### ****Monolithic****

### ****Monolithic Architecture Explained in Simple Words****

A **monolithic architecture** means **everything in the application is built together as one unit**. All features (like login, shopping cart, payment, and order tracking) are **tightly connected** in a single codebase and deployed as one package.

### ****How It Works?****

Imagine a **restaurant** where the kitchen, serving area, and billing counter are all inside one big room. If one thing goes wrong, the whole restaurant gets affected. This is how a monolithic application works—**everything is in one place**.

### ****Problems with Monolithic Architecture****

1. **One Change Affects the Whole App** – If you update one small feature (e.g., change the payment system), the entire app needs to be redeployed.
2. **Hard to Scale** – If only one part (e.g., checkout) gets more traffic, you still have to scale the entire app, which is expensive.
3. **Difficult for Developers** – Many developers working on the same code can cause conflicts, making updates slow.
4. **Becomes Complex Over Time** – As the app grows, it becomes harder to manage, debug, and improve.

### ****When is Monolithic Architecture Good?****

* Best for **small applications** with fewer features.
* Easier to develop **in the beginning** but **hard to scale** later.

### ****Why Move to Micro-services?****

To fix these issues, companies **break the big app into smaller, independent services** (micro-services), so they can update and scale them separately.

### ****Monolithic Architecture (Simple Words)****

* **One big app** where everything is connected.
* **One codebases, one deployment** – even a small change needs the whole app to be updated.
* **Hard to scale** – if one part needs more power, the whole app must be scaled.
* **Developers face issues** – many people working on the same code can create problems.
* **Over time, it becomes slow and difficult to manage.**
* **Good for small apps but not for big, growing applications.**

### ****Micro-services****

### ****Micro-services Architecture Explained in Simple Words****

Micro-services **architecture** breaks a large application into **smaller, independent services** that work together. Each service handles a specific feature (e.g., user management, payments, and orders) and **runs separately**.

### ****How It Works?****

Imagine a **shopping mall** where different stores (clothing, electronics, and food) operate **independently** but together form one big system. If one store changes its prices or closes for a day, others are **not affected**.

### ****Key Features of Micro-services:****

1. **Large apps are divided into small parts:** Instead of one big app, it's split into **small, self-contained services** (e.g., "User Service," "Payment Service").
2. **Different codebase for each service:** Each service has its **own code** and can be developed, tested, and deployed separately.
3. **Each module is managed independently:** Teams can work on different micro-services without waiting for others.
4. **Different tech stacks for each service:** One micro-service can use **Java**, another can use **Python**, and another can use **Node.js**, depending on the needs.
5. **Handling micro services is complex:**
6. More services mean **more communication** between them (via APIs).
7. **Deployment, monitoring, and debugging become harder** because there are many small parts instead of one big system.

### ****Advantages of Micro services:****

* Easier to **scale** (you can scale only the needed service, not the whole app).
* Faster **development and deployment** (each service can be updated independently).
* **More flexibility** in choosing technologies.

### ****Disadvantages of Micro services:****

❌ **Complex to manage** (more services = more infrastructure and monitoring needed).  
❌ **Communication issues** (services must talk to each other via APIs, which can fail).  
❌ **More deployment effort** (each service must be deployed and maintained separately).

### ****Monolithic vs. Micro service Architecture****

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| SN. | |  | | --- | | **Feature** | | |  | | --- | | **Monolithic Architecture** | | |  | | --- | | **Microservice Architecture** | |
| 1 | **Definition** | A single, unified codebase where all components (UI, business logic, and database) are tightly integrated. | A collection of loosely coupled, independently deployable services that communicate over APIs. |
| 2 | **Deployment** | A single deployment unit (e.g., WAR or JAR file). | Each service is deployed independently. |
| 3 | **Scalability** | Vertical scaling (increasing resources of a single server). | Horizontal scaling (scaling individual services based on demand). |
| 4 | **Development Speed** | Faster for small applications but slows down as the codebase grows. | Faster for large applications due to independent teams working on separate services. |
| 5 | **Technology Stack** | Single technology stack across the entire application. | Different services can use different technologies (Polyglot). |
| 6 | **Complexity** | Simple to develop initially but becomes complex as the application grows. | Complex to set up initially but easier to manage at scale. |
| 7 | **Fault Tolerance** | A failure in one module can bring down the entire system. | Failure in one service does not affect others (if designed properly). |
| 8 | **Communication** | Internal method calls between components. | Uses API calls (REST, gRPC, message queues) for inter-service communication. |
| 9 | **Testing & Debugging** | Easier to test as everything is in one place. | More challenging due to multiple services interacting. |
| 10 | **Maintenance** | Harder to maintain as the codebase grows. | Easier to update and maintain individual services. |
| 11 | **Time to Market** | Faster for small applications. | Suitable for long-term scalability and agility. |

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

### ****When to Choose Monolithic?****

* Small to medium-sized applications.
* When quick time-to-market is needed.
* When the team is small and not experienced with micro-services.
* When performance is critical and internal communication overhead should be minimized.

### ****When to Choose Micro-services?****

* Large-scale applications that require independent scaling.
* When different teams work on different parts of the system.
* When services need to be developed using different technologies.
* When frequent updates and deployments are needed.

### ****Conclusion****

Monolithic is simpler to start with but harder to scale, while micro-services are more flexible and scalable but introduce additional complexity in communication, deployment, and management.

This diagram represents typical **micro-services architecture** for a hotel service application.

**1. Client:**

* This represents any user or application that wants to use the hotel service (e.g., a web browser, a mobile app, another service).

**2. API Gateway:**

* This is a single entry point for all client requests. It acts as a reverse proxy, routing requests to the appropriate micro-service.
* It can also handle authentication, authorization, rate limiting, and other cross-cutting concerns.
* In this diagram, it's labeled "OKTA Auth", suggesting that Okta is used for authentication and authorization.

**3. Service Registry:**

* This acts as a directory of all available micro-services.
* When a micro-service starts up, it registers itself with the service registry, providing its name, location, and other relevant information.
* The API Gateway and other micro-services can then query the service registry to discover the location of other services they need to communicate with.
* In the diagram, it houses the following services:
  + **Hotel Service:** Responsible for managing hotel information, such as availability, amenities, and booking. It interacts with DB1 (likely a database for hotel data).
  + **User Service:** Manages user accounts and profiles. It interacts with DB2 (likely a database for user data).
  + **Rating Service:** Handles hotel ratings and reviews. It interacts with DB3 (likely a database for rating data).
  + **Config Service:** Provides configuration information to other micro-services. This allows for centralized configuration management and makes it easier to change settings without restarting services.
  + **Github:** This being inside the "Service Registry" box is unusual. It likely indicates that configurations or code deployments are managed through Gitub, but it's not a typical micro-service residing within the service registry.

**4. Databases (DB1, DB2, and DB3):**

* Each micro-service has its own dedicated database. This is a key principle of micro-services, ensuring isolation and independence between services.
* DB1 stores hotel data, DB2 stores user data, and DB3 stores rating data.

**5. Communication Flow:**

* The diagram shows the flow of a request from the client through the API Gateway to the relevant micro-service(s) and their respective databases.
* For example, a request to book a hotel room would likely go through the API Gateway to the Hotel Service, which would then interact with DB1 to check availability and create the booking.

**Key Concepts Illustrated:**

* **Decoupling:** Micro-services are independent of each other, allowing them to be developed, deployed, and scaled independently.
* **Single Responsibility:** Each micro-service has a specific function.
* **Scalability:** Micro-services can be scaled independently based on their individual needs.
* **Resilience:** If one micro-service fails, it doesn't necessarily bring down the entire application.
* **Technology Diversity:** Different micro-services can be built using different technologies, depending on their requirements.