. Design a Real-Time Payment Processing System

💡 *Scenario:* Build a high-throughput, low-latency system to process transactions in real time.

Key Discussion Points:

✅ **System Components:**

- API Gateway (Spring Cloud Gateway)
- Transaction Processing Service (Spring Boot, gRPC for low latency)
- Event-Driven Architecture (Kafka for async processing)
- ACID-compliant Database (PostgreSQL/MySQL with read replicas)
- Fraud Detection Module (Kafka Streams + ML models)
- Notifications (WebSockets, Webhooks, SMS/Email)

✅ **Concurrency & Consistency**

- Use **Optimistic Locking** or **Pessimistic Locking** for database updates
- Implement **idempotency keys** for retrying failed transactions

✅ **Scalability & Fault Tolerance**

- Use **Kafka with partitions** for event-driven transaction processing
- Implement **circuit breakers** (Resilience4J) to handle failures
- Deploy with **Kubernetes (K8s) + Auto-scaling**

✅ **Security & Compliance**

- **OAuth2 + JWT** for authentication
- **TLS & AES Encryption** for securing transactions
- **PCI-DSS Compliance** for storing sensitive data

Designing a **Real-Time Payment Processing System** involves multiple components to ensure high throughput, low latency, security, scalability, and fault tolerance. In this architecture, we'll leverage modern microservice patterns and tools such as **Spring Cloud Gateway**, **gRPC**, **Kafka**, **PostgreSQL**, and **Resilience4J** to build a highly efficient system.

System Overview

Here's the high-level architecture for the **Real-Time Payment Processing System**:

1. **API Gateway (Spring Cloud Gateway)**: Exposes a unified entry point for incoming requests.
2. **Transaction Processing Service (Spring Boot, gRPC)**: Handles real-time payment transactions with low latency.

3. **Event-Driven Architecture (Kafka)**: Ensures asynchronous and fault-tolerant processing of events, such as transactions, fraud detection, and notifications.
4. **Database (PostgreSQL/MySQL)**: ACID-compliant database with read replicas for high availability and performance.
5. **Fraud Detection Module (Kafka Streams + ML models)**: Analyzes transactions for fraudulent behavior.
6. **Notifications (WebSockets, Webhooks, SMS/Email)**: Notify customers of transaction statuses.
7. **Security and Compliance**: Ensure all data is securely processed using **OAuth2, JWT**, **TLS**, **AES Encryption**, and **PCI-DSS compliance**.

1. System Components Breakdown

1.1. API Gateway (Spring Cloud Gateway)

The **API Gateway** acts as the entry point to the system, routing requests to the appropriate services. It ensures security (OAuth2 + JWT), handles rate limiting, and provides monitoring metrics.

- **OAuth2 + JWT** for authentication and authorization.
- **Rate Limiting** and **Circuit Breakers** (Resilience4J) for resilience.

java
CopyEdit
```java
@EnableOAuth2Sso
@Configuration
public class GatewayConfig {

    @Bean
    public RouteLocator gatewayRoutes(RouteLocatorBuilder builder) {
        return builder.routes()
                .route(r -> r.path("/api/payment/**")
                        .uri("lb://TRANSACTION-SERVICE"))
                .build();
    }
}
```

Security Configuration:

yaml
CopyEdit
```yaml
spring:
  cloud:
    gateway:
      routes:
        - id: payment
          uri: lb://payment-service
          predicates:
            - Path=/api/payment/**
          filters:
            - name: AddRequestHeader
              args:
                Authorization: "Bearer {JWT_TOKEN}"
```

1.2. Transaction Processing Service (Spring Boot & gRPC)

To process transactions in **real-time** with **low latency**, we use **gRPC** for communication between services.

- **gRPC**: High-performance RPC for low-latency communication.
- **Spring Boot**: Manages transaction logic.

gRPC Transaction Service:

- **Transaction Service** to process payments.
- **Payment Request** is handled by gRPC service.

java
CopyEdit
```
@Service
public class PaymentServiceGrpcImpl extends
PaymentServiceGrpc.PaymentServiceImplBase {

    @Override
    public void processTransaction(PaymentRequest request,
StreamObserver<PaymentResponse> responseObserver) {
        // Handle the payment processing logic here
        boolean success = processPayment(request);
        PaymentResponse response = PaymentResponse.newBuilder()
                .setTransactionId(request.getTransactionId())
                .setSuccess(success)
                .build();
        responseObserver.onNext(response);
        responseObserver.onCompleted();
    }

    private boolean processPayment(PaymentRequest request) {
        // Simulate payment processing
        return true;
    }
}
```

gRPC Server Configuration:

java
CopyEdit
```
@Configuration
public class GrpcConfig {
    @Bean
    public GrpcServerFactory grpcServerFactory() {
        return new GrpcServerFactory();
    }
}
```

1.3. Event-Driven Architecture (Kafka)

Kafka handles all event-driven processing. This includes:

- **Transaction Events**: Every transaction is published to a Kafka topic.
- **Fraud Detection Events**: Kafka Streams listens for transaction events to detect potential fraud.
- **Notification Events**: After transaction completion, the system sends a notification.

Kafka topics:

- **payment-events**
- **fraud-detection**
- **notification-events**

java
CopyEdit
```
@Service
public class PaymentEventPublisher {

    @Autowired
    private KafkaTemplate<String, PaymentEvent> kafkaTemplate;

    public void publishPaymentEvent(PaymentEvent paymentEvent) {
        kafkaTemplate.send("payment-events", paymentEvent);
    }
}
```

Kafka Streams for **Fraud Detection**:

java
CopyEdit
```
@EnableKafkaStreams
@Configuration
public class FraudDetectionProcessor {

    @Bean
    public KStream<String, PaymentEvent>
fraudDetectionStream(StreamsBuilder builder) {
        KStream<String, PaymentEvent> paymentStream =
builder.stream("payment-events");

        paymentStream.filter((key, value) -> !isFraudulent(value))
                     .to("notification-events");

        return paymentStream;
    }

    private boolean isFraudulent(PaymentEvent paymentEvent) {
        // Simple fraud detection logic (e.g., large amounts,
high-frequency transactions)
        return paymentEvent.getAmount() > 10000;
    }
}
```

1.4. ACID-compliant Database (PostgreSQL/MySQL)

For **high availability and consistency**, we use **PostgreSQL** with **read replicas** to manage transactional data.

- **Primary Database**: Stores transactional data with **ACID** compliance.
- **Read Replicas**: For high scalability and availability.

Optimistic Locking for concurrency control:

java
CopyEdit
```
@Entity
public class PaymentTransaction {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private BigDecimal amount;

    @Version
    private Long version;  // Optimistic locking
}
```

Idempotency:

Each payment request is assigned a unique **idempotency key** to ensure that retrying failed transactions doesn't cause duplicates.

1.5. Notifications (WebSockets, Webhooks, SMS/Email)

Once a transaction is processed, the system sends a notification using different methods:

- **WebSockets** for real-time updates.
- **Webhooks** for external systems.
- **SMS/Email** for customer alerts.

java
CopyEdit
```
@Service
public class NotificationService {

    @Autowired
    private WebSocketService webSocketService;

    public void sendNotification(PaymentResponse response) {
        // Send to WebSocket
        webSocketService.sendMessage(response);

        // Send to Email/SMS
        // Send to external SMS/Email services
    }
}
```

---

2. Concurrency & Consistency

- **Optimistic Locking**: To handle concurrency issues when multiple services try to access the same database record concurrently, we can use **Optimistic Locking** with `@Version` in Spring Data JPA.
- **Idempotency**: Every payment request contains an **idempotency key**. If a transaction fails and is retried, it is ensured that the same transaction is not processed multiple times.

---

3. Scalability & Fault Tolerance

Kafka Partitions:

Kafka partitions ensure that the system can scale horizontally. Each Kafka partition can be processed by a separate instance of a consumer service, ensuring scalability.

Resilience4J Circuit Breaker:

To ensure that the system can gracefully handle failures in external systems like payment gateways, we use **Resilience4J Circuit Breakers**.

java
CopyEdit
```java
@Bean
public CircuitBreakerFactory circuitBreakerFactory() {
    return new Resilience4JCircuitBreakerFactory();
}

public void processPayment() {
    circuitBreakerFactory.create("paymentService").run(() -> {
        // Make an external payment request
        return externalPaymentService.process();
    });
}
```

Kubernetes:

Deploy the system using **Kubernetes (K8s)**, which provides:

- **Auto-scaling** of pods based on the load.
- **Self-healing** by restarting failed pods.

---

4. Security & Compliance

OAuth2 + JWT:

Authentication and authorization are handled using **OAuth2** and **JWT tokens**. This ensures secure access control to microservices.

java
CopyEdit
```
@EnableOAuth2Sso
public class SecurityConfig {

    @Bean
    public JwtDecoder jwtDecoder() {
        return
JwtDecoders.fromIssuerLocation("https://auth-server.com");
    }
}
```

TLS Encryption:

To protect transaction data in transit, we use **TLS encryption** for all communications between services.

PCI-DSS Compliance:

Sensitive payment data is **never stored in plaintext**. Use **AES encryption** for encrypting sensitive information like credit card numbers before storing them.

Conclusion

The **Real-Time Payment Processing System** provides:

- **High throughput and low latency** using **gRPC** and **Kafka**.
- **Resilience** through **Circuit Breakers** (Resilience4J) and **Kafka Streams** for fraud detection.
- **Scalability** with **Kafka partitions**, **Kubernetes** auto-scaling, and **read replicas** in databases.
- **Security** with **OAuth2**, **JWT**, and **TLS encryption**.
- **ACID-compliant database** ensures consistency and reliability in transaction processing.

2. Design a High-Frequency Trading (HFT) Platform

💡 *Scenario:* Design a system that can process millions of stock trades per second.

Key Discussion Points:

✅ **Low Latency Considerations:**

- Use **Netty or gRPC** for ultra-fast communication
- Leverage **Zero-Copy Mechanisms (e.g., Java NIO, memory-mapped files)**
- Use **LMAX Disruptor** for in-memory event processing

✅ **Market Data Processing**

- Subscribe to **Kafka streams** for real-time stock market updates
- Maintain **Order Books** in **Redis or Aerospike** for fast access

✅ **Risk Management & Compliance**

- Implement **circuit breakers** to throttle high-volume trades
- Use **pre-trade risk checks** to avoid violations

Designing a **High-Frequency Trading (HFT) Platform** requires handling high-throughput, low-latency processing with stringent performance and compliance requirements. The goal is to process millions of stock trades per second, which requires a well-architected system that ensures ultra-low latency, real-time market data processing, risk management, and compliance.

Here's a detailed design for the system based on the key discussion points provided.

---

1. System Overview and Components

The **HFT Platform** involves several components:

1. **Market Data Stream**: Real-time stock prices and market updates are consumed using Kafka streams or other fast messaging protocols.
2. **Order Matching Engine**: Matching buy and sell orders in real-time to facilitate trades.
3. **Risk Management & Compliance**: Check trades for compliance with risk rules before execution.
4. **Trade Execution**: Final execution of matched trades.
5. **Low-Latency Communication**: Ultra-low latency communication between the components using **Netty** or **gRPC**.
6. **Storage Layer**: Fast storage of market data and order books using **Redis** or **Aerospike**.

---

2. Low Latency Considerations

For an **HFT platform**, latency is paramount. Here are some strategies to ensure ultra-low latency:

2.1. Netty / gRPC for Ultra-Fast Communication

Netty is a high-performance network framework for low-latency communication, commonly used in trading systems. It enables efficient handling of I/O operations.

- **Netty**: Handles network communication with high throughput, low latency, and non-blocking I/O.

java

CopyEdit

```
public class TradingServer {



    public static void main(String[] args) throws
InterruptedException {

        EventLoopGroup bossGroup = new NioEventLoopGroup(1);

        EventLoopGroup workerGroup = new NioEventLoopGroup();

        try {

            ServerBootstrap b = new ServerBootstrap();

            b.group(bossGroup, workerGroup)

             .channel(NioServerSocketChannel.class)

             .option(ChannelOption.SO_BACKLOG, 128)

             .childOption(ChannelOption.SO_KEEPALIVE, true)

             .childHandler(new ChannelInitializer<SocketChannel>() {

                 @Override

                 public void initChannel(SocketChannel ch) throws
Exception {

                     ch.pipeline().addLast(new TradingHandler());
```

```
                    }
                });

                ChannelFuture f = b.bind(8080).sync();
                f.channel().closeFuture().sync();
            } finally {
                workerGroup.shutdownGracefully();
                bossGroup.shutdownGracefully();
            }
        }
    }
```

Alternatively, **gRPC** (using Protocol Buffers) can be used for communication between services, particularly for its efficiency in serializing and deserializing messages.

protobuf

CopyEdit

```
syntax = "proto3";

service TradingService {
    rpc placeOrder(Order) returns (TradeResponse);
}

message Order {
    string symbol = 1;
    string side = 2;
    double price = 3;

    int32 quantity = 4;

}


message TradeResponse {

    string status = 1;

    string tradeId = 2;

}
```

2.2. Zero-Copy Mechanisms (Java NIO & Memory-Mapped Files)

Zero-copy allows for data to be transferred directly from memory without additional copies. **Java NIO** and **Memory-Mapped Files** can be used to implement efficient data handling.

java

CopyEdit

```
RandomAccessFile file = new RandomAccessFile("tradeData.dat", "rw");

FileChannel fileChannel = file.getChannel();

MappedByteBuffer buffer =
fileChannel.map(FileChannel.MapMode.READ_WRITE, 0,
fileChannel.size());
```

This approach ensures low latency and faster data access by directly working with memory, reducing unnecessary copying between the application and the OS kernel.

2.3. LMAX Disruptor for In-Memory Event Processing

The **LMAX Disruptor** is an extremely low-latency, high-throughput messaging system that processes events in memory. It provides very high performance in scenarios like trading platforms where events are processed concurrently.

java

CopyEdit

```
Disruptor<TradeEvent> disruptor = new Disruptor<>(TradeEvent::new,
1024, executor);

disruptor.handleEventsWith(new TradeEventHandler());

disruptor.start();
```

The disruptor uses a **ring buffer** to store events in memory and ensures that each trade is processed with minimal latency.

---

3. Market Data Processing

For real-time stock updates, market data must be processed as it arrives. This can be achieved by subscribing to **Kafka Streams** or using other real-time streaming technologies.

3.1. Kafka for Real-Time Stock Market Updates

Kafka acts as the **data pipeline** for streaming stock data, such as price updates, order book data, and trade events. We use Kafka consumers to ingest market data in real-time.

java

CopyEdit

```
@EnableKafka

@Configuration

public class KafkaConfig {



    @Bean

    public
KafkaListenerContainerFactory<ConcurrentMessageListenerContainer<Str
ing, StockPrice>> kafkaListenerContainerFactory() {

        return new ConcurrentMessageListenerContainerFactory<>();

    }
    @KafkaListener(topics = "stock-market-updates", groupId =
"market-data")

    public void listenMarketData(StockPrice stockPrice) {

        // Process the stock price data

        updateMarketData(stockPrice);

    }


    private void updateMarketData(StockPrice stockPrice) {

        // Process the incoming stock price and update internal
market data structures

    }

}
```

3.2. Order Book Management in Redis or Aerospike

The **Order Book** (buy and sell orders) must be maintained in **Redis** or **Aerospike** for ultra-fast access and querying. **Redis** is widely used for this purpose due to its low latency and support for in-memory data structures.

java

CopyEdit

```
@Autowired

private StringRedisTemplate redisTemplate;


public void updateOrderBook(String symbol, Order order) {

    redisTemplate.opsForZSet().add(symbol, order.getId(),
order.getPrice());

}
```

```
public Set<Order> getOrderBook(String symbol) {

    Set<String> orderIds = redisTemplate.opsForZSet().range(symbol,
0, -1);

    // Retrieve order details from the database

    return fetchOrdersFromDb(orderIds);

}
```

This allows fast access to the current state of the market.

---

4. Risk Management & Compliance

4.1. Circuit Breakers to Throttle High-Volume Trades

In HFT, it's crucial to have circuit breakers in place to protect against market volatility. **Resilience4J** provides a circuit breaker mechanism that can be integrated into trading services.

java

CopyEdit

```
@Bean

public CircuitBreakerFactory circuitBreakerFactory() {

    return new Resilience4JCircuitBreakerFactory();

}


public void placeTradeOrder(TradeOrder order) {

    circuitBreakerFactory.create("tradeCircuitBreaker").run(() -> {

        // Execute trade order

        executeTrade(order);

    });
```

```java
}
```

This ensures that high-frequency trades are monitored, and if thresholds are breached, the system can slow down or stop processing to prevent losses.

4.2. Pre-Trade Risk Checks

Before executing any trade, the system performs **pre-trade risk checks** (such as verifying margin, verifying trade limits, and ensuring compliance with regulations).

java

CopyEdit

```java
public boolean preTradeRiskCheck(TradeOrder order) {

    boolean marginOk = checkMargin(order);

    boolean limitOk = checkTradeLimit(order);

    return marginOk && limitOk;

}
```

If a trade violates any risk or compliance rules, it is rejected before execution.

---

5. Trade Execution

Once a trade is successfully matched and passes the risk checks, it is executed. This involves executing the trade through a low-latency trade execution engine.

5.1. High-Speed Execution Engine

The execution engine interacts with external exchanges or liquidity providers to execute the trade. Here's a simple flow:

- **Order Matching**: Matches the buy and sell orders from the order book.
- **Trade Execution**: Sends orders to an external exchange for final settlement.

---

6. Scalability and Fault Tolerance

For HFT platforms, **scalability** and **fault tolerance** are critical. We achieve this by:

- **Kafka**: Kafka partitions can be scaled horizontally for higher throughput.
- **Sharding**: Distribute data across multiple databases or Redis nodes.
- **Kubernetes**: Use Kubernetes for auto-scaling, self-healing, and managing containerized applications.

---

7. Security & Compliance

Compliance with industry standards like **MiFID II** and **FINRA** is essential in an HFT platform:

- **OAuth2 + JWT**: Secure communication with external systems, providing access control and authentication.
- **TLS Encryption**: All communication should be encrypted with TLS to ensure confidentiality.
- **Data Integrity**: Use **checksum** or **HMAC** to verify that data is not tampered with during transmission.

Conclusion

Designing a **High-Frequency Trading (HFT) Platform** requires careful consideration of **low-latency**, **high-throughput**, and **fault-tolerant** design principles. The architecture uses technologies like **Netty**, **gRPC**, **Kafka**, **Redis**, **Aerospike**, and **LMAX Disruptor** for high-speed data processing. **Circuit Breakers** and **Pre-Trade Risk Checks** ensure that trades comply with the necessary risk management rules.

By combining these tools with efficient communication mechanisms and robust data storage systems, you can create a platform capable of processing millions of trades per second while ensuring the highest standards of compliance and reliability.

3. Design a Fraud Detection System for Banking Transactions

💡 *Scenario:* Design a system to detect fraudulent transactions in real time.

Key Discussion Points:

✅ **Real-Time Fraud Detection Pipeline:**

- **API Gateway → Kafka → Stream Processing (Flink/Spark) → ML Model → Alerts**

✅ **ML-Based Anomaly Detection:**

- Train fraud detection models (XGBoost, Random Forest)

- Deploy **model inference using TensorFlow Serving**

✅ **Data Storage & Retrieval:**

- Store historical transactions in **Cassandra or BigQuery**
- Maintain real-time session data in **Redis**

✅ **High Availability & Scalability:**

- Use **Kafka partitions** for scaling fraud detection
- Implement **fallback mechanisms** for delayed ML model responses

In today's digital banking ecosystem, detecting fraudulent transactions in real-time is crucial to ensuring the safety and security of customer accounts. The key challenge is to design a scalable, high-performance fraud detection system that can analyze transaction patterns, identify anomalies, and trigger alerts when necessary. This system will be based on stream processing, machine learning models, and high-availability architecture to ensure minimal latency and maximum reliability.

Key Components of the System:

1. **API Gateway** (Spring Cloud Gateway or Nginx)
   - Acts as a reverse proxy to route incoming banking transactions to the fraud detection system.
2. **Kafka** (for Real-Time Event Streaming)
   - Acts as the message broker to stream transaction data from the API Gateway to the fraud detection pipeline.
3. **Stream Processing (Flink or Spark)**
   - Process transaction data in real-time for detecting anomalies and applying machine learning models for fraud detection.
4. **Machine Learning Model**
   - The model will be used to detect fraudulent transactions based on features such as transaction amount, frequency, geolocation, user behavior, etc.
   - Models like **XGBoost** or **Random Forest** can be trained on historical transaction data.
   - The trained model will be deployed using **TensorFlow Serving** or **MLflow**.
5. **Alerting System**
   - Based on the results of the fraud detection model, an alert will be triggered to notify the bank or customer about the suspicious transaction.
6. **Data Storage & Retrieval**

    - **Historical Transactions** will be stored in **Cassandra** or **Google BigQuery** for historical analysis.
    - **Real-Time Session Data** (user behavior, transactions) will be stored in **Redis** for fast retrieval.
7. **High Availability & Scalability**

    - Use **Kafka partitions** to scale out the fraud detection pipeline, ensuring high throughput and parallel processing of incoming transactions.
    - Implement **fallback mechanisms** in case the ML model takes longer to respond than expected.

Step-by-Step Design:

1. Transaction Flow: API Gateway → Kafka

The API Gateway is responsible for receiving incoming banking transactions from mobile or web clients. After receiving the transactions, it routes them to Kafka, which acts as the message broker for the real-time event processing pipeline.

Components:

- **API Gateway (Spring Cloud Gateway)**: For routing incoming HTTP requests to appropriate services.
- **Kafka Producer**: For streaming transactions to Kafka topics.

Kafka Setup:

- Kafka topics, such as `banking-transactions`, will hold the transaction events.

java

CopyEdit

```
@Bean

public KafkaTemplate<String, Transaction> kafkaTemplate() {

    return new KafkaTemplate<>(new
DefaultKafkaProducerFactory<>(producerConfigs()));

}



public void sendTransaction(Transaction transaction) {
    kafkaTemplate.send("banking-transactions", transaction.getId(),
transaction);

}
```

2. Stream Processing (Apache Flink or Spark)

Once the transaction data is pushed into Kafka, stream processing technologies like **Apache Flink** or **Apache Spark Streaming** are used to process transactions in real time. This is where fraud detection models are applied to analyze each transaction and identify suspicious behavior.

Flink Setup Example:

java

CopyEdit

```
StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();

FlinkKafkaConsumer<Transaction> consumer = new FlinkKafkaConsumer<>(

        "banking-transactions", new SimpleStringSchema(),
properties);

DataStream<Transaction> transactions = env.addSource(consumer);


transactions

    .keyBy("userId")

    .flatMap(new FraudDetectionFunction()) // Apply ML models here

    .addSink(new AlertSink());
```

Fraud Detection Function (ML Inference Integration)

The fraud detection logic is encapsulated in a **FraudDetectionFunction**. This function will call a machine learning model for each transaction, detect anomalies, and propagate the results to the alert system.

3. Machine Learning Model for Fraud Detection

Training the Model:

We can use a variety of models like **XGBoost** or **Random Forest** for fraud detection. These models can be trained using historical transaction data that contains labeled fraud and non-fraud transactions.

Model Training Code Example (XGBoost):

python

CopyEdit

```
import xgboost as xgb

import pandas as pd

from sklearn.model_selection import train_test_split



# Load historical transaction data

data = pd.read_csv('transaction_data.csv')

X = data.drop('label', axis=1)  # Features

y = data['label']  # Fraud label



# Train-test split

X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2)



# Train the XGBoost model

model = xgb.XGBClassifier()

model.fit(X_train, y_train)



# Evaluate the model

accuracy = model.score(X_test, y_test)
```

```
print(f'Model accuracy: {accuracy}')
```

Once the model is trained, you can deploy it for inference.

Deploying the Model with TensorFlow Serving (or MLflow):

- Use **TensorFlow Serving** or **MLflow** to deploy the trained model as a service, so it can be used for inference in real-time.

Example with TensorFlow Serving:

bash

CopyEdit

```
docker run -p 8501:8501 --name=tf_model --mount
type=bind,source=/models/my_model,target=/models/my_model -e
MODEL_NAME=my_model -t tensorflow/serving
```

The fraud detection service will make API calls to the TensorFlow Serving container to get predictions on new transactions.

Example Inference Call:

java

CopyEdit

```
RestTemplate restTemplate = new RestTemplate();

String url = "http://localhost:8501/v1/models/my_model:predict";

HttpEntity<String> request = new HttpEntity<>(transactionData);

ResponseEntity<String> response = restTemplate.exchange(url,
HttpMethod.POST, request, String.class);
```

4. Alerts and Notifications

Once a transaction is flagged as potentially fraudulent, the system triggers an alert. The alert can be sent via multiple channels, such as:

- **Email**
- **SMS**

- **Webhooks**
- **Dashboard Notifications**

java

CopyEdit

```
public class FraudAlertService {


    public void sendAlert(Transaction transaction) {

        // Logic to send alerts, e.g., email, SMS, etc.

        emailService.sendEmail(transaction.getUserEmail(), "Fraud
Alert", "Suspicious transaction detected.");

    }

}
```

5. Data Storage & Retrieval

To maintain historical data and real-time session information:

- **Historical Transactions**: Store transaction history in **Cassandra** or **BigQuery**, which are optimized for real-time analytics on large datasets.

java

CopyEdit

```
public void saveTransactionToCassandra(Transaction transaction) {

    cassandraTemplate.insert(transaction);

}
```

- **Real-Time Session Data**: Store real-time session data (e.g., transaction count, time, IP address) in **Redis** for fast access.

java

CopyEdit

```
public void storeSessionDataInRedis(String userId, SessionData
sessionData) {

    redisTemplate.opsForHash().put("user-session:" + userId,
sessionData.getSessionId(), sessionData);

}
```

6. High Availability & Scalability

To handle a high volume of transactions:

- **Kafka Partitions**: Partition Kafka topics (e.g., `banking-transactions`) to allow parallel processing of transactions.

bash

CopyEdit

```
bin/kafka-topics.sh --create --topic banking-transactions
--partitions 6 --replication-factor 2 --bootstrap-server
localhost:9092
```

- **Stream Processing Scaling**: Scale Apache Flink or Spark jobs horizontally by adding more processing nodes in the cluster.
- **Fallback Mechanisms**: Implement fallback mechanisms in case the fraud detection model is delayed. Use **circuit breakers** (e.g., **Resilience4J**) to handle timeouts and retries.

java

CopyEdit

```
public class FraudDetectionService {


    private final CircuitBreaker circuitBreaker;


    public FraudDetectionService(CircuitBreaker circuitBreaker) {
        this.circuitBreaker = circuitBreaker;
    }

    public FraudResult detectFraud(Transaction transaction) {
        return circuitBreaker.run(() -> {
            // Make API call to ML model for fraud detection
            return fraudDetectionModel.predict(transaction);
        }, throwable -> new FraudResult("UNKNOWN", "Model
timeout"));
    }
}
```

Conclusion

The **Fraud Detection System for Banking Transactions** relies on an event-driven architecture with **Kafka**, **Stream Processing** (Apache Flink or Spark), and machine learning models for real-time fraud detection. The system processes transaction data in real-time, applies trained models to detect anomalies, and triggers alerts when suspicious transactions are identified.

- **Scalability**: Achieved with Kafka partitions, stream processing, and horizontal scaling of services.
- **Low Latency**: Kafka and stream processing allow for real-time transaction processing.
- **Fault Tolerance**: Implement circuit breakers and fallback mechanisms to handle failures gracefully.

This architecture ensures that the system can handle high-throughput transactions while detecting fraudulent activities quickly and accurately.

4. Design a Scalable Banking Ledger System

💡 *Scenario:* Build a ledger system that handles millions of transactions.

Key Discussion Points:

✅ **Data Model**

- **Event-Sourcing Architecture** (store immutable transactions)
- Use **CQRS (Command Query Responsibility Segregation)**

✅ **Storage & Consistency**

- ACID-compliant DB for critical data (PostgreSQL)
- Append-only **Kafka topics** for audit logs

✅ **Concurrency Handling**

- **Optimistic concurrency control**
- **Idempotency tokens** for retry handling

Designing a Scalable Banking Ledger System

In this design, we aim to build a **banking ledger system** capable of handling **millions of transactions** efficiently while ensuring **data consistency, fault tolerance, and scalability**. This system will handle transactional data in a way that it can maintain **immutability, provide auditability**, and support **high availability**.

The system will use **Event-Sourcing** architecture, implement **CQRS (Command Query Responsibility Segregation)**, and utilize a variety of scalable, high-performance technologies to manage concurrency, transactions, and storage.

Key Design Components & Architecture:

1. **Event-Sourcing Architecture**
    - **Event-sourcing** involves storing **events** (state transitions) rather than storing the state itself.
    - All **transactions** (debits, credits, etc.) will be stored as **immutable events** that describe the changes to the system, allowing for **auditability** and **replayability** of past events.
    - Events will be persisted to **Kafka topics** and will be used to reconstruct the **current state** of the ledger.
2. **CQRS (Command Query Responsibility Segregation)**
    - **CQRS** separates the **command side** (writes) from the **query side** (reads).
    - This allows the system to scale both read and write operations independently.
    - **Command side**: Handles operations like creating a new transaction or updating account balances.
    - **Query side**: Provides a read-optimized view of the data, such as querying account balances or transaction history.
3. **Data Model**

- The core data model will consist of **Transactions**, **Accounts**, and **Balance Events**.
- Each transaction will be represented as an event with properties like transaction ID, amount, type (debit/credit), timestamp, account ID, etc.
- The **Ledger** will be composed of a stream of events that can be replayed to reconstruct the system state.

4. **Example data model:**

    - **Transaction Event**: A simple event containing the transaction type, amount, timestamp, etc.
    - **Account**: The entity representing a customer's bank account.
    - **Balance**: Representing the balance in the account, computed from transaction events.

Example:

java
CopyEdit

```java
public class Transaction {

    private String transactionId;

    private String accountId;

    private BigDecimal amount;

    private String type; // "DEBIT" or "CREDIT"

    private LocalDateTime timestamp;

}



public class Account {

    private String accountId;

    private BigDecimal balance;

    private List<Transaction> transactionHistory;

}
```

5.
6. **Storage & Consistency**

- **PostgreSQL** will be used for **ACID-compliant** storage of critical data, such as account balances, because of its strong consistency and support for complex queries.
- Kafka will be used for **event logs**, storing immutable **transaction events** for the ledger, which will be append-only for audit purposes.

7. **Storage Components:**

    - **ACID-compliant DB**: PostgreSQL will store the final state of accounts (balances) and ensure atomicity, consistency, isolation, and durability.
    - **Append-only Kafka topics**: Kafka will store event streams for **audit logging** and event-sourcing.

8. Example:

    - **PostgreSQL schema** for storing accounts and balances:

sql
CopyEdit
```
CREATE TABLE accounts (

    account_id UUID PRIMARY KEY,

    balance NUMERIC(10,2) NOT NULL

);
```

9.
    - **Kafka Topic** for storing transaction events:

bash
CopyEdit
```
kafka-topics.sh --create --topic banking-transactions --partitions 3
--replication-factor 2 --bootstrap-server localhost:9092
```

10.
11. **Concurrency Handling**

    Concurrency handling ensures that the ledger can handle **high-frequency** updates from multiple transactions without data corruption.

    - **Optimistic Concurrency Control (OCC)**: OCC will be used to avoid conflicts when two processes try to modify the same data (e.g., account balance) simultaneously. The transaction will proceed if no other changes have been made since the last read, and a retry mechanism will handle conflicts.

    - **Idempotency Tokens**: For retries and ensuring that duplicate transactions don't occur, we will implement **idempotency tokens**. These tokens ensure that when the same transaction is retried, it will not be executed again.

Optimistic Concurrency Control (OCC) Example:

When a transaction is created, the system checks whether the **account balance** has changed since the last read. If it has, the system will **retry** the transaction.

java
CopyEdit

```
public class AccountService {

    public boolean transferAmount(String accountId, BigDecimal
amount) {

        Account account = accountRepository.findById(accountId);

        BigDecimal currentBalance = account.getBalance();

        if (currentBalance.compareTo(amount) < 0) {

            return false; // Insufficient funds

        }

        // Proceed with transaction only if account state is
unchanged

        if (accountRepository.updateBalance(accountId,
currentBalance.subtract(amount))) {

            transactionProducer.sendTransaction(new
Transaction(accountId, amount));

            return true;

        } else {

            return false; // Failed to update due to concurrent
modification

        }

    }

}
```

Idempotency Token Example:

Each transaction is assigned a unique **idempotency key** that is used to ensure that duplicate transaction submissions are ignored.

java
CopyEdit
```
public class TransactionService {

    private Map<String, String> transactionCache = new HashMap<>();


    public boolean processTransaction(String transactionId, String
idempotencyKey) {

        if (transactionCache.containsKey(idempotencyKey)) {

            return false; // Transaction has already been processed

        }

        // Process the transaction

        transactionCache.put(idempotencyKey, transactionId);

        transactionRepository.save(new Transaction(transactionId,
...));

        return true;

    }

}
```

12.
13. **Scalability & High Availability**

   - **Kafka** will handle high-throughput event streaming and ensure that all events are captured for audit and event-sourcing.
   - **Kubernetes** will be used for container orchestration and auto-scaling, ensuring that the system scales according to demand.
   - **PostgreSQL** will be deployed with **read replicas** to scale out read-heavy queries (balance checks, transaction history) while ensuring that write operations (transaction events) are handled by the primary database.

14. **Kubernetes Setup for Scalability**:

   - Deploy Kafka and PostgreSQL with replication for high availability.
   - Deploy microservices that handle transaction processing and query handling in multiple pods.

Example of Kubernetes deployment for PostgreSQL:

yaml
CopyEdit
```yaml
apiVersion: apps/v1

kind: Deployment

metadata:

  name: postgresql-deployment

spec:

  replicas: 3

  selector:

    matchLabels:

      app: postgresql

  template:

    metadata:

      labels:

        app: postgresql

    spec:

      containers:

      - name: postgresql

        image: postgres:12

        ports:

        - containerPort: 5432

        env:

        - name: POSTGRES_USER

          value: "postgres"
        - name: POSTGRES_PASSWORD

          value: "password"
```

15.

End-to-End Flow of the Scalable Banking Ledger System

1. **Transaction Initiation (Command Side)**
   - The API Gateway routes incoming requests for transactions to the appropriate microservice.
   - The **Transaction Service** receives the transaction, verifies the amount and the account balance, and creates an event representing this transaction.
2. **Event Publishing to Kafka**
   - The transaction event is published to a **Kafka topic** (`banking-transactions`), ensuring immutability and auditability.
3. **Processing Events (CQRS Query Side)**
   - On the **query side**, an event handler listens for new transactions from Kafka, and updates the **read model** of the ledger, which could be stored in PostgreSQL (for account balances and transaction history).
   - The **event handler** is optimized for reading the events and updating the state of the system.
4. **Concurrency & Retry Handling**
   - The system ensures consistency with **optimistic concurrency** and **idempotency tokens**.
5. **Querying the Ledger**
   - For querying, users or services interact with the **read-optimized database**, which is built to handle high-volume read operations (e.g., balance checks, transaction history queries).

Conclusion

This **scalable banking ledger system** provides high performance and fault tolerance for handling millions of transactions. Key aspects such as **event-sourcing**, **CQRS**, **Kafka-based event streaming**, **ACID-compliant PostgreSQL storage**, **optimistic concurrency**, and **idempotency tokens** ensure that the system is **highly available**, **scalable**, and **consistent**. The system also leverages **Kubernetes** for orchestration and scaling to meet growing demands.

. Design a Real-Time Payment Processing System

💡 *Scenario:* Build a high-throughput, low-latency system to process transactions in real time.

Key Discussion Points:

✅ **System Components:**

- API Gateway (Spring Cloud Gateway)
- Transaction Processing Service (Spring Boot, gRPC for low latency)
- Event-Driven Architecture (Kafka for async processing)
- ACID-compliant Database (PostgreSQL/MySQL with read replicas)
- Fraud Detection Module (Kafka Streams + ML models)
- Notifications (WebSockets, Webhooks, SMS/Email)

✅ **Concurrency & Consistency**

- Use **Optimistic Locking** or **Pessimistic Locking** for database updates
- Implement **idempotency keys** for retrying failed transactions

✅ **Scalability & Fault Tolerance**

- Use **Kafka with partitions** for event-driven transaction processing
- Implement **circuit breakers** (Resilience4J) to handle failures
- Deploy with **Kubernetes (K8s) + Auto-scaling**

✅ **Security & Compliance**

- **OAuth2 + JWT** for authentication
- **TLS & AES Encryption** for securing transactions
- **PCI-DSS Compliance** for storing sensitive data

In this design, the goal is to build a **high-throughput, low-latency real-time payment processing system**. This system will process millions of transactions per second, while maintaining **data consistency**, **scalability**, **fault tolerance**, and **security**.

Here's how we can break down and build this system using the outlined components:

Key Components

1. API Gateway (Spring Cloud Gateway)

- **Role**: The **API Gateway** acts as the entry point for all payment requests. It handles routing, authentication, rate-limiting, and can forward the requests to appropriate microservices for processing.
- **Key Features**:
    - **Authentication**: OAuth2 + JWT for securing API requests.
    - **Request Routing**: Routes payment requests to the **Transaction Processing Service**.

    - **Rate Limiting**: Protects backend services from being overwhelmed by too many requests.

2. Transaction Processing Service (Spring Boot, gRPC for Low Latency)

- **Role**: This service processes each payment transaction. It performs essential tasks such as:
    - **Transaction validation**: Ensure payment details are valid (account, amount, etc.).
    - **Debit or credit accounts**: Perform the required financial transaction (e.g., debit one account, credit another).
    - **Transaction status updates**: Track the state of each transaction (pending, completed, failed).
- **Low Latency**: Use **gRPC** for communication between services to ensure low-latency request/response cycles.
    - **gRPC Benefits**: Protocol Buffers for serialization (faster than JSON), HTTP/2 for multiplexing, and bidirectional communication.

3. Event-Driven Architecture (Kafka for Asynchronous Processing)

- **Role**: Kafka handles the **asynchronous processing** of events, ensuring that no part of the system gets blocked while waiting for operations to complete.
    - **Kafka Topics**: Separate topics for transaction requests, transaction results, fraud detection, and notifications.
    - **Event Processing**: Once the transaction is processed by the **Transaction Service**, an event is pushed to Kafka for further processing (e.g., fraud detection, notifications).

4. ACID-compliant Database (PostgreSQL/MySQL with Read Replicas)

- **Role**: The database stores the **transaction history** and **account balances** in a reliable, **ACID-compliant** manner.
    - **ACID Compliance**: Ensures data integrity, which is critical for payment systems. If a failure happens during a transaction, the system can roll back to a consistent state.
    - **Read Replicas**: Use **read replicas** for handling heavy read queries (e.g., checking account balance) without overloading the primary database.

5. Fraud Detection Module (Kafka Streams + ML Models)

- **Role**: The fraud detection module uses machine learning models to detect suspicious transactions in real-time. It processes events streamed from Kafka and raises alerts if any fraudulent activity is suspected.
    - **Kafka Streams**: A Kafka-based stream processing library that processes each transaction in real-time, checking it against fraud detection rules or ML models.
    - **ML Models**: Pre-trained machine learning models (e.g., XGBoost, Random Forest) for detecting patterns indicative of fraud.

6. Notifications (WebSockets, Webhooks, SMS/Email)

- **Role**: After processing the transaction and performing necessary checks (like fraud detection), the system sends notifications to the customer or backend systems.
  - **WebSockets**: For real-time notifications to users.
  - **Webhooks**: For notifying external systems (e.g., merchant systems, partner services).
  - **SMS/Email**: For sending transaction updates or alerts to users.

Concurrency & Consistency

1. Database Locking

- **Optimistic Locking**: Ideal for high-concurrency systems. We assume that conflicts are rare and allow multiple transactions to work in parallel, checking if data has changed before committing. If a conflict occurs (e.g., another process modified the same account), the system will retry the transaction.
  - **Implementation**: Include a version number in the transaction data. If the version has changed, the transaction is aborted and retried.
- **Pessimistic Locking**: In scenarios where conflicts are expected frequently, we lock the account record for a specific transaction, ensuring no other transaction can modify it until the first transaction is complete.
  - **Implementation**: When debiting/crediting accounts, lock the account to prevent race conditions.

2. Idempotency Keys

- **Idempotency** ensures that **retrying** a failed transaction does not result in multiple processing of the same transaction.
  - **Idempotency Key**: The client generates a unique key for each transaction. If a transaction request is received with the same key, the system will either return the previous result or ignore the duplicate request.

Scalability & Fault Tolerance

1. Kafka with Partitions for Event-Driven Processing

- **Kafka Partitions**: Kafka supports partitioning to scale transaction processing. Transactions will be processed in parallel across multiple consumers, with each partition containing a subset of the transaction stream.
- **Fault Tolerance**: Kafka ensures fault tolerance by replicating partitions. If a broker fails, another replica broker can take over without losing events.

2. Circuit Breakers (Resilience4J)

- **Resilience4J**: Implements circuit breakers to prevent cascading failures in case of issues with downstream systems (e.g., payment gateway failures, fraud detection service failures).
    - If a service becomes unavailable or slow, the circuit breaker opens, preventing requests from reaching the failing service until it recovers.

3. Kubernetes (K8s) + Auto-scaling

- **Kubernetes**: The system will be deployed using **Kubernetes** to ensure **scalability** and **fault tolerance**. Kubernetes can automatically scale services based on traffic and resource utilization.
    - Use **Horizontal Pod Autoscaler** to scale out **Transaction Processing Service** and **Fraud Detection Modules** dynamically.
    - Ensure high availability by using multiple replicas of services and configuring **Pod Disruption Budgets**.

Security & Compliance

1. OAuth2 + JWT for Authentication

- **OAuth2**: The system uses **OAuth2** for **authentication** and **authorization**. The API Gateway will authenticate the user before forwarding the request to microservices.
    - **JWT**: After successful authentication, a **JWT (JSON Web Token)** is issued to the client to be used for future requests, ensuring stateless authentication.

2. TLS & AES Encryption for Transaction Security

- **TLS**: All communication (API requests, Kafka messages, etc.) will be encrypted using **TLS (Transport Layer Security)** to protect data in transit.
- **AES Encryption**: **AES (Advanced Encryption Standard)** will be used to encrypt sensitive transaction data, such as payment details, before storing it in the database.

3. PCI-DSS Compliance

- **PCI-DSS**: The system will follow **PCI-DSS (Payment Card Industry Data Security Standard)** guidelines for storing and processing credit card information securely. Sensitive data such as card numbers will be encrypted and stored in accordance with these standards.

High-Level Architecture Diagram

Here’s a high-level architecture of the **Real-Time Payment Processing System**:

pgsql

CopyEdit

```
+--------------------+               +-------------------------+
|                    |               |                         |
|   Client (Mobile,   |    HTTPS      |      API Gateway        |
|   Web, etc.)        +--------------->+    (Spring Cloud        |
|                    |               |    Gateway)             |
+--------------------+               +-------------------------+
                                             |
                                             v
                                      +-------------------------+
                                      |   Transaction Processing |
                                      |      Service (Spring     |
                                      |       Boot, gRPC)        |
                                      +-------------------------+
                                             |
                                             v
                                        +----------------------+
                                        |   Kafka (Event-Driven |
                                        |   Architecture)      |
                                        +----------------------+
+-----------------------+
                                             |                   |
|
                           +---------------------+
+---------------------+     +----------------------+
                           |  Fraud Detection     |      |
Notifications (SMS,  |      | ACID Database (Post-|
                           | (Kafka Streams + ML) |      |
WebSockets, Email)   |     | GreSQL/MySQL)       |

                           +---------------------+
+---------------------+     +---------------------+
```

End-to-End Flow

1. **Client Requests Transaction**:
    - A user submits a payment request through the **API Gateway** (secured via OAuth2 + JWT).
    - The **API Gateway** routes the request to the **Transaction Processing Service**.
2. **Transaction Processing**:
    - The **Transaction Processing Service** verifies the payment details, checks for sufficient funds, and performs the necessary transaction (debit/credit).
    - The transaction event is published to a Kafka topic for processing by other services.
3. **Fraud Detection**:
    - The event is processed by the **Fraud Detection Module** via **Kafka Streams**. If fraud is detected, an alert is triggered.
4. **Notifications**:
    - Upon successful completion or failure, the system sends notifications via **SMS, Email, or WebSocket** to the user or merchant.
5. **ACID Database**:
    - The final state of the transaction (balance updates, transaction history) is stored in the **PostgreSQL/MySQL** database.

Conclusion

The **Real-Time Payment Processing System** ensures **high throughput**, **low latency**, and **scalability** by leveraging technologies like **Spring Cloud Gateway**, **gRPC**, **Kafka**, **PostgreSQL**, **Kafka Streams**, and **Resilience4J**. With a focus on security (OAuth2, JWT, TLS, AES encryption), **fraud detection**, **event-driven architecture**, and **PCI-DSS compliance**, this system is designed to handle millions of transactions per second while maintaining **data integrity**, **reliability**, and **fault tolerance**.

Below is a **high-level implementation** for a **Real-Time Payment Processing System** using **Spring Boot**, **gRPC**, **Kafka**, **PostgreSQL**, and other related components.

We'll break it down into the following steps:

1. API Gateway (Spring Cloud Gateway)

For routing requests to the appropriate services, we use **Spring Cloud Gateway**. The API Gateway will handle authentication, routing, and logging.

`application.yml` configuration for Spring Cloud Gateway

yaml

CopyEdit

```
spring:
  cloud:
    gateway:
      routes:
        - id: payment-service
          uri: lb://payment-service
          predicates:
            - Path=/payments/**
          filters:
            - AddRequestHeader=X-Request-Foo, Bar
            - AddResponseHeader=X-Response-Foo, Bar
```

- **OAuth2 Configuration** (for Authentication):

yaml

CopyEdit

```
security:
  oauth2:
    client:
      registration:
        google:
          client-id: your-client-id
          client-secret: your-client-secret
          scope: profile, email
          redirect-uri:
"{baseUrl}/login/oauth2/code/{registrationId}"
    resourceserver:
      jwt:
        issuer-uri: http://your-issuer-uri
```

This configuration will authenticate API requests and route them to the `payment-service`.

---

2. Transaction Processing Service (Spring Boot + gRPC)

Here, we implement the core payment processing logic. We'll use **gRPC** for low-latency communication.

pom.xml dependencies for Spring Boot and gRPC

xml

CopyEdit

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
```

```
            <groupId>org.springframework.boot</groupId>

            <artifactId>spring-boot-starter-data-jpa</artifactId>

        </dependency>

        <dependency>

            <groupId>org.springframework.cloud</groupId>

            <artifactId>spring-cloud-starter-gateway</artifactId>

        </dependency>

        <dependency>

            <groupId>io.grpc</groupId>

            <artifactId>grpc-spring-boot-starter</artifactId>

            <version>2.8.0</version>

        </dependency>

    </dependencies>
```

gRPC Service Definition (`payment.proto`)

proto

CopyEdit

```
syntax = "proto3";


service PaymentService {

  rpc ProcessPayment (PaymentRequest) returns (PaymentResponse);

}


message PaymentRequest {

  string account_number = 1;
  double amount = 2;

}


message PaymentResponse {

  string transaction_id = 1;

  string status = 2;

}
```

gRPC Server Implementation (Spring Boot)

java

CopyEdit

```
import io.grpc.stub.StreamObserver;

import org.springframework.stereotype.Service;

import com.proto.payment.PaymentRequest;

import com.proto.payment.PaymentResponse;

import com.proto.payment.PaymentServiceGrpc;


@Service

public class PaymentServiceImpl extends
PaymentServiceGrpc.PaymentServiceImplBase {


    @Override

    public void processPayment(PaymentRequest request,
StreamObserver<PaymentResponse> responseObserver) {

        String accountNumber = request.getAccountNumber();
        double amount = request.getAmount();

        // Call business logic here (check account, process
transaction, etc.)

        PaymentResponse response = PaymentResponse.newBuilder()
                .setTransactionId("TXN12345")
                .setStatus("SUCCESS")
                .build();

        responseObserver.onNext(response);
        responseObserver.onCompleted();
    }
}
```

This is the core service that handles the payment processing.

---

3. Event-Driven Architecture with Kafka

For asynchronous processing of events like **Fraud Detection**, **Transaction Logging**, etc., we'll use **Kafka**.

Kafka Configuration in `application.yml`

yaml

CopyEdit

```yaml
spring:
  kafka:
    producer:
      bootstrap-servers: localhost:9092

      key-serializer:
org.apache.kafka.common.serialization.StringSerializer

      value-serializer:
org.apache.kafka.common.serialization.StringSerializer

    consumer:

      bootstrap-servers: localhost:9092

      group-id: payment-group

      key-deserializer:
org.apache.kafka.common.serialization.StringDeserializer

      value-deserializer:
org.apache.kafka.common.serialization.StringDeserializer
```

Kafka Producer (for publishing transaction events)

java

CopyEdit

```
import org.apache.kafka.core.KafkaTemplate;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.kafka.core.KafkaTemplate;

import org.springframework.kafka.annotation.EnableKafka;

import org.springframework.kafka.core.KafkaTemplate;


@Service

public class KafkaTransactionProducer {


    @Autowired
    private KafkaTemplate<String, String> kafkaTemplate;


    public void publishTransactionEvent(String transactionId, String
status) {
        String message = "Transaction ID: " + transactionId + ",
Status: " + status;
        kafkaTemplate.send("transaction-topic", message);
    }
}
```

Kafka Consumer (for Fraud Detection)

java

CopyEdit

```
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.stereotype.Service;


@Service
public class KafkaTransactionConsumer {


    @KafkaListener(topics = "transaction-topic", groupId =
"payment-group")
    public void consumeTransactionEvent(ConsumerRecord<String,
String> record) {
        String event = record.value();
        // Logic to detect fraud based on the event
        System.out.println("Processing transaction event: " +
event);
    }
}
```

---

4. ACID-compliant Database (PostgreSQL)

For transaction data persistence, we will use **PostgreSQL** with **JPA** and **Spring Data**.

Database Configuration in `application.yml`

yaml

CopyEdit

```
spring:
  datasource:
    url: jdbc:postgresql://localhost:5432/payment_db
    username: user
    password: password
  jpa:
    hibernate:
      ddl-auto: update
    show-sql: true
```

JPA Entity for Transaction

java

CopyEdit

```
import javax.persistence.Entity;
import javax.persistence.Id;


@Entity

public class PaymentTransaction {

    @Id

    private String transactionId;

    private String accountNumber;

    private double amount;

    private String status;


    // Getters and Setters

}
```

JPA Repository

java

CopyEdit

```
import org.springframework.data.jpa.repository.JpaRepository;


public interface PaymentTransactionRepository extends
JpaRepository<PaymentTransaction, String> {

}
```

---

5. Fraud Detection Module (Kafka Streams + ML Models)

For fraud detection, we use **Kafka Streams** to process events in real time and trigger ML-based fraud detection.

Kafka Stream Processor (For Fraud Detection)

java

CopyEdit

```
import org.apache.kafka.streams.kstream.KStream;

import org.apache.kafka.streams.processor.Processor;

import org.apache.kafka.streams.processor.ProcessorContext;

import org.springframework.kafka.annotation.EnableKafkaStreams;

import org.springframework.kafka.annotation.StreamListener;

import org.springframework.stereotype.Service;


@EnableKafkaStreams

@Service

public class FraudDetectionService {


    @StreamListener("transaction-topic")

    public void processFraudDetection(KStream<String, String> input)
{

        input.foreach((key, value) -> {

            // ML-based fraud detection logic here

            if (isFraudulent(value)) {

                // Trigger alert

                System.out.println("Fraud detected for transaction:
" + value);

            }

        });
```

```java
    }


    private boolean isFraudulent(String transactionData) {

        // Implement ML model inference for fraud detection

        return false;

    }

}
```

---

6. Notifications via WebSockets/SMS/Email

We will send notifications to the user based on transaction status.

WebSocket Notification Example

java

CopyEdit

```java
import org.springframework.web.socket.WebSocketSession;

import org.springframework.web.socket.handler.TextWebSocketHandler;

import org.springframework.stereotype.Component;


@Component

public class WebSocketNotificationHandler extends
TextWebSocketHandler {


    @Override

    public void handleTextMessage(WebSocketSession session,
TextMessage message) throws Exception {

        // Send transaction update to WebSocket client
        session.sendMessage(new TextMessage("Transaction Completed: 
" + message.getPayload()));

    }

}
```

Email/SMS Notification

Using **Spring Mail** and **Twilio** (for SMS), we can send notifications.

java

CopyEdit

```
import org.springframework.mail.javamail.JavaMailSender;

import org.springframework.mail.javamail.MimeMessageHelper;

import org.springframework.beans.factory.annotation.Autowired;


@Service

public class NotificationService {


    @Autowired

    private JavaMailSender emailSender;


    public void sendEmailNotification(String to, String subject, 
String body) {

        MimeMessage message = emailSender.createMimeMessage();

        MimeMessageHelper helper = new MimeMessageHelper(message, 
true);

        helper.setTo(to);

        helper.setSubject(subject);
        helper.setText(body);

        emailSender.send(message);

    }

}
```

Conclusion

This is a high-level implementation for a **Real-Time Payment Processing System**. It includes core components like:

- **API Gateway (Spring Cloud Gateway)** for routing and authentication.
- **Transaction Processing Service (Spring Boot + gRPC)** for low-latency processing.
- **Event-Driven Architecture (Kafka)** for transaction handling and fraud detection.
- **ACID-compliant Database (PostgreSQL)** for reliable transaction storage.
- **Fraud Detection with Kafka Streams** and **ML models**.
- **Notifications via WebSockets, SMS, and Email**.

This system is designed to handle millions of transactions per second, with **low latency**, **scalability**, **fault tolerance**, and **security**. You can further extend it to handle **circuit breakers**, **retry mechanisms**, and other resilience strategies for robust real-time processing.