**Architecture Diagram**

Account Service

Transaction Service

User

Customer Service

Notification Service

Database

Kafka

Discovery Service

1. **High-Level Architecture Diagram**

From above diagram, we have 5 servicees, customer service, account service, transction service, notification service and service discovery service.

* **Customer Service:** Handles customer registration, authentication, profile updates.
* **Account Service:** Manages bank accounts (creation, balance inquiries, closure).
* **Transaction Service:** Manages financial transactions, such as deposits, withdrawals, and transfers.
* **Notification Service:** Listens to Kafka events and sends notifications about transactions and account changes to customers
* **Discovey Service:** This service register all services, to allow dynamic resolution of service endpoints.

**Message Broker (Kafka)** for inter-service communication between services, especially for publishing transaction events to the Notification Service.

**Database per Service:** Each service should have its own database to support the microservices architecture and prevent tight coupling. But for this project, I have used a single database.

**Security**

* **Authentication and Authorization**: Implemented OAuth 2.0 authentication where tokens are generated upon customer login and used across services. The customer service will handle token validation to ensure all requests are authenticated. Though this should be one through auth server or keycloak.
* **Encryption**. I have used hashing algorithm to encrypt sensitive customer data such as password. I have also masked account number while sending notifcation.

1. **Data Consistency.**

* For data consistency, I have used annotations such as @Transactional to have data consistency.
* I have used Locking mechasim to make sure update are made without causing the race conditions when the record is being updated by multiple threads.

1. **Scalability, High Availability.**

**Scalability**

* Stateless Services: Design services to be stateless, where feasible, to allow easy horizontal scaling.
* Database Partitioning and Replication: Use database partitioning and read replicas to scale data access independently from compute scaling.

**High Availability**

* Load Balancing: Use load balancers to distribute requests evenly across instances of each service.
* Redundancy: Deploy services across multiple availability zones to minimize single points of failure.

**Disaster Recovery**

* Automated Backups: Schedule regular backups of databases with mechanisms for point-in-time recovery.
* Active-Active Setup: For critical services, consider an active-active setup across geographically separated zones or regions.
* Failover Mechanisms: Implement failover mechanisms for databases and message brokers, like Kafka, to minimize downtime.