Architecture Design for Real-Time Streaming Platform

This architecture leverages modern tools to handle real-time data ingestion, processing, storage, aggregation, and retrieval for an online shopping network, ensuring high performance, scalability, and reliability.

Proposed Design Components

1. Event Ingestion Layer: Apache Kafka

- **Purpose**: To handle real-time ingestion of events such as product views, add-to-cart actions, and ad engagements from retailer websites.
- **Reason for Choosing Kafka**:
    - High throughput and low latency for real-time event ingestion.
    - Persistent storage with replay capabilities for fault tolerance and historical reprocessing.
    - Built-in partitioning and replication for scalability and reliability.

2. Stream Processing: Apache Flink

- **Purpose**: For real-time data aggregation, event enrichment, and complex transformations (e.g., window-based aggregations, data cleansing etc.).
- **Reason for Choosing Flink**:
    - Native support for event time and windowed processing for real-time analytics.
    - Fault tolerance with state snapshots stored in HDFS (or S3-like object storage).
    - Scalable and distributed architecture for handling high volumes of data.

3. Raw Data Storage: HDFS

- **Purpose**: To store raw events for future playback or batch processing.
- **Reason for Choosing HDFS**:
    - Cost-effective and reliable storage for large-scale, append-only logs.
    - Integration with big data frameworks for historical analytics and replay.

4. Aggregated and Descriptive Data Storage: ClickHouse

- **Purpose**: To store aggregated and descriptive data for OLAP-style querying and reporting.

- **Reason for Choosing ClickHouse**:
    - Columnar storage and compression for fast analytics and minimal storage footprint.
    - Real-time ingestion capabilities for aggregated data.
    - Highly optimized for time-series queries, making it suitable for dimensional and aggregated data.

5. Caching Layer: Redis

- **Purpose**: For storing highly aggregated data (e.g., day-wise, hourly, or 15-minute counts) to serve APIs with low latency.
- **Reason for Choosing Redis**:
    - Extremely low latency for read and write operations.
    - Support for data structures like hash maps and sorted sets, suitable for time-based aggregation.
    - Ability to expire data automatically, reducing the need for manual cleanup.

6. API and Data Streaming Layer: Spring WebFlux

- **Purpose**: To expose RESTful and streaming APIs for client applications (e.g., retailers, advertisers, analytics dashboards).
- **Reason for Choosing Spring WebFlux**:
    - Non-blocking and reactive programming model for handling high-concurrency requests.
    - Efficient for real-time streaming use cases.
    - Seamless integration with Kafka, Redis, and ClickHouse.

7. Authorization: JWT

- **Purpose**: To authenticate and authorize clients accessing the platform.
- **Reason for Choosing JWT**:
    - Stateless and scalable, as tokens can be validated without querying a centralized database.
    - Ability to include metadata like client name, client ID, and token expiration.

8. Monitoring and Observability: Prometheus and Grafana

- **Purpose**: To monitor system performance, resource utilization, and key metrics.
- **Reason for Choosing Prometheus**:
  - Time-series database optimized for monitoring and alerting.
  - Support for custom metrics from Kafka, Flink, Redis, etc.
- **Reason for Choosing Grafana**:
  - Rich visualization options for creating custom dashboards.
  - Ability to integrate data sources like Prometheus and ClickHouse.

Approach to Multi-Tenancy

Database-Level Multi-Tenancy

Choose one of the following database approaches depending on the level of isolation and cost considerations:

Separate Databases for Each Tenant:

- Each retailer has a separate database instance.
- Ensures maximum data isolation and security.
- Suitable for high-value tenants or regulatory requirements.
- Example: Retailer A's data is in `db_retailer_a` and Retailer B's data in `db_retailer_b`.
- Add a `WHERE client_id = ?` clause in all database queries if using a shared database.
- If using separate databases, use a Tenant Routing DataSource that dynamically routes database connections based on the tenant ID from the JWT or request context.

Shared Database with Separate Schemas:

- Use a single database instance but create separate schemas for each tenant.
- Provides logical isolation within the database.
- Example: Tables `retailer_a.click_events` and `retailer_b.click_events`.

Shared Database with Tenant Identifier:

- Store all tenants' data in the same database and tables but add a **client_id** column to segregate data.
- Requires strict query filtering by `client_id` to ensure data isolation.

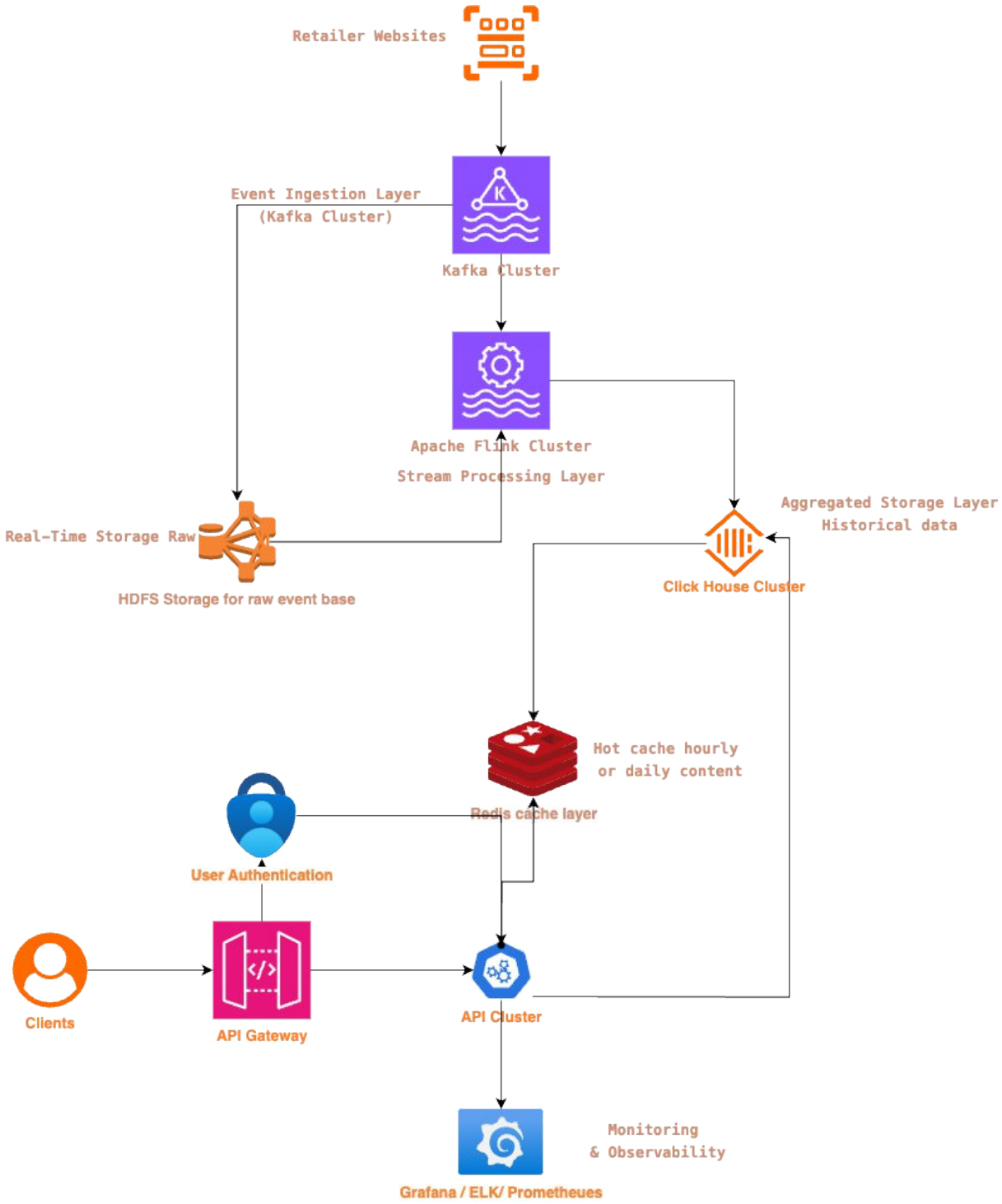
Authentication and Authorization

- Use **JWT tokens** to identify the tenant in each request.
- Include the `client_id` in the JWT payload upon authentication.
- Ensure that every API request is validated against the `client_id` to prevent unauthorized access to another tenant's data.

Security Measures

- **Encryption**:
  - Encrypt sensitive data at rest and in transit.
  - Encrypt tenant-specific database credentials.
- **Access Control**:
  - Use role-based access control (RBAC) to restrict actions for users of each tenant.

Architecture Diagram

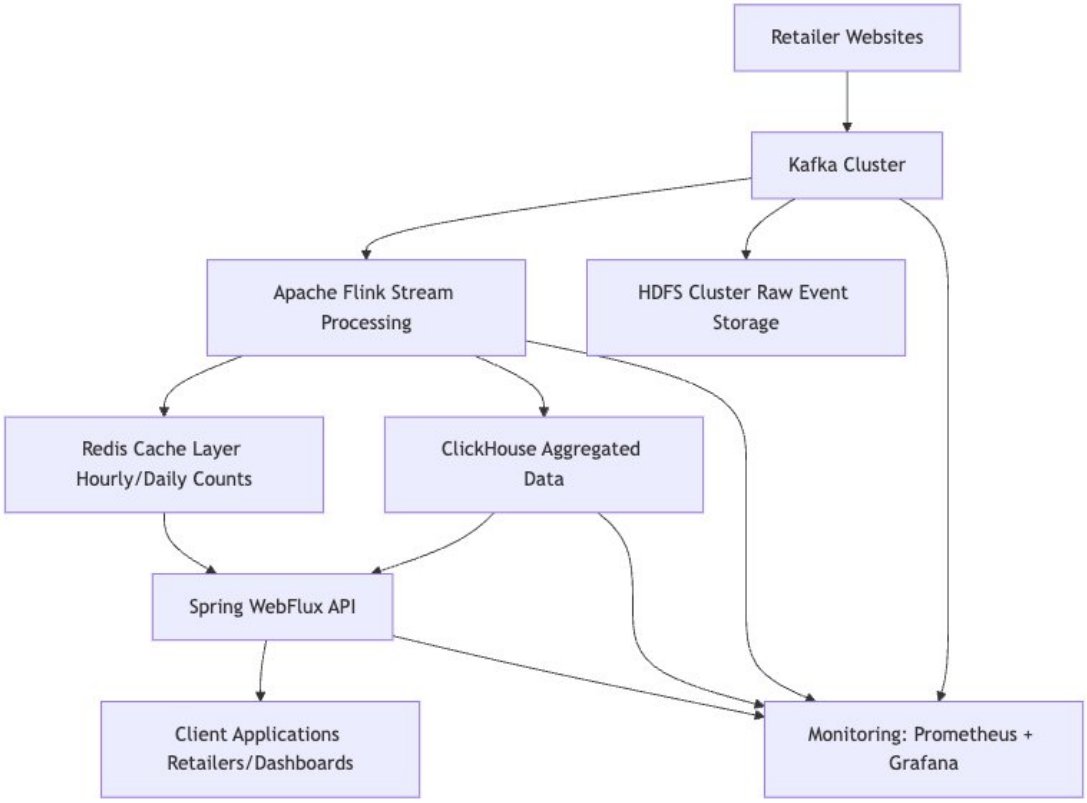


Workflow

1. **Event Ingestion**
   - Retailer websites push customer interactions (e.g., product views, clicks) to Kafka topics.
   - Kafka persists events and partitions them for parallel consumption.
   - Stores raw data in HDFS for replay or historical analytics.
2. **Stream Processing**
   - Apache Flink consumes Kafka topics and processes events in real-time.
   - Aggregates data based on dimensions (e.g. session, client, product, category, timestamp).
   - Stores aggregated data in ClickHouse for querying.
3. **Caching**
   - Hourly, daily, or short-term aggregated counts are stored in Redis for fast access.
   - Example: A Redis key could be structured as

     `client_<client_id>_campaign_<campaign_id>_hourly_<hour>`.
4. **API Layer**
   - Spring WebFlux APIs retrieve aggregated data from ClickHouse or Redis.
   - JWT tokens are validated before processing requests.
   - Streaming APIs stream real-time updates to clients (e.g., dashboards).
5. **Monitoring**
   - Metrics are pushed to Prometheus from Kafka, Flink, ClickHouse, and the API layer.
   - Grafana visualizes key performance indicators like throughput, latency, and error rates.

Deployment Diagram



Key Benefits

1. **Scalability**
   - Kafka and Flink ensure the system can handle millions of events per second.
   - Redis and ClickHouse optimize data retrieval for low-latency APIs.
2. **Fault Tolerance**
   - Kafka's replication, Flink's state snapshots, and HDFS ensure high availability.
3. **Data Isolation**
   - Retailer-specific data can be segregated using tenant IDs in ClickHouse, Redis, and APIs.

4. **Flexibility**
   - Modular design allows switching components (e.g., replacing HDFS with S3).
5. **Observability**
   - Prometheus and Grafana provide a real-time view of system health.

Advantages of This Design

1. **Scalability**: Supports both lightweight tenants (shared database) and high-value tenants (dedicated database).
2. **Data Isolation**: Prevents data leaks between retailers.
3. **Security**: Ensures compliance with regulations like GDPR by providing tenant-specific encryption and access control.
4. **Flexibility**: Easy to onboard new tenants or migrate existing ones to dedicated databases.