***Software Requirements Specification***

**Home Utility Tracker Project**

***Version<2.0>***

**By: Isha Limbasiya**

**Date: 13/06/2025**

1. **Introduction** 
   1. **Purpose**

The purpose of this Software Requirements Specification (SRS) is to define and document the functionalities, features, and technical requirements for the Home Utility Tracker App. This application is intended to assist household users, PG/flat owners, and society managers in efficiently managing their utility-related tasks including expense tracking, rent collection, society maintenance, appliance servicing, and reminders.

The SRS provides a foundation for the system’s design, development, testing, deployment, and future enhancements.

* 1. **Scope**

The Home Utility Tracker App is a full-stack web application built using Spring Framework (Java 8), Spring MVC, Hibernate, MySQL, and Thymeleaf as the frontend template engine. The system will allow users to:

* Track and manage household utility bills (electricity, water, gas, rent, internet)
* Set reminders for due dates and appliance servicing
* Manage PGs/flats, tenants, and monthly rent
* Log and track maintenance payments and expenses at the society level
* Provide service contact information for repair and support
* Use role-based access to restrict functionalities based on the user's role (Household User, PG Owner, or Society Manager)

The application will expose RESTful services and offer a web-based interface using Thymeleaf. In this phase, mobile or external integrations are out of scope but may be considered for future releases.

* 1. **Overview**

This document includes detailed information about:

* The system's intended functionality and use cases
* User roles and interactions
* Technical and non-functional requirements
* Database and architectural design (via ERD and DFD)
* API and UI interface structures
* Tools and technologies used for development and deployment

The SRS aims to ensure all stakeholders have a common understanding of the system’s goals and constraints, and serves as a contractual document between developers and project evaluators.

1. **General Description**
   1. **Product Perspective**

The Home Utility Tracker App is an independent, modular web application that aims to simplify utility and property management for individual users, PG/flat owners, and society administrators. It is built using the Spring Framework (Java 8+) and follows a layered architecture with clear separation between the frontend (Thymeleaf templates), backend logic (Spring MVC + RESTful APIs), and persistence layer (Hibernate + MySQL).

The application integrates Thymeleaf as the server-side view engine to render dynamic HTML pages and interact seamlessly with backend services.

This system is designed for scalability and can be enhanced in the future with features like mobile support, external APIs, and payment integrations. REST APIs are also provided for interoperability and mobile/backend integration.gration. It will support Docker-based containerization and AWS deployment.

* 1. **Product Functions**

The key functionalities of the system are categorized into the following modules:

**Household Utility Management:**

This module allows household users to record and manage their utility bills such as electricity, water, gas, internet, and rent. Users can categorize bills (e.g., fixed, recurring, variable), mark them as paid or unpaid, and receive due-date reminders. It also provides monthly and yearly summaries for easy expense tracking.

**PG/Flat Rental Management:**

Designed for PG or flat owners, this module helps in registering multiple rental units, managing tenant information, and tracking rent payments. Owners can log shared utility costs, receive reminders for rent due dates, and maintain service or maintenance requests for each property.

**Society Fund Management:**

This module enables society managers to register flats and their corresponding owners, record maintenance fee payments, and log common society expenses like cleaning or security. It can alert managers about defaulters and generate financial summaries and downloadable reports.

**Appliance Service Module:**

Users can register appliances such as air conditioners or water purifiers along with their brand and model details. Service intervals can be defined (e.g., every 180 days), and the system will notify users before the next service is due. Contact details of technicians can also be stored for each appliance type.

**User Management:**

This module supports user registration and login using email and password. During registration, users are assigned roles such as Household User, PG Owner, or Society Manager. Based on the assigned role, users are provided access to specific modules and dashboards within the application.

* 1. **User Classes and Characteristics**
* **Household User:** Regular user managing personal bills, appliances, and reminders
* **Society Manager:** Responsible for managing society-wide data like maintenance collections and reports
* **PG Owner:** Manages tenants, rental units, rent payments, shared utilities, and flat services
  1. **Technology Used**
* Frontend: HTML5, CSS3, Bootstrap, jQuery, Thymeleaf
* Backend: Java 8+, Spring Framework, Spring MVC, RESTful APIs
* Database: MySQL
* ORM & Persistence: Hibernate, Spring Data JPA
* Build Tools: Maven
* DevOps: Git, Jenkins (CI/CD), Docker (Containerization)
* Deployment: AWS EC2 / ECS

1. **Functional Requirements**

This section outlines the core features and behavior of the system based on the functional modules identified. Each feature is labeled for traceability.

* 1. **Household Utility Management**
* Add/update/delete utility records
* Mark bills as Paid/Unpaid with status tracking
* Set and receive due-date reminders
* View monthly and yearly expense summaries
* Categorize utility bills under fixed and recurring categories
  1. **Society Fund Management**
* Register flat/house numbers with owner details
* Log maintenance fee payments from each flat and show pending dues
* Track pending maintenance fees and generate alerts for defaulters
* Add society-level expenses such as cleaning, security, or repair
* Calculate monthly income, expenses, and net balance for the society.
* Download or view reports for specific time periods
  1. **PG/Flat Rental Management**
* Register multiple PGs/flats
* Allow adding, updating, and deleting tenant records with move-in/move-out dates
* Track monthly rent payments per tenant
* Owners shall receive rent due reminders
* Shared utility costs can be recorded and split among tenants
* Maintenance or service requests shall be logged per property
  1. **Appliance Service Reminder Module**
* Register appliances (e.g., AC, water purifier) with model and brand details
* Set service intervals (e.g., 180 days) for each appliance
* Track the last service date and calculate the next due date
* Send a reminder when a service is due
* View a list of appliances with upcoming service dates
* Provide an option to save technician contact information and initiate a call
  1. **User Management**
* Support user registration and login in using email and password
* Assigned a role to each user: Household User, Society Manager, or PG Owner.
* Role-based dashboards and access control shall determine available features per user
* User credentials shall be securely stored with encryption

1. **Interface Requirements**
   1. **User Interface (UI)**

The application will use Thymeleaf as the templating engine to render dynamic HTML pages via Spring MVC. Key UI components will include:

* Role-based dashboards (Household User, PG Owner, Society Manager)
* Forms for entering utility bills, rent records, tenant details, etc.
* Summary pages and reports (monthly expenses, maintenance collections)
* Responsive layout using Bootstrap for mobile-friendly access

Each UI page will be mapped to a corresponding controller endpoint, with data passed through model attributes.

* 1. **Application Programming Interface (API)**

The application exposes a comprehensive set of RESTful APIs, categorized by modules and mapped to user roles. These APIs are designed for interaction between the frontend (Thymeleaf or future mobile app) and backend services. All APIs accept and return data in JSON format and follow standard HTTP methods (GET, POST, PUT, DELETE). Future versions may include JWT-based authentication.

* **User & Authentication APIs**

|  |  |  |
| --- | --- | --- |
| **Endpoint** | **Method** | **Description** |
| /api/auth/register | POST | Register a new user |
| /api/auth/login | POST | Authenticate user login |
| /api/users/{id} | GET | Get user details by ID |
| /api/users/role/{role} | GET | Get users by role (optional) |

* **Household Utility APIs**

|  |  |  |
| --- | --- | --- |
| **Endpoint** | **Method** | **Description** |
| /api/utilities | POST | Add a new utility bill |
| /api/utilities/{utilityId} | PUT | Update a utility bill |
| /api/utilities/{utilityId} | DELETE | Delete a utility bill |
| /api/utilities/user/{userId} | GET | Get utility bills by user |
| /api/utilities/summary/{userId} | GET | Get monthly/yearly expense summary |

* **Society Management APIs**

|  |  |  |
| --- | --- | --- |
| **Endpoint** | **Method** | **Description** |
| /api/society/flats | POST | Register flat and owner |
| /api/society/flats | GET | View all registered flats |
| /api/society/payments | POST | Log maintenance payment |
| /api/society/payments/{flatId} | GET | View payment history by flat |
| /api/society/expenses | POST | Add society-level expenses |
| /api/society/expenses | GET | View all society expenses |
| /api/society/reports/monthly | GET | View monthly income vs expense summary |
| /api/society/defaulters | GET | List all flats with pending payments |

* **PG Management APIs**

|  |  |  |
| --- | --- | --- |
| **Endpoint** | **Method** | **Description** |
| /api/properties | POST | Add new flat or PG |
| /api/properties/user/{ownerId} | GET | Get all properties owned by a user |
| /api/tenants | POST | Add a new tenant |
| /api/tenants/{tenantId} | PUT | Update tenant info |
| /api/tenants/{tenantId} | DELETE | Delete a tenant |
| /api/tenants/property/{propertyId} | GET | Get tenants by property |
| /api/rent | POST | Log rent payment |
| /api/rent/tenant/{tenantId} | GET | View rent history per tenant |
| /api/shared-utilities | POST | Record shared utility cost |
| /api/shared-utilities/{propertyId} | GET | Get shared costs by property |
| /api/service-requests | POST | Log a service/maintenance request |
| /api/service-requests/{propertyId} | GET | View all service requests by property |

* **Appliance Service APIs**

|  |  |  |
| --- | --- | --- |
| **Endpoint** | **Method** | **Description** |
| /api/appliances | POST | Register an appliance |
| /api/appliances/user/{userId} | GET | View appliances by user |
| /api/appliances/{applianceId} | PUT | Update appliance details |
| /api/appliances/{applianceId} | DELETE | Delete an appliance |
| /api/reminders/upcoming/{userId} | GET | View upcoming service reminders |
| /api/technicians | POST | Add technician contact info |
| /api/technicians | GET | View all technician contacts |

All APIs will:

* Return responses in JSON format
* Be versioned *(/api/v1/)*
* Be protected using role-based authentication
  1. **Hardware Interfaces**

**Not Applicable.** The current system does not interact with any sensors or physical hardware. All data will be entered manually by users.

* 1. **Software Interfaces**
* **Database: MySQL for persistent storage, connected via Hibernate/JPA.**
* **Web Server: Embedded Tomcat (Spring Boot default).**
* **Build Tool: Maven for dependency management and project build.**
* **DevOps Tools: Jenkins for CI/CD, Docker for containerization.**
* **Node.js/NPM: Used only for frontend asset management if needed.**

1. **Performance Requirements**

* The application should handle up to 100 concurrent users without significant performance degradation.
* 95% of REST API calls should respond in under 1 second.
* Web pages rendered via Thymeleaf should load within 2 seconds on a standard broadband connection.
* System should support 10 transactions/second under load conditions (e.g., multiple utility updates).
* The backend should be optimized for batch reports and summaries without timeouts.

1. **Design Constrains**

* The application must be developed using:
* Java 8, Spring Framework, Spring MVC, Hibernate (JPA) for backend
* MySQL for database
* Thymeleaf as the frontend template engine
* Maven for project management and dependency resolution
* The architecture should follow MVC (Model-View-Controller) pattern with a layered structure.
* APIs must be RESTful, following standard HTTP methods and JSON format.
* The application must be Docker-compatible for deployment in containerized environments.
* The system must be deployable using Jenkins CI/CD pipelines.

1. **Non – Functional Attributes**

* **Security:** Role-based access control; password encryption (e.g., BCrypt); input validation to prevent SQL injection and XSS.
* **Maintainability:** Modular codebase with proper layering (Controller, Service, DAO, Model) and clean naming conventions.
* **Scalability:** Easily extendable to support future modules like payments or notifications; indexed database fields for performance.
* **Usability:** Responsive and user-friendly UI compatible with all screen sizes; minimal learning curve.
* **Portability:** Docker-based deployment ensures smooth migration across environments.
* **Availability:** 99.5% uptime with minimal planned maintenance (≤ 2 hours/month).

1. **Preliminary Schedule and Budget**
   1. **Schedule**

The preliminary development timeline (tentative based on individual or small team development):

|  |  |
| --- | --- |
| **Phase** | **Timeline** |
| Requirement Gathering | 1 days |
| System Design (ERD, DFD, APIs) | 1 days |
| Module Development (Backend) | 8-10 days |
| Integration & Testing | 1 days |
| Deployment on AWS (Docker) | 1 days |
| Documentation & Handover | 1 days |

Total Estimated Duration: **~3 weeks**

* 1. **Budget**

Since this is an academic or training project, no actual financial cost is involved. However, approximate cloud usage for hosting and CI/CD may include:

* AWS EC2 (Free Tier or ~$10–15/month)
* Domain & SSL (optional for production)
* Jenkins (Self-hosted) or GitHub Actions (Free Tier)

Total Project Budget: ₹0 to ₹1,200/month (if using paid services)

1. **Appendices**
   1. **Acronyms**

* **API** – Application Programming Interface
* **JWT** – JSON Web Token
* **CRUD** – Create, Read, Update, Delete
* **SRS** – Software Requirements Specification
* **CI/CD** – Continuous Integration / Continuous Deployment
* **DTO** – Data Transfer Object
* **ORM** – Object-Relational Mapping
* **UI –** User Interface
* **MVC** – Model View Controller
* **DB** – Database
  1. **Future Enhancement**
* Add SMS/Email notification support
* Integrate IoT-based sensors for automated readings (e.g., electric meters)
* Add vendor onboarding and customer support modules
* Mobile app (Android/iOS) for utility input and alerts
* Multi-language support for local users