Requirement Analysis

Functional requirements

1. **User Authentication and Authorization:**

a. User Registration:

- Create a "Users" table in the database to store user information.
- Include fields such as user ID, email, hashed password, and role.
- Implement a registration endpoint that receives user details, generates a unique user ID,hashes the password, and saves the user to the database.

b. User Login:

- Implement a login endpoint that accepts user credentials and verifies them against thestored hashed password.
- Generate a JWT token upon successful authentication and return it to the client.
- Secure the APIs by checking the validity of the JWT token for each request.

2. **User Profile Management:**

a. User Profile:

- Create a "UserProfile" table in the database to store user profile details, including name,contact information, and address.
- Associate each user profile with the corresponding user ID.

b. Multiple Properties:

- Create a "Properties" table in the database to store property information.
- Include fields such as property ID, address, meter number, and tariff plan.
- Establish a one-to-many relationship between the user profile and properties table to associate multiple properties with each user.

3. **Electricity Usage Monitoring:**

a. Real-time Usage:

- Integrate with smart meters or provide a manual entry form for users to submit their electricity usage data.
- Store the usage data in a "UsageData" table in the database, associating each recordwith the user and property IDs.

b. Historical Usage Analysis:

- Implement a data analysis module that processes the usage data and generateshistorical usage statistics.
- Use charting libraries or data visualization tools to present the analysis in a graphicalformat.

c. Abnormal Usage Detection:

- Develop algorithms to detect abnormal usage patterns based on predefined thresholdsor machine learning models.
- Implement a notification system to alert users when abnormal usage patterns are detected.

4. Bill Generation and Payment:

a. Bill Generation:

- Create a "Bills" table in the database to store bill details.
- Generate bills based on the user's electricity usage data and applicable tariff rates.
- Store the generated bills in the database, associating them with the user and propertyIDs.

b. Bill Presentation and Download:

- Implement an API endpoint to retrieve the user's bills in PDF format.
- Enable users to download their bills from their accounts.

c. Payment Integration:

- Integrate with one or more payment gateway providers that support credit/debit cards,net banking, and mobile wallets.
- Implement secure payment processing by encrypting and tokenizing sensitive payment information.

5. Bill Reminders and Notifications:

a. Upcoming Bill Notifications:

- Implement a notification system that sends reminders to users before their bill duedates.
- Configure the notification channels (email, SMS, push notifications) based on user preferences.

b. Payment Reminders:

- Send automated reminders to users for pending payments, with configurable intervalsand escalation mechanisms.

c. Payment Confirmation:

- Send notifications to users for successful payments, providing payment receipts or confirmation numbers.

6. Customer Support:

a. Support Ticket System:

- Create a "SupportTickets" table in the database to store support ticket details.
- Include fields such as ticket ID, user ID, issue description, status, and assigned staff ID.
- Implement an API endpoint to create and manage support tickets.

b. Ticket Assignment and Resolution:

- Assign support tickets to relevant staff members based on category or priority.
- Implement an update mechanism to keep users informed of the status of their tickets

Non-Functional requirements

Security and Privacy:

a. User Data Protection:

● User data should be securely stored and transmitted using encryption techniques.

● Implement industry-standard security practices, such as secure coding guidelines for secure development, including input validation, parameterized queries, and protection against common vulnerabilities (e.g., cross-site scripting, SQL injection).

● Use secure protocols (e.g., HTTPS) for data transmission over networks: Encrypt data during transit to prevent eavesdropping and ensure data integrity.

● Store sensitive data, like passwords and payment details, in an encrypted format using strong encryption algorithms: Apply encryption techniques (e.g., AES) to protect sensitive data at rest.

JWT Authentication:

We will be configuring Spring Security and JWT for performing 2 operations-

● Generating JWT - Expose a POST API with mapping /authenticate. On passing correct username and password it will generate a JSON Web Token(JWT), upon successful authentication and return it to the client.

● Validating JWT - If user tries to access GET API with mapping /hello. It will allow access only if request has a valid JSON Web Token(JWT)

This will ensure the security of the APIs by checking the validity of the JWT token for each request.

b. Access Controls:

● Implement authentication mechanisms to ensure only authorized users can access the application: Use secure authentication methods (e.g., username/password, multi-factor authentication (Email-Verification)) to verify user identities.

● Employ role-based access controls (RBAC) to assign different permissions to different user roles: Classify users into roles (e.g., admin, consumer, Department Officials) and define access permissions

based on these roles. Different type of user will be having the access to the different access points to avoid any kind of access point or data collision.

● Implement session management and enforce secure session handling to prevent unauthorized access: Generate secure session tokens, manage session expiration, and protect against session hijacking or fixation attacks.

Performance and Scalability:

Horizontal Scaling:

Horizontal scaling involves adding more instances (servers) to distribute the workload and handle increased user demand. Here are some ways to horizontally scale the application:

a. Load Balancing:

Implement a load balancer that distributes incoming requests across multiple application servers. The load balancer can use different algorithms (e.g., round-robin, least connections) to evenly distribute the traffic.

b. Auto-scaling:

Use an auto-scaling mechanism that automatically adjusts the number of application server instances based on the current workload. This can be achieved by setting up policies or rules that define when to add or remove instances based on CPU utilization, request queue length, or other performance metrics.

c. Database Scaling:

Scale the database by using techniques such as sharding, which involves partitioning data across multiple database servers. This can be done based on customer IDs, geographic regions, or any other logical division of data.

Vertical Scaling:

Vertical scaling involves increasing the resources (CPU, memory, storage) of individual servers to handle higher loads. Here are some approaches to vertically scale the application:

a. Upgrade Hardware:

Upgrade the existing servers with higher-capacity hardware components such as CPU, RAM, or storage. This can be achieved by adding more powerful processors, increasing the memory capacity, or switching to faster storage drives (e.g., SSD).

b. Vertical Database Scaling:

If the database is a performance bottleneck, vertically scale it by upgrading the database server hardware or using more powerful database instances. Additionally, optimize database configurations and indexing to improve performance.

c. Caching:

Implement caching mechanisms to offload the application and database servers. Use in-memory caches (e.g., Redis, Memcached) to store frequently accessed data, reducing the load on the backend systems and improving response times.

d. Asynchronous Processing:

Offload resource-intensive or time-consuming tasks to background workers or separate services. By performing these tasks asynchronously, the main application can respond faster to user requests, improving overall performance.

It's important to note that a combination of horizontal and vertical scaling can be employed based on specific requirements and constraints. Regular monitoring and performance testing will help identify scaling needs and ensure the appropriate scaling strategy is implemented.

Logging and Monitoring:

a. Logging:

Implement a logging framework to capture system logs, user activities, errors, and system events: Log relevant information for auditing, troubleshooting, and compliance purposes.

● We can use the SLF4J (Simple Logging Facade for Java) logger in our application for logging purposes. SLF4J provides a simple and flexible logging API that acts as a facade or abstraction layer for various logging frameworks, such as Logback, Log4j, and Java Util Logging (JUL).

● SLF4J allows you to write log statements in your code using a common API and provides the flexibility to switch between different logging implementations without changing your code. It provides various logging levels (e.g., INFO, DEBUG, WARN, ERROR) to differentiate the severity of log messages.

User Experience:

Design a user-friendly interface that is intuitive and easy to navigate, by using suitable frontend development. Ensure responsive design to provide a consistent experience across different devices and screen sizes. Optimize the application's performance to provide smooth and seamless user interactions. Provide clear and informative error messages and notifications to assist users in resolving issues.

Types of Exceptions:

Authentication and Authorization Exceptions:

- InvalidCredentialsException: Raised when the user provides incorrect login credentials.
- AccountLockedException: Raised when the user's account is locked or suspended.
- UnauthorizedAccessException: Raised when a user tries to access a feature or resource without sufficient permissions.

User Profile Management Exceptions:

- InvalidInputException: Raised when the user provides invalid or incomplete information while updating their profile.
- DuplicatePropertyException: Raised when a user tries to add a property that already exists in their profile.
- ProfileNotFoundException: Raised when a requested user profile is not found in the system.

Electricity Usage Monitoring Exceptions:

- MeterConnectionException: Raised when there is a problem connecting to the smart meter for real-time usage monitoring.

- InvalidReadingException: Raised when the user provides an invalid reading or meter data during manual entry.
- DataAccessException: Raised when there are issues accessing or storing the usage data in the database.

Bill Generation and Payment Exceptions:

- BillingCalculationException: Raised when there are errors in calculating the electricity bill based on usage data and tariff rates.
- PaymentProcessingException: Raised when there are issues with processing the payment transaction through the payment gateway.
- BillNotFoundException: Raised when a requested bill is not found in the system.

Bill Reminders and Notifications Exceptions:

- NotificationSendingException: Raised when there are errors in sending notifications to users (e.g., email sending failure).
- ReminderSettingException: Raised when there are issues in setting up or scheduling bill reminders for users.
- AbnormalUsageException: Raised whenever uses the electricity above from threshold of their tariff plan i.e. high consumption.

Customer Support Exceptions:

- InvalidQueryException: Raised when a user submits an invalid or incomplete query to the customer support feature.
- SupportTicketNotFoundException: Raised when a requested support ticket is not found in the system.
- SupportStaffUnavailableException: Raised when there are no available staff members to handle a support ticket.

Test Cases:

User Authentication and Authorization:

Test Case 1: Verify that a user can register a new account successfully.

Test Case 2: Validate that a registered user can log in with valid credentials.

Test Case 3: Ensure that an unauthorized user is restricted from accessing protected features.

Test Case 4: Validate that user roles are correctly assigned, and access permissions are enforced.

User Profile Management:

Test Case 5: Verify that users can view and update their personal information.

Test Case 6: Validate that users can add multiple properties and associate them with their profile.

Test Case 7: Ensure that changes made to user profiles are saved accurately.

Electricity Usage Monitoring:

Test Case 8: Verify that real-time electricity usage data is accurately retrieved and displayed.

Test Case 9: Validate that historical usage data is stored correctly and can be graphically presented.

Test Case 10: Ensure that notifications for abnormal usage patterns or high consumption are triggered correctly.

Bill Generation and Payment:

Test Case 11: Verify that bills are generated accurately based on the user's electricity usage data and tariff rates.

Test Case 12: Validate that users can view and download their bills in PDF format.

Test Case 13: Test the integration with payment gateways to ensure successful and secure payment processing.

Test Case 14: Validate that multiple payment options (credit/debit cards, net banking, mobile wallets) are working correctly.

Bill Reminders and Notifications:

Test Case 15: Verify that users receive notifications for upcoming bill due dates.

Test Case 16: Validate that automated reminders are sent for pending payments.

Test Case 17: Ensure that notifications are sent for successful payments and bill confirmations.

Customer Support:

Test Case 18: Verify that users can raise queries or report issues through the customer support feature.

Test Case 19: Validate that support tickets are assigned to relevant staff for timely resolution.

Test Case 20: Ensure that users receive updates on the status of their queries.

API endpoints to consider:

User Management Endpoints:

POST /api/register: Register a new user account.

POST /api/login: Authenticate a user and generate an authentication token.

GET /api/ user-profiles: Retrieve the user's profile information.

PUT /api/ user-profiles: Update the user's profile information.

GET /api/users/properties: Retrieve the properties (houses, apartments) associated with the user's profile.

POST /api/users/properties: Add a new property to the user's profile.

Electricity Usage Endpoints:

GET /api/ /usage-data: Retrieve real-time electricity usage data.

POST /api/ /usage-data: Submit manual entry for electricity usage.

GET /api/ /usage-data/history: Retrieve historical usage data for a specific user.

Bill Management Endpoints:

GET /api/bills: Retrieve a list of bills for a user.

GET /api/bills/{billId}: Retrieve details of a specific bill.

GET /api/bills/{billId}/pdf: Download the PDF version of a bill.

POST /api/payments: Submit a payment for a bill.

Notification Endpoints:

POST /api/notifications/reminder: Set a reminder for upcoming bill due dates.

POST /api/notifications/query: Raise a support query or report an issue.

UML DIAGRAM

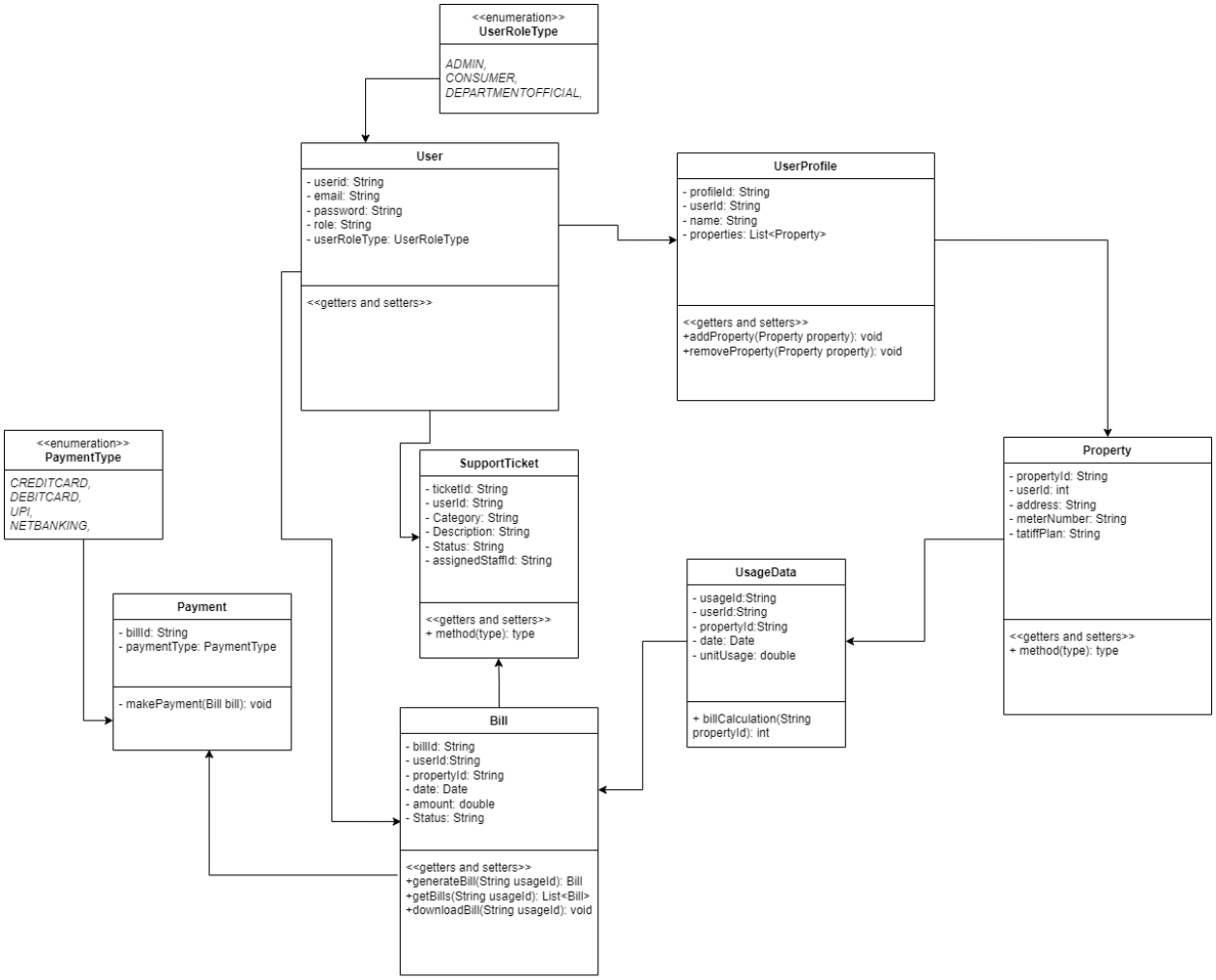

Figure 1 UML Diagram

User represents the user entity with attributes like userId, email, password, userRoleType and role.

UserProfile represents the user's profile entity with attributes like profileId, userId, name, contact, and address.

Property represents the property entity associated with each user, with attributes like propertyId, userId, address, meterNumber, and tariffPlan.

UsageData represents the electricity usage data entity with attributes like usageId, userId, propertyId, date, and usage.

Bill represents the bill entity with attributes like billId, userId, propertyId, date, amount, and status.

SupportTicket represents the support ticket entity with attributes like ticketId, userId, category, description, status, and assignedStaffId.

Code Link:

https://github.com/lavkumar3112/ElectricityBillSystem

HLD Implementation

High-Level Design (HLD) for an Electricity Payments Application:

System Overview:

The Electricity Payment Application System is designed to manage the details of Electricity, Bill, Connections, Store Record, Customer. It manages all the information about Electricity, Electricity Board, Customer, Electricity. It handles customer registration, meter reading, usage calculation, bill generation, payment processing, and customer support.

Architecture:

The system follows a modular and layered architecture to ensure separation of concerns and scalability. Common architectural patterns such as MVC (Model-View-Controller) or Microservices can be applied based on requirements.

User Interface Layer:

- User registration and login
- User profile management
- Billing information display
- Payment options and integration
- Notifications and reminders

Application Layer:

- Authentication and authorization
- User management
- Bill generation and calculation
- Usage data management
- Integration with external systems (smart meters, payment gateways)
- Support ticket management

Business Logic Layer:

- User authentication and authorization logic
- Billing logic (tariff rates, calculation methods)

- Usage data processing and analysis
- Abnormal usage detection algorithms
- Payment processing and integration logic
- Support ticket assignment and resolution logic

Data Layer:

- User and profile data storage
- Billing information storage
- Usage data storage
- Support ticket data storage

External Systems:

- Smart meters for real-time usage data
- Payment gateway providers for secure payment processing
- Notification channels (email, SMS, push notifications)

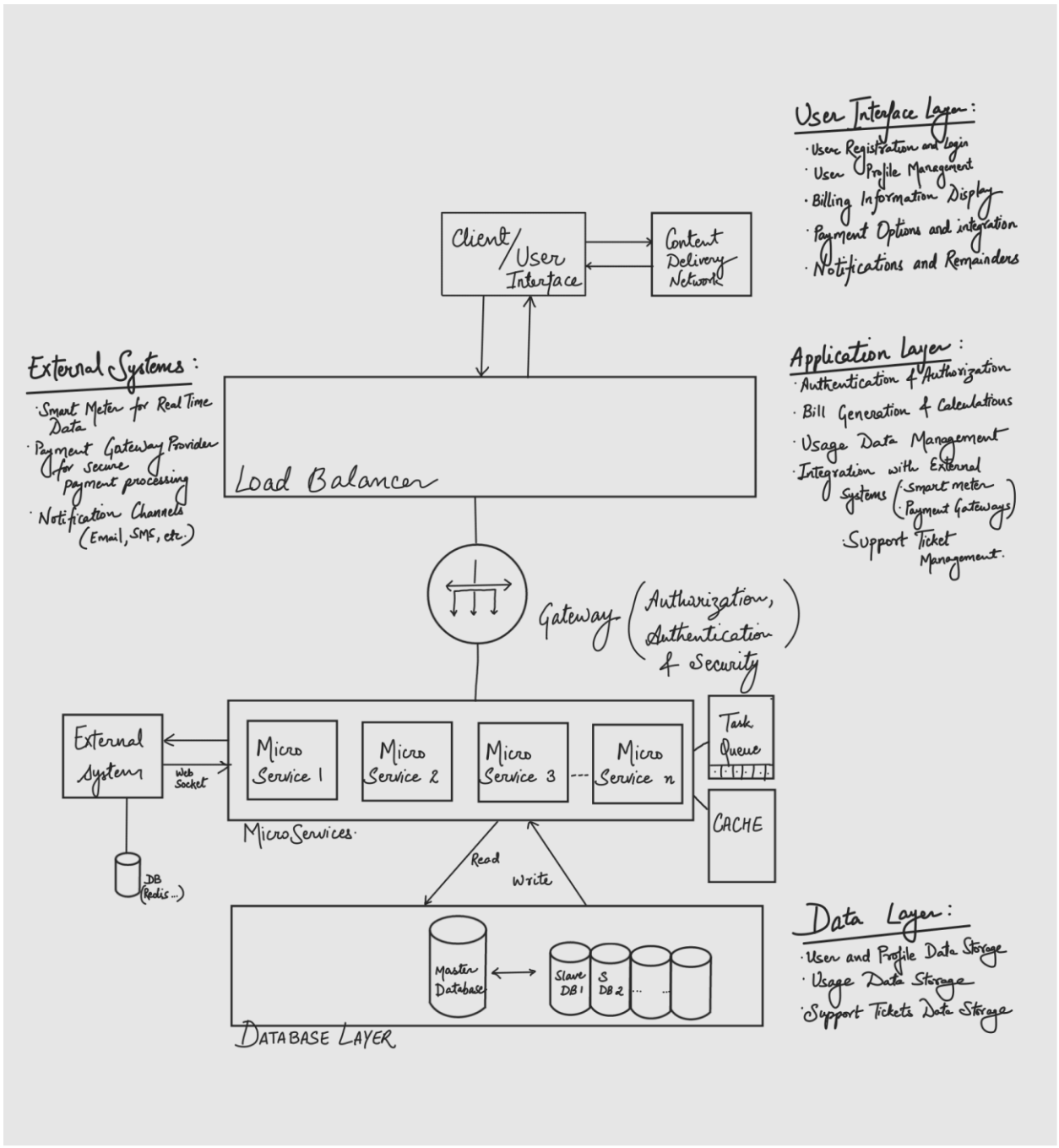

Figure 2 System Architecture

System Design:

1: Client/USER Interface:

Represents the end-user interface through which users interact with the application. It can be a web-based interface, mobile application, or any other user-facing interface.

2: Load Balancer:

Acts as a central entry point for incoming user requests. It distributes the incoming traffic across multiple CDN and web/application servers based on various algorithms (e.g., round-robin, least connections) to ensure load distribution, high availability, and scalability.

3: CDN (Content Delivery Network):

A geographically distributed network of servers that caches and delivers static content, such as images, CSS files, and JavaScript files, to users. It improves performance by reducing latency and minimizing the load on the application servers.

4: Web/Application Servers:

These servers host the application and handle user requests. They are responsible for processing business logic, retrieving data from databases or microservices, and generating dynamic content to be displayed to the users.

5: Gateway:

Acts as an API gateway and a central point for handling incoming requests. It provides functionalities such as request routing, authentication, authorization, rate limiting, request transformation, and security enforcement. The gateway routes requests to the appropriate microservices based on the request URL or other criteria.

6: Microservices:

Represent individual services responsible for specific functionalities in the system. Each microservice focuses on a specific business capability and operates independently. They communicate with each other through APIs, events, or message queues. Examples of microservices in an electricity payments application may include billing and payment services, consumption monitoring services, user management services, and notification services.

7: Messaging System:

Enables asynchronous communication between microservices. It facilitates the exchange of messages or events, allowing decoupling and scalability. Popular messaging systems such as Apache Kafka or RabbitMQ can be used to ensure reliable message delivery, event-driven architectures, and support for handling large volumes of messages. It helps in:

- Asynchronous Communication: Messaging systems enable decoupled communication between components, where senders and receivers do not need to be active simultaneously. This allows components to operate independently and asynchronously, improving system responsiveness and scalability.
- Loose Coupling: Messaging systems promote loose coupling between components by establishing a communication channel through messages or events. Components can send messages without knowing the details of the recipients or their implementation, allowing for easier maintenance, flexibility, and extensibility of the system.
- Scalability and Load Balancing: Messaging systems handle message distribution and load balancing across multiple consumers or services. They can distribute messages evenly across a group of consumers or dynamically scale the number of consumers based on the message load. This ensures efficient resource utilization and enables horizontal scalability.
- Fault Tolerance and Resilience: Messaging systems provide mechanisms to handle failures and ensure message delivery in the presence of failures or network disruptions. They can employ durable message storage, retries, and acknowledgments to guarantee reliable message processing. In the event of a component failure, messages can be persisted and redelivered when the component is back online, ensuring fault tolerance and system resilience.
- Event-Driven Architectures: Messaging systems support event-driven architectures, where components react to events by subscribing to specific message types or topics.

This enables real-time processing, event propagation, and event sourcing patterns. Event-driven architectures are beneficial in scenarios where different components need to react to changes or events in a decoupled and scalable manner.

8: Task Queues:

Manages and schedules background jobs, asynchronous tasks, and scheduled tasks. These tasks can include tasks like generating bills, processing payments, sending notifications, or performing batch operations. Tools like Celery or Apache Airflow can be used to manage and distribute tasks across the system efficiently. Here are some reasons why task queues are required:

- Asynchronous Task Processing: Task queues enable the asynchronous processing of tasks, allowing the task producer to continue its work without waiting for the task to be completed. This improves system responsiveness and overall performance.
- Scalability and Load Balancing: Task queues distribute tasks across multiple workers or consumers, enabling load balancing and scalability. They ensure that tasks are evenly distributed and processed by available workers, allowing the system to handle increased workloads efficiently.
- Fault Tolerance: Task queues provide fault tolerance by persisting tasks in case of failures. If a worker fails during task processing, the task remains in the queue and can be picked up by another available worker. This ensures that tasks are not lost and can be reliably processed even in the presence of failures.
- Prioritization and Scheduling: Task queues often support task prioritization and scheduling mechanisms. You can assign priorities to tasks based on their importance or urgency, ensuring that critical tasks are processed first. Additionally, you can schedule tasks to be executed at specific times or intervals, enabling advanced task management capabilities.
- Retries and Error Handling: Task queues typically handle retries and error handling for failed tasks. Failed tasks can be retried automatically based on predefined retry policies or sent to error queues for manual intervention and analysis. This ensures that tasks are processed reliably and errors are handled appropriately.

9: Caching Layer:

Utilizes in-memory caching systems such as Redis or Memcached to store frequently accessed data. Caching helps improve application performance by reducing the load on databases and reducing response times for data retrieval. Caching strategies such as cache invalidation, time-based expiration, or data versioning can be implemented to ensure data consistency.

In-Memory Caching: In-memory caching systems like Redis or Memcached store data in memory, allowing for fast data retrieval. They are suitable for caching frequently accessed data, such as user sessions, API responses, or result sets from expensive database queries. In-memory caching is beneficial when you need low-latency access to data and want to offload the load from the backend systems.

10: Distributed Database:

Refers to a scalable and distributed database system used to handle high volumes of data. Examples include Cassandra, MongoDB, or other NoSQL databases. Distributed databases provide horizontal scalability, fault tolerance, and high availability by replicating data across multiple nodes. They support efficient data storage, retrieval, and querying to meet the application's requirements.

SQL vs NoSQL Database

A distributed database is used in scenarios where there is a need for scalability, fault tolerance, and high availability of data across multiple nodes or locations. While SQL databases can be used in certain cases, distributed databases offer specific advantages in distributed systems:

1. Scalability: Distributed databases can handle large volumes of data and high traffic loads by distributing data across multiple nodes. This allows for horizontal scalability, meaning you can add more nodes to the system to accommodate increasing data or user load. SQL databases may have limitations in scaling horizontally due to their centralized architecture.

2. Fault Tolerance: Distributed databases provide built-in fault tolerance by replicating data across multiple nodes. If one node fails, the data is still accessible from other nodes, ensuring high availability and data reliability. SQL databases typically rely on backup and recovery mechanisms, which can introduce downtime during the recovery process.

3. High Availability: Distributed databases ensure that data is accessible even in the presence of failures or network disruptions. They can replicate data across geographically distributed locations, reducing the risk of data loss in the event of natural disasters or network outages. SQL databases often require additional configuration and infrastructure setup to achieve high availability.

4. Data Locality: Distributed databases can store data closer to the users or services that require it, reducing latency and improving performance. This is particularly beneficial in geographically distributed systems or systems with a global user base. SQL databases may not have built-in support for data distribution and may require complex data partitioning or sharding techniques to achieve similar benefits.

5. Flexibility: Distributed databases often support flexible data models, including document-based, key-value, or graph-based data structures. This allows developers to choose the most suitable data model for their application's requirements. SQL databases have a predefined table-based schema, which may not be ideal for all use cases.

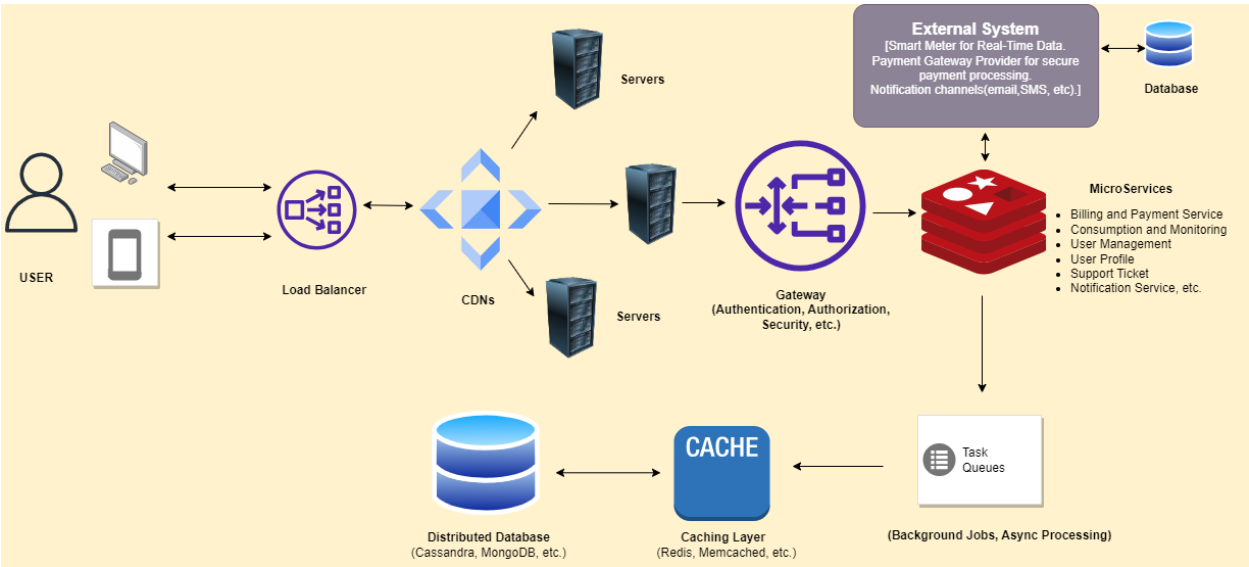

Figure 3 System Design

User Flow Diagram

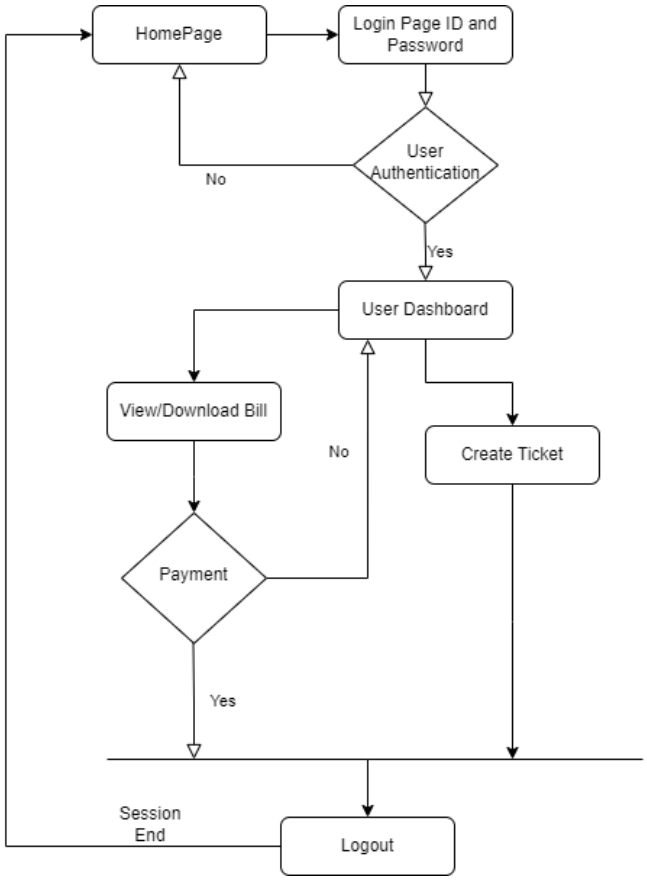

Figure 4 Flow Diagram