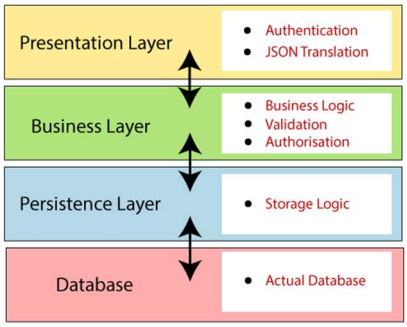
Microservices

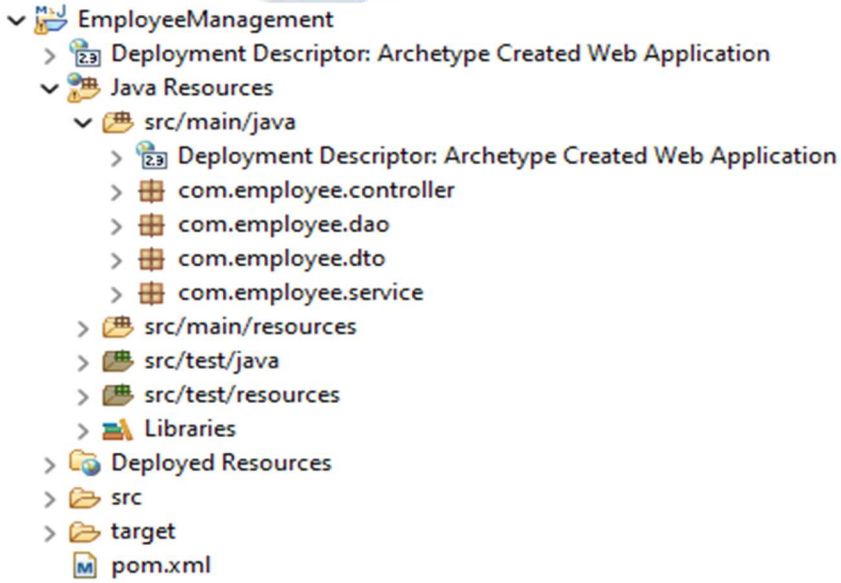
Course Content :

- Monolith Architecture,
- Drawbacks of Monolithic
- Microservices Architecture
- Pros & Cons with Microservices
- Service Registry (Eureka)
- Admin Server
- Zipkin Server,
- Config Server
- FeignClient (Interservice Comm)
- API Gateway (Filters + Routing)
- Circuit Breaker (Resillence4J)
- Load Balancer (Ribbon)

❖ Monolithic Architecture :



- Monolithic architecture in Java refers to a software design paradigm where the **entire application is built as a single, unified unit.**
- All components of the application, including the user interface (UI), business logic, and data access layer, are **tightly coupled** (if they are not carefully designed using strategy design pattern) and run within a single process.
- Please look into the pictures



- **Drawbacks with Monolith Architecture**
  - **Response Delay :**

    Response delay in a monolithic architecture occurs when the application takes longer than expected to respond to client requests. This delay can arise due to various. Most common factor is :

    Lack of load balancing :

    Here load balancing means distributing the requests multiple available servers. So that we can minimize the load traffic. But in case of traditional monolithic applications do not hold built in load balancer because this all traffic is redirected to one server so it may increase burden on server. If burden is not handled properly then there is chance of encountering issues like server down or server crashes.

  - **Technology Dependent :**

    In a monolithic architecture, the entire application is built as a single, unified codebase and typically uses a single technology stack.

    This approach means that all the components, such as the user interface, business logic, and data access, are tightly coupled together within the same application.

    For example, if a developer chooses to use Spring for building a monolithic application, then the entire system will be developed using Spring and related technologies (such as Spring Boot, Spring Data, Spring Security, etc.). It leads to challenges such as reduced flexibility and scalability as the application grows.
  - **Single Point of Failure (SPOF)**

    It refers to a part of a system that, if it fails, would cause the entire system or application to stop functioning.

    In the context of a monolithic architecture, SPOF can occur in several places, and it's one of the inherent risks associated with such an approach.

  - **Burden on Server (High Traffic * Overloaded Server)**
  - **Redeploying the entire application**

    Whenever you make even a small change, whether it's a bug fix, a minor feature update, or a change in configuration **redeploying** the entire app is necessary. This happens because all the components of the application are tightly coupled into a single codebase and deployment unit.

To overcome problems of Monolithic Architecture, people are using Microservices Architecture.

Microservices : (Not a technology | Not a Framework | Not a library)

==========

- It is widely recognized Universal **Architectural Design Pattern**.
- It provides guidelines on how to decompose large systems into independent, manageable components (Microservice(s)) that communicate with each other.
- Each microservice / component is focused on a specific business domain, has its own dedicated data store or inter linked data store, and communicates with other services over lightweight protocols, usually through APIs or messaging systems.
- The main aim of this design pattern is to achieve loosely coupling.
- It does not matter what technology is being used in project development we can apply this Design pattern (Java, Python, Dot Net etc…)

Advantages with Microservices:

- Loosely Coupled :

  It refers to a design principle where individual services are independent from one another. This means that each microservice operates as a self-contained unit and has minimal dependencies on other services, which allows for flexibility, scalability, and maintainability.

- Burden reduced on Servers :

  In the context of microservices, burden reduced on servers refers to the way in which microservices architecture helps in optimizing resource usage, leading to more efficient server utilization compared to monolithic architectures. Here's how microservices help reduce the load on servers.

  Independent Scaling of Services :

  If one service is facing high traffic or load, only that service needs to be scaled.

  Optimized Resource Allocation :

  Microservices allow the distribution of different components of the application across multiple servers. Each microservice can be deployed on different servers or containers, and resource allocation can be customized according to the service's needs.

- No single point of failure

  This refers to the ability of the system to remain operational and continue functioning even if one or more individual services fail. In a loosely coupled system like microservices, failure in one service doesn't lead to the collapse of the entire system,

- Easy Maintenance

  This design pattern allows developers to perform various operations on each service without effecting the other service.

  Faster Fixes: Smaller, isolated changes mean quicker identification and resolution of issues.
  Reduced Downtime: Services can be updated without taking the whole system down, allowing for high availability.
  Improved Developer Productivity: Independent teams can focus on their respective services without needing to coordinate changes across the entire system.
  Simplified Troubleshooting: Fault isolation and clear APIs make it easier to trace issues.
  Flexibility in Updates: New features, patches, or technology upgrades can be introduced to one service without impacting others.

- Technology Independent:

  This design pattern offers one important key feature that is Technology Independent.
  It means that we can use Java Spring Boot technology to develop service 1 and for developing service 2 we can use python and other languages also.

- Quick Deliveries

  It refers to the ability to rapidly develop, test, and deploy features or updates due to the decentralized, independent nature of the services.
  Microservices enable organizations to shorten the time between the development of a new feature and its deployment into production, which leads to faster delivery of value to customers. Here's how microservices contribute to quick deliveries:

Challenges with Microservices :

- Bounded Context :

  Deciding how to **partition** the application into microservices. When to create a new service and what functionality to include in it can be a complex decision.

  Example :
  Consider an e-commerce platform with the following possible services:

User Service	(manages user registration, login)
Product Service	(manages product catalog)
Order Service	(handles customer orders)
Payment Service	(processes payments)

  Scenario: If you decide to create a Customer Service that contains both user details (from the User Service) and order history (from the Order Service), it may result in duplicate data and

unwanted dependencies. This is a case where the bounded context is not clearly defined, and the responsibility for maintaining customer data is split across multiple services.

A better approach would be:
The User Service should manage user authentication and user details.
The Order Service should focus on handling orders and order history.
The Payment Service should handle payment processing separately. By clearly defining the boundaries, you avoid the overlap of responsibilities, leading to more maintainable and scalable microservices.

- Repeated Configuration:

The challenge with repeated configurations is ensuring consistency and manageability. Each service must have its own set of configuration files for common services like databases or messaging queues, and this can result in:

Code duplication: The same configurations are repeated in multiple services.
Difficult updates: If a change is needed (e.g., a change in the database credentials), it must be applied to all services individually.

Example:
Suppose you have the following microservices:

Order Service
Inventory Service
Payment Service

Each of these services may need to connect to the same database and send messages to the Kafka queue for communication. If each service has its own configuration files for these resources (e.g., database.properties, kafka-config.json), there will be a lot of repeated configurations.

The Order Service connects to the Inventory Service and Payment Service through Kafka to process orders.
If the Kafka configuration changes (e.g., a change in the broker URL or credentials), you must update the configuration in every service individually.
If the database configuration changes, each microservice must be updated with the new credentials.
This redundancy introduces the risk of configuration inconsistency across services and increases the effort to keep all services in sync.

To manage this challenge, a common solution is to use a centralized configuration management system like Spring Cloud Config, Consul, or HashiCorp Vault. These tools store configuration settings in a central location, allowing services to fetch their configurations dynamically, reducing duplication.

Visibility**:**

With multiple independent services running, maintaining visibility into the entire system's performance and behavior can be challenging. Some of the visibility-related issues include:

Tracking Requests:

Since each service is a separate entity, tracing a single request across multiple services (for example, a user request that triggers a series of interactions between different microservices) can become difficult.

Distributed Tracing:

Understanding how each service is performing and how errors are propagated through the system can be challenging without a proper tracing and logging system.

Failure Monitoring:

Identifying the root cause of failures is more difficult since the failure might propagate across multiple services, and logs from each service may not always be sufficient.

Consider an online shopping cart experience, where a customer:

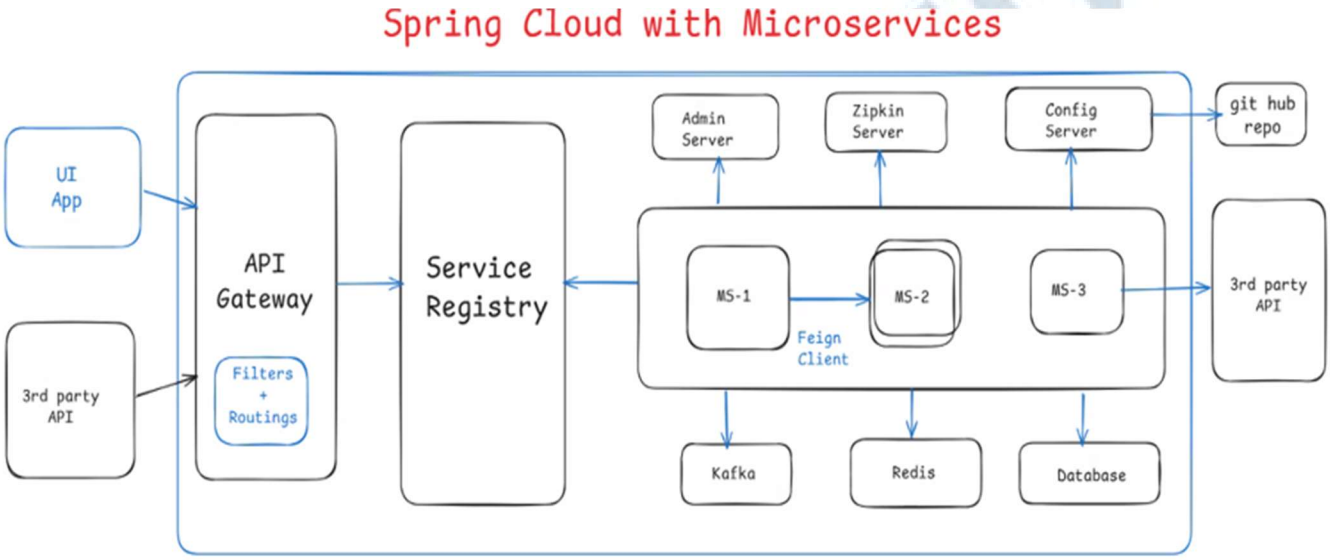
Adds items to their cart (Cart Service).
Proceeds to checkout (Order Service).
Makes a payment (Payment Service).

Each of these services is independent and communicates with others. If there's an issue with the payment (for example, payment is not being processed), it's hard to pinpoint whether the issue lies in the Payment Service, the Order Service, or the Cart Service.

Spring Cloud with Microservices :

==========

- Spring Cloud is a collection of tools that helps to implement common patterns found in microservices architectures, particularly in the context of distributed systems.
- It offers a set of frameworks and solutions that simplify the development of microservices in Java, leveraging the power of the Spring Framework and related Spring projects.
- Spring Cloud provides several components that help manage and solve challenges that arise when building microservices, such as **service discovery**, **load balancing**, **configuration management**, **messaging**, and **more**.
- Let's break down how Spring Cloud integrates with microservices and the key components it offers. Please look into the below picture.



Service Registry :

- A Service Registry, also known as a Discovery Server, is a central system that keeps track of all the microservices in a distributed application.
- Its primary purpose is to help microservices find and communicate with each other dynamically without relying on hardcoded configurations.
- In a microservices architecture, services often scale up and down dynamically, and their instances might change IP addresses or locations. A Service Registry addresses this challenge by maintaining a live directory of services and their details, such as:

Name:	The service identifier.
Status:	Whether the service is running or down.
URL:	The endpoint to access the service.
Instances:	The number of service instances.
Health:	The health status of each instance.

Example :

Eureka Server:

Eureka is a popular service registry implementation provided by Spring Cloud.

It offers:
1) A user-friendly UI for monitoring services.
2) A robust API for registering and discovering services.
3) Built-in health checks

Admin Server :

==========

- An Admin Server is a centralized tool used to monitor and manage microservices in a distributed system.
- It provides a web-based dashboard where developers and administrators can view the status, metrics, and details of all the microservices in real-time.

Purpose of an Admin Server

Service Monitoring:

Keeps track of the status (running, down, etc.) of all microservices.

Displays runtime metrics such as memory usage, CPU load, and active threads.

Health Checks:

Monitors the health of microservices and displays their current state.

Alerts administrators if a service is unresponsive or unhealthy.

Instance Management:

Shows all the instances of a service running in different environments (e.g., development, testing, production). Helps track the scaling and load-balancing status.

Logs and Metrics Visualization:

Aggregates logs from various services.

Visualizes metrics to identify bottlenecks or performance issues.

Configuration Overview:

Displays configuration settings for microservices, ensuring transparency and consistency.

Zipkin Server:

- Zipkin is a distributed tracing system designed to help monitor, troubleshoot, and understand the flow of requests across microservices in a distributed system.
- The Zipkin Server acts as a central component that collects and stores trace data from various microservices and provides tools to visualize and analyze these traces.

Purpose of Zipkin Server

Distributed Tracing:

Tracks the flow of requests across multiple microservices. Identifies how long each service takes to process a request.

Performance Monitoring:

Pinpoints bottlenecks or slow services in the system. Analyzes latency issues in real time.

Dependency Analysis:

Displays the relationships and dependencies between services.
Helps developers understand service interactions.

Error Debugging:

Detects errors or failures in service communication.
Tracks failed requests to identify the root cause.

Config Server :

In a microservices architecture, each service is designed to be independent and self-contained. However, as the number of services grows, managing configurations across services becomes challenging. Spring Cloud Config Server addresses this issue by providing centralized configuration management.
Spring Cloud Config Server is a component of the Spring Cloud framework that provides a central place to manage external properties for applications across all environments (development, staging, production, etc.). It enables microservices to fetch their configuration properties dynamically from a centralized location, ensuring consistency and easier management.

Key Features of Spring Cloud Config Server:

Centralized Configuration: All configuration data is stored in a single place, typically in a Git repository or file system.

Environment-Specific Configurations: Supports configurations for different environments (e.g., application-dev.yml, application-prod.yml).

Dynamic Updates: When paired with tools like Spring Cloud Bus and Actuator, services can refresh their configurations dynamically without restarting.

Security: Supports encrypted properties to store sensitive data securely.
Version Control: Leveraging Git or other version control systems allows auditing, rollback, and tracking of configuration changes.

How Config Server Works in a Microservices Architecture:

Centralized Configuration Storage:

- The configurations are stored in a repository (e.g., Git).
- Files are named to associate with specific services and environments (e.g., serviceA-prod.yml, serviceB-dev.yml).

Config Server Setup:

- The Spring Cloud Config Server is a standalone service that fetches configurations from the repository.
- It exposes REST endpoints for microservices to retrieve their configurations.

Microservices Integration:

- Microservices are configured as Spring Cloud Config Clients.
- They fetch their configurations from the Config Server using the exposed REST endpoints.
- The configuration is loaded into the microservice's Environment during startup.

Feign Client

- Feign is a declarative web service client provided by Spring Cloud.
- It simplifies making HTTP requests to other microservices in a microservices architecture.
- With Feign, you can define an interface that specifies how to communicate with a remote service, and Spring generates the implementation at runtime.
- It is primarily used for inter-service communication in a microservices architecture.
- It simplifies the process of one service calling another by providing a declarative and easy-to-use interface for making HTTP requests to other services.

Kafka Server : (In case of messaging, it works based on publisher-subscriber model)

- Kafka is commonly used in event-driven architectures within microservices. Here are some use cases:
  Asynchronous Communication : Instead of synchronous REST or gRPC calls, microservices can communicate by publishing and subscribing to events on Kafka topics. This decouples services, allowing them to scale and evolve independently.
- Example:
  An Order Service publishes an "Order Created" event to a Kafka topic.
  A Payment Service consumes the event to process the payment.

- Event Sourcing
  Kafka can act as a system of record where events represent changes in the state of an application. These events can be replayed to rebuild state or troubleshoot issues.
- Data Streaming
  Kafka enables microservices to process and analyze data streams in real-time, such as clickstream analysis, fraud detection, or IoT data processing.
- Integration with Legacy Systems
  Kafka acts as a bridge between legacy systems and modern microservices by enabling data integration in real-time.

Redis Server:

- Redis (Remote Dictionary Server) is an in-memory data structure store, that can be used as a database, cache, and message broker.
- It is highly performant and supports various data structures like strings, lists, sets, hashes, and more.
- In microservices architecture, Redis is commonly used to enhance performance by reducing the number of JDBC calls, scalability, and resilience.
- Key Features of Redis:
  In-Memory Store: Redis keeps data in memory, ensuring low latency and high throughput.
  Data Structures: Supports advanced data types like Strings, Lists, Sets, Hashes, Sorted Sets, and Streams.
  Persistence: Offers optional persistence by saving snapshots or writing logs to disk.
  Pub/Sub Messaging: Acts as a lightweight message broker for inter-service communication.
  High Availability: Supports replication and clustering for fault tolerance and scalability.
  Atomic Operations: All operations in Redis are atomic, ensuring data consistency.

Uses of Redis in Microservices

Caching:
Redis is widely used to cache frequently accessed data to improve application performance.
Common caching scenarios include:

- Storing API responses.
- Session storage for web applications.
- Storing database query results.

Distributed Locking

Redis provides mechanisms for implementing distributed locks in a microservices architecture to manage concurrent access to shared resources.

Pub/Sub Messaging

Redis supports the publish/subscribe (Pub/Sub) pattern for lightweight inter-service communication or real-time updates.

Session Management

In stateless microservices, Redis can store session data, enabling consistent user sessions across distributed services.

Rate Limiting

Redis's INCR command can track API usage counts for rate-limiting purposes.

Leaderboards

Using Sorted Sets, Redis can maintain leaderboards for applications like gaming or analytics.

Event Streaming and Queues

Redis Streams can serve as a lightweight alternative to message brokers like Kafka or RabbitMQ.

API Gateway:

- An API Gateway is a crucial component in microservices architecture that acts as a single-entry point for all client requests to interact with multiple backend microservices.
- It simplifies and centralizes common functionalities, enhancing scalability, security, and efficiency.
- Managing direct communication between clients and services can lead to complexity, tight coupling, and inefficiency.
- The API Gateway solves these problems by acting as an intermediary between clients and microservices.
- Key Responsibilities :
- Request Routing:
  Routes client requests to the appropriate microservices based on the URL or other request parameters.
- Protocol Translation:
  Translates protocols (e.g., HTTP to gRPC) and data formats (e.g., XML to JSON) as needed.
- Authentication and Authorization:
  Validates client credentials and enforces security policies.
- Load Balancing:
  Distributes incoming traffic across multiple instances of microservices to ensure high availability.
- Rate Limiting and Throttling:
  Controls the rate of incoming requests to prevent service overload.
- Caching:
  Caches frequent responses to improve performance and reduce backend load.
- Logging and Monitoring:
  Captures request and response data for debugging, monitoring, and analytics.
- Response Aggregation:
  Combines responses from multiple microservices into a single response to the client.

Route(s) in API Gateway

A route in an API Gateway refers to the mechanism that defines how incoming requests are directed (or routed) to different microservices or backend endpoints based on specific criteria, such as the URL path, HTTP method, or other request parameters.

Filter in API Gateway

A filter is a powerful mechanism used to intercept requests and responses as they pass through the API Gateway. Filters allow you to perform actions on the request before it's sent to the backend service or modify the response before it is returned to the client.

Filters are typically used for cross-cutting concerns like authentication, logging, metrics, rate-limiting, and modifying the request or response data.

Types of Filters:

Pre-filters:

Execute before the request is routed to the backend service.
Common use cases: Authentication, authorization, logging, input validation, rate limiting.

Post-filters:

Execute after the response is received from the backend service but before it is sent back to the client.

Common use cases:

Modifying the response (e.g., adding headers), logging the response, metrics collection.

Common Use Cases of Filters:

Authentication/Authorization: Check whether a request is authenticated before routing it to the appropriate service.

Logging: Log details of the request and/or response for monitoring purposes.

Rate Limiting: Implement limits on the number of requests a client can make in a certain period.

Modifying Headers: Add, remove, or modify headers before sending the request to a backend service.

Request Transformation: Modify the request body or query parameters before routing.