**TELECOM BILLING AND PAYMENTS SYSTEM**

**ABSTRACT**

In modern software architecture, microservices provide a flexible and scalable approach for complex applications by dividing them into smaller, autonomous services. This documentation details a billing and payment management system composed of six microservices. The Invoice Management Microservice oversees invoice creation and retrieval, while the Payment Processing Microservice manages payment transactions. The Billing History Microservice tracks and maintains customer billing records. The Payment Notification Microservice is responsible for sending notifications about payment activities. The Refund Management Microservice processes and handles refund requests. Finally, the Customer Microservice manages customer data and account information. Each service operates independently but interacts to form a cohesive system, ensuring efficient management of financial operations and customer interactions.

**INTRODUCTION**

This documentation provides a comprehensive overview of a suite of six interconnected microservices designed to handle various aspects of a billing and payment management system.Each microservice is a modular component of the system, focusing on specific responsibilities to ensure a scalable and maintainable architecture.

1. **Customer Microservice**: Dedicated to handling customer data and accounts, this microservice provides endpoints for retrieving and creating customer information, also using Spring Boot, Spring Data JPA, and a PostgreSQL database.

2. **Billing History Microservice**: This service tracks and manages billing history, providing an endpoint to retrieve the billing history for a specific customer. It is implemented using Spring Boot, Spring Data JPA, and a PostgreSQL database.

3. **Payment Notification Microservice**: Responsible for sending notifications related to payments, this microservice uses Spring Boot, JavaMailSender to handle notifications and messaging.

4. **Payment Processing Microservice**: Focused on processing payments and managing transactions, this microservice offers endpoints for initiating payments and retrieving payment details. It also uses Spring Boot and Spring Data JPA with PostgreSQL.

5. **Invoice Management Microservice**: This service is responsible for generating and managing invoices. It provides endpoints to retrieve invoices by customer ID and to create new invoices, leveraging Spring Boot, Spring Data JPA, and a PostgreSQL database.

6. **Refund Management Microservice**: This service manages refund requests and processes, offering endpoints to request and retrieve refund details. It employs Spring Boot and Spring Data JPA with PostgreSQL.

The documentation includes detailed descriptions of endpoints, technologies used, and interactions between services, providing a clear guide for developers and stakeholders involved in the system.

**Microservice Architecture**

The system adopts a microservice architecture, where each microservice operates as an independent unit responsible for a particular domain:

**1. Customer Microservice**

**Functionality**: Manages customer data and accounts.

**Interaction**: Provides customer information to the Invoice Management and Payment Processing Microservices and receives notifications from the Payment Notification Microservice.

2. **Billing History Microservice**

**Functionality**: Tracks and provides access to billing history.

**Interaction**: Aggregates data from the Invoice Management and Payment Processing

Microservices to present a historical view of customer billing.

3.  **Payment Notification Microservice**

**Functionality**: Sends notifications related to payment activities.

**Interaction**: Receives triggers from the Payment Processing Microservice and possibly the Refund Management Microservice to notify customers of payment and refund statuses.

4. **Payment Processing Microservice**

**Functionality**: Manages payment transactions and records.

**Interaction:** Works closely with the Invoice Management Microservice to link payments with invoices and with the Refund Management Microservice for processing refunds.

5. **Invoice Management Microservice**

**Functionality:** Handles creation and retrieval of invoices.

**Interaction:** Interfaces with the Payment Processing and Customer Microservices to manage and associate invoices with payments and customer data.

**6. Refund Management Microservice**

**Functionality**: Handles refund requests and processes refunds.

**Interaction**: Interfaces with the Payment Processing Microservice to manage refunds and the Invoice Management Microservice to adjust invoice records as necessary.

**Technology Stack**

* **Spring Boot**: Provides the foundational framework for developing each microservice, ensuring consistency and reliability across the system.
* **Spring Data JPA**: Facilitates data access and management, enabling each microservice to interact with its database using an object-relational mapping approach.
* **PostgreSQL**: Relational databases used for storing transactional and persistent data, chosen based on the specific needs of each microservice.
* **JavaMailSender**: Utilized by the Payment Notification Microservice for sending email notifications.
* **Spring DevTools**: Enhances the development process with features like automatic restarts, live reload, and improved debugging.

**Service Interactions**

* **Customer Microservice**: Provides essential customer data to other microservices and receives updates and notifications related to customer activities.
* **Invoice Management and Billing History**: Billing History aggregates data from Invoices to provide a historical view.
* **Invoice Management and Payment Processing**: Payment Processing updates invoice records and retrieves invoice details to ensure payments are correctly applied.
* **Payment Processing and Refund Management**: Refund Management queries Payment Processing to process and track refunds.
* **Payment Processing and Payment Notification**: Payment Notification Microservice receives payment status updates to send notifications.

**Design Considerations**

* **Scalability**: Each microservice is designed to scale independently based on its specific load and usage patterns, ensuring that the overall system can handle varying amounts of traffic and data.
* **Modularity**: The separation of concerns allows for easier updates, maintenance, and deployment. Each microservice can be developed, tested, and deployed independently.
* **Fault Tolerance**: Microservices are designed to handle failures gracefully, with error handling mechanisms in place to ensure system stability even if one service encounters issues.
* **Security**: Each microservice should implement appropriate security measures, including authentication and authorization, to safeguard sensitive data and ensure secure interactions between services.
* **Data Consistency**: Techniques like eventual consistency and distributed transactions are used to manage data consistency across microservices, ensuring that updates are reflected accurately throughout the system.

**CUSTOMER MICROSERVICE**

**The Customer Microservice is designed to handle all operations related to customer data and accounts. It is a crucial component of the overall system, providing functionalities to manage customer information, including personal details and subscription plans.**

**Responsibilities**

**1. Manage Customer Data:**

- Maintain records of customer information, such as name, email, phone number, and subscription details.

- Provide mechanisms to retrieve and update customer information.

**2. Manage Subscription Plans:**

- Associate customers with specific plans.

- Track plan details like duration and price.

**Fields**

**The `Customer` entity has the following fields:**

**- ID: Unique identifier for each customer (Primary Key).**

**- Name: The name of the customer.**

**- Email: The email address of the customer.**

**- PhoneNo: The phone number of the customer.**

**- Plan: The plan associated with the customer.**

**- Duration: The duration of the current plan.**

**- Price: The price of the current plan.**

**Endpoints**

**The Customer Microservice exposes several RESTful endpoints to interact with customer data. Below are the key endpoints, their HTTP methods, and descriptions:**

**1. Create a New Customer**

**-Endpoint: `POST /api/customers`**

**- Description: Creates a new customer record.**

**- Request Body:**

**json**

**{**

**"Name": "saii",**

**"Email": "saiii@example.com",**

**"PhoneNo": "987-777-6666",**

**"Plan": "Prepaid",**

**"Duration": 12 ,**

**"Price": 99.99**

**}**

**```**

**- Response:**

**- Success: Returns the created customer record with `CustomerID`.**

**- Failure: Error message detailing the issue.**

**2. Retrieve Customer Information**

**- Endpoint: `GET /api/customers/{id}`**

**- Description: Retrieves the details of a customer by their unique `CustomerID`.**

**- Path Parameter: `id` - The unique identifier of the customer.**

**- Response:**

**json**

**{**

**"CustomerID": 1,**

**"Name": "bala",**

**"Email": "balaa@example.com",**

**"PhoneNo": "999-666-5555",**

**"Plan": "Prepaid",**

**"Duration": 12,**

**"Price": 99.99**

**}**

**```**

**- Failure: Error message if the customer is not found.**

**3. Update Customer Information**

**- Endpoint: `PUT /api/customers/{id}`**

**- Description: Updates the details of an existing customer.**

**- Path Parameter: `id` - The unique identifier of the customer.**

**- Request Body:**

**```json**

**{**

**"Name": "sarmeena",**

**"Email": "sarmeena@newdomain.com",**

**"PhoneNo": "965-654-2222",**

**"Plan": "postpaid",**

**"Duration": 6,**

**"Price": 49.99**

**}**

**```**

**- Response:**

**- Success: Returns the updated customer record.**

**- Failure: Error message if the update fails.**

**4. Delete a Customer**

**- Endpoint: `DELETE /api/customers/{id}`**

**- Description: Deletes a customer record by `CustomerID`.**

**- Path Parameter: `id` - The unique identifier of the customer.**

**- Response:**

**- Success: Confirmation message indicating the customer has been deleted.**

**- Failure: Error message if the customer could not be deleted.**

**5. List All Customers**

**- Endpoint: `GET /api/customers`**

**- Description: Retrieves a list of all customers.**

**- Response:**

**json**

**[**

**{**

**"CustomerID": 1,**

**"Name": "sam",**

**"Email": "sam@example.com",**

**"PhoneNo": "976-555-2222",**

**"Plan": "Postpaid",**

**"Duration": 12,**

**"Price": 99.99**

**},**

**{**

**"CustomerID": 2,**

**"Name": "ampt",**

**"Email": "ampt@example.com",**

**"PhoneNo": "+0987654321",**

**"Plan": "postpaid",**

**"Duration": "6 months",**

**"Price": 49.99**

**}**

**]**

**```**

**- Failure: Error message if the list cannot be retrieved.**

**Methods**

**The Customer Microservice provides several methods to interact with customer data:**

**1. Create Customer**

**- Method: `POST`**

**- Description: Adds a new customer to the system.**

**2. Read Customer**

**- Method: `GET`**

**- Description: Fetches details of a specific customer based on `CustomerID`.**

**3. Update Customer**

**- Method: `PUT`**

**- Description: Updates the information of an existing customer.**

**4. Delete Customer**

**- Method: `DELETE`**

**- Description: Removes a customer record from the system.**

**5. List Customers**

**- Method: `GET`**

**- Description: Retrieves a list of all customers.**

**Workflow**

* **Customer Creation:**
* **When a new customer is created via POST /api/customers, the service validates the input and stores the customer record in the database.**
* **Customer Retrieval:**
* **The GET /api/customers/{id} request queries the database for the customer with the specified CustomerID and returns their details.**
* **Customer Update:**
* **The PUT /api/customers/{id} endpoint processes requests to modify the existing customer record based on the provided data.**
* **Customer Deletion:**
* **The DELETE /api/customers/{id} endpoint removes the customer record from the database, ensuring no dependent records exist.**
* **List Customers:**
* **The GET /api/customers endpoint retrieves and returns a list of all customers for administrative purposes or reporting.**

**BILLING HISTORY MICROSERVICE**

**Key Components:**

**Class Definition:**

**The class is annotated with @RestController, making it a RESTful web service controller.**

**It is mapped to the base URL /api/billing using the @RequestMapping annotation.**

**Dependency Injection:**

**The controller depends on BillingRepository, which is injected via the constructor to manage CRUD operations for Billing entities.**

**Endpoint for Fetching Billing History:**

**Endpoint: @GetMapping("/history/{customerId}")**

**This endpoint retrieves a list of billing records associated with a particular customer ID.**

**It accepts customerId as a path variable from the URL.**

**Method: getBillingHistory(Long customerId):**

**Input: A customerId as a PathVariable in the request URL.**

**Output: Returns a List<Billing> containing all billing entries for the provided customerId.**

**Repository Call: Uses billingRepository.findByCustomerId(customerId) to fetch billing data from the PostgreSQL database.**

**Flow Explanation:**

**Client Request:**

**A client sends an HTTP GET request to /api/billing/history/{customerId}.**

**Controller Processing:**

**The getBillingHistory method is invoked with the customerId extracted from the URL.**

**The method calls the BillingRepository to query the database for billing entries matching the provided customerId.**

**Data Retrieval:**

**The BillingRepository interacts with the database (PostgreSQL in this case) to fetch the relevant billing records.**

**Response:**

**The retrieved list of Billing objects is returned as a JSON response to the client.**

**Supporting Classes:**

**Billing Entity:**

**Represents the Billing table in the database.**

**Contains fields: id, customerId, amount, and description.**

**BillingRepository Interface:**

**Extends JpaRepository<Billing, Long>, providing built-in CRUD operations.**

**Custom query method findByCustomerId(Long customerId) to fetch billing history by customer ID.**

**Application Configuration:**

**Database Configuration:**

**Configured to use PostgreSQL with connection details such as URL, username, password, etc.**

**spring.jpa.hibernate.ddl-auto=update ensures the schema is automatically updated.**

**spring.jpa.show-sql=true enables SQL logging for debugging purposes.**

**PAYMENT NOTIFICATION MICROSERVICE**

**Endpoints**

* **Send Notification**
* **Endpoint: POST /api/notification/notifications**
* **Fields:**
* **userId (string) - User identifier**
* **type (string) - Type of notification (e.g., email, SMS)**
* **message (string) - Notification message**
* **Get Notification Status**
* **Endpoint: GET /api/notification/notifications/{notificationId}**
* **Fields:**
* **notificationId (string) - Notification identifier**

**Methods**

* **Send Notification: Send a notification to a user**
* **Get Notification Status: Retrieve the status of a specific notification**

**Workflow**

* **Billing Creation**
* **A bill is created in the Billing microservice.**
* **The Bill ID is stored for future reference.**
* **Invoice Generation**
* **An invoice is generated based on the bill in the Invoice microservice.**
* **The Invoice ID is returned.**
* **Payment Process**
* **The user initiates a payment through the Payment microservice, linked to the bill.**
* **The Payment ID is generated.**
* **Refund Process**
* **If needed, a refund is requested through the Refund microservice, linked to the payment.**
* **The Refund ID is returned.**
* **Notification**
* **Notifications are sent to users for bill creation, invoice generation, payment confirmation, and refund status using the Notification microservice.**
* **Notification status can be checked to ensure delivery.**

**PAYMENT PROCESSING MICROSERVICE**

**Endpoints**

* **Initiate Payment**
* **Endpoint: POST /api/payment/payments**
* **Fields:**
* **billId (string) - The associated bill**
* **userId (string) - User identifier**
* **amount (decimal) - Amount being paid**
* **paymentMethod (string) - Payment method (e.g., credit card, bank transfer)**
* **Get Payment Details**
* **Endpoint: GET /api/payment/payments/{paymentId}**
* **Fields:**
* **paymentId (string) - Payment identifier**
* **Get User Payments**
* **Endpoint: GET /api/payment/users/{userId}/payments**
* **Fields:**
* **userId (string) - User identifier**

**Methods**

* **Initiate Payment: Process a new payment**
* **Get Payment Details: Retrieve details of a specific payment**
* **Get User Payments: Retrieve all payments made by a use**

**INVOICE MICROSERVICE**

**Endpoints**

* **Create Invoice**
* **Endpoint: POST /api/invoice/invoices**
* **Fields:**
* **billId (string) - The associated bill**
* **userId (string) - User identifier**
* **amount (decimal) - Invoice amount**
* **issueDate (date) - Invoice issue date**
* **Get Invoice Details**
* **Endpoint: GET /api/invoice/invoices/{invoiceId}**
* **Fields:**
* **invoiceId (string) - Invoice identifier**
* **Update Invoice**
* **Endpoint: PUT /api/invoice/invoices/{invoiceId}**
* **Fields:**
* **amount (decimal) - Updated amount**
* **issueDate (date) - Updated issue date**
* **Delete Invoice**
* **Endpoint: DELETE /api/invoice/invoices/{invoiceId}**
* **Fields:**
* **invoiceId (string) - Invoice identifier**

**Methods**

* **Create Invoice: Generate a new invoice**
* **Get Invoice Details: Retrieve details of a specific invoice**
* **Update Invoice: Modify an existing invoice**
* **Delete Invoice: Remove an invoice from the system**

**REFUND MICROSERVICE**

**Endpoints**

* Initiate Refund
* Endpoint: POST /api/refund/refunds
* Fields:
* paymentId (string) - The payment to be refunded
* amount (decimal) - Refund amount
* reason (string) - Reason for the refund
* Get Refund Details
* Endpoint: GET /api/refund/refunds/{refundId}
* Fields:
* refundId (string) - Refund identifier
* Get Payment Refunds
* Endpoint: GET /api/refund/payments/{paymentId}/refunds
* Fields:
* paymentId (string) - Payment identifier

**Methods**

* Initiate Refund: Process a new refund
* Get Refund Details: Retrieve details of a specific refund
* Get Payment Refunds: Retrieve all refunds associated with a specific payment

**Integration and Data Flow**

* **Customer Management and Invoicing:**
* **When a new customer is created or updated in the Customer Microservice, it can trigger actions in the Invoice Management Microservice to generate or update invoices accordingly.**
* **Example: A new customer is created with a subscription plan. This plan may automatically generate an invoice through the Invoice Management Microservice.**
* **Invoice and Payment Processing:**
* **The Invoice Management Microservice generates invoices, which are then used by the Payment Processing Microservice to handle payments.**
* **Example: When an invoice is generated, the system triggers a payment request to the Payment Processing Microservice. Payment details are recorded and linked to the invoice.**
* **Payment Notifications:**
* **After a payment is processed by the Payment Processing Microservice, it triggers the Payment Notification Microservice to send notifications to the customer.**
* **Example: Upon successful payment, an email notification is sent to the customer detailing the payment information.**
* **Billing History Tracking:**
* **Payment transactions and invoice details are recorded in the Billing History Microservice to maintain a comprehensive record of customer billing activities.**
* **Example: Each time a payment or invoice is processed, the details are updated in the billing history.**
* **Refund Management:**
* **Refund requests are managed by the Refund Management Microservice, which interacts with the Invoice Management Microservice to verify invoice details before processing a refund.**
* **Example: If a customer requests a refund, the system verifies the invoice through the Invoice Management Microservice and processes the refund accordingly.**

**System Workflow Summary**

* **Customer Creation:**
* **New customer is added.**
* **If a plan is assigned, an invoice is created.**
* **Invoice Management:**
* **Invoice is generated based on customer details and plan.**
* **Invoice details are used by the Payment Processing Microservice to process payments.**
* **Payment Processing:**
* **Payment is processed.**
* **Upon successful payment, a notification is sent.**
* **Payment and invoice details are updated in the Billing History Microservice.**
* **Billing History:**
* **Billing history is maintained and updated with each transaction.**
* **Refund Management:**
* **Refund requests are processed, and refunds are managed according to the invoice details.**
* **Notifications:**
* **Customers are notified about payment activities and refunds.**

**By following this workflow, each microservice interacts effectively, ensuring seamless operations within the billing and payment management system.**

**Conclusion**

**The integration of the six microservices—Customer Microservice, Invoice Management Microservice, Payment Processing Microservice, Billing History Microservice, Payment Notification Microservice, and Refund Management Microservice—creates a robust and scalable billing and payment management system. This architecture leverages the strengths of microservices to ensure a modular, efficient, and maintainable system.the integrated microservice architecture provides a well-structured, scalable, and efficient solution for managing complex billing and payment operations. It enables robust functionality, rapid development, and seamless user experiences, positioning the system for success in dynamic business environments.**