***Software Requirements Specification***

**Home Utility Tracker Project**

***Version<1.0>***

**By: Isha Limbasiya**

**Date: 13/06/2025**

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

The purpose of this document is to define the Software Requirements Specification (SRS) for the *Home Utility and Property Management System*. This system is intended to help users manage and track household expenses, society maintenance funds, PG/flat rental activities, and appliance servicing schedules. The document will serve as a foundation for design, development, testing, and future enhancements of the application.

* 1. **Scope**

This project aims to provide a unified backend system (without frontend integration initially) that can:

* Track household utility bills such as electricity, gas, water, and rent.
* Manage society-level funds and maintenance collections.
* Handle rental and tenant details for PGs and flats.
* Send reminders for appliance servicing and provide contact details for local technicians.
* Allow role-based access for different types of users: household members, society managers, and PG owners.

The application will expose its features via RESTful APIs, built using Java, Spring Boot, JPA, and Hibernate. It will be containerized using Docker and deployed on cloud platforms like AWS. Jenkins will be used for CI/CD automation.

* 1. **Overview**

This SRS outlines both functional and non-functional requirements of the system. It begins with a high-level system description and follows with detailed use-case-specific requirements. The document includes descriptions of each module, their respective user interactions, expected inputs and outputs, and external dependencies. The goal is to develop a scalable, modular backend system that can later be integrated with a frontend interface or mobile application.

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

The Home Utility and Property Management System is an independent backend application designed to manage and track multiple categories of domestic and property-related information. It is built as a modular and service-oriented architecture that can later be extended with a web or mobile frontend. The system will be implemented using Spring Boot (Java), JPA, Hibernate for persistence, and will expose REST APIs for integration. 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 Tracker:** Enables users to record and manage utility bills such as electricity, gas, water, rent, and internet. It allows tracking of bill payments, reminders for due dates, and monthly summaries.
* **Society Fund Manager:** Used by society administrators to track maintenance fee collections, expenses, and generate financial reports for society-level accounting.
* **PG/Flat Rental Manager:** Lets landlords manage tenant information, rent payment tracking, move-in/move-out details, and split shared utilities.
* **Appliance Service Reminder Module:** Tracks appliances like ACs and reminds users when they are due for servicing. Allows contact with assigned technicians.
  1. **User Classes and Characteristics**
* **Household User:** Manages personal utility bills and receives reminders for services.
* **Society Manager:** Oversees fund collection and expense management for a residential society.
* **PG Owner:** Handles multiple rental units, tenant data, rent tracking, and utilities division.
  1. **Technology Used**
* Programming Language: Java 8+
* Frameworks: Spring Boot, Hibernate, JPA
* Database: MySQL
* Build Tools: Maven
* DevOps: Jenkins, Docker
* 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
* Set and receive due-date reminders
* View monthly summaries
* Categorize utility bills under user-defined categories
  1. **Society Fund Management**
* Register flat/house numbers with owner details
* Log maintenance fee payments from each flat
* 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
* Associate tenants with properties and track move-in/move-out dates
* Track monthly rent payments per tenant
* Send rent due reminders to the owner (and tenants, in future scope)
* Log and divide shared utility costs among tenants
* Maintain a log of service/maintenance requests for each flat
  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.
* Assigned a role to each user: Household User, Society Manager, or PG Owner.
* Role-based access shall determine which modules/features a user can interact with.

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

UI to be added later; backend APIs will be structured for easy integration.

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

The system will expose RESTful APIs for each major module. Key interface specifications include:

* **Authentication API**
  + - */api/auth/register*
    - */api/auth/login*
    - Uses JWT for token-based authentication
* **Household Utility APIs**
  + - */api/household/utilities* – CRUD operations for utility bills
    - */api/household/reminders* – Bill reminder endpoints
    - */api/household/reports* – Monthly/yearly expense summaries
* **Society Management APIs**
  + - */api/society/flats* – Manage flat and owner info
    - */api/society/payments* – Record & track maintenance payments
    - */api/society/expenses* – Add society-level expenses
    - */api/society/reports* – Generate income vs expense reports
* **PG Management APIs**
  + - */api/pg/properties* – CRUD for PG or flat properties
    - */api/pg/tenants* – Manage tenant details and occupancy
    - */api/pg/rent* – Track rent payments and dues
* **Appliance Service APIs**
  + - */api/appliances* – Register and update appliances
    - */api/appliances/reminders* – Track and notify for servicing
    - */api/appliances/contacts* – Store technician contact details

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 or PostgreSQL (configured via application properties)
* **Deployment:** Docker containers
* **Cloud Hosting:** AWS EC2 (initially); S3, RDS or other services may be integrated later
* **DevOps Toolchain:** Jenkins for CI/CD pipeline integration
* **Monitoring Tools:** Logs via console and/or external tools like AWS CloudWatch (optional)

1. **Performance Requirements**

* 95% of API calls should complete in <1s
* System should support 10 transactions/sec
* Support scaling via Docker containers
* Uptime: 99.5%
* Handle up to 100,000 records with backup/recovery

1. **Design Constrains**

* Use Java 17+, Spring Boot, Hibernate, JPA
* Follow RESTful API design and JSON formatting
* Use JWT for security
* Containerized using Docker
* Compatible with AWS and Jenkins pipelines

1. **Non – Functional Attributes**

* **Reliability:** System should perform consistently with minimal failures.
* **Availability:** Maintain at least 99.5% uptime with minimal downtime.
* **Maintainability:** Follow modular design and coding standards for easy updates.
* **Usability:** APIs should be easy to use and well-documented for future frontend integration.
* **Portability:** Fully containerized using Docker for easy deployment across environments.
* **Security:** Secure APIs using JWT, hash passwords, and enforce role-based access control.

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
  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