

Spring Boot Day 1

Spring -> dependency injection framework to make java application loosely coupled

IOC -> create object, hold them in memory, inject them by DI

-> by giving beans info and config(xml file where we declare beans and its dependencies)

dependency injection -> design pattern

Inversion of control - dependency injection -> where objects define their dependencies , object instance is constructed or returned from factory method container then injects those dependencies when it creates the bean.

Bean Factory provides the configuration framework and basic functionality
ApplicationContext adds more enterprise-specific functionality.

IoC container

In Spring Boot, the IoC container is **responsible for managing the creation, configuration, and life cycle of objects, known as beans**. Instead of the traditional approach where the application code explicitly creates and manages dependencies, Spring Boot enables developers to declare dependencies and let the container handle their creation and wiring.

Here's how IoC works in Spring Boot:

- **Dependency Injection:** Spring Boot implements IoC through dependency injection, where the dependencies of a class are injected by the container. Dependencies can be expressed using annotations like `@Autowired`, constructor injection, or setter methods.
- **Bean Configuration:** In Spring Boot, beans are configured using annotations such as `@Component`, `@Service`, `@Repository`, and `@Controller`, or through XML configuration files. These annotations identify classes as beans and specify their roles within the application.
- **Container Initialization:** When a Spring Boot application starts, the IoC container, often referred to as the Application Context or Bean Factory, is initialized. The container scans the classpath, detects annotated classes, and creates bean instances based on the configuration.
- **Automatic Bean Wiring:** The IoC container resolves dependencies between beans automatically. It analyzes the dependencies declared in the application code or configuration and injects the required instances.
- **Lifecycle Management:** Spring Boot manages the lifecycle of beans within the IoC container. It ensures that beans are created, initialized, and destroyed appropriately, allowing for proper resource management.

Dependency Injection

Setter DI

- poor readability
- must include getter and setter
- requires `@Autowired`
- high coupling
- circular or partial dependencies
- preferred when properties are less

Constructor DI

- good readability
- constructor otherwise `BeanCreationException`
- do not required `@Autowired`
- loose coupling required
- no circular or partial because it resolve before object creation
- preferred when properties are more

Field DI

Bean -> an instance of class managed by spring container

Naming beans

```
Java
@Component("myBean")
@Component
    @Qualifier("specialBean")
@Component
    @Primary
```

Aliasing a bean outside the bean definition

utilizing the `BeanFactory` or `ApplicationContext` interfaces

```
Java
ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");
    // Retrieve the original bean by its defined name
```

```

MyBean originalBean = context.getBean("myBean", MyBean.class);
originalBean.doSomething();
// Create an alias for the bean
context.getAutowireCapableBeanFactory().registerAlias("myAlias",
"myBean");

// Retrieve the bean using the alias
MyBean aliasedBean = context.getBean("myAlias", MyBean.class);
aliasedBean.doSomething();

```

Instantiating beans

- > by XML configuration
- > Java Configuration (@configuration on class and @Bean on getter of Bean class)
- > Component scan (@Component on bean class)
- > Autowiring(@Bean on getter of bean class)

Instantiation with a constructor

- > constructor should be default constructor
- > if there is a parameterized constructor then default constructor should be explicit

Instantiation with a static factory method

- > we use class attribute to specify class containing static factory method
- > attribute named factory-method to specify the name of factory

Java

```

<bean id="clientService"
      class="examples.ClientService"
      factory-method="createInstance"/>

```

```

public class ClientService {
    private static ClientService clientService = new
ClientService();
    private ClientService() {}
    public static ClientService createInstance() {
        return clientService;
    }
}

```

Why IOC and DI

Using Spring's Inversion of Control (IoC), IoC containers, and Dependency Injection (DI) offer several benefits in software development. Here are some key advantages:

- **Loose Coupling:** IoC promotes loose coupling between components by decoupling the creation and configuration of objects from their usage. With DI, components depend on abstractions (interfaces) rather than concrete implementations, allowing for flexibility and easier swapping of implementations.
- **Modular and Reusable Code:** DI enables the creation of modular and reusable code by breaking down complex systems into smaller, independent components. Each component focuses on a specific responsibility, making it easier to understand, test, and maintain. Reusability is enhanced as components can be easily wired together in different configurations.
- **Testability:** IoC and DI greatly facilitate unit testing. By injecting dependencies into components, it becomes easier to isolate and test individual components in isolation. Mocking or substituting dependencies during testing becomes simpler, allowing for more effective and targeted testing.
- **Scalability and Extensibility:** IoC containers help manage the lifecycle and configuration of objects. This makes it easier to scale applications by adding or removing components without affecting the overall system. Additionally, with the use of annotations or XML configuration, extending the application with new functionality or modifying existing behavior can be achieved without modifying the core codebase.
- **Encapsulation and Information Hiding:** DI helps encapsulate implementation details within components, allowing for information hiding. Components only need to know about the interfaces of their dependencies, not the specific implementations. This reduces the impact of changes to dependencies, as long as the interfaces remain unchanged.

ApplicationContext

In the Spring Framework, the ApplicationContext is the primary interface for accessing the Spring IoC container. **It is responsible for managing and providing access to application components (beans) and their dependencies.**

The ApplicationContext interface **extends the BeanFactory interface**, providing additional functionalities and capabilities. It acts as a container that initializes, configures, wires, and manages the lifecycle of beans within a Spring application.

Here are some key features and functionalities of the ApplicationContext in Spring:

- **Dependency Injection (DI):** The ApplicationContext performs dependency injection by resolving and injecting dependencies into beans based on their configurations. It supports various types of dependency injection, including constructor injection, setter injection, and field injection.

- **Bean Lifecycle Management:** The ApplicationContext manages the lifecycle of beans within the application. It creates bean instances, initializes them, and performs any necessary cleanup or destruction when the application context is closed.
- **Configuration Management:** The ApplicationContext supports different approaches for configuring beans and their dependencies. This includes XML-based configuration, annotation-based configuration using annotations like @Component, @Service, @Autowired, etc., and Java-based configuration using @Configuration classes.
- **Resource Management:** The ApplicationContext offers resource loading and management capabilities. It allows the loading of resources, such as files or classpath resources, from various sources within the application.
- **Integration with Other Spring Modules:** The ApplicationContext integrates with other Spring modules and functionalities, such as Spring MVC, Spring Data, Spring Security, and more. It provides seamless integration and collaboration between different parts of a Spring application.

There are different implementations of the ApplicationContext interface in Spring, such as **ClassPathXmlApplicationContext**, **AnnotationConfigApplicationContext**, and **FileSystemXmlApplicationContext**, each suited for different use cases and configuration styles.

- ClassPathXmlApplicationContext
- FileSystemXmlApplicationContext
- AnnotationConfigApplicationContext

Spring Boot Day 2

Bean

A bean is an instance of a class that is created, configured, and managed by the Spring container. The container is responsible for instantiating beans, injecting their dependencies, and managing their lifecycle.

Spring Bean Configuration

- Annotation Based Configuration(using Service or Component)
- XML Based Configuration
- Java Based Configuration(using componentScan, Configuration or Bean)

Spring Benefits why

- **Lightweight and Non-Intrusive:** Spring is known for its lightweight nature and non-intrusive programming model. It doesn't require extensive configuration or impose strict programming patterns. Spring allows developers to focus on business logic without being tied to the framework's specific implementation details.

- **Dependency Injection (DI) and Inversion of Control (IoC):** Spring's core principle is DI and IoC, which **promote loose coupling** and modular design. With DI, dependencies are injected into classes rather than being created internally, making components highly reusable, testable, and easily maintainable. IoC allows the Spring container to manage object creation and lifecycle, reducing boilerplate code and enhancing flexibility.
- **AOP (Aspect-Oriented Programming) Support:** Spring seamlessly integrates AOP, allowing the separation of cross-cutting concerns such as logging, security, caching, and transaction management from the core business logic. AOP enhances code modularity and maintainability by providing a way to address concerns that cut across different components.
- **Wide Range of Modules and Integrations:** Spring offers various modules and integrations to address different aspects of application development. Spring modules cover data access (Spring Data), web development (Spring Web), security (Spring Security), messaging (Spring Integration), batch processing (Spring Batch), and more. These modules provide ready-to-use functionalities, making development faster and more efficient.
- **Flexible Configuration Options:** Spring supports multiple configuration options, including XML configuration, annotation-based configuration, and Java-based configuration using `@Configuration` classes. This flexibility allows developers to choose the configuration style that suits their preferences and project requirements.
- **Testing and Mocking Support:** Spring provides excellent support for unit and integration testing. It offers integration with testing frameworks such as JUnit and provides features like TestContext Framework for setting up and tearing down the application context in tests. Spring's DI makes it easy to mock dependencies, enabling effective testing of individual components.
- **Seamless Integration with Other Technologies:** Spring integrates smoothly with various technologies and frameworks. It supports integration with popular ORM frameworks like Hibernate and JPA, messaging systems like Apache Kafka and RabbitMQ, caching providers like Ehcache and Redis, and many other third-party libraries and systems.

Spring vs Spring Boot vs Spring MVC

Spring Framework:

- The Spring Framework is a comprehensive Java framework that provides various modules and features for building enterprise-grade applications.
- It is **built on the principles of Dependency Injection (DI) and Inversion of Control (IoC)** to facilitate **loose coupling, modularity, and testability**.
- The Spring Framework includes modules for data access (Spring Data), transaction management (Spring Transaction), web development (Spring Web), security (Spring Security), and more.
- It offers powerful abstractions, such as the Spring Core container and the Application Context, for managing and wiring application components (beans) together.

Spring Boot:

- Spring Boot is a framework built on top of the Spring Framework that aims to simplify the development of Spring-based applications.
- It provides **auto-configuration**, which **reduces the boilerplate code** and configuration required to set up a Spring application.
- Spring Boot embraces **convention-over-configuration**, allowing developers to quickly get started with minimal configuration and focus more on writing business logic.
- **It includes an embedded server (Tomcat, Jetty, or Undertow) by default**, making it easy to create standalone and self-contained web applications.
- Spring Boot offers a wide range of starters, which are pre-configured dependencies that provide ready-to-use features for various purposes, such as database connectivity, messaging, testing, and more.

Spring MVC:

- Spring MVC (Model-View-Controller) is a web framework within the Spring ecosystem for building web applications.
- **It follows the MVC architectural pattern**, where the model represents the data, the view represents the presentation layer, and the controller handles the request processing and acts as an intermediary between the model and the view.
- Spring MVC provides **powerful features for handling HTTP requests, managing session and cookies, handling form submissions, and rendering views.**
- It supports various view technologies, including JSP (JavaServer Pages), Thymeleaf, and others.
- Spring MVC integrates seamlessly with other Spring modules, such as Spring Security for handling authentication and authorization, and Spring Data for database access.
- No embedded server

Spring

- or building applications
- dependency injection
- allow loosely coupled applications
- boilerplate code
- need of server to test
- no support of database
- manually define pom.xml

SpringBoot

- to develop REST API's

- shorten the code length
- AutoConfiguration
- standalone application with less configuration
- reduces boilerplate code
- provides embedded server
- embedded database
- provides pom.xml for downloading dependencies in starter pack

Spring MVC

- framework
- powerful and flexible to develop web applications
- internally use Servlet API
- MVC architectural pattern
- models->POJOs (Plain Old Java Objects) or JavaBeans.
- views-> presentation layer, responsible for rendering and displaying data, implemented using JSP, Thymeleaf. also generate XML, JSON responses
- controller-> intermediate b/w model and view. receives request , process it and return response

separate each role modal, view, controller-
power configuration
uses core features like IOC

Runner

interface, it will be executed after application startup

void run() - function

lambda function can be used and spring boot will automatically call run method

why?

after initialization container we don't have any control to know if some beans exists or not

- initiate things if needed
- load data right before application starts
- load data from other microservices

CommandLineRunner, ApplicationRunner

CommandLineRunner:

- The CommandLineRunner interface provides a single method run(), which is called when the application starts.
- Implementing the CommandLineRunner interface allows you to define custom logic **that runs after the application context has been loaded and before the application starts processing requests.**

- The run() method **takes an array of String arguments**, which can be useful for passing command-line arguments to the application.

ApplicationRunner:

- The ApplicationRunner interface is similar to CommandLineRunner but provides a more advanced and flexible way of running custom logic during application startup.
- It provides a single method run(), **which is called when the application starts, after the CommandLineRunner beans have been executed.**
- The run() method **takes an ApplicationArguments object as a parameter**, which provides access to command-line arguments, as well as other application-specific arguments and options.

Both CommandLineRunner and ApplicationRunner provide a way to execute specific tasks during application startup. They are particularly **useful for performing initialization, setup, or other custom actions that need to be executed automatically when the application starts.** You can have multiple beans implementing these interfaces, and they will be invoked in the order they are declared as beans in the application context.

YAML and .properties

- in properties we use key-value format
- in YAML it is a hierarchical format
- we don't keep both in same package at same time it can cause abnormal behavior

extracting values from properties

```
Java
@Value annotation
env.getProperty
ConfigurationProperties(prefix=" ")
```

Spring Configuration

- XML (defines beans in xml)
- Component(Autowired an object in @Service)
- Configuration(creating object in class with @Configuration tag and @Bean on object getter setter)

Spring Boot Day 3

POJO

- > less control
- > not mandatory to implement serializable
- > can be accessed by name
- > fields can have any access modifiers
- > any kind of constructor

Bean

- > complete protection of fields
- > should implement Serializable
- > can be accessed by getters and setters
- > only private fields
- > no arg- constructor
- > default constructor can change state when object is immutable

Bean Life cycle

property set
init()
then we can read or write
destroy()

JPA Entities

- > @Entity
- > @Id + GeneratedValue
- > @Table
- > @Column
- > @Transient (non persistent field such as age which is calculated using dob)
- > @Temporal (Date)
- > @Enumerated(to store enum)

-> **fetchType**

-> **cascade**

Java classes map to database tables

Java instances map to database rows

Java fields map to database columns

Relational Entities in JPA

@oneToOne

@OneToMany(MAPPED BY FOR BIDIRECTIONAL RELATIONAL)

@ManyToMany

@ManyToOne

Example:

In student class:

@OneToMany

Private List<Laptops> laptops;

In laptop class:

@ManyToOne

Private Student student;

JPA -> for managing relational data

it follows ORM which has Entitymanager for processing queries and transactions

JPA

-> for mapping relation data

-> does not provide implementation

-> platform independent

-> uses Entity Manager

-> for accessing, persisting and managing data b/w objects or classes and a relational database

Hibernate

-> ORM framework

-> provide implementation of classes

-> use session for handling persistence

JPA Persistence

4 stages

Transient(in local memory)

Managed(after persist/save control given to JPA)

Detached(after managed in case of merge/update/ detach/close)

Removed(after managed in case of persist/save/remove)

From managed and removed if flush is called it is moved to DB

What is the difference between JPA and Hibernate? Answehr: JPA is a specification that defines a set of interfaces and annotations for ORM in Java, while Hibernate is an ORM framework that implements the JPA specification. JPA provides a standard programming model, allowing developers to switch between different ORM providers, including Hibernate, without changing their code.

How do POJOs and JavaBeans relate to persistence? Answer: POJOs and JavaBeans serve as the foundation for implementing persistence in Java applications. They represent the domain objects that need to be persisted in a database. By following specific conventions and design patterns, such as providing getter and setter methods and adhering to naming conventions, POJOs and JavaBeans can be easily mapped to database tables using ORM frameworks like Hibernate or JPA.

How does the Java Persistence Ecosystem integrate with frameworks like Spring?

Answer: The Java Persistence Ecosystem integrates with frameworks like Spring through the use of annotations and dependency injection. Frameworks like Spring provide support for integrating JPA and Hibernate seamlessly into applications, managing transactions, and providing additional features like declarative transaction management and dependency injection.

JOB Runnr

Distributed scheduler - if you want to run job on multiple nodes

Then it guarantees that only one instance is executing the job

Normally in spring multiple nodes and they are running jobs but if we want one instance of some job we need jobrunr

We use storage to know which keeps track of who is the primary node which is running and who is the secondary node which wants to run.

Using Java

```
@Component
class JobFactory(private val scheduler: JobScheduler) {

    @PostConstruct
    fun schedule() {
        scheduler.enqueue {
            println("Fire & Forget")
        }

        scheduler.schedule(Instant.now().plusSeconds(30)) {
            println("Scheduled")
        }

        scheduler.scheduleRecurrently(Cron.daily()) {
            println("Daily")
        }
    }
}
```

Using Spring

```
17 @Job(name = "Sync")
18 @Recurring(id = "sync", cron = "0/5 * * * *", zoneId = "Europe/Berlin")
19 fun sync() {
20     log.info("Syncing..")
21 }
22
23 @Job(name = "Backup")
24 @Recurring(id = "backup", cron = "0/5 * * * *", zoneId = "Europe/Berlin")
25 fun backup(context: JobContext) {
26     log.info("Backing up..")
27     val bar = context.progressBar(100)
28     for (i in 1..4) {
29         TimeUnit.SECONDS.sleep(5)
30         bar.setValue(i * 25)
31     }
32 }
```

h and jabran is doing the same thing

Spring Boot Day 4

Key Components and Internals of Spring Boot

- **SpringBoot starter**
 - Simplify the setup and configuration
 - Handling dependency management
 - Provide pre-configured set of dependencies
 - Help to avoid manual configuration and reduce boilerplate code we can include starters as dependency in pom.xml
- **SpringBoot AutoConfigurator**
 - It scan the classpath, check what are the dependencies which is needed and should be enabled and disabled according to conditions
 - It eliminates the need for explicit configuration. Automatically configures applications based on dependency in path.
- **Spring Boot CLI**
 - Command line tool let you run groovy scripts(a script which we can use to bootstrap a complete web application)
 - It allows developers to develop and test spring boot applications quickly.
 - We can quickly prototype and run applications without complex project setup.
- **SpringBoot Actuator**
 - Provides production ready feature to monitor and manage application
 - Include endpoints such as health, metrics, logging bean and more

Getting started with Hibernate:

It is an ORM(object relational mapping) framework.

We need connectors to connect database and application as in application we have classes and in database we have tables so instead of writing queries we use JDBC.

We have to configure the driver name, database url, username , and password in the config-file so that we can create a Session Factory.

Hibernate Cache

Instead of extracting the same data again and again from the database, after the first time hibernate stores it in the first level cache and if you are in the same session hibernate gives you that data it saved.

There is also a second level cache. What it does is for other than the first session all the other sessions can get the data from the second level cache you are asking the same from different sessions.

Types of Second Level cache

- Ehcache
 - First you need to download from maven repository
 - Also download the jar file
 - By default second level cache is disabled so you have to enable it in xml file and also mention the name of provider
 - Use entity
 - @cachable
 - @cache

- **Getting started with Hibernate:**
 - JDBC
 - JDBC is a Java API that provides a standard way to interact with relational databases.
 - It allows you to connect to a database, execute SQL queries, retrieve and manipulate data, and manage database transactions.
 - Hibernate utilizes JDBC underneath to communicate with the database.
 - **Syntax support, transaction management**
 - Object Relational Mapping (ORM)
 - ORM is a technique that maps object-oriented concepts to relational database structures.
 - It provides a way to store and retrieve Java objects in/from a relational database without writing explicit SQL queries.

- ORM frameworks, such as Hibernate, handle the mapping between objects and database tables, simplifying data persistence and retrieval.
- Supported Databases
 - Hibernate supports a wide range of databases, including popular ones like MySQL, Oracle, PostgreSQL, SQL Server, and more.
- **Hibernate Layered Architecture**
 - **Entity Classes:** These are Java classes that represent entities or objects mapped to database tables.
 - **Session Factory:** It is a central factory for creating Hibernate Sessions, providing a connection to the database and managing the lifecycle of persistent objects.
 - **Session:** A Session is a runtime interface between the Java application and Hibernate. It acts as a transactional context for performing database operations.
 - **Mapping Metadata:** It defines the mapping between Java entities and database tables, specifying how the data is stored and retrieved.
 - **Transaction Management:** Hibernate supports transaction management to ensure data integrity and consistency.
 - **Query Language:** Hibernate Query Language (HQL) allows you to write database-agnostic queries using object-oriented concepts.
 - **Caching:** Hibernate provides caching mechanisms to improve performance by reducing the number of database round-trips.
 - **Transaction Control:** Hibernate supports various transaction control mechanisms, such as manual or declarative transaction management

Architecture

classes-> databases -> configurations

Configuration Object(initialize sessions)

The Configuration object is the first Hibernate object you create in any Hibernate application. It is usually created only once during application initialization. It represents a configuration or properties file required by Hibernate.

The Configuration object provides two keys components

- **Database Connection**

This is handled through one or more configuration files supported by Hibernate. These files are hibernate.properties and hibernate.cfg.xml.
- **Class Mapping Setup**

This component creates the connection between the Java classes and database tables.

SessionFactory Object

Configuration object is used to create a SessionFactory object which in turn configures Hibernate for the application using the supplied configuration file and allows for a Session object to be instantiated. The SessionFactory is a thread safe object and used by all the threads of an application.

The SessionFactory is a heavyweight object. it is usually created during application start up and kept for later use. You would need one SessionFactory object per database using a separate configuration file. So, if you are using multiple databases, then you would have to create multiple SessionFactory objects.

Session Object

A Session is used to get a physical connection with a database. The Session object is lightweight and designed to be instantiated each time an interaction is needed with the database. Persistent objects are saved and retrieved through a Session object.

The session objects should not be kept open for a long time because they are not usually thread safe and they should be created and destroyed as needed.

Transaction Object

A Transaction represents a unit of work with the database and most of the RDBMS supports transaction functionality. Transactions in Hibernate are handled by an underlying transaction manager and transaction (from JDBC or JTA).

This is an optional object and Hibernate applications may choose not to use this interface, instead managing transactions in their own application code.

Query Object

Query objects use SQL or Hibernate Query Language (HQL) string to retrieve data from the database and create objects. A Query instance is used to bind query parameters, limit the number of results returned by the query, and finally to execute the query.

Criteria Object

Criteria objects are used to create and execute object oriented criteria queries to retrieve objects.

Need of JPA

As we have seen so far, JPA is a specification. It provides common prototype and functionality to ORM tools. By implementing the same specification, all ORM tools (like Hibernate, TopLink, iBatis) follow the common standards. In the future, if we want to switch our application from one ORM tool to another, we can do it easily.

JPA vs Hibernate

JPA	Hibernate
Java Persistence API (JPA) defines the management of relational data in the Java applications.	Hibernate is an Object-Relational Mapping (ORM) tool which is used to save the state of Java object into the database.
It is just a specification. Various ORM tools implement it for data persistence.	It is one of the most frequently used JPA implementation.
It is defined in javax.persistence package.	It is defined in org.hibernate package.
The EntityManagerFactory interface is used to interact with the entity manager factory for the persistence unit. Thus, it provides an entity manager.	It uses SessionFactory interface to create Session instances.
It uses EntityManager interface to create, read, and delete operations for instances of mapped entity classes. This interface interacts with the persistence context.	It uses Session interface to create, read, and delete operations for instances of mapped entity classes. It behaves as a runtime interface between a Java application and Hibernate.
It uses Java Persistence Query Language (JPQL) as an object-oriented query language to perform database operations.	It uses Hibernate Query Language (HQL) as an object-oriented query language to perform database operations.

JPA Entities

Entities in JPA are nothing but POJOs representing data that can be persisted in the database. An entity represents a table stored in a database. Every instance of an entity represents a row in the table.

Class with an `@entity` annotation. (Model Class)

```
@Entity
@Table(name = "tbl_employees")
@Data
@AllArgsConstructor
@NoArgsConstructor
public class Employee {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)

    private Long id;

    private String name;
    private String email;
    private String department;

    public void setId(Long id) {
        this.id = id;
    }
    public Long getId() {
        return id;
    }
}
```

```
}  
}
```

JobRunr

JobRunr is an open-source Java library that provides a job scheduling and processing framework for background jobs. It allows you to execute long-running, asynchronous tasks in a distributed and fault-tolerant manner.

JobRunr integrates well with the Spring Framework, providing seamless integration and making it easy to use within Spring-based applications. Here's how you can use JobRunr with Spring

- **Add the JobRunr dependency to your project:** Include the JobRunr dependency in your project's build file. You can use a dependency management tool like Maven or Gradle to do this.
- **Configure JobRunr:** Create a configuration class and annotate it with `@Configuration` to indicate that it is a Spring configuration class. Inside this class, you can define beans and configure JobRunr.
- **Set up a Job Processing Server:** JobRunr uses a background processing server to execute jobs. You need to configure and start a Job Processing Server in your application. You can do this by creating a bean of type `JobServer` and configuring it with the necessary settings.

Auto-configuration

In Spring Boot, auto-configuration is a feature that automatically configures the Spring application based on the dependencies and classpath present in the project. It aims to reduce the amount of manual configuration required and provide sensible defaults for various components, making it easier to get started with Spring applications.

Spring Boot Starter

In Spring Boot, a starter is a dependency that **provides a set of pre-configured dependencies and configurations** for a specific technology or feature. Starters are designed to simplify the dependency management and configuration process, making it easier to bootstrap Spring Boot applications.

Spring Boot starters typically follow the naming convention of `spring-boot-starter-*`, where `*` represents the name of the technology or feature they provide. Some common starters provided by Spring Boot include:

Spring Boot CLI

The Spring Boot CLI (Command-Line Interface) is a **command-line tool provided by Spring Boot that allows you to quickly create, run, and manage Spring Boot applications**. It provides a convenient way to interact with Spring Boot without requiring a full-fledged development environment or build tools like Maven or Gradle.

Spring Boot Actuator

Spring Boot Actuator is a powerful feature of the Spring Boot framework that **provides various production-ready monitoring and management capabilities for your application**. It allows you to gain **insights into your application's runtime behavior, health, metrics, and more**, without requiring additional configuration or dependencies.

Hibernate architecture

The Hibernate architecture is categorized in four layers.

- Java application layer
- Hibernate framework layer
- Backend api layer
- Database layer

Spring Boot Day 5-6

Querying with Spring Boot Data

In order to define SQL to execute for a Spring Data repository method, we can annotate the method with the `@Query` annotation — its value attribute contains the JPQL or SQL to execute.

The `@Query` annotation takes precedence over named queries, which are annotated with `@NamedQuery` or defined in an `pom.xml` file.

It's a good approach to place a query definition just above the method inside the repository rather than inside our domain model as named queries. The repository is responsible for persistence, so it's a better place to store these definitions.

Basic Auto-configuration - Datasource and Pooling

Spring Boot uses an opinionated algorithm to scan for and configure a `DataSource`. This allows us to easily get a fully-configured `DataSource` implementation by default.

In addition, Spring Boot automatically configures a lightning-fast connection pool, either HikariCP, Apache Tomcat, or Commons DBCP, in that order, depending on which are on the classpath.

While Spring Boot's automatic `DataSource` configuration works very well in most cases, sometimes we'll need a higher level of control, so we'll have to set up our own `DataSource` implementation, hence skipping the automatic configuration process.

DispatcherServlet

Simply put, in the Front Controller design pattern, a single controller is responsible for directing incoming `HttpRequests` to all of an application's other controllers and handlers.

Spring's `DispatcherServlet` implements this pattern and is, therefore, responsible for correctly coordinating the `HttpRequests` to their right handlers.

It has

1. Handler Mapping(DS gets an entry of handler mapping from the XML file and forwards request to the controller)
2. Controller(It takes requests and calls service methods based on GET or POST methods. Service has all the business logic)
3. View Resolver(pick up the defined view for the request)
4. View(model data to view which is rendered on browser)

Datasource:

A datasource is an object that provides a connection to the database. It contains the necessary information to establish a database connection, such as the URL, username, password, and other configuration properties.

In a Spring Boot application, you typically configure the datasource in the `application.properties` or `application.yml` file, specifying the appropriate properties for your database.

Connection Pooling:

Connection pooling is a technique used to manage and reuse database connections, improving application performance and reducing the overhead of creating new connections for each database operation.

With connection pooling, a pool of pre-initialized database connections is created and maintained by the datasource. When a database connection is required, a connection is borrowed from the pool. After the operation is complete, the connection is returned to the pool instead of being closed.

Spring Boot Data:

1. Spring Boot Data provides a unified and streamlined approach to interact with different data sources in a Spring Boot application.
2. It eliminates much of the boilerplate code and configuration required for data access, allowing developers to focus on business logic rather than low-level data access details.
3. Spring Boot Data supports various data sources, including relational databases, NoSQL databases, in-memory databases, and more.
4. It provides consistent APIs and abstractions for working with different data sources, making it easier to switch between databases or data access technologies.
5. Spring Boot Data integrates seamlessly with other Spring modules, such as Spring MVC and Spring Security, to provide a cohesive development experience.

Data Source Configuration
Repository Interface
Automatic Implementation
Auto-Configuration
Customization
Integration with Spring Boot

Spring Boot Data JPA:

Spring Boot JPA Data simplifies database access and data manipulation in Java applications. It promotes productivity by reducing the amount of manual configuration and repetitive code, enabling developers to focus on the business logic of their applications.

6. Spring Boot Data JPA builds upon Spring Data JPA, which is a subproject of the larger Spring Data framework.
7. It simplifies the development of JPA (Java Persistence API) repositories and entities in Spring Boot applications.
8. Spring Boot Data JPA provides auto-configuration and default behavior for JPA repositories, reducing the amount of manual configuration required.
9. It offers powerful features such as automatic CRUD (Create, Read, Update, Delete) operations, query generation, pagination, and more.
10. Spring Boot Data JPA supports different JPA implementations, including Hibernate, EclipseLink, and OpenJPA.
11. It promotes the use of JPA annotations and conventions to define entities and their relationships, simplifying the mapping between Java objects and database tables.

Spring Boot JPA Data provides the following features and benefits:

- **Entity Mapping:** With Spring Boot JPA Data, you can define entity classes that represent your domain objects. These entities are annotated with JPA annotations, such as `@Entity`, `@Table`, and `@Column`, to specify the mapping between the Java objects and the database tables.
- **Repository Abstraction:** Spring Boot JPA Data introduces the concept of repositories, which provide a high-level abstraction for working with data. Repositories are interfaces that extend the `JpaRepository` or related interfaces provided by Spring Data JPA. They offer a set of predefined methods for common CRUD operations (Create, Read, Update, Delete), as well as support for paginated queries and sorting.
- **Query Methods:** Spring Boot JPA Data allows you to define query methods in repository interfaces by following a naming convention. By simply defining a method with a specific naming pattern, such as `findByFirstName(String firstName)`, the framework automatically generates the corresponding query and executes it against the underlying database. This eliminates the need for writing explicit SQL queries for common data retrieval tasks.

- **Pagination and Sorting:** Spring Boot JPA Data provides built-in support for pagination and sorting of query results. By passing Pageable or Sort objects as method parameters in query methods, you can easily control the number of results to fetch and the order in which they are returned.
- **Transaction Management:** Spring Boot JPA Data integrates seamlessly with Spring's transaction management capabilities. It automatically manages transactions for database operations, ensuring consistency and data integrity. Transactions can be declaratively managed using annotations such as `@Transactional`.
- **Automatic Query Generation:** Spring Boot JPA Data can automatically generate SQL queries based on the structure of your entity classes and their relationships. It handles common database operations like fetching related entities, lazy loading, and cascading updates or deletions. This saves developers from writing complex SQL queries and reduces the amount of boilerplate code.
- **Database Agnostic:** Spring Boot JPA Data supports various databases, including popular relational databases like MySQL, PostgreSQL, Oracle, and SQL Server. It provides a consistent programming model that abstracts away the underlying database details, allowing you to switch databases easily without changing your application code.

Pageable

Pageable is an interface provided by the Spring Framework that represents a request for a specific page of data in a query result. It is commonly used in combination with pagination features in Spring Data repositories to control the number of results returned from a query and specify the sorting order.

The Pageable interface defines methods for retrieving information about pagination, such as the page number, page size, and sorting options. Here are some key methods provided by the Pageable interface:

- **`int getPageNumber()`:** Returns the zero-based index of the requested page.
- **`int getPageSize()`:** Returns the maximum number of items to be returned per page.
- **`int getOffset()`:** Returns the offset of the first item to be returned from the result set. This is calculated based on the page number and page size.
- **`Sort getSort()`:** Returns the sorting options applied to the query result. The Sort object represents the properties and directions used for sorting.
- **`Pageable next()`:** Returns a new Pageable object representing the next page. This is useful for implementing pagination controls in an application.

- **Pageable previousOrFirst():** Returns a new Pageable object representing the previous page or the first page if the current page is the first page.
- **boolean hasPrevious():** Indicates whether there is a previous page available.

The Pageable interface is commonly used as a method parameter in Spring Data repository methods that support pagination. By passing a Pageable object to these methods, you can control the number of results returned and the order in which they are

Page

In the context of pagination, a "**page**" refers to a **subset of data that is retrieved from a larger result set**. It represents a partition of the complete set of data into smaller, manageable portions.

When working with large amounts of data, it is often impractical or inefficient to retrieve the entire dataset at once. Instead, data is fetched and presented to the user in smaller chunks or pages, allowing for better performance and usability.

Each page typically contains a fixed number of items or records, determined by the page size. For example, if the page size is set to 10, each page will contain 10 records.

By dividing the data into pages, users can navigate through the result set, viewing and interacting with one page at a time. Pagination controls, such as next page, previous page, and specific page number, are commonly provided to facilitate navigation.

For instance, consider a scenario where you have a database table with 100 records and a page size of 20. The data would be divided into five pages, with each page containing 20 records. The user can then navigate through these pages to view different subsets of the data.

Pagination is particularly useful when working with web applications, where displaying a large amount of data in a single page can negatively impact performance and user experience. By fetching and presenting data in smaller, more manageable chunks, users can interact with the application more efficiently.

Implementing pagination often involves specifying the page number and page size when making a query or fetching data, which can be achieved using the Pageable interface in Spring Boot or similar constructs in other frameworks.

Auto-configuration

Auto-configuration is a key feature of Spring Boot that aims to simplify the setup and configuration of Spring applications by **automatically configuring beans and components based on the classpath and the dependencies present in the project**. It allows developers to focus on writing business logic rather than spending time on manual configuration.

In Spring Boot, basic auto-configuration refers to the automatic configuration of fundamental components that are commonly required in most Spring applications. These components include the application context, web server, database connections, and various other features. Here are some examples of basic auto-configuration provided by Spring Boot:

- Application Context
- Web Server:
- Database Connections
- Logging
- Error Handling
- Spring MVC

DispatcherServlet

The DispatcherServlet is a central component for **handling HTTP requests** and **managing the request/response lifecycle in web applications**. It acts as the **front controller**, receiving incoming requests and **dispatching them to the appropriate handlers for processing**.

The DispatcherServlet is automatically configured by Spring Boot when you include the necessary dependencies for web development. Here's how it works:

- **Request Handling:** When an HTTP request is received by the web server, it is directed to the DispatcherServlet by default. The DispatcherServlet is mapped to a specific URL pattern (e.g., "/"), and any requests matching that pattern are handled by the servlet.
- **Request Routing:** The DispatcherServlet examines the request URL and determines which controller or handler should process the request. It consults the application's request mappings, which are defined using annotations such as `@RequestMapping`, `@GetMapping`, `@PostMapping`, and others, to map URLs to controller methods.
- **Handler Execution:** Once the DispatcherServlet has determined the appropriate handler (controller method), it invokes the method to perform the actual request processing. The method can access the request parameters, headers, and body, and perform any necessary business logic. The result of the handler method's execution is typically a model or a view that represents the response data.
- **View Resolution:** After the handler method has finished its execution, the DispatcherServlet processes the handler's result, which can be a view name or a model object. It then resolves the view, typically a template file such as a JSP or Thymeleaf template, to generate the final response to be sent back to the client.
- **Response Handling:** Once the view has been resolved and rendered, the DispatcherServlet constructs the response object, including the appropriate headers and body. It sends the response back to the client through the web server.

Spring Boot Data REST

Spring Boot Data REST is a module of the Spring Boot framework that provides a convenient way to expose RESTful APIs for your data repositories. It allows you to quickly create a RESTful API layer on top of your Spring Data repositories without writing explicit controller code.

By leveraging Spring Boot Data REST, you can automatically expose your repository interfaces as RESTful endpoints with support for common CRUD operations (Create, Read, Update,

Delete), querying, and pagination. Here are some key features and concepts of Spring Boot Data REST:

- **Repository Exposé:** Spring Boot Data REST automatically detects your Spring Data repositories and exposes them as RESTful endpoints. Each repository interface is mapped to a corresponding RESTful resource, allowing you to perform CRUD operations on entities.
- **RESTful Endpoints:** For each repository, Spring Boot Data REST generates a set of standard RESTful endpoints following REST conventions. These endpoints allow you to perform operations such as retrieving entities by ID, searching entities using query methods, creating new entities, updating existing entities, and deleting entities.
- **HATEOAS Support:** Spring Boot Data REST follows the principles of HATEOAS (Hypermedia as the Engine of Application State), which means that the responses from the RESTful API include hypermedia links that guide clients to related resources and actions. HATEOAS enables self-discoverable APIs and simplifies client integration and navigation.
- **Querying and Filtering:** Spring Boot Data REST supports a variety of ways to query and filter data. It allows you to perform searches using query methods in repository interfaces, query by example, or even create custom queries using the `@Query` annotation. You can also apply sorting and pagination to query results.
- **Resource Projection:** Spring Boot Data REST provides the ability to customize the shape and content of API responses using resource projections. Resource projections allow you to select specific fields to include or exclude from the response, create nested representations, and define custom views of your data.
- **Event Handling:** Spring Boot Data REST offers event handling mechanisms to intercept and react to different events occurring during the REST API lifecycle. You can use events to perform additional business logic, execute custom validations, or integrate with other parts of your application.
- **Security and Access Control:** Spring Boot Data REST integrates seamlessly with Spring Security to provide security and access control for your RESTful APIs. You can secure endpoints based on roles, define custom authorization rules, and control access to resources and operations.

Security

authentication (who are you?)

authorization (what are you allowed to do?)

An AuthenticationManager can do one of 3 things in its `authenticate()` method:

- Return an Authentication (normally with `authenticated=true`) if it can verify that the input represents a valid principal.
- Throw an `AuthenticationException` if it believes that the input represents an invalid principal.
- Return null if it cannot decide.
- **Form-Based Authentication:**

- Form-based authentication is a widely used authentication mechanism in web applications.
- It involves presenting a login form to the user, collecting the username and password, and verifying them against a user database or authentication provider.
- Spring Security, the security module of Spring Boot, provides built-in support for form-based authentication.
- It handles the process of validating user credentials, managing authentication sessions, and redirecting users to the appropriate pages based on their authentication status.

- **HTTP Basic Authentication:**

- HTTP Basic Authentication is a simple authentication scheme that involves sending the username and password as part of the HTTP request headers.
- The server verifies the credentials and grants access if they are valid.
- Spring Security supports HTTP Basic Authentication out of the box.
- It automatically prompts the user for credentials when accessing secured endpoints and validates the provided credentials against the configured authentication provider.

- **HTTP Digest Authentication:**

- HTTP Digest Authentication is an enhanced version of HTTP Basic Authentication that provides an additional layer of security.
- It involves sending hashed values of the username, password, and other parameters along with the request.
- Spring Security supports HTTP Digest Authentication and handles the process of validating the digest and granting access to protected resources.

- **JSON Web Token (JWT) Authentication:**

- JSON Web Token (JWT) is a stateless authentication mechanism that allows authentication information to be securely transmitted between parties as a JSON object.
- It involves generating a token upon successful authentication, which is then sent with subsequent requests to authenticate the user.
- Spring Security provides support for JWT authentication through various libraries and extensions.
- It allows you to configure token-based authentication, validate and decode JWT tokens, and authenticate users based on the information contained in the token.

- **OAuth2 Authentication:**

- OAuth2 is an open standard for authentication and authorization that enables users to grant access to their resources without sharing their credentials.
- Spring Security supports OAuth2 authentication and provides mechanisms to integrate with OAuth2 providers, such as Google, Facebook, or custom OAuth2 servers.
- It handles the OAuth2 flow, including token exchange, authorization, and user authentication.

Not use websecurityconfigurationAdapter

Spring Boot Day 7

Hibernate is the default ORM (Object-Relational Mapping) framework used for database access. Hibernate provides caching mechanisms to improve the performance of database operations by reducing the number of round trips to the database. Spring Boot provides integration with Hibernate caching and offers several caching strategies that can be used to optimize database access.

Here are some key points regarding Hibernate caching in Spring Boot:

- **First Level Cache (Session Cache):** Hibernate has a built-in first level cache, also known as the session cache. It is enabled by default and operates within the boundaries of a **single database session**. The first level cache stores the entities fetched from the database during a session, allowing subsequent requests for the same entity to be served from the cache rather than hitting the database again. This cache is scoped to the current session and is cleared automatically when the session is closed.
- **Second Level Cache:** In addition to the first level cache, Hibernate provides a second level cache, which is a shared cache that can be used across multiple sessions. The second level cache holds entities, collections, and query results in a shared cache region. It can be configured to use various caching providers such as Ehcache, Hazelcast, Infinispan, or a distributed cache like Redis. By enabling and configuring the second level cache, you can improve performance by reducing database access even across multiple sessions.
- **Caching Strategies:** Hibernate offers different caching strategies that determine how entities and query results are cached. Some common caching strategies include:
 - **Read-Write:** This strategy caches both read and write operations, providing the highest level of caching. It keeps the cache in sync with the database by updating or invalidating cached entries when changes occur.
 - **Read-Only:** This strategy caches read operations only and assumes that the data does not change. It provides better performance for read-heavy applications.
 - **Nonstrict Read-Write:** This strategy allows read operations to be cached but does not update the cache when write operations occur. It is suitable for scenarios where read operations greatly outnumber write operations.
 - **Transactional:** This strategy caches data within the scope of a transaction, ensuring consistency within the transaction boundaries.
- **Query Cache:**
- **Cache Configuration:** In a Spring Boot application, Hibernate caching can be configured using properties in the application.properties or application.yml file. You can specify the cache provider, cache region names, cache expiration policies, and other

caching-related settings. Additionally, you can use annotations such as `@Cacheable` and `@CachePut` to control caching at the entity or method level.

load() vs get()

load() method:

- The `load()` method is used to lazily fetch an object from the database. It returns a proxy object (a lightweight placeholder) that is initialized with the actual data when accessed for the first time.
- If the requested object doesn't exist in the database, a `HibernateException` may be thrown when you access the object's properties or methods.
- The `load()` method may return a proxy object without hitting the database immediately, which can be useful when you want to delay the database retrieval until necessary.
- If the identifier is not found in the database and you attempt to access properties or methods of the proxy object, a `ObjectNotFoundException` will be thrown.
- The `load()` method is more efficient when it comes to lazy loading and retrieving objects by identifier.

get() method:

- The `get()` method is used to eagerly fetch an object from the database. It returns the actual object if found, or null if not found.
- If the requested object doesn't exist in the database, the `get()` method returns null.
- The `get()` method immediately hits the database and retrieves the object's data.
- The `get()` method is straightforward and convenient to use when you want to retrieve an object and handle the case when it doesn't exist in the database.

update() vs merge()

update() method:

- The `update()` method is used to save the object in the database. We can use the `update()` method when the Session does not contain any persistent object with the same primary key. If the Session has a persistent object with the same primary key, it will throw an exception.
- This method adds an entity object to the persistent state, and more changes are tracked and saved when the transaction is committed. The `update()` method does not return anything. The `update()` method is provided by the Session interface and is available in `org.hibernate.session` package.

merge() method:

The `merge()` method is used when we want to change a detached entity into the persistent state again, and it will automatically update the database. The main aim of the `merge()` method is to update the changes in the database made by the persistent object.

In summary, the `update()` method is used to reattach a detached object to the session and synchronize its state with the corresponding persistent object. It requires the persistent object to be already present in the session cache. On the other hand, the `merge()` method merges the state of a detached object with a new or existing persistent object, and it creates a new persistent object if necessary.

Object States in Hibernate

Transient State: An object is in the transient state when it is not associated with any Hibernate session. It is not managed by Hibernate and has no connection to the database. Transient objects are not persisted, and changes made to them are not automatically synchronized with the database.

Persistent State: When an object is associated with an active Hibernate session, it enters the persistent state. In this state, Hibernate tracks changes made to the object's properties, and those changes can be persisted to the database. Persistent objects are also known as "managed" objects. Hibernate manages the persistence of persistent objects and ensures that changes are synchronized with the database.

Detached State: An object becomes detached when it was previously associated with a Hibernate session but is no longer connected to an active session. In the detached state, changes made to the object are not automatically tracked by Hibernate, and they are not automatically persisted to the database. Detached objects can be reattached to a new session for further modification and persistence.

Removed/Deleted State: When a persistent object is explicitly marked for deletion using the `delete()` or `remove()` method, or when the persistent object is removed from a collection association, it enters the removed/deleted state. In this state, the object is scheduled for deletion from the database. The deletion will be executed when the Hibernate session is flushed or the transaction is committed.

Immutable Entity Class in Hibernate

In Hibernate, an immutable entity class refers to a class whose state cannot be modified once it is persisted. Immutable entities are read-only and do not allow any changes to their properties after they are created and persisted to the database. This immutability can be achieved by following certain guidelines and annotations in Hibernate.

Here are the key aspects of creating an immutable entity class in Hibernate:

- **Class Definition:** Define your entity class using the `@Entity` annotation to mark it as an entity in Hibernate. Make sure to provide a public no-argument constructor or a constructor that initializes all the properties.
- **Property Mapping:** Map the properties of the entity class using appropriate annotations such as `@Id`, `@Column`, and `@GeneratedValue` as necessary. These annotations define the primary key, column names, and other mapping details.
- **Immutable Properties:** Declare the properties of the entity class as final, ensuring that they cannot be modified once the object is created. Alternatively, you can make the properties private and provide only getter methods without corresponding setter methods.
- **Entity Identity:** Ensure that the identity of the entity (e.g., primary key) is assigned and immutable. Use appropriate strategies such as assigning a fixed value, using natural keys, or using identifier generation mechanisms that generate immutable values.
- **Collection Properties:** If your entity has collection properties (e.g., a collection of related entities), ensure that the collection is initialized with an immutable implementation such as `Collections.unmodifiableList()` or `Collections.unmodifiableSet()`.
- **Transaction Management:** When working with immutable entities, it's important to avoid modifying them within Hibernate transactions. Transactions should be read-only, preventing any changes to the immutable entities during the transaction's scope.

Dirty read, Phantom Read and Non Repeatable Read

Dirty Read: A dirty read occurs when one transaction reads data that has been modified by another transaction that has not yet been committed. In other words, a transaction reads uncommitted data from another transaction. In other words, the transaction that made the modification is later rolled back, the data read by the first transaction becomes invalid or incorrect. Dirty reads can lead to incorrect and inconsistent results.

Phantom Read: A phantom read occurs when a transaction retrieves a set of records based on a certain condition, and another transaction inserts or deletes records that satisfy the same condition before the first transaction completes. As a result, when the first transaction tries to retrieve the same set of records again, it observes different records, as if "phantoms" appeared or disappeared. Phantom reads can lead to non-deterministic behavior and inconsistency in query results.

Non-Repeatable Read: A non-repeatable read occurs when a transaction reads the same record multiple times within its own transaction but observes different values each time. This inconsistency arises when another transaction modifies the record between the reads, resulting in different values being seen in subsequent reads. Non-repeatable reads can lead to unexpected and inconsistent query results within the same transaction.

Isolation and Propagation

Isolation: Isolation refers to the degree of isolation and separation between concurrent transactions accessing the same data. It ensures that each transaction sees a consistent snapshot of the database, regardless of other concurrent transactions.

Common isolation levels defined by the ANSI/ISO SQL standard are:

- **Read Uncommitted:** Allows dirty reads, non-repeatable reads, and phantom reads.
- **Read Committed:** Allows non-repeatable reads and phantom reads.
- **Repeatable Read:** Prevents dirty reads and non-repeatable reads, but allows phantom reads.
- **Serializable:** Provides the highest level of isolation, preventing dirty reads, non-repeatable reads, and phantom reads.

Isolation levels control aspects such as visibility of uncommitted data, handling of concurrent modifications, and behavior regarding locks and data consistency.

Propagation: Propagation describes how a transaction propagates across multiple method invocations or components. It defines how transactions are managed when methods or components call each other and need to participate in the same transaction context.

Common propagation options include:

- **Required:** The method or component must run within an existing transaction. If no transaction exists, a new one is created.
- **Requires New:** The method or component always runs in a new, separate transaction. If a transaction exists, it is suspended.
- **Mandatory:** The method or component must run within an existing transaction. If no transaction exists, an exception is thrown.
- **Not Supported:** The method or component executes outside any transaction context. If a transaction exists, it is suspended.

Propagation options define how transactional behavior is inherited or established across different layers or components in an application.

Cascading

In the context of Spring, cascading refers to a mechanism for propagating changes or actions from one object to another in a related or associated manner. It is commonly used in object-relational mapping (ORM) frameworks like Hibernate to manage the persistence of related entities.

In a cascading scenario, **when an operation is performed on a particular object (referred to as the source object), the same operation is automatically applied to other related objects (referred to as target objects)**. These related objects are typically associated with the source object through some kind of relationship, **such as a one-to-one, one-to-many, or many-to-many** relationship.

The cascading behavior in Spring can be configured using **annotations such as @Cascade or through XML configuration**. The cascading options define the specific operations to be cascaded, such as saving, updating, or deleting, and can be customized based on the needs of the application.

For example, let's consider a scenario where you have two entities: Author and Book, with a one-to-many relationship where an author can have multiple books. If cascading is enabled for the Author entity's books field, operations performed on the Author object (e.g., saving or deleting) will automatically cascade to the associated Book objects.

Here's an example using JPA annotations:

```
java Copy code  
  
@Entity  
public class Author {  
    // ...  
  
    @OneToOne(mappedBy = "author", cascade = CascadeType.ALL)  
    private List<Book> books;  
  
    // ...  
}  
  
@Entity  
public class Book {  
    // ...  
  
    @ManyToOne  
    private Author author;  
  
    // ...  
}
```

Cascade Types

- **CascadeType.ALL:** This cascade type indicates that all operations (including persist, merge, remove, and refresh) should be cascaded to the related entities. It's a shorthand for specifying all possible cascade types.

- **CascadeType.PERSIST:** This cascade type indicates that the persist operation should be cascaded. It means that when a new entity is persisted (saved) or associated with a persistent entity, the persistence operation should be propagated to the related entities.
- **CascadeType.MERGE:** This cascade type indicates that the merge operation should be cascaded. It means that when changes are made to a detached entity (an entity that is not being managed by the ORM framework), the changes should be merged into the related entities.
- **CascadeType.REMOVE:** This cascade type indicates that the remove operation should be cascaded. It means that when an entity is removed (deleted), the remove operation should be propagated to the related entities.
- **CascadeType.REFRESH:** This cascade type indicates that the refresh operation should be cascaded. It means that when an entity is refreshed (reloaded from the database), the refresh operation should be propagated to the related entities.
- **CascadeType.DETACH:** This cascade type indicates that the detach operation should be cascaded. It means that when an entity is detached from the persistence context, the detach operation should be propagated to the related entities.

@Entity vs @Table

@Entity:

- The **@Entity** annotation is used to mark a Java class as an entity, indicating that instances of this class will be persisted in the database. It is typically placed on the class declaration.
- When an entity class is annotated with **@Entity**, it signifies that the class is a persistent entity, and its instances will be managed by the ORM framework. Each entity instance represents a row in the corresponding database table.
- In addition to marking the class as an entity, the **@Entity** annotation provides additional options and attributes to configure the entity's behavior, such as specifying the table name, defining primary key generation strategies, mapping inheritance hierarchies, and more.

@Table:

- The **@Table** annotation is used to define the details of the database table to which an entity is mapped. It is typically placed on the entity class and is used in conjunction with the **@Entity** annotation.
- By default, the name of the table in the database is assumed to be the same as the name of the entity class. However, the **@Table** annotation allows you to explicitly specify the table name and provides additional options to customize the table's schema, catalog, and other attributes.

- The `@Table` annotation provides flexibility in mapping entities to existing database tables with different names or structures, and it allows you to define unique constraints, indexes, and other database-specific properties.

In summary, `@Entity` is used to mark a Java class as a persistent entity, while `@Table` is used to customize the mapping details of the corresponding database table. `@Entity` is a higher-level annotation that encompasses the entire entity, whereas `@Table` is a more specific annotation used to fine-tune the table mapping properties.

PlatformTransactionManager

`PlatformTransactionManager` is an interface that defines the contract for managing transactions in a Spring application. It provides a consistent API for transaction management across different transactional resources and technologies.

The `PlatformTransactionManager` interface is designed to be independent of any specific transaction management implementation. It acts as an abstraction layer that allows Spring to work with different transactional resources, such as databases, JMS providers, JPA implementations, and more, without being tightly coupled to a particular technology or vendor. The `PlatformTransactionManager` interface defines several methods, including:

- **getTransaction:** This method is responsible for creating a new transaction or retrieving an existing transaction associated with the current context.
- **commit:** This method is used to commit a transaction, making all changes within the transaction permanent.
- **rollback:** This method is used to roll back a transaction, discarding any changes made within the transaction.
- **setRollbackOnly:** This method marks the current transaction as rollback-only, indicating that it should be rolled back instead of committed.

Different implementations of `PlatformTransactionManager` are available in Spring to support various transaction management strategies. Some commonly used implementations include:

- **DataSourceTransactionManager:** This implementation is used for managing transactions in applications that use JDBC and interact with a relational database.
- **JtaTransactionManager:** This implementation is used for managing transactions in distributed environments that support the Java Transaction API (JTA), such as when working with multiple transactional resources or across multiple application servers.
- **JpaTransactionManager:** This implementation is specifically tailored for managing transactions in applications that use Java Persistence API (JPA) for object-relational mapping.

- **HibernateTransactionManager:** This implementation is similar to JpaTransactionManager but is specifically designed for applications that use Hibernate as the ORM framework.

JPA Auditing

JPA auditing is a feature provided by the Java Persistence API (JPA) that allows automatic tracking and management of auditing information for entities. It enables the recording of metadata such as creation timestamp, modification timestamp, and the user responsible for the changes, without requiring explicit manual handling in each entity operation.

Enable JPA auditing in your Spring configuration: In your Spring configuration class, annotate it with `@EnableJpaAuditing` to enable JPA auditing support.

```
@Configuration
@EnableJpaAuditing
public class JpaConfig {
    // Configuration code...
}
```

Annotate audited entities: In your entity classes that require auditing, annotate them with `@EntityListeners(AuditingEntityListener.class)` to enable auditing support for those entities.

```
@Entity
@EntityListeners(AuditingEntityListener.class)
public class User {
    // Entity fields and mappings...
}
```

Add audit fields to the entities: Declare the audit fields in your entity classes to store the auditing information. These fields will be automatically populated by JPA with the appropriate values during entity lifecycle events.

```

@Entity
@EntityListeners(AuditingEntityListener.class)
public class User {
    @Id
    private Long id;

    // Other entity fields...

    @CreatedBy
    private String createdBy;

    @CreatedDate
    private LocalDateTime createdDate;

    @LastModifiedBy
    private String lastModifiedBy;

    @LastModifiedDate
    private LocalDateTime lastModifiedDate;

    // Getters and setters...
}

```

In this example, `createdBy` and `lastModifiedBy` fields store the user responsible for the creation and modification of the entity, respectively. `createdDate` and `lastModifiedDate` fields store the timestamps of creation and modification, respectively.

With the above configuration, JPA will automatically populate these audit fields during entity lifecycle events. When you save a new entity, JPA will set the `createdBy`, `createdDate`, `lastModifiedBy`, and `lastModifiedDate` fields. When you update an existing entity, JPA will update the `lastModifiedBy` and `lastModifiedDate` fields.

Remember to configure an appropriate mechanism for resolving the user information for `createdBy` and `lastModifiedBy` fields. This can be done by implementing `AuditorAware` interface to provide the currently logged-in user or any custom logic to determine the user information.

FindByID() vs GetOne()

findById():

- findById() is a method provided by the CrudRepository or JpaRepository interfaces in Spring Data JPA.
- It returns an Optional object that may contain the entity matching the given ID.
- If no entity is found with the specified ID, findById() returns an empty Optional.
- findById() retrieves the entity from the database immediately when the method is called.

```
Optional<User> userOptional = userRepository.findById(userId);
if (userOptional.isPresent()) {
    User user = userOptional.get();
    // Process the retrieved user entity
} else {
    // Handle the case when the entity is not found
}
```

getOne():

- getOne() is a method provided by the JpaRepository interface in Spring Data JPA.
- It returns a proxy object representing the entity with the given ID.
- If no entity is found with the specified ID, a EntityNotFoundException is thrown when accessing properties or methods of the proxy object.
- getOne() does not immediately retrieve the entity from the database. Instead, it creates a reference (proxy) to the entity and loads it from the database only when necessary (e.g., when accessing its properties or invoking methods).

```
User user = userRepository.getOne(userId);
// Access properties or invoke methods on the user entity
```

Entity Manager

- Hibernate provides implementation of JPA interfaces EntityManagerFactory and EntityManager.
- EntityManagerFactory provides instances of EntityManager for connecting to the same database. All the instances are configured to use the same setting as defined by the default implementation. Several entity manager factories can be prepared for connecting to different data stores.

- JPA EntityManager is used to access a database in a particular application. It is used to manage persistent entity instances, to find entities by their primary key identity, and to query over all entities.

EntityManager is an interface provided by JPA that allows for the management and interaction with entities, including persisting, retrieving, and executing queries.

Fetch

In Spring, the term "fetch" refers to the strategy used to retrieve associated data when querying entities in an object-relational mapping (ORM) context. It determines how related entities or collections are loaded from the database along with the main entity being fetched.

There are different fetch strategies available in Spring for defining how associated data should be fetched. These strategies can be specified using annotations or query hints, depending on the ORM framework you are using (such as Hibernate) and the specific JPA implementation.

Here are some commonly used fetch strategies:

Eager Fetching:

- Eager fetching is a strategy where associated data is loaded immediately and fully when the main entity is fetched from the database.
- With eager fetching, related entities or collections are retrieved together with the main entity using join queries or additional queries. This means that all the required data is available in memory without the need for further database access.
- Eager fetching can be specified using the fetch attribute of JPA annotations such as @OneToOne, @OneToMany, and @ManyToOne

```
@OneToMany(fetch = FetchType.EAGER)
private List<OrderItem> items;
```

Lazy Fetching:

- Lazy fetching is a strategy where associated data is loaded on-demand, only when it is explicitly accessed or needed.
- With lazy fetching, related entities or collections are not automatically loaded when the main entity is fetched. Instead, a proxy or placeholder object is created, and the associated data is loaded from the database when accessed.
- Lazy fetching is the default fetch strategy for most JPA associations. It helps to optimize performance by loading only the necessary data and avoiding unnecessary database queries.
- Lazy fetching can be specified using the fetch attribute with the value FetchType.LAZY

```
@ManyToOne(fetch = FetchType.LAZY)
private User user;
```

Cascade and Fetch Together

Yes, it is possible to use cascade and fetch together in Spring Data JPA. The cascade options control how changes to one entity are cascaded to related entities, while fetch strategies determine how associated entities are loaded during querying

```
@Entity
public class Author
{ // ... @OneToMany(mappedBy = "author", cascade = CascadeType.ALL, fetch
= private List<Book> books;}
@Entity
public class Book { // ... @ManyToOne(fetch = FetchType.EAGER)
private Author author;
}
```

N+1 query problem

The N+1 query problem is a common performance issue that can occur when fetching entities and their associated entities in an ORM context. It refers to the situation where an initial query fetches a collection of entities, but then, for each entity in the collection, an additional query is executed to fetch its associated entities. This results in an unnecessary number of queries, leading to decreased performance and increased database overhead.

To address the N+1 query problem, you can apply the following solutions:

Eager Fetching: Use eager fetching (`fetch = FetchType.EAGER`) to retrieve the associated entities eagerly along with the main entity. For example, in the Author entity:

```
@OneToMany(mappedBy = "author", fetch = FetchType.EAGER)
private List<Book> books;
```

This will fetch the books collection immediately when the Author entity is retrieved, minimizing additional queries.

Join Fetching: Use join fetching (`JOIN FETCH`) in the query itself to fetch the associated entities in a single query. For example:

```
SELECT DISTINCT a FROM Author a LEFT JOIN FETCH a.books
```

The `JOIN FETCH` clause ensures that the associated books collection is fetched in the same query, eliminating the N+1 queries.

Batch Fetching: Utilize batch fetching to optimize fetching of associated entities. Batch fetching allows you to fetch multiple entities or collections in batches, reducing the number of queries. It is usually configured at the ORM framework level, such as Hibernate.

```
@Entity
@BatchSize(size = 10)
public class Author {
    // ...
}
```

Lombok

Lombok is a popular Java library that helps reduce boilerplate code in Java classes by providing annotations that automatically generate common code for getters, setters, constructors, and other repetitive tasks. Lombok integrates with your IDE and builds tools, such as Maven or Gradle, to automatically generate the required code during compilation.

Some of the commonly used Lombok annotations include:

- **@Getter and @Setter:** These annotations generate getter and setter methods for class fields automatically.
- **@NoArgsConstructor, @AllArgsConstructor, and @RequiredArgsConstructor:** These annotations generate no-argument, all-argument, and required-argument constructors, respectively.
- **@ToString:** This annotation generates a `toString()` method implementation that includes all class fields.
- **@EqualsAndHashCode:** This annotation generates `equals()` and `hashCode()` methods based on the class fields.
- **@Data:** This annotation is a combination of `@ToString`, `@EqualsAndHashCode`, `@Getter`, and `@Setter`. It generates all these methods together.
- **@Builder:** This annotation generates a builder pattern for creating instances of the class, allowing for fluent and readable code when constructing complex objects.

Using Lombok annotations can significantly reduce the amount of boilerplate code you need to write, making your code more concise and readable. It also helps to maintain consistency and reduces the chance of introducing errors when manually writing repetitive code.

Spring Boot Day 8

Beans Scopes

There are five types of [spring bean](#) scopes:

- **singleton** - only one instance of the spring bean will be created for the spring container. This is the default spring bean scope. While using this scope, make

sure bean doesn't have shared instance variables otherwise it might lead to data inconsistency issues.

- **prototype** – A new instance will be created every time the bean is requested from the spring container.
- **request** – This is the same as prototype scope, however it's meant to be used for web applications. A new instance of the bean will be created for each HTTP request.
- **session** – A new bean will be created for each HTTP session by the container.
- **global-session** – This is used to create global session beans for Portlet applications.

@Profile

Profiles are a core feature of the framework — allowing us to map our beans to different profiles for example, dev, test, and prod.

We can then activate different profiles in different environments to bootstrap only the beans we need.

We use the @Profile annotation — we are mapping the bean to that particular profile; the annotation simply takes the names of one (or multiple) profiles.

Consider a basic scenario: We have a bean that should only be active during development but not deployed in production.

We annotate that bean with a dev profile, and it will only be present in the container during development. In production, the dev simply won't be active:

Java

```
@Component
@Profile("dev")
public class DevDatasourceConfig
```

As a quick side note, profile names can also be prefixed with a NOT operator, e.g., !dev, to exclude them from a profile.

In the example, the component is activated only if the dev profile is not active:

Unset

```
@Component
@Profile("!dev")
public class DevDatasourceConfig
```

Can you use @Component together with @Profile?

Yes, `@Profile` annotation can be used together with `@Component` on top of the class representing spring bean.

If, class annotated with `@Component` does not have `@Profile`, that bean will exist in all profiles. You can specify one, multiple profiles or profile in which bean should not exists:

Can you use `@Bean` together with `@Profile`?

Yes, you can use the `@Bean` annotation together with the `@Profile` annotation in Spring Boot to conditionally create and register beans based on the active profiles. The `@Profile` annotation allows you to associate beans with specific profiles, while the `@Bean` annotation is used to declare a bean definition.

Can you make the `@Bean` method final in Spring?

No, it is not possible to make a method annotated with `@Bean` final in Spring.

The `@Bean` annotation is used to declare a method as a bean definition method within a Spring configuration class. This method is responsible for creating and configuring the bean instance to be registered in the Spring application context.

The final keyword in Java is used to indicate that a method or class cannot be overridden or extended, respectively. When a method is marked as final, it prevents subclasses from overriding that method.

What is the default bean id if you only use `@Bean`? How can you override this?

When you use the `@Bean` annotation in Spring without explicitly specifying a name or ID for the bean, the default bean ID is derived from the name of the `@Bean` method. By default, the bean ID will be the same as the name of the method.


```
java Copy code  
  
@Configuration  
public class MyConfiguration {  
  
    @Bean  
    public MyBean myBean() {  
        return new MyBean();  
    }  
}
```

In this case, the default bean ID for the `MyBean` instance will be "myBean", which is the same as the name of the `myBean()` method.

If you want to override the default bean ID and provide a custom ID for the bean, you can do so using the name attribute of the `@Bean` annotation.

Here's an example that demonstrates overriding the default bean ID:

java

 Copy code

```
@Configuration
public class MyConfiguration {

    @Bean(name = "customBeanId")
    public MyBean myBean() {
        return new MyBean();
    }
}
```

In this modified example, the name attribute of the `@Bean` annotation is set to "customBeanId". This means that the bean ID for the `MyBean` instance will be "customBeanId" instead of the default "myBean".

By providing a custom name using the name attribute, you can explicitly define the bean ID for the `@Bean` method.

SQL logging in Spring boot

In Spring Boot, you can enable SQL logging to see the executed SQL statements and their corresponding parameters. This can be useful for debugging, optimizing database queries, and understanding the interaction between your application and the database.

To enable SQL logging in Spring Boot, you can configure the logging level for the relevant logger used by your database library (such as Hibernate, MyBatis, or Spring Data JPA). The actual logger name may vary depending on the database library you are using, but the general concept remains the same.

Here's an example of enabling SQL logging using Hibernate as the database library:

1. Open the `application.properties` or `application.yml` file in your Spring Boot project.
2. Add the following configuration to enable SQL logging for Hibernate:

```
# Enable SQL logging for Hibernate
logging.level.org.hibernate.SQL=DEBUG
```

Debugging: SQL logging allows you to see the actual SQL statements executed by your application. It can help identify issues such as incorrect queries, missing or incorrect parameters, or unexpected query results. By examining the SQL logs, you can debug and troubleshoot database-related problems more effectively.

Embedded containers does Spring Boot support

Servers

- **Apache Tomcat:** Tomcat is a widely used Java web server and servlet container. It is the default embedded server provided by Spring Boot. Tomcat is known for its simplicity, lightweight nature, and good performance. It is suitable for most applications and provides excellent support for Java Servlets and JavaServer Pages (JSP).
- **Jetty** is another popular Java web server and servlet container. It is known for its scalability, high-performance, and low memory footprint. Jetty integrates well with Spring Boot and is a good choice for applications that require high concurrency or need to handle a large number of concurrent connections.
- **Undertow** is a lightweight and high-performance web server designed for modern applications. It offers non-blocking I/O and asynchronous request processing, making it suitable for highly scalable and efficient applications. Undertow provides excellent support for both traditional and reactive web programming models.

How to Lazily Initialize your spring boot application?

In Spring Boot, you can configure lazy initialization of beans to improve application startup performance. By default, Spring Boot eagerly initializes all beans during application startup. However, you can selectively enable lazy initialization for specific beans or components that are not immediately required.

To lazily initialize beans in Spring Boot, you can use the `@Lazy` annotation at the bean definition level or use the `@Lazy` attribute when autowiring dependencies.

Using `@Lazy` annotation at the bean definition level:

- Import the necessary annotations: `@Configuration`, `@Bean`, and `@Lazy`.
- Add the `@Lazy` annotation to the `@Bean` method or the class-level `@Component` or `@Service` annotation.

```
@Configuration
public class MyConfiguration {

    @Bean
    @Lazy
    public MyBean myBean() {
        return new MyBean();
    }
}
```

In this example, the `myBean()` method is annotated with `@Bean` and `@Lazy`. This means that the `MyBean` bean will be lazily initialized, and its instantiation will be delayed until it is actually requested.

Spring Batch

Spring Batch is a processing framework designed for robust execution of jobs. It simplifies the development of batch jobs that involve reading, processing, and writing large volumes of data. Batch processing refers to the execution of a series of tasks or operations in a sequential and automated manner, typically on a large dataset. Examples of batch processing tasks include data import/export, ETL (Extract, Transform, Load) processes, report generation, and data cleanup.

Spring Batch provides a set of features and components that facilitate the development and execution of batch jobs.

In the context of the Spring Framework, a singleton bean is a bean with a singleton scope, which means that a single instance of the bean is created and shared across multiple threads within the container.

By default, singleton beans in Spring are not thread-safe. If multiple threads concurrently access and modify the state of a singleton bean without any synchronization mechanisms, it can lead to race conditions and unexpected behavior.

Bean Lifecycle

In Spring Boot, the lifecycle of a bean is managed by the Spring IoC (Inversion of Control) container. The container creates, initializes, uses, and destroys beans based on their configuration and the lifecycle callbacks defined within them. Here is an overview of the bean lifecycle in Spring Boot:

Bean Definition: A bean is defined in the application context configuration, typically using annotations like `@Component`, `@Service`, or `@Repository`, or XML-based configuration. The bean definition specifies the scope, dependencies, and other configuration details.

Bean Instantiation: When the Spring container starts up, it reads the bean definitions and instantiates the beans accordingly. It creates an instance of each bean based on the configuration provided.

Dependency Injection: After instantiating the beans, the container performs dependency injection. It injects any required dependencies into the beans, typically using annotations like `@Autowired` or constructor-based injection.

Bean Initialization: Once the dependencies are injected, the container calls any initialization callbacks defined within the bean. These callbacks can be annotated with `@PostConstruct` or implemented using the `InitializingBean` interface.

Bean Usage: At this stage, the beans are fully initialized and available for use. Other components or services can use these beans as required.

Bean Destruction: When the Spring container is shut down or when a bean is no longer needed, the container calls any destruction callbacks defined within the bean. These callbacks can be annotated with `@PreDestroy` or implemented using the `DisposableBean` interface.

Spring Boot Day 9

Microservices Intro:

▶ Microservices explained - the What, Why and How?

Basic concept -> divide and conquer

Microservice Architecture is about splitting a large, complex systems vertically (per functional or business requirements) into smaller sub-systems which are processes (hence independently deployable) and these sub-systems communicates with each other via lightweight, language-agnostic network calls either synchronous (e.g. REST, gRPC) or asynchronous (via Messaging) way.

Microservices is an architecture wherein all the components of the system are put into individual components, which can be built, deployed, and scaled individually.

*Microservices use **service discovery** which acts as a guide to find the route of communication between each of them. Microservices then communicate with each other via a stateless server i.e. either by **HTTP Request/Message Bus**.*

Independent

Scalability

Decentralized

Real time load balancing

Available

Continuous delivery

Seamless API integration and monitoring

Isolation from failures

Auto provisioning

Benefits

This solution has a number of benefits:

- **Simple services** - each service consists of a small number of subdomains - possibly just one - and so is easier to understand and maintain

- **Team autonomy** - a team can develop, test and deploy their service independently of other teams
- **Fast deployment pipeline** - each service is fast to test since it's relatively small, and can be deployed independently
- **Support multiple technology stacks** - different services can use different technology stacks and can be upgraded independently
- **Segregate subdomains by their characteristics** - subdomains can be segregated by their characteristics into separate services in order to improve scalability, availability, security etc

Drawbacks

This solution has a number of (potential) drawbacks:

- Some distributed operations might be **complex**, and **difficult to understand** and **troubleshoot**
- Some distributed operations might be **potentially inefficient**
- Some operations might need to be implemented using **complex, eventually consistent** (non-ACID) transaction management since loose coupling requires each [service to have its own database](#).
- Some distributed operations might involve **tight runtime coupling** between services, which reduces their availability.
- Risk of tight design-time coupling between services, which requires time consuming **lockstep changes**

Microservices Design Patterns

- ***Database per Microservice:** Every Microservice has its own Data store, so that there is no strong-coupling between services in the database layer.
- ***Shared Database anti-pattern:**
- **Event sourcing:** Event sourcing is a design pattern used in microservice architectures to capture and store all changes (events) that occur within a system as a sequence of events. It involves persisting the state of an application as a series of events rather than just the current state.
- **Command Query Responsibility Segregation (CQRS):** If we use Event Sourcing, then reading data from the Event Store becomes challenging. To fetch an entity from the Data store, we need to process all the entity events. Also, sometimes we have different consistency and throughput requirements for reading and writing operations. In such use cases, we can use the CQRS pattern. In the CQRS pattern, the system's data modification part (Command) is separated from the data read (Query) part.
 - We use event handlers, each service has its own.
 - We can also use Triggers and Procedures in database to create/update/delete the same data in query database

Drawbacks

Replication delay + extra complexity + code duplication

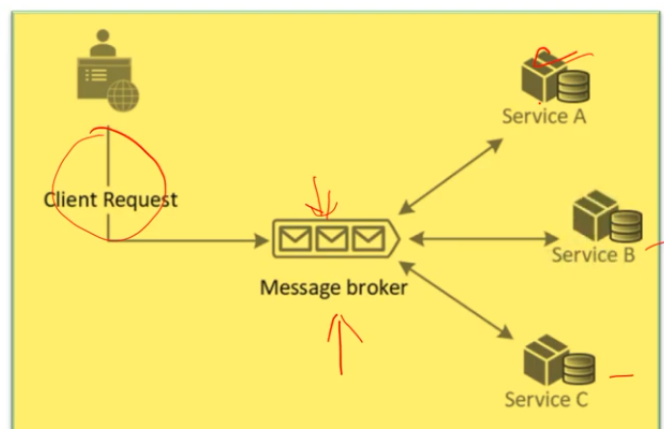
Benefits

Distributed systems + simpler command and query + flexibility + easy to scale + data Analytics efficient(because we have to create a new database to read only database for this)

- **Saga:** If you use Microservice Architecture with Database per Microservice, then managing consistency via distributed transactions is challenging. You cannot use the traditional [Two-phase commit protocol](#) as it either does not scale (SQL Databases) or is not supported (many NoSQL Databases). You can use the Saga pattern for distributed transactions in Microservice Architecture. Saga is an old pattern developed in 1987 as a conceptual alternative for long-running database transactions in SQL databases. But a modern variation of this pattern works amazingly for the distributed transaction as well. Saga pattern is a local transaction sequence where each transaction updates data in the Data Store within a single Microservice and publishes an Event or Message. The first transaction in a saga is initiated by an external request (Event or Action). Once the local transaction is complete (data is stored in the Data Store, and a message or event is published), the published message/event triggers the next local transaction in the Saga.
 - Types of Transactions:
 - Compensable Transactions (can be reversed by another transaction)
 - Pivot Transactions(last compensable transaction or first retryable transaction)
 - Retryable Transactions(follow the pivot and going to succeed)
 - Implementation
 - Choreography
 - Decentralised co-ordinations where each Microservice produces and listen to other Microservice's events/messages and decides if an action should be taken or not.

- **Benefits**
 - *good for simple workflows which have few services*
 - *No additional service*
 - *No single point of failure*

- **Drawbacks**
 - *Can be confusing when adding new steps*
 - *Cyclic dependency risk*
 - *Integration testing is difficult*



- **Orchestration**
 - Centralised co-ordinations where an Orchestrator tells the participating Microservices which local transaction needs to be executed.

When to use:

- Ensure data consistency without tight coupling
- Roll back if operation in the sequence fails

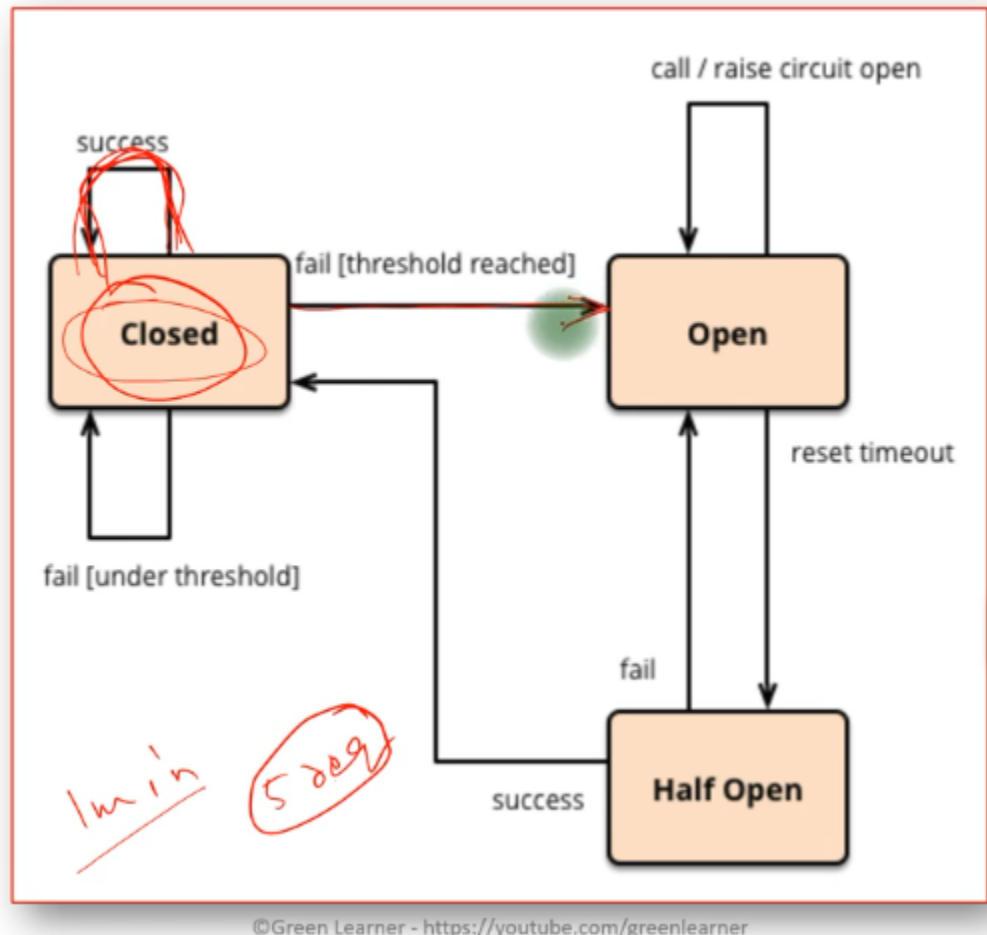
Less suitable for:

- Tight coupled transaction
- Compensating transaction
- Cyclic dependencies
- **Backends for Frontends (BFF):** In modern business application developments and especially in Microservice Architecture, the Frontend and the Backend applications are decoupled and separate Services. They are connected via API or GraphQL. If the application also has a Mobile App client, then using the same backend Microservice for both the Web and the Mobile client becomes problematic. The Mobile client's API requirements are usually different from Web clients as they have different screen size, display, performance, energy source, and network bandwidth. Backends for Frontends pattern could be used in scenarios where each UI gets a separate backend customized for the specific UI. It also provides other advantages, like acting as a Facade for downstream Microservices, thus reducing the chatty communication between the UI and downstream Microservices. Also, in a highly secured scenario where downstream Microservices are deployed in a DMZ network, the BFF's are used to provide higher security.
- **API Gateway:** In Microservice Architecture, the UI usually connects with multiple Microservices. If the Microservices are finely grained (FaaS), the Client may need to connect with lots of Microservices, which becomes chatty and challenging. Also, the services, including their APIs, can evolve. Large enterprises will like to have other cross-cutting concerns (SSL termination, authentication, authorization, throttling, logging, etc.). One possible way to solve these issues is to use API Gateway. API Gateway sits between the Client APP and the Backend Microservices and acts as a facade. It can work as a reverse proxy, routing the Client request to the appropriate Backend Microservice. It can also support the client request's fanning-out to multiple Microservices and then return the aggregated responses to the Client. It additionally supports essential cross-cutting concerns.
 - DrawBack:
 - High coupling
 - Extra application
 - Complexity of overall application increased
 - Increased latency
 - Implementation
 - Netflix Zuul
 - Spring cloud gateway
 - 3 rd parties Long, Apigee , Amazon etc
- **Circuit breaker:** A service client should invoke a remote service via a proxy that functions in a similar fashion to an electrical circuit breaker. When the number of

consecutive failures crosses a threshold, the circuit breaker trips, and for the duration of a timeout period all attempts to invoke the remote service will fail immediately. After the timeout expires the circuit breaker allows a limited number of test requests to pass through. If those requests succeed the circuit breaker resumes normal operation. Otherwise, if there is a failure the timeout period begins again.

- **States**

- Closed
- Open
- Half Open



- **Advantages:**

- Smooth handling of exception
- Cascading failure

- **Implementation:**

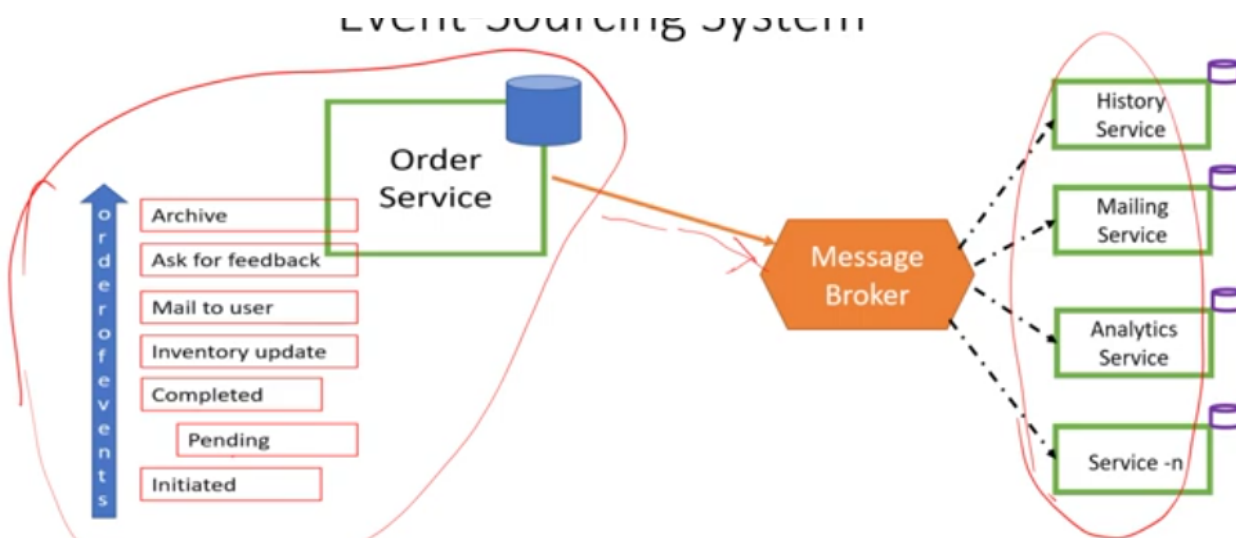
- Netflix-hystrix
- Hystrix
- Spring resilience

- **Aggregator:** collects related items of data and displays them combining data form different services and displaying it. Based on DRY principle

- **Event Sourcing Pattern**

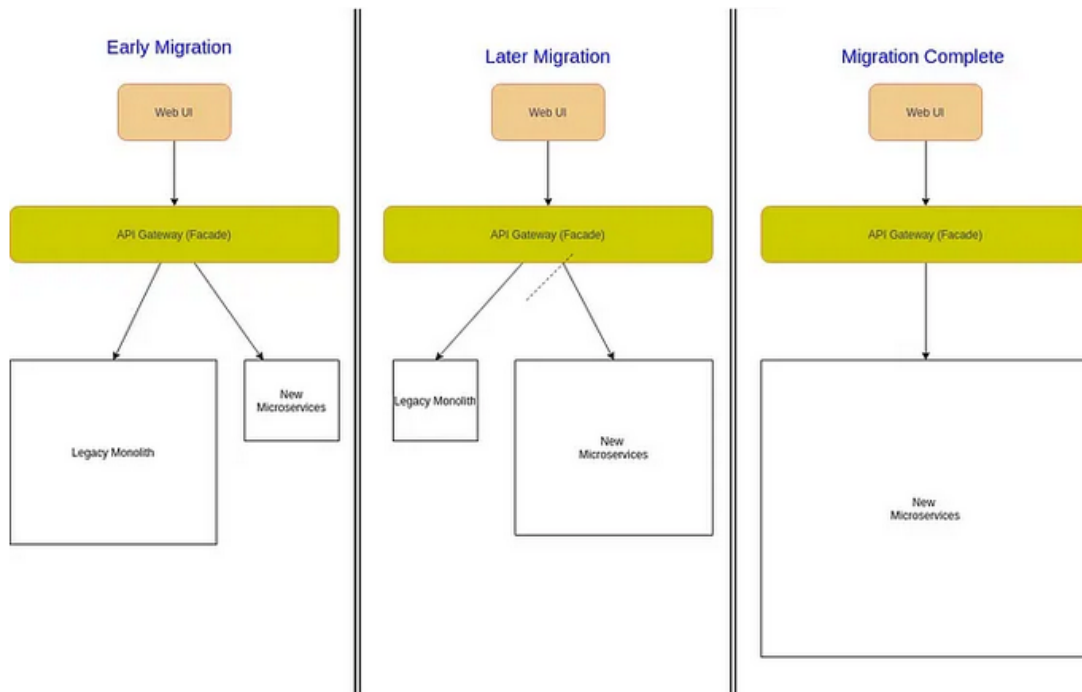
A service command typically needs to create/update/delete aggregates in the database and send messages/events to a message broker.

- State is stored as series of events
- Any record is about the state change from the previous one.
- We can always replay events to get the state at any point of time
- Efficient
- Asynchronous communication



- **Strangler**

If we want to use Microservice Architecture in a brownfield project, we need to migrate legacy or existing Monolithic applications to Microservices. Moving an existing, large, in-production Monolithic applications to Microservices is quite challenging as it may disrupt the application's availability.



What is Hystrix?

The Hystrix framework library helps to control the interaction between services by providing fault tolerance and latency tolerance. It improves the overall resilience of the system by isolating the failing services and stopping the cascading effect of failures.

SonarLint

SonarLint is a free IDE plugin which is easy to set up and can give instant feedback which will help you write better code.

It not only has a lot of rules but each rule provides a description and an example of compliant and non compliant code. Plugin comes with a default list of predefined rules that can be enabled/disabled. Configuration of rules can be exported so your teammates can also start using it. Small drawback is that no new rules can be added, but a good thing is that [SonarQube](#) [does support](#) custom rules.

Plugin offers a report about the current editing class but also it can be run against the whole project.

What Is a Feign Client?

Netflix provides Feign as an abstraction over REST-based calls, by which microservices can communicate with each other, but deNetflix eureka developers don't have to bother about REST internal details.

Inter process communication

Netflix eureka **the Netflix Service Discovery Server and Client**

@discoveryserver

Spring cloud gateway is intended **to sit between a requester and a resource that's being requested, where it intercepts, analyzes, and modifies every request.**

Keycloak

Resilience + TimeLimiter for circuit breaker

Sleuth + Zipkins for distributed tracing

Spring Cloud Sleuth is used to generate and attach the trace id, span id to the logs so that these can then be used by tools like Zipkin and ELK for storage and analysis. Zipkin is a distributed tracing system. It helps gather timing data needed to troubleshoot latency problems in service architectures.

First session and Mutual SSL

The idea of the first session is very simple. Users need to login to the application once, and then they can access all the services in the application. But, each user has to initially communicate with an authentication service. Well, this can definitely result in a lot of traffic between all the services and might be cumbersome for the developers to figure out failures in such a scenario.

Coming to Mutual SSL, applications often face traffic from users, 3rd parties and also microservices communicating with each other. But, since these services are accessed by the 3rd parties, there is always a risk of attacks. Now, the solution to such scenarios is mutual SSL or mutual authentication between microservices. With this, the data transferred between the services will be encrypted. The only problem with this method is that, when the number of microservices increase, then since each and every service will have its own TLS certificate, it will be very tough for the developers to update the certificates.

QUESTIONS

[Micro Services Questions](#)

Spring Boot Day 12

Purpose of the Assert Class

The Spring Assert class helps us validate arguments. By using methods of the Assert class, we can write assumptions which we expect to be true. And if they aren't met, a runtime exception is thrown.

Each Assert's method can be compared with the Java [assert](#) statement. Java assert statement throws an Error at runtime if its condition fails. The interesting fact is that those assertions can be disabled.

Here are some characteristics of the Spring Assert's methods:

- Assert's methods are static

- They throw either **IllegalArgumentException** or **IllegalStateException**
- The first parameter is usually an argument for validation or a logical condition to check
- The last parameter is usually an exception message which is displayed if the validation fails
- The message can be passed either as a String parameter or as a Supplier<String> parameter

Also note that despite the similar name, Spring assertions have nothing in common with the assertions of [JUnit](#) and other testing frameworks. Spring assertions aren't for testing, but for debugging.

Logical Assertions

- **isTrue**

Let's define a Car class with a public method drive():

Unset

```
public class Car {
    private String state = "stop";

    public void drive(int speed) {
        Assert.isTrue(speed > 0, "speed must be positive");
        this.state = "drive";
        // ...
    }
}
```

We can see how speed must be a positive number. The above row is a short way to check the condition and throw an exception if the condition fails:

Unset

```
if (!(speed > 0)) {
    throw new IllegalArgumentException("speed must be positive");
}
```

Each Assert's public method contains roughly this code – a conditional block with a runtime exception from which the application is not expected to recover.

If we try to call the drive() method with a negative argument, an **IllegalArgumentException** exception will be thrown:

Unset

```
Exception in thread "main" java.lang.IllegalArgumentException: speed
must be positive
```

This assertion was discussed above. It accepts a boolean condition and throws an `IllegalArgumentException` when the condition is false.

- **state()**

The `state()` method has the same signature as `isTrue()` but throws the **`IllegalStateException`**.

As the name suggests, it should be used when the method mustn't be continued because of an illegal state of the object.

Imagine that we can't call the `fuel()` method if the car is running. Let's use the `state()` assertion in this case:

Unset

```
public void fuel() {
    Assert.state(this.state.equals("stop"), "car must be
stopped");
    // ...
}
```

Of course, we can validate everything using logical assertions. But for better readability, we can use additional assertions which make our code more expressive.

Object and Type Assertions

- **notNull()**

We can assume that an object is not null by using the `notNull()` method:

Unset

```
public void changeOil(String oil) {
    Assert.notNull(oil, "oil mustn't be null");
    // ...
}
```

Throws **`IllegalArgumentException`**

- **isNull()**

On the other hand, we can check if an object is null using the isNull() method:

Java

```
public void replaceBattery(CarBattery carBattery) {  
    Assert.isNull(  
        carBattery.getCharge(),  
        "to replace battery the charge must be null");  
    // ...  
}
```

If the object is null then does not throw exception else, throws
IllegalArgumentException

- **isInstanceOf()**

To check if an object is an instance of another object of the specific type we can use the isInstanceOf() method:

Java

```
public void changeEngine(Engine engine) {  
    Assert.isInstanceOf(ToyotaEngine.class, engine);  
    // ...  
}
```

In our example, the check passes successfully as ToyotaEngine is a subclass of Engine.
Throws **IllegalArgumentException**

- **isAssignable()**

To check types, we can use Assert.isAssignable():

Unset

```
public void repairEngine(Engine engine) {  
    Assert.isAssignable(Engine.class, ToyotaEngine.class);  
    // ...  
}
```

Two recent assertions represent an is-a relationship. Throws **IllegalArgumentException**

Text Assertions

Text assertions are used to perform checks on String arguments.

- **hasLength()**

We can check if a String isn't blank, meaning it contains at least one whitespace. Throws **IllegalArgumentException**

Unset

```
public void startWithHasLength(String key) {  
    Assert.hasLength(key, "key must not be null and must not be  
empty");  
    // ...  
}
```

- **hasText()**

We can strengthen the condition and check if a String contains at least one non-whitespace character. Throws **IllegalArgumentException**

Unset

```
public void startWithHasText(String key) {  
    Assert.hasText(  
        key,  
        "key must not be null and must contain at least one  
non-whitespace character");  
    // ...  
}
```

- **doesNotContain()**

We can determine if a String argument doesn't contain a specific substring. If the string does not contain the value then does not throw else throws **IllegalArgumentException**.

Java

```
public void startWithNotContain(String key) {  
    Assert.doesNotContain(key, "123", "key mustn't contain 123");  
    // ...  
}
```

```
}
```

If key = yes then doesn't throw if key = yes123 or 123yes then throws exception

Collection and Map Assertions

- **notEmpty() for Collections**

As the name says, the notEmpty() method asserts that a collection is not empty meaning that it's not null and contains at least one element. Throws **IllegalArgumentException**

C/C++

```
public void repair(Collection<String> repairParts) {  
    Assert.notEmpty(  
        repairParts,  
        "collection of repairParts mustn't be empty");  
}
```

- **notEmpty() for Maps**

The same method is overloaded for maps, and we can check if a map is not empty and contains at least one entry. Throws **IllegalArgumentException**

Java

```
public void repair(Map<String, String> repairParts) {  
    Assert.notEmpty(  
        repairParts,  
        "map of repairParts mustn't be empty");  
    // ...  
}
```

Array Assertions

- **notEmpty() for Arrays**

Finally, we can check if an array is not empty and contains at least one element by using the notEmpty() method. Throws **IllegalArgumentException**

Java

```
public void repair(String[] repairParts) {  
    Assert.notEmpty(  
        repairParts,  
        "array of repairParts mustn't be empty");  
    // ...  
}
```

- **noNullElements()**

We can verify that an array doesn't contain null elements by using the `noNullElements()` method:

Unset

```
public void repairWithNoNull(String[] repairParts) {  
    Assert.noNullElements(  
        repairParts,  
        "array of repairParts mustn't contain null elements");  
    // ...  
}
```

Note that this check still passes if the array is empty, as long as there are no null elements in it.

Assumptions

Assumptions are used to run tests only if certain conditions are met. This is typically used for external conditions that are required for the test to execute properly, but which are not directly related to whatever is being unit tested.

- If the **assumeTrue()** condition is true, then run the test, else aborting the test.
- If the **assumeFalse()** condition is false, then run the test, else aborting the test.
- The **assumingThat()** is much more flexible, it allows part of the code to run as a conditional test.

In JUnit, assumptions are a way to conditionally skip tests based on certain conditions that are not met. They are different from regular assertions because regular assertions cause a test to fail when the condition is not met, whereas assumptions allow a test to be skipped.

JUnit provides a set of static methods in the `org.junit.Assume` class to define assumptions. If an assumption fails, the test method is skipped and marked as "ignored" rather than failing. This can be useful when a test relies on specific conditions or prerequisites that, if not met, would make the test meaningless or likely to fail.

```

public class ExampleTest {

    @Test
    public void someTest() {
        // Assume that a certain condition is true, otherwise skip the test
        assumeTrue(someCondition());

        // If the assumption passes, proceed with the test
        int result = someMethod();
        assertEquals(42, result);
    }

    private boolean someCondition() {
        // Implement your condition check here
        // Return true if the condition is met, false otherwise
        return true;
    }

    private int someMethod() {
        // Implement your test logic here
        return 42;
    }
}

```

In this example, if `someCondition()` returns false, the `someTest()` method will be skipped, and it will be marked as "ignored" in the test report. However, if `someCondition()` returns true, the test will proceed as usual, and the `assertEquals` assertion will be evaluated.

Assumptions are particularly useful when testing in different environments or configurations, where certain tests may only be relevant under specific circumstances. They allow you to avoid running tests that would be bound to fail and potentially waste time and resources.

Keep in mind that skipped tests due to assumptions are different from failing tests. If a test fails due to a regular assertion (e.g., `assertEquals`), it means there is a problem with the test or the code under test. On the other hand, skipped tests due to assumptions indicate that the test was intentionally skipped because the conditions required for its valid execution were not met.

Assertion vs Assumptions

Assumptions:

- Purpose: Assumptions are used to conditionally skip tests based on certain conditions that are not met. They allow you to specify prerequisites for the test to be meaningful and valid, and if those prerequisites are not met, the test is skipped rather than being marked as a failure.

- Behavior: When an assumption fails, the test is skipped, and it is marked as "ignored" in the test report. The assumption failure does not indicate a problem with the code under test but rather that the test's prerequisites were not satisfied.
- Outcome: Skipped tests due to assumptions indicate that the test is not relevant in the current context and saves time and resources by not executing irrelevant tests.

Assertions:

- Purpose: Assertions are used to validate expected conditions during test execution. They are used to check whether the actual results match the expected results and help identify discrepancies or bugs in the code being tested.
- Behavior: When an assertion fails, it indicates that there is an issue with the code under test. The test is considered to have failed, and the test execution stops at that point.
- Outcome: Failed assertions result in a failed test, indicating that there is a problem with the application or test logic.

Test Order in JUnit 5

In JUnit 5, we can use `@TestMethodOrder` to control the execution order of tests.

We can use our own `MethodOrderer`, as we'll see later.

Or we can select one of three built-in orderers:

- Alphanumeric Order
- `@Order` Annotation
- Random Order

Using Alphanumeric Order

JUnit 5 comes with a set of built-in `MethodOrderer` implementations to run tests in alphanumeric order.

For example, it provides `MethodOrderer.MethodName` to sort test methods based on their names and their formal parameter lists:

Java

```
@TestMethodOrder(MethodOrderer.MethodName.class)
public class AlphanumericOrderUnitTest {
    private static StringBuilder output = new StringBuilder("");

    @Test
    void myATest() {
        output.append("A");
    }
}
```

```

    }

    @Test
    void myBTest() {
        output.append("B");
    }

    @Test
    void myaTest() {
        output.append("a");
    }

    @AfterAll
    public static void assertOutput() {
        assertEquals("ABa", output.toString());
    }
}

```

Similarly, we can use [MethodOrderer.DisplayName](#) to sort methods alphanumerically based on their display names.

Please keep in mind that [MethodOrderer.Alphanumeric](#) is another alternative. However, this implementation is deprecated and will be removed in 6.0.

@Order Annotation

We can use the `@Order` annotation to enforce tests to run in a specific order.

In the following example, the methods will run `firstTest()`, then `secondTest()` and finally `thirdTest()`:

Java

```

@TestMethodOrder(MethodOrderer.OrderAnnotation.class)
public class OrderAnnotationUnitTest {
    private static StringBuilder output = new StringBuilder("");

    @Test

```

```

    @Order(1)
    void firstTest() {
        output.append("a");
    }

    @Test
    @Order(2)
    void secondTest() {
        output.append("b");
    }

    @Test
    @Order(3)
    void thirdTest() {
        output.append("c");
    }

    @AfterAll
    public static void assertOutput() {
        assertEquals("abc", output.toString());
    }
}

```

Using Random Order

We can also order test methods pseudo-randomly using the `MethodOrderer.Random` implementation:

Java

```

@TestMethodOrder(MethodOrderer.Random.class)
public class RandomOrderUnitTest {

```

```

    private static StringBuilder output = new StringBuilder("");

    @Test
    void myATest() {
        output.append("A");
    }

    @Test
    void myBTest() {
        output.append("B");
    }

    @Test
    void myCTest() {
        output.append("C");
    }

    @AfterAll
    public static void assertOutput() {
        assertEquals("ACB", output.toString());
    }
}

```

As a matter of fact, JUnit 5 uses `System.nanoTime()` as the default seed to sort the test methods. This means that the execution order of the methods may not be the same in repeatable tests. However, we can configure a custom seed using the `junit.jupiter.execution.order.random.seed` property to create repeatable builds. We can specify the value of our custom seed in the `junit-platform.properties` file:

Java

```
junit.jupiter.execution.order.random.seed=100
```

Using a Custom Order

Finally, we can use our own custom order by implementing the `MethodOrderer` interface. In our `CustomOrder`, we'll order the tests based on their names in a case-insensitive alphanumeric order:

C/C++

```
public class CustomOrder implements MethodOrderer {
    @Override
    public void orderMethods(MethodOrdererContext context) {
        context.getMethodDescriptors().sort(
            (MethodDescriptor m1, MethodDescriptor m2)->

m1.getMethod().getName().compareToIgnoreCase(m2.getMethod().getName(
))) );
    }
}
```

Then we'll use CustomOrder to run the same tests from our previous example in the order myATest(), myATest() and finally myBTest():

C/C++

```
@TestMethodOrder(CustomOrder.class)
public class CustomOrderUnitTest {

    // ...

    @AfterAll
    public static void assertOutput() {
        assertEquals("AaB", output.toString());
    }
}
```

Set Default Order

JUnit 5 provides a convenient way to set a default method orderer through the junit.jupiter.testmethod.order.default parameter.

Similarly, we can configure our parameter in the junit-platform.properties file:

Java

```
junit.jupiter.testmethod.order.default =  
org.junit.jupiter.api.MethodOrderer$DisplayName
```

The default orderer will be applied to all tests that aren't qualified with `@TestMethodOrder`. Another important thing to mention is that the specified class must implement the `MethodOrderer` interface.

Test Order in JUnit 4

For those still using JUnit 4, the APIs for ordering tests are slightly different. Let's go through the options to achieve this in previous versions as well.

Using MethodSorters.DEFAULT

This default strategy compares test methods using their hash codes.

In case of a hash collision, the lexicographical order (lexicographical order refers to the default order in which test methods are executed when no explicit test execution order is specified.) is used:

Java

```
@FixMethodOrder(MethodSorters.DEFAULT)  
public class DefaultOrderOfExecutionTest {  
    private static StringBuilder output = new StringBuilder("");  
  
    @Test  
    public void secondTest() {  
        output.append("b");  
    }  
  
    @Test  
    public void thirdTest() {  
        output.append("c");  
    }  
  
    @Test  
    public void firstTest() {  
        output.append("a");  
    }  
}
```

```

    @AfterClass
    public static void assertOutput() {
        assertEquals(output.toString(), "cab");
    }
}

```

When we run the tests in the class above, we will see that they all pass, including `assertOutput()`.

Using MethodSorters.JVM

Another ordering strategy is `MethodSorters.JVM`.

This strategy utilizes the natural JVM ordering, which can be different for each run:

```

Java

@FixMethodOrder(MethodSorters.JVM)
public class JVMOrderOfExecutionTest {
    // same as above
}

```

Each time we run the tests in this class, we get a different result.

Using MethodSorters.NAME_ASCENDING

Finally, this strategy can be used for running tests in their lexicographic order:

```

Java

@FixMethodOrder(MethodSorters.NAME_ASCENDING)
public class NameAscendingOrderOfExecutionTest {
    // same as above

    @AfterClass
    public static void assertOutput() {
        assertEquals(output.toString(), "abc");
    }
}

```

When we run the tests in this class, we see that they all pass, including `assertOutput()`. This confirms the execution order that we set with the annotation.

@Nested

We can use JUnit5's `@Nested` annotation to create nested test classes. The annotation must be added at a class level, for each inner class that contains tests:

Java

```
public class NestedTest {
    @Nested
    class FirstNestedClass {
        @Test
        void test() {
            System.out.println("FirstNestedClass.test()");
        }
    }

    @Nested
    class SecondNestedClass {
        @Test
        void test() {
            System.out.println("SecondNestedClass.test()");
        }
    }
}
```

Additionally, if we add setup or tear-down methods, they will be executed in their declaration. For instance, if we add a `@BeforeEach` method for each of the three classes, we'll expect each test to execute the setup method of the parent class, then the one from its own class, and, after that, the test itself:

Java

```
public class NestedTest {
    @BeforeEach()
    void beforeEach() {
        System.out.println("NestedTest.beforeEach()");
    }
}
```

```

    @Nested
    class FirstNestedClass {
        @BeforeEach()
        void beforeEach() {
            System.out.println("FirstNestedClass.beforeEach()");
        }

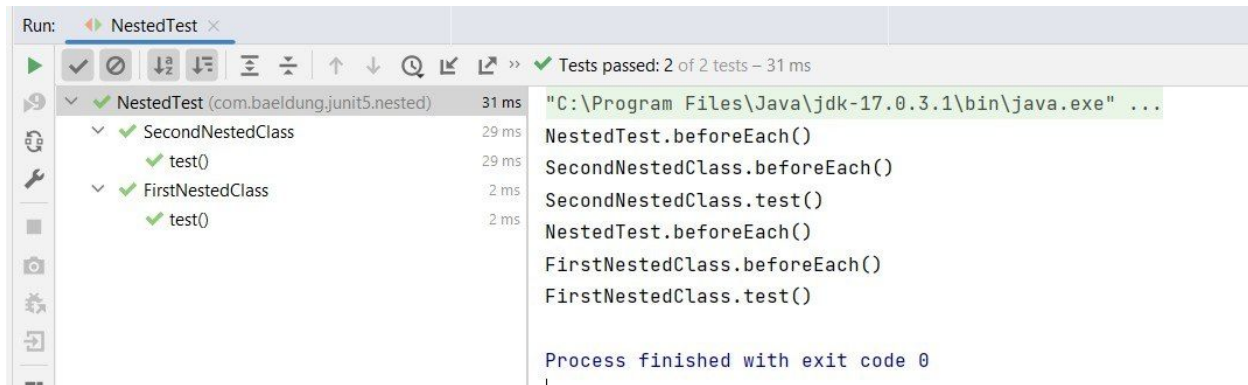
        @Test
        void test() {
            System.out.println("FirstNestedClass.test()");
        }
    }

    @Nested
    class SecondNestedClass {
        @BeforeEach()
        void beforeEach() {
            System.out.println("SecondNestedClass.beforeEach()");
        }

        @Test
        void test() {
            System.out.println("SecondNestedClass.test()");
        }
    }
}

```

Let's run the test class and check the order of the print statements in the console:



@ParameterizedTest

In JUnit 5, the `@ParameterizedTest` annotation is used to define and run parameterized tests.

Parameterized tests allow you to execute the same test logic with different sets of input parameters, making it easy to test multiple scenarios with minimal code duplication.

When using `@ParameterizedTest`, you need to provide a data source, which supplies the input values for each execution of the test method. JUnit 5 provides various built-in data sources (e.g., `@ValueSource`, `@CsvSource`, `@MethodSource`, etc.), or you can create your custom data sources by implementing the `ArgumentsProvider` interface.

Here's a basic example of using `@ParameterizedTest` with the `@ValueSource` data source:

```
import static org.junit.jupiter.api.Assertions.assertEquals;

public class ParameterizedTestExample {

    @ParameterizedTest
    @ValueSource(ints = {1, 2, 3, 4, 5})
    void testIsPositive(int number) {
        assertTrue(number > 0);
    }

    @ParameterizedTest
    @ValueSource(strings = {"apple", "banana", "orange"})
    void testStringLength(String fruit) {
        assertTrue(fruit.length() > 0);
    }

    @ParameterizedTest
    @ValueSource(doubles = {1.5, 2.75, 3.2})
    void testIsDoubleGreaterThanOne(double number) {
        assertTrue(number > 1.0);
    }
}
```

```
@ExtendWith(CalculatorParameterResolver.class)
public class CalculatorParameterizedTest {

    private final Calculator calculator = new Calculator();

    @ParameterizedTest
    @MethodSource("provideAdditionData")
    void testAddition(int a, int b, int expected) {
        int result = calculator.add(a, b);
        assertEquals(expected, result);
    }

    @ParameterizedTest
    @MethodSource("provideSubtractionData")
    void testSubtraction(int a, int b, int expected) {
        int result = calculator.subtract(a, b);
        assertEquals(expected, result);
    }

    @ParameterizedTest
    @MethodSource("provideMultiplicationData")
    void testMultiplication(int a, int b, int expected) {
        int result = calculator.multiply(a, b);
        assertEquals(expected, result);
    }

    @ParameterizedTest
    @MethodSource("provideDivisionData")
    void testDivision(int a, int b, int expected) {
        int result = calculator.divide(a, b);
        assertEquals(expected, result);
    }
}
```

```

private static Stream<int[]> provideAdditionData() {
    return Stream.of(new int[]{2, 3, 5}, new int[]{5, 2, 7});
}

private static Stream<int[]> provideSubtractionData() {
    return Stream.of(new int[]{5, 2, 3}, new int[]{8, 3, 5});
}

private static Stream<int[]> provideMultiplicationData() {
    return Stream.of(new int[]{3, 4, 12}, new int[]{5, 2, 10});
}

private static Stream<int[]> provideDivisionData() {
    return Stream.of(new int[]{10, 2, 5}, new int[]{12, 3, 4});
}

```

Parameter resolvers

Here's an example to illustrate how a parameter resolver can be beneficial:

Suppose we have a `PersonService` class that provides functionality related to persons, and we want to write tests for this service. Additionally, let's assume we have a `PersonRepository` class that we want to use as a dependency within the `PersonService`. In this scenario, we want to perform dependency injection for the `PersonRepository` into the `PersonService` during testing.

```

public class PersonService {
    private final PersonRepository repository;

    public PersonService(PersonRepository repository) {
        this.repository = repository;
    }

    public String getPersonNameById(int personId) {
        Person person = repository.findById(personId);
        return person != null ? person.getName() : "Person not found";
    }
}

public class PersonRepository {
    public Person findById(int personId) {
        // Implementation to fetch Person object by ID from the database
        return null; // For the sake of this example, we'll return null fo
    }
}

```


Now, we want to write tests for the `PersonService`. We can use a parameter resolver to provide the `PersonRepository` instance to the test methods automatically.

```
@ExtendWith(PersonServiceTest.PersonServiceRepositoryResolver.class)
public class PersonServiceTest {

    static class PersonServiceRepositoryResolver implements ParameterResolver {
        @Override
        public boolean supportsParameter(ParameterContext parameterContext,
            return parameterContext.getParameter().getType() == PersonRepository.class;
        }

        @Override
        public Object resolveParameter(ParameterContext parameterContext,
            return new PersonRepository(); // Create and provide the PersonRepository instance
        }
    }

    @Test
    void testGetPersonNameById(PersonRepository repository) {
        PersonService personService = new PersonService(repository);
        String personName = personService.getPersonNameById(1);
        assertEquals("John Doe", personName); // Assuming person ID 1 corresponds to John Doe
    }
}
```

In this example, the `PersonServiceRepositoryResolver` parameter resolver is used to provide the `PersonRepository` instance to the `testGetPersonNameById` test method. The resolver creates a new instance of `PersonRepository` and injects it as a method parameter for the test method. This way, we can easily perform dependency injection and isolate the testing of `PersonService`.

@ParameterizedTest vs Parameter Resolver

- **@ParameterizedTest:** @ParameterizedTest is an annotation provided by JUnit 5 for creating parameterized tests. Parameterized tests allow you to run the same test logic with different sets of input parameters. By annotating a test method with @ParameterizedTest, you indicate that the test should be executed multiple times, once for each set of input parameters provided by the specified data source.

```
@ParameterizedTest
@ValueSource(ints = {1, 2, 3})
void testIsPositive(int number) {
    assertTrue(number > 0);
}
```

In this example, the test method testIsPositive will be executed three times, once with number = 1, then with number = 2, and finally with number = 3.

@ParameterizedTest works with various built-in data sources like @ValueSource, @CsvSource, @MethodSource, and more. Additionally, you can create your custom data sources by implementing the **ArgumentsProvider** interface.

- **Parameter Resolver**

Parameter Resolver: Parameter resolvers are custom extensions in JUnit 5 that allow you to resolve method parameters dynamically at runtime. It's a way to inject dependencies or data into your test methods automatically.

```
public class CalculatorParameterResolver implements ParameterResolver {

    @Override
    public boolean supportsParameter(ParameterContext parameterContext, Extension
        return parameterContext.getParameter().getType() == Calculator.class;
    }

    @Override
    public Object resolveParameter(ParameterContext parameterContext, Extension
        return new Calculator();
    }
}
```

In this example, we have a custom parameter resolver `CalculatorParameterResolver` that provides an instance of the `Calculator` class as a method parameter for all test methods of the test class.

Parameter resolvers can be used to provide any kind of parameter (not limited to test data) required by test methods. They are useful when you have dependencies that need to be created or initialized and reused across multiple test methods, allowing you to keep your test methods clean and focused on the actual testing logic.

To summarize, `@ParameterizedTest` is used to execute the same test method with different sets of input parameters, while parameter resolvers are used to dynamically resolve and provide parameter values (including dependencies) to test methods at runtime. Both features can be used together to create powerful and flexible test scenarios in JUnit 5.

@NullSource

The `@ValueSource` annotation doesn't accept null values. There is one special annotation called `@NullSource` that will provide a null argument for the test. Another special annotation is `@EmptySource`, which provides an empty value for either a `String`, `List`, `Set`, `Map`, or an array. In this example, we are passing the values of null, an empty string, and a blank string to the test method.

Java

```
@ParameterizedTest
```

```
@NullSource
```

```
@EmptySource
```

```
@ValueSource(strings = { " " })
```

```

void nullEmptyAndBlankStrings(String text) {

    assertTrue(text == null || text.trim().isEmpty());

}

```

@RepeatedTest

@RepeatedTest annotation. This feature allows you to run the same test multiple times with different inputs or scenarios, which can be helpful in testing for stability and consistency. The **@RepeatedTest** annotation is applied to the test method, and you can specify the number of repetitions you want. By default, each repetition will execute the test independently.

```

@TestInstance(TestInstance.Lifecycle.PER_METHOD)
public class RepeatedTestExample {

    @RepeatedTest(5)
    void testRandomNumber() {
        int randomNumber = getRandomNumber();
        assertTrue(randomNumber >= 1 && randomNumber <= 100, "Random number");
    }

    private int getRandomNumber() {
        // Imagine this method generates a random number between 1 and 100
        return (int) (Math.random() * 100) + 1;
    }
}

```

@TestInstance

In JUnit 5, the **@TestInstance** annotation is used to configure the life cycle of test instances in a test class. It allows you to control how test instances are created and reused during the execution of test methods.

By default, JUnit 5 creates a new test instance for each test method (**TestInstance.Lifecycle.PER_METHOD**). This means that the test class is instantiated multiple times, once for each test method, ensuring that each test runs in isolation and does not share any state with other tests.

However, you can change this behavior and configure the lifecycle of test instances using the `@TestInstance` annotation with one of the following options:

- **`TestInstance.Lifecycle.PER_CLASS`:** This option tells JUnit 5 to create a single test instance for the entire test class. All test methods within the same test class will share this instance, allowing them to share state between test methods. Keep in mind that sharing state between tests can introduce dependencies and make tests less isolated, potentially leading to test interference and non-deterministic behavior.
- **`TestInstance.Lifecycle.PER_METHOD`:** This is the default behavior, as mentioned earlier. It creates a new test instance for each test method. Each test method runs in isolation and doesn't share state with other test methods.

@Timeout

In JUnit 5, you can use the `@Timeout` annotation to specify a timeout for a test method. A timeout ensures that if a test method takes longer than the specified duration to complete, the test will be automatically marked as a failure.

The `@Timeout` annotation can be applied to individual test methods or to the test class as a whole. When applied to a test class, the timeout value will be applied to all test methods in that class.

Parallel execution

JUnit 5 supports parallel execution of tests, which allows you to run multiple test methods concurrently, leveraging the power of multi-core processors to speed up test execution. By default, JUnit 5 does not run tests in parallel. To enable parallel execution, you need to configure it in your test suite.

There are two levels at which you can enable parallel execution:

- **Per Method Parallelism:** You can specify that individual test methods should be executed in parallel. This means that different test methods will run concurrently, but each test method will still run sequentially (non-concurrently).
- **Class Level Parallelism:** You can specify that entire test classes should be executed in parallel. This allows multiple test classes to run simultaneously, and within each test class, the test methods will still run sequentially (non-concurrently).

To enable parallel execution, you need to set the appropriate configuration in your build tool (e.g., Maven, Gradle) or in the testing framework configuration.

Maven Configuration:

In Maven, you can enable parallel execution of tests by using the `forkCount` and `reuseForks` properties in the Surefire plugin configuration:

```

<build>
  <plugins>
    <plugin>
      <artifactId>maven-surefire-plugin</artifactId>
      <version>3.0.0-M5</version>
      <configuration>
        <forkCount>4</forkCount> <!-- Number of parallel threads -->
        <reuseForks>false</reuseForks> <!-- Reuse forked JVMs or not -->
      </configuration>
    </plugin>
  </plugins>
</build>

```

@BeforeAll

@BeforeAll is a JUnit 5 annotation used to mark a static method that will be executed once before all test methods in a test class. This method is typically used for setup operations that need to be performed before running any of the test methods in the class.

The @BeforeAll method must be static because it is executed before any test instance is created. It should not have any dependencies on the state of the test instance or its fields, as it runs at the class level and not at the instance level.

@BeforeEach

@BeforeEach is a JUnit 5 annotation used to mark a method that will be executed before each test method in a test class. This method is typically used for setup operations that need to be performed before running each individual test method.

Unlike @BeforeAll, which runs only once before all test methods in a test class, @BeforeEach runs before each test method, ensuring that the setup operations are performed before every test, independently of other tests.

The @BeforeEach method can be used to initialize test data, set up test fixtures, or prepare the test environment specific to each test case. It is especially useful when you have tests that modify the state of the test class or require fresh data for each test execution.

@AfterAll

@AfterAll is a JUnit 5 annotation used to mark a static method that will be executed once after all test methods in a test class have completed their execution. This method is typically used for cleanup operations that need to be performed after running all the test methods in the class.

The @AfterAll method must be static because it is executed after all test methods have completed, and there are no more test instances. It should not have any dependencies on the state of the test instance or its fields, as it runs at the class level and not at the instance level.

@AfterEach

@AfterEach is a JUnit 5 annotation used to mark a method that will be executed after each test method in a test class. This method is typically used for cleanup operations that need to be performed after the execution of each individual test method.

Unlike @AfterAll, which runs only once after all test methods in a test class have completed their execution, @AfterEach runs after each test method, ensuring that the cleanup operations are performed after every test independently.

The @AfterEach method can be used to release resources, close connections, or clean up the test environment specific to each test case. It is especially useful when you have tests that modify the state of the test class or require cleanup after each test execution.

Mockito:

Mockito is a popular open-source mocking framework for Java. It is used for writing unit tests by creating mock objects that simulate the behavior of real objects in a controlled manner. With Mockito, you can easily mock dependencies and interactions between objects to isolate the code being tested and make the unit tests more focused and reliable.

Mockito provides methods to create mock objects, set expectations on their behavior, and verify interactions. It is commonly used in conjunction with testing frameworks like JUnit or TestNG to write effective unit tests.

Why Use Mockito?

- **Unit Testing:** Mockito is primarily used for unit testing, where you want to test a specific component or class in isolation, independent of its dependencies.
- **Isolation:** Mock objects help isolate the code under test from its dependencies (e.g., external services, databases) to ensure that unit tests are not affected by external factors.
- **Behavior Verification:** Mockito provides methods to verify that specific interactions between objects have occurred during the test.

Key Concepts:

- **Mock Objects:** Mock objects are objects that simulate the behavior of real objects. They are used to replace the actual dependencies of the code being tested. Mockito creates mock objects dynamically at runtime.
- **Mocking:** The process of creating and using mock objects in tests is called mocking.
- **Stubbing:** Stubbing refers to specifying the behavior of mock objects when specific methods are called. You can define return values or exceptions to be thrown during method invocations on mock objects.
- **Verification:** Mockito allows you to verify that specific methods on mock objects have been called with expected arguments or certain numbers of times.

Mock Vs Stubs

Stubs:

- **Purpose:** Stubs are objects that provide predetermined responses to method calls during testing. They are used to simulate the behavior of a real dependency for the test scenario, without focusing on how the method is called or what interactions occur with it.
- **Behavior:** Stubs are simple and typically return fixed values or predefined responses. They do not have any logic or verification of how they are used during the test.
- **Usage:** Stubs are most suitable when you want to create a controlled environment for a specific test case, and you don't need to check how the object was used. They are often used to provide fake data or simulate specific scenarios for testing.
- **Example:** Suppose you have a service that interacts with a database, and you want to test a method that retrieves data from the database. You can create a stub for the database access object (DAO) that always returns a fixed set of data, regardless of the method arguments or the number of times it was called.

Mocks:

- **Purpose:** Mocks, on the other hand, are objects that allow you to set expectations on how they should be used during the test. They verify that specific methods are called with specific arguments and in the expected order.
- **Behavior:** Unlike stubs, mocks have behavior expectations. You specify how the mock should be interacted with, and it will throw an error if the expectations are not met during the test.
- **Usage:** Mocks are used when you want to verify that the code being tested interacts correctly with its dependencies. They ensure that certain methods are called with the correct arguments, helping you check the interactions between components.
- **Example:** Continuing with the database example, if you want to test a service method that saves data to the database, you can use a mock for the database access object (DAO). You set expectations on the mock, such as expecting the save() method to be called with specific arguments, and the mock will verify that the expectations are met.

In summary, stubs are used for providing fixed responses to method calls and are generally simpler and less involved. They are suitable when you want to isolate the code being tested from its dependencies without worrying about how they are used.

Mocks, on the other hand, are used for behavior verification and allow you to set expectations on how the dependencies should be interacted with during the test. They are suitable when you want to ensure that the code being tested interacts correctly with its dependencies and follows specific patterns of usage.

@Mock:

- **Purpose:** The @Mock annotation is used to create a mock object of a class or an interface. It instructs Mockito to create a mock instance of the annotated type, which can then be used as a replacement for real dependencies in the test.
- **Usage:** You can use @Mock along with a testing framework, such as JUnit, to automatically initialize the mock objects before each test method is executed.


```

class MyClassTest {
    @Mock
    private MyDependency myDependency;

    @BeforeEach
    void setUp() {
        MockitoAnnotations.openMocks(this);
    }

    @Test
    void testMyMethod() {
        // Use myDependency mock in the test
    }
}

```

@Spy:

- **Purpose:** The @Spy annotation is used to create a partial mock of a real object. Unlike a regular mock, a spy allows you to keep the real behavior of the object for certain methods while stubbing or verifying behavior for other methods.
- **Usage:** @Spy is also used in conjunction with a testing framework, such as JUnit, to automatically initialize the spy objects before each test method is executed.

```

class MyClassTest {
    @Spy
    private MyService myService;

    @BeforeEach
    void setUp() {
        // Ensure that the real behavior of myService is retained for certain methods
        myService = Mockito.spy(new MyService());
    }

    @Test
    void testMyMethod() {
        // Use myService, with certain methods real and others mocked, in the test
    }
}

```

@Captor:

- **Purpose:** The @Captor annotation is used to capture arguments passed to mocked methods. It allows you to capture the arguments for further verification or processing in your tests.
- **Usage:** You can use @Captor along with ArgumentCaptor to capture arguments when methods are called on mock objects.

```
class MyClassTest {  
    @Mock  
    private MyDependency myDependency;  
  
    @Captor  
    private ArgumentCaptor<String> stringCaptor;  
  
    @BeforeEach  
    void setUp() {  
        MockitoAnnotations.openMocks(this);  
    }  
  
    @Test  
    void testMyMethod() {  
        // Use myDependency in the test, and capture the argument passed to  
        myDependency.someMethod("Hello");  
        Mockito.verify(myDependency).someMethod(stringCaptor.capture());  
        String capturedValue = stringCaptor.getValue();  
        // Perform further verification or assertions with the captured value  
    }  
}
```

@InjectMocks:

- **Purpose:** The @InjectMocks annotation is used to automatically inject mock or spy objects into the target class being tested. It helps simplify the process of setting up the object under test with its dependencies (mocks or spies).
- **Usage:** You can use @InjectMocks along with a testing framework, such as JUnit, to automatically inject the mock objects into the target class.

```

class MyClassTest {
    @Mock
    private MyDependency myDependency;

    @InjectMocks
    private MyClass myClass;

    @BeforeEach
    void setUp() {
        MockitoAnnotations.openMocks(this);
    }

    @Test
    void testMyMethod() {
        // Use myClass in the test, and myDependency will be automatically
    }
}

```

@Mock vs @Spy

@Mock

Purpose: The @Mock annotation is used to create a mock object of a class or an interface. Mock objects are entirely fake objects with no real implementation; they provide a controlled environment for testing by allowing you to define the behavior of specific methods.

Behavior: Mock objects do not retain any real behavior of the original class or interface. All method calls on a mock return default values based on the return type (e.g., null for objects, 0 for integers, false for booleans).

Usage: @Mock is commonly used when you want to replace real dependencies with fake ones during unit testing. Mocks are useful for isolating the unit under test (class or method) from its dependencies, allowing you to focus on testing the specific behavior of the unit.

When a method in when() it accepts parameters then, it will return the value specified in the thenResult() method otherwise, it will return 0 for integers and null for string for the same function with different parameters and for different functions too

With Mock Object Method (Mock Method)

```
@Test
@Order(1)
void MockMethod()
{
    when(mockCalculator.add( a: 5, b: 6)).thenReturn( t: 10);
    // result will be 10
    int result = mockCalculator.add( a: 5, b: 6);
    assertEquals( expected: 10, result);
    System.out.println("Mock Res "+ result);
}
```

Output

Test Passed

```
OpenJDK 64-Bit Server VM warning: Sharing is only
Mock Res 10
```

Method (Same Method with different Parameters)

```
@Order(2)
@Test
void differentParams()
{
    // same method different parameters
    int r = mockCalculator.add( a: 6, b: 5);
    System.out.println("Mock Different Param "+ r);
    assertEquals( expected: 11, r);
}
```

Output

Test Failed

```
Mock Different Param 0

org.opentest4j.AssertionFailedError:
Expected :11
Actual   :0
<Click to see difference>
```

Method (Different Method that returns int)

```
@Order(3)
@Test
void differentMethodInt()
{
    // different method
    int res = mockCalculator.multiply(a: 7, b: 8);
    System.out.println("Mock Result "+ res);
    assertEquals("expected: 56, res);
}
```

Output

Test Failed

```
Mock Result 0

org.opentest4j.AssertionFailedError:
Expected :56
Actual   :0
Printed: Mock Result 0
```

Method (Different Method with return Type as String)

```
@Order(4)
@Test
void differentMethodString() {

    //String method that returns hello
    String resString = mockCalculator.returnHello();
    System.out.println("Mock String "+ resString);
    assertEquals( expected: "Hello", resString);
}
```

Output

Test Failed

```
Mock String null

org.opentest4j.AssertionFailedError:
Expected :Hello
Actual   :null
<Click to see difference>
```

@Spy:

Purpose: The @Spy annotation is used to create a partial mock of a real object. A spy retains the real behavior of the original class or object for all methods unless specified otherwise. You can choose to override the behavior of specific methods if needed.

Behavior: When you use a spy, all method calls on the spy invoke the actual implementation of the class, unless the method is explicitly stubbed using Mockito's when() and thenReturn() methods.

Usage: @Spy is useful when you want to test a class or method that relies on a real implementation but requires some methods to behave differently during testing.

Method (Spy Method)

```
@Test
@Order(1)
void MockMethod()
{
    when(spyCalculator.add( a: 5, b: 6)).thenReturn( t: 10);
    int result = spyCalculator.add( a: 5, b: 6);
    System.out.println("Mock Res "+ result);
    assertEquals( expected: 10, result);
}
```

Output

Test Passed

```
OpenJDK 64-Bit Server VM warning: Sharing is on
Mock Res 10
```

Method(Same Method with different Parameters)

```
@Order(2)
@Test
void differentParams()
{
    // same method different parameters
    int r = spyCalculator.add( a: 6, b: 5);
    System.out.println("Mock Different Param "+ r);
    assertEquals( expected: 11, r);
}
```

Output

Test Passed

```
Mock Different Param 11
```

Method (Different Method that returns int)

```
@Order(3)
@Test
void differentMethodInt()
{
    // different method
    int res = spyCalculator.multiply(a: 7, b: 8);
    System.out.println("Mock Result " + res);
    assertEquals(expected: 56, res);
}
```

Output

Test Passed

```
Mock Result 56
```


Method (Different Method with return Type as String)

```
@Order(4)
@Test
void differentMethodString() {

    //String method that returns hello
    String resString = spyCalculator.returnHello();
    System.out.println("Mock String "+ resString);
    assertEquals("expected: \"Hello\", resString);
}
```

Output

Test Passed

```
Mock String Hello
```

Udemy Course

SOAP

Defines XML request and XML response. It creates an Envelope which contains header and body where header is optional.

Transport

- SOAP over MQ
- SOAP over HTTP

REST

No restriction for data exchange format.

Transport only HTTP

Implement Content Negotiation

```
<dependency>
  <groupId>com.fasterxml.jackson.dataformat</groupId>
  <artifactId>jackson-dataformat-xml</artifactId>
</dependency>
```

when a client makes a request to a RESTful API, it can specify the preferred format of the response (e.g., JSON, XML, HTML) and the server should attempt to provide the response in the desired format if possible.

Content negotiation is typically achieved through the use of HTTP headers. The following are the key HTTP headers involved in content negotiation:

Add Validation in @Path Variable

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-validation</artifactId>
</dependency>
```

If the arguments in the body are not up to the defined set of rules, the request will not work.

```

public class User {
    4 usages
    private Integer id;
    4 usages
    @Size(min = 2, message = "Name Should Have At least 2 characters!")
    private String name;
    4 usages
    @Past(message = "Birth Date Should be in the past")
    private LocalDate birthDate;

```

Here @Size(min = 2, message) and @Past are part of validation. We can define attributes ourselves such as size etc. @Past means that the date should be in the past and not of future

Create Automatic Documentation

To create automatic documentation, use Swagger/openAPI

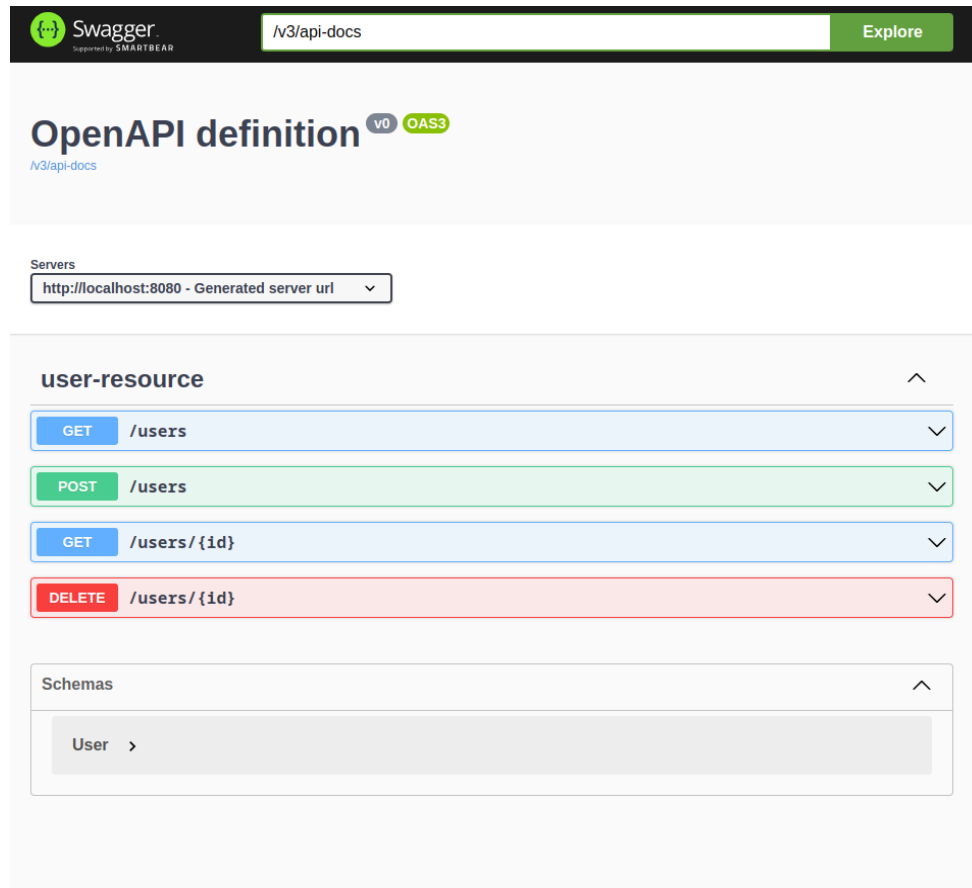
Dependency

```

<dependency>
<groupId>org.springdoc</groupId>
<artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
<version>2.0.2</version>
</dependency>

```

To check the document, go to <http://localhost:8080/swagger-ui/index.html> after adding the dependency.



Internationalization i18n

Internationalization, often abbreviated as "i18n" (where "18" represents the number of letters between "i" and "n"), is the process of designing and developing software applications or products in a way that allows them to be easily adapted for different languages, regions, and cultures without requiring code changes. The primary goal of internationalization is to make an application globally accessible and usable by people from diverse linguistic and cultural backgrounds.

Internationalization involves separating the presentation and content of an application from the underlying code. This allows for the localization of the application, which is the process of adapting it to a specific language, region, or cultural context.

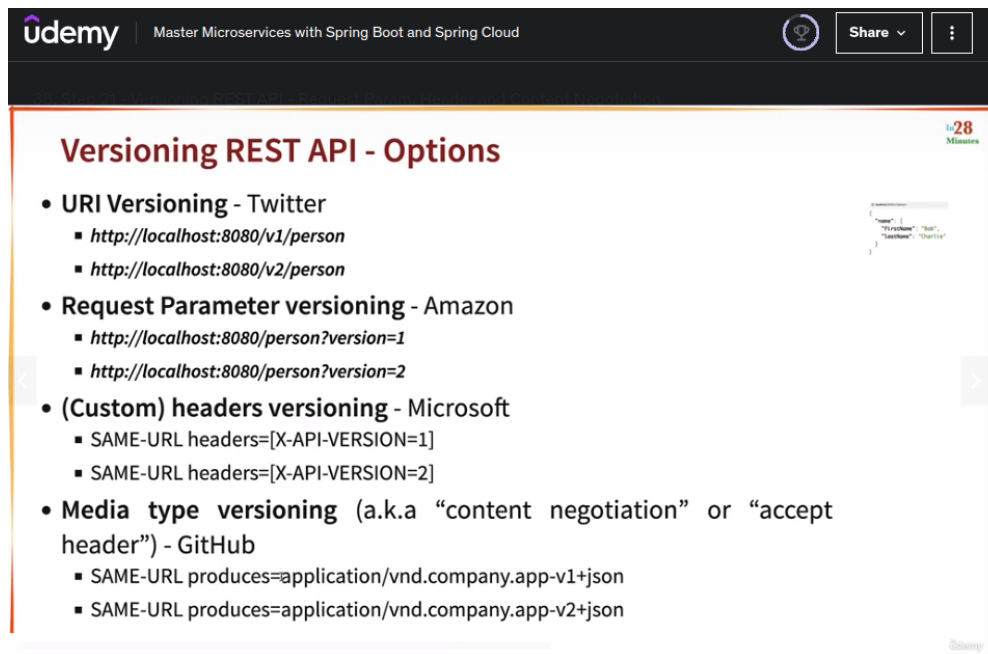
Key aspects of internationalization include:

- **Character Encoding:** Ensuring that the application supports various character encodings, such as UTF-8, to accommodate characters from different languages and scripts.
- **User Interface (UI) Strings:** Storing all user-visible strings (e.g., labels, messages, tooltips) in external resource files or databases to facilitate translation and localization.
- **Date and Time Formatting:** Using locale-specific date and time formats based on the user's region and cultural conventions.
- **Number Formatting:** Adapting numerical representations (e.g., decimal separators, digit grouping) based on the user's locale.

- **Currency Formatting:** Supporting various currency symbols and formats as per the user's region.
- **Text Directionality:** Allowing for right-to-left (RTL) text rendering for languages like Arabic and Hebrew, in addition to the default left-to-right (LTR) rendering.
- **Support for Plurals:** Handling plural forms correctly in different languages, as different languages have varying rules for pluralization.
- **Message Externalization:** Keeping error messages and other user feedback outside of the codebase to enable easy translation and modification without code changes.

Versioning REST API

REST APIs can be mapped to different versions by the following methods



The screenshot shows a Udemy video player interface. The video title is "Master Microservices with Spring Boot and Spring Cloud". The video content is a slide titled "Versioning REST API - Options" with a duration of 28 minutes. The slide lists four methods for versioning REST APIs:

- **URI Versioning - Twitter**
 - `http://localhost:8080/v1/person`
 - `http://localhost:8080/v2/person`
- **Request Parameter versioning - Amazon**
 - `http://localhost:8080/person?version=1`
 - `http://localhost:8080/person?version=2`
- **(Custom) headers versioning - Microsoft**
 - SAME-URL headers=[X-API-VERSION=1]
 - SAME-URL headers=[X-API-VERSION=2]
- **Media type versioning (a.k.a "content negotiation" or "accept header") - GitHub**
 - SAME-URL produces=application/vnd.company.app-v1+json
 - SAME-URL produces=application/vnd.company.app-v2+json

HATEOAS

HATEOAS stands for "Hypermedia as the Engine of Application State." It is a constraint and design principle of RESTful APIs that promotes a self-descriptive and discoverable communication style between clients and servers. The HATEOAS constraint is part of Roy Fielding's doctoral dissertation, where he introduced and defined the REST architectural style. In traditional web applications, clients are tightly coupled to server endpoints and have prior knowledge of the available resources and their URLs. HATEOAS aims to decouple clients from the server's URL structure and instead provide a dynamic and navigable API through hypermedia links.

In a HATEOAS-compliant RESTful API, each response from the server includes hypermedia links that represent the available actions or state transitions that the client can perform. These links are typically provided in the response payload using standardized formats such as JSON Hypertext Application Language (HAL), JSON-LD, or Atom.

The fundamental idea behind HATEOAS is that clients should interact with a RESTful API solely based on the hypermedia links present in the response without the need for any prior knowledge of resource URLs. Clients start by making an initial request to a known URL (e.g., the API root), and from there, they follow the hypermedia links to discover and access other resources and actions. **Used to add hyperlinks to already existing methods**

```
@GetMapping("/{id}")
public EntityModel<User> retrieveUser(@PathVariable int id){
    User user = service.findOne(id);
    if(user==null){
        throw new UserNotFoundException("id: " + id);
    }

    EntityModel<User> entityModel = EntityModel.of(user); // this is used to add a hyperlink in
    WebMvcLinkBuilder linkBuilder = WebMvcLinkBuilder
        .linkTo(WebMvcLinkBuilder
            .methodOn(this.getClass())
                .retrieveAllUsers()); // this is used to get the URL of particular m

    entityModel.add(linkBuilder.withRel("All Users")); // this is used to specify the heading/pu
    return entityModel;
}
```

EntityModel

In the context of Spring Framework and Spring HATEOAS, an EntityModel is a class that represents a single resource in a RESTful API response. It is designed to encapsulate the resource's data along with hypermedia links (HATEOAS links) that provide information about the available actions or state transitions related to the resource.

The EntityModel class is part of the Spring HATEOAS module, which enables easy implementation of the HATEOAS principle in RESTful APIs. It is typically used to wrap the domain model or entity in the API response, making it self-descriptive and discoverable for the client.

Here's a brief overview of the key features of EntityModel:

- **Wrapping a Resource:** EntityModel provides a way to wrap a single resource, such as a domain object or data transfer object (DTO), along with hypermedia links. It is used to create a self-contained and self-descriptive representation of the resource.
- **Hypermedia Links:** In addition to the resource data, EntityModel includes hypermedia links that represent related actions or state transitions that can be performed on the resource. These links allow clients to navigate the API without the need for prior knowledge of specific resource URLs.
- **Representation Model:** EntityModel is an example of a "representation model" in the context of RESTful APIs. Representation models are classes that define how a resource should be represented in the API response, including the data and hypermedia elements.

```

@RestController
public class MyController {

    @GetMapping("/api/resource/{id}")
    public EntityModel<MyResource> getResource(@PathVariable Long id) {
        // Assume MyResource is a domain object representing the resource
        MyResource resource = getResourceById(id);

        // Create an EntityModel and add the resource data
        EntityModel<MyResource> resourceModel = EntityModel.of(resource);

        // Add hypermedia links using WebMvcLinkBuilder
        Link selfLink = WebMvcLinkBuilder.linkTo(WebMvcLinkBuilder.methodOn(MyController.class)
            .getResource(id)).withSelfRel();
        resourceModel.add(selfLink);

        // Add more links as needed based on resource actions or relationships

        return resourceModel;
    }

    // Assume this method retrieves MyResource from the database
    private MyResource getResourceById(Long id) {
        // Implementation details...
    }
}

```

In this example, the `getResource` method returns an `EntityModel<MyResource>` representing a single resource with the specified ID. The resource data is wrapped in the `EntityModel`, and a self-referencing link is added using `WebMvcLinkBuilder`. Clients can use the hypermedia link to navigate to the resource or related actions without having prior knowledge of the specific URL structure.

Using `EntityModel` allows developers to adhere to the HATEOAS principle and create more discoverable and self-descriptive RESTful APIs in a Spring application.

Customizing REST API Responses - Filtering and more..

- **Serialization:** Convert object to stream (example: JSON)
 - Most popular JSON Serialization in Java: Jackson
- How about customizing the REST API response returned by Jackson framework?
- **1:** Customize field names in response
 - @JsonProperty
- **2:** Return only selected fields
 - **Filtering**
 - Example: Filter out Passwords
 - **Two types:**
 - **Static Filtering:** Same filtering for a bean across different REST API
 - @JsonIgnoreProperties, @JsonIgnore
 - **Dynamic Filtering:** Customize filtering for a bean for specific REST API
 - @JsonFilter with FilterProvider

```
localhost:8080/filtering-1st
[
  {
    "field2": "value2",
    "field3": "value3"
  },
  {
    "field2": "value22",
    "field3": "value32"
  }
]

localhost:8080/filtering
{
  "field1": "value1",
  "field2": "value2",
  "field3": "value3"
}
```

@siony

Serialization

Serialization in Spring Boot refers to the process of converting Java objects into a format suitable for data storage or transmission, typically in the form of bytes, JSON, XML, or other data formats. When building RESTful APIs or web applications in Spring Boot, serialization plays a crucial role in converting Java objects to JSON or XML to be sent as HTTP responses or stored in databases.

Spring Boot leverages Jackson as its default serialization and deserialization library for converting Java objects to JSON and vice versa. Jackson is a high-performance JSON processor that provides robust and flexible handling of JSON data.

Weekly Sync Questions Session

Highlighted Topics

Set instead of List in fetch

In the context of data retrieval and persistence operations, such as fetching data from a database, using a Set instead of a List can be beneficial in certain scenarios. The choice between Set and List depends on the specific requirements and characteristics of the data you are working with.

Here are some reasons why you might choose to use a Set instead of a List in a fetch operation:

1. **Uniqueness:** A Set guarantees uniqueness of elements, meaning that duplicate elements are automatically eliminated. This can be useful when you want to ensure that the fetched data contains only distinct values. If you're not concerned about duplicates, a List can be more appropriate.
2. **Order:** A Set does not impose any specific order on its elements, while a List maintains the order in which elements are added. If the order of the fetched data is not important, using a Set can be more efficient since it does not need to maintain the order.
3. **Fast Lookup:** Set implementations, such as HashSet or TreeSet, offer fast lookup times for specific elements. If you frequently need to check whether a particular element exists in the fetched data, a Set can provide better performance compared to a List, which requires iterating over the entire list to find an element.
4. **Equality-based Operations:** If your fetched data objects override the equals() and hashCode() methods to define equality based on specific attributes, using a Set can help you avoid duplicates based on that specific equality criteria.

It's important to note that the decision to use a Set or a List depends on the specific requirements of your application and the nature of the data you are working with. If you need to maintain duplicates, care about the order of elements, or require frequent positional access to elements, a List would be more suitable. However, if you need uniqueness, fast lookup, or don't care about the order, a Set might be a better choice.

@RequestParam

Java

```
@GetMapping("/api/foos")
@ResponseBody
public String getFoos(@RequestParam String id) {
    return "ID: " + id;
}
```

Java

<http://localhost:8080/spring-mvc-basics/api/foos?id=abc>

@PathVariable

Java

```
@GetMapping("/api/employees/{id}")
@ResponseBody
public String getEmployeesById(@PathVariable String id) {
    return "ID: " + id;
}
```

Java

<http://localhost:8080/api/employees/111>

SpringBootApplication

- Auto- Configuration
- Component Scan
- Spring Boot Configuration.

Spring vs Spring Boot

- [Spring Vs Spring Boot](#)

@RequestMapping

In Spring Boot, `@RequestMapping` is an annotation used to map HTTP requests to specific methods or controller classes. It is a fundamental annotation for building RESTful APIs in Spring Boot, allowing you to define the URL path and HTTP method(s) that a method or class should handle.

The `@RequestMapping` annotation can be applied at the method level or the class level:

- **Method-Level Mapping:**

```
@RestController
public class MyController {
    @RequestMapping(value = "/users", method = RequestMethod.GET)
    public ResponseEntity<List<User>> getUsers() {
        // Handle GET request to /users
        // ...
    }
}
```

In the above example, the `getUsers()` method handles a GET request to the `/users` endpoint. The `@RequestMapping` annotation specifies the URL path (`"/users"`) and the HTTP method (`RequestMethod.GET`) that this method should handle.

- **Class-Level Mapping:**

```
@RestController
@RequestMapping("/api")
public class MyController {
    @RequestMapping(value = "/users", method = RequestMethod.GET)
    public ResponseEntity<List<User>> getUsers() {
        // Handle GET request to /api/users
        // ...
    }
}
```

In this example, the class `MyController` has a class-level `@RequestMapping` annotation with the base path (`"/api"`). The `getUsers()` method's `@RequestMapping` annotation specifies the additional path (`"/users"`), resulting in the final endpoint as `/api/users`.

The `@RequestMapping` annotation supports various attributes to further define the mapping, including:

- **value or path:** Specifies the URL path pattern for the endpoint.

- **method:** Specifies the HTTP method(s) that the method should handle.
- **params:** Defines additional request parameter conditions.
- **headers:** Specifies the request header conditions.
- **consumes:** Specifies the media types that the method consumes.
- **produces:** Specifies the media types that the method produces.

ResponseEntity

In Spring Boot, `ResponseEntity` is a class that represents the entire HTTP response, including the response body, headers, and status code. It is often used as the return type of a controller method to customize the response sent back to the client.

Using `ResponseEntity` provides more flexibility and control over the HTTP response compared to returning a simple object or a predefined type. It allows you to customize the response status code, headers, and body based on the specific requirements of your application.

Here's an example of using `ResponseEntity` in a Spring Boot controller:

```
@RestController
public class MyController {
    @GetMapping("/users/{id}")
    public ResponseEntity<User> getUser(@PathVariable Long id) {
        User user = userService.getUserById(id);

        if (user != null) {
            return ResponseEntity.ok(user);
        } else {
            return ResponseEntity.notFound().build();
        }
    }
}
```

In the above example, the `getUser()` method retrieves a `User` object from the `userService` based on the provided `id`. If a user is found, `ResponseEntity.ok(user)` is returned with a status code of 200 (OK) and the user object as the response body. If no user is found, `ResponseEntity.notFound().build()` is returned with a status code of 404 (Not Found). `ResponseEntity` provides various static factory methods to create different types of responses, such as:

- **ok():** Creates a response with a status code of 200 (OK) and a body.
- **created():** Creates a response with a status code of 201 (Created) and a location header.
- **noContent():** Creates a response with a status code of 204 (No Content) and no body.
- **badRequest():** Creates a response with a status code of 400 (Bad Request) and an optional body.

- **notFound():** Creates a response with a status code of 404 (Not Found) and an optional body.

You can also customize the response headers by using methods like `header()`, `headers()`, or `contentType()`. Additionally, you can specify any other desired HTTP status code by using the `status()` method.

JDBC VS JPA

- JDBC is database-dependent, which means that different scripts must be written for different databases. On the other hand, JPA is database-agnostic, meaning that the same code can be used in a variety of databases with few (or no) modifications.
- Because JDBC throws checked exceptions, such as `SQLException`, we must write it in a try-catch block. On the other hand, the JPA framework uses only [unchecked exceptions, like Hibernate](#). Hence, we don't need to catch or declare them at every place we're using them.
- JPA-based applications still use JDBC under the hood. Therefore, when we utilize JPA, our code is actually using the JDBC APIs for all database interactions. In other words, JPA serves as a layer of abstraction that hides the low-level JDBC calls from the developer, making database programming considerably easier.
- In JDBC, transaction management is handled explicitly by using `commit` and `rollback`. On the other hand, transaction management is implicitly provided in JPA.

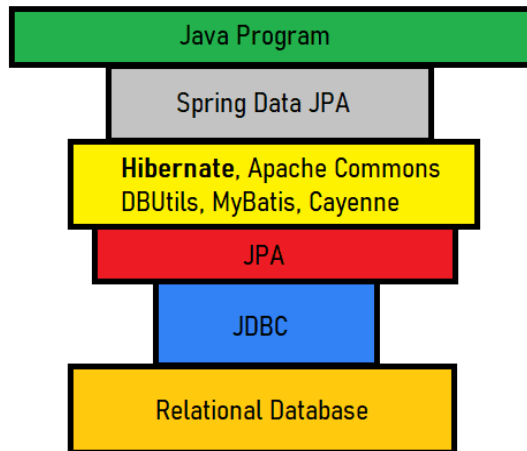
JDBC VS Spring JDBC

Spring provides a simplification in handling database access with the Spring JDBC Template. The Spring JDBC Template has the following advantages compared with standard JDBC.

- The Spring JDBC template allows to clean-up the resources automatically, e.g. release the database connections.
- The Spring JDBC template converts the standard JDBC `SQLExceptions` into `RuntimeExceptions`. This allows the programmer to react more flexibly to the errors. The Spring JDBC template converts also the vendor specific error messages into better understandable error messages.

JPA vs Hibernate

JPA is the interface and Hibernate is the implementation.




Spring Data JPA

Part of the large [Spring Data](#) family, Spring Data JPA is built as an abstraction layer over the JPA. So, we have all the features of JPA plus the Spring ease of development.

For years, developers have written boilerplate code to create a [JPA DAO](#) for basic functionalities. Spring helps to significantly reduce this amount of code by providing minimal interfaces and actual implementations.

Spring JPA implementation

java

 Copy code

```
@Entity
@Table(name = "users")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "username")
    private String username;

    // Getters and setters, constructors
}
```

```
@Service
public class UserService {
    private final UserRepository userRepository;

    public UserService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    public List<User> getAllUsers() {
        return userRepository.getAllUsers();
    }

    public User getUserById(Long id) {
        return userRepository.getUserById(id);
    }

    public User saveUser(User user) {
        return userRepository.saveUser(user);
    }

    public void deleteUser(Long id) {
        userRepository.deleteUser(id);
    }
}
```

```
@RestController
@RequestMapping("/users")
public class UserController {
    private final UserService userService;

    public UserController(UserService userService) {
        this.userService = userService;
    }

    @GetMapping
    public List<User> getAllUsers() {
        return userService.getAllUsers();
    }

    @GetMapping("/{id}")
    public ResponseEntity<User> getUserById(@PathVariable Long id) {
        User user = userService.getUserById(id);
        if (user != null) {
            return ResponseEntity.ok(user);
        } else {
            return ResponseEntity.notFound().build();
        }
    }

    @PostMapping
    public ResponseEntity<User> createUser(@RequestBody User user) {
        User savedUser = userService.saveUser(user);
        return ResponseEntity.status(HttpStatus.CREATED).body(savedUser);
    }
}
```



```

@PostMapping
public ResponseEntity<User> createUser(@RequestBody User user) {
    User savedUser = userService.saveUser(user);
    return ResponseEntity.status(HttpStatus.CREATED).body(savedUser);
}

@DeleteMapping("/{id}")
public ResponseEntity<Void> deleteUser(@PathVariable Long id) {
    userService.deleteUser(id);
    return ResponseEntity.noContent().build();
}
}

```

In this plain JPA implementation, we create a `UserRepository` class that manages the database operations using the `EntityManager` provided by JPA. The repository class handles CRUD operations and uses JPQL (Java Persistence Query Language) queries to retrieve data. The `UserService` class handles business logic and utilizes the `UserRepository` to perform database operations. Finally, the `UserController` handles HTTP requests related to user operations, using the `UserService`.

Note that in a plain JPA implementation, you need to handle transaction management explicitly using `@Transactional` annotations or programmatic transaction demarcation. Additionally, you'll need to configure the JPA `EntityManagerFactory` and `DataSource` manually, which is typically done in a persistence configuration file.

```

@Repository
public class UserRepository {
    @PersistenceContext
    private EntityManager entityManager;

    public List<User> getAllUsers() {
        TypedQuery<User> query = entityManager.createQuery("SELECT u FROM U");
        return query.getResultList();
    }

    public User getUserById(Long id) {
        return entityManager.find(User.class, id);
    }

    public User saveUser(User user) {
        entityManager.persist(user);
        return user;
    }

    public void deleteUser(Long id) {
        User user = entityManager.find(User.class, id);
        if (user != null) {
            entityManager.remove(user);
        }
    }
}

```

@Transactional

The `@Transactional` annotation is used in Spring to define the transactional behavior of methods or classes. It is applied to methods or classes to specify that they should be executed within a transactional context. Transactions provide atomicity, consistency, isolation, and durability (ACID) properties to ensure data integrity in a database.

When the `@Transactional` annotation is applied to a method or class, Spring manages the transactional behavior, including the transaction boundaries, commit, and rollback operations. It is typically used in service or repository classes that interact with the database.

Here's an example of how to use `@Transactional` in a Spring Boot application:

```

@Service
public class UserService {
    private final UserRepository userRepository;

    public UserService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    @Transactional
    public List<User> getAllUsers() {
        return userRepository.findAll();
    }

    @Transactional
    public User saveUser(User user) {
        return userRepository.save(user);
    }

    @Transactional
    public void deleteUser(Long id) {
        userRepository.deleteById(id);
    }
}

```

In the above example, the `@Transactional` annotation is applied to the methods `getAllUsers()`, `saveUser()`, and `deleteUser()` in the `UserService` class. This means that these methods will execute within a transactional context. If an exception occurs, the transaction will be rolled back, reverting any changes made during the transaction.

The `@Transactional` annotation provides various attributes to customize the transactional behavior, such as `readOnly`, `propagation`, `isolation`, and `rollbackFor`, among others. These attributes allow you to define whether the transaction is read-only, how transactions are propagated, the isolation level, and which exceptions trigger a rollback.

It's important to note that `@Transactional` works with Spring's transaction management, which supports both JDBC and JPA transactions. To enable transaction management in a Spring Boot application, you typically need to configure a transaction manager bean and annotate the configuration class with `@EnableTransactionManagement`.

@Configuration

The `@Configuration` annotation is used in Spring to indicate that a class contains bean definitions and configuration logic. It is typically used in conjunction with other annotations like `@Bean`, `@ComponentScan`, or `@Import` to configure and bootstrap the Spring application context.

The `@Configuration` annotation serves as a replacement for XML-based configuration. It allows you to define beans and configure the application context using Java-based configuration instead of XML files.

@ComponentScan

The `@ComponentScan` annotation is used in Spring to enable component scanning and automatic bean detection within a specified package or packages. It allows Spring to scan and discover classes annotated with various stereotype annotations such as `@Component`, `@Service`, `@Repository`, `@Controller`, and more.

```
Java
@Configuration
@ComponentScan("com.example")
public class AppConfig {
    // Configuration logic and other bean definitions
}
```

In the above example, the `AppConfig` class is annotated with `@ComponentScan` and provided with the base package `"com.example"`. This instructs Spring to scan the specified package and its sub-packages to detect classes annotated with stereotype annotations and register them as beans in the application context.

@SpringBootConfiguration

The `@SpringBootConfiguration` annotation is a specialized form of the `@Configuration` annotation in the Spring Boot framework. It is used to mark a class as the primary configuration class for a Spring Boot application.

Here's a brief explanation of `@SpringBootConfiguration` and its usage:

```
Java
@SpringBootConfiguration
public class AppConfig {
    // Configuration logic and other bean definitions
}
```

In the above example, the `AppConfig` class is annotated with `@SpringBootApplication`, indicating that it serves as the primary configuration class for the Spring Boot application. This class typically contains the main configuration logic and bean definitions for the application. The `@SpringBootApplication` annotation is a meta-annotation that combines the functionality of the `@Configuration` and `@Component` annotations. It allows the class to be used as a configuration class and be scanned by Spring's component scanning mechanism. By using `@SpringBootApplication`, you ensure that the class is recognized as a configuration class by Spring Boot, and it is automatically picked up during the application startup process. In most cases, when creating a Spring Boot application, the main class annotated with `@SpringBootApplication` (which is itself a meta-annotation combining `@Configuration`, `@EnableAutoConfiguration`, and `@ComponentScan`) serves as the primary configuration class. However, you can also have additional configuration classes annotated with `@SpringBootApplication` to provide additional configurations and bean definitions.

@Component

The `@Component` annotation is a fundamental annotation in Spring that is used to mark a class as a Spring-managed component. It indicates that a class is a candidate for auto-detection and bean registration in the Spring application context.

@EnableAutoConfiguration

The `@EnableAutoConfiguration` annotation is a key annotation in Spring Boot that automatically configures the Spring application context based on the classpath and defined dependencies. It enables automatic configuration of beans and other components based on the application's classpath and the dependencies present.

When `@EnableAutoConfiguration` is present, Spring Boot scans the classpath for configuration classes, auto-configures beans, and sets up infrastructure components such as data sources, transaction managers, and others based on sensible defaults and predefined rules.

The auto-configuration process utilizes Spring Boot's starter dependencies, which include pre-packaged configurations for common use cases such as database access, web applications, security, and more. By having these starters on the classpath, the corresponding auto-configuration is automatically applied.

@PathVariable

The `@PathVariable` annotation is used in Spring MVC to bind a method parameter to a path variable in the request URL. It allows you to extract dynamic values from the URL and use them as method arguments in your Spring MVC controller.

When a request is made to `/users/123`, the value `123` will be extracted from the path and passed as an argument to the `getUserById` method. You can then use the `id` value within the method implementation to retrieve the corresponding user.

The `@PathVariable` annotation supports various options for customization, such as specifying the variable name and providing default values. Here's an example that demonstrates these

```

@RestController
@RequestMapping("/users")
public class UserController {
    @GetMapping("/{id}")
    public ResponseEntity<User> getUserById(@PathVariable Long id) {
        // Method implementation
    }
}

```

In this example, the `getUser` method has three path variables: `id`, `name`, and `optionalVariable`. The `@PathVariable("id")` and `@PathVariable("name")` annotations explicitly specify the names of the path variables. The `@PathVariable(required = false)` annotation marks `optionalVariable` as optional, allowing the method to handle requests with or without this path variable. By using `@PathVariable`, you can easily extract and use dynamic values from the request URL within your Spring MVC controller methods. It provides a flexible and convenient way to handle URL parameters and build dynamic RESTful APIs.

@ResponseBody

The `@ResponseBody` annotation is used in Spring MVC to indicate that the return value of a controller method should be serialized and included directly in the HTTP response body. It tells Spring MVC to skip the view resolution process and instead write the return value directly to the response.

```

@RestController
@RequestMapping("/users")
public class UserController {
    @GetMapping("/{id}")
    @ResponseBody
    public ResponseEntity<User> getUserById(@PathVariable Long id) {
        // Method implementation
    }
}

```

In the above example, the `getUserById` method is annotated with `@GetMapping` to handle HTTP GET requests to the `/users/{id}` URL pattern. The `@ResponseBody` annotation is used to indicate that the return value of the method should be written directly to the response body.

@Temporal

In Spring Boot, the `@Temporal` annotation is not specifically related to the framework itself. Instead, it is an annotation provided by the Java Persistence API (JPA) to indicate how a temporal attribute of an entity should be mapped to the corresponding database column. The `@Temporal` annotation is used in conjunction with the `java.util.Date` or `java.util.Calendar` types to specify the desired temporal precision. It can have two possible values:

- **TemporalType.DATE:** This value indicates that only the date component should be stored in the database column, excluding the time component.
- **TemporalType.TIMESTAMP:** This value indicates that both the date and time components should be stored in the database column.

@Enumerated

In Spring Boot and the Java Persistence API (JPA), the `@Enumerated` annotation is used to specify how an enumeration type (enum) should be mapped to the corresponding database column.

The `@Enumerated` annotation can be applied to an enum field or property within an entity class. It has two possible values:

- **EnumType.ORDINAL:** This value indicates that the enum should be stored in the database column as an integer representing the ordinal value of the enum constant. The ordinal values are based on the order in which the enum constants are declared.
- **EnumType.STRING:** This value indicates that the enum should be stored in the database column as a string representing the name of the enum constant.

Idempotency and safety

Idempotency and safety are two properties that describe how a REST API behaves when it receives a request from a client. Idempotency means that sending the same request multiple times will produce the same result, without changing the state of the server or the resource.

Security filter

A security filter describes a set of records in a table that a user has permission to access. You can specify, for example, that a user can only read the records that contain information about a particular customer. This means that the user cannot access the records that contain information about other customers.

Reducer

Operation that returns one value by combining the elements of a stream
`collect()` can only work with mutable result objects. `reducer()` is designed to work with immutable result objects

Bean profile

Propagation

Custom Exception

Beans scopes

RestController vs controller

Beans profile

CompletableFuture in Java

The basic concept behind `CompletableFuture` in Java is to provide a way to represent and manipulate asynchronous computations, allowing you to write code that can perform tasks concurrently and handle their results in a non-blocking manner.

flatMap

Functional programming(streams API's, lambda's)

Functional programming is a programming paradigm that emphasizes the use of pure functions, immutability, and higher-order functions. It aims to solve problems by modeling computations as the evaluation of mathematical functions and avoiding changing state and mutable data.

Lambda Expressions: Lambda expressions in Java provide a concise way to represent anonymous functions. They enable you to pass behavior as a method argument or declare it inline.

Functional Interfaces: Functional interfaces in Java are interfaces that have only one abstract method. They serve as the basis for lambda expressions and method references.

Streams API: The Streams API in Java provides a declarative and functional approach to process collections of data. A stream represents a sequence of elements and supports various operations such as filtering, mapping, reducing, and collecting. Streams can be created from various sources like collections, arrays, or I/O channels.

Rest API's revisit

REST (Representational State Transfer) is an architectural style for designing networked applications.

Stateless, Resource-Based, CRUD Operations, Uniform Interface, Hypermedia as the Engine of Application State (HATEOAS): HATEOAS is a constraint of RESTful APIs that allows clients to navigate the API's resources by following hyperlinks provided in the response. This enables self-discoverability of API resources and actions.

List and Sets use case in hibernate

Set is fast as compare to list

orphan removal

- If **orphanRemoval=true** is specified the disconnected Address instance is automatically removed. This is useful for cleaning up dependent objects (e.g. Address) that should not exist without a reference from an owner object (e.g. Employee).
- If only **cascade=CascadeType.REMOVE** is specified no automatic action is taken since disconnecting a relationship is not a remove operation

Circuit breaker

Volatile vs transient

- **volatile**: In Java, the **volatile** keyword is used to mark a variable as "volatile," which means that its value may be modified by multiple threads. It ensures that any thread reading the variable will see the most recent value written to it by any other thread, preventing thread-caching of the variable. It makes changes in the main memory.
- **transient**: In Java, the **transient** keyword is used to indicate that a variable should not be serialized when an object is converted to a byte stream (e.g., during Java object serialization). When deserializing the object, the transient variable will be initialized to its default value (e.g., null for objects, zero for numeric types).

Spring quartz

- Quartz has a modular architecture. It consists of several basic components that we can combine as required. In this tutorial, we'll focus on the ones that are common to every job: **Job**, **JobDetail**, **Trigger** and **Scheduler**.
- Although we'll use Spring to manage the application, each individual component can be configured in two ways: the **Quartz** way or the **Spring** way (using its convenience classes).
- We'll cover both options as far as possible, for the sake of completeness, but we may adopt either. Now let's start building, one component at a time.

Spring context

- application
- beans

Spring batch implementation

- Spring Batch is a lightweight, comprehensive batch processing framework provided by the Spring ecosystem. It facilitates the development of batch applications that process large volumes of data efficiently. It supports reading, processing, and writing data in

chunks, making it suitable for various batch processing scenarios, such as ETL (Extract, Transform, Load) operations or periodic data processing jobs.

Idempotent methods

post+ patch non idempotent

HTTP methods like POST and PATCH are considered non-idempotent. This means that their effects are not guaranteed to be the same if called multiple times with the same parameters. For instance:

- **POST**: Used to submit data to be processed to a specified resource. Repeatedly sending the same POST request may result in multiple resource creations or other state changes, depending on the server implementation.
- **PATCH**: Used to apply partial modifications to a resource. As with POST, sending the same PATCH request multiple times may lead to different outcomes if the server state changes between requests.

Try with resource block

- The **try-with-resources** statement is a feature introduced in Java 7 that simplifies the management of resources that need to be closed explicitly (e.g., streams, database connections) to avoid resource leaks. It ensures that the resources declared within the try block are automatically closed, whether the code within the block executes successfully or throws an exception.

Optional keyword

In Java, the **Optional** class, introduced in Java 8, provides various methods to work with potentially absent values in a more functional and expressive way. These methods allow you to handle the presence or absence of a value without resorting to explicit null checks, thus reducing the chances of encountering null pointer exceptions. Below are some of the important methods provided by the **Optional** class and their purposes:

1. **of(value)**: Creates an **Optional** instance containing the specified non-null value. If the provided value is **null**, it will throw a **NullPointerException**.
2. **ofNullable(value)**: Creates an **Optional** instance containing the specified value. It can accept a null value, and if the value is null, the resulting **Optional** will represent an empty (absent) value.
3. **empty()**: Creates an empty **Optional** instance, representing the absence of a value.
4. **isPresent()**: Returns **true** if the **Optional** contains a non-null value, otherwise returns **false**.
5. **ifPresent(consumer)**: Executes the specified consumer function with the value inside the **Optional**, if it is present.

6. `orElse(other)`: Returns the value contained in the `Optional` if it is present, otherwise returns the provided default value (`other`).
7. `orElseGet(supplier)`: Returns the value contained in the `Optional` if it is present, otherwise calls the provided supplier function to get a default value.
8. `orElseThrow(exceptionSupplier)`: Returns the value contained in the `Optional` if it is present, otherwise throws an exception created by the provided supplier function.
9. `filter(predicate)`: If the `Optional` contains a value and it satisfies the given predicate, returns the same `Optional`. Otherwise, returns an empty `Optional`.
10. `map(mapper)`: If the `Optional` contains a value, applies the given mapper function to it and returns a new `Optional` containing the result. If the `Optional` is empty, it returns an empty `Optional`.
11. `flatMap(mapper)`: Similar to `map`, but the provided mapper function returns an `Optional` itself. If the `Optional` contains a value, the result is the `Optional` returned by the mapper. If the `Optional` is empty, it returns an empty `Optional`.

Finally block is to deallocate the resources and it automatically deallocates resources we don't have to specifically write the code to deallocate.

trace id and span id

Hystrix dashboard, eureka server, keycloak, interservice communication

Rest Template vs Web Client

`RestTemplate` and `WebClient` are both classes in Spring Framework used to make HTTP requests to external services or APIs. They serve similar purposes but have different designs and are intended for different use cases.

RestTemplate: `RestTemplate` is a synchronous HTTP client that has been part of Spring Framework since its early versions. It is built on the `HttpClient` from Apache Commons and provides a simple and straightforward way to make HTTP requests. However, it's a blocking client, meaning when you make a request, your application waits for the response before proceeding to the next line of code. This can lead to inefficient resource utilization, especially in scenarios where your application needs to handle concurrent requests.

WebClient: `WebClient` is a more modern and non-blocking HTTP client introduced in Spring WebFlux, which is the reactive programming module in Spring. It is designed to handle asynchronous requests and is based on Reactor, a reactive library. `WebClient` is non-blocking, meaning it can send multiple requests concurrently without blocking the execution thread. This makes it more suitable for high-performance and scalable applications, especially in scenarios where you have a lot of concurrent requests or need to perform operations like streaming.

Sleuth

Spring Cloud Sleuth is a distributed tracing solution provided by the Spring Cloud ecosystem. It is designed to integrate with Spring Boot applications and helps you trace and monitor requests as they flow through various microservices in a distributed system.

Sleuth is based on the OpenTracing standard and provides the following key features:

Trace and Span IDs: Sleuth generates and assigns unique trace and span IDs to each request. These IDs are used to correlate related requests across different microservices.

Automatic Instrumentation: Sleuth automatically instruments your Spring Boot application, including REST templates, message queues, and asynchronous processing. It adds tracing information to requests, making it easier to monitor the flow of requests across services.

Propagation: Sleuth ensures that trace and span IDs are propagated correctly between services. This allows you to follow the entire lifecycle of a request as it moves through multiple microservices.

Integration with Distributed Tracing Systems: Sleuth integrates seamlessly with popular distributed tracing systems like Jaeger, Zipkin, and others. These systems collect and aggregate tracing data from all microservices, enabling you to visualize and analyze the entire request flow.

TraceId

This is an id that is assigned to a single request, job, or action. Something like each unique user initiated web request will have its own traceId.

SpanId

Tracks a unit of work. Think of a request that consists of multiple steps. Each step could have its own spanId and be tracked individually. By default, any application flow will start with the same TraceId and SpanId.

flat map vs reducers in streams

Serializable

"Serializable" is an interface in Java and other programming languages that indicates an object's ability to be converted into a stream of bytes. The primary purpose of serialization is to enable objects to be easily saved to a file, sent over a network, or stored in a database, and later deserialized (reconstructed) back into their original form.

In Java, the Serializable interface is part of the java.io package, and it has no methods to implement. It is known as a "marker interface" since it only serves to mark a class as serializable. When a class implements the Serializable interface, it indicates that objects of that class can be converted into a byte stream using Java's built-in serialization mechanism.

```
import java.io.Serializable;

public class Person implements Serializable {
    private String name;
    private int age;

    // Constructor, getters, setters, and other methods

    // ...
}
```

To serialize an object in Java, you typically use an `ObjectOutputStream` to write the object to a file or any other output stream. Conversely, to deserialize an object, you use an `ObjectInputStream` to read the byte stream and reconstruct the object. It's important to note that not all classes can be serialized. To be serializable, a class must meet certain requirements, such as:

- The class must implement the `Serializable` interface.
- All non-transient fields within the class must also be serializable.
- The class must have a default no-argument constructor (a constructor with no parameters).

In some cases, you might want to control the serialization process or exclude certain fields from serialization. To achieve this, you can use the `transient` keyword to mark fields as non-serializable.

Pom.xml

Plugins, build, properties, jfrog

dependency management(to override the dependency we use dependency management, to override dependency in nested dependency) - when maven finds nested dependency it go through dependency management.

How we can define our repositories in pom.xml

We make settings file in .mvn

Redundant code, static code analyzer >> SonarQube

Spring security

feign client vs rest template

Jvm, jdk architecture

Dependency Management in pom.xml

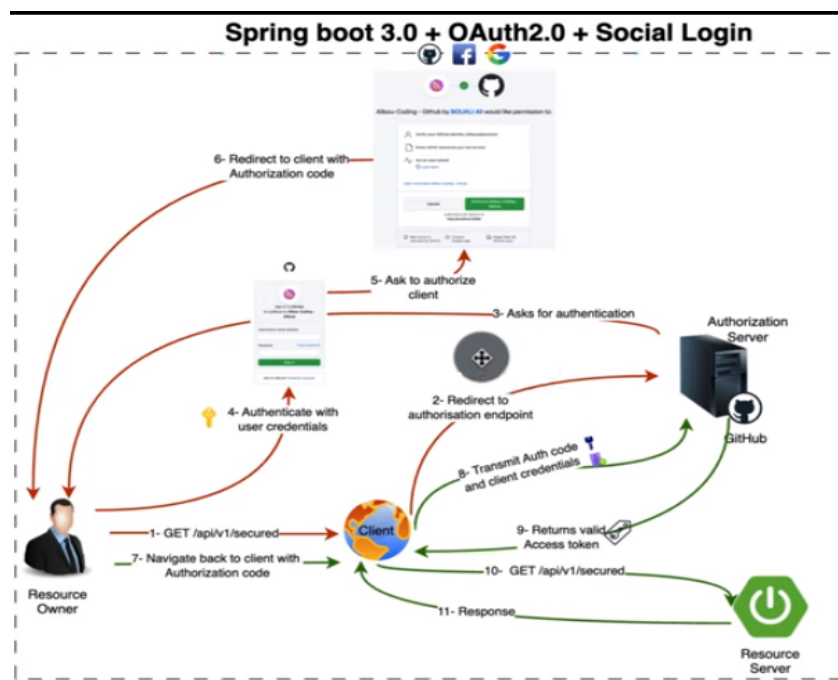
In a pom.xml file, which is used in Apache Maven projects, dependency management serves several important purposes related to managing the project's external dependencies.

Dependencies are external libraries or modules that your project relies on to build, run, and function properly. These dependencies might include third-party libraries, frameworks, and other modules that are needed to fulfill the project's requirements.

Here's what dependency management does in a pom.xml:

- **Centralizes Dependency Versions:** Dependency management allows you to define a set of common dependencies along with their versions in a centralized section of the pom.xml. This means you can specify the version number of a particular library in one place, and all modules or sub-projects within your Maven project can inherit that version. It helps avoid version conflicts and ensures consistency across the project.
- **Simplifies Version Updates:** If you need to update a particular library to a newer version, you can do it in the dependency management section, and all the modules using that dependency will automatically pick up the new version. This makes it easier to keep your project up-to-date with the latest versions of libraries.
- **Reduces Repetition:** Instead of specifying dependencies and their versions in each individual module or sub-project's pom.xml, you can define them once in the dependency management section. The modules can then declare their dependencies without explicitly specifying version numbers, as they inherit them from the management section.
- **Facilitates Dependency Exclusion:** Sometimes, two or more dependencies might depend on different versions of the same library, leading to conflicts. Dependency management can help resolve these conflicts by excluding specific transitive dependencies from certain modules while still using them in others.

OAuth 2.0 Flow



JPA Entities Configurations

- **MySQL:** org.hibernate.dialect.MySQLDialect
- **PostgreSQL:** org.hibernate.dialect.PostgreSQLDialect
- **Oracle:** org.hibernate.dialect.OracleDialect
- **Microsoft SQL Server:** org.hibernate.dialect.SQLServerDialect
- **H2 Database Engine:** org.hibernate.dialect.H2Dialect
- **SQLite:** org.hibernate.dialect.SQLiteDialect

spring.jpa.hibernate.ddl-auto

The spring.jpa.hibernate.ddl-auto property is used in Spring Boot applications to control the database schema generation behavior for Hibernate (the JPA provider). It specifies how Hibernate should create or update the database schema based on the entity mappings defined in your application.

The possible values for spring.jpa.hibernate.ddl-auto are:

none: This is the default option. Hibernate will not perform any schema generation or management. You are responsible for creating the database schema manually or using database migration tools.

create: Hibernate will drop and recreate the database schema every time the application starts. This means all existing data will be lost. It is typically used during development and testing.

create-drop: Similar to the create option, Hibernate will drop and recreate the database schema when the application starts. However, the schema will be dropped when the application shuts down as well, effectively removing all data.

update: Hibernate will attempt to update the existing database schema based on the entity mappings. It will add new tables, columns, and constraints, but it will not drop or modify existing columns or tables. This option is useful during development when you want to evolve your database schema alongside your application code.

validate: Hibernate will validate the database schema against the entity mappings but will not make any changes. If there are differences between the schema and the entity mappings, it will log an error.

Background jobs (CRON expressions)

Background jobs are tasks that run at specified intervals or times in the background of an application or system. These jobs are often used for periodic tasks, data processing, sending emails, generating reports, and other tasks that need to be executed on a regular basis without requiring user interaction. CRON expressions are commonly used to define the schedule for these background jobs.


A CRON expression is a string consisting of five or six fields, each representing different parts of the schedule. The format is as follows:

<second> <minute> <hour> <day-of-month> <month> <day-of-week>

- **<second>**: Specifies the second(s) at which the job should run (0-59).
- **<minute>**: Specifies the minute(s) at which the job should run (0-59).
- **<hour>**: Specifies the hour(s) at which the job should run (0-23).
- **<day-of-month>**: Specifies the day(s) of the month on which the job should run (1-31).
- **<month>**: Specifies the month(s) on which the job should run (1-12 or JAN-DEC).
- **<day-of-week>**: Specifies the day(s) of the week on which the job should run (0-7 or SUN-SAT, where both 0 and 7 represent Sunday).


Note: The fields for <day-of-month> and <day-of-week> are mutually exclusive. In most CRON implementations, if you specify a value for one of them, you should use '?' (question mark) for the other to indicate that it doesn't matter.

- Run the job every day at 3:30 AM:

 Copy code


```
0 30 3 * * ?
```

- Run the job every 15 minutes:

 Copy code


```
0 0/15 * * * ?
```

- Run the job every Monday and Friday at 8:00 PM:

 Copy code

```
0 0 20 ? * MON,FRI
```

- Run the job every weekday (Monday to Friday) at 9:00 AM:

 Copy code

```
0 0 9 ? * MON-FRI
```

CRON expressions can be used in various systems and programming languages to schedule background jobs. Popular frameworks and tools often have built-in support for CRON expressions or provide utilities to handle background job scheduling. For example, in Java, the Spring Framework has the `@Scheduled` annotation that allows you to define scheduled tasks

using CRON expressions. In UNIX-based systems, the cron utility uses CRON expressions to schedule tasks.

@Scheduled

In Java and specifically with the Spring Framework, `@Scheduled` is an annotation that is used to schedule the execution of a method at regular intervals. It allows you to easily define background tasks or jobs without the need for external schedulers. The `@Scheduled` annotation is part of Spring's Task Scheduling support.


To use `@Scheduled`, you need to enable task scheduling in your Spring Boot application. This can be done by adding the `@EnableScheduling` annotation at the configuration level (usually in the main class) or in a configuration class.

```
@SpringBootApplication
@EnableScheduling
public class MyApplication {
    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args);
    }
}
```

Once scheduling is enabled, you can use the `@Scheduled` annotation on a method to define its execution schedule.

Example of using `@Scheduled` to schedule a method:

java

 Copy code

```
import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.stereotype.Component;


@Component
public class MyScheduledTasks {

    @Scheduled(fixedRate = 5000) // Run every 5 seconds
    public void myTask() {
        // Code to be executed periodically
        System.out.println("Executing myTask at " + new Date());
    }
}
```

In this example, the `myTask()` method will be executed every 5 seconds due to the `fixedRate` attribute of `@Scheduled`. You can use various attributes of the `@Scheduled` annotation to define different scheduling options, such as `fixedDelay`, `initialDelay`, and `cron`.

Example using `cron` expression with `@Scheduled`:

java

 Copy code

```
@Scheduled(cron = "0 0 12 * * ?") // Run daily at 12:00 PM
public void dailyTask() {
    // Code to be executed daily at 12:00 PM
    System.out.println("Executing dailyTask at " + new Date());
}
```

Keep in mind that when using `@Scheduled`, the annotated method will be executed by default in the same thread as the scheduler. If the method's execution takes a long time, it might block subsequent executions. If you need more control or asynchronous execution, consider using Spring's `TaskExecutor` or `@Async` annotations.

Spring boot environments(dev, test, prod)

```
spring:
  profiles:
    active: dev
  security:
    oauth2:
      client:
        registration:
          github:
            client-id: Iv1.30f43960321cb71e
            client-secret: 49a1f26c988e6e245a675f003a7e6bb858b9c3ae
          google:
            client-id: 1012450397734-9oa1s64n62q8hvv49jvr5h162ak0bm5v.apps.googleusercontent.com
            client-secret: G0CSPX-8ZWvBhN4tIW06RtAZz7T4up11b5U
        user:
          password: password #for form based login so that we don't have to copy password from console again and again
          roles: ADMIN
          name: user
  org:
    jobrunr:
      background-job-server:
        enabled: true
      dashboard:
        enabled: true
```

Exception Handling in spring boot

@ControllerAdvice

@ControllerAdvice is a Spring annotation used in web applications to handle exceptions globally across multiple controllers. It allows you to define exception handling logic that will be applied to all controllers within the application.

When an exception occurs during the processing of a request, Spring will search for an appropriate exception handler method within the @ControllerAdvice-annotated class to handle that exception. This can help centralize and streamline exception handling in your application.

```
@ControllerAdvice
public class CustomizedResponseBodyExceptionHandler extends ResponseEntityExceptionHandler{

    @ExceptionHandler(Exception.class)
    public final ResponseEntity<ErrorDetails> handleAllExceptions(Exception ex, WebRequest request) throws Exception {
        ErrorDetails errorDetails = new ErrorDetails(LocalDate.now(),
            ex.getMessage(), request.getDescription( includeClientInfo: false));

        return new ResponseEntity<ErrorDetails>(errorDetails, HttpStatus.INTERNAL_SERVER_ERROR);
    }

    @ExceptionHandler(UserNotFoundException.class)
    public final ResponseEntity<ErrorDetails> handleUserNotFoundException(Exception ex, WebRequest request) throws Exception {
        ErrorDetails errorDetails = new ErrorDetails(LocalDate.now(),
            ex.getMessage(), request.getDescription( includeClientInfo: false));

        return new ResponseEntity<ErrorDetails>(errorDetails, HttpStatus.NOT_FOUND);
    }

    no usages
    protected ResponseEntity<Object> handleMethodArgumentNotValid(
        MethodArgumentNotValidException ex, HttpHeaders headers, HttpStatusCode status, WebRequest request) {

        ErrorDetails errorDetails = new ErrorDetails(LocalDate.now(),
            message: ex.getErrorCount() + " First Error: " + ex.getFieldError().getDefaultMessage(), request.getDescription( includeClientInfo: false));

        return new ResponseEntity(errorDetails, HttpStatus.BAD_REQUEST);
    }
}
```

Spring AOP

Spring AOP (Aspect-Oriented Programming) is a powerful feature of the Spring Framework that allows you to modularize cross-cutting concerns in your application. Cross-cutting concerns are functionalities that are spread across multiple modules and do not fit well with the primary business logic, such as logging, security, transaction management, and caching.

Spring AOP achieves this by dynamically injecting additional behavior into the existing code at specific join points, which are predefined points in the program's execution flow. The injected behavior is defined in separate modules called aspects.

Key Concepts in Spring AOP:

Aspect: An aspect is a module that encapsulates cross-cutting concerns. It defines the additional behavior that needs to be executed at specific join points. Aspects are represented using a combination of pointcuts and advice.

Pointcut: A pointcut is an expression that defines where the additional behavior should be applied in the application's execution flow. It specifies the join points in the code where the advice will be executed.

Advice: Advice is the actual behavior that is executed at a specific join point. There are different types of advice:

- **Before advice:** Execute before a join point.
- **After returning advice:** Executed after a join point successfully completes.
- **After throwing advice:** Executed after a join point throws an exception.
- **After (finally) advice:** Executed after a join point, regardless of success or failure.
- **Around advice:** Wraps the join point, allowing the advice to control the method's invocation.

Join Point: A join point is a specific point in the application's execution flow where an aspect can be applied. Examples of join points are method calls, field access, object instantiation, and exception handling.

Weaving: Weaving is the process of applying aspects to the target object to create the advised object. There are two types of weaving in Spring AOP:

- **Compile-time weaving:** Aspects are woven during the compilation process.
- **Runtime weaving:** Aspects are woven at runtime using proxies or bytecode manipulation.

How to avoid code duplication in microservices

Using centralized services,

Thread Lifecycle

new -> runnable -> running -> blocked -> dead(from any state when stop() is called)

prefer constructor injection

- Once the dependencies are set they cannot be changed.
- when we create jar file the other developer cannot see the private field, he can only see the constructor and its dependencies

running dependency on custom configuration

- exclude alias in spring boot autoconfiguration
- `@ConditionalOnMissingClass()` -> `boot.autoconfigure.condition`

running application through command line runner

- `mvn: spring boot run`
- `mvn spring-boot:run -Dspring-boot.run.arguments="--server.port=8081"`
- `java -jar your-app.jar --server.port=8081`

layers in hibernate

- Entities
- Session Factory

- Session
- Transaction Management
- Data Access Object
- Database

default logger

Logback

`spring-boot-starter-logging`

@Value import

Unset

`org.springframework.beans.factory.annotation`

read-only data using transaction

@Transaction(read-only=true)

Spring boot components

- starter
- auto configurator
- CLI
- actuator

we can run spring boot as java application using different ways

- Using `main` Method
- Using Maven or Gradle
- Using `java -jar`
- Using IDE
- Using Docker

maven vs Gradle

how spring boot simplify testing

@SpringBootTest

Junit3 vs Junit4

setup vs tier-down

test private functions

- by making public wrapper functions which calls the private functions

checkout also do discard

git combining multiple commits

- git rebase -i HEAD~n
- pick (to keep commit as default) , squash(combines commit with previous commit), fixup(same as squash, but discard the commit message)
- git add
- git rebase --continue
- git push --force

reverting commit from remote branch

```
# Fetch the latest changes from the remote repository
git fetch origin
```

```
# Checkout the branch you want to revert the commit on
git checkout <branch_name>
```

```
# Create a new commit that reverts the changes from the target commit
git revert <commit_hash>
```

```
# Push the revert commit to the remote repository
git push origin <branch_name>
```

reverting commit from local

```
# Checkout the branch you want to revert the commit on
git checkout <branch_name>
```

```
# Create a new commit that reverts the changes from the target commit
git revert <commit_hash>
```

OR

```
# Checkout the branch you want to revert the commit on
git checkout <branch_name>
```

```
# Reset the branch to the commit before the target commit
git reset --hard HEAD~1
```

@Retention :

Specifies whether the annotation metadata can be accessed at runtime by the application (will determine whether the compiled bytecode is affected)