

// Get properties by owner
public List<Property> getPropertiesByOwner(User owner) {
    return propertyDao.findByOwner(owner);
}

// Get properties by status
public List<Property> getPropertiesByStatus(Property.PropertyStatus status) {
    return propertyDao.findByStatus(status);
}

// Get pending properties (for admin)
public List<Property> getPendingProperties() {
    return propertyDao.findByStatus(Property.PropertyStatus.PENDING);
}
```

```
// Approve a property
public Property approveProperty(Long id) {
    Property property = propertyDao.findById(id);
    if (property != null) {
        property.setStatus(Property.PropertyStatus.APPROVED);
        property.setUpdatedAt(LocalDateTime.now());
        propertyDao.update(property);
    }
    return property;
}

// Reject a property
public Property rejectProperty(Long id) {
    Property property = propertyDao.findById(id);
    if (property != null) {
        property.setStatus(Property.PropertyStatus.REJECTED);
        property.setUpdatedAt(LocalDateTime.now());
        propertyDao.update(property);
    }
    return property;
}

// Mark property as sold
public Property markAsSold(Long id) {
    Property property = propertyDao.findById(id);
    if (property != null) {
        property.setStatus(Property.PropertyStatus.SOLD);
        property.setUpdatedAt(LocalDateTime.now());
        propertyDao.update(property);
    }
    return property;
}

// Mark property as rented
public Property markAsRented(Long id) {
    Property property = propertyDao.findById(id);
    if (property != null) {
        property.setStatus(Property.PropertyStatus.RENTED);
        property.setUpdatedAt(LocalDateTime.now());
        propertyDao.update(property);
    }
```

```
    return property;
}

// Get properties for sale
public List<Property> getPropertiesForSale() {
    return propertyDao.findByListingType(Property.ListingType.SALE);
}

// Get properties for rent
public List<Property> getPropertiesForRent() {
    return propertyDao.findByListingType(Property.ListingType.RENT);
}

// Get properties by type
public List<Property> getPropertiesByType(Property.PropertyType type) {
    return propertyDao.findByPropertyType(type);
}

// Get properties by city
public List<Property> getPropertiesByCity(String city) {
    return propertyDao.findByCity(city);
}

// Get properties by price range
public List<Property> getPropertiesByPriceRange(BigDecimal minPrice,
                                                BigDecimal maxPrice) {
    return propertyDao.findByPriceRange(minPrice, maxPrice);
}

// Search properties with filters
public List<Property> searchProperties(String city,
                                       Property.PropertyType propertyType,
                                       Property.ListingType listingType,
                                       BigDecimal minPrice,
                                       BigDecimal maxPrice,
                                       Integer bedrooms) {
    return propertyDao.searchProperties(city, propertyType, listingType,
                                        minPrice, maxPrice, bedrooms);
}

// Count properties by owner
```

```
    public Long countPropertiesByOwner(User owner) {
        return propertyDao.countByOwner(owner);
    }

    // Count pending properties
    public Long countPendingProperties() {
        return propertyDao.countPending();
    }

    // Check if user is the owner of the property
    public boolean isOwner(Long propertyId, Long userId) {
        Property property = propertyDao.findById(propertyId);
        return property != null &&
               property.getOwner().getId().equals(userId);
    }
}
```

Listing 2: PropertyService Implementation

10.3 ViewingService.java

The ViewingService class implements the viewing request workflow, managing property viewing appointments between buyers/renters and property owners with approval mechanisms.

java

```
package com.realestate.management.service;

import com.realestate.management.dao.ViewingDao;
import com.realestate.management.dao.PropertyDao;
import com.realestate.management.dao.UserDao;
import com.realestate.management.model.PropertyViewing;
import com.realestate.management.model.Property;
import com.realestate.management.model.User;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import java.time.LocalDateTime;
import java.util.List;
```

```
@Service
public class ViewingService {

  private final ViewingDao viewingDao;
  private final PropertyDao propertyDao;
  private final UserDao userDao;

  @Autowired
  public ViewingService(ViewingDao viewingDao, PropertyDao propertyDao,
            UserDao userDao) {
    this.viewingDao = viewingDao;
    this.propertyDao = propertyDao;
    this.userDao = userDao;
  }

  // Request a viewing
  @Transactional
  public PropertyViewing requestViewing(Long propertyId, Long userId,
                        LocalDateTime viewingDate,
                        String message) {
    // Check if already has pending/approved request
    if (viewingDao.hasViewingRequest(userId, propertyId)) {
      throw new RuntimeException(
        "You already have a viewing request for this property");
    }

    Property property = propertyDao.findById(propertyId);
    User user = userDao.findById(userId);

    if (property == null || user == null) {
      throw new RuntimeException("Property or User not found");
    }

    // Check if viewing date is in the future
    if (viewingDate.isBefore(LocalDateTime.now())) {
      throw new RuntimeException("Viewing date must be in the future");
    }

    PropertyViewing viewing = new PropertyViewing(property, user,
                      viewingDate, message);
    viewingDao.save(viewing);
```

```
    return viewing;
}

// Get viewing by ID
public PropertyViewing getViewingById(Long id) {
    return viewingDao.findById(id);
}

// Get all viewings for a user
public List<PropertyViewing> getUserViewings(Long userId) {
    return viewingDao.findByUser(userId);
}

// Get all viewings for a property
public List<PropertyViewing> getPropertyViewings(Long propertyId) {
    return viewingDao.findByProperty(propertyId);
}

// Get all viewing requests for property owner
public List<PropertyViewing> getOwnerViewingRequests(Long ownerId) {
    return viewingDao.findByPropertyOwner(ownerId);
}

// Get pending viewing requests for owner
public List<PropertyViewing> getPendingViewingsForOwner(Long ownerId) {
    return viewingDao.findPendingByOwner(ownerId);
}

// Get upcoming approved viewings for user
public List<PropertyViewing> getUpcomingViewings(Long userId) {
    return viewingDao.findUpcomingByUser(userId);
}

// Approve viewing request
@Transactional
public void approveViewing(Long viewingId) {
    PropertyViewing viewing = viewingDao.findById(viewingId);
    if (viewing != null) {
        viewing.setStatus(PropertyViewing.ViewingStatus.APPROVED);
        viewingDao.update(viewing);
    }
```

```
}

// Reject viewing request
@Transactional
public void rejectViewing(Long viewingId) {
  PropertyViewing viewing = viewingDao.findById(viewingId);
  if (viewing != null) {
    viewing.setStatus(PropertyViewing.ViewingStatus.REJECTED);
    viewingDao.update(viewing);
  }
}

// Cancel viewing
@Transactional
public void cancelViewing(Long viewingId) {
  PropertyViewing viewing = viewingDao.findById(viewingId);
  if (viewing != null) {
    viewing.setStatus(PropertyViewing.ViewingStatus.CANCELLED);
    viewingDao.update(viewing);
  }
}

// Mark viewing as completed
@Transactional
public void completeViewing(Long viewingId) {
  PropertyViewing viewing = viewingDao.findById(viewingId);
  if (viewing != null) {
    viewing.setStatus(PropertyViewing.ViewingStatus.COMPLETED);
    viewingDao.update(viewing);
  }
}

// Count pending viewings for owner
public Long countPendingViewings(Long ownerId) {
  return viewingDao.countPendingByOwner(ownerId);
}

// Check if user owns the property
public boolean isPropertyOwner(Long viewingId, Long userId) {
  PropertyViewing viewing = viewingDao.findById(viewingId);
  return viewing != null &&
```

```
            viewing.getProperty().getOwner().getId().equals(userId);
    }

    // Check if user is the requester
    public boolean isRequester(Long viewingId, Long userId) {
        PropertyViewing viewing = viewingDao.findById(viewingId);
        return viewing != null &&
            viewing.getUser().getId().equals(userId);
    }
}
```

Listing 3: ViewingService - Viewing Request Management

10.4 ApplicationService.java

The ApplicationService class handles the complete application lifecycle for rental and purchase transactions, including submission validation, status management, and owner approval workflows.

```
java
package com.realestate.management.service;

import com.realestate.management.dao.ApplicationDao;
import com.realestate.management.dao.PropertyDao;
import com.realestate.management.dao.UserDao;
import com.realestate.management.model.Application;
import com.realestate.management.model.Property;
import com.realestate.management.model.User;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import java.util.List;

@Service
public class ApplicationService {

    private final ApplicationDao applicationDao;
    private final PropertyDao propertyDao;
    private final UserDao userDao;
```

```
@Autowired
public ApplicationService(ApplicationDao applicationDao,
                          PropertyDao propertyDao, UserDao userDao) {
  this.applicationDao = applicationDao;
  this.propertyDao = propertyDao;
  this.userDao = userDao;
}

// Submit an application
@Transactional
public Application submitApplication(Long propertyId, Long userId,
                                     Application application) {
  // Check if user already has an application for this property
  if (applicationDao.hasApplication(userId, propertyId)) {
    throw new RuntimeException(
      "You already have an application for this property");
  }

  Property property = propertyDao.findById(propertyId);
  User user = userDao.findById(userId);

  if (property == null || user == null) {
    throw new RuntimeException("Property or User not found");
  }

  // Set the relationships
  application.setProperty(property);
  application.setUser(user);

  // Set application type based on property listing type
  if (property.getListingType() == Property.ListingType.SALE) {
    application.setApplicationType(
      Application.ApplicationType.PURCHASE);
  } else {
    application.setApplicationType(
      Application.ApplicationType.RENTAL);
  }

  applicationDao.save(application);
  return application;
}
```

```
// Get application by ID
public Application getApplicationById(Long id) {
    return applicationDao.findById(id);
}

// Get all applications for a user
public List<Application> getUserApplications(Long userId) {
    return applicationDao.findByUser(userId);
}

// Get all applications for a property
public List<Application> getPropertyApplications(Long propertyId) {
    return applicationDao.findByProperty(propertyId);
}

// Get all applications for property owner
public List<Application> getOwnerApplications(Long ownerId) {
    return applicationDao.findByPropertyOwner(ownerId);
}

// Get pending applications for owner
public List<Application> getPendingApplicationsForOwner(Long ownerId) {
    return applicationDao.findPendingByOwner(ownerId);
}

// Approve application
@Transactional
public void approveApplication(Long applicationId) {
    Application application = applicationDao.findById(applicationId);
    if (application != null) {
        application.setStatus(Application.ApplicationStatus.APPROVED);
        applicationDao.update(application);
    }
}

// Reject application
@Transactional
public void rejectApplication(Long applicationId) {
    Application application = applicationDao.findById(applicationId);
    if (application != null) {
```

```
      application.setStatus(Application.ApplicationStatus.REJECTED);
      applicationDao.update(application);
    }
}

// Mark as under review
@Transactional
public void markUnderReview(Long applicationId) {
    Application application = applicationDao.findById(applicationId);
    if (application != null) {
      application.setStatus(
          Application.ApplicationStatus.UNDER_REVIEW);
      applicationDao.update(application);
    }
}

// Withdraw application
@Transactional
public void withdrawApplication(Long applicationId) {
    Application application = applicationDao.findById(applicationId);
    if (application != null) {
      application.setStatus(Application.ApplicationStatus.WITHDRAWN);
      applicationDao.update(application);
    }
}

// Count pending applications for owner
public Long countPendingApplications(Long ownerId) {
    return applicationDao.countPendingByOwner(ownerId);
}

// Check if user owns the property
public boolean isPropertyOwner(Long applicationId, Long userId) {
    Application application = applicationDao.findById(applicationId);
    return application != null &&
        application.getProperty().getOwner().getId().equals(userId);
}

// Check if user is the applicant
public boolean isApplicant(Long applicationId, Long userId) {
    Application application = applicationDao.findById(applicationId);
```

```
        return application != null &&
                application.getUser().getId().equals(userId);
    }

    // Count applications by user
    public Long countUserApplications(Long userId) {
        return applicationDao.countByUser(userId);
    }
}
```

Listing 4: ApplicationService - Application Processing Workflow