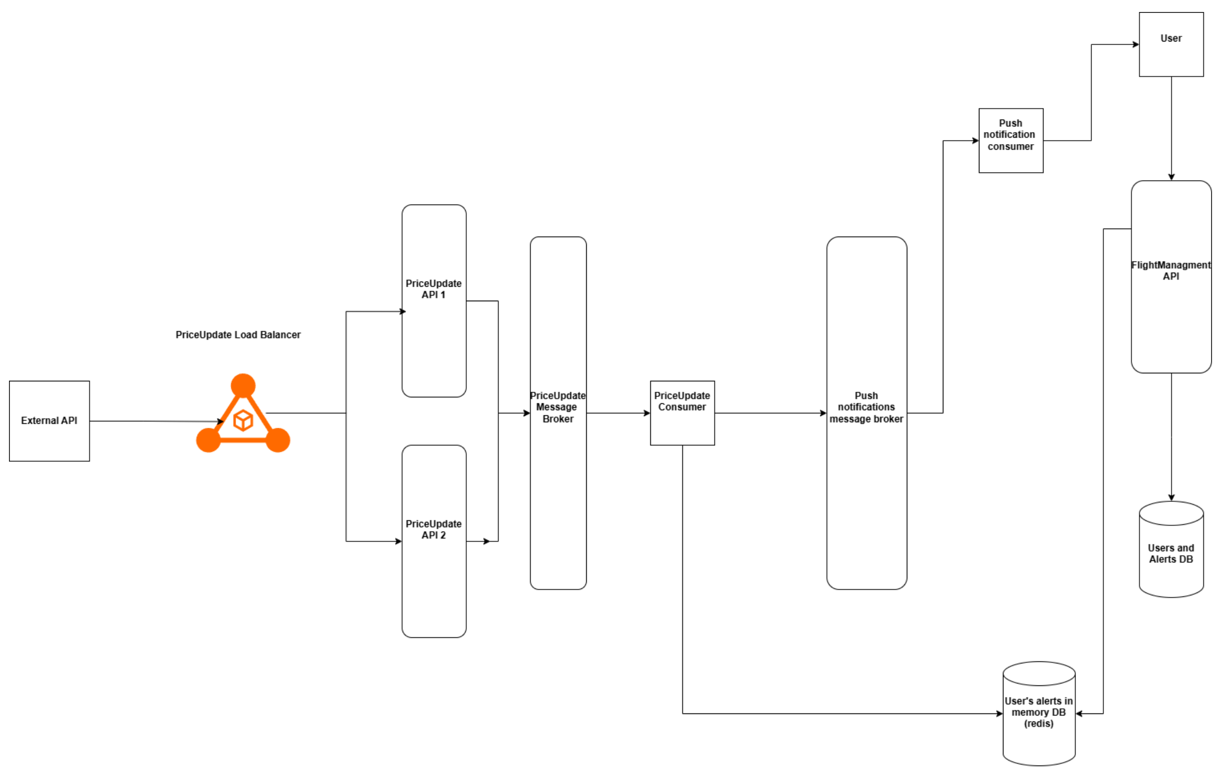
Home Exam - .NET Developer

Architecture Diagram

- Made using Draw.io


External API
PriceUpdate Load Balancer
PriceUpdate API 1
PriceUpdate API 2
PriceUpdate Message Broker
PriceUpdate Consumer
Push notifications message broker
Push notification consumer
User
FlightManagment API
Users and Alerts DB
User's alerts in memory DB (redis)

Data Structures

1. **Load balancer**
    a. The load balancer primarily manages network connections and does not maintain complex data structures related to the application data.
    b. It uses internal tables to keep track of active sessions and routing information.
    c. It will route the request to a certain instance of API service based on the rules (Http rules) using Kubernetes.

2. **API Service (Deduplication & Publishing)**
    a. Redis or any in-memory storage (to check for duplication):
        i. Utilized to store unique identifiers of recently processed price updates.
    b. Queue - (Message Broker like Azure Service Bus, Kafka, RabbitMQ):
        i. To manage incoming price updates requestions before processing them, ensuring that they go in order.
3. **Price Update Message Broker**
    a. Topics**:**
        i. Serve as logical channels to which messages are published and from which consumers subscribe.
    b. Partitions:
        i. Each topic is divided into partitions to allow for parallel processing and scalability.
    c. Message Queues:
        i. Within partitions, messages are stored in a log-structured format, ensuring ordered delivery.
4. **Price Update Consumer Service**
    a. Subscription Lists:
        i. Maintain lists or sets of active subscriptions to various topics or channels.
    b. Processing Queues:
        i. Manage tasks or messages that are pending processing.
5. **Push Notification Message Broker**
    a. Topics:
        i. Serve as logical channels to which messages are published and from which consumers subscribe.
    b. Partitions:
        i. Each topic is divided into partitions to allow for parallel processing and scalability.

   c. Message Queues:
      i. Within partitions, messages are stored in a log-structured format, ensuring ordered delivery.

6. **Push Notification Consumer Service**
   a. Delivery Status Tables:
      i. Track the status of sent notifications, including successes, failures, and retries.

7. **Users and alerts Database**
   a. Relational Tables or NoSQL Collections:
      i. Depending on the database type, used to store user profiles, alert configurations, and preferences.
   b. Indexes:
      i. Implemented to expedite query performance on frequently searched fields, such as user IDs or alert criteria.

Data Flow

1. **External API:**
   a. An external service sends a price update to my Load Balancer.
2. **Load Balancer**:
   a. The Load Balancer distributes the incoming request to one of the available instances of your API Service, ensuring optimal resource utilization and availability. (This will be made using Kubernetes).
3. **API Service (Deduplication & Publishing):**
   a. When receiving the price update, the API Service will do the following:
      i. Duplication Check:
         1. It will check if the in the in-memory storage (**redis**) if the price update already exists in the database.
         2. If it exists in the cache, the message will be discarded.
         3. If it doesn’t exist, the message will be saved in the cache with a short expiration time and it will be published to the price update message broker on the relevant topic (**flights.Israel.France**).
4. **Price Update Message Broker**:

   a. The Message Broker receives the published price update and ensures it is delivered to all subscribed Consumer Services.

5. **Price Update Consumer Service:**
   a. Upon receiving the price update message from the Message Broker, the Consumer Service performs the following:
      i. Queries the In-Memory Cache to retrieve user alert subscriptions that match the flight route (e.g., Israel to France).
      ii. For each user with a matching route alert, checks if the new price meets or is below the user's specified price threshold.
      iii. For users whose alert criteria are met, sends a message to the Push notifications Message Broker.
6. **Users Alert in-memory Database (Redis):**
   a. The In-Memory Cache stores active user alert subscriptions for quick access, reducing the need for frequent database queries
7. **Push Notification Message Broker:**
   a. The Message Broker receives the notification and ensures it is delivered to all the subscriber consumer services.
8. **Push Notification – Consumer Service:**
   a. Upon receiving the notification from the Message Broker, the consumer service will send the notification to the user’s phone.

The data structure and flow will ensure us:

1. **Scalability:** The Load Balancer and API Service are designed to handle high volumes of incoming price updates efficiently.
2. **Deduplication:** Implementing deduplication at the API Service level ensures that duplicate messages are filtered out early, conserving resources downstream.
3. **Performance:** Utilizing an In-Memory Cache for both deduplication and user alerts ensures rapid data access and processing, facilitating timely notifications to users.