Solution Documentation

Introduction

Sri-Care is a modern customer care web portal and mobile app solution developed for **Sri Tel Ltd (STL)**, a leading telecommunications provider in Sri Lanka. The primary goal of Sri-Care is to enhance customer experience and streamline customer interactions with STL by offering a digital, self-service platform. This solution is designed to cater to STL's customers by providing them with an efficient and user-friendly way to manage their telecom services.

Through Sri-Care, customers can easily:

Create and Manage Accounts: Existing customers can create accounts securely without the need for manual intervention. The platform offers a simple process for account creation and password recovery.
View and Pay Bills Online: Customers can access their current and past bills, and make online payments using credit or debit cards. This eliminates the hassle of visiting physical locations for bill payments.
Activate and Deactivate Services: Customers can activate or deactivate a variety of value-added services (VAS) such as international roaming, data top-ups, and personalized ring tones.
Receive Notifications: The solution ensures that customers stay informed by receiving timely notifications via email, SMS, or push alerts regarding bills, service issues, and potential disconnections.

Sri-Care's digital-first approach aims to improve customer satisfaction by giving customers more control over their accounts and services, while reducing customer frustrations associated with manual processes and service interruptions.

Conceptual and Implementation Architecture

The conceptual architecture for **Sri-Care** outlines the key components and how they interact at a high level to provide a seamless customer experience. The following components are involved:

Sri-Care Web Portal and Mobile Apps (iOS and Android):

- Customers access the system through a **web interface** or **mobile apps** for iOS and Android. These apps provide user-friendly interfaces for account management, bill payments, and service activation.

Backend Services:

- These are core services responsible for handling authentication, billing, and service management. Examples include:
    - **Authentication Service**: Manages account creation, login, and password recovery using secure mechanisms (e.g., OAuth2).
    - **Billing Service**: Retrieves current and past bills and provides payment status.
    - **Service Management Service**: Manages activation and deactivation of telecom services like international roaming or data top-ups.

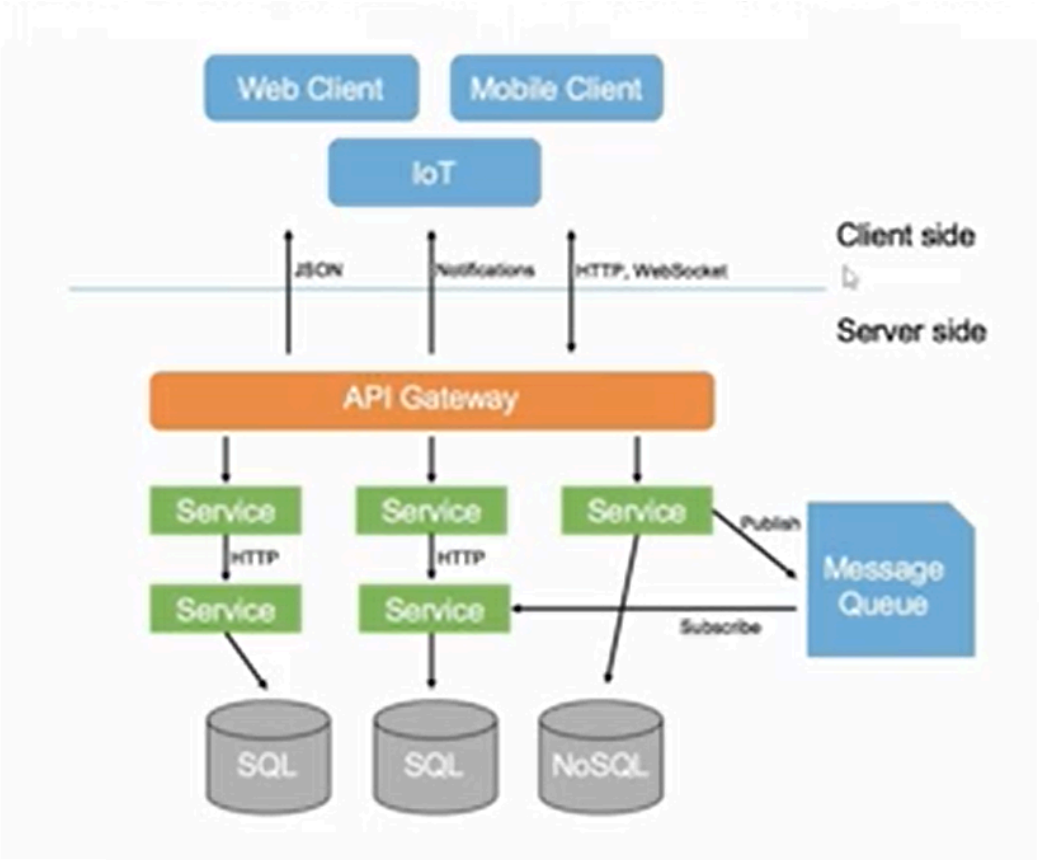
Third-Party Services:

- **Payment Gateway Integration**: The platform integrates with secure third-party payment gateways to process credit and debit card payments.
- **Notification Services**: Sends email, SMS, or push alerts for bills, service changes, or issues. This service ensures customers are informed in real-time, using services like **Twilio** for SMS, **Firebase Cloud Messaging** for push notifications, and email servers for email alerts.

Databases:

- A **centralized database** is used for storing customer profiles, bills, service records, and transaction history. This ensures that the system can retrieve and store information efficiently.

High-Level Conceptual Architecture Diagram



Implementation Architecture

Technologies to consider

- **Front-end**
  React.js : web portal,
  Flutter : mobile app

- **Backend (Business Logic)**
  Spring Boot: You can use Spring Boot for the REST API and microservices.
  Node.js: Alternatively, Node.js with Express.js can be used for REST services.

- **Middleware**
  Kafka: Used for message brokering, especially if high scalability is a priority.

- **Service Orchestration**
  Apache Camel : Use Camel to integrate different services

Alternative Architectures

- **Monolithic Architecture**: Consider a simpler monolithic structure as one alternative but explain why it lacks scalability and flexibility.
- **Serverless Architecture**: Another alternative is a serverless model using AWS Lambda or Google Cloud Functions, but it may be less flexible for complex orchestrations.

Reasons for choosing Microservices Architecture

Advantages:

- **Scalability**:
  In a microservices architecture, each service (such as user management, billing, notifications, etc.) is independent and can be scaled individually based on its load. For example, if the billing service is under heavy use (during bill payments), only that service can be scaled up without affecting the others.
- **Isolation of Services**:
  Each service is decoupled from the others, meaning that changes or updates in one service (e.g., the notification system) won't affect the rest of the system. This isolation enhances the reliability of the system because failures in one service (like the chat service) will not crash the entire application.
- **Technology Flexibility**:
  Microservices allow different technologies to be used for different services. For instance, you can use **Node.js** for high-concurrency tasks like chat and **Spring Boot** for managing complex workflows in account creation. This flexibility ensures that the best tool is chosen for each task.
- **Easier Maintenance & Deployment**:
  In a monolithic system, even a small change (like modifying the user registration process) would require redeploying the entire application, which increases downtime and risk. In a microservices setup, you can deploy individual services without impacting the whole system. This allows for **continuous delivery** and **faster updates**.
- **Resilience**:
  Because each service runs independently, if one service fails, it won't cause a total system failure. For example, if the notification service goes down, the billing system and other services will continue to function normally.

Disadvantages (and how to mitigate them):

- **Increased Operational Complexity**:
  Managing multiple services can be complex due to the need for orchestrating communication between them. However, this can be handled using tools like **Kubernetes** for container orchestration and **API gateways** for routing requests.

About Monolithic Architecture

Advantages:

- **Simplicity in Development & Deployment**:
  A monolithic architecture is simpler to build and deploy initially because the entire system is in a single codebase, and you only need to deploy one unit.

Why it's not chosen:

- **Lack of Scalability**:
  Scaling a monolithic application means scaling the entire system, even if only one part (e.g., the bill payments service) needs extra resources. This is inefficient and costly.
- **Tight Coupling**:
  In a monolithic structure, components are tightly interwoven. This means that if one part of the system (e.g., the service activation module) crashes or requires maintenance, the entire system could be affected.
- **Difficulty in Continuous Delivery**:
  A monolithic system slows down development because deploying changes requires thorough testing and redeployment of the entire system.

About Serverless Architecture

Advantages:

- **Event-Driven Execution**:
  Serverless works well for event-driven tasks where you pay only for the time the service is running. This can be useful for tasks like sending push notifications, which are triggered by specific events.
- **Automatic Scaling**:
  Serverless platforms automatically scale up and down based on the load, which eliminates the need for manual management of servers.

Why it's not chosen:

- **Increased Complexity**:
  While serverless architectures are useful for small, event-driven tasks, they become complex when orchestrating multiple services with interdependencies, as in STL's system. For example, managing the state of long-running processes like account registration or service activation across multiple serverless functions can be cumbersome and error-prone.
- **Cold Start Issues**:
  Serverless functions can experience delays during startup, known as **cold starts**, especially when infrequently used. This delay can cause latency in real-time services like activating or deactivating telecom services or responding to user interactions.
- **Vendor Lock-In**:
  Serverless architectures often rely on specific cloud providers (e.g., AWS Lambda, Google Cloud Functions), which can lead to vendor lock-in and limit flexibility in deployment strategies.

Conclusion

The **microservices architecture** strikes the best balance between scalability, resilience, and flexibility. Unlike the monolithic architecture, it allows for fine-grained scaling and independent service management. While serverless architectures have their strengths for certain tasks, they add unnecessary complexity and introduce potential performance issues for a system like **Sri-Care**, which needs to manage long-running processes and continuous service availability.

Thus, the microservices approach is chosen to ensure the system can grow, adapt, and maintain its performance under a high load, all while being manageable and cost-effective.

Architectural & Integration Patterns

- **Microservices Pattern**:
  Each service (e.g., user registration, billing, notifications) runs independently but communicates through well-defined RESTful interfaces.
- **Message Queue/Message Broker Pattern**:
  For handling the high volume of notifications (SMS/Email/Push Alerts), use a message broker like RabbitMQ or Kafka for efficient, asynchronous communication.
- **API Gateway Pattern**:
  Use an API Gateway to expose services securely to the front-end and manage authentication, rate-limiting, etc.
- **Service Orchestration Pattern**:
  The ESB will manage orchestration between services like the Provisioning System and Payment Gateway.

Rationale for Selecting These Patterns

- **Microservices** ensures scalability, ease of deployment, and service isolation.
- **Message Broker** handles the high volume of notifications asynchronously, reducing load on the system.
- **API Gateway** improves security by centralizing external requests and managing rate limits.

Security Considerations

This section is critical for ensuring the system is secure from external threats

Encryption:

- Use **TLS** (Transport Layer Security) to encrypt data in transit.
- Use **AES-256** encryption for sensitive data like passwords and billing information.

Authentication & Authorization:

- Implement **OAuth 2.0** or **JWT** (JSON Web Tokens) for authenticating users and managing sessions securely.
- Use **Role-Based Access Control (RBAC)** to ensure different users have appropriate permissions.

Input Validation:

- Ensure all inputs (account creation, service activation, etc.) are validated to avoid injection attacks (e.g., SQL injection).

API Security:

- Use an **API Gateway** to apply rate limits and prevent **DDoS** attacks.
- Add **CORS (Cross-Origin Resource Sharing)** policies to protect APIs from cross-site scripting attacks.

Data Storage:

- Store sensitive information like passwords using **bcrypt** hashing.

Monitoring & Logging:

- Implement **centralized logging** (using tools like ELK Stack) for all service calls to monitor for any suspicious activities.
- Use **Intrusion Detection Systems (IDS)** to monitor system activities in real-time.