People Tech – Assignment Week_3

Swetha Kare

Core Components of System Design

Architectural Patterns:

Monolithic Architecture: A single-tiered architecture where all functions of an application are tightly integrated.

Microservices Architecture: A design where a system is composed of loosely coupled services that communicate over a network, improving scalability and fault isolation.

Service-Oriented Architecture (SOA): Focuses on reusing services that are components within the system, allowing for scalability and distributed computing.

Scalability:

Vertical Scaling (Scaling Up): Increasing the capacity of a single server by adding more resources like CPU, memory, etc.

Horizontal Scaling (Scaling Out): Adding more servers or instances to handle increased load, typically using load balancers.

Auto-scaling: Dynamically adjusts the number of servers based on current traffic/load.

Load Balancing:

Round-Robin: Distributes requests to each server in turn.

Least Connections: Sends requests to the server with the fewest active connections.

IP Hash: Distributes requests based on the client's IP address.

Improves availability and ensures that no single server becomes overwhelmed.

Data Storage:

Relational Databases (RDBMS): Structured data, consistency, and ACID properties. (e.g., MySQL, PostgreSQL)

NoSQL Databases: Scalable, flexible schema, supports large amounts of unstructured data. (e.g., MongoDB, Cassandra, DynamoDB)

Sharding: Dividing a database into smaller, more manageable pieces across multiple servers to increase read/write performance.

Replication: Copying data to multiple servers for availability and fault tolerance.

Caching:

In-memory Caching: Stores frequently accessed data in memory to reduce latency and improve performance (e.g., Redis, Memcached).

Cache Invalidation: Ensures that stale data is removed or updated in the cache to maintain consistency.

Message Queuing and Streaming:

Message Queues: Asynchronous communication between services via message brokers (e.g., RabbitMQ, SQS).

Streaming Platforms: Real-time data processing through stream processing frameworks (e.g., Apache Kafka, AWS Kinesis).

APIs:

RESTful API Design: Stateless, resource-based endpoints, communicating over HTTP.

GraphQL: Query language that allows clients to specify exactly what data they need.

gRPC: High-performance RPC framework that uses protocol buffers for data serialization.

Security Considerations:

Authentication & Authorization: Implement OAuth2, JWT, or SSO (Single Sign-On) mechanisms.

Data Encryption: Encrypt data at rest (e.g., with AES) and in transit (e.g., with TLS).

Rate Limiting: Protects the system from abuse by restricting the number of requests a user can make.

Audit Logging: Keep detailed logs for system events, security incidents, and performance metrics.

Design Considerations - Building a Notification Service, Key requirements and Tradeoffs for building a system.

High Availability (HA):

Redundancy: Ensure there are no single points of failure by having multiple instances of critical services.

Failover Mechanisms: Automatic switchover to a standby server in case of failure.

Performance:

Latency: Time taken to process a request. Reduced through caching, CDN, or optimizations in database queries.

Throughput: The number of requests the system can handle per second. Scaled using horizontal scaling or load balancing.

Consistency vs Availability (CAP Theorem):

Consistency: Ensures all clients see the same data at the same time.

Availability: Guarantees that requests are processed even in case of system failures.

Partition Tolerance: The system continues to operate even if a network partition occurs.

Monitoring and Observability:

Monitoring: Use tools like Prometheus, Grafana, or ELK stack to track system health and performance metrics.

Logging: Keep structured logs for debugging, audits, and tracking user behavior.

Alerting: Set thresholds and triggers for performance metrics (e.g., CPU usage, request latency) to notify system operators of potential issues.

Disaster Recovery:

Backup and Restore: Regularly back up databases and critical data to ensure data can be recovered in case of a failure.

Geographic Distribution: Store backups or deploy services in multiple geographic locations to ensure availability in case of a regional failure.

Key Trade-offs in System Design

Consistency vs. Availability:

Systems like NoSQL databases often need to balance between strict data consistency and high availability. For example, eventual consistency might be chosen for a highly available system.

Latency vs. Throughput:

Increasing throughput might introduce higher latency if the system is under heavy load. Trade-offs need to be considered based on the nature of the application (e.g., real-time systems prioritize low latency).

Complexity vs. Simplicity:

Microservices offer greater scalability and fault tolerance but come with added complexity for service discovery, data consistency, and inter-service communication.

A monolithic design is simpler to manage, but scaling and fault isolation become harder as the system grows.

Designing a **Notification Service System** involves creating an architecture that can handle the delivery of notifications to users across multiple channels (e.g., email, SMS, push notifications) in a scalable, reliable, and efficient manner.

Here's how you can design a notification service system, breaking it down into key components, considerations, and flow.

Key Requirements
Multiple Notification Channels: The system should support different channels like:

- Email
- SMS
- Push notifications
- In-app notifications

- **Scalability**: The system should handle a large volume of notifications, especially during peak times.
- **Reliability**: Ensure that notifications are sent reliably, with retry mechanisms in case of failure.

- **User Preferences**: Users should be able to configure how and when they receive notifications (e.g., only via email, no push notifications after 10 PM).
- **Prioritization and Rate Limiting**: Notifications can have different levels of importance, and rate limiting should prevent system overload.
- **Observability**: The system should provide logs, monitoring, and error reporting to track sent notifications

System Components
Notification Producer (Event Source):

- This could be any service within your application that generates events triggering notifications, such as:
- **Order Confirmation** after a purchase
- **Password Reset Requests**
- **Marketing Campaign Messages**
- **System Alerts**
- Each event should contain metadata such as user details, message content, preferred channels, and notification type.
- **Event Queue (Message Broker):**
- Use a message broker like **Kafka**, **RabbitMQ**, or **AWS SQS** to decouple the event producer from the notification processor. This ensures:
- **Asynchronous processing**: Notifications can be queued and processed later, allowing the system to scale.
- **Failure isolation**: If the notification service is down, the event producer won't be affected.
- **Durability**: Events are stored safely until consumed by a worker.
- **Notification Processor (Dispatcher Service):**
- This service processes events from the queue and determines:
- The **recipient** of the notification.
- The **type of notification** (email, SMS, push).
- **User preferences** for delivery (which channel to use, notification timing restrictions).
- **Content generation**, which could involve templates (e.g., email templates, SMS formats).
- The processor can use a **rule engine** to decide the delivery method based on user preferences, notification importance, and availability of channels.
- **Notification Channel Services:**
- For each type of notification, you will have a separate service or module that handles communication with third-party providers or APIs:
- **Email Service**: Uses services like **SMTP**, **SendGrid**, **Mailgun** to send emails.
- **SMS Service**: Integrates with SMS providers like **Twilio** or **Nexmo**.
- **Push Notification Service**: Sends notifications via services like **Firebase Cloud Messaging (FCM)** or **Apple Push Notification Service (APNs)**.

- **In-App Notification Service**: Updates the user interface or user database to reflect new notifications.
- **Retry and Error Handling:**
- **Dead Letter Queue (DLQ)**: If a message fails to be processed multiple times (e.g., due to a third-party service failure), it should be sent to a DLQ for manual intervention or delayed retries.
- **Exponential Backoff**: Retries can be scheduled with increasing delays to avoid overwhelming downstream systems.
- **Audit Logs**: Maintain logs of failed or delayed notifications for later analysis.
- **User Preferences Management:**
- Store user preferences in a database like **PostgreSQL**, **MongoDB**, or **Redis**.
- Track preferences such as:
  - Preferred notification channels (email, SMS, etc.)
  - Do-not-disturb times or days
  - Notification categories they've subscribed to (marketing vs. transaction alerts)
- The system must check these preferences before sending out notifications.
- **Prioritization and Throttling:**
- High-priority notifications (e.g., security alerts) should be sent immediately, while low-priority ones (e.g., marketing messages) can be delayed or batched.
- Implement **rate limiting** to control the frequency of notifications sent to a user (to prevent spamming) and to avoid overloading external providers.
- **Observability and Monitoring:**
- Use **logging and metrics** to track the status of notifications (e.g., sent, failed, queued).
- Set up dashboards in tools like **Prometheus** and **Grafana** or integrate with **ELK stack** for centralized logging and error detection.
- Implement **alerting** to notify the system operators if a critical service (e.g., SMS provider) is down or underperforming.

- **System Flow**
- **Event Generation**:
  - The application generates an event that needs a notification (e.g., a new purchase).
  - This event is pushed to the **event queue** (Kafka, RabbitMQ).
- **Event Processing**:
  - A worker in the **Notification Processor** consumes the event from the queue.
  - The processor looks up user preferences and determines which notification channels to use.

- The message content is generated (e.g., formatted email, SMS text).

- **Channel Dispatch**:
- The processor sends the message to the respective **channel service** (Email, SMS, Push).
- Each channel service sends the notification through third-party APIs (e.g., Twilio for SMS, SendGrid for email).

- **Error Handling and Retries**:
- If a notification fails (e.g., SMS provider is down), the message is either retried (with exponential backoff) or sent to a **Dead Letter Queue** for later processing.

- **Logging and Monitoring**:
- Notifications are logged, and metrics are generated (e.g., number of emails sent, number of failed SMS deliveries).
- Operators can track system health through a monitoring dashboard.

- ---
- **Technologies to Use**
- **Message Broker**: Kafka, RabbitMQ, AWS SQS
- **Email API**: SendGrid, Mailgun, Amazon SES
- **SMS API**: Twilio, Nexmo
- **Push Notifications**: Firebase Cloud Messaging (FCM), Apple Push Notification Service (APNs)
- **Database**: PostgreSQL (for transactional data), Redis (for caching user preferences)
- **Monitoring**: Prometheus, Grafana, ELK stack
- **Logging**: Fluentd, Logstash, Elasticsearch

- ---
- **Challenges and Considerations**
- **Message Delivery Guarantees**:
- Ensure messages are delivered reliably, and retry on failure.

- **Scaling**:
- Handle bursts of traffic during peak times, such as marketing campaign notifications.
- Use horizontal scaling for the event queue and notification processors.

- **Rate Limiting**:
- Implement rate limiting to prevent users from being overwhelmed with notifications and to comply with third-party provider limits.

- **Data Consistency**:

- Ensure that user preferences are consistently updated and respected across different parts of the system.

- **Cost Considerations**:
- Be mindful of the costs associated with sending notifications via third-party providers, especially for SMS and email services.