1. Introduction

- Purpose: Brief overview of the application and what the TDD covers.
- Scope: Features and modules included.

2. Architecture Overview

- Diagram: High-level diagram showing frontend, backend, database, and integrations.
- Description: Summary of the architecture.

3. Frontend Design

- Pages/Components: List main pages and components.
- Navigation Flow: Diagram or description of user navigation.
- State Management: If applicable, describe how state is managed (e.g., in JS).

4. Backend Design

- Express.js Structure: Describe folder structure (routes, controllers, models, etc.).
- API Endpoints: Table with endpoint, method, parameters, and description.
  Middleware: List and describe middleware functions.

5. Database Design

- ER Diagram: Visual representation of tables and relationships.
- Table Definitions: For each table, list fields, types, constraints, and indexes.
  Sample Data: (Optional) Example rows for key tables.

6. Real-Time Notifications (Socket.io)

- Event List: Table of events (name, direction: client/server, payload).
- Flow Diagram: Sequence of real-time interactions.

7. Email Sending (Nodemailer)

- Triggers: When are emails sent (e.g., registration, password reset)?
- Templates: List and describe email templates used.
  Configuration: SMTP settings and security considerations.

8. Security Considerations

- Authentication/Authorization: How are users authenticated? Which endpoints are protected?
- Data Validation: How is input validated/sanitized?
  Other: CORS, rate limiting, etc.

9. Deployment Architecture

- Environments: Describe dev, staging, production setups.
- Dependencies: List external services, libraries, and tools.

10. Appendix

- Glossary: Define any domain-specific terms.
- References: Link to code, diagrams, or external docs.

10.1. Enhanced System Architecture Diagram

Purpose: This diagram illustrates the complete technical architecture of the resident welfare application, showing how different tiers interact and the specific technologies used.

Key Components:

- **Client Tier**: Web browsers (desktop and mobile) that consume the application
- **Load Balancer**: Optional component for scaling (can be added later)
- **Application Tier**: The core Express.js server running on Node.js with multiple middleware layers
- **Business Logic**: Separated service layers for different functional domains
- **Email System**: Dedicated subsystem for handling asynchronous email notifications
- **Data Tier**: MySQL database with connection pooling and static file storage
- **External Services**: Gmail SMTP for email delivery and environment configuration

Technical Details:

- Server runs on port 3000 with configurable host binding
- Database connection pool limited to 10 concurrent connections
- Email system uses Gmail SMTP on port 465 with SSL
- Static files served directly by Express for performance

Benefits: This architecture provides clear separation of concerns, scalability through connection pooling, and reliable email delivery through queuing.

10.2. Comprehensive Database Schema (ERD)

Purpose: This Entity Relationship Diagram shows the complete data model with all tables, relationships, constraints, and data types used in the application.

Key Tables:

- **USERS**: Core user authentication and basic information
- **USER_PROFILES**: Extended user information for resident management
- **COMMUNITY_POSTS & COMMENTS**: Social features for community interaction
- **NOTIFICATIONS**: In-app messaging and alert system
- **PAYMENTS**: Financial transaction management
- **COMPLAINTS**: Issue tracking and resolution system
- **PASSWORD_RESETS**: Security feature for account recovery

Relationship Types:

- One-to-One: User to UserProfile (each user has one profile)
- One-to-Many: User to Posts, Comments, Payments, Complaints
- Foreign Key Constraints: Ensure data integrity across tables

Data Integrity Features:

- Primary keys with auto-increment for unique identification
- Foreign key constraints to maintain referential integrity
- Unique constraints on email addresses to prevent duplicates
- Enum types for status fields to ensure valid values
  Timestamp fields with automatic updates for audit trails

10.3. Complete API Architecture with Middleware

Purpose: This diagram shows the complete HTTP request flow through the Express.js middleware stack and how requests are routed to different endpoint groups.

Middleware Stack:

1. **CORS Middleware**: Handles cross-origin requests for web security

2. **Static File Middleware**: Serves HTML, CSS, and JavaScript files
3. **JSON Parser**: Processes JSON request bodies
4. **Body Parser**: Handles form data and other content types
5. **Express Router**: Routes requests to appropriate handlers

Endpoint Groups:

- **Authentication**: User registration, login, password reset
- **Community**: Post creation, commenting, social features
- **User Management**: Profile management, notifications
- **Payment System**: Payment processing, history, statistics
- **Complaint System**: Issue submission and tracking
- **Admin Panel**: Administrative functions and oversight

Backend Services:

- **Database Service**: Handles all database operations
- **Email Service**: Manages email notifications
- **Validation**: Input validation and sanitization
  Error Handler: Centralized error processing and logging

10.4. Enhanced Email System with Error Handling

Purpose: This flowchart details the asynchronous email notification system, including queue management, error handling, and different email types.

Queue Management:

- Emails are added to an in-memory queue to prevent blocking
- Single processor handles queue sequentially to avoid overwhelming SMTP server
- Promise-based system allows for proper error handling and logging

Email Types:

- **Welcome Email**: Sent upon user registration
- **Login Notifications**: Security alerts for account access
- **Post Notifications**: Community engagement alerts
- **Payment Receipts**: Transaction confirmations
- **Complaint Updates**: Status change notifications

- **Password Reset**: Security recovery emails
- **Admin Alerts**: System notifications for administrators

Error Handling:

- Connection timeouts with retry logic
- Authentication failures logged and reported
- Invalid email addresses filtered out
- Rate limiting to prevent spam classification
- Server unavailability handled gracefully

Benefits: This system ensures reliable email delivery without blocking the main application, provides comprehensive logging, and handles various failure scenarios.

10.5. Complete Use Case Diagram

Purpose: This diagram maps all possible user interactions with the system, showing what different types of users can accomplish.

Actor Types:

- **Resident User**: Regular community members with standard access
- **Admin**: Administrative users with elevated privileges
- **System**: Automated processes and background tasks

Use Case Categories:

- **Authentication**: Account creation, login, password management
- **Community Features**: Social interaction, posting, commenting
- **Profile Management**: Personal information maintenance
- **Notification System**: Communication and alerts
- **Payment System**: Financial transaction management
- **Complaint System**: Issue reporting and resolution
- **Admin Functions**: User management, system oversight
- **System Processes**: Automated background tasks

Access Control: The diagram clearly shows which actors can perform which actions, establishing the security model and user permissions.

10.6. State Diagrams for Key Entities

Purpose: These state diagrams show the lifecycle and possible state transitions for the two most important business entities: Payments and Complaints.

Payment Lifecycle:

- **Created**: Admin creates a new payment requirement
- **Pending**: Payment awaits user action (sends reminders)
- **Completed**: User successfully processes payment (sends receipt)
- **Failed**: Payment processing unsuccessful (allows retry)

Complaint Lifecycle:

- **Submitted**: User creates new complaint
- **Pending**: Awaiting admin review (notifies administrators)
- **In Progress**: Admin actively working on resolution
- **Resolved**: Issue closed (final notification sent)

Business Rules: These states enforce business logic, ensure proper workflow, and trigger appropriate notifications at each transition.

10.7. Detailed Sequence Diagram - Community Post Flow

Purpose: This sequence diagram shows the complete flow of creating a community post, including all database operations, notifications, and email processing.

Flow Steps:

1. User submits post data via API
2. API validates input and creates post in database
3. Immediate response sent to user (non-blocking)
4. Asynchronous processing begins for notifications
5. System retrieves all other users from database
6. Notifications created for each user
7. Email tasks queued for each recipient
8. Email queue processor sends notifications
9. Users can retrieve notifications via API

Performance Considerations:

- Immediate response to user prevents UI blocking
- Asynchronous processing handles time-consuming operations
- Batch operations optimize database performance
  Email queue prevents SMTP server overload

10.8. Deployment Architecture

Purpose: This diagram shows how the application should be deployed in a production environment, including all necessary infrastructure components.

Production Components:

- **Web Server**: Node.js application managed by PM2 process manager
- **Database Server**: MySQL with automated backup scheduling
- **Email Service**: Gmail SMTP with connection pooling
- **File System**: Static assets and application logs
- **Network Security**: Firewall configuration and SSL certificates

Configuration Details:

- Application runs on port 3000 with external access

- Database on port 3306 with restricted access
- SSL encryption for secure communication
- Environment variables for configuration management
- Scheduled database backups for data protection

Scalability Considerations: This architecture can be extended with load balancers, multiple application instances, and database clustering as the user base grows.

10.9. Class Diagram - Core Business Objects

Purpose: This object-oriented design diagram shows the main business entities, their properties, methods, and relationships within the application.

Core Classes:

- **User**: Handles authentication and basic user operations
- **UserProfile**: Manages extended user information
- **Admin**: Extends User with administrative capabilities
- **CommunityPost**: Manages community content creation
- **Comment**: Handles post interactions
- **Payment**: Manages financial transactions
- **Complaint**: Handles issue tracking
- **Notification**: Manages in-app messaging

Service Classes:

- **EmailService**: Centralized email management
- **DatabaseService**: Database abstraction layer
- **EmailTask**: Individual email processing units

Data Transfer Objects:

- **PaymentStats**: Payment analytics data
- **ComplaintStats**: Complaint analytics data
- **DashboardStats**: Administrative overview data

Design Patterns:

- **Service Layer Pattern**: Separates business logic from data access

- **Queue Pattern**: Manages asynchronous email processing
- **Factory Pattern**: Creates different types of email content
- **Repository Pattern**: Abstracts database operations

Benefits: This design provides clear separation of concerns, reusable components, and maintainable code structure that follows object-oriented principles.

These diagrams collectively provide a complete technical specification for the resident welfare application, covering architecture, data design, API structure, business processes, deployment, and object-oriented design. They serve as a comprehensive reference for development, maintenance, and future enhancements.

11. v

    11.1. n