***Spring Framework***

Spring is a lightweight and popular open-source Java-based framework. It is used to develop enterprise-level applications. It provides support to many other frameworks such as Hibernate, Tapestry, EJB, JSF, Struts, etc. so it is also called a framework of frameworks. It’s an application framework and IOC (Inversion of Control) container for the Java platform. The spring contains several modules like IOC, AOP, DAO, Context, WEB MVC, etc.

[**Spring Framework**](https://www.geeksforgeeks.org/introduction-to-spring-framework/)is a comprehensive and versatile platform for enterprise Java development. It is known for its **Inversion of Control (IoC)**and **Dependency Injection (DI)**capabilities that simplify creating modular and testable applications. Key features include **Spring MVC**for web development, **Spring Boot**for rapid application setup, and **Spring Security**for effective authentication and authorization, **Spring Data**for database interactions and **Spring Cloud**for building microservices.

Its architecture is designed around two core principles: [Dependency Injection (DI)](https://www.geeksforgeeks.org/spring-dependency-injection-with-example/) and [Aspect-Oriented Programming (AOP)](https://www.geeksforgeeks.org/aspect-oriented-programming-and-aop-in-spring-framework/).

***IOC***

Spring IoC (Inversion of Control) Container is the core of [Spring Framework](https://www.geeksforgeeks.org/introduction-to-spring-framework/). It creates the objects, configures and assembles their dependencies, manages their entire life cycle. The Container uses Dependency Injection(DI) to inject or assign values to the fields or object which are required to assign values in the class . It gets the information about the objects from a configuration file(XML) or Java Code or Java Annotations and Java POJO class. These objects are called Beans.   
**There are 2 types of IoC containers:**

* [BeanFactory](https://www.geeksforgeeks.org/spring-beanfactory/)
* [ApplicationContext](https://www.geeksforgeeks.org/spring-applicationcontext/)

That means if you want to use an IoC container in spring whether we need to use a BeanFactory or ApplicationContext. The BeanFactory is the most basic version of IoC containers, and the ApplicationContext extends the features of BeanFactory. The followings are some of the main features of Spring IoC,

* Creating Object for us,
* Managing our objects,
* Helping our application to be configurable,
* Managing dependencies

Example of IOC:- We have bank application . interface called bank and it has 2 abstract methods like deposit account and check balance. We have 2 bank class(SBI & ICICI) which is implementing the bank interface. Now we need to deposit amount in to the bank account. Now we are going to main method and we are creating object to SBI bank class to deposit. In future we want to deposit amount in to ICICI, Again we need to go main class and we need to create ICICI object to deposit. It is ok for now. In future another bank came and we need to deposit amount to that bank. Again we need to go source code and we need to do changes. That’s not a good to go source code and change . We don’t want to touch source code again and again. But we need to use all banks. To avoid this changes , Spring IOC came into the picture. Simply we need to create 1 file called beans.xml and in mail method we can use any one of the IOC container to get the beans or objects.

In xml file we can create bean like :- **<bean id="sbi" class="SBI"></bean>.**

To get the bean from xml, In main class/method we need to create container like :- **Bank bank = applicationContext.getBean("sbi", SBI.class);**

Spring – ApplicationContext:-

**ApplicationContext** is the sub-interface of BeanFactory. It is used when we are creating an enterprise-level application or web application. ApplicationContext is the superset of BeanFactory, whatever features provided by BeanFactory are also provided by ApplicationContext.

**ApplicationContext Implementation Classes**

1. AnnotationConfigApplicationContext container
2. AnnotationConfigWebApplicationContext
3. XmlWebApplicationContext
4. FileSystemXmlApplicationContext
5. ClassPathXmlApplicationContext

*Dependency Injection:-*

Dependency Injection is the main functionality provided by Spring IOC(Inversion of Control). The Spring-Core module is responsible for injecting dependencies through either Constructor or Setter methods. The design principle of Inversion of Control keeping the Java classes independent of each other and the container frees them from object creation and maintenance. Dependency Injection in Spring also ensures loose coupling between the classes.

**Types of Spring Dependency Injection**

There are two types of Spring Dependency Injection. They are:

* **Setter Dependency Injection (SDI):** This is the simpler of the two DI methods. In this, the DI will be injected with the help of setter and/or getter methods. Now to set the DI as SDI in the bean, it is done through the bean-configuration file For this, the property to be set with the SDI is declared under the <property> tag in the bean-config file.

Ex:- <**bean** class="full class path" name="bean/object name">

   <**property** name="field name in class ">

               <**value**> value </**value**>

       <**property**/>

<**property** name="field name in class">

               <**value**> value </**value**>

          <**property**/>

    </**bean**>

* **Constructor Dependency Injection (CDI):** In this, the DI will be injected with the help of constructors. Now to set the DI as CDI in bean, it is done through the bean-configuration file For this, the property to be set with the CDI is declared under the <constructor-arg> tag in the bean-config file.

<bean id="object name" class="YourClassName">

<constructor-arg name="attributes that you have defined

in your class" value="And its corresponding values"/>

</bean>

*Bean :-*

Bean means object. In Spring, a **bean** is simply an object that is managed by the Spring container.

It is created, configured, and destroyed by the container, and it is typically a class that you define in your application to perform a specific task.

**Different Methods to Create a Spring Bean**

Here we are going to discuss how to create a Spring Bean in 3 different ways as follows:

**Method 1: Creating Bean Inside an XML Configuration File (beans.xml) :-** One of the most popular ways to create a spring bean is to define a bean in an XML configuration file something like this.

<bean id="AnyUniqueId" class="YourClassName">

</bean>

**Method 2: Using @Component Annotation:-**

@Component("collegeBean")

 // Class

**public** **class** College {

}

**Method 3: Using @Bean Annotation :-** One of the most important annotations in spring is the @Bean annotation which is applied on a method to specify that it returns a bean to be managed by Spring context. Spring Bean annotation is usually declared in Configuration classes methods.

@Configuration

public class CollegeConfig {

  // Using Bean annotation to create

    // College class Bean

    @Bean

 // Here the method name is the

    // bean id/bean name

    public College collegeBean()

    {

        // Return the College object

        return new College();

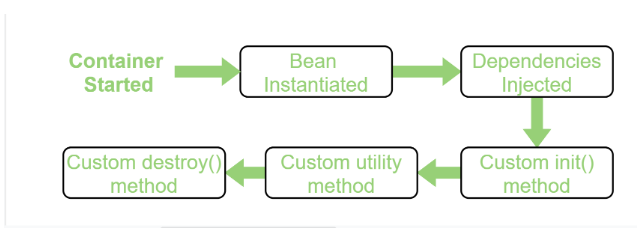
    }

}

*Bean life cycle in Java Spring*

The lifecycle of any object means when & how it is born, how it behaves throughout its life, and when & how it dies. Similarly, the bean life cycle refers to when & how the bean is instantiated, what action it performs until it lives, and when & how it is destroyed. In this article, we will discuss the life cycle of the bean.

Bean life cycle is managed by the spring container. When we run the program then, first of all, the spring container gets started. After that, the container creates the instance of a bean as per the request, and then dependencies are injected. And finally, the bean is destroyed when the spring container is closed. Therefore, if we want to execute some code on the bean instantiation and just after closing the spring container, then we can write that code inside the custom **init()** method and the **destroy()** method.



**Ways to implement the life cycle of a bean**

**By XML**:- <beans>

    <bean id="hw" class="beans.HelloWorld"

            init-method="init" destroy-method="destroy"/>

     </beans>

**By Programmatic Approach:-** To provide the facility to the created bean to invoke custom init() method on the startup of a spring container and to invoke the custom destroy() method on closing the container, we need to implement our bean with two interfaces namely InitializingBean, DisposableBean and will have to override afterPropertiesSet() and destroy() method. afterPropertiesSet() method is invoked as the container starts and the bean is instantiated whereas, the destroy() method is invoked just after the container is closed.

Note: To invoke destroy method we have to call a close() method of ConfigurableApplicationContext.

**Using Annotation:**

To provide the facility to the created bean to invoke custom **init()** method on the startup of a spring container and to invoke the custom **destroy()** method on closing the container, we need to annotate **init()** method by **@PostConstruct** annotation and **destroy()** method by **@PreDestroy** annotation.

*Bean Scope in Spring*

In Spring, **Bean Scope** defines how and when a bean is created and shared within the application. It controls the lifecycle of a bean instance and determines whether a new instance is created each time it's requested or if the same instance is reused.

**Types of Bean Scopes in Spring**

1. **Singleton (Default)**
   * **Definition**: A single instance of the bean is created and shared across the entire Spring container.
   * **Use Case**: Use when you want to share the same instance across multiple parts of the application.
   * **Example**:@Scope("singleton")
2. **Prototype**
   * **Definition**: A new instance of the bean is created every time it is requested.
   * **Use Case**: Use for stateful beans or beans that need to be short-lived.
   * **Example**:@Scope("prototype")
3. **Request (Web-specific)**
   * **Definition**: A new bean instance is created for each HTTP request.
   * **Use Case**: Use in web applications where each user request needs its own bean instance.
   * **Example**:@Scope("request")
4. **Session (Web-specific)**
   * **Definition**: A single bean instance is created for the lifetime of an HTTP session.
   * **Use Case**: Use for storing user-specific data in a session.
   * **Example**:@Scope("session")
5. **Application (Web-specific)**
   * **Definition**: A single bean instance is created and shared for the lifecycle of the ServletContext.
   * **Use Case**: Use for application-wide shared data.
   * **Example**:@Scope("application")
6. **WebSocket (Web-specific)**
   * **Definition**: A single bean instance is created and shared for the lifecycle of a WebSocket.
   * **Use Case**: Use for managing WebSocket communication.
   * **Example**:@Scope("websocket")

**Setting Bean Scope in Spring**

You can define the scope using annotations or configuration files:

**Using Annotations:**

import org.springframework.context.annotation.Scope;

import org.springframework.stereotype.Component;

@Component

@Scope("prototype") // or "singleton", "request", etc.

public class MyBean {

// Bean definition

}

**Using XML Configuration:**

xml

Copy code

<bean id="myBean" class="com.example.MyBean" scope="prototype" />

**Summary:**

* **Singleton**: One bean for the whole application.
* **Prototype**: New bean each time you request it.
* **Request/Session/Application/WebSocket**: Scopes for specific web application contexts.

***Spring – Autowiring***

**Autowiring** in the Spring framework can inject dependencies automatically. The Spring container detects those dependencies specified in the configuration file and the relationship between the beans. This is referred to as **Autowiring in Spring**. To enable Autowiring in the Spring application we should use[@Autowired](https://www.geeksforgeeks.org/spring-autowired-annotation/)annotation. Autowiring in Spring internally uses constructor injection.

**Modes of Autowiring**

**1. No**

This mode tells the framework that autowiring is not supposed to be done. It is the default mode used by Spring.

<bean id="state" class="sample.State">  
 <property name="name" value="UP" />  
</bean>  
<bean id="city" class="sample.City"></bean>

**2. byName**

It uses the name of the bean for injecting dependencies. However, it requires that the name of the property and bean must be the same. It invokes the **setter** method internally for autowiring.

<bean id="state" class="sample.State">  
 <property name="name" value="UP" />  
</bean>  
<bean id="city" class="sample.City" autowire="byName"></bean>

**3. byType**

It injects the dependency according to the type of the bean. It looks up in the configuration file for the class type of the property. If it finds a bean that matches, it injects the property. If not, the program throws an error. The names of the property and bean can be different in this case. It invokes the setter method internally for autowiring.

<bean id="state" class="sample.State">  
 <property name="name" value="UP" />  
</bean>  
<bean id="city" class="sample.City" autowire="byType"></bean>

**4. constructor**

It injects the required dependencies by invoking the constructor. It works similar to the “byType” mode but it looks for the class type of the constructor arguments. If none or more than one bean are detected, then it throws an error, otherwise, it autowires the “byType” on all constructor arguments.

<bean id="state" class="sample.State">  
 <property name="name" value="UP" />  
</bean>  
<bean id="city" class="sample.City" autowire="constructor"></bean>

**5. autodetect**

The autodetect mode uses two other modes for autowiring – constructor and byType. It first tries to autowire via the constructor mode and if it fails, it uses the byType mode for autowiring. It works in Spring 2.0 and 2.5 but is deprecated from Spring 3.0 onwards.

<bean id="state" class="sample.State">  
 <property name="name" value="UP" />  
</bean>  
<bean id="city" class="sample.City" autowire="autodetect"></bean>

**Spring – Stereotype Annotations**

*So the stereotype annotations in spring are* ***@Component, @Service, @Repository, and @Controller****.*

**Spring @Qualifier**

If we are using @Autowire , then it will check for the “byType” to inject the values. If we have 2 objects/beans for the same class, it tries to resolve by name (“byName”). When this automatic resolution fails due to ambiguity, @Qualifier can be used to explicitly specify which bean should be injected.

**Spring @Required Annotation**

If there is any field that mandatory to insert the value , then we should use @Required to assign values without miss.

Ex:-In Student class we have fields like roll no, name, age. Here roll no should be mandatory so we can use @Required to assign values.

***Spring Data JPA***

Spring Data JPA is a part of Spring Framework that makes developer **easy to work with databases** using the **Java Persistence API (JPA)**. It provides a layer of abstraction so you can interact with your database without writing a lot of boilerplate code.

**Core Features of Spring Data JPA:**

1)***Repository Abstraction****:*

* Spring Data JPA introduces **repository interfaces** (like JpaRepository, CrudRepository, and PagingAndSortingRepository) to handle common database operations such as saving, deleting, and fetching data. These interfaces provide default implementations for these operations.

->Ex:- If we have EmployeeRepository and we need to paly with DB for data , Then simply we can extends JpaRepository , This enables operations like findAll(), save(), deleteById(), etc., without writing any implementation.

public interface **EmployeeRepository** extends **JpaRepository**<Employee, Long> {}

2) ***Derived Query Methods***:

* By defining methods in your repository interface following a specific naming convention, Spring Data JPA can automatically generate queries for you.

->Example: JAP provides methods like find(), findAll(), findById() etc….. But if we want to write our own method like findByNmae this is our naming conventions. So for this also JPA will generates queries to talk with DB. Below are the syntax….

List<Employee> findByDepartment(String department);

List<Employee> findByAgeGreaterThan(int age);

These methods are translated into SQL queries automatically based on method names.

3) ***Custom Queries***:

* For more complex queries, you can use the @Query annotation to define **JPQL** (Java Persistence Query Language) or even raw SQL queries.

->Example: When we are writing our own queries to get the data from DB, at that time we can use @Query annotation to write SQL queries.

@Query("SELECT e FROM Employee e WHERE e.salary > :salary")

List<Employee> findEmployeesWithHighSalary(@Param("salary") double salary);

4) ***Pagination and Sorting****:*

**Pagination** is the process of splitting large datasets into smaller chunks or "pages" to efficiently retrieve and display data. It helps in reducing memory usage and improving application performance by avoiding the retrieval of large datasets all at once.

**Key Interfaces:**

* **Pageable**: Defines the pagination information, like the page number, size, and sorting criteria.
* **Page**: Represents a single page of data along with metadata such as total records, total pages, and current page information.
* **Slice**: Similar to Page, but only provides the data for the current page without total count information (useful for performance).

**How to Implement Pagination:**

1. Pass a Pageable parameter to your repository method.
2. Use the returned Page or Slice to handle the paginated results.

#### **Example:**

Suppose you want to fetch employees in pages of size 10:

public interface EmployeeRepository extends JpaRepository<Employee, Long> {

Page<Employee> findByDepartment(String department, Pageable pageable);

}

Usage:- Pageable pageable = PageRequest.of(0, 10); // Page 0, 10 records per page

**Sorting** allows you to order the results by one or more fields, either in ascending or descending order.

**Key Interface:**

* **Sort**: Encapsulates sorting criteria.

**How to Implement Sorting:**

You can pass a Sort parameter to your repository method or include it in the Pageable for combined pagination and sorting.

**Example:**

Fetch all employees sorted by name in ascending order:

->List<Employee> findAll(Sort sort);

Usage:

->Sort sort = Sort.by(Sort.Direction.ASC, "name");

List<Employee> employees = employeeRepository.findAll(sort);

5) ***Integration with Spring Boot****:*

* Spring Boot autoconfigures Spring Data JPA, so developers only need to specify database connection details and a spring-boot-starter-data-jpa dependency.
* Example configuration in application.properties:

properties

spring.datasource.url=jdbc:mysql://localhost:3306/employees

spring.datasource.username=root

spring.datasource.password=root

spring.jpa.hibernate.ddl-auto=update

**Common Use Case:**

Suppose you’re building an application to manage employees. Using Spring Data JPA:

1. Define an **entity class** (Employee) to map to a database table.
2. Create a **repository interface** (EmployeeRepository) extending JpaRepository.
3. Use methods like findById() or findByName() to retrieve data without writing SQL.
4. Use @Query or native SQL for custom requirements.

Spring Data JPA uses **Hibernate** as a default JPA provider**.**

***Spring Data Commons and Spring Data JPA Repositories/interfaces***

***Spring Data Commons*** is part of the  **Spring Data** project that provides shared infrastructure across the Spring Data projects. It contains repository interfaces as well as a metadata model for persisting Java classes.

Spring Data Commons project provides the following interfaces:

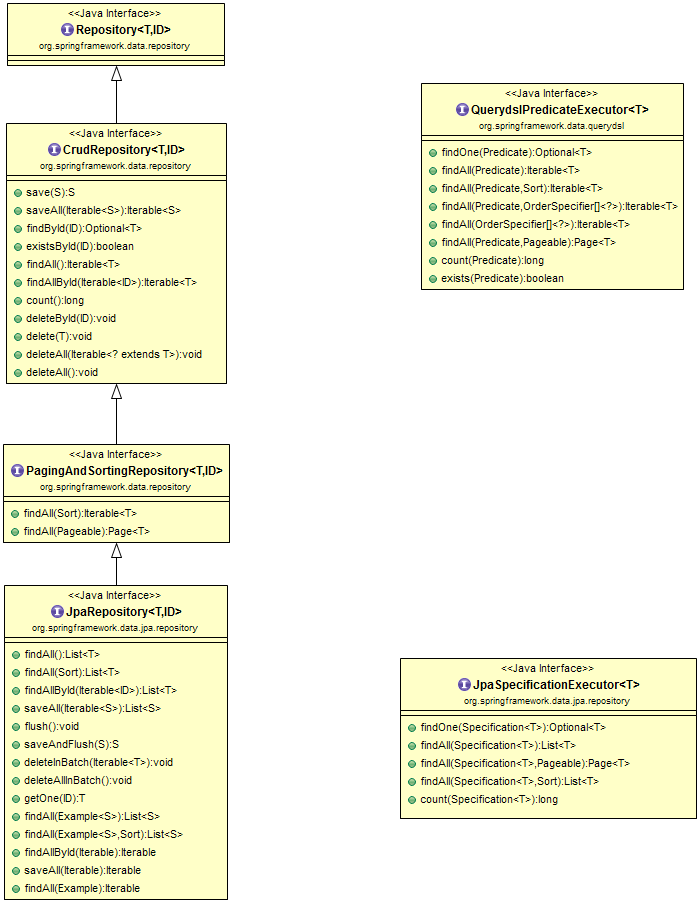
1. Repository<T, ID extends Serializable> interface
2. CrudRepository<T, ID extends Serializable> interface
3. PagingAndSortingRepository<T, ID extends Serializable> interface
4. QueryDslPredicateExecutor interface

***Spring Data JPA***module deals with enhanced support for JPA-based data access layers.

Spring Data JPA project provides the following interfaces:

* JpaRepository<T, ID extends Serializable> interface
* JpaSpecificationExecutor interface

The below diagram shows the main interfaces from Spring Data Commons and Spring Data JPA modules.



***Spring Data JPA Pagination and Sorting***

As we know, pagination allows the users to see a small portion of data at a time.

Pagination consists of two fields – **page size** and **page number** - These two fields we will use while implementing pagination using Spring Data JPA.

To use paging and sorting APIs provided by Spring Data JPA, your repository interface must extend the **PagingAndSortingRepository** interface.

PagingAndSortingRepository is an extension of the **CrudRepository** to provide additional methods to retrieve entities using the pagination and sorting abstraction. It provides two methods :

* **Page findAll**(Pageable pageable) – returns a Page of entities meeting the paging restriction provided in the Pageable object.
* **Iterable findAll**(Sort sort) – returns all entities sorted by the given options. No paging is applied here.

For pagination, we are going to use the below method from the *PagingAndSortingRepository* interface:

**Page < T > findAll(Pageable pageable);**

Note: Spring Data JPA has *SimpleJPARepository* class which implements *PagingAndSortingRepository* interface methods so we don't have to write a code to implement *PagingAndSortingRepository* interface methods.

**// create pageable object**

Pageable pageable = PageRequest.of(pageNo, pageSize);

**//Sort**

Sort sort = sortDir.equalsIgnoreCase(Sort.Direction.ASC.name())?

Sort.by(sortBy).ascending(): Sort.by(sortBy).descending();

List<Product> products = productRepository.findAll(sort);

***Spring Data JPA + Hibernate Mappings***

**One to One Mapping**

One-to-One Mapping is a type of database relationship where one entity is directly associated with one other entity, and the navigation or access to the associated entity is only possible from one side. In this case, only one entity maintains the reference to the other, and the mapping is not bidirectional.

1. One-to-One Relationship:
   * Each record in one entity (table) is linked to exactly one record in another entity (table). For example:
     + A user can have exactly one profile.
     + A passport is issued to exactly one person.
2. Unidirectional Access:
   * In unidirectional mapping, only one side of the relationship knows about the other.
   * Example: If User has a Profile, only User can access Profile. The Profile entity does not know about User.

Bidirectional Navigation:

Both entities have references to each other.

For example:

* + A User has a Profile, and the Profile also knows about the User.

🡪uses the @OneToOne annotation with @JoinColumn, while the inverse side uses @OneToOne(mappedBy) to map back to the owning side. This allows bidirectional navigation of the relationship, making it flexible for data access.

1. How It Works in JPA:
   * The entity with the reference to the other entity uses the @OneToOne annotation.
   * A foreign key column is created in the owning table to establish the relationship.

**EX:-**

Entity Classes

Let's model a User and Profile relationship.

* User has one Profile, but Profile does not maintain any reference to User.
* // Unidirectional
* @OneToOne(cascade = CascadeType.ALL)
* @JoinColumn(name = "profile\_id", referencedColumnName = "id")
* private Address profileId;

Key Annotations Explained:

1. @OneToOne:
   * Defines a one-to-one relationship between User and Profile.
2. @JoinColumn:
   * Specifies the foreign key column (profile\_id) in the User table that references the Profile entity.

Generated Tables:

1. User Table:
   * Columns: id, name, profile\_id (foreign key referencing Profile).
2. Profile Table:
   * Columns: id, bio, email.

The *@OneToOne* annotation is used to specify a one-to-one relationship between the *user* entity and the Profile entity. The cascade attribute is set to *CascadeType.ALL*, which means that any changes made to the *user* entity will be propagated to the associated *profile* entity.

**One-to-Many Unidirectional Mapping**

One-to-Many Unidirectional Mapping is a type of relationship in JPA where one entity (the parent) is associated with multiple instances of another entity (the child), but the navigation is only possible from the parent entity to the child entities. In this case, the parent maintains the reference to the child entities, but the child entities do not have a reference back to the parent.

// one to many unidirectional mapping

// default fetch type for OneToMany: LAZY

@OneToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER)

@JoinColumn(name = "order\_id", referencedColumnName = "id")

private Set<OrderItem> orderItems = new HashSet<>();

 **One-to-Many Relationship**:

* One record in the parent entity is related to multiple records in the child entity.
* For example:
  + A **Department** can have many **Employees**.

 **Unidirectional Navigation**:

* The relationship can only be navigated from the parent to the child. The child does not know about the parent.

 **Foreign Key Handling**:

* In this type of mapping, the foreign key is stored in the child entity's table to establish the relationship.

 **How It Works in JPA**:

* The parent entity uses the @OneToMany annotation to declare the relationship and maintains a List or Set of child entities.

**Many-to-Many Bidirectional Mapping**

Many-to-Many Bidirectional Mapping is a relationship in JPA where multiple records in one entity (Entity A) are associated with multiple records in another entity (Entity B), and the relationship is navigable from both sides. Each entity maintains a reference to the other, creating a bidirectional link.

@ManyToMany(fetch = FetchType.EAGER, cascade = CascadeType.ALL)

@JoinTable(

name = "users\_roles",

joinColumns = @JoinColumn(

name = "user\_id", referencedColumnName = "id"

),

inverseJoinColumns = @JoinColumn(

name = "role\_id", referencedColumnName = "id"

)

)

** Many-to-Many Relationship:**

* Each record in one entity is associated with multiple records in another entity, and vice versa.
* For example:
  + A Student can enroll in multiple Courses, and each Course can have multiple Students.

** Bidirectional Navigation:**

* Both entities maintain references to each other.
* The relationship can be navigated from Entity A to Entity B and Entity B to Entity A.

** Join Table:**

* A join table is used to represent the relationship in the database.
* This table contains two foreign keys: one for each entity.

** Owning Side and Inverse Side:**

* Owning Side: The side that manages the relationship and defines the join table.
* Inverse Side: The side mapped to the owning side using the mappedBy attribute**.**

**Many-to-One Relationship**

Many-to-One Mapping is a relationship in JPA where multiple records in one entity (the "many" side) are associated with a single record in another entity (the "one" side). The navigation is typically from the "many" side to the "one" side, making it a unidirectional relationship by default.

**@ManyToOne**

**@JoinColumn(name = "department\_id") // Foreign key column in Employee table private Department department;**

** Many-to-One Relationship:**

* Many records in one entity are linked to one record in another entity.
* For example:
  + Many Employees belong to one Department.
  + Many Orders are placed by one Customer.

** Owning Side:**

* The entity on the "many" side is the owning side of the relationship because it contains the foreign key column.

** Foreign Key Handling:**

* A foreign key is stored in the "many" side's table to reference the primary key of the "one" side.

** Unidirectional or Bidirectional:**

* In its simplest form, the relationship is unidirectional (from "many" to "one").
* It can also be bidirectional if the "one" side maintains a reference back to the "many" side.

**Spring Data JPA Repository Testing using Spring Boot @DataJpaTest**

@DataJpaTest is a specialized test annotation in Spring Boot designed specifically for testing JPA repositories. It configures the testing environment to include only the JPA-related components, ensuring a focused and lightweight testing context.

Sometimes we might want to test the persistence layer components of our application, which doesn’t require the loading of many components like controllers, security configuration, and so on. So Spring Boot provides the *@DataJpaTest* annotation to test the only repository/persistence layer components of our Spring boot application ( The *@DataJpaTest* annotation doesn’t load other Spring beans (@Components, @Controller, @Service, and annotated beans) into ApplicationContext). For that We need to use **Spring boot starter test** dependency to write JUnit tests for Repository layer components.

The Spring Boot Starter Test dependency is a primary dependency for testing the Spring Boot Applications. It holds all the necessary elements required for the testing.

This starter includes:

1. Spring-specific dependencies
2. Dependencies for auto-configuration
3. Set of testing libraries - JUnit, Mockito, Hamcrest, AssertJ, JSONassert, and JsonPath.

Spring Boot Starter Test dependency adds all the required dependencies to Unit test  Repository layer components so we don't have to add testing libraries**.**

Spring Boot provides the ***@DataJpaTest*** annotation to test the persistence layer components that will autoconfigure in-memory embedded databases and scan for *@Entity* classes and Spring Data JPA repositories.

The *@DataJpaTest* annotation doesn’t load other Spring beans (*@Components*, *@Controller*, *@Service*, and annotated beans) into *ApplicationContext*.

To test Spring Data JPA repositories or any other JPA-related components for that matter, Spring Boot provides the *@DataJpaTest* annotation. We can just add it to our unit test and it will set up a Spring application context

***Spring Security***

Spring Security is a powerful framework in the Spring ecosystem designed to handle authentication, authorization, and other security-related features for Java-based applications. It provides customizable solutions for securing applications, such as protecting APIs, managing user sessions, and enforcing security policies.

**Key Concepts in Spring Security**

1. **Authentication**

The process of verifying the identity of a user. In Spring Security, this typically involves validating credentials (e.g., username and password) provided by the user.

* Example: Logging into an application.

2. **Authorization**

Determining whether a user has the right to access specific resources or perform certain actions.

* Example: Checking if a logged-in user has the role "ADMIN" to access an admin dashboard.

3. **Principal**

Represents the currently authenticated user. It can be accessed in code to get user details like username, roles, etc.

4. **GrantedAuthority**

Defines the permissions or roles assigned to a user. For example, a user might have roles like "USER" or "ADMIN".

5. **SecurityContext**

Holds the details of the authenticated user (principal) for the current session. It can be accessed globally during a request lifecycle.

**6. UserDetails**

The *UserDetails* interface is a central piece in Spring Security, representing the user information that Spring Security uses for authentication and authorization processes. It provides core user information to the framework, such as:

* Username: The unique identifier for the user.
* Password: The user's password, usually stored in a hashed format.
* Enabled: Indicates whether the user is enabled or disabled. A disabled user cannot be authenticated.
* AccountNonExpired, credentialsNonExpired, accountNonLocked: These boolean flags provide additional details to support complex security requirements, such as account expiration policies and locking mechanisms.
* Authorities: A collection of GrantedAuthority objects representing the roles or permissions assigned to the user, which are crucial for authorization decisions.

Implementing *UserDetails* allows you to integrate your application's user entity with Spring Security seamlessly.

**7. UserDetailsService**

*UserDetailsService* is an interface used by Spring Security to retrieve user-related data. It has a single method, *loadUserByUsername(String username)*, which locates the user based on the username. The returned UserDetails object then becomes available to Spring Security for further authentication and authorization processes.  
  
Implementing your own *UserDetailsService* involves creating a service that interacts with your user database (or another user storage mechanism) to fetch user details and convert them into a *UserDetails* object. This custom service becomes a bridge between your user data and Spring Security's requirements.

**9. AuthenticationManager**

At the core of the Spring Security authentication process is the *AuthenticationManager* interface. It defines a single method, [authenticate](https://www.javaguides.net/2024/04/core-components-of-spring-security.html)*(Authentication authentication)*, which attempts to authenticate the passed [Authentication](https://www.javaguides.net/2024/04/core-components-of-spring-security.html) object. The *AuthenticationManager* is responsible for orchestrating the authentication process by delegating the request to one or more *AuthenticationProvider* instances.

***5 Ways we can deal with user credentials in Spring Security***

1. Default credentials
2. We can configure user credentials in application.properties file.

Ex:-spring.security.user.name=\*\*\*\*\*

Spring.security.user.password=\*\*\*\*

1. In-Memory credentials
2. DataBase Authentication
3. LDAP Authentication

*JWT:-*

**JWT (JSON Web Token)** is a widely used standard for securing APIs by transmitting user information between the client and server in a secure and compact way. Spring Security provides a powerful way to implement JWT authentication and authorization.

**Key Components in JWT Authentication**

**1. JWT Basics**

* Header: Contains metadata about the token, including the signing algorithm (e.g., HS256).
* Payload (Claims): Contains user-specific data, such as username, roles, or other claims.
* Signature: Ensures the token is not tampered with. It's generated using a secret key and signing algorithm.

**2. Main Components in Spring Security**

* AuthenticationManager: Validates the username and password.
* AuthenticationFilter: Captures login credentials from HTTP requests and issues the JWT upon successful login.
* AuthorizationFilter: Validates the JWT in subsequent requests to secure endpoints.
* UserDetailsService: Loads user-specific data, typically from a database, for authentication.
* JWT Utility: Handles token generation, validation, and extraction of claims.
* Security Configuration: Configures Spring Security, setting up filters and defining which endpoints are secured or open.

**Steps to Implement JWT Authentication in Spring Security**

**1. Add Dependencies**

* Spring Security provides the base for implementing authentication and authorization mechanisms.
* The JWT (JSON Web Token) library (jjwt) enables working with tokens for generation, parsing, and validation.

**2. Security Configuration**

The security configuration class extends WebSecurityConfigurerAdapter and defines how security is applied to your application.

Key Methods:

1. configure(HttpSecurity http):
   * Configures endpoint security.
   * Enables/disables CSRF and specifies which endpoints are protected or open.
   * Adds authentication and authorization filters to the Spring Security filter chain.

**Code Example:**

http.csrf().disable()

.authorizeRequests()

.antMatchers("/api/auth/\*\*").permitAll() // Open access to login and registration

.anyRequest().authenticated(); // Protect all other endpoints

1. **authenticationManager():**
   * A built-in Spring component that manages the authentication process.

**3. JWT Utility**

The JWTUtil class handles operations related to JWT tokens, such as generating, validating, and extracting claims.

**Key Methods:**

1. **generateToken(String username):**
   * Creates a JWT token with a subject (user’s username) and expiration time.
   * Uses a signing key and algorithm (HS256) to ensure the token’s integrity.

**Theory:**

* + The setSubject method sets the username.
  + setExpiration defines the token validity duration.
  + The token is signed with the secret key to prevent tampering.

**Code Example:**

public String generateToken(String username) {

return Jwts.builder()

.setSubject(username)

.setIssuedAt(new Date())

.setExpiration(new Date(System.currentTimeMillis() + 1000 \* 60 \* 60 \* 10)) // 10 hours

.signWith(SignatureAlgorithm.HS256, SECRET\_KEY)

.compact();

}

1. **extractUsername(String token):**
   * Parses the token to extract the subject (username).
2. **isTokenValid(String token):**
   * Checks if the token is valid by parsing it using the secret key.
   * If parsing fails, the token is considered invalid.

**4. Authentication Filter**

The JWTAuthenticationFilter handles login requests by authenticating the user and issuing a JWT token upon success.

**Key Methods:**

1. **attemptAuthentication(HttpServletRequest request, HttpServletResponse response):**
   * Extracts login credentials (username and password) from the request.
   * Authenticates the credentials using AuthenticationManager.
   * Throws an exception if the credentials are invalid.

**Theory:**

* + Reads the JSON request body (login data) using ObjectMapper.
  + Creates an UsernamePasswordAuthenticationToken with the credentials.
  + Passes the token to the AuthenticationManager.

**Code Example:**

UserCredentials creds = new ObjectMapper().readValue(request.getInputStream(), UserCredentials.class);

return authenticationManager.authenticate(

new UsernamePasswordAuthenticationToken(creds.getUsername(), creds.getPassword(), Collections.emptyList())

);

1. **successfulAuthentication(HttpServletRequest request, HttpServletResponse response, FilterChain chain, Authentication authResult):**
   * Called when authentication succeeds.
   * Generates a JWT token for the authenticated user.
   * Adds the token to the response header.

**Theory:**

* + The authResult.getName() retrieves the username.
  + The JWT token is returned in the Authorization response header.

**Code Example:**

String token = jwtUtil.generateToken(authResult.getName());

response.addHeader("Authorization", "Bearer " + token);

**5. Authorization Filter**

The JWTAuthorizationFilter processes incoming requests to validate the JWT token.

**Key Methods:**

1. **doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain chain):**
   * Intercepts requests to check for a valid Authorization header.
   * Validates the token and sets the authenticated user in the security context.

**Theory:**

* + Extracts the token from the Authorization header.
  + Validates the token using jwtUtil.isTokenValid().
  + If valid, extracts the username and creates a new UsernamePasswordAuthenticationToken to set the user in the security context.

**Code Example:**

String header = request.getHeader("Authorization");

if (header == null || !header.startsWith("Bearer ")) {

chain.doFilter(request, response);

return;

}

String token = header.replace("Bearer ", "");

if (jwtUtil.isTokenValid(token)) {

String username = jwtUtil.extractUsername(token);

SecurityContextHolder.getContext().setAuthentication(

new UsernamePasswordAuthenticationToken(username, null, Collections.emptyList())

);

}

**chain.doFilter(request, response);**

**6. UserDetailsService**

This component loads user-specific data from a data source (e.g., database). It provides details about the user, such as their username and roles.

**Key Methods:**

1. **loadUserByUsername(String username):**
   * Fetches the user by their username from the database.
   * Returns a UserDetails object containing the user’s information (e.g., username, password, roles).

**Theory:**

* + Implements the UserDetailsService interface.
  + Typically queries the user repository to find the user.
  + Throws a UsernameNotFoundException if the user doesn’t exist.

**Code Example:**

User user = userRepository.findByUsername(username)

.orElseThrow(() -> new UsernameNotFoundException("User not found"));

return new org.springframework.security.core.userdetails.User(user.getUsername(), user.getPassword(), new ArrayList<>());

**7. Securing Endpoints**

Define REST controllers to handle requests.

**Key Points:**

1. **Open Endpoints:**
   * Endpoints like /login and /register should be open for unauthenticated users.
   * Configured using permitAll() in SecurityConfig.
2. **Protected Endpoints:**
   * Secure all other endpoints by requiring a valid JWT token.
   * Configured using authenticated() in SecurityConfig.

**Code Example:**

@GetMapping("/protected")

public ResponseEntity<?> protectedEndpoint() {

return ResponseEntity.ok("Accessed protected resource");

}

**Summary of Workflow**

1. **Login:**
   * User sends a POST request with username and password to /login.
   * JWTAuthenticationFilter validates credentials and returns a JWT token.
2. **Access Protected Resource:**
   * User includes the token in the Authorization header for all requests to secure endpoints.
   * JWTAuthorizationFilter validates the token and grants access.

**Understanding Security Filters in Spring Security**

**Security filters** in Spring Security are part of the **filter chain** that processes incoming requests before they reach your application logic. Filters are used to:

* Authenticate users.
* Authorize access to resources.
* Handle security-related tasks like session management, CSRF protection, and exception handling.

**What is a Security Filter Chain?**

The **filter chain** is a sequence of filters configured by Spring Security. When a request arrives:

1. Each filter processes the request in order.
2. Filters either:
   * Perform some security-related logic (e.g., authentication).
   * Pass the request to the next filter.
3. If any filter determines that a request is invalid (e.g., missing credentials), it stops further processing.

The entry point for configuring the filter chain is the HttpSecurity object in the SecurityConfig class.

**Key Security Filters in Spring Security**

**1. UsernamePasswordAuthenticationFilter**

* Handles **username and password-based login** requests.
* Extracts credentials from the request (usually from the body or headers).
* Delegates the authentication logic to an AuthenticationManager.

**Key Methods:**

* **attemptAuthentication()**: Reads login credentials and attempts authentication.
* **successfulAuthentication()**: Called when authentication succeeds; can be used to generate tokens like JWTs.

**2. BasicAuthenticationFilter**

* Handles **HTTP Basic Authentication**.
* Decodes the Authorization header (Basic <Base64-encoded credentials>).
* Passes the extracted username and password to the AuthenticationManager.

**Use Case:**

Useful for APIs that use basic HTTP authentication instead of forms or tokens.

**3. JWTAuthenticationFilter**

* A **custom filter** you define for handling JWT-based login.
* Extends UsernamePasswordAuthenticationFilter to authenticate users using JWT tokens.
* Adds JWTs to the response header upon successful login.

**Key Logic:**

* Captures login data.
* Authenticates the user.
* Issues a JWT token.

**4. JWTAuthorizationFilter**

* Another **custom filter** for JWT-based authorization.
* Validates the JWT token included in the Authorization header of requests.
* Sets the authenticated user in the Spring Security context.

**Key Logic:**

* Checks if the token is present and valid.
* Parses the token to retrieve the username.
* Grants or denies access based on token validity.

**5. ExceptionTranslationFilter**

* Handles security-related exceptions raised by filters in the chain (e.g., authentication or authorization failures).
* Converts exceptions into proper HTTP responses.

**Example:**

* If authentication fails, it can return a 401 Unauthorized response.
* For access denial, it returns a 403 Forbidden response.

**6. FilterSecurityInterceptor**

* The **final filter** in the chain for authorization decisions.
* Inspects the SecurityContextHolder to determine if the user has the required authority for accessing a resource.

**Key Features:**

* Delegates the access control decision to an AccessDecisionManager.
* Throws a 403 Forbidden exception if access is denied.

**7. CsrfFilter**

* Protects against **Cross-Site Request Forgery (CSRF)** attacks.
* Ensures that requests, such as form submissions, include a valid CSRF token.
* Automatically included in the filter chain unless explicitly disabled.

**8. AnonymousAuthenticationFilter**

* Ensures that unauthenticated users are treated as **anonymous users** rather than null.
* Assigns a default authentication token (AnonymousAuthenticationToken) for such users.

**Use Case:**

Allows open endpoints (e.g., /login) to be accessed by unauthenticated users without errors.

**9. SecurityContextPersistenceFilter**

* Manages the **security context** for each request.
* Ensures that the SecurityContext is:
  + Restored at the beginning of a request.
  + Cleared after the request is processed.

**10. LogoutFilter**

* Handles logout requests.
* Invalidates the user session and clears the security context.

**Use Case:**

Triggered when a request to a specified logout URL (e.g., /logout) is made.

**How Filters Work Together in a Chain**

Here’s a typical flow for handling requests in the filter chain:

1. **Request Enters the Filter Chain**:
   * Filters like SecurityContextPersistenceFilter prepare the security context.
2. **Authentication**:
   * Filters like UsernamePasswordAuthenticationFilter or BasicAuthenticationFilter authenticate the user.
3. **Authorization**:
   * Filters like FilterSecurityInterceptor decide whether the user can access the resource.
4. **Response**:
   * If all filters pass, the request is allowed to proceed to the controller.

**Custom Filters**

In JWT-based authentication, you’ll typically define **custom filters**:

1. **JWTAuthenticationFilter**:
   * Handles login and issues tokens.
2. **JWTAuthorizationFilter**:
   * Validates tokens for protected endpoints.

***Spring Boot***

**What is Spring Boot?**

Spring Boot is a **framework** built on top of the Spring Framework to simplify the process of building and deploying Spring-based applications. It eliminates much of the configuration and boilerplate code required in traditional Spring applications.

In Spring Boot application we need to create all packages as sub packages of main class, Otherwise our code will not run as expected. If we are not following the structure then we need to use some annotations to spring let to know the packages .

If we are not following the standard package structure in a Spring Boot application, you need to explicitly specify the base packages to scan for components (like @Controller, @Service, @Repository, etc.) using the **@ComponentScan** annotation in your main application class.

**Solution: Use @ComponentScan**

Add the @ComponentScan annotation in your main application class and specify the base packages to scan:

**What Does @ComponentScan Do?**

* It tells Spring where to look for components annotated with @Component, @Controller, @Service, @Repository, etc.
* Without it, Spring would only scan the package where the main class resides and its sub-packages.

**Additional Annotations**

If you are using Spring JPA or other specific features, ensure you also configure their package scanning:

1. **For JPA Repositories** Add @EnableJpaRepositories to specify the base packages for repository scanning:
2. **For Entity Scanning** Use @EntityScan to specify the package containing JPA entities:

**Why Spring Boot Came Into the Picture?**

While the Spring Framework is powerful and flexible, setting up and configuring a Spring application could be tedious and time-consuming. Developers faced challenges like:

1. **Manual Configuration Overhead**:
   * Traditional Spring required extensive XML or Java-based configuration to wire beans, manage dependencies, and define settings.
   * This process was error-prone and time-consuming.
2. **Dependency Management Complexity**:
   * Developers had to manually specify all dependencies and ensure compatibility between library versions.
3. **Difficulty in Setting Up**:
   * Configuring servers, data sources, and security often required a lot of manual work.
4. **Standardization for Microservices**:
   * As microservices architecture became popular, there was a need for a framework that could:
     + Start quickly.
     + Embed application servers like Tomcat or Jetty.
     + Enable easy configuration for cloud-native apps.
5. **Rapid Prototyping and Deployment**:
   * Traditional Spring lacked tools to bootstrap applications quickly for development, testing, or production.

**Key Features of Spring Boot**

1. **Auto-Configuration**:
   * Automatically configures the application based on the libraries and dependencies present in the classpath.
   * For example, if you include spring-boot-starter-web, it configures Spring MVC, an embedded Tomcat server, and other web-related settings.
2. **Opinionated Defaults**:
   * Provides sensible default configurations, reducing the need for manual setup.
   * Developers can override these defaults if needed.
3. **Embedded Servers**:
   * Includes embedded servers like Tomcat, Jetty, or Undertow, allowing you to run the application as a standalone JAR without needing an external server.
4. **Spring Boot Starters**:
   * Provides predefined dependencies for common tasks, such as web development, JPA, security, etc.
   * Example: spring-boot-starter-web includes all libraries needed for a web application.
5. **Production-Ready Features**:
   * Includes monitoring and management tools (via Spring Boot Actuator).
   * Enables health checks, metrics, and application insights.
6. **Simplified Dependency Management**:
   * Uses a curated set of dependencies with compatible versions, avoiding conflicts.
7. **CommandLineRunner and ApplicationRunner**:
   * Enables running code immediately after the application starts.
8. **Spring Initializr**:
   * A web-based tool (or IDE plugin) to bootstrap a Spring Boot application with your chosen dependencies.

**Why Are We Using Spring Boot?**

1. **Ease of Development**:
   * Spring Boot reduces boilerplate code, making development faster and easier.
   * Auto-configuration and starters eliminate repetitive tasks.
2. **Standalone Applications**:
   * Spring Boot apps run as standalone JARs with an embedded server.
   * No need to deploy WAR files to external servers like Tomcat.
3. **Rapid Prototyping**:
   * Perfect for creating prototypes and MVPs due to quick setup.
4. **Microservices-Friendly**:
   * Ideal for microservices, providing built-in support for REST APIs, service discovery, and distributed configuration.
5. **Production-Ready**:
   * Includes Actuator, which offers built-in health checks, metrics, and monitoring.
6. **Simplified Testing**:
   * Integrates well with testing frameworks.
   * Provides tools like @SpringBootTest for testing Spring Boot applications.
7. **Flexibility**:
   * While opinionated, Spring Boot is not restrictive. Developers can customize configurations if defaults don't meet requirements.
8. **Community Support and Ecosystem**:
   * A large and active community ensures frequent updates, extensive documentation, and third-party integration support.

**When to Use Spring Boot?**

1. **RESTful APIs**:
   * Rapidly building REST APIs with JSON or XML responses.
2. **Microservices**:
   * Perfect for creating and managing microservices-based architectures.
3. **Simple to Medium Complexity Applications**:
   * Ideal for CRUD apps, prototypes, and cloud-native services.
4. **Cloud and Container Deployments**:
   * Its portability (standalone JARs) makes it easy to deploy on platforms like Docker, Kubernetes, or AWS.
5. **Enterprise Applications**:
   * Works well with advanced configurations and large-scale enterprise apps, combined with the flexibility of Spring.

**Comparison: Spring vs. Spring Boot**

| **Feature** | **Spring Framework** | **Spring Boot** |
| --- | --- | --- |
| **Configuration** | Requires extensive manual configuration. | Offers auto-configuration. |
| **Startup Time** | Slower due to manual setup. | Faster due to embedded servers and defaults. |
| **Embedded Server** | Not included by default. | Comes with Tomcat, Jetty, etc., pre-configured. |
| **Dependency Management** | Developers manage all dependencies. | Provides curated starters for dependency management. |
| **Learning Curve** | Steeper due to complex configurations. | Easier with built-in tools and defaults. |
| **Use Case** | For large, complex, and highly customized projects. | For rapid development, prototypes, and microservices. |

**Summary**

Spring Boot was introduced to simplify the development process for Spring applications by:

* Reducing configuration overhead.
* Automating common tasks like dependency management and server setup.
* Making it easier to build, test, and deploy applications quickly.

**Spring Boot starters**

Spring Boot **starters** are pre-defined dependency descriptors that make it easy to include commonly used libraries in a Spring Boot project. Each starter includes a curated list of dependencies that work well together, simplifying dependency management.

**Categories of Spring Boot Starters**

**1. Core Starters**

These starters are fundamental to most Spring Boot projects.

* **spring-boot-starter**:
  + The core starter that includes basic dependencies like Spring Framework, logging (SLF4J + Logback), and more.
  + **Usage**: Essential for all Spring Boot projects.
* **spring-boot-starter-logging**:
  + Provides logging support with SLF4J and Logback.
  + Automatically included in the spring-boot-starter.

**2. Web and API Development**

Used for building web and RESTful applications.

* **spring-boot-starter-web**:
  + For building web applications with Spring MVC and embedded Tomcat.
  + Includes Jackson for JSON serialization and deserialization.
* **spring-boot-starter-webflux**:
  + For building reactive web applications using Spring WebFlux.
* **spring-boot-starter-thymeleaf**:
  + For applications using the Thymeleaf template engine for server-side rendering.
* **spring-boot-starter-mustache**:
  + For server-side rendering with Mustache templates.
* **spring-boot-starter-freemarker**:
  + For applications using FreeMarker templates.

**3. Data Access**

These starters help with database connectivity and ORM tools.

* **spring-boot-starter-data-jpa**:
  + For building applications using Spring Data JPA and Hibernate.
* **spring-boot-starter-jdbc**:
  + For JDBC-based database access.
* **spring-boot-starter-data-mongodb**:
  + For applications using Spring Data MongoDB.
* **spring-boot-starter-data-redis**:
  + For applications using Redis as a key-value store.
* **spring-boot-starter-data-cassandra**:
  + For working with Apache Cassandra using Spring Data Cassandra.
* **spring-boot-starter-data-elasticsearch**:
  + For Elasticsearch integration.
* **spring-boot-starter-data-neo4j**:
  + For working with Neo4j graph databases.
* **spring-boot-starter-data-rest**:
  + For building RESTful APIs directly from Spring Data repositories.

**4. Security**

For integrating authentication and authorization.

* **spring-boot-starter-security**:
  + Provides Spring Security for authentication and authorization mechanisms.

**5. Messaging**

For building messaging and event-driven systems.

* **spring-boot-starter-amqp**:
  + For working with RabbitMQ using Spring AMQP.
* **spring-boot-starter-kafka**:
  + For Apache Kafka integration.
* **spring-boot-starter-activemq**:
  + For applications using ActiveMQ for messaging.

**6. Testing**

For adding dependencies needed for testing.

* **spring-boot-starter-test**:
  + Includes JUnit, Mockito, Spring Test, and other testing utilities.

**7. Cloud and Distributed Systems**

For building cloud-native and distributed systems.

* **spring-boot-starter-cloud-config**:
  + For integrating with Spring Cloud Config Server.
* **spring-boot-starter-eureka-client**:
  + For service discovery using Netflix Eureka.
* **spring-boot-starter-hystrix**:
  + For adding circuit breaker capabilities.
* **spring-boot-starter-cloud-bus**:
  + For propagating configuration changes across services.

**8. Actuator and Monitoring**

For monitoring and management.

* **spring-boot-starter-actuator**:
  + Provides production-ready features like health checks, metrics, and monitoring.

**9. Miscellaneous**

* **spring-boot-starter-mail**:
  + For sending emails using JavaMailSender.
* **spring-boot-starter-quartz**:
  + For scheduling jobs using Quartz.
* **spring-boot-starter-validation**:
  + For validation using Hibernate Validator.

**Custom Starters**

Developers can also create custom starters to encapsulate dependencies and configurations specific to their applications.

**How to Use Spring Boot Starters?**

1. Add the starter dependency to your pom.xml (Maven):

xml

Copy code

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

1. Or, in build.gradle (Gradle):

groovy

Copy code

implementation 'org.springframework.boot:spring-boot-starter-web'

Spring Boot automatically configures the required libraries and settings for the starter you include.

**Benefits of Using Spring Boot Starters**

1. **Simplified Dependency Management**:
   * You don't have to manually manage each library version.
   * Starters ensure compatibility between dependencies.
2. **Reduces Boilerplate Code**:
   * Auto-configuration and sensible defaults mean you write less setup code.
3. **Streamlines Development**:
   * With one dependency, you get everything needed for a specific use case.
4. **Community Support**:
   * Widely used starters are well-documented and maintained by the Spring community.

***Spring Boot Exception Handling***

Exception handling in Spring Boot is a powerful mechanism to deal with unexpected errors and ensure the application behaves gracefully. It provides developers with tools to capture and process exceptions consistently.

**Key Components of Spring Boot Exception Handling**

1. **Controller-Level Exception Handling**
   * Specific to a single controller.
2. **Global Exception Handling**
   * Applies to all controllers in the application.
3. **ResponseEntityExceptionHandler**
   * A base class for handling common exceptions with structured responses.
4. **Custom Exceptions**
   * Developers can define their exceptions for application-specific errors.

**Annotations Used in Exception Handling**

1. **@ExceptionHandler**
   * Used to define a method to handle specific exceptions in a controller.
   * Can be applied at both the controller and global levels.
2. **@ControllerAdvice**
   * A global mechanism for exception handling across all controllers.
   * Used in conjunction with @ExceptionHandler.
3. **@ResponseStatus**
   * Maps an exception to an HTTP status code.
   * Useful for associating custom exceptions with specific responses.
4. **@ResponseBody**
   * Ensures the exception handler returns the response body directly in JSON or XML format.
5. **@RestControllerAdvice**
   * Combines @ControllerAdvice and @ResponseBody for REST API projects.

### ****Approaches to Exception Handling****

#### **1. Using** @ExceptionHandler **in a Controller**

This handles exceptions specific to a controller.

**2. Using @ControllerAdvice or @RestControllerAdvice**

Handles exceptions globally across all controllers.

**3. Using @ResponseStatus for Simple Mapping**

This is useful for mapping exceptions to HTTP status codes directly.

**4. Extending ResponseEntityExceptionHandler**

This approach is used to override default behavior for built-in exceptions.

**Key Methods in Exception Handling**

1. **@ExceptionHandler Method**
   * Signature:

java

Copy code

public ResponseEntity<?> handleExceptionName(ExceptionType ex) { ... }

1. **handleExceptionInternal (in ResponseEntityExceptionHandler)**
   * Can be overridden to customize the response for specific exceptions.
2. **Utility Methods**
   * For logging exceptions or building custom error messages.

**Best Practices for Exception Handling in Spring Boot**

1. **Use Custom Exceptions**:
   * Define specific exceptions for different error scenarios (e.g., ResourceNotFoundException, ValidationException).
2. **Centralize Exception Handling**:
   * Use @RestControllerAdvice or @ControllerAdvice to manage exceptions globally.
3. **Log Exceptions**:
   * Log exceptions to help with debugging and monitoring.
4. **Return Meaningful Error Messages**:
   * Provide users with clear error messages and details.
5. **Avoid Exposing Internal Details**:
   * Don’t expose stack traces or sensitive information in API responses.

**Global Exception Handling**

Implementing global exception handling and business logic in a Spring Boot REST application ensures that errors are consistently managed and proper responses are sent back to the client. This approach enhances maintainability and improves user experience.

**Steps to Implement Global Exception Handling**

1. **Use @ControllerAdvice for Centralized Exception Handling:**
   * @ControllerAdvice is a specialized component in Spring that allows you to handle exceptions across the whole application from a single location.
2. **Define Custom Exceptions:** Create custom exception classes for specific scenarios in your application.
3. **Throw Exceptions in Your Business Logic:** Use the custom exceptions in your service or controller layer to indicate specific error conditions.
4. **Customize Error Responses:** Instead of returning plain error messages, you can send a structured error response with additional details like a timestamp, error code, etc.

* The combination of @ControllerAdvice & @ExceptionHandler will help us to implements the **Global Exception Handling.**

***Spring Cloud***

**What is Spring Cloud?**

**Spring Cloud** is a suite of tools designed to help developers build **distributed systems** and **cloud-native applications** with ease. It extends the capabilities of the Spring Framework to enable common patterns required in distributed environments, such as **service discovery**, **circuit breakers**, **configuration management**, **load balancing**, and more.

Spring Cloud is often used with **microservices architectures**, where applications are divided into smaller, independently deployable services.

**Why Spring Cloud?**

Modern applications are often deployed in cloud environments or follow microservices architectures. These setups bring challenges like:

* Dynamically locating services (Service Discovery).
* Managing configuration across multiple environments (Centralized Configuration).
* Handling network failures gracefully (Circuit Breakers).
* Load balancing among multiple instances of services.
* Monitoring and tracing distributed requests.

Spring Cloud provides solutions to address these challenges.

**Core Concepts of Spring Cloud**

1. **Service Discovery**
   * Allows services to find and communicate with each other without hardcoding URLs.
   * Uses tools like **Netflix Eureka** or **Consul**.
2. **Centralized Configuration**
   * Externalizes application configuration to a **Spring Cloud Config Server**, making it easier to manage across multiple environments.
3. **Load Balancing**
   * Distributes incoming traffic to multiple instances of a service using **Ribbon** or **Spring Cloud LoadBalancer**.
4. **Circuit Breakers**
   * Protects your system from cascading failures by monitoring and handling failures using tools like **Resilience4j** or **Hystrix**.
5. **API Gateway**
   * Acts as a single entry point to a system, routing requests to appropriate microservices using **Spring Cloud Gateway**.
6. **Distributed Tracing and Monitoring**
   * Enables monitoring of microservices interactions using tools like **Zipkin** or **Sleuth**.
7. **Event-Driven Architectures**
   * Facilitates asynchronous communication using message brokers like **RabbitMQ** or **Kafka**.
8. **Security**
   * Provides tools for OAuth2 and SSO (Single Sign-On) using **Spring Security** and **Spring Cloud Security**.

**Key Components of Spring Cloud**

**1. Spring Cloud Config**

* Provides a **centralized configuration server** for managing application properties.
* Supports versioning and profiles (e.g., dev, prod).

**Example**:

* + A config-server retrieves configuration files stored in a Git repository.

**2. Spring Cloud Netflix**

* A set of Netflix OSS tools integrated into Spring Cloud:
  + **Eureka**: Service discovery.
  + **Hystrix**: Circuit breaker.
  + **Zuul**: API Gateway (deprecated in favor of Spring Cloud Gateway).
  + **Ribbon**: Client-side load balancing.

**3. Spring Cloud Gateway**

* A modern, reactive API gateway for routing and filtering requests.
* Replaces Netflix Zuul for API Gateway functionality.

**4. Spring Cloud Sleuth**

* Adds unique trace IDs to logs for tracking requests across microservices.
* Used with **Zipkin** for distributed tracing.

**5. Spring Cloud Stream**

* Simplifies building event-driven architectures.
* Provides integration with messaging systems like Kafka, RabbitMQ, and AWS Kinesis.

**6. Spring Cloud Security**

* Adds OAuth2 and JWT-based security for microservices.
* Works seamlessly with Spring Security.

**7. Spring Cloud Kubernetes**

* Integrates Spring Cloud with Kubernetes for service discovery, configuration, and scaling.

**8. Spring Cloud Contract**

* Simplifies Consumer-Driven Contract Testing between microservices.

**9. Spring Cloud Task**

* Helps create short-lived microservices or tasks, such as batch jobs.

**How Does Spring Cloud Work?**

1. **Service Registration and Discovery**
   * Microservices register themselves with a service registry like **Eureka**.
   * Other services use the registry to discover and communicate with these services.
2. **Centralized Configuration**
   * A **Config Server** serves environment-specific configurations to microservices.
3. **API Gateway**
   * Clients interact with an API Gateway instead of directly communicating with microservices.
4. **Circuit Breakers**
   * Monitors requests to external services. If failures exceed a threshold, the circuit opens and subsequent calls fail immediately.
5. **Tracing and Monitoring**
   * Tools like **Sleuth** generate trace IDs, enabling detailed analysis of requests across services.

**Advantages of Spring Cloud**

1. **Ease of Use**
   * Built on Spring Boot, making it developer-friendly and easily integrated.
2. **Reduces Boilerplate**
   * Provides pre-built solutions for common patterns in distributed systems.
3. **Flexibility**
   * Modular design allows you to pick and choose components as needed.
4. **Scalability**
   * Enables building applications that scale efficiently in cloud environments.
5. **Cloud Agnostic**
   * Can work with various cloud providers like AWS, Azure, Google Cloud, and Kubernetes.

**Example Architecture Using Spring Cloud**

* **Client Request** → **Spring Cloud Gateway** → **Service Discovery (Eureka)** → **Microservices** → **Database**
* Logs and metrics are sent to **Sleuth** and **Zipkin**.
* Configuration is fetched from the **Config Server**.

***MicroServices***

***Monolithic Architecture***

A **monolithic architecture** is a software design where the entire application, including all its components (UI, business logic, and database), is built and deployed as a single, unified unit.

**Characteristics of Monolithic Architecture**

1. **Single Unit:**
   * All modules, including user interface, business logic, and database access, are combined into a single codebase.
   * The application is deployed as a single file or binary (e.g., a WAR or JAR in Java).
2. **Tight Coupling:**
   * Components within the architecture are closely interdependent. Changes in one module can impact others.
3. **Shared Resources:**
   * All parts of the application use shared resources like a single database or in-memory state.
4. **Centralized Deployment:**
   * The entire application must be built, tested, and deployed together.

**Advantages of Monolithic Architecture**

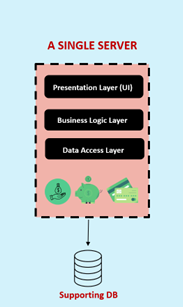
1. **Simplicity:**
   * Easier to develop for small applications due to the single codebase.
   * Straightforward debugging and testing since everything resides in one place.
2. **Performance:**
   * Communication between components happens in-process, which can be faster than distributed architectures.
3. **Ease of Deployment:**
   * A single deployable unit makes deployment and hosting simpler.
4. **Reduced Overhead:**
   * No need for managing distributed systems, inter-service communication, or separate databases.

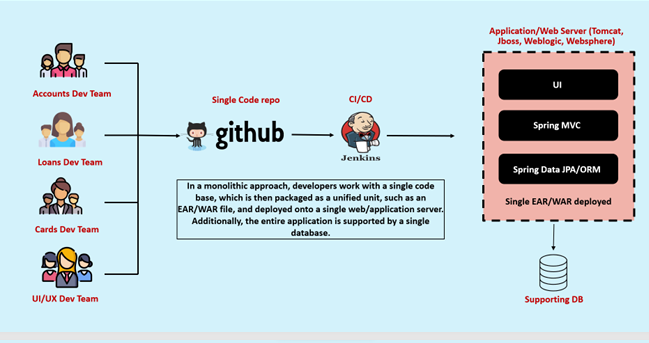
**Disadvantages of Monolithic Architecture**

1. **Scalability Challenges:**
   * Scaling requires replicating the entire application, even if only one part (e.g., a specific module) needs more resources.
2. **Reduced Flexibility:**
   * Difficult to adopt new technologies for specific components because of the tight coupling.
3. **Complexity in Large Applications:**
   * As the application grows, the codebase becomes harder to manage, test, and deploy.
4. **Limited Fault Isolation:**
   * A failure in one part of the system can potentially bring down the entire application.
5. **Deployment Bottleneck:**
   * Changes in a single module require redeploying the entire application, leading to downtime.

**Use Cases for Monolithic Architecture**

1. **Small to Medium-Sized Applications:**
   * Startups or small teams often choose monolithic architecture for simplicity.
2. **Stable and Simple Business Logic:**
   * If the application requirements are not expected to grow significantly.
3. **When Quick Deployment is Needed:**
   * Monolithic systems are faster to build and deploy compared to microservices initially.





**Service-Oriented Architecture (SOA)** came into picture to address the limitations of monolithic architecture, especially in large, complex systems.

In a monolithic system, tightly coupled components make scalability, flexibility, and maintenance challenging. SOA solves this by breaking down the application into smaller, loosely coupled **services**, each responsible for a specific business function, communicating via standard protocols (like SOAP or REST).

***Service-Oriented Architecture (SOA)***

**Service-Oriented Architecture (SOA)** is a software design approach where an application is composed of discrete, independent **services**. Each service performs a specific business function, and these services communicate with each other over a network using standardized protocols.

**Key Characteristics of SOA**

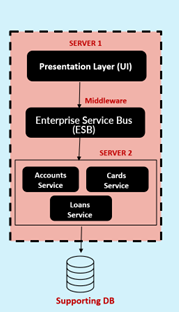
1. **Service Independence:**
   * Each service is self-contained and can operate independently.
   * Services can be reused across different applications.
2. **Loose Coupling:**
   * Services interact through well-defined interfaces and protocols, reducing dependency on internal implementations.
3. **Standardized Communication:**
   * Uses protocols like SOAP, REST, or messaging systems (e.g., JMS) for service communication.
4. **Discoverable Services:**
   * Services can be registered in a central directory or service registry, making them discoverable to other applications.
5. **Interoperability:**
   * SOA supports heterogeneous systems and technologies, allowing different programming languages or platforms to work together.

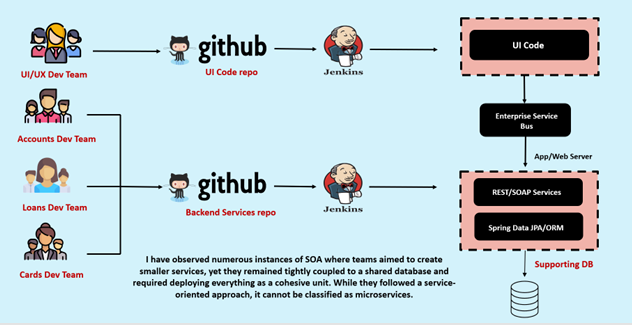
**Advantages of SOA**

1. **Reusability:**
   * Services can be reused in multiple applications, reducing duplication.
2. **Scalability:**
   * Individual services can be scaled as needed.
3. **Flexibility:**
   * Easier to replace or upgrade a specific service without affecting others.
4. **Interoperability:**
   * Allows integration of different systems, even across organizations.
5. **Fault Tolerance:**
   * Issues in one service do not necessarily impact others.

**Disadvantages of SOA**

1. **Complexity:**
   * Managing and orchestrating multiple services can be challenging.
2. **Performance Overhead:**
   * Communication between services over a network adds latency compared to in-process communication in monolithic systems.
3. **High Initial Setup Cost:**
   * Requires effort to design, implement, and deploy services and infrastructure.
4. **Governance Challenges:**
   * Standardizing protocols and ensuring compliance can be difficult in large organizations.





While **Service-Oriented Architecture (SOA)** introduced the concept of modular, loosely coupled services, it had its challenges, particularly with the complexity of managing large-scale systems. **Microservices** evolved as a refinement of SOA, aiming to address these issues with a more decentralized and agile approach.

***Microservices***

**Microservices Architecture** is a software design approach where an application is built as a collection of small, independent, and loosely coupled services. Each microservice focuses on a specific business capability, and the services communicate with each other using lightweight protocols, such as REST, HTTP, or messaging queues.

**Key Characteristics of Microservices**

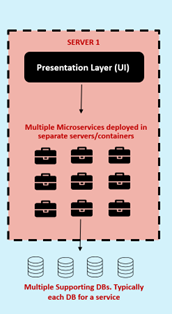
1. **Single Responsibility:**
   * Each microservice is designed to perform one specific business function (e.g., user authentication, order processing).
2. **Independence:**
   * Services are developed, deployed, and scaled independently.
3. **Decentralization:**
   * Each service has its own database, infrastructure, and logic, reducing inter-service dependencies.
4. **Lightweight Communication:**
   * Services communicate through APIs (e.g., REST, gRPC) or messaging systems (e.g., Kafka, RabbitMQ).
5. **Technology Diversity:**
   * Teams can choose the best-suited technology stack for each service (e.g., Java for one, Python for another).
6. **Fault Isolation:**
   * Failures in one microservice do not crash the entire application.

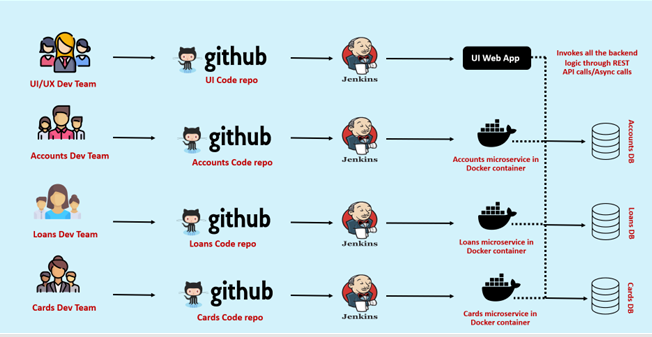
**Advantages of Microservices**

1. **Scalability:**
   * Each service can scale independently, optimizing resource use.
2. **Faster Development:**
   * Teams can work on different microservices simultaneously without interfering with each other.
3. **Flexibility:**
   * Easier to adopt new technologies for individual services.
4. **Resilience:**
   * Failure in one service does not bring down the entire application.
5. **Continuous Deployment:**
   * Services can be deployed independently, enabling frequent updates.

**Disadvantages of Microservices**

1. **Complexity:**
   * Managing multiple services, databases, and deployments can be challenging.
2. **Communication Overhead:**
   * Services need to interact over the network, which can introduce latency.
3. **Testing Challenges:**
   * Integration testing becomes more complicated as the number of services increases.
4. **Deployment Overhead:**
   * Requires sophisticated infrastructure and DevOps practices.
5. **Data Management:**
   * Maintaining consistency across distributed services can be difficult.

****

****

**How we can migrate our monolithic app to microservice?**

Migrating a monolithic application to a microservices architecture is a multi-step process that requires careful planning, design, and execution. Here’s a structured approach to achieve this migration:

**Step 1: Evaluate and Plan**

**1. Assess Your Monolith**

* Identify the current architecture, dependencies, and bottlenecks.
* Analyze the database structure and shared state.
* Determine why you need microservices (e.g., scalability, maintainability, faster development).

**2. Define Goals**

* Set clear goals: improved scalability, better fault isolation, technology diversity, etc.
* Prioritize business needs and critical services.

**3. Break Down the Application**

* Identify modules or components that can be independently extracted.
* Look for boundaries such as domain-driven design (DDD) contexts or functionality-based separations.

**Step 2: Choose the Right Strategy**

**1. Strangler Fig Pattern**

* Gradually replace parts of the monolith with microservices while keeping the monolith running.
* Redirect traffic to new services as they are built.

🡪 The **Strangler Fig Pattern** is a software development strategy used for **incrementally replacing or modernizing an existing system**. This pattern gets its name from the way a strangler fig plant grows around a tree, eventually replacing it. In software, it involves gradually building a new system around the old one and transferring functionality piece by piece until the legacy system can be retired.

The pattern allows the legacy system to coexist with the new system during the transition. Instead of replacing the entire system in one go (a "big bang" approach), functionality is migrated incrementally, reducing risks and disruptions.

**Example: Strangler Fig Pattern in Action**

**Scenario:**

A banking application needs to migrate from a monolithic system to a microservices architecture.

1. **Legacy System**:
   * Handles Accounts, Loans, and Cards in a single application.
2. **Step 1: Proxy Setup**:
   * Implement an API gateway to route all traffic to the monolith initially.
3. **Step 2: Migrate Accounts**:
   * Develop an Accounts microservice.
   * Redirect API calls for account-related requests (e.g., /accounts) to the new microservice.
4. **Step 3: Migrate Loans and Cards**:
   * Repeat the process for Loans and Cards, redirecting /loans and /cards traffic to their respective microservices.
5. **Step 4: Decommission the Monolith**:
   * Once all functionalities are migrated and tested, the monolith is retired.

**2. Feature-by-Feature Migration**

* Migrate one feature or functionality at a time.
* Ensure seamless integration between the new microservices and the monolith.

🡪 **Feature-by-Feature Migration** is a systematic approach to modernizing a legacy application by incrementally migrating individual features or functionalities to a new system. Instead of rebuilding the entire system at once, this approach focuses on moving features one at a time, ensuring stability and continuity throughout the process.

**Example: Feature-by-Feature Migration in a Banking Application**

**Scenario:**

A bank's monolithic application handles accounts, loans, and cards. The goal is to migrate to a microservices-based architecture.

1. **Feature Identification**:
   * Features include account creation, loan applications, card transactions, etc.
2. **Prioritize Migration**:
   * Start with the **Accounts** feature as it is high-impact and has minimal dependencies.
3. **Develop New Feature**:
   * Build an **Accounts microservice** to handle account-related operations.
4. **Redirect Traffic**:
   * Use an API gateway to route account-related requests to the new microservice.
   * Keep loan and card operations in the legacy system for now.
5. **Validate**:
   * Ensure the new microservice handles all account-related operations correctly.
6. **Move to the Next Feature**:
   * Migrate the Loans feature, then Cards, following the same process.
7. **Retire the Monolith**:
   * After all features are migrated, decommission the monolithic system.

The main difference between the Strangler Fig Pattern and Feature-by-Feature Migration is that the Strangler Fig Pattern is a gradual process that replaces an old application with a new one, while Feature-by-Feature Migration is a step-by-step process for installing a new application over an existing one

***Input Validation which clients provides: -***

**Input Validation** in a REST application is a critical process that ensures the data sent by clients is well-formed, valid, and adheres to business rules before processing it. This validation can happen at various levels of your Spring Boot application.

For ex:- client not following the mail format which set at the time of developing, sending account number as 5 digits instead of full, sending 3 or 4 alphabets instead of full name. In this case we need to send invalid data provided , for that we need to validate our data base with provided data. Here it will take time to query our DB. Instead of query the DB, In spring Boot app it self will throw the errors.

**Types of Input Validation in a Spring Boot REST App**

1. **Basic Validation:**
   * Validates that fields are not empty, have the correct format, or meet certain constraints.
   * Example: Checking if a field is null, empty, or has an invalid email format.
2. **Business Rule Validation:**
   * Ensures that the data makes sense within the context of your application's business logic.
   * Example: Ensuring the age field is greater than or equal to 18.
3. **Relational Validation:**
   * Ensures consistency between fields.
   * Example: The startDate should be earlier than the endDate.

### ****Steps for Input Validation in a Spring Boot REST Application****

#### 1. **Use DTOs for Input Objects**

Define a DTO class to receive data from the client and apply validation annotations.



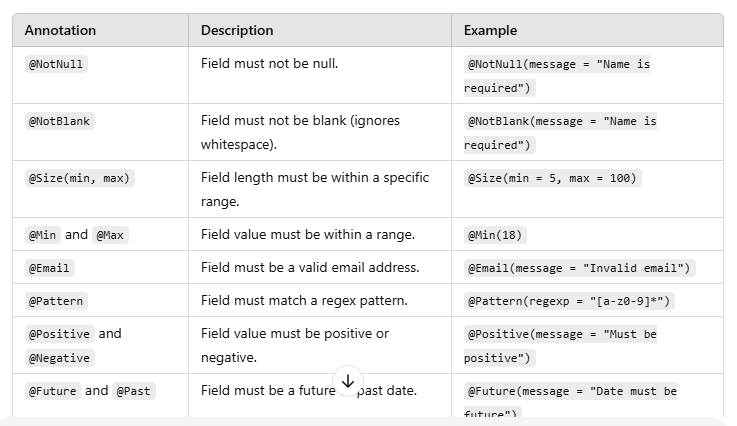
**2.Validate Input in Controller**

Use the @Valid annotation to trigger validation on the incoming request body.



**3. Handle Validation Errors Globally**

When validation fails, Spring Boot throws a MethodArgumentNotValidException. You can handle this globally in a @ControllerAdvice.

***Documentation of REST API’s***

for documentation of rest api’s we can user OPENAPI source and we can add below dependency in pom.xml

<dependency>

<groupId>org.springdoc</groupId>

<artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>

<version>2.7.0</version>

</dependency>

This will automatically deploy swagger-ui to a spring-boot application:

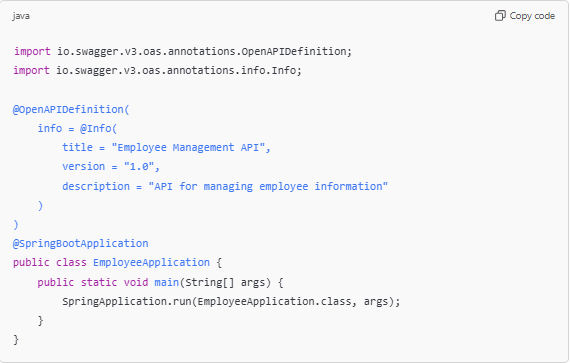
* Documentation will be available in HTML format, using the official [swagger-ui jars](https://github.com/swagger-api/swagger-ui.git)
* The Swagger UI page will then be available at http://server:port/context-path/swagger-ui.html and the OpenAPI description will be available at the following url for json format: http://server:port/context-path/v3/api-docs
  + server: The server name or IP
  + port: The server port
  + context-path: The context path of the application

The @OpenAPIDefinition annotation in SpringDoc OpenAPI is used to customize and enhance the metadata of your OpenAPI documentation. This annotation provides a way to define application-level information like title, version, description, terms of service, contact details, and license information for your API.

**Key Features of @OpenAPIDefinition**

**1. Basic Metadata**

You can define general information about your API, such as the title, version, and description:



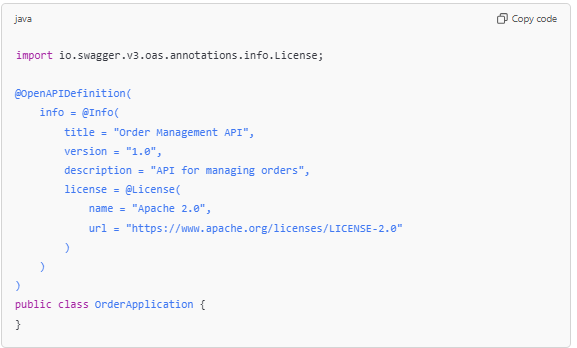
**2. Add Contact Information**

You can include contact details of the API owner or maintainer:



**3. Include License Information**

You can specify licensing information for your API:



**4. Server Information**

You can define server-related information such as base URLs for different environments:



***key OpenAPI annotations***

Here are the key OpenAPI annotations that can be used to enhance your API documentation when working with **SpringDoc** in a Spring Boot project.

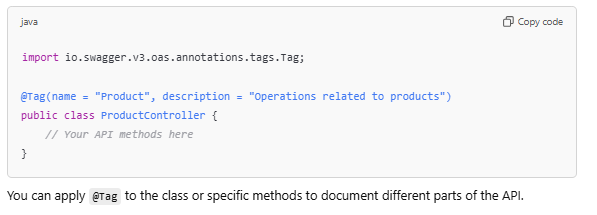
**1. @OpenAPIDefinition**

This annotation provides metadata for the entire API, such as title, description, version, and contact information. It’s generally placed on a configuration class or the main application class.



**2. @Tag**

The @Tag annotation is used to group endpoints logically. It's often used to describe specific resources in your API.



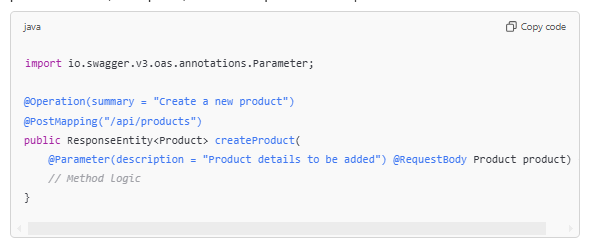
**3. @Operation**

The @Operation annotation is used to describe a single API operation (i.e., an endpoint method). It provides metadata about what the operation does.



**4. @Parameter**

The @Parameter annotation is used to describe method parameters. You can add details like parameter names, descriptions, or whether a parameter is required.



**5. @RequestBody**

The @RequestBody annotation is used to describe the body of the request. While it’s used with @Parameter, it’s often necessary to describe the request body type.



**6. @ApiResponse & @ApiResponses**

You can use @ApiResponse or @ApiResponses to specify possible responses for an operation, including status codes and response schemas.



**7. @Content**

The @Content annotation is used to specify the content type (e.g., application/json) for the response and optionally provide a schema.



**8. @Schema**

The @Schema annotation is used to provide metadata about models or properties. It can describe the type, format, default values, and more.



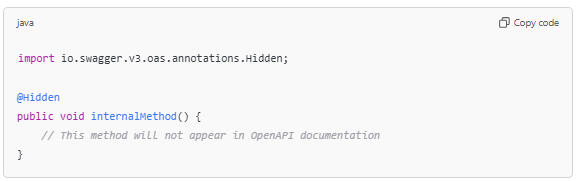
**9. @SecurityRequirement**

This annotation is used to specify security requirements for an operation, such as authentication methods (e.g., OAuth, API key).



**10. @Hidden**

The @Hidden annotation can be used to hide operations or models from the generated OpenAPI documentation.



**Summary of Key Annotations:**

* **@OpenAPIDefinition**: Provides global metadata for the API.
* **@Tag**: Used to categorize and group operations.
* **@Operation**: Provides metadata about individual API operations.
* **@Parameter**: Describes method parameters.
* **@RequestBody**: Describes the request body content.
* **@ApiResponse / @ApiResponses**: Describes possible responses for an operation.
* **@Content**: Specifies the content type of the response.
* **@Schema**: Provides metadata about model classes or fields.
* **@SecurityRequirement**: Specifies security requirements for operations.
* **@Hidden**: Hides operations or models from the OpenAPI documentation.

***Apache Kafka***

Apache Kafka is an open-source distributed platform used for building real-time data pipelines and streaming applications. It lets systems send and receive messages (data) reliably and at scale. Kafka is designed for high performance, fault tolerance, and the ability to handle large volumes of data in real time.

**Fault-tolerant** means the ability of a system to keep working even when something goes wrong,

**Example to Understand:**

Think of a fault-tolerant system like a **backup generator** in a building. If the main power goes out, the generator kicks in to keep everything running without interruption. Similarly, a fault-tolerant system keeps functioning by using backups or redundant processes when failures happen.

**Core Concepts:**

1. **Topic**:
   * Kafka organizes messages into categories called topics.
   * Each topic is divided into partitions to enable parallel processing.
2. **Producer**:
   * Applications or services that send messages to Kafka topics.
3. **Consumer**:
   * Applications or services that read messages from Kafka topics.
4. **Broker**:
   * Kafka servers that store data and serve client requests (producers and consumers).
5. **Partition**:
   * A topic is split into multiple partitions, which allows for scalability and parallelism.
6. **Offset**:
   * Each message within a partition has a unique offset to identify it.
7. **Consumer Group**:
   * A group of consumers that work together to read data from a topic.

what is kafka cluster ?

A **Kafka cluster** is a group of servers (called brokers) working together to manage and process large amounts of data streams in a distributed and fault-tolerant way.

**Key Features of a Kafka Cluster:**

1. **Brokers**:
   * Each server in the cluster is called a broker.
   * Brokers store data (messages) and serve client requests (from producers and consumers).
2. **Scalability**:
   * You can add more brokers to the cluster to handle more data or traffic.
3. **Replication**:
   * Data is replicated across multiple brokers to ensure fault tolerance. If one broker fails, another can take over without losing data.
4. **Partitions**:
   * Topics in Kafka are divided into partitions, and each partition is distributed across the brokers for parallel processing.
5. **Zookeeper (or KRaft)**:
   * Zookeeper (or Kafka’s newer built-in KRaft mode) manages the cluster’s metadata and ensures coordination among brokers.

**Why Use a Kafka Cluster?**

* **Fault Tolerance**: If one broker fails, the cluster continues to work without losing data.
* **High Availability**: Data is always accessible, even in case of failures.
* **Performance**: Multiple brokers working together can handle large amounts of data efficiently.

*what is zookeeper?*

**ZooKeeper** is a distributed coordination service that is used to manage and coordinate distributed systems. In the context of Apache Kafka, ZooKeeper helps manage and maintain the Kafka cluster's metadata, such as broker information, topic partitions, and consumer group state.

**Key Roles of ZooKeeper in Kafka:**

1. **Cluster Metadata Management**:
   * ZooKeeper stores information about which brokers are in the Kafka cluster, the topics, and the partitions. It helps Kafka brokers discover each other and know which partitions they are responsible for.
2. **Leader Election**:
   * For each partition, one broker is designated as the **leader**. ZooKeeper handles the leader election process to ensure there is only one leader for each partition at any given time.
   * If a leader broker fails, ZooKeeper helps elect a new leader from the available replicas.
3. **Consumer Group Coordination**:
   * ZooKeeper helps track consumer group state and ensures that each partition is only consumed by one consumer within a consumer group. It manages the process of rebalancing partitions when consumers join or leave the group.
4. **Broker Discovery**:
   * Kafka brokers register themselves with ZooKeeper when they start, and ZooKeeper keeps track of all the active brokers in the cluster. This allows new brokers to discover the existing brokers and join the cluster.

**How ZooKeeper Works with Kafka:**

* ZooKeeper is a separate system that runs alongside Kafka.
* Kafka brokers interact with ZooKeeper for tasks like metadata updates, partition assignments, and leader elections.
* ZooKeeper helps Kafka maintain its fault tolerance and distributed nature by keeping track of important state information.

What is kafka brocker?

A **Kafka broker** is a server in a Kafka cluster responsible for storing and managing messages (data) and handling communication between **producers**, **consumers**, and other brokers.

A **Kafka broker** is like a "data hub" that ensures producers and consumers can exchange data reliably in a Kafka cluster.

what is producer?

A **producer** in Kafka is an application or process that sends (or publishes) messages (data) to Kafka topics. Producers act as the **source of data** in a Kafka ecosystem.

why producer sends data to cluster/topic. why not directly to the consumer?

A Kafka **producer** sends data to a **cluster/topic** instead of directly to consumers because of the **decoupled architecture** of Kafka. This design offers multiple benefits:

**Scalability:**

* Producers don’t need to track or manage multiple consumers.
* Kafka handles the distribution of data to consumers, allowing for seamless scaling of consumers without impacting the producer.

**Reliability and Fault Tolerance:**

* Messages are stored in Kafka topics, ensuring they are not lost if a consumer is down or delayed.
* Consumers can retrieve missed messages when they come back online.

**Replay Capability:**

* Kafka stores messages for a configurable period. Consumers can re-read messages from a specific offset to replay data, which is impossible in a direct producer-to-consumer model.

**Multiple Consumers:**

* A single topic can have multiple consumer groups, each processing the data independently (e.g., for analytics, monitoring, and reporting). Producers don’t need to send the same data multiple times.

what is consumer?

A **consumer** in Kafka is an application or process that reads (or subscribes to) messages from Kafka topics. Consumers act as the **endpoints** that process the data produced by producers.

**Consumer Groups:**

* A **consumer group** is a group of consumers that work together to process messages from a topic.
* Each partition of a topic is assigned to only one consumer within the group, ensuring no duplication of work.
* If a consumer fails, the partitions are reassigned to other consumers in the group.

what is kafka topic?

A **Kafka topic** is a logical channel or category in Apache Kafka to which producers send messages and from which consumers read messages. It is the core abstraction in Kafka that organizes and stores data for real-time processing.

**Key Characteristics of a Kafka Topic:**

1. **Topics Are Partitioned**:
   * A topic is divided into **partitions**, which allows parallel processing and scalability.
   * Each partition is a sequential log of messages.
2. **Immutable Data**:
   * Messages in a topic are immutable and remain unchanged once written.
3. **Retention**:
   * Kafka topics store messages for a configurable **retention period** (e.g., 7 days).
   * After the retention period, messages are deleted or purged.
4. **Replication**:
   * Partitions within a topic are **replicated** across multiple brokers for fault tolerance.
5. **Producers and Consumers**:
   * **Producers** write messages to a topic.
   * **Consumers** read messages from a topic, processing them independently or in groups.

what is offset?

In Kafka, an **offset** is a unique identifier assigned to each message within a partition of a topic. It acts as a **sequence number** that allows consumers to keep track of which messages they have read and processed.

**How Offsets Work:**

* When a producer sends messages to a topic, Kafka assigns an offset to each message within the partition.
* Consumers use offsets to know:
  + Which message to read next.
  + Where to resume if they are restarted or disconnected.

Commands to start KAFKA:-

1)  Open the **Command Prompt** and navigate to your Kafka directory EX (C:\Users\M HARI KUMAR\Downloads\kafka).

 Run the following command:

First we need to start zookeeper

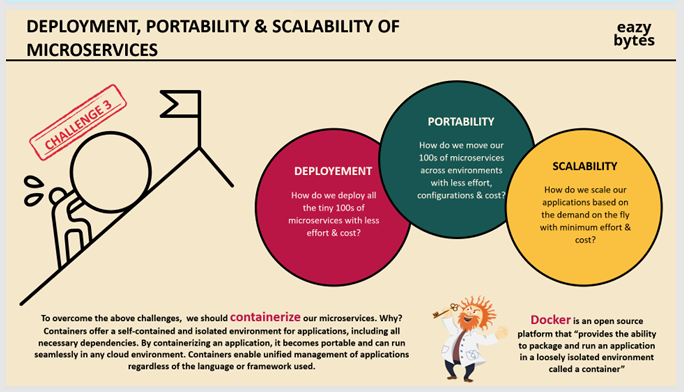
bin\windows\zookeeper-server-start.bat config\zookeeper.properties

Then we need to start Kafka with command bin\windows\kafka-server-start.bat config\server.properties

After successfully connect , we need to see our topics for that need to use below command

bin\windows\kafka-topics.bat --list --bootstrap-server localhost:9092

***Containers & Docker***

****

**what is container in docker?**

a **container** is a lightweight, standalone, and executable package that includes everything needed to run a piece of software, such as the application code, runtime, system tools, libraries, and dependencies.

**Why Do We Need Docker Containers?**

Imagine you're developing an app on your laptop. It works perfectly there, but when you move it to another computer or server, it doesn't work. Why?  
The new machine might have:

* Different operating system settings.
* Missing libraries or dependencies.
* Conflicting software versions.

**Docker containers solve this problem** by packaging:

1. Your application.
2. Its libraries, dependencies, and configurations.
3. A lightweight operating system environment.

Everything is bundled into a single, portable unit that will run consistently anywhere.

**what is Software containerization?**

Software containerization is a technology that packages an application and its dependencies together into a standardized unit called a **container**. This allows the application to run consistently across different computing environments, whether it's on a developer's machine, a testing server, or a production system.

**Popular Tools for Containerization:**

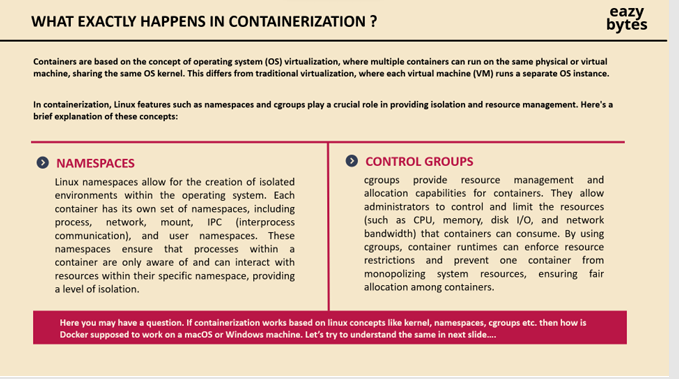
* **Docker**: The most widely used platform for building, running, and managing containers.
* **Kubernetes**: A container orchestration tool that automates deployment, scaling, and management of containerized applications.
* **Podman**: An alternative to Docker for container management.
* **Containerd**: A runtime for managing the lifecycle of containers.

**Benefits of Containerization:**

* **Rapid Deployment**: Containers start quickly, improving deployment speed.
* **Scalability**: Easily scale applications up or down using orchestration tools.
* **Simplified Development and Testing**: Developers can create consistent environments for testing and deployment.
* **Improved Resource Utilization**: Multiple containers can run on the same host without heavy overhead.

**Use Cases:**

* **Microservices**: Containers are often used to package microservices, allowing teams to develop, deploy, and scale independently.
* **Continuous Integration/Continuous Deployment (CI/CD)**: Containers streamline the CI/CD pipeline by providing consistent environments.
* **Cloud-Native Applications**: Containers are a core component of cloud-native architectures.



**What is Docker?**

Docker is an open-source platform that enables developers to build, ship, and run applications in containers. It simplifies application deployment by bundling the application and its dependencies into a standardized container that can run consistently across different environments.

**Key Features of Docker:**

1. **Containerization**:
   * Docker uses containers to package applications with everything they need to run (e.g., libraries, dependencies, configurations) into a single, portable unit.
2. **Portability**:
   * Docker containers can run on any system with the Docker runtime, ensuring consistency across development, testing, and production environments.
3. **Efficiency**:
   * Containers are lightweight and share the host operating system's kernel, consuming fewer resources than virtual machines.
4. **Versioning**:
   * Docker allows version control for containers. Developers can track changes, roll back to previous versions, and ensure consistency.
5. **Isolation**:
   * Each Docker container runs in its own isolated environment, preventing conflicts between applications or their dependencies.
6. **Scalability**:
   * Docker works well with orchestration tools like Kubernetes to manage and scale applications in production environments.

**Docker Components:**

1. **Docker Engine**:
   * The runtime environment that enables you to build and run Docker containers.
   * It consists of:
     + **Docker Daemon**: Handles container management tasks (e.g., creating, starting, and stopping containers).
     + **Docker CLI**: Command-line interface for interacting with Docker.
     + **Docker API**: Enables communication with the Docker Daemon.
2. **Docker Images**:
   * Read-only templates containing the application and its dependencies.
   * Used to create Docker containers.
3. **Docker Containers**:
   * Runtime instances of Docker images. They can be started, stopped, or modified independently.
4. **Dockerfile**:
   * A text file containing instructions to build a Docker image. It defines the application's environment, dependencies, and setup.
5. **Docker Hub**:
   * A cloud-based repository where developers can store, share, and retrieve Docker images.

**Benefits of Docker:**

1. **Simplified Deployment**: Eliminates "it works on my machine" issues by ensuring consistent environments.
2. **Faster Development and Testing**: Quickly create and replicate environments for testing and development.
3. **Improved Resource Utilization**: Containers are lightweight compared to virtual machines, optimizing resource usage.
4. **Ease of Integration**: Compatible with CI/CD pipelines, enabling automated builds and deployments.

**Common Use Cases:**

1. **Microservices**: Run microservices in separate containers for modular development and scaling.
2. **Continuous Integration/Continuous Deployment (CI/CD)**: Automate the build and deployment process with Docker.
3. **Cloud Deployment**: Deploy containerized applications on cloud platforms like AWS, Azure, or Google Cloud.
4. **Local Development Environments**: Easily replicate production environments for local development.

**How We can create Docker Images?**

We can use multiple ways to create images, Below are the samples…

**1. Dockerfile**

A **Dockerfile** is a text file that contains a set of instructions to build a Docker image. It provides a step-by-step blueprint for creating an image, including the base image, dependencies, configuration, and the application itself.

**Key Points:**

* **Syntax**: Dockerfile instructions are declarative and executed sequentially.
* **Common Instructions**:
  + FROM: Specifies the base image (e.g., FROM ubuntu:20.04).
  + RUN: Executes shell commands to install packages or configure the environment.
  + COPY/ADD: Adds files from the host to the container.
  + CMD/ENTRYPOINT: Defines the container's default behavior when started.
  + EXPOSE: Declares the port the container listens on.
* **Usage**:
  + Write the Dockerfile in your project directory.
  + Build the image using: docker build -t my-image .

**2. Docker Native Buildpacks**

**Docker Buildpacks** provide a higher-level abstraction for building Docker images without the need for a Dockerfile. They analyze the application source code and automatically build an image tailored for it. Buildpacks are part of the **Cloud Native Buildpacks** project, standardized by the CNCF.

**Key Points:**

* **Automatic Detection**: Determines the type of application (e.g., Java, Python, Node.js).
* **Layered Approach**: Creates efficient and reusable image layers.
* **Customizable**: Allows developers to define their own buildpacks if needed.
* **Simplifies CI/CD**: No manual Dockerfile is required.
* **Usage**:
  + Requires Docker 19.03+.
  + Example command:

bash

Copy code

docker buildx build --builder buildpacks --platform linux/amd64 -t my-app .

**Example:**

For a Java Spring Boot app, the buildpack will:

1. Detect the Java runtime.
2. Install dependencies.
3. Create an image with a Java runtime optimized for the app.

**Advantages:**

* Best for language- or framework-specific projects.
* Automatically optimizes images for performance and security.

**3. Jib**

**Jib** is a Java containerization tool provided by Google, designed to create optimized Docker images for Java applications. Unlike Dockerfiles, it integrates directly with Java build tools such as Maven or Gradle, eliminating the need to write a Dockerfile.

**Key Points:**

* **Integration**: Works seamlessly with Maven and Gradle.
* **Daemonless**: No need for a Docker daemon; builds images directly.
* **Reproducible Builds**: Ensures consistent image builds across environments.
* **Optimized Layers**: Separates application layers (e.g., dependencies, resources, classes) for caching and faster updates.

**Usage:**

* **Maven**: Add Jib as a plugin in your pom.xml:

<plugin>

<groupId>com.google.cloud.tools</groupId>

<artifactId>jib-maven-plugin</artifactId>

<version>3.3.2</version>

</plugin>

Build the image: mvn compile jib:build -Dimage=my-java-app

**When to Choose Each**

**1. Dockerfile**

* **When to use**:
  + If you want **complete control** over how your container is built.
  + If your app uses **multiple languages**, is old (legacy), or doesn’t follow common patterns.
* **Why**:
  + A Dockerfile lets you define every step (like installing packages, copying files, and setting commands). This is perfect when your app doesn’t fit standard templates.

**Example**: If you're building an app with a custom setup (e.g., a mix of Python and C++), Dockerfile is your best option.

**2. Docker Native Buildpacks**

* **When to use**:
  + If you need a **quick and easy way** to create a container without writing much configuration.
  + Best for popular languages like **Java, Python, or Node.js**.
  + Perfect for modern apps (e.g., microservices) that run in environments like **Kubernetes**.
* **Why**:
  + Buildpacks automatically figure out your app's requirements, install dependencies, and create a well-optimized container.

**Example**: If you're building a Node.js app and don’t want to worry about Dockerfile details, use Buildpacks.

**3. Jib**

* **When to use**:
  + If your app is written in **Java** and you want an efficient and **repeatable process** to create a container.
  + Ideal if you don’t want to install Docker but still want a Docker-compatible image.
* **Why**:
  + Jib integrates directly with Java build tools (like Maven and Gradle), making it easy to generate a container image without extra steps.

**Example**: If you have a Spring Boot app and want a fast, automated way to containerize it, Jib is the best choice.