**MicroServices**

**1. What is Microservices Architecture**

Microservices Architecture is a way to build an application by splitting it into small, independent services.  
Each service does one specific job, runs separately, and can be developed, deployed, and scaled on its own.

Example:  
Instead of having one big application that handles users, products, and orders together, you split them into three small services:

* User Service
* Product Service
* Order Service

Each service is independent but they work together.

**Database per Service:**

* Each service has its own dedicated database and does not share it with other services.
* This increases the independence of services and makes data management more efficient.

**Example:**

* **User Service** might use an **SQL database**.
* **Product Service** might use a **NoSQL database**.

**2. Benefits vs Monolithic Architecture**

**Benefits of Microservices:**

* **Scalability:**  
  You can scale only the parts of the application that need more resources, without scaling the entire system.
* **Independent Deployment:**  
  Teams can deploy their services separately without waiting for others. This speeds up development and reduces risks.
* **Flexibility in Technology:**  
  Each microservice can use the programming language or database that best fits its needs.
* **Better Fault Isolation:**  
  If one service fails, it doesn’t crash the entire application. Only the affected service needs to be fixed.
* **Faster Development and Innovation:**  
  Small, focused teams can work independently on different services, leading to quicker releases and updates.
* **Easier Maintenance and Understanding:**  
  Small services are easier to understand, modify, and test compared to a large monolithic codebase.

**Challenges of Monolithic Architecture:**

* **Scaling Issues:**  
  You have to scale the whole application even if only one part needs more resources, which is wasteful.
* **Slow Deployment:**  
  A small change requires redeploying the entire application, increasing downtime and risk of errors.
* **Tight Coupling:**  
  All parts of the system are dependent on each other, making it hard to modify or replace components.
* **Technology Lock-In:**  
  You are often forced to use the same technology stack for the entire application.
* **Hard to Understand:**  
  As the application grows, the codebase becomes huge and complex, making it difficult to manage.

**Summary:**  
Microservices offer better scalability, flexibility, and faster development, but add complexity.  
Monolithic architecture is simple at the beginning but becomes hard to manage and scale as the application grows

**3. When (and when NOT) to use Microservices**

**When to Use Microservices:**

* **Large and Complex Systems:**  
  When your application is too big for a single team to manage, splitting it into smaller services makes development easier.
* **Need for Scalability:**  
  If different parts of the system need to scale independently, microservices are ideal.
* **Multiple Teams:**  
  If you have many teams working on different features, microservices allow them to work independently.
* **Different Technology Needs:**  
  When parts of your system would benefit from using different programming languages, databases, or technologies.
* **Frequent Updates:**  
  If you want to release features quickly without affecting the whole system.

**When NOT to Use Microservices:**

* **Small Applications:**  
  If your app is simple and manageable by a small team, microservices will add unnecessary complexity.
* **Lack of Team Experience:**  
  If your team is not familiar with distributed systems, starting with microservices can lead to problems.
* **Tight Deadlines:**  
  Building microservices takes time (design, communication between services, handling failures). If you’re under pressure to deliver fast, it’s better to start with a monolith.
* **Simple Business Needs:**  
  If your business domain is small and clear, a monolithic architecture is easier and faster.

**Summary:**  
Use microservices when the system is big and complex. Avoid them for small, simple projects to keep things easy.

**4. Characteristics of Microservices**

* **Independent Deployability:**  
  Each service can be deployed without affecting the others. You can update one service without redeploying the entire system.
* **Scalability:**  
  Services can be scaled individually. If only one part of your system needs more power, you scale just that service.
* **Decentralized Data Management:**  
  Each microservice manages its own database. Services don’t share databases directly to avoid coupling.
* **Focused and Small Services:**  
  Each service is built around a specific business functionality (e.g., payment service, user service).
* **Resilience:**  
  Microservices handle failures gracefully. If one service fails, others keep working.
* **Communication through APIs:**  
  Services talk to each other over well-defined APIs, usually using HTTP (REST) or messaging systems.
* **Different Technology Stacks:**  
  Each microservice can use the most suitable language, framework, and database for its needs.

**Summary:**  
Microservices are small, independent, and resilient. They manage their own data and communicate through APIs.

**5. How do Microservices Communicate while staying Independent?**

* **Each Service is Fully Independent:**  
  It can be developed, tested, and deployed separately without depending on other services.
* **If a Service Needs Data from Another:**  
   It should NOT access the other service’s database directly.  
   It should NOT call the other service’s code internally.  
   It should communicate **through APIs** — send a request and wait for the response.
* **What if the API URL Changes or the Connection Fails?**  
  That’s why we use:
  + **Service Discovery (like Consul or Eureka):**  
    Services register themselves and find each other dynamically without hardcoding URLs.
  + **API Gateway:**  
    All services are hidden behind a gateway. The client or service talks to the gateway, and the gateway forwards requests to the right service.
  + **Retry Policies & Circuit Breaker:**  
    To handle temporary failures smartly. If a service is down, retries or fallback logic can be applied automatically.

**Security:**

**Without a Gateway:**

* **Authentication & Authorization** needs to be handled by **each individual microservice**.
  + Every microservice will need to verify who the user is (authentication) and check what actions the user is allowed to perform (authorization).
  + This means each service needs its own logic to handle authentication, check roles, and verify permissions.

**With a Gateway:**

* The **API Gateway** takes on the responsibility for **authentication and authorization** for all the microservices.
  + The Gateway checks the user's identity and their permissions first.
  + Once it verifies this, it passes the request to the appropriate microservice, which doesn’t need to worry about security anymore.
  + This makes the security process more centralized and easier to manage because you don't need to repeat the same logic in every service.

In short, **without a Gateway**, each service handles security on its own.  
With a **Gateway**, the responsibility for security is centralized in the Gateway, and it makes the whole system easier to manage.

**Service Communication**

**How microservices talk to each other**.

There are two ways:

* **Synchronous (direct talk)**
* **Asynchronous (indirect talk)**

**1. Synchronous Communication (direct)**

* Example: Like two people talking on the phone — **they wait for each other** to reply.
* In microservices: One service sends a request to another and **waits for a response**.

**Tools used:**

* **HTTP APIs** (REST) → normal web requests (GET, POST, etc.)
* **gRPC** → faster communication with small messages (better for big systems).

**2. Asynchronous Communication (indirect)**

In microservices: One service sends a message, and moves on without waiting.

**Tools used:**

* **RabbitMQ**, **Kafka** → systems that **hold the message** until the other service reads it.

**3. API Gateway Pattern**

In microservices:  
Instead of clients calling every microservice directly, they go through **one point** → the **API Gateway**.

**Popular tools:**

* Ocelot, YARP, etc.

**Benefits:**  
Handles security, logging, and routing  
Hides internal service details from clients

**4. Service Discovery**

* Example: Like a **contact list** on your phone.  
  You don’t memorize numbers — you just search for the name.
* In microservices:  
  Services **register themselves** somewhere, and when another service needs to talk to them, it **looks up** where they are.

**Tools used:**

* **Eureka**, **Consul**

**Why?**  
In microservices, services can move around (change IP addresses), so Service Discovery **automatically finds them**.