System Design Architecture

1. Data Sources

- **Live Data Streams**: Two live data streams directly from NASDAQ.
- **Historical Data Storage**: A database (e.g., PostgreSQL, MongoDB, or a time-series database like TimescaleDB) to store historical stock prices.

2. Components

1. **API Gateway**
    - Acts as the entry point for client requests.
    - Handles authentication, rate limiting, and routing to appropriate services.
2. **Live Price Service**
    - Consumes live data streams from NASDAQ.
    - Processes and caches live stock prices for quick retrieval.
    - Publishes updates to a **real-time messaging system** (e.g., WebSocket, Kafka, or Redis Pub/Sub) for clients needing live updates.
3. **Historical Price Service**
    - Fetches historical stock prices from the database.
    - Handles queries with date ranges and aggregates data if needed (e.g., daily, weekly, or monthly averages).
4. **Data Ingestion Service**
    - Ingests live data streams from NASDAQ.
    - Stores snapshots of stock prices periodically into the historical database for long-term storage.
5. **Cache Layer**
    - A caching system (e.g., Redis or Memcached) to store frequently accessed live and historical data for faster response times.
6. **Database**

- **Time-Series Database**: Stores historical stock prices with timestamps for efficient querying.
- **Relational Database**: Stores metadata about stocks (e.g., ticker symbols, company names).

7. **Real-Time Messaging System**
   - Enables clients to subscribe to live stock price updates via WebSocket or similar protocols.

3. Workflow

1. **Live Price Retrieval**
   - Client sends a request to the API Gateway for a live stock price.
   - API Gateway routes the request to the Live Price Service.
   - Live Price Service retrieves the data from the cache or live data stream and returns it to the client.
2. **Historical Price Retrieval**
   - Client sends a request to the API Gateway with a ticker and date range.
   - API Gateway routes the request to the Historical Price Service.
   - Historical Price Service queries the database and returns the data to the client.
3. **Data Ingestion**
   - Data Ingestion Service continuously consumes live data streams from NASDAQ.
   - Updates the cache with the latest prices for live queries.
   - Periodically writes snapshots to the historical database.
4. **Real-Time Updates**
   - Clients subscribe to a WebSocket endpoint for live updates.
   - Live Price Service pushes updates to the messaging system, which broadcasts them to subscribed clients.

4. Technology Stack

- **API Gateway**: AWS API Gateway, NGINX, or Express.js.
- **Backend Framework**: Node.js, Python (FastAPI), or Java (Spring Boot).
- **Database**: PostgreSQL with TimescaleDB, MongoDB, or InfluxDB.
- **Cache**: Redis or Memcached.
- **Messaging System**: Kafka, RabbitMQ, or Redis Pub/Sub.
- **Real-Time Communication**: WebSocket or Server-Sent Events (SSE).
- **Cloud Infrastructure**: AWS, Azure, or GCP for scalability and reliability.

5. Scalability Considerations

- **Horizontal Scaling**: Scale API servers and data ingestion services independently.
- **Load Balancer**: Distribute traffic across multiple API servers.
- **Partitioning**: Partition historical data by ticker symbol or time range for efficient querying.
- **High Availability**: Use replication and failover mechanisms for the database and cache.

在设计中，**live price of a ticker（某个股票的实时价格）** 通常直接从 **NASDAQ API 的实时数据流** 中获取，而不是从快照中来。以下是原因和详细说明：

为什么实时价格直接从 NASDAQ API 获取？

1. **实时性要求**
    - 客户端请求实时价格时，通常需要最新的价格数据。
    - 从 NASDAQ API 获取实时数据可以确保最低的延迟，而快照可能会有一定的时间间隔（例如每分钟生成一次）。

2. **快照的用途**
    - 快照是定期生成的，用于存储历史数据（如开盘价、收盘价、最高价、最低价等）。
    - 快照不适合用于实时价格查询，因为它可能不是最新的。

数据来源的区别

1. **实时价格**
    - 来源：直接从 NASDAQ API 的实时数据流中获取。
    - 使用场景：客户端请求最新的股票价格，或通过 WebSocket 推送实时更新。
2. **历史价格**
    - 来源：从快照中生成并存储到历史数据库。
    - 使用场景：客户端请求某个时间范围内的历史价格数据。

Data Ingestion Service (Snapshots) 的主要职责是从实时数据流中生成快照并存储到 **Historical Database**，因此它的作用范围应该只在 **实时数据流** 和 **数据库** 之间，而不需要与缓存或实时价格服务直接交互。

1. **实时价格**
    - **Live Price Service** 直接从 **Live Data Streams（NASDAQ 实时数据流）** 获取最新的股票价格。
    - 实时价格被存储到 **Cache Layer（Redis/Memcached）**，以便快速响应客户端请求。
    - 客户端通过 API 请求实时价格时，数据直接从缓存或实时数据流中获取。
2. **历史数据**

- **Data Ingestion Service** 订阅 **Live Data Streams**，定期生成快照（例如每分钟）。
- 快照数据包括开盘价、收盘价、最高价、最低价和成交量。
- 生成的快照被存储到 **Historical Database** 中。
- 客户端通过 API 请求历史价格时，数据从 **Historical Database** 中查询。

Live Data Streams 通常使用 **消息队列** 或类似 **WebSocket 的协议** 来传输实时数据。这种设计可以确保数据的实时性、可靠性和高效性。以下是常见的实现方式：

1. 使用消息队列

消息队列 是一种常见的实时数据传输机制，适用于高吞吐量和可靠性要求的场景。

- **常见协议和工具**：
  - **Kafka**：分布式消息队列，支持高吞吐量和分区消费。
  - **RabbitMQ**：支持多种消息传输协议（如 AMQP）。
  - **Redis Pub/Sub**：轻量级的发布/订阅机制，适合小规模实时数据流。
  - **AWS Kinesis**：云端流处理服务，适合大规模实时数据流。
- **优点**：
  - 支持高并发和高吞吐量。
  - 消息可以被多个消费者订阅，支持扩展性。
  - 提供消息持久化和重试机制，确保数据可靠性。
- **适用场景**：
  - NASDAQ 将实时数据流推送到消息队列。
  - 消费者（如实时价格服务和数据摄取服务）订阅消息队列，处理实时数据。

2. 使用 WebSocket 或类似协议

WebSocket 是一种全双工通信协议，适用于低延迟的实时数据传输。

- **特点**：
    - 建立后保持长连接，支持服务器主动推送数据。
    - 数据传输延迟低，适合实时性要求高的场景。
- **优点**：
    - 实现简单，客户端直接通过 WebSocket 连接 NASDAQ 的数据流。
    - 数据传输效率高，适合实时价格更新。
- **缺点**：
    - 不支持消息持久化，断开连接后可能丢失数据。
    - 不适合高并发场景（需要额外的负载均衡机制）。
- **适用场景**：
    - NASDAQ 提供 WebSocket 接口，实时价格服务直接订阅 WebSocket 数据流。
    - 客户端通过 WebSocket 接收实时更新。

3. 混合使用

在实际系统中，可能会混合使用消息队列和 WebSocket：

- **消息队列**：
    - NASDAQ 将实时数据推送到消息队列（如 Kafka）。
    - 后端服务（如 Data Ingestion Service 和 Live Price Service）订阅消息队列，处理实时数据。
- **WebSocket**：
    - 后端服务通过 WebSocket 将实时数据推送给客户端。
    - 客户端订阅 WebSocket，接收实时价格更新。

NASDAQ Live Stream 能将实时数据推送到消息队列，通常是因为 NASDAQ 提供了一个 **实时数据流 API**，而开发者可以通过该 API 将数据消费后推送到消息队列中。这并不是 NASDAQ 自行推送到消息队列，而是系统架构设计者通过以下方式实现的：

实现方式

1. **NASDAQ 提供实时数据流 API**
    - NASDAQ 通常通过以下协议提供实时数据：
        - **WebSocket**：提供实时推送的双向通信。
        - **TCP/UDP**：提供高吞吐量的低延迟数据流。
        - **REST API**：提供轮询方式获取最新数据（不适合高频实时场景）。
    - 开发者需要订阅 NASDAQ 的数据服务，获取实时数据流。
2. **数据消费服务**
    - 系统中会有一个专门的服务（例如 **Data Stream Consumer**），负责从 NASDAQ 的实时数据流中消费数据。
    - 该服务会解析 NASDAQ 提供的数据格式（如 JSON、Protobuf 等），并将其转换为内部可用的格式。
3. **推送到消息队列**
    - 消费服务将解析后的数据推送到消息队列（如 Kafka、RabbitMQ、Redis Pub/Sub）。
    - 消息队列的作用是解耦数据生产者（NASDAQ 数据流）和消费者（实时价格服务、数据摄取服务等），并提供可靠性和扩展性。

NASDAQ 实时数据流 API 与消息队列的逻辑

1. **NASDAQ 实时数据流 API 的行为**

- NASDAQ 提供的实时数据流 API（例如 WebSocket 或 TCP 流）会在 **数据有更新时** 主动推送最新的数据。
- 这些更新可能包括：
  - 股票价格的变化（如买入价、卖出价、最新成交价）。
  - 成交量的变化。
  - 市场状态的变化（如暂停交易、恢复交易等）。
- 数据是 **事件驱动** 的，只有在有更新时才会推送，而不是定期推送。

2. **同步到消息队列的逻辑**
   - 系统中会有一个 **数据消费服务（Data Stream Consumer Service）**，订阅 NASDAQ 的实时数据流。
   - 当 NASDAQ 推送新的数据时，消费服务会立即接收并处理这些数据。
   - 处理后的数据会被 **实时推送到消息队列**，而不是定期同步。
   - 消息队列会将这些实时数据存储并分发给其他服务（如实时价格服务和数据摄取服务）。

1. NASDAQ 实时数据流 API 推送更新数据（事件驱动）。
2. 数据消费服务接收数据，解析并处理。
3. 数据消费服务将处理后的数据推送到消息队列。
4. 消息队列分发数据给订阅的服务（如实时价格服务和历史数据存储服务）。

为什么不是定期同步？

实时性要求：

股票市场的数据变化非常频繁，定期同步可能导致延迟，无法满足实时性需求。

事件驱动的推送机制可以确保数据在变化时立即被处理。

数据流量优化：

如果采用定期同步（例如每秒同步一次），即使数据没有变化也会发送冗余数据，浪费带宽和计算资源。

事件驱动的机制只在数据变化时推送，减少了不必要的流量。

Data Stream Consumer Service 需要主动 **订阅、监听和消费 NASDAQ 的实时数据流 API**，才能接收数据更新并处理。这是整个数据流工作的前提。以下是详细说明：

1. 订阅 NASDAQ 实时数据流 API

- **订阅机制**：
    - NASDAQ 通常要求开发者通过认证（如 API 密钥）订阅其实时数据流服务。
    - 订阅后，开发者可以通过指定的协议（如 WebSocket、TCP、或专用 SDK）连接到 NASDAQ 的数据流端点。
- **订阅的内容**：
    - 开发者可以选择订阅特定的股票（如 AAPL、GOOGL）或整个市场的数据流。
    - 订阅时可能需要指定过滤条件，例如：
        - 只接收某些股票的更新。
        - 只接收特定类型的数据（如价格更新、成交量变化等）。

2. 监听 NASDAQ 数据流

- **监听机制**：

    - 一旦订阅成功，**Data Stream Consumer Service** 会保持与 NASDAQ 数据流的长连接（如 WebSocket）。
    - 数据流是事件驱动的，只有在数据有更新时，NASDAQ 才会主动推送数据。
- **监听的实现**：
    - 使用 WebSocket 客户端或其他协议的客户端库，持续监听 NASDAQ 的数据流。

3. 消费 NASDAQ 数据流

- **消费机制**：
    - 当 NASDAQ 推送数据时，**Data Stream Consumer Service** 会立即接收并处理这些数据。
    - 数据通常以 JSON、Protobuf 或其他格式传输，消费服务需要解析这些数据。
- **处理后的操作**：
    - 将解析后的数据推送到消息队列（如 Kafka、RabbitMQ）。
    - 或者直接存储到缓存（如 Redis）或数据库中。

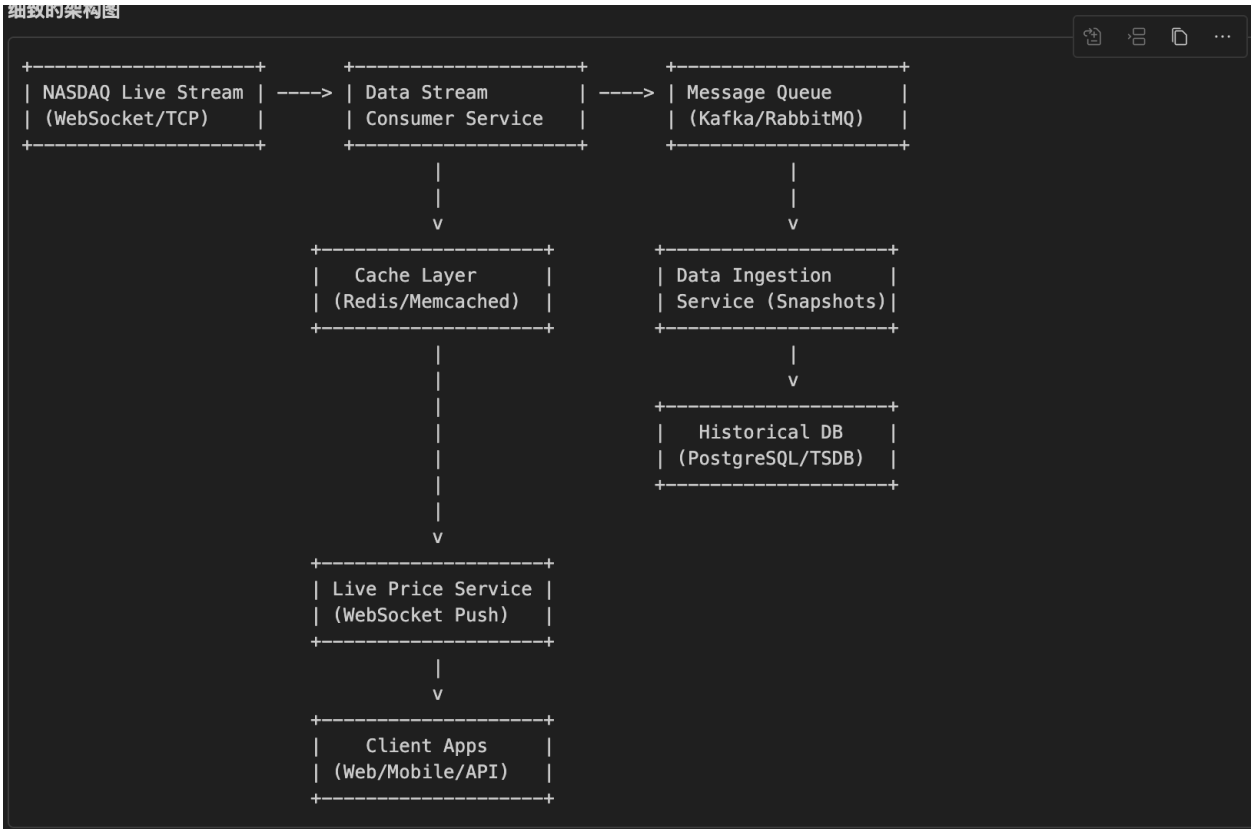
4. 关键点

- **长连接**：
    - Data Stream Consumer Service 必须保持与 NASDAQ 数据流的长连接（如 WebSocket 或 TCP）。
    - 如果连接断开，需要实现自动重连机制。
- **订阅管理**：

    - 如果需要动态调整订阅的股票（如新增或移除某些股票），可以通过 API 调用或发送订阅消息实现。
- **数据可靠性**：
    - 如果 NASDAQ 数据流支持重放（Replay）功能，消费服务可以在断开连接后请求丢失的数据。
    - 如果不支持重放，消费服务需要确保消息队列或缓存的高可用性，避免数据丢失。

总结

- **Data Stream Consumer Service** 必须主动订阅 NASDAQ 的实时数据流 API。
- 它需要持续监听数据流，并在有更新时消费数据。
- 消费后的数据可以推送到消息队列、缓存或数据库，供其他服务使用。



架构说明

1. NASDAQ Live Stream

- **数据来源**:
    - NASDAQ 提供的实时数据流 API（如 WebSocket 或 TCP）。
    - 包含股票的最新价格、成交量、买卖盘等信息。
- **推送机制**:
    - 数据是事件驱动的，只有在数据有更新时才会推送。

2. Data Stream Consumer Service

- **职责**:
    - 订阅 NASDAQ 的实时数据流，监听并消费数据。

    - 解析 NASDAQ 推送的数据（如 JSON 或 Protobuf 格式）。
    - 将解析后的数据推送到消息队列。
- **关键功能**：
    - **订阅管理**：支持动态订阅特定股票或市场数据。
    - **自动重连**：在连接断开时自动重连 NASDAQ 数据流。
    - **数据处理**：对数据进行清洗、格式化和校验。

3. Message Queue

- **职责**：
    - 解耦 NASDAQ 数据流与后端服务。
    - 提供可靠的消息传递机制，支持多消费者订阅。
- **技术选型**：
    - **Kafka**：适合高吞吐量和分区消费。
    - **RabbitMQ**：支持灵活的路由和多种协议。
    - **Redis Pub/Sub**：轻量级的发布/订阅机制。

4. Cache Layer

- **职责**：
    - 存储最新的实时价格数据，供 **Live Price Service** 快速查询。
    - 减少对消息队列的直接依赖，提升查询性能。
- **技术选型**：
    - **Redis**：支持高性能的键值存储。
    - **Memcached**：适合简单的缓存需求。

5. Data Ingestion Service

- **职责**:
    - 从消息队列中消费实时数据，定期生成快照。
    - 快照包括开盘价、收盘价、最高价、最低价和成交量。
    - 将快照存储到 **Historical Database**。
- **关键功能**:
    - **定期任务**：每分钟或每小时生成一次快照。
    - **数据聚合**：对实时数据进行聚合，生成历史记录。

6. Historical Database

- **职责**:
    - 存储历史价格数据，支持时间范围查询。
    - 数据来源于 **Data Ingestion Service** 的快照。
- **技术选型**:
    - **PostgreSQL + TimescaleDB**：支持时间序列数据的高效存储和查询。
    - **InfluxDB**：专为时间序列数据设计的数据库。

7. Live Price Service

- **职责**:
    - 提供实时价格查询和推送功能。
    - 从缓存中获取最新的实时价格。
    - 通过 WebSocket 将实时价格推送给客户端。
- **关键功能**:
    - **实时查询**：响应客户端的实时价格请求。

- **实时推送**：通过 WebSocket 主动推送价格更新。

8. Client Apps

- **职责**：
  - 提供用户界面，展示实时价格和历史价格。
  - 通过 API 或 WebSocket 与后端服务交互。
- **支持的客户端**：
  - Web 应用、移动应用、第三方 API 客户端。

数据流过程

1. **实时数据流**：
   - NASDAQ 推送实时数据到 **Data Stream Consumer Service**。
   - Consumer Service 解析数据并推送到 **Message Queue**。
2. **实时价格服务**：
   - **Live Price Service** 从 **Cache Layer** 获取最新价格。
   - 客户端通过 API 或 WebSocket 获取实时价格。
3. **历史数据生成**：
   - **Data Ingestion Service** 从 **Message Queue** 消费数据，定期生成快照。
   - 快照存储到 **Historical Database**，供历史价格查询。
4. **客户端交互**：
   - 客户端通过 API 查询实时价格或历史价格。
   - 实时价格通过 WebSocket 推送，历史价格从数据库查询。

Live Price Service 不需要从消息队列（queue）中获取数据，而是直接从 **缓存层（Cache Layer）** 或 **NASDAQ 实时数据流 API** 中获取最新的实时数据。这是因为实时价格服务的核心需求是 **低延迟**，而消息队列通常用于解耦和异步处理，可能会引入额外的延迟。

为什么 Live Price Service 不需要从 Queue 获取数据？

1. **实时性要求**
    - 实时价格需要尽可能低的延迟，直接从缓存或 NASDAQ 数据流中获取数据可以确保实时性。
    - 消息队列通常用于异步处理（如生成快照），不适合实时性要求高的场景。
2. **数据流的职责分离**
    - 消息队列的主要职责是解耦和分发数据，供其他服务（如 Data Ingestion Service）使用。
    - Live Price Service 的职责是提供最新的实时价格，直接从缓存或 NASDAQ 数据流中获取即可。
3. **缓存的作用**
    - 缓存层（如 Redis）存储了最新的实时价格，Live Price Service 可以直接从缓存中读取，避免重复处理数据流。

Live Price Service 的数据来源

1. **缓存层（Cache Layer）**
    - NASDAQ 数据流的最新数据会被存储到缓存中。
    - Live Price Service 从缓存中读取最新的价格数据，响应客户端请求。
2. **直接从 NASDAQ 数据流**
    - 如果缓存中没有数据（如缓存未命中），Live Price Service 可以直接从 NASDAQ 数据流中获取最新数据。

消息队列（Message Queue） 在这个架构中主要用于 **Data Ingestion Service** 的数据处理，具体来说是用来为 **创建快照（Snapshots）** 提供数据支持。以下是更详细的说明：

消息队列的职责

1. **解耦实时数据流与数据摄取服务**
    - NASDAQ 的实时数据流直接推送到 **Data Stream Consumer Service**。
    - **Data Stream Consumer Service** 将解析后的数据推送到消息队列。
    - **Data Ingestion Service** 从消息队列中消费数据，生成快照并存储到历史数据库。
2. **异步处理**
    - 消息队列允许 **Data Ingestion Service** 异步处理数据，而不需要直接依赖 NASDAQ 数据流的实时性。
    - 即使 NASDAQ 数据流的速度很快，消息队列可以缓冲数据，确保 **Data Ingestion Service** 不会因为处理速度不足而丢失数据。
3. **可靠性**
    - 消息队列提供持久化和重试机制，确保数据不会因为网络中断或服务故障而丢失。
    - 这对历史数据的完整性非常重要。
4. **扩展性**
    - 如果需要扩展 **Data Ingestion Service**（例如增加多个实例），消息队列可以支持多个消费者同时处理数据。

消息队列的作用范围

- **仅用于 Data Ingestion Service**：
    - 消息队列的唯一消费者是 **Data Ingestion Service**。
    - 它从队列中消费数据，定期生成快照并存储到历史数据库。

- **不用于实时价格服务**:
    - 实时价格服务直接从缓存或 NASDAQ 数据流中获取数据，不依赖消息队列。

消息队列（Message Queue） 在这个架构中主要用于 **Data Ingestion Service** 的数据处理，具体来说是用来为 **创建快照（Snapshots）** 提供数据支持。以下是更详细的说明:

消息队列的职责

1. **解耦实时数据流与数据摄取服务**
    - NASDAQ 的实时数据流直接推送到 **Data Stream Consumer Service**。
    - **Data Stream Consumer Service** 将解析后的数据推送到消息队列。
    - **Data Ingestion Service** 从消息队列中消费数据，生成快照并存储到历史数据库。
2. **异步处理**
    - 消息队列允许 **Data Ingestion Service** 异步处理数据，而不需要直接依赖 NASDAQ 数据流的实时性。
    - 即使 NASDAQ 数据流的速度很快，消息队列可以缓冲数据，确保 **Data Ingestion Service** 不会因为处理速度不足而丢失数据。
3. **可靠性**
    - 消息队列提供持久化和重试机制，确保数据不会因为网络中断或服务故障而丢失。
    - 这对历史数据的完整性非常重要。
4. **扩展性**
    - 如果需要扩展 **Data Ingestion Service**（例如增加多个实例），消息队列可以支持多个消费者同时处理数据。

消息队列的作用范围

- **仅用于 Data Ingestion Service**:
    - 消息队列的唯一消费者是 **Data Ingestion Service**。
    - 它从队列中消费数据，定期生成快照并存储到历史数据库。
- **不用于实时价格服务**:
    - 实时价格服务直接从缓存或 NASDAQ 数据流中获取数据，不依赖消息队列。

数据流过程

1. **实时数据流**
    - NASDAQ 推送实时数据到 **Data Stream Consumer Service**。
    - Consumer Service 解析数据并推送到 **Message Queue**。
2. **消息队列**
    - 缓冲 NASDAQ 数据流，确保数据可靠传递。
    - 提供异步处理能力，供 **Data Ingestion Service** 消费。
3. **Data Ingestion Service**
    - 从消息队列中消费数据，定期生成快照。
    - 快照包括开盘价、收盘价、最高价、最低价和成交量。
    - 将快照存储到 **Historical Database**。
4. **历史数据库**
    - 存储快照数据，供历史价格查询使用。

总结

- **消息队列** 的唯一作用是为 **Data Ingestion Service** 提供数据支持，用于生成历史快照。

- **实时价格服务** 不依赖消息队列，而是直接从缓存或 NASDAQ 数据流中获取数据。
- 这种设计确保了实时性和历史数据处理的解耦，同时提高了系统的可靠性和扩展性。

Data Stream Consumer Service 和 **Message Queue** 之间的交互通常是通过 **主动推送（Publish）** 的方式，而不是定期推送。这种交互模式是事件驱动的，基于监听和消费机制，而不是定时任务。以下是详细说明:

1. Data Stream Consumer Service 和 Message Queue 的交互机制

事件驱动的主动推送

- **Data Stream Consumer Service** 监听 NASDAQ 的实时数据流（通过 WebSocket、TCP 等协议）。
- 当 NASDAQ 数据流有更新时，Consumer Service 会立即接收数据并处理。
- 处理后的数据会被 **主动推送（Publish）** 到消息队列中，而不是定期推送。

为什么是主动推送?

- **实时性**：数据流是事件驱动的，只有在有更新时才会触发推送。主动推送可以确保数据尽快进入队列，减少延迟。
- **避免冗余**：定期推送可能会导致重复数据或无效数据的传输，而事件驱动的主动推送只在有更新时发送数据。

2. 消息队列的工作机制

发布/订阅模式（Publish/Subscribe）

- **Data Stream Consumer Service** 是消息队列的生产者（Producer）。
- 它将处理后的数据发布到消息队列的特定主题（Topic）或队列中。
- **Data Ingestion Service** 是消息队列的消费者（Consumer），订阅特定的主题或队列，并从中消费数据。

消息队列的特点

- **异步处理**：生产者和消费者之间是解耦的，生产者不需要等待消费者处理完成。
- **可靠性**：消息队列通常支持消息持久化和重试机制，确保数据不会丢失。
- **扩展性**：多个消费者可以同时订阅同一个主题，支持水平扩展。

1. NASDAQ 数据流有更新时，推送数据到 Data Stream Consumer Service。

2. Data Stream Consumer Service 处理数据后，主动推送到消息队列。

3. 消息队列将数据存储并等待消费者（如 Data Ingestion Service）消费。

4. Data Ingestion Service 从消息队列中消费数据，生成快照并存储到历史数据库。

Data Ingestion Service 是基于 **消费（consume）和监听（subscribe）消息队列** 来进行快照更新的。这种设计是标准的 **发布-订阅模式（Publish-Subscribe Pattern）**，消息队列在这里起到了解耦和缓冲的作用。

Data Ingestion Service 的工作机制

1. **订阅消息队列**
    - **Data Ingestion Service** 作为消息队列的消费者（Consumer），订阅特定的主题（Topic）或队列。
    - 例如，订阅主题 nasdaq-data，接收 NASDAQ 实时数据流的更新。
2. **监听消息队列**
    - 服务会持续监听消息队列，当有新消息到达时立即消费。
    - 消息队列会将 NASDAQ 数据流的更新推送给 Data Ingestion Service。
3. **处理消息**
    - 消费到的数据会被解析和处理。

- Data Ingestion Service 会根据这些数据生成快照（Snapshots），包括开盘价、收盘价、最高价、最低价和成交量等。

4. **存储快照**
    - 生成的快照会被存储到 **Historical Database**，供历史价格查询使用。

进行快照更新 通常是直接将生成的快照数据写入到数据库表中（即更新或插入到数据库的表中）。不过，具体的操作方式取决于快照的设计和业务需求。以下是两种常见的方式：

1. 插入新记录（Append-Only 模式）

- **操作方式**：
    - 每次生成快照时，将其作为一条新记录插入到数据库表中。
    - 数据库表会存储所有的快照历史记录。
- **适用场景**：
    - 需要保留所有历史快照数据。
    - 方便进行时间范围查询（如获取某只股票在过去一小时的价格变化）。
- **优点**：
    - 数据完整性高，所有历史记录都被保留。
    - 查询灵活，可以按时间范围或其他条件查询。
- **缺点**：
    - 数据量会随着时间增长，需要定期归档或清理旧数据。