Java Spring

- Very popular loosely coupled framework for building Java Applications
- Reduces Java boiler plate code
- Spring Framework enables POJO (Plain Old Java Object) Programming which in turn enables continuous integration and testability.
- Managing services using Application context and object creation using Inversion of control
- Managing dependencies between various objects using Dependency Injection
- Data Access to simplify connection to databases using Spring JDBC
- ORM - provides integration layers for popular object-relational mapping APIs, such as Spring JPA
- Ability to build web apps and Rest API using Spring MVC
- Aspect Oriented Programming (AOP) - Supporting development separating application business logic from system services
- In addition, there are various modules in Spring framework like Spring Core Container, AOP, Web, Data Access, Test.

Spring Modules

Module	Description
Core Container	Core features including IoC, DI, and context management.
Spring Core	Provides IoC and DI functionalities.
Spring Beans	Manages beans within the application context.
Spring Context	Offers framework-style access to objects.
Spring SpEL	Expression language for querying and manipulating object graphs.
Data Access/Integration	Simplifies data access and integration with various data sources.
Spring JDBC	Simplifies JDBC code and provides transaction management.
Spring ORM	Integrates with ORM frameworks like Hibernate and JPA.
Spring OXM	Supports Object/XML mapping with different libraries.
Spring JMS	Simplifies JMS API usage and integrates message-driven POJOs.
Spring Transaction	Provides a consistent model for transaction management.

Web	Facilitates web application development with MVC and reactive support.
Spring Web	Basic web integration features.
Spring WebMVC	Implements the Model-View-Controller (MVC) pattern.
Spring WebFlux	Provides reactive programming support for web applications.
AOP and Instrumentation	Offers aspect-oriented programming and instrumentation capabilities.
Spring AOP	Enables aspect-oriented programming for cross-cutting concerns.
Spring Aspects	Integrates with AspectJ for advanced AOP features.
Spring Instrumentation	Supports class instrumentation and classloader implementations.
Messaging	Provides support for messaging architectures.
Spring Messaging	Supports messaging protocols like STOMP.

Testing	Facilitates unit and integration testing of Spring components.
Spring Test	Supports testing with consistent loading of the Spring IoC container.
Additional Modules	Offers specialized functionalities for security, cloud, data, and more.
Spring Security	Provides comprehensive security services.
Spring Boot	Simplifies Spring application development with convention-over-configuration and production-ready features.
Spring Data	Simplifies data access with a repository abstraction over various data stores.
Spring Cloud	Provides tools for building cloud-native applications, including support for microservices patterns.
Spring Batch	Offers robust batch processing capabilities.

Spring Boot vs Spring

- Spring Boot is built on top of spring to simplify the development of Spring apps
- Minimal setup with auto configuration features
- XML configuration no longer necessary
- Quick project setup using Spring Initializr
- In built embedded server
- Optimized for microservices with built-in support for RESTful services, Spring Cloud integration.

Springboot and Java compatibility

- Springboot 3.x - Java 17, 21, 23
- Springboot 2.7x - Java 8, 11, 17

Spring Boot start & work

➢

Q3) Working of Spring Boot

1. Spring Boot starts by scanning the starter dependencies in pom.xml
   Then **download** and **auto-configure** the module as you included in pom.xml
2. For example we have to create web application then we have to put **spring-boot-starter-web** dependency in pom.xml.
   When we start the project spring boot downloads all the dependency required for web and configure the things like spring mvc.

➢

Q4) How Spring Boot Starts?

1. Starts by calling **main()** method of you main class.
2. The **run()** method of SpringApplicaiton is called. This method starts the application by creating an application **context** and **initializing** it.
3. Once the application context is initialized, the run() method starts the application's embedded web server.

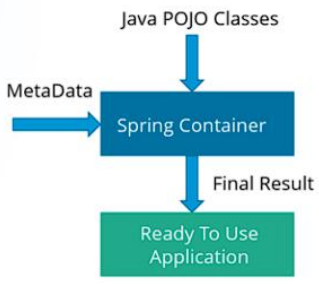
```
@SpringBootApplication
public class SpringBootWorkApplication {

    public static void main(String[] args) {
        SpringApplication.run(SpringBootWorkApplication.class, args);
    }
}
```

Spring IOC Container

- Part of the core of Spring Framework
- Responsible for managing all the beans
- Performs dependency injection

What is Spring IOC Container?

At the core of the Spring Framework, lies the Spring container. The container creates the object, wires them together, configures them and manages their complete life cycle. The Spring container makes use of Dependency Injection to manage the components that make up an application. The container receives instructions for which objects to instantiate, configure, and assemble by reading the configuration metadata provided. This metadata can be provided either by XML, Java annotations or Java code.



Spring Bean

- An instance of a class (object) managed by Spring Container

Dependency Injection

Dependency injection, an aspect of Inversion of Control (IoC), is a general concept stating that we do not create our objects manually but instead describe how they should be created.

It allows a class to receive its dependencies from an external source rather than creating them internally

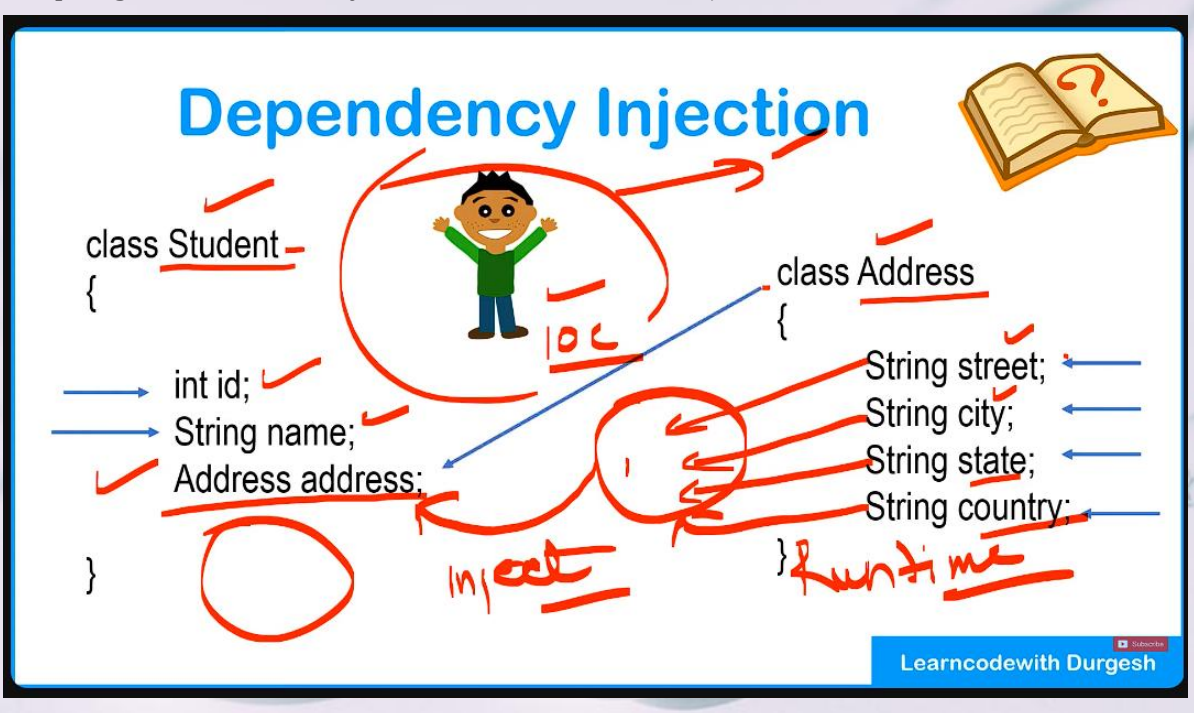
You don't connect your components and services together in the code directly, but describe which services are needed by which components in the configuration file. Dependency injection can be done in three ways, namely :

Constructor Injection

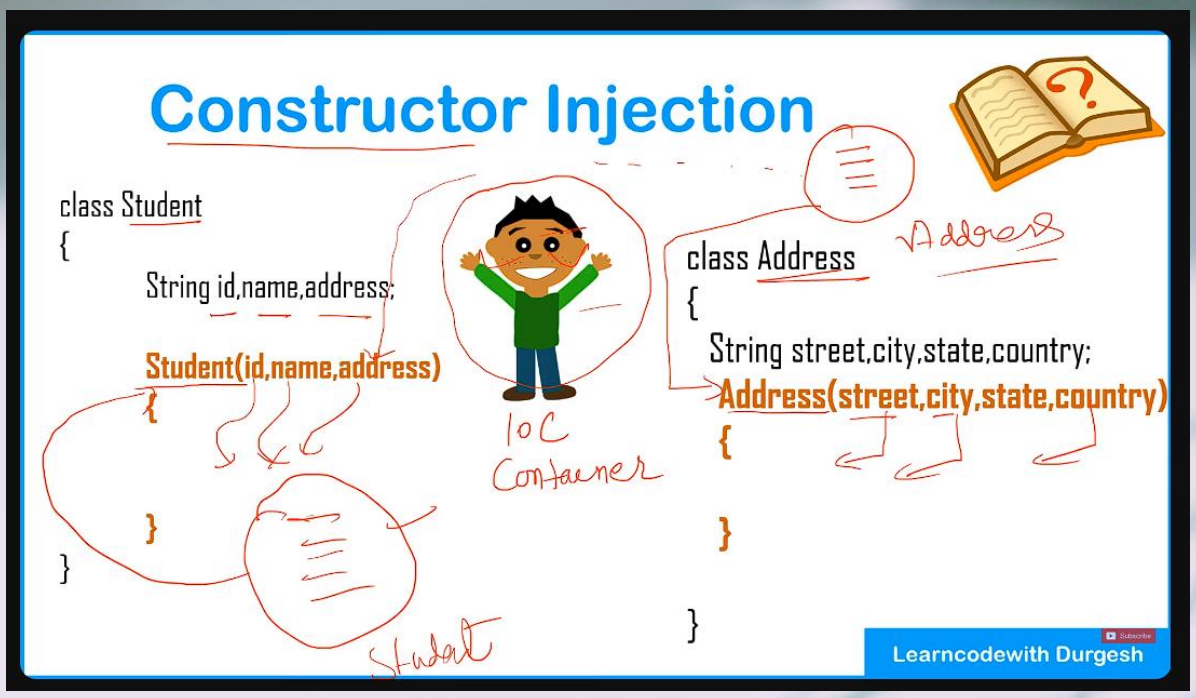
Setter/Property Injection

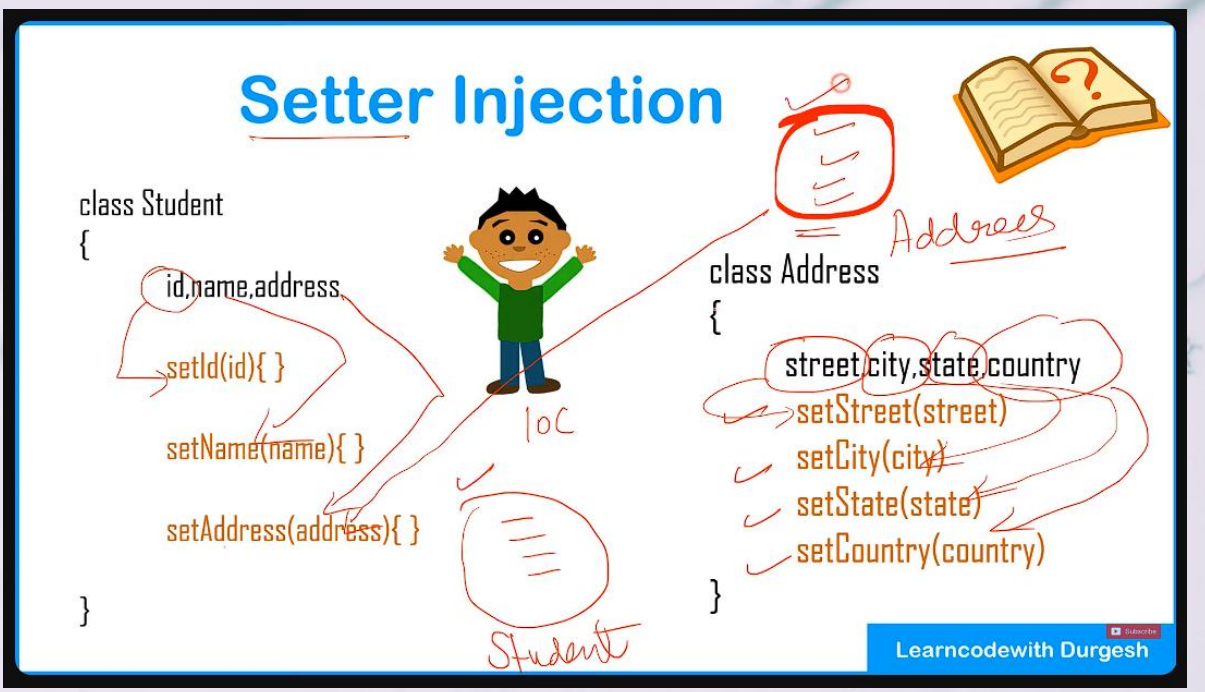
Interface Injection

In Spring Framework, only constructor and setter injections are used.



Constructor Injection vs Setter Injection





Recommended approach is to use constructor arguments for mandatory dependencies and setters for optional ones. This is because constructor injection allows injecting values to immutable fields and makes testing easier.

Example of *setter injection* in config file:

If we need to add Student bean

App.java | Student.java | config.xml

```xml
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context.xsd">


    <bean class="com.springcore.Student" name="student1">
        <property name="studentId">                        <!-- setter injection -->
            <value>10</value>
        </property>
        <property name="studentName">
            <value>Messi</value>
        </property>
        <property name="studentAddress">
            <value>PSG</value>
        </property>
    </bean>

```

If we need to Class B object in Class A

```xml


    <bean class="com.springcore.ref.B" name="bbean">
        <property name="y" value="7" />
    </bean>

    <bean class="com.springcore.ref.A" name="abean">
        <property name="x" value="10" />
        <property name="obj" ref="bbean"/>

    </bean>
```

If we are using constructor injection(no need of setters in class)

Person.java | Certificates.java | ci-config.xml

```xml
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context.xsd">


    <bean name="cert" class="com.springcore.ci.Certificates">
        <constructor-arg value="Java" />
    </bean>

    <bean name="person" class="com.springcore.ci.Person">
        <constructor-arg value="1" />
        <constructor-arg value="Cech" />
        <constructor-arg ref="cert" />      <!--injecting another object of another class using CI  -->
        <constructor-arg>                   <!--injecting collection using CI  -->
            <list>
                <value>Federer</value>
                <value>Nadal</value>
                <value>Thiem</value>
            </list>
        </constructor-arg>
    </bean>

```

Differences:

Constructor Injection vs Setter Injection

Constructor Injection	Setter Injection
There is no partial injection.	There can be partial injection.
It doesn't override the setter property.	It overrides the constructor property.
It will create a new instance if any modification is done.	It will not create new instance if any modification is done.
It works better for many properties.	It works better for few properties.

IOC Containers (Bean Factory and Application Context)

- BeanFactory is the basic container whereas ApplicationContext is the advanced container (built on top of). ApplicationContext extends the BeanFactory interface. ApplicationContext provides more facilities than BeanFactory such as integration with spring AOP, message resource handling for i18n etc.
- BeanFactory default implementation instantiates beans lazily when getBean() is called. Application Context default implementation instantiates beans eagerly when the application starts.

a. **BeanFactory**: BeanFactory is like a factory class that contains a collection of beans. It instantiates the bean whenever asked for by clients.
b. **ApplicationContext**: The ApplicationContext interface is built on top of the BeanFactory interface. It provides some extra functionality on top BeanFactory.

14. Differentiate between BeanFactory and ApplicationContext.

BeanFactory vs ApplicationContext

BeanFactory	ApplicationContext
It is an interface defined in org.springframework.beans.factory.**BeanFactory**	It is an interface defined in org.springframework.context.**ApplicationContext**
It uses Lazy initialization	It uses Eager/ Aggressive initialization
It explicitly provides a resource object using the syntax	It creates and manages resource objects on its own
It doesn't supports internationalization	It supports internationalization
It doesn't supports annotation based dependency	It supports annotation based dependency

Autowiring

Autowiring is an **easy way of achieving dependency injection automatically in Spring**.It internally uses setter or constructor injection. (Manual injection is like using <ref> tag in xml in earlier examples)
Autowiring can't be used to inject primitive and string values. It works with reference only. It requires the less code because we don't need to write the code to inject the

dependency explicitly.

The autowiring modes are given below:

No.	Mode	Description
1)	no	this is the default mode, it means autowiring is not enabled.
2)	byName	injects the bean based on the property name. It uses setter method.
3)	byType	injects the bean based on the property type. It uses setter method.
4)	constructor	It injects the bean using constructor

Autowire example using xml (no need to use <ref> tag):

Address.java | EmployeeRun.java | Employee.java | auto-wire-config.xml

```java
package com.springcore.auto.wire;

public class Employee {
    private Address add111;   //if autowire byName, this 'add111' will be used to search bean name in xml
                              //if autowire byType, this 'Address' class will be used to search in xml
    public Employee() {
        super();
        // TODO Auto-generated constructor stub
    }

    public Employee(Address add) {
        super();
        this.add111 = add;
        System.out.println("Inside constructor");
    }

    public Address getAdd() {
        return add111;
    }

    public void setAdd(Address add) {
        System.out.println("Inside setter");
```

Address.java | EmployeeRun.java | Employee.java | auto-wire-config.xml

```xml

    <bean class="com.springcore.auto.wire.Address" name="add111">
        <property name="city" value="London"/>
        <property name="country" value="England"/>
    </bean>

    <!-- byName matches the bean name and the object name created in Employee class -->
    <bean class="com.springcore.auto.wire.Employee" name="emp" autowire="byName" />

    <!-- byName matches the bean Type i.e. class name, Address class, if there are 2 beans of same type(class),
    then it may throw exception -->
    <bean class="com.springcore.auto.wire.Employee" name="emp" autowire="byType" />

    <!--using byconstructor, then it uses constructor instead of setter  -->
    <bean class="com.springcore.auto.wire.Employee" name="emp" autowire="constructor" />

    <!-- this bean will throw duplicate error in byType case -->
    <!-- <bean class="com.springcore.auto.wire.Address" name="add2">
            <property name="city" value="Madrid"/>
            <property name="country" value="Spain"/>
        </bean> -->

```

Another example of autowiring using @Autowired annotation and using @Qualifier tp avoid duplicate bean error

```java
public class Employee {

	//adding @Autowired above property for automatic creating ref object & injecting values
		// no need to write anything xtra in xml, similar to autowire byType
    @Autowired
    @Qualifier("add2")      //this is used to avoid duplicate bean error
    private Address address;

    public Employee() {
        super();
        // TODO Auto-generated constructor stub
    }

    //@Autowired               //adding this above constructor for automatic creating ref object
    public Employee(Address add) {
        super();
        this.address = add;
        System.out.println("Inside constructor");
    }

    //@Autowired               //adding this above setter for automatic creating ref object
    public void setAdd(Address add) {
        System.out.println("Inside setter");
        this.address = add;
    }
```

Just creating bean in xml, nothing much (xml can be avoided later by totally using annotations later)

```xml
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context.xsd">


    <context:annotation-config />

    <bean class="com.springcore.auto.wire.annotation.Address" name="add1">
        <property name="city" value="London"/>
        <property name="country" value="England"/>
    </bean>

    <bean class="com.springcore.auto.wire.annotation.Address" name="add2">
        <property name="city" value="Paris"/>
        <property name="country" value="France"/>
    </bean>


    <!-- just created bean,nothing else added -->
    <bean class="com.springcore.auto.wire.annotation.Employee" name="emp" />
```

Bean Scope

In order to set Spring Bean's scope, we can use @Scope annotation or "scope" attribute in XML configuration files. Note that there are five supported scopes:

Singleton

(Default) Creates a single shared instance of the bean. Singleton beans are not thread-safe, as thread safety is about execution, whereas the singleton is a design pattern focusing on creation. Thread safety depends only on the bean implementation itself.

Prototype

Creates a new bean instance for each request

Request
Creation of bean instance per HTTP request
Session
Creation of bean instance per HTTP session
Global-session
Creation of bean instance per Global HTTP session

Example of prototype bean scope:

stereo-config.xml | Student.java | Student2.java | StudentRun.java

```java
package com.springcore.stereotype;

import org.springframework.beans.factory.annotation.Value;

@Component
@Scope("prototype")
public class Student2 {

    @Value("9")
    private int id;

    @Value("Torres")
    private String name;

    public Student2() {
        super();
        // TODO Auto-generated constructor stub
    }

    @Override
    public String toString() {
        return "Student2 [id=" + id + ", name=" + name + "]";
    }

```

stereo-config.xml | Student.java | Student2.java | StudentRun.java

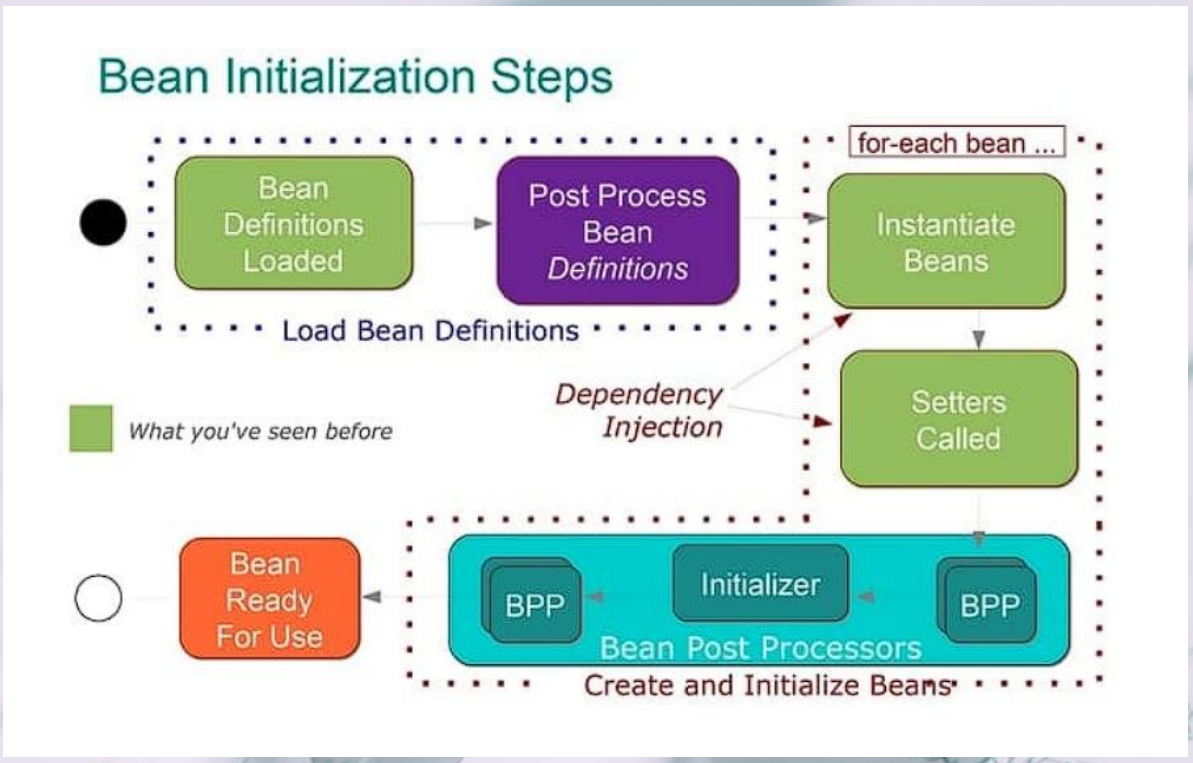
```java
package com.springcore.stereotype;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class StudentRun {

    public static void main(String[] args) {

        ClassPathXmlApplicationContext context =
                new ClassPathXmlApplicationContext("com/springcore/stereotype/stereo-config.xml");

        //'student' is default bean name for 'Student' class
        Student s = context.getBean("student", Student.class); //'student' is default bean name for

        System.out.println(s);

        Student2 sa = context.getBean("student2", Student2.class); //'student2' is default bean name
        Student2 su = context.getBean("student2", Student2.class);

        System.out.println(sa.hashCode());    //same hashcode for both objects if singleton scope
        System.out.println(su.hashCode());  //different if prototype scope is used as new obj created
        System.out.println(sa);

    }
```

Console

```
<terminated> StudentRun [Java Application] C:\Program Files\Java\jdk-17\bin\javaw.exe  (Mar 5, 2024, 1:40:36 AM – 1:40:37 AM) [pid: 29980]
Student [id=8, name=Lampard]
131206411
2012330741
Student2 [id=9, name=Torres]
```

Bean Life Cycle

First, a Spring bean needs to be instantiated based on Java or XML bean definition. Then, Spring populates all of the properties using the dependency injection, as specified in the bean definition. The factory calls setBeanName() and setBeanFactory(). preProcessBeforeInitialization() methods are called. postProcessAfterInitialization() methods are called later. After that, when the bean is no longer required, it will be removed from the IoC container.
Initializing bean in xml: init-method or using annotation @PostConstruct
Destroying bean in xml: destroy-method or using annotation @PreDestroy



Spring Design Patters

Singleton Pattern – singleton-scoped beans
Factory Pattern – Bean Factory classes
Prototype Pattern – prototype-scoped beans
Adapter Pattern – Spring Web and Spring MVC
Proxy Pattern – Spring Aspect-Oriented Programming support
Template Method Pattern – *JdbcTemplate*, *HibernateTemplate*, etc.
Front Controller – Spring MVC *DispatcherServlet*
Data Access Object – Spring DAO support
Model View Controller – Spring MVC

ContextLoaderListener

- The ContextLoaderListener is a Spring Framework component used in Java web applications to load the Spring ApplicationContext when the web application starts
- It initializes the IoC container, manages beans, and allows components to access Spring beans for their processing.

Annotations

- @SpringBootApplication
  - Used for the main class, this class should be in the outermost package to enable creation/scanning of other beans. Inside it, annotations like @ComponentScan, @SpringBootConfiguration, @EnableAutoConfiguration are present
  - @ComponentScan(basePackages = "com.example.services")

- @Configuration
  - Creates Bean for that class

- Stereotype annotations are: @Component, @Controller, @RestController, @Service, @Repository

- @Component
  - Creates Bean for that class automatically(not applicable to method) by Spring only if classpath scan detects it
  - It works without @Configuration
  - Types are @Service, @Repository

- @Bean
  - Creates bean for a method
  - Used explicitly
  - Works only when class is annotated with @Configuration

- @Controller
  - Controller is the presentation layer. It accepts the incoming request first
  - When you annotate a class with @Controller, Spring recognizes it as a controller and allows it to handle HTTP requests and generate HTTP responses
  - The methods in a @Controller class usually return a ModelAndView object or a view name (a string) that resolves to a view template.

- @Service
  - Service layer is where the business logic gets executed

- @Repository

- Repository is the Data access layer (DAO). It interacts with the database.

- @Autowired
  - The @Autowired annotation in Spring is used to automatically wire (inject) dependencies into a Spring bean.
  - It enables automatic dependency injection, meaning that Spring will automatically find and inject the required dependencies into the bean without the need for manual configuration.

- @Qualifier
  - When you create more than one bean of the same type and want to wire only one of them with a property you can use the @Qualifier annotation along with @Autowired to remove the ambiguity by specifying which exact bean should be wired.

- @RequestMapping
  - It is used for mapping a particular HTTP request method to a specific class/ method in controller that will be handling the respective request.
  - It can be applied both at class and method level

- @RestController
  - It is combination of @Controller + @Response Body. It returns the object which is directly written as JSON or XML to HTTP response. (@Controller returns the normal view)
  - Methods in a @RestController return data objects (e.g., POJOs) or collections of objects. These objects are automatically serialized to JSON (or XML, depending on configuration) and sent back to the client. This is suitable for creating REST APIs.

- @PathVariable
  - Indicates that a method parameter should be bound to a URI template variable
  - @GetMapping("/player/{id}") //to find a player by id, notice {id}
    public ResponseEntity<Chelsea> getPlayerByKitId(@PathVariable("id") Integer id)

- @RequestParam
  - Indicates that a method parameter should be bound to a web request parameter.
  - // Using RequestParam, put url like this in postman: ..//player?id=1 (just enter key & value)
    @DeleteMapping("/player")
    public String deletePlayer(@RequestParam("id") Integer id) {

- @RequestBody

- Indicates that a method parameter should be bound to the body of the web request.
- @PostMapping("/player") //to add new player
  public ResponseEntity<Chelsea> addPlayer(@RequestBody Chelsea che) { //@RequestBody to input as json

- @ModelAttribute
  - @ModelAttribute in Spring MVC is to facilitate data transfer between the Controller and the View
  - It binds method parameters or method return values to model attributes

- @Conditional
  - Indicates that a component is only instantiated when a specified condition is met.

- @Profile
  - Indicates that a component is only registered/bean inititalized when one or more specified profiles are active.

- @SpringBootTest
  - Indicates that the annotated class is a test class that will bootstrap the entire Spring Boot context for integration testing.

- @MockBean
  - Used to add mocks to a Spring ApplicationContext.

- @EnableWebSecurity
  - Enables Spring Security's web security support

- Spring Actuator Dependency
  - Spring Boot Actuator provides a wide range of useful features for monitoring and managing your application. By leveraging its built-in endpoints and customizations, you can gain valuable insights into your application's health, performance, and configuration, while ensuring that sensitive information is properly secured.

Why in some cases Bean cannot be loaded?

- Misconfigured Component Scanning -
  - If the package containing the bean is not included in the component scan, Spring will not detect and load the bean.
- Incorrect Bean Definition -

- Errors in bean definitions in XML configuration or Java configuration can prevent the bean from being loaded. Eg: returning null

- Bean Not Marked with Stereotype Annotations
- Dependency Injection Issues
  - - If a bean has unsatisfied dependencies, it won't be loaded. This can happen if a required bean is missing or if there are circular dependencies.
    @Component
    public class MyService {
    @Autowired
    private MyRepository myRepository; // If MyRepository is not a bean, MyService won't be loaded
    }
- Profile-Specific Beans Not Active -
  - If a bean is defined for a specific profile but that profile is not active, the bean will not be loaded.
- Bean Definition Overriding -
  - If two beans with the same name are defined, one might override the other, potentially leading to the absence of the intended bean.

Circular/Cyclic Dependency in Spring

- Circular dependency in Spring occurs when two or more beans are mutually dependent on each other, creating a cycle.
- This situation can lead to problems during the bean initialization process and cause the application context to fail to start.
- Bean A->Bean B->Bean A

```java
@Component
public class BeanA {
    @Autowired
    private BeanB beanB;
}

@Component
public class BeanB {
    @Autowired
    private BeanA beanA;
}
```

- When Spring encountered this type of issue, it raise an exception BeanCurrentlyInCreationException while loading context.
- To solve it:

- Refactor the design, add interface C in between
- Use @Lazy - The injected bean will only be fully created when it's first needed and at the time of bean creation, it injects the proxy bean as a dependency.

```
@Component
public class BeanA {

    private BeanB beanB;

    @Autowired
    public BeanA(@Lazy BeanB beanB) {
        this.beanB = beanB;
    }
}
```

- Use Setter/Field Injection: Instead of going for constructor-based dependency injection in the context of circular dependency, we should go for Setter-based dependency injection as suggested by Spring documentation

```
@Component
public class BeanA {

    BeanA() {
        System.out.println("BeanA constructor called !!!");
    }

    private BeanB beanB;

    @Autowired
    public void setBeanB(BeanB beanB) {
        System.out.println("Setting property beanB of BeanA instance");
        this.beanB = beanB;
    }
}
```

Difference between @Autowired and @Inject

Both support inject bean by type but

Differences

1. **Package**: `@Autowired` is from the Spring Framework, while `@Inject` is from the Java Dependency Injection API.
2. **Configuration**: While `@Autowired` is specific to Spring, `@Inject` is part of the Java EE specification, making it potentially more portable across different frameworks.
3. **Support for Optional Dependencies**: `@Autowired` supports optional dependencies through the `required` attribute, while `@Inject` does not have such an attribute. In `@Inject`, optional dependencies are typically handled differently, often by using the `Optional` class.
4. **Additional Features**: Spring's `@Autowired` provides additional features such as support for primary beans and qualifiers, which can be used to disambiguate dependencies when multiple beans of the same type exist in the application context.

Difference between JDBC and Spring JPA

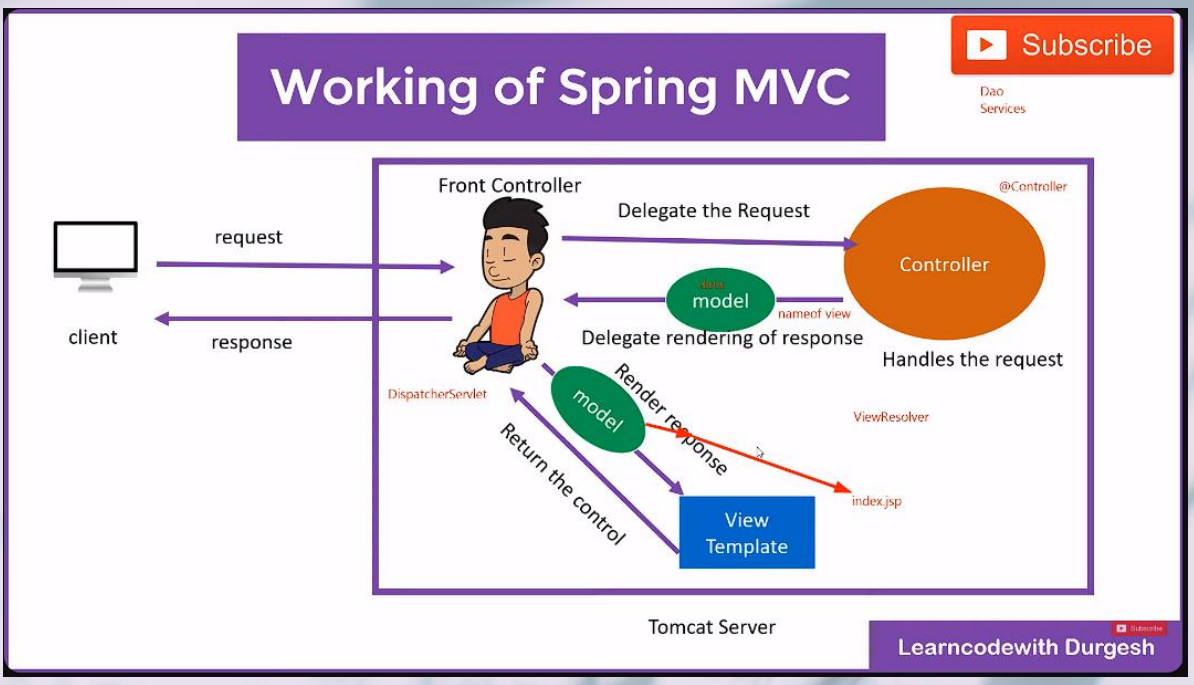
Aspect	JDBC (Java Database Connectivity)	Spring JPA (Java Persistence API)
Level of Abstraction	Low-level API	High-level API
Ease of Use	Manual coding with boilerplate	Reduced boilerplate with repository interfaces
Features	Direct SQL execution, no ORM	Full ORM capabilities, repository interfaces, JPQL, Criteria API
Transaction Management	Manual handling	Declarative transactions using annotations
Performance	Fine-tuned control, potentially better performance	Abstraction overhead, typically acceptable for most applications
Code Example	Requires detailed handling of connections, statements, and result sets	Simplified with annotations and repository methods

Spring MVC

- Spring MVC is sub framework of Spring which is used to build a web application
- It follows Model View Controller design pattern
- It is built on top of Servlet API
- Model - Basically object holding user data, View- Front end view page, Controller -

Backend code and other service

- Spring Interceptors are components in the Spring MVC framework that allow you to intercept and process HTTP requests and responses. They provide a way to perform pre-processing and post-processing tasks before and after the actual request is handled by a controller or after the response is generated.



This Spring MVC design can be made in following steps:

1. **Configure the dispatcher servlet in web.xml**
   This is done so that front controller i.e. dispatcher servlet can handle all the web requests. The DispatcherServlet is automatically instantiated and initialized by the Servlet container during web application startup
2. **Create Spring Configuration file**
   We have to create a spring configuration file like xml file with name as 'dispather servlet' to *configure the view resolver* (which handles the prefix/suffix as .jsp) and other things like spring jdbc, hibernate etc
3. **Create controller class**
   Now we have to create controller class with @RequestMapping to receive all the requests. After creating other service layers, we can send data from controller to view using Model or ModelAndView
   Eg:
   Model

```java
@Controller
@RequestMapping("/first")              //class level annotation
public class HomeController {

    //@RequestMapping("/home")           //method level annotation
    @RequestMapping(path = "/home", method = RequestMethod.GET)
    public String home(Model model) {  //if Model, then only add it in fuction parameters

        //Using Model

        model.addAttribute("club", "Chelsea CFC");
        model.addAttribute("jersey", 826);

        List<String> li = new ArrayList<String>();
        li.add("Lampard");
        li.add("Drogba");
        li.add("Terry");
        model.addAttribute("legends", li);

        return "hello";
    }
```

ModelAndView

```java
    @RequestMapping("/house")
    public ModelAndView houseDetails() {  //if ModelAndView, then change only return type

        //Using ModelAndView

        //creating ModelAndView object
        ModelAndView modelAndView = new ModelAndView();

        //setting data into it
        modelAndView.addObject("name","Ronaldo");
        modelAndView.addObject("jersey",7);
        modelAndView.addObject("name","Ronaldo");
        LocalDateTime dateTime = LocalDateTime.now();
        modelAndView.addObject("time",dateTime);

        //setting viewName
        modelAndView.setViewName("ready");

        return modelAndView;
    }

}
```

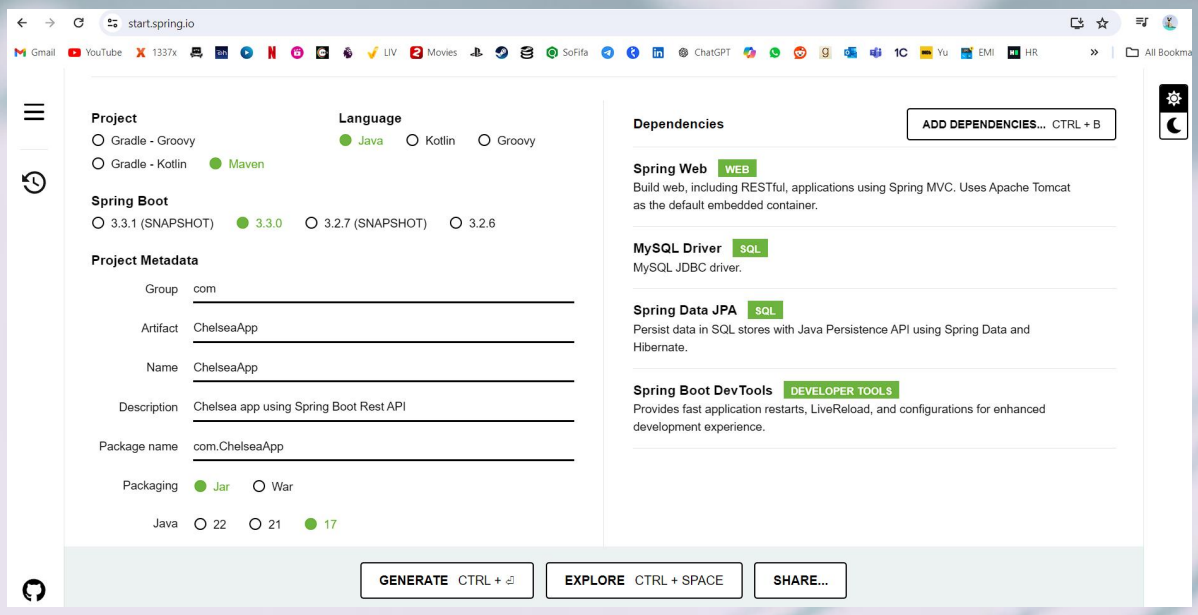
4. Creating view page

Finally we just have to create a view page like jsp with the corresponding name which was given in controller using model. Request.getAttribute("stringname") can be used to fetch the data which was put in model to go to jsp

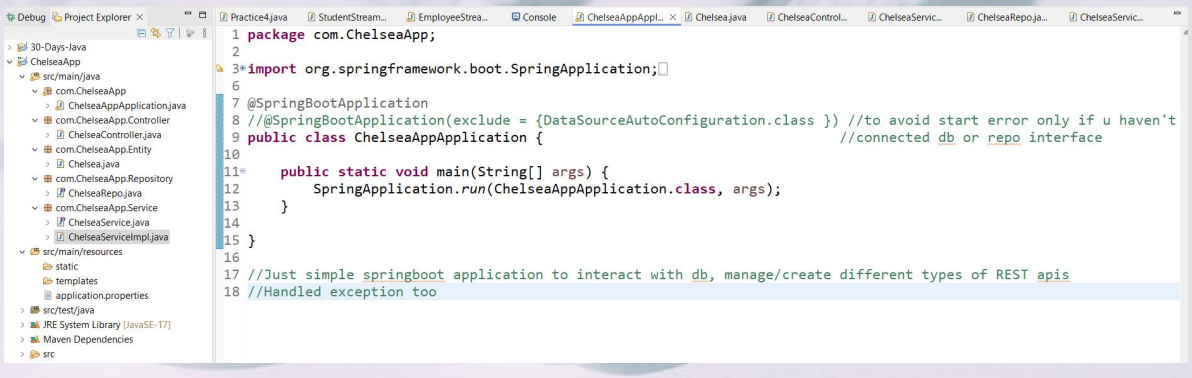
Creating Simple SpringBootApp (ChelseaApp Project)

- We are creating 'ChelseaApp' which has following CRUD Urls:
  - GET /player - Get all players
  - GET /player/{playerNo} - Get single player of given kit number
  - POST /player - Add new player
  - PUT /player - Update player
  - DELETE /player/{playerNo} - Delete player of given kit number

- Create spring boot folder using the Spring Initialzr website (start.spring.io); add the necessary dependencies



- Note: Spring Boot DevTools dependency is for auto restart server when changes are saved
- Unzip, download and extract, and import as maven project, then update maven project
- Run the below main file as Java/SpringBoot



- Now, first step after that is to create Controller class using @RestController annotation so that it can act both as controller and response body
  - Add @RestController
  - Add @RequestMapping("/chelsea") to append all the urls with prefix
  - Create the methods like
    - public List<Chelsea> getPlayers() with @GetMapping("/players")
    - public ResponseEntity<Chelsea> addPlayer(@RequestBody Chelsea che) with @PostMapping("/players")
      (Note: ResponseEntity<Object> is used as return type as it returns object+Http Status code)

```java
package com.ChelseaApp.Controller;

import java.util.List;

@RestController
@RequestMapping("/chelsea")
public class ChelseaController {

    @Autowired                          //make Spring create object (not use new object ourselves)
    ChelseaServiceImpl chelseaServiceImpl;

    @GetMapping("/home")
    public String welcome(){
        return "Welcome to Chelsea";
    }

    @PostMapping("/players")          //to add new player
    public ResponseEntity<Chelsea> addPlayer(@RequestBody Chelsea che) { //@RequestBody to input as json
        return new ResponseEntity<Chelsea>(chelseaServiceImpl.addPlayer(che),HttpStatus.OK);
    }       // return a 'new' ResponseEntity with a body and status code.

    @GetMapping("/players")       //to get all the existing players
    public List<Chelsea> getPlayers(){
        return chelseaServiceImpl.getPlayers();
    }

}
```

- Both these methods above should return chelseaServiceImpl.methods so create ChelseaServiceInterface like below and later implement them in class ChelseaServiceImpl. Add @Service annotation to both of them and then make Spring create object in controller by adding @Autowired to chelseaServiceImpl

```java
package com.ChelseaApp.Service;

import java.util.List;

@Service
public interface ChelseaService {

    public Chelsea addPlayer(Chelsea che); //add 1 player, takes 1 che object as input and returns the same

    public List<Chelsea> getPlayers(); //gets all the players from db as List<>

}
```

- Now, before creating serviceimpl class, create the entity class Chelsea. Add @Entity and @Id

```java
package com.ChelseaApp.Entity;

import jakarta.persistence.Entity;

@Entity
//@Table(name="chelsea")       //to explicitly name table
public class Chelsea {

    @Id
    //@GeneratedValue(strategy = GenerationType.AUTO)     //for auto incrementing id if needed
    private int id;
    private String name;
    private double rating;

    public Chelsea() {
        super();
    }

    public Chelsea(int id, String name, double rating) {
        super();
        this.id = id;
        this.name = name;
        this.rating = rating;
    }

    public int getId() {
        return id;
```

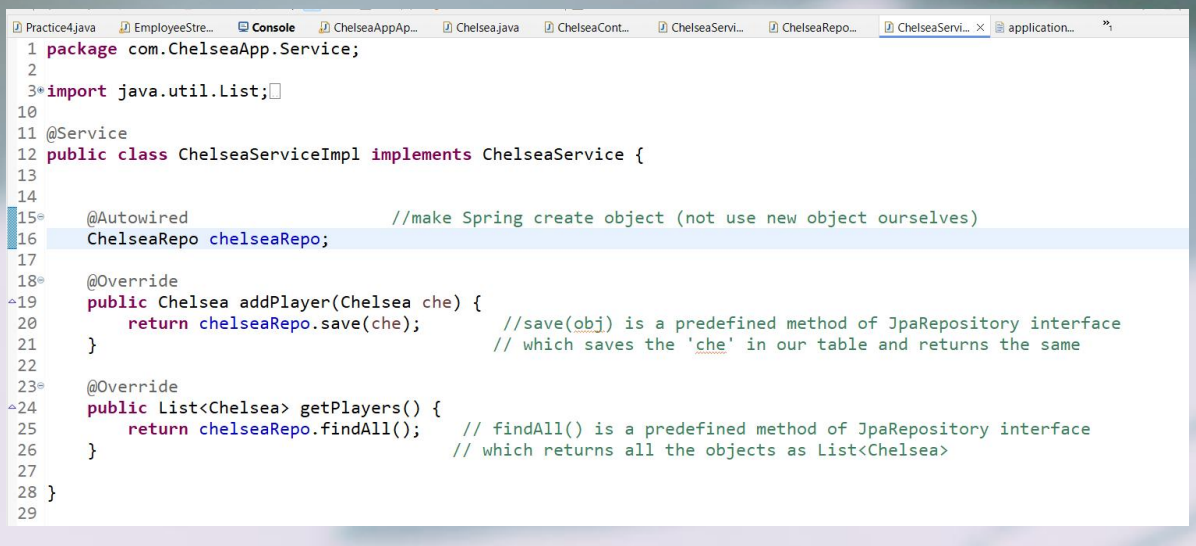
- Now, create a repo interface ChelseaRepo and extend with JpaRepository<Chelsea, Integer> to connect to DB and use in built methods of JpaRepository

```java
package com.ChelseaApp.Repository;

import java.util.List;

//@Repository is not needed i guess
public interface ChelseaRepo extends JpaRepository<Chelsea, Integer> { //<EntityName, Primary key fieldtype>

    @Query(value = "select * from chelsea c where c.rating>?", nativeQuery = true)
    public List<Chelsea> getPlayerByRating(Double rating);

}
```

- For getting DB connection, make the following changes in application.properties

```
spring.application.name=ChelseaApp

#database configuration of mysql
spring.datasource.url=jdbc:mysql://localhost:3306/book?createDatabaseIfNotExist=true
spring.datasource.username=root
spring.datasource.password=admin
spring.datasource.driver-class-name= com.mysql.cj.jdbc.Driver

#hibernate configuration
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect
#spring.jpa.hibernate.ddl-auto=create    #i think only enable 'create' first time
spring.jpa.hibernate.ddl-auto=update
logging.level.org.hibernate.sql=DEBUG
logging.level.org.hibernate.type=TRACE

#port configuration
server.port=8080
```

- Finally, go to ChelseaServiceImpl if created and implement it with ChelseaService interface and add its methods definitions. Also, create object of ChelseaRepo chelseaRepo by using @Autowired to use its methods for DB operations



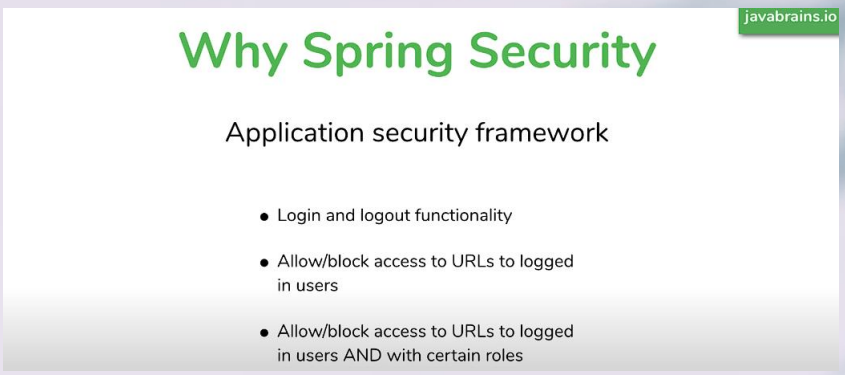
```
package com.ChelseaApp.Service;

import java.util.List;

@Service
public class ChelseaServiceImpl implements ChelseaService {


    @Autowired                    //make Spring create object (not use new object ourselves)
    ChelseaRepo chelseaRepo;

    @Override
    public Chelsea addPlayer(Chelsea che) {
        return chelseaRepo.save(che);        //save(obj) is a predefined method of JpaRepository interface
    }                                       // which saves the 'che' in our table and returns the same

    @Override
    public List<Chelsea> getPlayers() {
        return chelseaRepo.findAll();    // findAll() is a predefined method of JpaRepository interface
    }                                   // which returns all the objects as List<Chelsea>

}

```

- For all other things like
  - Http.PUT, DELETE, UPDATE methods, Global Exception handling, Swagger UI - Refer the ChelseaApp eclipse code
  - For small Spring MVC with html error pages using Thymeleaf dependency - Refer ChelseaMVC eclipse code
  - For using RestTemplate to pass 1 api from 1 microservice to other, refer the ChelseaAcademy eclipse code
  - For Cache implementation, use @EnableCaching at main program and @Cacheable(key=”#id”, value=”desiredName”) at serviceImpl class methods (for more details, refer theory somewhere)
  - For Junit testing, create Test class in test folder using annotation @SpringBootTest, use @Autowired to declare the main class object; write test case methods using @Test and mention the expected and actual results and check by using Assertions.assertEquals(expected, actual) by running the class; you can also use @Mockbean to declare main class object in Test class and then use Mockito.when(…’obj.method()’..).thenReturn(‘..expectedvalue..’); (for more details like test coverage, refer outside)
  - For environment change or reading anything from application.properties file, use @Value("${db.name}") and assign it a variable below like private String dbName. It reads the ‘db.name=mysql’ line from the property file. You can also create multiple application.properties file like application-dev.properties etc and then specify the current using one in the main application.properties file by adding line ‘spring.profiles.active=dev’

Spring Security

- Spring Security is part of application security framework
- Spring Security is a separate module of the Spring framework that focuses on providing authentication and authorization methods in Java applications.

- It also takes care of most of the common security vulnerabilities such as CSRF attacks. (Cross-Site Request Forgery (CSRF) is a type of security vulnerability in web applications. It occurs when an attacker tricks a user into performing actions on a web application in which the user is authenticated, without their knowledge or consent.)
- To use Spring Security in web applications, we can get started with the simple annotation @EnableWebSecurity.



- Spring Security Concepts
  - Authentication : who is the user ?
  - Authorization : is the user allowed to do this
  - Principal : currently logged in user
  - Granted Authority : way of providing Authorization.
  - Roles : Group of authority.

Authentication

- It basically questions the user 'Who are you?' and user need to answer it with proof
- It has 3 types of authentication
  - Knowledge based - Password, pincode etc
  - Possession based - Phone text msg, id card etc
  - Multi factor authentication - Knowledge + Possession based

Authorization

- It basically questions 'Is the user allowed to do this'
- Authentication is kinda needed first for authorization

Principal

- Principal is the currently logged in user
- It is the person you have identified through authentication

Granted Authority

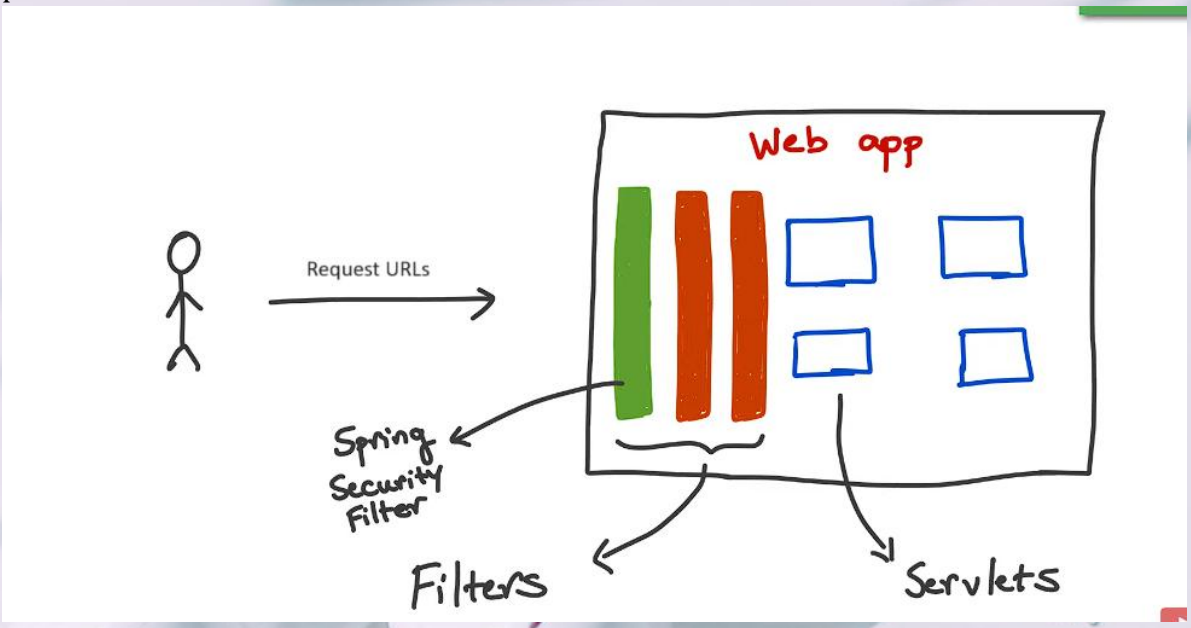
- It is the process of giving authority/permission to the users
- For example, in a shop a clerk has authorities to check inventory, make announcements etc
- Fine grained grouping of permissions

Roles

- It is a group of authorities which is assigned together
- For example, if in a shop if you hire a clerk multiple times, then you don't need to hand them all different authorities each time, you can just add them into a common 'clerk role'
- Coarse grained grouping of permissions

Basic Spring Security

- It can be achieved just by adding 'spring-boot-starter-security' dependency in pom.xml



- By default, it will create a filter for the servlets
  - Adds mandatory **authentication** for all URLs (except obviously error page)
  - Adds login form
  - Handles login error
  - Creates a 'user' and sets a default password (pwd in the console)
- We can change the default credentials by adding our own in application.properties
  - spring.security.user.name = foo
  - spring.security.user.password = boo

Configure Spring Security Authentication & Authorization

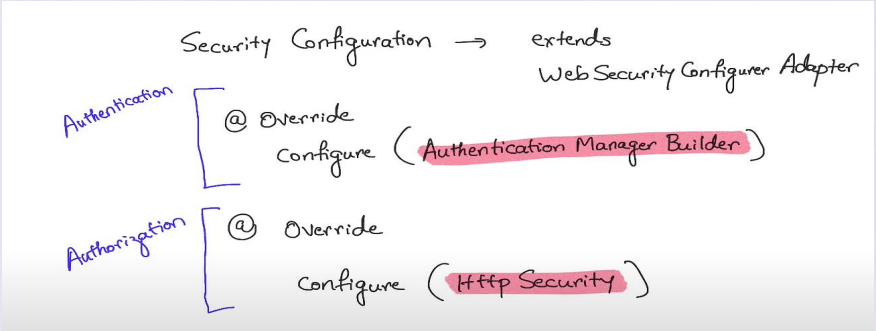
- Instead of above default thing, we can make use of Spring Security's Authentication Manager
- First we have to make sure we have older spring version and the corresponding spring security version to enable the class 'WebSecurityConfigurerAdapter' (Note: this class is now deprecated, so we can use 'SecurityFilterChain' Bean

```xml
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.7.0</version>  <!--changed spring version from 3.3 to 2.7 to enable 'WebSecurityConfigurerAdapter'' -->
        <relativePath/>
    </parent>
    <groupId>com</groupId>
    <artifactId>ChelseaAppBasicSecurity</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>ChelseaAppBasicSecurity</name>
    <description>Chelsea project for Basic Spring Boot Security</description>

    <properties>
        <java.version>17</java.version>
        <!-- The spring-security version specifically added to enable 'WebSecurityConfigurerAdapter' -->
        <spring-security.version>5.6.5</spring-security.version>
    </properties>

    <dependencies>

        <!--just adding Spring security dependency-->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-security</artifactId>
        </dependency>
```

- Now if the role apis are like:

```java
@RestController
public class MyController {

    @GetMapping("/welcome")
    public String welcome() {            //all should be able to access this irrespective of roles
        return "Welcome to Chelsea";
    }

    @GetMapping("/fans")                //only users with fan role or manager role should be able to access this
    public String seePlayers() {
        return "Fans can see the players only";
    }

    @GetMapping("/manager")             //only users with manager role should be able to access this
    public String manager() {
        return "Manager can see both players and staff";
    }

```

- Steps for configuring Authentication :
  - First we have to get hold of **AuthenticaionManagerBuilder** and then we have to configure it
    - What type of authentication using auth.inMemoryAuthentication()(Eg: memory)
    - Then tell the username, pwd and role and chain multiple using and()
    - Then finally we will get our own Authentication Manager
- Steps for configuring Authorization
  - We have to configure **HttpSecurity** first and configure it
    - We have to add each roles and specifc url with antMatchers using http.authorizeRequests()

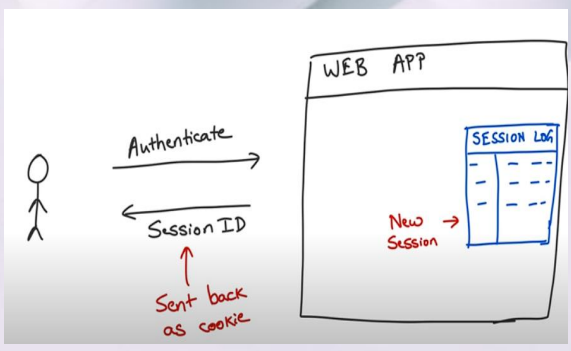


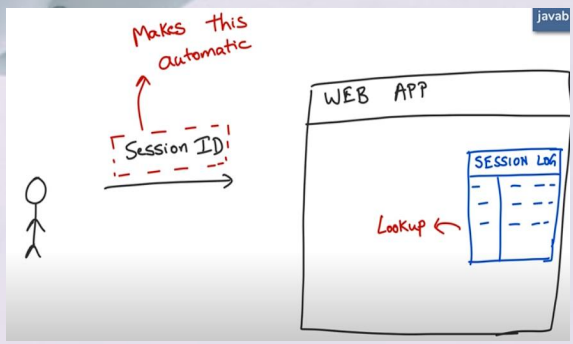
- Code wise:

```
@EnableWebSecurity
public class MySecurityConfiguration extends WebSecurityConfigurerAdapter {

    @Override //just type configure, tab, and override this method automatically
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        // set your authentication configuration on the 'auth' object

        auth.inMemoryAuthentication()
            .withUser("fasil")          //here you chain the users and give them role
            .password("fasil")          // but role wont be meaningful unless authority of that role
            .roles("Fan")               // is configured
            .and()
            .withUser("mourinho")
            .password("mourinho")
            .roles("Manager");
    }

    @Bean
    public PasswordEncoder getPasswordEncoder() {   // PasswordEncoder is needed to encode the password n run
        return NoOpPasswordEncoder.getInstance();   // not encoding password for now
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        // set your authorization configuration on the 'http' object
        http
            .authorizeRequests()                                //start from most sercure->>least secure role
            .antMatchers("/manager").hasRole("Manager")
            .antMatchers("/fans").hasAnyRole("Manager","Fan")
            .antMatchers("/").permitAll()                   //no authorization for /welcome
            .and().httpBasic();     //basic javascript type; postman easy; but no logout url
            //.formLogin();  //uses browser form login type; easier to logout; can't be used in postman
    }
```

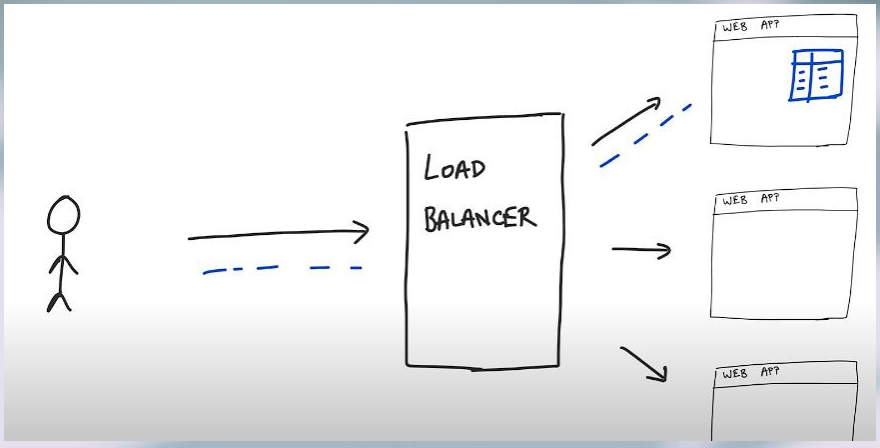
JWT (Json Web Tokens) Authentication

- There are mainly 2 types of Authorization strategies
  - Conventional Session token
  - JWT
- Conventional strategy uses cookies to store the session ID by a user and check it later against the session log when it passes along with the new request
- Session token is like a 'Reference token' , reference to the state of the server

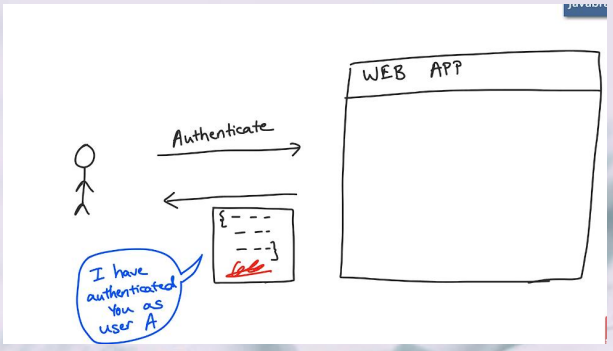


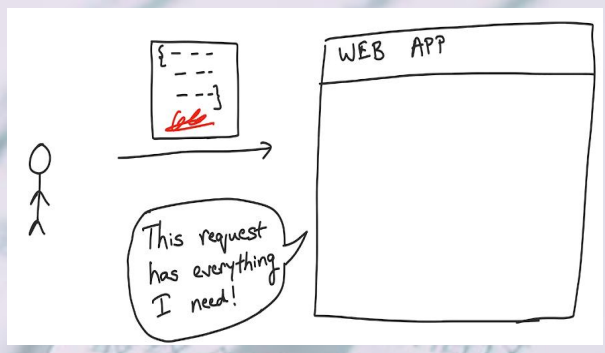


- However in case of microservices, 1 server might not have session id from another server and won't be able to check against their own session log so this will fail the authentication



- Now in **JWT**, while authentication instead of giving session id, the server returns a signed json token to the user and this signed JWT is verified in next request and authorization is validated. In short, server don't have to remember any info or store any session log. It is stateless token
- JWT is like a 'value token'





Steps to add JWT Authentication

1. Firstly, setup your spring boot project with all the right dependencies like spring boot starter security, jsonwebtoken etc

2. Create Authentication Controller
   a) Define a REST endpoint to authenticate users based on their username and password and generate JWT tokens upon successful authentication after going to the UserDetailsServiceImpl class(step3) and then to JWT utility classes(step4)

3. Create Custom UserDetailsServiceImpl class
    a) Implement the UserDetailsService interface to load user-specific data.
    b) Use hardcoded user details for simplicity in this example (typically, user details would be fetched from a database or an external source).
    c) Just check if the username & password are valid (u can set role authority now, but I guess it's not needed yet; add exception if no existing user is found)

4. Implement JWT Utility Classes
    a) Create a utility class 'JwtAuthenticationEntryPoint'
    b) Create a utility class to handle JWT creation and validation.
    c) This class will contain methods to generate tokens, extract claims, and validate tokens.
    d) While creating the token -
        i. Define claims of the token, like Issuer, Expiration, Subject, and the ID
        ii. Sign the JWT using the HS512 algorithm and secret key present in application.properties

5. Create JWT Request Filter class
    a) To Create a filter to intercept incoming requests from UserController (it also goes when AuthenticationController is called first time but since token hasn't been generated then, so no need to validate then)
    b) Ensure the filter extracts the token from the Authorization bearer token header and then do initial null validation or jwt token expiration date etc. Note: we will need secret key set in application.properties to retrieve any information from the jwt token
    c) It again calls the UserDetailsService class methods and check wrt database/hardcoded context if the token is valid and sets the authorities based on the role found

6. Configure Spring Security
    a) Finally in our SecurityConfiguration class which has annotation @EnableWebSecurity and which extends WebSecurityConfigurerAdapter class and where we override AuthenticationManagerBuilder and HttpSecurity configure() methods,we can classify which all apis can be accessed from this specific user's authority/role
    b) Add the JWT filter to the security filter chain.

7. Test the application
    a) Authenticate Users:
        i. Use the /authenticate endpoint to generate JWT tokens for authenticated users.
    b) Access Secured Endpoints:
        i. Use the generated JWT token in the Authorization header (Bearer <token>) to access secured endpoints.

Spring Cloud

- Spring Cloud is a powerful framework that equips developers with the tools necessary to build and manage microservices architectures.
- It enhances the capabilities of Spring Boot to create cloud-native applications that are scalable, resilient, and flexible, making it easier to manage complex distributed systems.

Component	Tools/Frameworks	Description	Example Use Cases	Benefits
Service Discovery	Eureka, Consul, ZooKeeper	Allows microservices to discover each other via a service registry.	Service registration and lookup	Dynamic service discovery and load balancing
Configuration Management	Spring Cloud Config	Externalizes configuration properties, allowing centralized management across environments.	Centralized configuration	Simplified configuration management, dynamic updates
Load Balancing	Ribbon, Spring Cloud LoadBalancer	Distributes requests across multiple instances of a service.	Load distribution	Improved performance and availability
Circuit Breaker	Hystrix, Resilience4j	Provides latency and fault tolerance to prevent cascading failures.	Fault tolerance, service resiliency	High availability, graceful handling of service failures

-

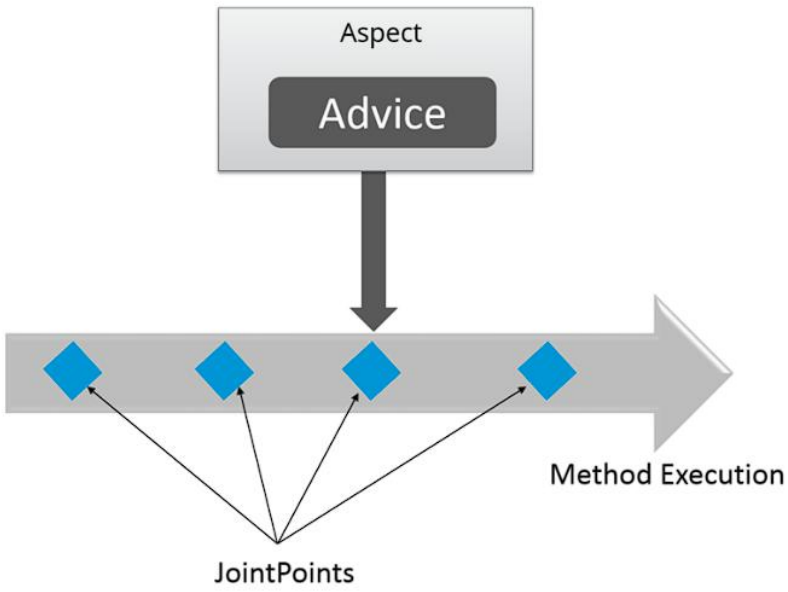
Routing and API Gateway	Zuul, Spring Cloud Gateway	Routes requests to appropriate microservices and provides monitoring, resiliency, and security.	API management, routing	Centralized routing, security, and monitoring
Distributed Tracing	Spring Cloud Sleuth, Zipkin	Tracks the flow of requests across microservices.	Debugging, performance monitoring	Enhanced observability, traceability
Messaging	Spring Cloud Stream	Builds message-driven microservices using abstractions for messaging systems.	Event-driven architectures	Decoupling of services, asynchronous communication
Security	Spring Cloud Security	Integrates with Spring Security for OAuth2 support and token relay.	Secure microservices	Simplified authentication and authorization
Centralized Logging	ELK (Elasticsearch, Logstash, Kibana), Graylog	Collects and analyzes logs f… multiple microservices.	Log aggregation, monitoring	Improved monitoring, easier troubleshooting

➢

Spring AOP (Aspect Oriented Programming)

- Spring AOP provides a way to separate cross-cutting concerns (like logging, security, and transaction management) from the business logic, improving modularity and simplifying code maintenance.
- It supports cohesive development by separating application business logic from system services.

An Action taken by an aspect at a particular joinpoint is known as an Advice. Spring AOP uses an advice as an interceptor, maintaining a chain of interceptors "around" the join point.



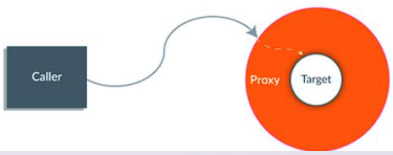
- Aspect: A module encapsulating a concern that cuts across multiple classes.
- Join Point: A point in the program execution, such as method execution.
- Advice: The action taken by an aspect at a join point (types: Before, After, After Returning, After Throwing, Around).
- Pointcut: A predicate that matches join points to determine where advice should be applied.
- Weaving: Linking aspects with other types or objects to create an advised object (can occur at compile-time, load-time, or runtime).

45. What do you mean by Proxy in Spring Framework?

An object which is created after applying advice to a target object is known as a Proxy. In case of client objects the target object and the proxy object are the same.

46. In Spring, what is Weaving?

The process of linking an aspect with other application types or objects to create an advised object is called Weaving. In Spring AOP, weaving is performed at runtime. Refer the below diagram:



Benefits

- **Separation of Concerns**: Keeps business logic and cross-cutting concerns separate.
- **Reusability**: Reusable aspects across different parts of the application.
- **Decoupling**: Decouples system services from business logic.
- **Ease of Maintenance**: Simplifies updates to cross-cutting concerns.

Use Cases

- **Logging**: Log method entry, exit, execution time, and exceptions.
- **Security**: Check permissions before method execution.
- **Transaction Management**: Manage transactions around method executions.
- **Caching**: Cache method results for better performance.

Spring AOP makes it easier to manage cross-cutting concerns, resulting in cleaner and more maintainable code.