Microservic

1.api gateway in mc

* An API Gateway acts as a middleman between clients and a group of backend services in a microservices setup.
* An API gateway serves as a central entry point for all client requests, acting as a reverse proxy and a facade for the underlying microservices.

It handles various tasks such as request routing, composition, and protocol translation. Here’s a brief overview:

1. **Request Routing**: The API Gateway routes incoming requests to the appropriate microservice based on the request path or other criteria.
2. **Composition**: It can aggregate responses from multiple microservices into a single response, simplifying the client’s interaction with the system.
3. **Protocol Translation**: The gateway can translate between different protocols (e.g., HTTP to WebSocket), allowing microservices to communicate using their preferred protocols.

Additionally, an API Gateway can handle cross-cutting concerns like authentication, authorization, rate limiting, and logging, which helps in maintaining a clean and manageable microservices architecture.

2.Microservices architecture

* A microservices architecture is loosely coupled service **consists of a collection of small, autonomous services**. Each service is self-contained and should implement a single business.
* Microservices break complex applications into smaller, independent services that work together, enhancing scalability, and maintenance.

**What are the main components of Microservices Architecture?**

Main components of microservices architecture include:

* **Microservices:**Small, loosely coupled services that handle specific business functions, each focusing on a distinct capability.
* [**API Gateway:**](https://www.geeksforgeeks.org/what-is-api-gateway-system-design/)Acts as a central entry point for external clients also they manage requests, authentication and route the requests to the appropriate microservice.
* [**Service Registry and Discovery:**](https://www.geeksforgeeks.org/service-discovery-and-service-registry-in-microservices/) Keeps track of the locations and addresses of all microservices, enabling them to locate and communicate with each other dynamically.
* [**Load Balancer:**](https://www.geeksforgeeks.org/load-balancer-system-design-interview-question/) Distributes incoming traffic across multiple service instances and prevent any of the microservice from being overwhelmed.
* [**Containerization**](https://www.geeksforgeeks.org/containerization-architecture-in-system-design/)**:** Docker encapsulate microservices and their dependencies and orchestration tools like Kubernetes manage their deployment and scaling.
* **Event Bus/**[**Message Broker**](https://www.geeksforgeeks.org/what-are-message-brokers-in-system-design/)**:** Facilitates communication between microservices, allowing pub/sub asynchronous interaction of events between components/microservices.
* [**Database per Microservice**](https://www.geeksforgeeks.org/database-per-service-pattern-for-microservices/)**:** Each microservice usually has its own database, promoting data autonomy and allowing for independent management and scaling.
* [**Caching:**](https://www.geeksforgeeks.org/caching-system-design-concept-for-beginners/) Cache stores frequently accessed data close to the microservice which improved performance by reducing the repetitive queries.
* [**Fault Tolerance**](https://www.geeksforgeeks.org/fault-tolerance-in-system-design/)**and**[**Resilience**](https://www.geeksforgeeks.org/resilient-system-system-design/)**Components:** Components like [circuit breakers](https://www.geeksforgeeks.org/what-is-circuit-breaker-pattern-in-microservices/) and [retry mechanisms](https://www.geeksforgeeks.org/retry-pattern-in-microservices/) ensure that the system can handle failures gracefully, maintaining overall functionality.

3.microservices vs monolithic

**Microservices Architecture**

* **Structure**: Composed of small, independent services, each responsible for a specific business function.
* **Development**: Teams can work on different services simultaneously, using different technologies and languages.
* **Deployment**: Services can be deployed independently, allowing for frequent updates and scaling.
* **Scalability**: Individual services can be scaled based on demand.
* **Resilience**: Failure in one service doesn't necessarily affect others.

**Monolithic Architecture**

* **Structure**: A single, unified codebase where all components are interconnected and interdependent.
* **Development**: Typically uses a single technology stack, making it easier to develop initially.
* **Deployment**: The entire application is deployed as a single unit, which can make updates more challenging.
* **Scalability**: Scaling requires scaling the entire application, even if only one part needs more resources.
* **Resilience**: A failure in one part of the application can affect the entire system.

**When to Use Which?**

* **Microservices**: Best for large, complex applications that require frequent updates, scalability, and flexibility.
* **Monolithic**: Suitable for smaller applications or those with tightly coupled components where simplicity and ease of development are priorities.

| **Aspect** | **Microservices Architecture** | **Monolithic Architecture** |
| --- | --- | --- |
| **Structure** | Small, independent services | Single, unified codebase |
| **Development** | Teams work on different services simultaneously | Typically uses a single technology stack |
| **Deployment** | Services deployed independently | Entire application deployed as a single unit |
| **Scalability** | Individual services scaled based on demand | Scaling requires scaling the entire application |
| **Resilience** | Failure in one service doesn't affect others | Failure in one part can affect the entire system |
| **Complexity** | Managing inter-service communication and data consistency | Simpler initially but complex as the application grows |
| **Technology Diversity** | Different technologies and languages for each service | Single technology stack |
| **Flexibility** | High flexibility for updates and changes | Less flexible for updates and changes |
|  |  |  |

4.Microservices components.

1. **Microservices**: Independent services, each responsible for a specific business function. They communicate with each other through APIs.
2. **API Gateway**: Acts as a single-entry point for client requests, handling routing, composition, and protocol translation.
3. **Service Registry**: Keeps track of all available microservices and their instances, allowing services to discover and communicate with each other.
4. **Service Discovery**: Mechanism that enables microservices to find and connect with each other dynamically.
5. **Load Balancer**: Distributes incoming traffic across multiple service instances to prevent any single service from being overwhelmed.
6. **Configuration Management**: Manages configuration settings for microservices, ensuring consistency and ease of updates.
7. **Monitoring and Logging**: Tools and systems that track the performance and health of microservices, providing insights and alerts for issues.
8. **Security**: Ensures that microservices are protected from unauthorized access and attacks, often involving authentication and authorization mechanisms.
9. **Data Management/**.**Databases**: Handles data storage and retrieval, often using databases that are optimized for specific microservices.

Each service typically manages its database in a microservice architecture, allowing for data autonomy and ensuring that database dependencies do not hinder its operations. This approach, known as database per service, helps isolate the services, improve fault tolerance, and allow the use of different database technologies suited to each service's needs.

1. **Orchestration and Containerization**: Tools like Kubernetes and Docker that manage the deployment, scaling, and operation of microservices in containers.

What is service registery?

A Service Registry serves as a centralized database or directory where information about available services and their locations is stored and maintained. It acts as a vital component of service discovery by providing a central point for service registration, lookup, and management.

load balancer?

A load balancer is crucial for distributing incoming network traffic across multiple instances of a service. This ensures that no single instance becomes overwhelmed, leading to improved resource utilization, enhanced application availability, and reduced latency.

1. Purpose of Load Balancing in Microservices:

* **Distributing Traffic:**

Load balancers forward incoming requests to different instances of a microservice, preventing any one instance from handling all the traffic.

* **Scalability:**

Load balancers enable easy scaling by adding or removing service instances as needed, allowing the application to handle increased traffic demands.

* **High Availability:**

If one instance fails, the load balancer redirects traffic to healthy instances, ensuring continuous service availability.

* **Resource Optimization:**

Load balancing ensures that all available instances are utilized effectively, maximizing resource utilization.

2. How Load Balancing Works:

* Load balancers typically use algorithms to determine which instance should receive a request (e.g., round-robin, least connections, weighted round-robin).
* They also monitor the health of each instance and can automatically remove unhealthy instances from the pool.

3. Types of Load Balancing:

* **Server-side Load Balancing:**

Load balancers are placed in front of the servers, distributing traffic based on various factors like server health, load, or predefined rules.

* **Client-side Load Balancing:**

Clients (e.g., applications) make their own decision on which instance to access, often using a service discovery mechanism like Eureka.

**Examples of Load Balancers**:

* **Amazon Web Services (AWS) Elastic Load Balancer (ELB):** A cloud-based load balancing service.
* **Google Cloud Load Balancing:** Offers various load balancer types for different needs, including Application Load Balancers and Network Load Balancers.
* **Spring Cloud LoadBalancer:** A library for client-side load balancing in Spring Boot applications.
* **Kubernetes Service:** Provides built-in load balancing for containers within a Kubernetes cluster.

**how Microservices communicate?**

Microservices communicate with each other using various protocols and patterns, including synchronous and asynchronous approaches.

**Synchronous** communication involves direct calls, like HTTP or gRPC (short for **g**oogle **Remote Procedure Call**) , where a service waits for a response.

**Asynchronous** communication utilizes message brokers or events, allowing services to communicate without blocking each other.

* This communication can be **synchronous**, where one service waits for a response from another (using methods like RESTful APIs or gRPC), or **asynchronous**, where services send messages without waiting for immediate responses (using message brokers like Kafka or RabbitMQ).

**Types of Inter-Service Communication in Microservices**

Inter-Service Communication in microservices can be broadly categorized into two main types: synchronous and asynchronous communication. Each type has its methods, protocols, and use cases:

**1. Synchronous Communication**

1. **REST (Representational State Transfer):**
   * The most common method, where services communicate over HTTP using standard HTTP methods like GET, POST, PUT, and DELETE.
   * Services wait for a response, which makes it simple but potentially slower, as one service is blocked until the other responds.
2. **gRPC (gRPC Remote Procedure Call):**
   * A high-performance, open-source RPC framework that uses HTTP/2 for transport, Protocol Buffers (Protobuf) for data serialization, and supports multiple programming languages.
   * Faster and more efficient than REST, especially for high-throughput systems, but more complex to implement.

**2. Asynchronous Communication**

1. **Message Queues (e.g., RabbitMQ, Amazon SQS):**
   * Services communicate by sending messages to a queue, where other services can read and process them.
   * Decouples the services, allowing them to operate independently, which can improve performance and reliability.
2. **Event-Driven Architecture (e.g., Apache Kafka, AWS EventBridge):**
   * Services publish events to a message broker, and other services subscribe to these events and react to them.
   * Supports real-time data processing and is well-suited for systems that need to handle high volumes of data or require low latency.

**what is service discovery in microservices:**

**Service Discovery**

* **What it is**: A **process** or **mechanism** that enables services to find and communicate with each other dynamically.
* **Purpose**: Helps services locate other services without hardcoding their network locations.
* **How it works**: It queries a **service registry** to get the address of a service instance.
* **Who uses it**: Service consumers (like other microservices or API gateways).

**Service Registry**

* **What it is**: A **database or directory** that stores the network locations (IP, port) of service instances.
* **Purpose**: Keeps track of all available services and their instances.
* **How it works**: Services register themselves with the registry when they start and deregister when they stop.
* **Who uses it**: Both service providers (to register) and service consumers (to discover).

**🧠 Analogy**

Think of it like a **phone book**:

* The **service registry** is the phone book.
* **Service discovery** is the act of looking up someone’s number in the phone book.

**🔁 How They Work Together**

1. A service starts and **registers** itself with the **service registry**.
2. Another service needs to call it, so it uses **service discovery** to **query the registry**.
3. It gets the address and makes the request.

**Fault Tolerance?**

Fault tolerance in microservices ensures that even when individual services fail, the overall system continues to function, either with minimal disruption.

* **Fault tolerance is the ability of a system**   **to** **continue operating properly even when some of its components fail**.
* In microservices, it's crucial to prevent a failure in one service from cascading and bringing down the entire application.
* **🧱 Key Fault Tolerance Techniques**
* **Retries with Backoff**
* Automatically retry failed requests after a delay.
* Use exponential backoff to avoid overwhelming the system.
* **Circuit Breaker Pattern**
* Prevents a service from repeatedly trying to call a failing service.
* Opens the circuit after a threshold of failures, then retries after a cooldown.
* **Fallbacks**
* Provide a default response or alternative logic when a service fails.
* **Timeouts**
* Set limits on how long to wait for a response to avoid hanging requests.
* Run multiple instances of a service to ensure availability.
* **🔁 Example Scenario**
* Imagine a payment service fails:
* The order service uses a **circuit breaker** to stop calling it temporarily.
* It shows a **fallback message** to the user: “Payment is currently unavailable. Please try again later.”
* Meanwhile, monitoring tools alert the team to fix the issue.

**circuit breaker:**

In microservices architecture, a circuit breaker is a design pattern used to prevent cascading failures when one service fails. It acts as a proxy for a remote service, monitoring its health and switching to a fallback mechanism if the service becomes unavailable or unresponsive.

How it works:

1. **1. Closed State:**

The circuit breaker is initially in a closed state, allowing normal traffic to the remote service.

* Everything is normal.
* Requests flow freely.
* Failures are counted.

1. **2. Open State:**
2. Too many failures occurred.
3. Requests are **blocked immediately** to prevent further strain.
4. After a timeout, it transitions to **half-open**.

If the remote service experiences repeated failures (e.g., timeouts or exceptions), the circuit breaker transitions to the open state. In this state, it immediately returns a fallback response instead of making further requests to the failing service.

1. **3. Half-Open State:**
2. A few test requests are allowed through.
3. If they succeed → back to **closed**.
4. If they fail → back to **open**.

After a specified time or number of failures, the circuit breaker transitions to the half-open state. This allows a limited number of test requests to the remote service to see if it has recovered.

1. **4. Closed State (Resumed):**

If the test requests succeed, the circuit breaker transitions back to the closed state, and normal traffic resumes. If the test requests fail, the circuit breaker returns to the open state.

**what are the rest template methods? and explain?**  
RestTemplate in Spring Framework provides a set of methods for making HTTP requests to RESTful web services. These methods simplify the process of interacting with REST APIs by handling the complexities of HTTP requests and responses. They support various HTTP methods like GET, POST, PUT, DELETE, and PATCH.

Key RestTemplate Methods and their explanations:

getForObject(String url, Class<T> responseType):

Executes a GET request to the given URL and returns the response body as an object of the specified type.

getForEntity(String url, Class<T> responseType):

Executes a GET request and returns an object of type ResponseEntity<T>, which contains both the HTTP status code and the response body as an object of the specified type.

postForObject(String url, Object request, Class<T> responseType):

Executes a POST request to the given URL, sending the specified object as the request body and returning the response body as an object of the specified type.

postForLocation(String url, Object request, Map<String, Object> uriVariables):

Executes a POST request to the given URL, sending the specified object as the request body, and returns the Location header value (which typically indicates the URI of the newly created resource).

put(String url, Object request):

Executes a PUT request to the given URL, sending the specified object as the request body.

delete(String url):

Executes a DELETE request to the given URL.

exchange(HttpMethod method, String url, HttpEntity<?> requestEntity, Class<T> responseType):

Executes a request using the specified HTTP method (GET, POST, PUT, etc.), sending the request entity (which can include headers and a request body) and returning a ResponseEntity<T> containing the HTTP status code and response body.

execute(String url, HttpMethod method, RequestCallback requestCallback, ResponseExtractor<T> responseExtractor):

A more flexible method that allows for custom handling of request and response using RequestCallback and ResponseExtractor.

These methods simplify the process of interacting with RESTful web services in Spring applications by abstracting away the complexities of HTTP requests and responses. They provide a high-level API for making calls to remote services.