What is Micro-Service – Collection of many services working together to achieve some common functionality.

In software engineering, a **microservice** architecture is an [architectural pattern](https://en.wikipedia.org/wiki/Architectural_pattern) that organizes an application into a collection of loosely coupled, fine-grained services that communicate through lightweight protocols.

Architectural – Pattern

Loosely – coupled –

Spider Man Toy – Normal (Tightly coupled system)

Lego Spider Man toy – Loosely coupled system.

Design Pattern – It’s a proven way to solve a particular problem/challenge.

Why – Easy to Test, Easy to Build, Autonomous (It’s independent of other services)

How – There are plenty of ways of to create it.

1. Using any progr Lang (Java/Python/Node/.net … )
2. Using different protocols (http)
3. Use different DB (SQL/No-SQL)

What is Monolith application. (Bank)

1. Customer (Service users)
2. Account [ Saving/Current/Deposit/Loan]
3. Transaction

Online shopping Sites

1. Product
2. Users/Customers
3. Order
4. Payments
5. Shipping
6. Rating & Reviews
7. Offers

Monolith application – It’s a single heavy weight tightly couples application offering particular service to customer.

Environment Setup (Java Based)

1. STS – Spring Tool Suite (https://spring.io/tools ) / VS Code/ Intellij
2. JDK (JDK17/JDK 21)
3. MySQL Server

Annotations

1. @Component – It will help to specify a Class as a Spring Bean (Generic)
2. @Controller – Helps to assign a class as API end point
3. @RestController – REST ful API end points (Http Methods)
4. @ReuestMapping -- @Get, @Post, @Put, @Delete
5. @Repository -- DAO component
6. @Service – Business Logic

Stereo Type - @Component, @Bean, @Service, @Repository

DTO – Data Transfer Objects (Temporary)

<https://www.postman.com/downloads/>

We can create a collection for testing a particular service. And we can re-run the test n no of times.

By default in browser – http method used is GET method only.

CRUD – HTTP METHOD

Create/Insert Operation – POST (For sending data from client to server) – Entity in Request Body

Read (ReadAll, ReadById) – GET (For getting data from server to client)

Update – PUT (For modifying existing resource – ID is needed, Entity in Request Body)

Delete – DELETE (For removing a particular record/row from the table – ID is needed)

Test endpoints of API

1. Using any HttpClient ( Postman, SOAPUI)
2. Using CLI – cURL (From command prompt)
3. With the help of API Documentation (Swagger)

Hard Delete – Permenant removal of data.

Soft Delete – Temporary delete / isActive,

External Dependency -

API

1. Read Operation (It won’t alter the DB contents) - Idempotent
2. Modify Operation ( Update/Delete/Insert) -

# Monolith VS Micro-Service

# Module 1: Introduction to Microservices Architecture

## What is Microservices Architecture?

* Microservices is an architectural style where an application is structured as a collection of small, independent, and autonomous services that collaborate to deliver business value. Each service corresponds to a single business capability and can be developed, deployed, and maintained independently.
* *Analogy*: Think of microservices as a fleet of independent ships working together rather than a single, cumbersome vessel.

## Monolithic vs. Microservices: The Evolution

* **Monolithic Applications**:
  + Consist of one tightly coupled codebase where all business logic, data access, and UI are packaged together.
  + Scaling or updating one part often impacts the entire application, making it less flexible and harder to maintain.
* **Key challenges of monolithic architecture**:
  + Difficulty in scaling specific components.
  + Slower deployment cycles—any change, big or small, requires the entire app to be rebuilt and redeployed.
  + Harder to update or experiment because everything is interdependent.
* **How Microservices Solve These Issues**:
  + Independent development, deployment, and scaling of each service.
  + Technology diversity—different microservices can use different programming languages and data stores.
  + Organizations can innovate and scale different business functions independently.

Real-World Example

* Amazon transformed its monolithic e-commerce platform into a suite of microservices, enabling rapid feature rollout, better resiliency, and the ability to scale specific features like payments or recommendations independently.

# Module 2: Key Principles of Microservices

## Core Principles

* **Decentralization**: Each microservice is developed and managed independently, reducing single points of failure and bottlenecks.
* **Autonomy**: Microservices own their logic and data. Each service can be built, tested, deployed, and scaled independently.
* **Independence**: Each service serves a particular capability (e.g., order-processing, payment, customer management) and updates don’t affect the entire system.
* **Resilience**: Faults are isolated at the service level. If one microservice fails, it shouldn’t break the whole system (patterns like Circuit Breaker help here).
* **Domain-Driven Design (DDD)**: Organize services around business domains to better align technology infrastructure with organizational structure.

## Supporting Principles

* API-first design with clear and versioned contracts
* Single Responsibility Principle: every service has one clear purpose.
* Polyglot persistence: each service can choose its optimal database type.
* DevOps & CI/CD: Accelerate agile practices and enable rapid iterations.

# Module 3: Designing Microservices – Domains & Boundaries

## The Importance of Domain-Driven Design (DDD)

* **Business Domains**: Each service matches a business domain (e.g., user management, inventory).
* **Bounded Contexts**: A clear boundary defines the responsibilities and data of each microservice. This minimizes overlap, redundancy, and cross-service dependencies.
* **Context Mapping**: Identify where business concepts begin and end in your app; use context maps to visualize and communicate these boundaries.

## How to Break Down a Monolith Step-by-Step Process:

1. **Identify Business Domains**:
   * Collaborate with subject matter experts to map current business processes and capabilities.
2. **Define Bounded Contexts**:
   * Ensure each microservice boundary aligns with a logical, independent span of business functionality.
   * Avoid splitting by technical layers (UI, data), focus on business function.
3. **Design Service Interfaces**:
   * APIs establish communication. Aim for robust, decoupled, versioned contracts.
4. **Establish Data Ownership**:
   * Give each microservice its own data store: avoid shared databases.
5. **Implement Teams Around Services**:
   * Craft cross-functional squads that own the lifecycle of each microservice.

## Practical Example

* For a banking platform, split services around capabilities: Account Management, Transaction Processing, Customer KYC, Loan Management. Each is managed, tested, and deployed independently.

# Module 4: Microservices Design Patterns

## Essential Design Patterns

| **Pattern** | **Description** | **Benefits** |
| --- | --- | --- |
| API Gateway | Single entry point for all clients, routes requests to the right microservice, handles auth, monitoring, and rate-limiting. | Simplifies client interactions, security, and routing management. |
| Service Discovery | Enables dynamic location of microservices—services register and find each other as they come/go in the system. | Handles dynamic scaling, reduces manual config. |
| Circuit Breaker | Protects the system from cascading failures. Breaks the connection after repeated failures, auto-recovers. | Increases resilience, localizes faults. |
| Strangler Fig | Enables gradual migration from monolith to microservices. New features go to microservices, legacy code is phased out. | Reduces risk, supports incremental migration. |

## Deep Dive into Key Patterns

* **API Gateway**: Handles requests from clients, routes to correct microservice, manages cross-cutting concerns—authentication, rate limiting, aggregate responses for clients (e.g., web/mobile).
* **Service Discovery**: Maintains a registry of all running services so new or scaled instances can be found automatically (e.g., Netflix Eureka, Consul).
* **Circuit Breaker**: Detects failing services, prevents calls to unhealthy endpoints until they recover, and provides fallback solutions (e.g., Hystrix).
* **Strangler Fig**: Slowly decompose a monolithic application by routing new features to microservices, ensuring continuous delivery with minimal disruption.
* **Database per Service**: Each service owns its data, promoting loose coupling and independent scaling; requires distributed data management strategies.
* **Event-driven/Message Broker (AMQP)** : Use messaging queues/streams for decoupled, async communication, increasing scalability, and resilience.
* **Saga Pattern**: Manage transactions across services via a series of local transactions and events.