SPRING & SPRING BOOT

[100+ Spring Framework, Spring Boot, Cloud, Data JPA and Spring Security Interview Questions in 2023 | by javinpaul | Javarevisited | Medium]

ANNOTATIONS

Some commonly used annotations in Spring and Spring Boot:

Spring Core Annotations:

- @Component: Indicates that a class is a Spring-managed component. It marks a class as a Spring bean that can be automatically detected and registered in the Spring application context. It serves as a generic stereotype for any Spring-managed component.
- @Autowired: Performs automatic dependency injection by searching for beans that match the required type. It can be used with constructors, methods, or fields.
- @Qualifier: Helps resolve ambiguities when multiple beans of the same type are available for autowiring. This annotation helps specify which particular bean should be autowired.
- @Value: Injects values from property files or environment variables into variables or constructor arguments.

Spring MVC Annotations:

- @Controller: Marks a class as a Spring MVC controller. It is responsible for handling HTTP requests and generating responses.
- @RequestMapping: Maps HTTP requests to specific controller methods.
- @PathVariable: Extracts values from the request URI and binds them to method parameters.
- @RequestParam: Retrieves query parameters from the request URL and binds them to method parameters.
- @ResponseBody: Converts the return value of a method into an HTTP response body.
- @ModelAttribute: Binds request data to a method parameter or model attribute.

Spring Boot Annotations:

- @SpringBootApplication: Annotates the main class of a Spring Boot application, enabling auto-configuration and component scanning. It combines @Configuration, @EnableAutoConfiguration, and @ComponentScan.
- @RestController: Combines @Controller and @ResponseBody annotations, making it suitable for building RESTful APIs. It is used to create RESTful controllers that return data in the response body.
- @EnableAutoConfiguration: Enables auto-configuration based on the classpath and dependencies.
- @ConfigurationProperties: Binds external configuration properties to a Java class. In other word, it binds properties from the application properties or YAML files to a POJO (Plain Old Java Object)

- @ConditionalOnProperty: Conditionally enables a bean or configuration based on the presence or value of a specific property.
- @EnableCaching: Enables Spring's caching support in the application.

Spring Data Annotations:

- @Repository: Marks an interface/class as a Spring Data repository. It is used for database access and works with Spring's data access exception translation mechanism.
- @Entity: Marks a class as a JPA entity, representing a table in the database.
- @Table: Specifies the table name for an entity class in the database.
- @Id: Marks a field as the primary key of an entity.
- @Query: Defines custom queries using JPQL, SQL, or other query languages.

Spring Security Annotations:

- @EnableWebSecurity: Enables Spring Security for web applications.
- @Secured: Secures a method or class with defined roles/authorities.
- @PreAuthorize: Specifies a pre-authorization expression for method-level security.
- @Configuration: Marks a class as a source of bean definitions and configurations. It is used in conjunction with @Bean to define beans.

WHAT IS SPRING

- Application framework
- Programming & Configuration model
- Provides infrastructure support

Problems with Spring

- Huge Framework
- Multiple setup steps
- Multiple configuration steps
- Multiple build & deploy steps

SPRING BOOT

- Opinionated
- Convenient over configuration
- Stand-alone
- Production ready

MAVEN: [Maven – Introduction (apache.org)]
Maven is a build and dependency management tool that is based on POM (project object model - maven configuration file). It is used for projects build, dependency and documentation.

- It makes a project easy to build
- It provides uniform build process (maven project can be shared by all the maven projects)
- It provides project information (log document, cross referenced sources, mailing list, dependency list, unit test reports etc.)
- It is easy to migrate for new features of Maven

Transitive Dependency: When one dependency is dependent on another dependency that is called Transitive Dependency. For eg. we need mysql and we added a mysql dependency in POM but mysql also needs another dependency which we don't need.

What is the difference between artifactId & groupId in pom.xml?

- **groupId** uniquely identifies your project across all projects. A group ID should follow Java's package name rules. This means it starts with a reversed domain name you control. eg, org.apache.maven, org.apache.commons

  Maven does not enforce this rule. There are many legacy projects that do not follow this convention and instead use single word group IDs. However, it will be difficult to get a new single word group ID approved for inclusion in the Maven Central repository.

  You can create as many subgroups as you want. A good way to determine the granularity of the groupId is to use the project structure. That is, if the current project is a multiple module project, it should append a new identifier to the parent's groupId. eg, org.apache.maven, org.apache.maven.plugins, org.apache.maven.reporting
- **artifactId** is the name of the jar without version. If you created it, then you can choose whatever name you want with lowercase letters and no strange symbols. If it's a third party jar, you have to take the name of the jar as it's distributed. eg. maven, commons-math
- **version** if you distribute it, then you can choose any typical version with numbers and dots (1.0, 1.1, 1.0.1, ...). Don't use dates as they are usually associated with SNAPSHOT (nightly) builds. If it's a third party artifact, you have to use their version number whatever it is, and as strange as it can look. eg, 2.0, 2.0.1, 1.3.1

What is Archetype?

Archetype is a Maven project templating toolkit. An archetype is defined as an original pattern or model from which all other things of the same kind are made. The name fits as we are trying to provide a system that provides a consistent means of generating Maven projects. Archetype will help authors create Maven project templates for users, and provides users with the means to generate parameterized versions of those project templates

How does Maven work?

When we add any dependency in pom.xml, basically we're asking maven that we need this dependency. Now the question is how maven will know from where to get these dependencies. So what it does is it first searches for the file in our local machine, but where? So basically in every machine, when we have a maven installed, will have something called a M2 folder(.m2). In that M2 folder we will see the option of repository. If we are using Maven for the first time, we will not have any folders there.

So what is happening is every time we are asking for any dependency, maven first searches for that thing in the local machine first in the M2 folder. If it is there, that's great. What if it is not there? In that case, it goes to the Maven Central and from there it downloads the dependency. And next time when we want to use it, we have a copy of it in our local machine.

Maven lifecycle:

- *clean* — delete target directory
- *validate* — validate, if the project is correct
- *compile* — compile source code, classes stored in target/classes
- *test* — run tests
- *package* — take the compiled code and package it in its distributable format, e.g. JAR, WAR
- verify — run any checks to verify the package is valid and meets quality criteria
- *install* — install the package into the local repository
- deploy — copies the final package to the remote repository

Maven phase cmd(project build cmd):

- clean project: This command will delete target directory -> mvn clean
- validate project: validate the project is correct and all necessary information is available -> mvn validate
- compile project: compile source code, classes stored in target/classes -> mvn compile
- test project: run tests using a suitable unit testing framework -> mvn test
- package project: take the compiled code and package it in its distributable format, such as a JAR /WAR -> mvn package
- verify project: run any checks to verify the package is valid and meets quality criteria -> mvn verify
- install project: install the package into the local repository, for use as a dependency in other projects locally -> mvn install
- deploy project: done in an integration or release environment, copies the final package to the remote repository for sharing with other developers and projects -> mvn deploy

Skip running tests:

- Compiles the tests, but skips running them -> mvn install -DskipTests=true
- Skips compiling the tests and does not run them -> mvn install -Dmaven.test.skip=true

Project site generation:

- Generate site without tests reports (tests are not executed) -> mvn site:site
- Generate site with unit tests reports -> mvn test site:site
- Generate site with unit and integration tests reports -> mvn verify site:site

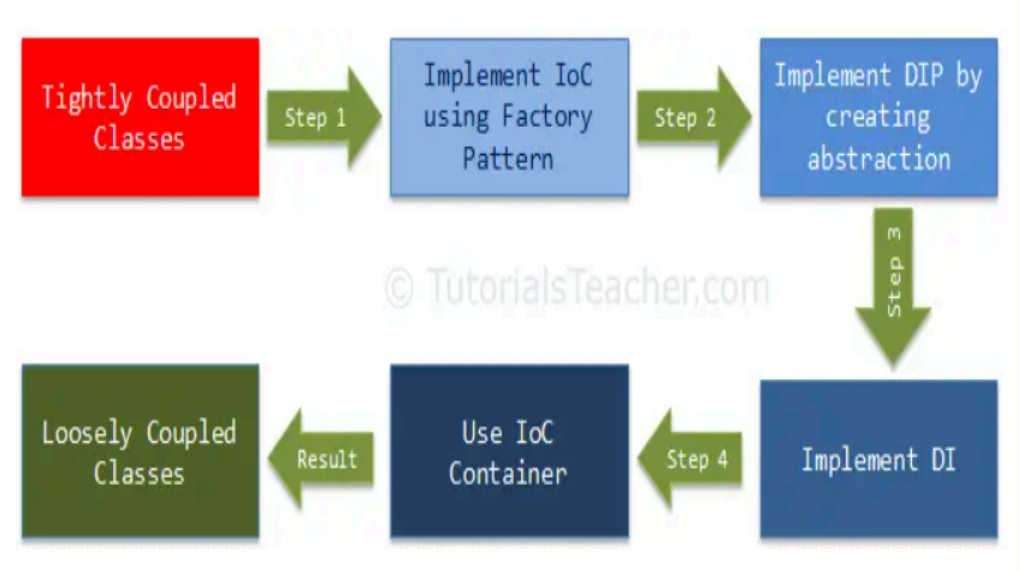
Code quality analysis:

- Analyse code quality with Sonar ->
  mvn clean install -DskipTests=true
  mvn sonar:sonar

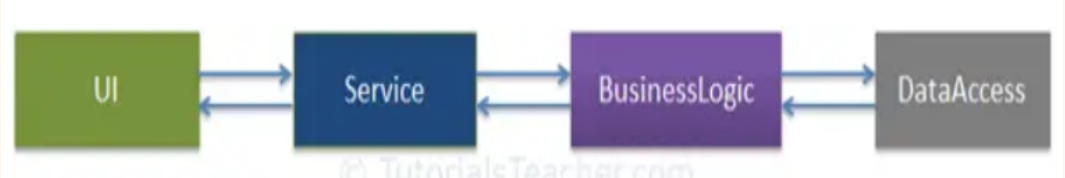
BEAN:
In Spring, the objects that form the backbone of your application and that are managed by the Spring IoC container are called beans. A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container.

INVERSION OF CONTROL(IoC): **Introduction IoC, DIP, DI and IoC Container (tutorialsteacher.com)**
IoC is a design principle which recommends the inversion of different kinds of controls in object-oriented design to achieve loose coupling between application classes. Here, controls refer to any additional responsibilities a class has, other than its main responsibility. This includes control over the flow of an application, and control over the flow of an object creation or dependent object creation and binding.

In layman's terms, suppose you drive a car to your workplace. This means you control the car. The IoC principle suggests to invert the control, meaning that instead of driving the car yourself, you hire a cab, where another person will drive the car. Thus, this is called inversion of the control - from you to the cab driver. You don't have to drive a car yourself and you can let the driver do the driving so that you can focus on your main work.



In an object-oriented design, classes should be designed in a loosely coupled way. Loosely coupled means changes in one class should not force other classes to change, so the whole application can become maintainable and extensible.



In the typical n-tier architecture, the User Interface (UI) uses Service layer to retrieve or save data. The Service layer uses the BusinessLogic class to apply business rules on the data. The BusinessLogic class depends on the DataAccess class which retrieves or saves the data to the underlying database. This is a simple n-tier architecture design.
Let's focus on the BusinessLogic and DataAccess classes to understand IoC.
The following is an example of BusinessLogic and DataAccess classes for a customer.

```
public class CustomerBusinessLogic
{
    DataAccess _dataAccess;

    public CustomerBusinessLogic()
    {
        _dataAccess = new DataAccess();
    }

    public string GetCustomerName(int id)
    {
        return _dataAccess.GetCustomerName(id);
    }
}

public class DataAccess
{
    public DataAccess()
    {
    }

    public string GetCustomerName(int id) {
        return "Dummy Customer Name"; // get it from DB in real app
    }
}
```

In the above example, CustomerBusinessLogic and DataAccess are tightly coupled classes because the CustomerBusinessLogic class includes the reference of the concrete DataAccess class. It also creates an object of the DataAccess class and manages the lifetime of the object.

Problems in the above example classes:

1. CustomerBusinessLogic and DataAccess classes are tightly coupled classes. So, changes in the DataAccess class will lead to changes in the CustomerBusinessLogic class. For example,

if we add, remove or rename any method in the DataAccess class then we need to change the CustomerBusinessLogic class accordingly.

2. Suppose the customer data comes from different databases or web services and, in the future, we may need to create different classes, so this will lead to changes in the CustomerBusinessLogic class.
3. The CustomerBusinessLogic class creates an object of the DataAccess class using the new keyword. There may be multiple classes which use the DataAccess class and create its objects. So, if you change the name of the class, then you need to find all the places in your source code where you created objects of DataAccess and make the changes throughout the code. This is repetitive code for creating objects of the same class and maintaining their dependencies.
4. Because the CustomerBusinessLogic class creates an object of the concrete DataAccess class, it cannot be tested independently (TDD). The DataAccess class cannot be replaced with a mock class.

Solve above problems:

To solve all of the above problems and get a loosely coupled design, we can use the IoC and DIP principles together. Let's use the *Factory* pattern to implement IoC in the above example, as the first step towards attaining loosely coupled classes.

First, create a simple Factory class which returns an object of the DataAccess class as shown below.

```
public class DataAccessFactory
{
    public static DataAccess GetDataAccessObj()
    {
        return new DataAccess();
    }
}
```

Now, use this DataAccessFactory class in the CustomerBusinessLogic class to get an object of DataAccess class.

```
public class CustomerBusinessLogic
{

    public CustomerBusinessLogic()
    {
    }

    public string GetCustomerName(int id)
    {
        DataAccess _dataAccess =  DataAccessFactory.GetDataAccessObj();

        return _dataAccess.GetCustomerName(id);
    }
}
```

The CustomerBusinessLogic class uses the DataAccessFactory.GetCustomerDataAccessObj() method to get an object of the DataAccess class instead of creating it using the *new* keyword. Thus, we have inverted the control of creating an object of a dependent class from the CustomerBusinessLogic class to the DataAccessFactory class.

DEPENDENCY INJECTION PRINCIPLE(DPI):

1. High-level modules should not depend on low-level modules. Both should depend on the abstraction
2. Abstractions should not depend on details. Details should depend on abstractions

As per the DIP definition, a high-level module should not depend on low-level modules. Both should depend on abstraction. So, first, decide which is the high-level module (class) and the low-level module. A high-level module is a module which depends on other modules. In our example, CustomerBusinessLogic depends on the DataAccess class, so CustomerBusinessLogic is a

high-level module and DataAccess is a low-level module. So, as per the first rule of DIP, CustomerBusinessLogic should not depend on the concrete DataAccess class, instead both classes should depend on abstraction. The second rule in DIP is "Abstractions should not depend on details. Details should depend on abstractions".
In English, abstraction means something which is non-concrete. In programming terms, the above CustomerBusinessLogic and DataAccess are concrete classes, meaning we can create objects of them. So, abstraction in programming means to create an interface or an abstract class which is non-concrete. This means we cannot create an object of an interface or an abstract class. As per DIP, CustomerBusinessLogic (high-level module) should not depend on the concrete DataAccess class (low-level module). Both classes should depend on abstractions, meaning both classes should depend on an interface or an abstract class.
Now, what should be in the interface (or in the abstract class)? As you can see, CustomerBusinessLogic uses the GetCustomerName() method of the DataAccess class (in real life, there will be many customer-related methods in the DataAccess class). So, let's declare the GetCustomerName(int id) method in the interface, as shown below.

```
public interface ICustomerDataAccess
{
    string GetCustomerName(int id);
}
```

Now, implement ICustomerDataAccess in the CustomerDataAccess class, as shown below (so, instead of the DataAccess class, let's define the new CustomerDataAccess class).

```
public class CustomerDataAccess: ICustomerDataAccess
{
    public CustomerDataAccess()
    {
    }

    public string GetCustomerName(int id) {
        return "Dummy Customer Name";
    }
}
```

Now, we need to change our factory class which returns ICustomerDataAccess instead of the concrete DataAccess class, as shown below.

```
public class DataAccessFactory
{
    public static ICustomerDataAccess GetCustomerDataAccessObj()
    {
        return new CustomerDataAccess();
    }
}
```

Now, change the CustomerBusinessLogic class which uses ICustomerDataAccess instead of the concrete DataAccess class as shown below.

```
public class CustomerBusinessLogic
{
    ICustomerDataAccess _custDataAccess;

    public CustomerBusinessLogic()
    {
        _custDataAccess = DataAccessFactory.GetCustomerDataAccessObj();
    }

    public string GetCustomerName(int id)
    {
        return _custDataAccess.GetCustomerName(id);
    }
}
```

Thus, we have implemented DIP in our example where a high-level module (CustomerBusinessLogic) and low-level module (CustomerDataAccess) are dependent on an

abstraction (ICustomerDataAccess). Also, the abstraction (ICustomerDataAccess) does not depend on details (CustomerDataAccess), but the details depend on the abstraction.
The following is the complete DIP example discussed so far.

```
public interface ICustomerDataAccess
{
    string GetCustomerName(int id);
}

public class CustomerDataAccess: ICustomerDataAccess
{
    public CustomerDataAccess() {
    }

    public string GetCustomerName(int id) {
        return "Dummy Customer Name";
    }
}

public class DataAccessFactory
{
    public static ICustomerDataAccess GetCustomerDataAccessObj()
    {
        return new CustomerDataAccess();
    }
}

public class CustomerBusinessLogic
{
    ICustomerDataAccess _custDataAccess;

    public CustomerBusinessLogic()
    {
        _custDataAccess = DataAccessFactory.GetCustomerDataAccessObj();
    }

    public string GetCustomerName(int id)
    {
        return _custDataAccess.GetCustomerName(id);
    }
}
```

The advantages of implementing DIP in the above example is that the CustomerBusinessLogic and CustomerDataAccess classes are loosely coupled classes because CustomerBusinessLogic does not depend on the concrete DataAccess class, instead it includes a reference of the ICustomerDataAccess interface. So now, we can easily use another class which implements ICustomerDataAccess with a different implementation.

DEPENDENCY INJECTION(DI):
Dependency Injection (DI) is a design pattern used to implement IoC. It allows the creation of dependent objects outside of a class and provides those objects to a class through different ways. Using DI, we move the creation and binding of the dependent objects outside of the class that depends on them.
Dependency injection (DI) is a process whereby objects define their dependencies (that is, the other objects with which they work) only through constructor arguments, arguments to a factory method, or properties that are set on the object instance after it is constructed or returned from a factory method. The container then injects those dependencies when it creates the bean. This process is fundamentally the inverse (hence the name, Inversion of Control) of the bean itself controlling the instantiation or location of its dependencies on its own by using direct construction of classes or the Service Locator pattern.

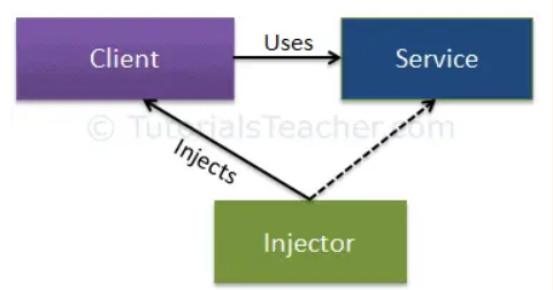
Need For DI:

Code is cleaner with the DI principle and decoupling is more effective when objects are provided with their dependencies. The object does not look up its dependencies and does not know the

location or class of the dependencies. As a result, your classes become easier to test, particularly when the dependencies are on interfaces or abstract base classes, which allow for stub or mock implementations to be used in unit tests.

The Dependency Injection pattern involves 3 types of classes:

1. Client Class: The client class (dependent class) is a class which depends on the service class
2. Service Class: The service class (dependency) is a class that provides service to the client class
3. Injector Class: The injector class injects the service class object into the client class



Dependency Injection

Types of Dependency Injection:

1. Constructor Injection: In the constructor injection, the injector supplies the service (dependency) through the client class constructor.
2. Property Injection: In the property injection (aka the Setter Injection), the injector supplies the dependency through a public property of the client class.
3. Method Injection: In this type of injection, the client class implements an interface which declares the method(s) to supply the dependency and the injector uses this interface to supply the dependency to the client class.

DI using the constructor injection:

when we provide the dependency through the constructor, this is called a constructor injection.

```
public class CustomerBusinessLogic
{
    ICustomerDataAccess _dataAccess;

    public CustomerBusinessLogic(ICustomerDataAccess custDataAccess)
    {
        _dataAccess = custDataAccess;
    }

    public CustomerBusinessLogic()
    {
        _dataAccess = new CustomerDataAccess();
    }

    public string ProcessCustomerData(int id)
    {
        return _dataAccess.GetCustomerName(id);
    }
}

public interface ICustomerDataAccess
{
    string GetCustomerName(int id);
}

public class CustomerDataAccess: ICustomerDataAccess
{
    public CustomerDataAccess()
    {
    }

    public string GetCustomerName(int id)
    {
        //get the customer name from the db in real application
        return "Dummy Customer Name";
    }
}
```

In the above example, CustomerBusinessLogic includes the constructor with one parameter of type ICustomerDataAccess. Now, the calling class must inject an object of ICustomerDataAccess.

```
public class CustomerService
{
    CustomerBusinessLogic _customerBL;

    public CustomerService()
    {
        _customerBL = new CustomerBusinessLogic(new CustomerDataAccess());
    }

    public string GetCustomerName(int id) {
        return _customerBL.ProcessCustomerData(id);
    }
}
```

In the above example, the CustomerService class creates and injects the CustomerDataAccess object into the CustomerBusinessLogic class. Thus, the CustomerBusinessLogic class doesn't need to create an object of CustomerDataAccess using the new keyword or using factory class. The calling class (CustomerService) creates and sets the appropriate DataAccess class to the CustomerBusinessLogic class. In this way, the CustomerBusinessLogic and CustomerDataAccess classes become "more" loosely coupled classes.

DI using Property injection:

The dependency is provided through a public property.

```
public class CustomerBusinessLogic
{
    public CustomerBusinessLogic()
    {
    }

    public string GetCustomerName(int id)
    {
        return DataAccess.GetCustomerName(id);
    }

    public ICustomerDataAccess DataAccess { get; set; }
}

public class CustomerService
{
    CustomerBusinessLogic _customerBL;

    public CustomerService()
    {
        _customerBL = new CustomerBusinessLogic();
        _customerBL.DataAccess = new CustomerDataAccess();
    }

    public string GetCustomerName(int id) {
        return _customerBL.GetCustomerName(id);
    }
}
```

The CustomerBusinessLogic class includes the public property named DataAccess, where you can set an instance of a class that implements ICustomerDataAccess. So, CustomerService class creates and sets CustomerDataAccess class using this public property.

DI using the method injection:

Dependencies are provided through methods. This method can be a class method or an interface method.

```
interface IDataAccessDependency
{
    void SetDependency(ICustomerDataAccess customerDataAccess);
}

public class CustomerBusinessLogic : IDataAccessDependency
{
    ICustomerDataAccess _dataAccess;

    public CustomerBusinessLogic()
    {
    }

    public string GetCustomerName(int id)
    {
        return _dataAccess.GetCustomerName(id);
    }

    public void SetDependency(ICustomerDataAccess customerDataAccess)
    {
        _dataAccess = customerDataAccess;
    }
}

public class CustomerService
{
    CustomerBusinessLogic _customerBL;

    public CustomerService()
    {
        _customerBL = new CustomerBusinessLogic();
        ((IDataAccessDependency)_customerBL).SetDependency(new CustomerDataAccess());
    }

    public string GetCustomerName(int id) {
        return _customerBL.GetCustomerName(id);
    }
}
```

In the above example, the CustomerBusinessLogic class implements the IDataAccessDependency interface, which includes the SetDependency() method. So, the injector class CustomerService will now use this method to inject the dependent class (CustomerDataAccess) to the client class.
Another example: IoC vs. DI - DZone; Spring IoC and DI: A Practical Guide | Medium

SPRING CONTAINER:

What is Spring Container?

Spring containers are simple java classes and interfaces.
BeanFactory is the fundamental spring container and the basis on which spring's DI is based. Spring's modules are built on top of the core container. Spring's core container provides the fundamental functionality of the Spring Framework. The container defines how beans are created, configured, and managed, more of the nuts and bolts of Spring.

Types of Spring container:

There are many spring containers among which two are very important -

1. Bean factories defined by the **org.springframework.beans.factory.BeanFactory** interface are the simplest of containers, providing basic support for dependency injection (DI). This BeanFactory is the super interface of all the spring containers
2. Application contexts defined by the **org.springframework.context.ApplicationContext** interface build on the notion of a bean factory by providing application framework services

These **BeanFactory** and **ApplicationContext** are java interfaces, but in spring world these are called containers.

BeanFactory:

- BeanFactory is an implementation of the Factory design pattern. As the name says it is a factory of Beans whose responsibility is to create and dispense beans.
- BeanFactory creates and dispenses many types of beans
- Including creation and dispensing, bean factory knows about many objects within an application, it is able to create associations between collaborating objects as they are instantiated. Hence we need not to worry about configuring the bean itself and the bean's

client. As a result, when a bean factory hands out objects, those objects are fully configured, are aware of their collaborating objects, and are ready to use

Application Context:

ApplicationContext interface extends the BeanFactory interface. To take full advantage of Spring framework most of the time we load beans using more advanced spring container ApplicationContext.

To initialize the BeanFactory container we have to create the object of any of the classes which implements BeanFactory interface - ApplicationContext interface extends the BeanFactory interface. Hence when we initialize an ApplicationContext container, BeanFactory Container will also initialize. Hence to initialize a BeanFactory container we can also create an object of class that implements ApplicationContext.

Among the many implementations of ApplicationContext are three that are commonly used -

1. **ClassPathXmlApplicationContext:** Loads a context definition from an XML file located in the classpath, treating context definition files as classpath resources

ClassPathXmlApplicationContext

```
package com.java4coding;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Test {
        public static void main(String[] args) {
                BeanFactory factory = new ClassPathXmlApplicationContext("studentconfig.xml");
                Student student = (Student) factory.getBean("studentbean");
                student.displayInfo();
        }
}
```

2. **FileSystemXmlApplicationContext:** Loads a context definition from an XML file in the file system.(will look for xml file in a specific location within the file system)

FileSystemXmlApplicationContext

```
package com.java4coding;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.context.support.FileSystemXmlApplicationContext;

public class Test {
        public static void main(String[] args) {
                BeanFactory factory = new FileSystemXmlApplicationContext("D:\\Application\\studentconfig.xml");
                Student student = (Student) factory.getBean("studentbean");
                student.displayInfo();
        }
}
```

3. **XmlWebApplicationContext:** Loads context definitions from an XML file contained within a web application.(will look for xml file anywhere in the classpath including JAR files).

Why is ApplicationContext preferred over BeanFactory in most applications?

Some of the additional functionality it provides over BeanFactory -

1. Ability to work with multiple spring configuration files. If a spring project is involved in multiple modules of spring then we need to deal with multiple spring configuration files, in that situation ApplicationContext is useful.
2. Pre instantiation of spring beans.
3. Support to read bean property values from properties file.

4. Application contexts provide a means for resolving text messages, including support for internationalization (I18N) of those messages.
5. Application contexts provide a generic way to load file resources, such as images.
6. Application contexts can publish events to beans that are registered as listeners.
7. We can stop the ApplicationContext server by calling close() method , but the BeanFactory container cannot be stopped explicitly.

In summary, BeanFactory is the most basic version of IOC containers, while ApplicationContext extends the features of BeanFactory. ApplicationContext is considered a heavy IOC container because its eager-loading strategy loads all the beans at startup. This is because ApplicationContext provides more advanced features such as integration with Spring AOP, message resource handling, and event publication. It provides advanced features that are geared towards enterprise applications, while BeanFactory comes with only basic features
Because of the additional functionality it provides, an ApplicationContext is preferred over a BeanFactory in nearly all applications.

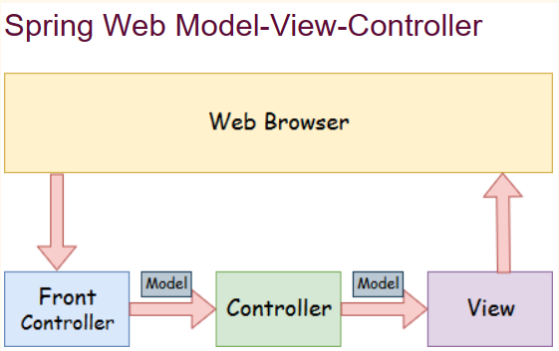
Feature	**BeanFactory**	**ApplicationContext**
Lazy Loading	Loads beans on-demand	Loads all beans at startup
Eager Loading	Not supported	Loads all beans instantly
AOP	Not supported	Supports AOP
Message Resource Handling	Not supported	Supports message resource handling
Event Publication	Not supported	Supports event publication

BEAN SCOPES:
Bean Scopes refers to the life cycle of Bean that means when the object of Bean will be instantiated, how long does that object live, and how many objects will be created for that bean throughout. Basically, it controls the instance creation of the bean and it is managed by the spring container.
The spring framework provides five scopes for a bean. We can use three of them only in the context of web-aware **Spring ApplicationContext** and the rest of the two is available for both **IoC container and Spring-MVC container** -

1. **Singleton:** Only one instance will be created for a single bean definition per Spring IoC container and the same object will be shared for each request made for that bean.
2. **Prototype:** A new instance will be created for a single bean definition every time a request is made for that bean.
3. **Request:** A new instance will be created for a single bean definition every time an HTTP request is made for that bean. But Only valid in the context of a web-aware Spring ApplicationContext.
4. **Session:** Scopes a single bean definition to the lifecycle of an HTTP Session. But Only valid in the context of a web-aware Spring ApplicationContext.
5. **Global-Session:** Scopes a single bean definition to the lifecycle of a global HTTP Session. It is also only valid in the context of a web-aware Spring ApplicationContext.

SPRING MVC: Learn Spring MVC Tutorial (geeksforgeeks.org)
Spring MVC is a Java-based framework that is mostly used for developing web applications. It follows the MVC (Model-View-Controller) Design Pattern. This design pattern specifies that an application consists of a data model, presentation information, and control information.

- **Model –** A model contains the application's data. A data set might be a single object or a group of things.
- **Controller –** A controller houses an application's business logic. The @Controller annotation is used here to identify the class as the controller



- **View –** A view is a representation of the delivered information in a certain format. In most cases, JSP+JSTL is utilized to construct a view page. Spring does, however, support additional view technologies such as Apache Velocity, Thymeleaf, and FreeMarker.
- **Front Controller –** The DispatcherServlet class serves as the front controller in Spring Web MVC. It is in charge of managing the flow of the Spring MVC application.

Spring MVC vs Spring Boot:

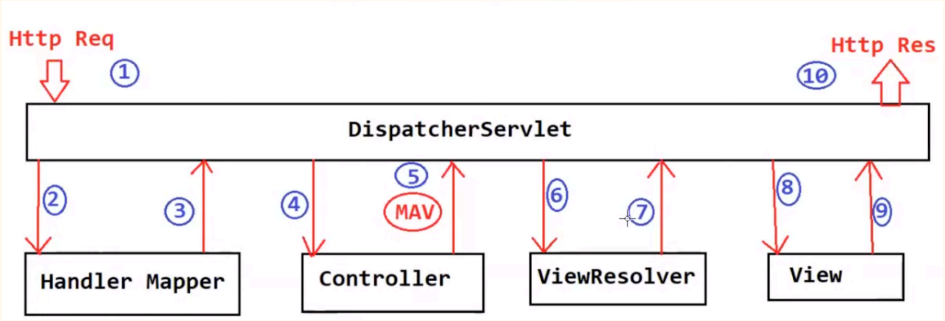
S.No.	SPRING MVC	SPRING BOOT
1.	Spring MVC is a Model View, and Controller based web framework widely used to develop web applications.	Spring Boot is built on top of the conventional spring framework, widely used to develop REST APIs.
2.	If we are using Spring MVC, we need to build the configuration manually.	If we are using Spring Boot, there is no need to build the configuration manually.
3.	In the Spring MVC, a deployment descriptor is required.	In the Spring Boot, there is no need for a deployment descriptor.
4.	Spring MVC specifies each dependency separately.	It wraps the dependencies together in a single unit.
5.	Spring MVC framework consists of four components : Model, View, Controller, and Front Controller.	There are four main layers in Spring Boot: Presentation Layer, Data Access Layer, Service Layer, and Integration Layer.
6.	It takes more time in development.	It reduces development time and increases productivity.
7.	Spring MVC does not provide powerful batch processing.	Powerful batch processing is provided by Spring Boot.
8.	Ready to use features are provided by it for building web applications.	Default configurations are provided by it for building a Spring powered framework.

Advantages of Spring MVC:

1. Separate roles - The Spring MVC separates each role, where the model object, controller, command object, view resolver, DispatcherServlet, validator, etc. can be fulfilled by a specialized object.
2. Light-weight - It uses light-weight servlet container to develop and deploy your application.

3. Powerful Configuration - It provides a robust configuration for both framework and application classes that includes easy referencing across contexts, such as from web controllers to business objects and validators.
4. Rapid development - The Spring MVC facilitates fast and parallel development.
5. Reusable business code - Instead of creating new objects, it allows us to use the existing business objects.
6. Easy to test - In Spring, generally we create JavaBeans classes that enable you to inject test data using the setter methods.
7. Flexible Mapping - It provides the specific annotations that easily redirect the page.

How does Spring MVC work internally?



This diagram represents how an HTTP request is processed from start to end in the Spring MVC framework step by step.
Step 1 - When the client (browser) sends an HTTP request to a specific URL. The DispatcherServlet of Spring MVC receives the request.
Step 2 - DispatcherServlet consults HandlerMapper to identify which controller is responsible to handle the HTTP request.
Step 3 - HandlerMapper selects the controller which is mapped to the incoming request URL and returns the (selected Handler) and Controller details to DispatcherServlet.
Step 4 - Now DispatcherServlet knows which controller is responsible to process the request so DispatcherServlet will forward that request to the corresponding controller to process the request.
Step 5 - Now the Controller processes the request, validates the request, and creates a model with data. Finally, the Controller returns the logical name of view and model to the DispatcherServlet.
Step 6 - DispatcherServlet consults ViewResolver to resolve a logical view with the physical view that exists in the application.
Step 7 - ViewResolver responsible to map logical view with actual view and return the actual view details back to the DispatcherServlet.
Step 8 - Now DispatcherServlet sends the view and model to the View component.
Step 9 - The View component merges view and model and forms a plain HTML output. Finally, the View component sends HTML output back to the DispatcherServlet.
Step 10 - The DispatcherServlet finally sends HTML output as a response back to the browser for rendering.

Prefix, Suffix:

In Spring MVC, prefix and suffix are used to configure the view resolver. The view resolver is responsible for mapping view names to actual views. The prefix is a string that is prepended to the view name, and the suffix is a string that is appended to the view name. The resulting string is the path to the view file. For example, if the prefix is set to `/WEB-INF/views/` and the suffix is set to `.jsp`, then the view resolver will look for the JSP file in the `/WEB-INF/views/` directory with the name of the view appended to it.
Here is an example configuration in application.properties file:

```
spring.mvc.view.prefix=/WEB-INF/views/
spring.mvc.view.suffix=.jsp
```

This configuration tells the view resolver to look for JSP files in the `/WEB-INF/views/` directory with the `.jsp` extension.

@ModelAttribute: The @ModelAttribute annotation in Spring MVC serves multiple roles, providing a robust solution for data mapping between a client's request and the server's model object. This annotation can be applied to method parameters, method return types, or even methods themselves. It refers to the property of the Model object.

Life Cycle of @ModelAttribute in the Spring Context:
When a method parameter is annotated with @ModelAttribute, Spring goes through the following steps:

1. **Lookup:** Spring first tries to find an existing model attribute with the same name as the parameter's name.
2. **Instantiation:** If no existing model attribute is found, Spring will instantiate a new object of the corresponding class.
3. **Population:** Spring then takes each form field and matches it against the properties in the model object, populating them using their corresponding setter methods.
4. **Addition to Model:** Finally, the populated object is added to the model, making it available for rendering in the view layer.

SPRING ORM:
19. Object Relational Mapping (ORM) Data Access (spring.io)
Spring ORM (Object-Relational Mapping) is a module within the larger Spring Framework that provides integration with various ORM frameworks, making it easier to work with relational databases in a Java environment. ORM is a programming technique that allows developers to interact with databases using an object-oriented programming paradigm rather than directly using SQL queries.
Spring ORM simplifies database programming by abstracting away many of the complexities associated with database interaction. It supports integration with popular ORM frameworks such as Hibernate, JPA (Java Persistence API), MyBatis, and others. The primary goal is to provide a consistent and efficient way to handle database operations while promoting loose coupling and ease of testing.
ORM (Object Relational Mapping) covers many persistence technologies. They are as follows:

1. **JPA(Java Persistence API):** It is mainly used to persist data between Java objects and relational databases. It acts as a bridge between object-oriented domain models and relational database systems.
2. **JDO(Java Data Objects):** It is one of the standard ways to access persistent data in databases, by using plain old Java objects (POJO) to represent persistent data.
3. **Hibernate:** It is a Java framework that simplifies the development of Java applications to interact with the database.
4. **Oracle Toplink, and iBATIS:** Oracle TopLink is a mapping and persistence framework for Java development.

Here are some key concepts and components related to Spring ORM:

1. **DataSource:** The DataSource interface in Spring provides a connection pool to manage database connections. It abstracts the details of creating and managing database connections.
2. **SessionFactory:** In the context of Hibernate (a popular ORM framework), the SessionFactory is a key component. It is responsible for creating and managing Hibernate Session instances, which represent a unit of work with the database.
3. **EntityManagerFactory:** In the context of JPA, the EntityManagerFactory is analogous to the SessionFactory in Hibernate. It creates and manages JPA EntityManager instances.
4. **Transaction Management:** Spring provides a consistent and declarative way to manage transactions across different ORM frameworks. It supports both programmatic and declarative transaction management.

5. **DAO (Data Access Object):** The DAO pattern is commonly used in Spring ORM to abstract and encapsulate the interaction with the underlying database. DAOs provide methods to perform CRUD (Create, Read, Update, Delete) operations on domain objects.
6. **Spring Data JPA:** Spring Data JPA is a part of the larger Spring Data project, which simplifies data access in Spring applications. It provides additional abstractions and features for working with JPA-based repositories.
7. **XML Configuration or Annotation-based Configuration:** Spring ORM can be configured using XML-based configuration or annotation-based configuration. XML configuration is traditional, while annotation-based configuration is more concise and can be used with Java-based configuration.

Benefits of using the Spring Framework to create your ORM DAOs include:

1. **Easier testing:** Spring's IoC approach makes it easy to swap the implementations and configuration locations of Hibernate SessionFactory instances, JDBC DataSource instances, transaction managers, and mapped object implementations (if needed). This in turn makes it much easier to test each piece of persistence-related code in isolation.
2. **Common data access exceptions:** Spring can wrap exceptions from your ORM tool, converting them from proprietary (potentially checked) exceptions to a common runtime DataAccessException hierarchy. This feature allows you to handle most persistence exceptions, which are non-recoverable, only in the appropriate layers, without annoying boilerplate catches, throws, and exception declarations. You can still trap and handle exceptions as necessary. Remember that JDBC exceptions (including DB-specific dialects) are also converted to the same hierarchy, meaning that you can perform some operations with JDBC within a consistent programming model.
3. **General resource management:** Spring application contexts can handle the location and configuration of Hibernate SessionFactory instances, JPA EntityManagerFactory instances, JDBC DataSource instances, and other related resources. This makes these values easy to manage and change. Spring offers efficient, easy, and safe handling of persistence resources. For example, related code that uses Hibernate generally needs to use the same Hibernate Session to ensure efficiency and proper transaction handling. Spring makes it easy to create and bind a Session to the current thread transparently, by exposing a current Session through the Hibernate SessionFactory. Thus Spring solves many chronic problems of typical Hibernate usage, for any local or JTA transaction environment.
4. **Integrated transaction management:** You can wrap your ORM code with a declarative, aspect-oriented programming (AOP) style method interceptor either through the @Transactional annotation or by explicitly configuring the transaction AOP advice in an XML configuration file. In both cases, transaction semantics and exception handling (rollback, and so on) are handled for you. You can also swap various transaction managers, without affecting your ORM-related code. For example, you can swap between local transactions and JTA, with the same full services (such as declarative transactions) available in both scenarios. Additionally, JDBC-related code can fully integrate transactionally with the code you use to do ORM. This is useful for data access that is not suitable for ORM, such as batch processing and BLOB streaming, which still need to share common transactions with ORM operations.

HIBERNATE:

Hibernate ORM (Object Relational Mapping) is a popular Java-based framework used for mapping an object-oriented domain model to a relational database. It provides an efficient way of storing and retrieving data from a database by mapping database tables to Java classes and vice versa.

Features of Hibernate ORM:

1. **Object-Relational Mapping:** Hibernate ORM provides support for mapping Java classes to database tables and vice versa. This allows developers to work with objects rather than low-level SQL statements.
2. **Lazy Loading:** Hibernate ORM provides support for lazy loading, which is a technique for loading objects on demand rather than loading them all at once. This can help to improve performance and reduce memory usage.
3. **Caching:** Hibernate ORM provides support for caching, which is a technique for storing frequently accessed data in memory. This can help to improve performance and reduce the number of database queries.
4. **Transactions:** Hibernate ORM provides support for transactions, which are a mechanism for ensuring data consistency and integrity. Transactions can be used to group a set of database operations into a single atomic unit of work.
5. **Query Language:** Hibernate ORM provides support for HQL (Hibernate Query Language), which is a powerful and flexible query language for querying objects. HQL is similar to SQL but operates on objects rather than tables.

Advantages:

1. **Simplifies Database Operations:** Hibernate ORM simplifies database operations by providing an easy-to-use API for storing and retrieving data from the database. This reduces the amount of boilerplate code required to interact with the database.
2. **Improves Performance:** Hibernate ORM provides support for caching, which improves application performance by reducing the number of database queries. It caches frequently accessed data in memory, which reduces the time taken to access the data from the database.
3. **Platform Independence:** Hibernate ORM is platform-independent, which means it can be used with any relational database. This provides flexibility and makes it easy to switch databases without changing the code.
4. **Open-Source:** Hibernate ORM is an open-source framework, which means it is freely available and can be used by anyone. This reduces the cost of development and makes it easy to get started with Hibernate ORM.

SPRING DATA JPA:

What is Spring Data JPA? - GeeksforGeeks

JPA is a Java specification(Jakarta Persistence API) and it manages relational data in Java applications. To access and persist data between Java objects(Plain Old Java object)/ class and relational databases, we can use JPA. It has the runtime EntityManager API and it is responsible for processing queries and transactions on the Java objects against the database. The main highlight is it uses JPQL (Java Persistence Query Language) which is platform-independent.

Here are some key features and concepts related to Spring Data JPA:

1. **Repository Interface:** Spring Data JPA introduces the concept of repository interfaces. A repository interface is an interface that extends the JpaRepository interface provided by Spring Data JPA. It allows you to define methods for common CRUD operations and queries without the need to write the actual implementation.
2. **Query Methods:** Spring Data JPA enables the creation of query methods by simply declaring their method signature. The framework automatically generates the required query based on the method name. For example, a method named findByLastName(String lastName) would automatically generate a query to find entities with a specified last name.
3. **Derived Queries:** In addition to query methods, Spring Data JPA supports derived queries. Derived queries are automatically generated based on method names, and they can include conditions, sorting, and more. For example, findByLastNameAndFirstName(String lastName, String firstName).

4. **Pagination & Sorting:** Spring Data JPA provides built-in support for pagination and sorting. By extending the PagingAndSortingRepository interface, you gain access to methods for paginated queries and sorting without the need for additional boilerplate code.
5. **Custom Queries:** For more complex queries, Spring Data JPA allows you to define custom queries using the @Query annotation. You can write JPQL (Java Persistence Query Language) or native SQL queries.
6. **Entity Manager:** Spring Data JPA utilizes the EntityManager for managing JPA entities. The EntityManager is injected into repository implementations, allowing them to interact with the underlying JPA provider (e.g. Hibernate).

Advantages of using JPA:

1. No need to write DDL/DML queries, instead we can map by using XML/annotations.
2. JPQL is used and since it is platform-independent, we no need to depend on any native SQL table. Complex expressions and filtering expressions are all handled via JPQL only.
3. Entity can be partially stored in one database like MySQL and the rest can be in Graph database Management System.
4. Dynamic generation of queries is possible.
5. Integration with Spring framework is easier with a custom namespace.

Units comprised in JPA are available under javax persistence package:

Persistence	It has static methods to obtain an EntityManagerFactory instance
EntityManagerFactory	Factory class for EntityManager and responsible for managing multiple instances of EntityManager
EntityManager	It is an interface that works for the Query instance
Entity	They are persistent objects and stored as records in the database
Persistence Unit	Set of all entity classes
EntityTransaction	It has a one-to-one relationship with EntityManager.
Query	To get relation objects that meet the criteria.

JPA vs Hibernate:

JPA	Hibernate
JPA : It is a Java specification for mapping relational data in Java application. It is not a framework	Hibernate is an ORM framework and in that way data persistence is possible.

In JPA, no implementation classes are provided.	In Hibernate, implementation classes are provided.
Main advantage is it uses JPQL (Java Persistence Query Language) and it is platform-independent query language.	Here it is using HQL (Hibernate Query Language).
It is available under javax.persistence package.	It is available under the org.hibernate package.
In Hibernate, EclipseLink, etc. we can see its implementation.	Hibernate is the provider of JPA.
Persistence of data is handled by EntityManager.	Persistence of data is handled by Session.

In the context of Spring Data JPA and Hibernate, there are three types of queries that can be used to interact with a relational database: Native Queries, JPQL (Java Persistence Query Language), and HQL (Hibernate Query Language). Each type of query has its own syntax and use cases.

1. **Native Queries:**
    a. Syntax: Native queries are written in the native SQL dialect of the underlying database.
    b. Usage: Native queries are useful when you need to execute database-specific SQL statements that might not be easily expressible in JPQL or HQL.
    c. Use:
        i. When you need to execute database-specific SQL statements.
        ii. When you want to take advantage of database-specific features.

```
@Query(value = "SELECT * FROM users WHERE user_name = :username", nativeQuery = true)
User findByUsernameNativeQuery(@Param("username") String username);
```

2. **JPQL (Java Persistence Query Language):**
    a. Syntax: JPQL is a query language that is similar to SQL but operates on Java objects rather than database tables.
    b. Usage: JPQL is database-agnostic, making it suitable for applications that need to support multiple database systems. It allows queries to be expressed in terms of Java objects rather than database tables.
    c. Use:
        i. When you want to write queries in a database-agnostic way.
        ii. When working with multiple database systems, you want your queries to be portable.

```
@Query("SELECT u FROM User u WHERE u.username = :username")
User findByUsernameJPQL(@Param("username") String username);
```

3. **HQL (Hibernate Query Language):**

a. Syntax: HQL is a query language specific to Hibernate, and it is similar to JPQL. However, it may have some Hibernate-specific features not present in JPQL.
b. Usage: HQL is useful when you want to take advantage of Hibernate-specific features or optimizations.
c. Use:
    i. When you are using Hibernate as the JPA provider and want to leverage Hibernate-specific features.
    ii. When you prefer Hibernate's specific extensions or optimizations.

```
@Query("FROM User u WHERE u.username = :username")
User findByUsernameHQL(@Param("username") String username);
```

Use of @param in JPQL:

In Spring Data JPA, the @Param annotation is used to bind method parameters to named parameters in JPQL or native queries defined in repository interfaces. It helps in creating dynamic queries by allowing you to reference method parameters by name in the query.

```
@Query("SELECT u FROM User u WHERE u.username = :name")
List<User> findByUsername(@Param("name") String username);
```

SPRING REST:

REST is an acronym for REpresentational State Transfer and an architectural style for distributed hypermedia systems. REST is not a protocol or a standard, it is an architectural style.
REST API is a way of accessing web services in a simple and flexible way without having any processing.
REST technology is generally preferred to the more robust Simple Object Access Protocol (SOAP) technology because REST uses less bandwidth, simple and flexible making it more suitable for internet usage. It's used to fetch or give some information from a web service. All communication done via REST API uses only HTTP requests.

Working:

A request is sent from client to server in the form of a web URL as HTTP GET or POST or PUT or DELETE request. After that, a response comes back from the server in the form of a resource which can be anything like HTML, XML, Image, or JSON. But now JSON is the most popular format being used in Web Services.

In HTTP there are five methods that are commonly used in a REST-based Architecture i.e., POST, GET, PUT, PATCH, and DELETE. These correspond to create, read, update, and delete (or CRUD) operations respectively. There are other methods which are less frequently used like OPTIONS and HEAD.

- **GET:** The HTTP GET method is used to read (or retrieve) a representation of a resource. In the safe path, GET returns a representation in XML or JSON and an HTTP response code of 200 (OK). In an error case, it most often returns a 404 (NOT FOUND) or 400 (BAD REQUEST).
- **POST:** The POST verb is most often utilized to create new resources. In particular, it's used to create subordinate resources. That is, subordinate to some other (e.g. parent) resource. On successful creation, return HTTP status 201, returning a Location header with a link to the newly-created resource with the 201 HTTP status.

- **PUT:** It is used for updating the capabilities. However, PUT can also be used to create a resource in the case where the resource ID is chosen by the client instead of by the server. In other words, if the PUT is to a URI that contains the value of a non-existent resource ID. On successful update, return 200 (or 204 if not returning any content in the body) from a PUT. If using PUT for create, return HTTP status 201 on successful creation. PUT is not safe operation but it's idempotent.
- **PATCH:** It is used to modify capabilities. The PATCH request only needs to contain the changes to the resource, not the complete resource. This resembles PUT, but the body contains a set of instructions describing how a resource currently residing on the server should be modified to produce a new version. This means that the PATCH body should not just be a modified part of the resource, but in some kind of patch language like JSON Patch or XML Patch. PATCH is neither safe nor idempotent.
- **DELETE:** It is used to delete a resource identified by a URI. On successful deletion, return HTTP status 200 (OK) along with a response body.

Idempotence: An idempotent HTTP method is a HTTP method that can be called many times without different outcomes. It would not matter if the method is called only once, or ten times over. The result should be the same. Again, this only applies to the result, not the resource itself.

1. a = 4

// It is Idempotence, as final value(a = 4) would not change after executing it multiple times.

2. a + +

// It is not Idempotence because the final value will depend upon the number of times the statement is executed.

Here are the key components and concepts related to building RESTful web services with Spring:

1. **Spring mvc & REST Controllers:** Spring MVC is a module within the Spring Framework that facilitates building web applications. RESTful web services can be implemented using Spring MVC controllers annotated with @RestController. The @RestController annotation is a specialized version of the @Controller annotation for RESTful services.
2. **Request Mapping:** Use the @RequestMapping annotation to map URLs to methods in your controller. It helps define the base URI for the controller and further refines the URL mapping for individual methods.
3. **HTTP Methods & Annotations:** Use annotations like @GetMapping, @PostMapping, @PutMapping, @DeleteMapping, etc., to map specific HTTP methods to controller methods. These annotations simplify the creation of RESTful endpoints.
4. **PathVariable & RequestParam:** Use @PathVariable to extract values from the URI path, and use @RequestParam to extract values from query parameters.
5. **RequestBody & ResponseBody:** Use @RequestBody to bind the method parameter to the body of the HTTP request. Use @ResponseBody to annotate the return value to be serialized directly to the response body.
6. **ResponseEntity:** Use ResponseEntity to have more control over the HTTP response. It allows you to set status codes, headers, and the response body.
7. **Exception Handling:** Implement exception handling to provide meaningful error responses. Use @ExceptionHandler to handle exceptions within the controller.
8. **Content Negotiation:** Spring supports content negotiation to handle different types of responses (JSON, XML, etc.). You can use produces and consumes attributes in the @RequestMapping annotation.
9. **RESTful URI Design:** Follow best practices for designing RESTful URIs. Use nouns to represent resources and HTTP methods to represent actions.

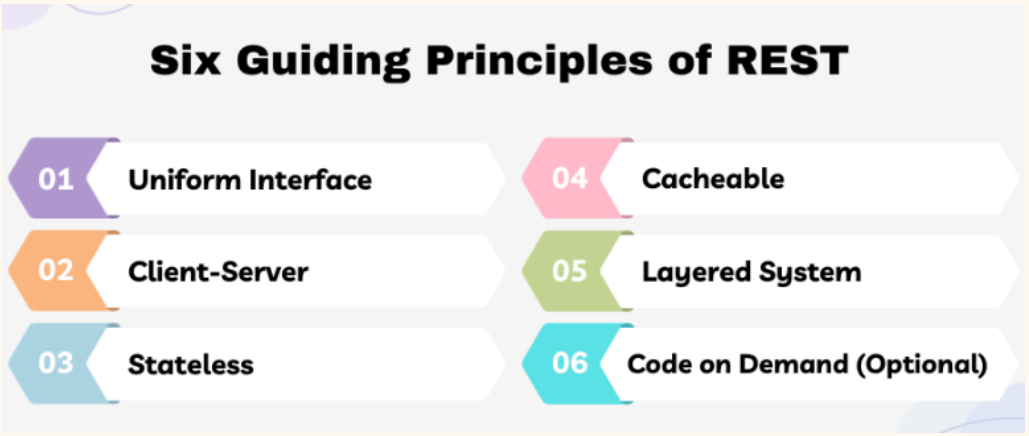
10. **Spring Boot Support:** If you're using Spring Boot, building RESTful web services is even more straightforward. Spring Boot provides auto-configuration, reducing the need for boilerplate code.

```
@RestController
@RequestMapping("/api/users")
public class UserController {
@GetMapping(value = "/{id}", produces = MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity<User> getUserById(@PathVariable Long id, @RequestParam String type) {
   // Logic to fetch user by ID and type
   // Return ResponseEntity with user data        }
  @PostMapping, consumes= MediaType.APPLICATION_JSON_VALUE, produces =
MediaType.APPLICATION_JSON_VALUE)
   public ResponseEntity<User> createUser(@RequestBody User user) {
      // Logic to create a new user
      // Return ResponseEntity with the created user data
   }
}
```

Six Guiding Principles of REST:

REST is based on some constraints and principles that promote simplicity, scalability, and statelessness in the design.

The six guiding principles are:



1. **Uniform Interface:** Multiple architectural constraints help in obtaining a uniform interface and guiding the behavior of components.
The following four constraints can achieve a uniform REST interface:

1.1. **Identification of resources –** The interface must uniquely identify each resource involved in the interaction between the client and the server.

1.2. **Manipulation of resources through representations –** The resources should have uniform representations in the server response. API consumers should use these representations to modify the resource state in the server.

1.3. **Self-descriptive messages –** Each resource representation should carry enough information to describe how to process the message. It should also provide information of the additional actions that the client can perform on the resource.

1.4. **Hypermedia as the engine of application state –** The client should have only the initial URI of the application. The client application should dynamically drive all other resources and interactions with the use of hyperlinks.

In simpler words, REST defines a consistent and uniform interface for interactions between clients and servers. For example, the HTTP-based REST APIs make use of the standard HTTP methods (GET, POST, PUT, DELETE, etc.) and the URIs (Uniform Resource Identifiers) to identify resources.

2. **Client-Server:** This constraint essentially means that client applications and server applications MUST be able to evolve separately without any dependency on each other. A client should know only resource URIs, and that's all.

The client-server design pattern enforces the separation of concerns, which helps the client and the server components evolve independently. By separating the user interface concerns (client) from the data storage concerns (server), we improve the portability of the user interface across multiple platforms and improve scalability by simplifying the server components. While the client and the server evolve, we have to make sure that the interface/contract between the client and the server does not break.

3. **Stateless:** Stateless recommends making all the client-server interactions stateless. What it means is that the server will not store anything about the latest HTTP request the client made. It will treat every request as new. No session, no history. If the client application needs to be a stateful application for the end-user, where the user logs in once and does other authorized operations after that, then each request from the client should contain all the information necessary to service the request – including authentication and authorization details.
4. **Cacheable:** The cacheable constraint requires that a response should implicitly or explicitly label itself as cacheable or non-cacheable. If the response is cacheable, the client application gets the right to reuse the response data later for equivalent requests and a specified period. Caching brings performance improvement for the client side and better scope for scalability for a server because the load has been reduced. Caching can be implemented on the server or client side.
5. **Layered System:** The layered system style allows an architecture to be composed of hierarchical layers by constraining component behavior. In a layered system, each component cannot see beyond the immediate layer they are interacting with.
6. **Code of Demand:** REST also allows client functionality to extend by downloading and executing code in the form of applets or scripts. The downloaded code simplifies clients by reducing the number of features required to be pre-implemented. Servers can provide part of features delivered to the client in the form of code, and the client only needs to execute the code.

json & jackson:

JSON (JavaScript Object Notation) is a lightweight data interchange format that is easy for humans to read and write, and easy for machines to parse and generate. JSON data is represented as key-value pairs, similar to objects in many programming languages.

```
{
  "name": "John Doe",
  "age": 30,
  "city": "New York",
  "isStudent": false,
  "grades": [90, 85, 92],
  "address": {
    "street": "123 Main St",
    "zipCode": "10001"
  }
}
```

- **Syntax:** JSON consists of key-value pairs, where keys are strings and values can be strings, numbers, objects, arrays, booleans, or null.
- **Data Types:** JSON supports various data types, including strings, numbers, booleans, arrays, objects, and null.
- **Use Cases:** JSON is commonly used for data exchange between a server and a web client, configuration files, and as a data format for APIs.

Jackson is a widely-used and popular Java library for working with JSON. It provides a set of high-performance tools for processing JSON data in Java applications. Jackson can serialize Java objects to JSON and deserialize JSON back to Java objects. It is commonly used in Java applications, including web applications, for handling JSON data.

- **Object Mapping:** Jackson provides ObjectMapper, which is used for converting Java objects to JSON (serialization) and JSON to Java objects (deserialization)

```
ObjectMapper objectMapper = new ObjectMapper();
```

```
// Serialization: Java object to JSON
String json = objectMapper.writeValueAsString(myObject);
// Deserialization: JSON to Java object
MyObject myObject = objectMapper.readValue(json, MyObject.class);tion).
```

- **Annotations:** Jackson supports annotations for customizing the serialization and deserialization process. For example, you can use @JsonProperty to map a Java field to a specific JSON key.

```
public class MyObject {
   @JsonProperty("full_name")
   private String fullName;
}
```

- **Date Handling:** Jackson provides options for handling date formats during serialization and deserialization.

```
objectMapper.setDateFormat(new SimpleDateFormat("yyyy-MM-dd"));
```

- **Tree Model & Streaming Api:** Jackson supports both a Tree Model and a Streaming API. The Tree Model allows you to work with JSON as a tree structure, and the Streaming API allows you to process JSON data in a streaming fashion.

```
// Tree Model
JsonNode jsonNode = objectMapper.readTree(json);
String name = jsonNode.get("name").asText();

// Streaming API
JsonParser jsonParser = objectMapper.getFactory().createParser(json);
while (jsonParser.nextToken() != null) {
   // Process JSON tokens
}
```

- **Custom Serialization & Deserialization:** You can implement custom serializers and deserializers to handle specific cases or customize the serialization/deserialization process.

```
public class CustomDateSerializer extends JsonSerializer<Date> {
   @Override
   public void serialize(Date value, JsonGenerator gen, SerializerProvider serializers) throws
IOException {
      gen.writeString(new SimpleDateFormat("yyyy-MM-dd").format(value));
   }
}
```

Advantages of REST API:

REST API is a lightweight and flexible architecture that can be easily implemented on any platform or language.

REST API is stateless, which means that each request contains all the necessary information to complete the request. This allows for scalability and reduces the load on the server.

REST API is widely adopted and supported by most modern programming languages and frameworks.

REST API provides a simple and standardized way of accessing resources over the internet.

Disadvantages of REST API:

REST API may not be the best choice for complex business logic and workflows.

REST API can be difficult to implement in certain situations, especially when dealing with complex data structures.

REST API does not provide a built-in authentication or authorization mechanism, which means that developers must implement their own security measures.

RESTful API:
RESTful API (Representational State Transferful API) is often used to describe an API that not only follows the principles of REST but also embodies the idea of being "RESTful" in its design. A RESTful API typically adheres to the principles of REST in a comprehensive and consistent manner. It strives to provide a clean and intuitive design that aligns with the core concepts of REST.

- **Uses Standard HTTP Methods:** It leverages standard HTTP methods (GET, POST, PUT, DELETE) to perform CRUD (Create, Read, Update, Delete) operations on resources.
- **Resource Naming**: Resources are identified by URIs, and the URIs are designed to be hierarchical, meaningful, and consistent.
- **Statelessness:** It adheres to the stateless communication principle, meaning that each request contains all the information necessary to understand and process it.
- **Hypermedia as the Engine of Application State (HATEOAS):** HATEOAS is an additional constraint that suggests including hypermedia links in the API responses, allowing clients to navigate the API dynamically by following links.

Advantages of RESTful API:

1. RESTful API is a standardized implementation of the REST architecture, which makes it easier for developers to build and maintain web services.
2. RESTful API provides a uniform interface for accessing resources, which simplifies the development process.
3. RESTful API supports caching, which can improve performance by reducing the number of requests sent to the server.
4. RESTful API can be used with a variety of data formats, including XML and JSON.

Disadvantages of RESTful API:

1. RESTful API may not be suitable for complex business logic and workflows.
2. RESTful API can be difficult to implement in certain situations, especially when dealing with complex data structures.
3. RESTful API can be slower than other web service architectures, especially when dealing with large amounts of data.

Similarities between REST API and RESTful API:

1. Both REST API and RESTful API are based on the REST architecture.
2. Both REST API and RESTful API are stateless, which means that each request contains all the necessary information to complete the request.
3. Both REST API and RESTful API provide a simple and standardized way of accessing resources over the internet.

REST vs RESTful API:

Factors	REST API	RESTful API
Define	Develops APIs to enable client-server interaction.	Web application follows REST architecture, providing interoperability between different systems.
Working	Uses web services and is based on request and response.	Working is completely based on REST applications.

Nature	Highly adaptable and user-friendly	Too flexible
Protocol	Strong protocol and more secure, built-in architecture layers.	Has a transport protocol, is less secure as compared to REST.
Architecture	Has a cacheable, client-server, stateless, layer system with a uniform interface.	All features of REST architecture along with some additional unique features.
Format of Data	Format of data is based on HTTP.	Format of data is based on HTTP, text, and JSON.
Bandwidth	This consumes minimum bandwidth.	This consumes less bandwidth.
Cache	It represents cacheable and non-cacheable data and displaces the non-cacheable data when not required.	The client can access cacheable information anytime and anywhere.

SPRING AOP:

9. Aspect Oriented Programming with Spring

Spring AOP - YouTube

Aspect-Oriented Programming is a paradigm that complements Object-Oriented Programming (OOP). While OOP is concerned with organizing code into classes and objects, AOP focuses on cross-cutting concerns – functionalities that affect multiple parts of an application. Cross-cutting concerns include logging, security, transactions, and more. The key unit of modularity in OOP is the class, whereas in AOP the unit of modularity is the aspect.

Why do we need AOP?

Aspect-Oriented Programming (AOP) addresses the need for modularizing cross-cutting concerns in software development. Cross-cutting concerns are aspects of an application that affect multiple modules and typically involve functionalities such as logging, security, transaction management, error handling, and performance monitoring. These concerns often result in code duplication, scattered code, and increased maintenance challenges.

Here are some key reasons why AOP is beneficial and addresses the needs of software development:

1. **Separation of Concerns:** AOP allows developers to separate cross-cutting concerns from the main business logic. This separation enhances the modularization of code, making it easier to understand, maintain, and evolve. Each module can focus on its specific functionality without being cluttered with unrelated concerns.
2. **Code Reusability:** By encapsulating cross-cutting concerns in aspects, AOP promotes code reusability. The same aspect can be applied to multiple parts of the codebase, reducing redundancy and promoting a more consistent and maintainable code structure.
3. **Improved Modularity:** AOP improves modularity by providing a way to encapsulate and manage cross-cutting concerns independently of the main business logic. This separation

makes it easier to add, remove, or modify aspects without affecting the core functionality of the application.

4. **Easier Maintenance:** With AOP, modifications to cross-cutting concerns can be made in a single place, namely the aspect, rather than requiring changes across multiple modules. This reduces the likelihood of introducing errors during maintenance and upgrades.
5. **Enhanced Readability:** By removing cross-cutting concerns from the main codebase, the readability of the core business logic is improved. Developers can focus on the essential aspects of each module without being distracted by unrelated details.
6. **Aspect Reusability Across Projects:** Aspects can be defined independently of specific application logic, making them potentially reusable across different projects. This can be particularly beneficial for standardizing and enforcing certain behaviors or policies across an organization's software projects.
7. **Dynamic Cross-Cutting Concerns:** AOP allows for dynamic weaving of aspects at runtime, providing flexibility in applying cross-cutting concerns based on changing requirements. This dynamic nature is particularly valuable in scenarios where different aspects need to be applied under different conditions.
8. **Maintenance of Non-Functional Requirements:** Cross-cutting concerns often correspond to non-functional requirements such as logging, security, and performance. AOP provides a clean and centralized way to address these non-functional requirements, improving the overall maintainability of the application.
9. **AspectJ Expressiveness:** AOP frameworks like AspectJ provide powerful and expressive features for defining pointcuts and expressing complex join points. This flexibility allows developers to create fine-grained and targeted aspects tailored to specific requirements.

AOP Concepts:

1. **Aspect:** a modularization of a concern that cuts across multiple classes. Transaction management is a good example of a crosscutting concern in enterprise Java applications. In Spring AOP, aspects are implemented using regular classes (the schema-based approach) or regular classes annotated with the @Aspect annotation (the @AspectJ style).
2. **Join point:** a point during the execution of a program, such as the execution of a method or the handling of an exception. In Spring AOP, a join point always represents a method execution.
3. **Advice:** action taken by an aspect at a particular join point. Different types of advice include "around," "before" and "after" advice. (Advice types are discussed below.) Many AOP frameworks, including Spring, model an advice as an interceptor, maintaining a chain of interceptors around the join point.

4. **Pointcut:** a predicate that matches join points. Advice is associated with a pointcut expression and runs at any join point matched by the pointcut (for example, the execution of a method with a certain name). The concept of join points as matched by pointcut expressions is central to AOP, and Spring uses the AspectJ pointcut expression language by default.
5. **Introduction:** declaring additional methods or fields on behalf of a type. Spring AOP allows you to introduce new interfaces (and a corresponding implementation) to any advised object. For example, you could use an introduction to make a bean implement an IsModified interface, to simplify caching. (An introduction is known as an inter-type declaration in the AspectJ community.)

6. **Target object:** object being advised by one or more aspects. Also referred to as the advised object. Since Spring AOP is implemented using runtime proxies, this object will always be a proxied object.
7. **AOP proxy:** an object created by the AOP framework in order to implement the aspect contracts (advise method executions and so on). In the Spring Framework, an AOP proxy will be a JDK dynamic proxy or a CGLIB proxy.
8. **Weaving:** linking aspects with other application types or objects to create an advised object. This can be done at compile time (using the AspectJ compiler, for example), load time, or at runtime. Spring AOP, like other pure Java AOP frameworks, performs weaving at runtime.

Types of advice:

1. **Before advice:** Advice that executes before a join point, but which does not have the ability to prevent execution flow proceeding to the join point (unless it throws an exception).
2. **After returning advice:** Advice to be executed after a join point completes normally: for example, if a method returns without throwing an exception.
3. **After throwing advice:** Advice to be executed if a method exits by throwing an exception.
4. **After (finally) advice:** Advice to be executed regardless of the means by which a join point exits (normal or exceptional return).
5. **Around advice:** Advice that surrounds a join point such as a method invocation. This is the most powerful kind of advice. Around advice can perform custom behavior before and after the method invocation. It is also responsible for choosing whether to proceed to the join point or to shortcut the advised method execution by returning its own return value or throwing an exception.

Spring AspectJ AOP annotations:

- @Aspect: It declares the class as an aspect.
- @Pointcut: It declares the pointcut expression.

Common AspectJ annotations used to create advice are as follows:

- @Before: It runs before the method execution.
- @AfterReturning: It runs after the result is returned by the method.
- @AfterThrowing: It runs after an exception is thrown by the method.
- @After(Finally): It is executed after method execution or after an exception is thrown or the result is returned by the method.
- @Around: It can perform the behavior before and after the method invocation.
- @Pointcut: Pointcut is a signature that matches the join points.

Steps in AOP:

- Write Aspects
- Configure where the aspects apply

Advantage:

1. Maintenance
2. Debugging
3. By having cross-cutting concerns it helps us to improve the understandability and maintainability.
4. Reuse of aspects and classes
5. Reduce the cost of coding.
6. Shorter code

Disadvantage:

1. Code bloat: In AOP small sources can lead to large objects.

2. Toolchain, profiler, and debuggers are not available.
3. Runtime overhead

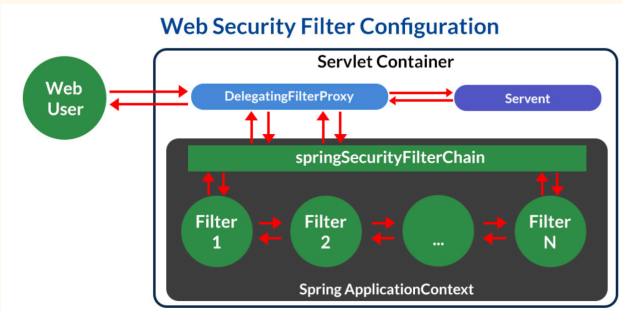
SPRING SECURITY:

Spring Security is a powerful and customizable authentication and access control framework for Java applications. This aims to make it easy for developers to secure web applications against common exploits such as **Cross-Site Request Forgery** (CSRF) attacks. It provides comprehensive security services for Java EE-based enterprise software applications. The primary goal of Spring Security is to simplify and standardize the implementation of security features in Java applications, including securing web applications, REST APIs, and method invocations. Internally, the spring security framework contains a series of servlet filters that handle various aspects of security.

Spring security works on the following four core concepts:

1. Authentication – Is the user really who he claims to be?
2. Authorization – Does the user have the appropriate role?
3. Password Storage – How is the password stored? In Memory or a database.
4. Servlet Filters – Are there any new filters that we need to add or just use the default ones provided by the spring team?

1. **Authentication:** The identity of users is verified for providing access to the system. If the user is verified as per the saved credentials, the request is accepted and the desired response is given to the client from the server. Some of the methods are as follows:
   1.1. **Login Form -** It is a web page to a website that requires user identification and authentication, performed by entering a username and password.
   1.2. **HTTP authentication -** In this, the server can request authentication information (user ID and password) from a client.
   1.3. **Customer Authentication Method -** customer authentication is a new regulation designed to prevent online transaction fraud.
2. **Authorization:** Giving the user permission to access a specific resource or function. Some of the methods:
   2.1. **Access Control for URLs -** Security of URLs allows you to restrict access to specific Internet sites based on the contents of the URL(keywords).
   2.2. **Secure Objects and Methods -** The Class method is called by a security interceptor implementation to ensure that the configured AccessDecisionManager supports the type of secure object or not.
   2.3. **Access Control Lists -** An ACLs specifies which users are granted access to objects, as well as what operations are allowed to them.



3. **Filter:** A filter is a function that is invoked at the time of preprocessing and postprocessing of a request. Spring Security maintains a filter chain where all filters have different responsibilities and filters are added or removed depending on which services are required.

Advantages of Spring Security:

These are some of the major advantages of Spring security.

- Protection against attacks like session fixation, CSRF and clickjacking.
- Spring MVC integration.
- Support Java Configuration.
- Portable

- Integration of Servlet API
- Protect against brute force attacks.
- Active community and open source, with updates against new exploits.

Spring Security Features:

1. Authorization
2. Single sign-on
3. Software Localization
4. Remember-me
5. LDAP (Lightweight Directory Access Protocol)
6. JAAS (Java Authentication and Authorization Service) LoginModule
7. Web Form Authentication
8. Digest Access Authentication
9. HTTP Authorization
10. Basic Access Authentication

1. Authorization: This functionality is provided by Spring Security and allows the user to be authorized before accessing resources. It enables developers to set access controls for resources.
2. Single sign-on: This feature allows a user to utilize a single account to access different apps (username and password).
3. Software Localization: This capability enables us to create user interfaces for applications in any language.
4. Remember-me: With the help of HTTP Cookies, Spring Security provides this capability. It remembers the user and prevents them from logging in from the same workstation until they log out.
5. LDAP (Lightweight Directory Access Protocol): That is an open application protocol for managing and interacting with dispersed directory information services over the Internet Protocol.
6. JAAS (Java Authentication and Authorization Service) LoginModule: This is a Java-based Pluggable Authentication Module. It is supported by Spring Security's authentication procedure.
7. Web Form Authentication: Web forms capture and authenticate user credentials from the web browser during this procedure. While we wish to build web form authentication, Spring Security supports it.
8. Digest Access Authentication: We can make the authentication procedure more secure with this functionality than with Basic Access Authentication. Before delivering sensitive data over the network, it requests that the browser verify the user's identity.
9. HTTP Authorization: Using Apache Ant paths or regular expressions, Spring provides this functionality for HTTP authorization of web request URLs.
10. Basic Access Authentication: Spring Security has support for Basic Access Authentication, which is used to give a user name and password when performing network requests.

Features Added in Spring Security 6.0:

1. OAuth 2.0 Login: This feature allows users to connect to the app using their current GitHub or Google accounts. The Authorization Code Grant defined in the OAuth 2.0 Authorization Framework is used to implement this functionality.
2. Reactive Support: Spring Security 6.0 adds support for reactive programming and reactive web runtimes, as well as the ability to interact with Spring WebFlux.
3. Modernized Password Encoding: Spring Security 6.0 introduces the DelegatingPasswordEncoder, a new way to store passwords. The format for storing passwords is: {id} encodedPassword. List of ids for various password encoders are:

- {bcrypt}$2a$10$dXJ3SW6G7P50lGmMkkmwe.20cQQubK3.HZWzG3YB1tlRy.fqvM/BG
- {noop}password

- {pbkdf2}5d923b44a6d129f3ddf3e3c8d29412723dcbde72445e8ef6bf3b508fbf17fa4ed4d6b99ca763d8dc
- {scrypt}$e0801$8bWJaSu2IKSn9Z9kM+TPXfOc/9bdYSrN1oD9qfVThWEwdRTnO7re7Ei+fUZRJ68k9lTyuTeUp4of4g24hHnazw==$OAOec05+bXxvuu/1qZ6NUR+xQYvYv7BeL1QxwRpY5Pc=
- {sha256}97cde38028ad898ebc02e690819fa220e88c62e0699403e94fff291cfffaf8410849f27605abcbc0

How does Spring Security Work?

1. **Filter Chain:** At the core of Spring Security is a filter chain that intercepts incoming requests and performs security-related tasks. The filter chain consists of various filters responsible for tasks such as authentication, authorization, session management, and more.
2. **SecurityContextHolder:** The `SecurityContextHolder` is a central repository that holds security-related information, such as the authenticated user's principal and authorities. It is accessible throughout the application and is used by various components to obtain security context information.
3. **AuthenticationManager:** The `AuthenticationManager` is responsible for authenticating users. It typically delegates authentication to one or more AuthenticationProvider implementations. Developers can configure multiple authentication providers to support different authentication mechanisms.
4. **AuthenticationProvider:** An `AuthenticationProvider` is responsible for verifying the credentials provided by a user during the authentication process. It supports various authentication mechanisms, such as in-memory authentication, JDBC-based authentication, LDAP authentication, and more.
5. **UserDetailsService:** The `UserDetailsService` interface is used to load user-specific data during the authentication process. Developers can implement a custom `UserDetailsService` to load user details from a database or any other external source.
6. **Access Control:** Access control is enforced through the configuration of security rules. Developers can define rules based on URL patterns, HTTP methods, and user roles using expressions, annotations, or configuration. Spring Security checks these rules to determine whether a user has the required permissions to access a particular resource.
7. **AuthenticationEntryPoint:** The `AuthenticationEntryPoint` handles the initiation of the authentication process when an unauthenticated user attempts to access a secured resource. It may trigger a login form, redirect to an external authentication provider, or return an error response.
8. **Security Annotations:** Spring Security supports annotations like @Secured, @PreAuthorize, @PostAuthorize, and @RolesAllowed for method-level security. These annotations allow developers to express access control rules directly within the code.

Spring Security provides a comprehensive solution for securing Java applications, and its modular and extensible nature makes it suitable for a wide range of use cases. By configuring and customizing various components, developers can tailor Spring Security to meet the specific security requirements of their applications.

Spring Security	OAuth2
Security framework that provides authentication and authorization.	Authorization framework designed for token-based authentication.

Can be configured to use OAuth2 as an authentication method.	A standalone protocol that can be integrated into various applications.
Specific to the Spring ecosystem.	Universal protocol not limited to Spring or Java.

JWT (JSON Web Token):

JWT (JSON Web Token) and OAuth 2.0 are two related but distinct technologies commonly used in modern web and mobile applications for authentication and authorization.

1. **Definition:** JWT is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. It is commonly used to represent claims between two parties, such as an identity provider (IDP) and a resource server.
2. **Structure:** JWTs consist of three parts: a header, a payload, and a signature. These parts are Base64Url-encoded and separated by dots.

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIy.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c

   a. **Header:** Describes the cryptographic operations applied to the JWT.

Decoded header:

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

In the header, "alg" indicates the algorithm used for the signature, and "typ" indicates the type of token, which is JWT in this case. The algorithm here is HMAC SHA-256 (HS256).

   b. **Payload:** Contains claims. Claims are statements about an entity (typically, the user) and additional data.

Decoded payload:

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

In the payload, "sub" represents the subject (user identifier), "name" is the user's name, and "iat" is issued at timestamp.

   c. **Signature:** Used to verify that the sender of the JWT is who it says it is and to ensure that the message wasn't changed along the way.

SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c

The signature is created by applying the specified algorithm (HS256) to the encoded header, encoded payload, and a secret key known only to the issuer.

3. **Use Cases:** JWTs are commonly used for authentication and authorization in stateless applications, such as single-page applications (SPAs) and microservices. They are also often used in OAuth 2.0 for access tokens.

OAuth 2.0:

[OAuth terminologies and flows explained - OAuth tutorial - Java Brains]

1. **Definition:** OAuth 2.0 is an authorization framework that enables third-party applications to obtain limited access to a user's resources without exposing the user's credentials. It is widely used for delegated access scenarios, such as allowing a third-party application to access a user's data on another service.

2. **Components:**

2.1. **Resource Owner (User):** The entity that can grant access to a protected resource. It is typically the end-user. So the resource owner is the end-user who owns the resources (such as data or services) that the client (third-party application) wants to access. Example: In a social media application, the resource owner is the user who owns the profile and posts.

2.2. **Client:** The client is the application that requests access to the protected resource on behalf of the resource owner. It can be a web application, mobile app, or other types of applications. **Example:** A mobile application wants to access a user's photos from a photo-sharing service.

2.3. **Authorization Server:** The server that authenticates the resource owner and issues access tokens to the client after obtaining proper authorization. **Example:** Google's authorization server, which authenticates a user and issues access tokens to third-party applications.

2.4. **Resource Server:** The server that hosts the protected resources that the client wants to access. It verifies the access token and serves the requested resources. **Example:** A cloud storage service where a user's files are stored and accessed.

2.5. **Authorization Grant:** The authorization grant is a credential representing the resource owner's consent to allow the client to access specific resources. **Example:** Types of authorization grants include authorization code, implicit, client credentials, and resource owner password credentials.

2.6. **Access Token:** The access token is a credential representing the authorization granted to the client. It is presented by the client when making requests to access protected resources. **Example:** A string like eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9... that represents the user's authorization.

3. OAuth 2.0 Flow Types:

3.1. **Authorization Code Grant:** Suitable for server-side applications where the client secret can be kept confidential. It involves obtaining an authorization code and exchanging it for an access token. **Example:** A web application that redirects the user to an authorization server for authentication and receives an authorization code in return.

3.2. **Implicit Grant:** Designed for clients that run in a user's browser, where the client credentials cannot be kept confidential. The access token is obtained directly. **Example:** A single-page application (SPA) that authenticates the user in the browser and receives an access token without involving a back-end server.

3.3. **Client Credentials Grant:** Used for machine-to-machine communication, where the client is acting on its own behalf rather than on behalf of a user. It involves the client presenting its credentials (client ID and secret) to the authorization server. **Example:** A server-side application accessing a resource server on its own behalf.

3.4. **Resource Owner Password Credentials Grant:** Allows the client to directly obtain user credentials and exchange them for an access token. It is not recommended for client applications, but it's useful for trusted clients. **Example:** A legacy application where the client directly takes the user's username and password and exchanges them for an access token.

OAuth 2.0 Example Flow:

- The user (resource owner) logs into the client application.
- The client application redirects the user to the authorization server.
- The user authenticates with the authorization server.
- The authorization server returns an authorization code to the client.
- The client exchanges the authorization code for an access token.

- The client uses the access token to make requests to the resource server on behalf of the user.

4. **OAuth 2.0 and JWT:** OAuth 2.0 can be used with JWTs as a token format. When OAuth 2.0 is combined with JWTs, the access tokens issued by the authorization server are JWTs. These JWTs can be self-contained and carry information about the user, scopes, and other claims.

5. **Use Cases:** OAuth 2.0 is widely used for enabling third-party access to protected resources in a secure and standardized way. It is commonly used in scenarios such as user authentication, authorization, and access delegation in web and mobile applications.

6. **Access Tokens:** In OAuth 2.0, access tokens are used to represent the authorization granted to a client. These tokens may be opaque, or they may be in JWT format (known as JWT access tokens).

In summary, while JWT is a token format commonly used for representing claims, OAuth 2.0 is a framework for authorization. OAuth 2.0 can use JWTs as a token format, and the combination of OAuth 2.0 and JWTs is often used for securing modern web and mobile applications.

Cookies, Session and Tokens:

Cookies, sessions, and tokens are all mechanisms used in web development to manage and maintain user identity and state. Each has its own characteristics, use cases, and advantages. Here are the key differences between cookies, sessions, and tokens:

Cookies:

1. **Definition:** Cookies are small pieces of data stored on the client's browser. They are sent back and forth with each HTTP request and response between the client and the server.
2. **Storage Location:** Stored on the client side (browser).
3. **Data Format:** Key-Value Pairs- Cookies typically store data as key-value pairs, including information such as user preferences, session identifiers, or tracking data.
4. **Purpose:**
    4.1. **State Management:** Cookies are often used for state management, user preferences, and tracking information.
    4.2. **Security:** Cookies can be susceptible to security vulnerabilities, such as cross-site scripting (XSS) and cross-site request forgery (CSRF) attacks. Developers can mitigate these risks through secure practices and attributes like HttpOnly and Secure.
5. **Scope:** Domain and Path - Cookies can have domain and path attributes that define the scope of their availability.
6. **Expiration:** Cookies have an expiration time and they can be set to persist for a specific duration or until the browser is closed.

Sessions:

1. **Definition:** Sessions are a server-side mechanism for tracking user state. A session is typically initiated when a user logs in, and a unique session identifier is assigned.
2. **Storage Location:** Stored on the server side.
3. **Data Format:** Key-Value Pairs - Session data is often organized as key-value pairs.
4. **Purpose:**
    4.1. **User State:** Sessions are used to store user-specific data and maintain the state of a user across multiple requests.

4.2. **Security:** Sessions are generally considered more secure than cookies because the data is stored on the server. However, session fixation and session hijacking are potential security concerns that need to be addressed.

5. **Expiration:** Sessions have a configurable timeout, and the session data is usually cleared when the user logs out or after a period of inactivity.

Tokens:

1. **Definition:** Tokens, particularly JSON Web Tokens (JWT), are compact, URL-safe means of representing claims to be transferred between two parties. JWTs can be used for authentication and authorization.
2. **Storage Location:** Tokens can be stored on the client-side (e.g., as cookies or local storage) or server-side.
3. **Data Format:** Tokens are often encoded and can carry more complex data structures like JSON Web Tokens (JWTs). JWTs consist of three parts: a header, a payload, and a signature. They can contain user information, expiration time, and other claims. They are often used for stateless authentication.
4. **Purpose:**
   4.1. **Authentication and Authorization:** Tokens are commonly used for authentication and authorization. They represent a user's identity and contain claims that grant specific permissions.
   4.2. **Security:** JWTs are secure if implemented correctly with security features, including digital signatures for integrity and encryption for confidentiality. The data is signed (and optionally encrypted), ensuring integrity. However, developers need to be cautious about securely handling tokens, avoiding vulnerabilities like token leakage.
5. **Expiration:** Tokens often have a configurable expiration time, and their lifetime can be extended by refreshing them.

Use Cases:

- **Cookies:** User preferences, tracking user behavior, maintaining a user's session.
- **Sessions:** User authentication, maintaining user-specific data, storing temporary information during a user's visit.
- **Tokens:** User authentication in stateless environments (e.g., single-page applications), authorizing API requests, transmitting claims and permissions.

Interaction:

- **Cookies and Sessions:** Cookies are often used to store a session identifier. The server uses this identifier to look up session data.
- **Tokens and Sessions:** Tokens can replace sessions in some scenarios, especially in stateless architectures where the server does not store session data. Tokens can carry user identity and authorization information.

Summary:

1. **Storage Location**:
   a. Cookies: Client side (browser)
   b. Sessions: Server side
   c. Tokens: Client-Side or Server-Side
2. **Content:**
   a. Cookies: Key-value pairs
   b. Sessions: Server-stored data, identified by a session identifier
   c. Tokens: Compact, signed (and optionally encrypted) data, often containing user claims

3. **Security:**
    a. Cookies: Can be vulnerable to XSS and CSRF attacks
    b. Sessions: Generally considered more secure than cookies
    c. Tokens: Secure if implemented correctly, but need to be handled with care to avoid token leakage
4. **Expiration:**
    a. Cookies: Have an expiration time
    b. Sessions: Have a configurable timeout
    c. Tokens: Include an expiration claim

In summary, cookies, sessions, and tokens serve different purposes in web development. Cookies and sessions are more traditional mechanisms for managing user state, while tokens are commonly used in modern architectures, particularly for authentication and authorization in stateless environments. The choice between them depends on the specific requirements and constraints of the application.