1. **What is Dependency Injection in Spring, and how is it different from regular object creation using new? Explain in your own words with a real-world example.**

Dependency injection is a design pattern. Its main task is to inject dependency- means inject one object into another object.

If a class needs some objects(dependencies), these are provided from outside, instead of creating them in that class.

This ensures the code is easier to manage, easier to test, more flexible and less tightly connected. (has low coupling).

Achieved in two ways:  
1) Setter method DI

2) Constructor DI

Regular object creation instead of DI means hardcoding the values. It involves use of setters to inject the dependency. Described below:  
 Student st=new Student();

st.setAddress(new Address());

This makes the code less maintainable. It may also introduce bugs, if the dependency class structure is modified externally.

For example, a Student class can have a property ‘address’ of type Address. This property can be passed externally through the constructor Student() while creating an object, instead of creating it inside the constructor.

**Without DI:**

Class Student{

public Address address;

Student(){

this.address=new Address(“Ratnagiri”);

// Other property initialization  
 }  
}

OR   
 Student st=new Student();

st.setAddress(new Address());

**With Dependency injection:**

Class Student{

public Address address;

Student(Address address){

this.address=address;

// Other property initialization  
 }  
}

**Use as:**

Address add=new Address(“Pune”);  
 Student s=new Student(add);

1. **You have two beans of the same type. What will happen if you use @Autowired without @Qualifier or @Primary? Write a code snippet that would cause this problem.**

Spring will not know which bean to inject, and will throw an exception.

Code snippet causing the problem:

**Engine.java**  
package com.example.cardemo;

public interface Engine {

void start();

}

**Engine1.java**

package com.example.cardemo;

import org.springframework.stereotype.Component;

@Component("engine1") // Named bean "engine1"

public class Engine1 implements Engine {

@Override

public void start() {

System.out.println("Engine1 is starting...");

}

}

Engine2.java

package com.example.cardemo;

import org.springframework.stereotype.Component;

@Component("engine2") // Named bean "engine2"

public class Engine2 implements Engine {

@Override

public void start() {

System.out.println("Engine2 is starting...");

}

}

Car.java

package com.example.cardemo;

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

import org.springframework.stereotype.Component;

@Component

public class Car {

private final Engine engine;

// No @Qualifier or @Primary used here!

@Autowired

public Car(Engine engine) {

this.engine = engine;

}

public void drive() {

engine.start();

System.out.println("Car is driving...");

}

}

CardemoApplication.java

package com.example.cardemo;

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

import org.springframework.boot.CommandLineRunner;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class CardemoApplication implements CommandLineRunner {

private final Car car;

@Autowired

public CardemoApplication(Car car) {

this.car = car;

}

public static void main(String[] args) {

SpringApplication.run(CardemoApplication.class, args);

}

@Override

public void run(String... args) {

car.drive(); // This will fail due to multiple beans of type Engine

}

}

1. **Explain how the Spring IoC container works behind the scenes when your application starts. Mention key steps like bean scanning, creation, injection, etc.**

Spring IoC container creates objects (beans), manages their lifecycle, handles dependency injection between them. IoC in spring boot is powered by ApplicationContext.

When we run a spring boot app, spring boot initializes application context, which in the spring container.

Spring looks at the code, and searches for classes annotated with @Component, @Service , @Repository, @Controller, @Configuration.

The component scanning starts from the package, where @SpringBootApplication class is located. Spring creates bean definition for each class.

Then, spring goes through bean definition,and starts creating objects (beans). If bean has no dependency, Spring creates it using new, Else, spring first finds the dependencies,and injects into the constructor or fields.

Once bean is created, Spring handles its lifecycle. - May call initialization methods like @PostConstruct , keep track of singleton bean, @PreDestroy for cleanup .

Whenever bean needs something, Spring looks up container, and injects required dependency.

1. **You have a service class with multiple dependencies. Which injection method would you prefer (constructor, field, setter) and why? Justify your answer.**

As class has multiple dependencies, Constructor injection is preferred.

**Field injection** is hard to test. We can’t mark fields as final. It is less safe.

Example:  
@Component

public class MyService {

@Autowired

private UserRepository userRepository;

@Autowired

private EmailService emailService;

public void process() {

// use dependencies

}

}

Setter dependency injection method allows to change dependency after object creation. Object is not fully initialized after construction. It might break if setters are not called.

Example:  
@Component

public class MyService {

private UserRepository userRepository;

private EmailService emailService;

@Autowired

public void setUserRepository(UserRepository userRepository) {

this.userRepository = userRepository;

}

@Autowired

public void setEmailService(EmailService emailService) {

this.emailService = emailService;

}

}

Constructor injection is safer. It allows all required dependencies to be listed in one place. Great for testing.

Example:  
@Component

public class MyService {

private final UserRepository userRepository;

private final EmailService emailService;

@Autowired

public MyService(UserRepository userRepository, EmailService emailService) {

this.userRepository = userRepository;

this.emailService = emailService;

}

}

1. **What is the role of @Component, @Service, @Repository, and @Controller? Are they all the same? If not, how are they different?**

The annotation tell spring, to manage the class as a bean inside IoC container. All of annotations are detected at component scanning and registered as beans. But each one defines different roles for the class.

@Component is base annotation. It marks a class as spring managed bean. It doesn’t specify special role of class.

e.g. @Component

public class MyHelper {

public void help() {

System.out.println("Helping...");

}

}

@Service is specialization of @Component. It indicates that class contains business logic. Makes code more organized.

e.g @Service

public class UserService {

public void createUser() {

System.out.println("Creating user...");

}

}

@Repository is specialization of @Component. It marks class as DAO(Data Access Object) used for database interaction. It automatically transates exceptions into Springs’s DataAccessException. i.e. If SQLException occurs, spring wil convert it to more generic Spring exception.

e.g. @Repository

public class UserRepository {

public void save(User user) {

// Insert user into DB

}

}

@Controller is a specialization of @Component. It defines Spring MVC Controller. It handles web requests, maps URLs to methods.

e.g. @Controller

public class UserController {

@GetMapping("/hello")

public String helloPage() {

return "hello";

}

}

1. **Write a small Spring Boot configuration class that defines a custom bean of type RestTemplate using @Bean. (No need to import everything – focus on core logic.)**

@Configuration

public class AppConfig {

@Bean

public RestTemplate restTemplate() {

return new RestTemplate();

}

}

@Configuration tells Spring that this class contains bean definitions.

@Bean tells Spring to create and manage this object as a bean in the IoC container.

restTemplate() – this method returns an instance of RestTemplate, and Spring will automatically register it.

It can be injected in app as:  
@Service

public class ApiService {

private final RestTemplate restTemplate;

@Autowired

public ApiService(RestTemplate restTemplate) {

this.restTemplate = restTemplate;

}

public String getDataFromApi() {

String url = "https://api.example.com/data";

return restTemplate.getForObject(url, String.class);

}

}

1. **What would happen if you forget to annotate your custom service class with @Component or any stereotype annotation? Explain and give a short code example to demonstrate.**

In this case, Spring will not know about our class. So it will not create bean for it. Dependency injection will fail.

Spring only manages beans that are annotated with @Component or its subtypes or are manually registered as @Bean in a @Configuration class.

Un-annotated class is ignored during component scanning.

Example:

// @Service annotation Is missing  
public class MyService {

public void doSomething() {

System.out.println("Doing something important...");

}

}

@Component

public class MyController {

private final MyService myService;

@Autowired

public MyController(MyService myService) {

this.myService = myService;

}

public void run() {

myService.doSomething();

}

}

1. **In application.properties, you define app.env=dev. How can you use this to conditionally load a bean only in dev environment? Write a short code using @Profile.**

TO make @Profile(“Dev”) work, we need to add following in application.properties:  
spring.profiles.active=dev

This tells spring, to run app in “dev” profile.

Then define two beans with different profiles:  
public interface EnvService {

void printEnv();

}

---------------------------------------------------

@Profile("dev")

@Component

public class DevEnvService implements EnvService {

public void printEnv() {

System.out.println("Running in DEV environment");

}

}

----------------------------------------------

@Profile("prod")

@Component

public class ProdEnvService implements EnvService {

public void printEnv() {

System.out.println("Running in PROD environment");

}

}

Only **one** of these beans will be created depending on the active profile.

Then, inject the bean in application

@Component

public class AppRunner implements CommandLineRunner {

private final EnvService envService;

@Autowired

public AppRunner(EnvService envService) {

this.envService = envService;

}

@Override

public void run(String... args) {

envService.printEnv();

}

}

1. **What’s the difference between @ComponentScan and @EnableAutoConfiguration in Spring Boot? Explain with one practical use case where both are useful.**

@ComponentScan scans the code for Spring beans. It tells spring, where to look for the annotated classes, to register as beans

If we put @ComponentScan on Main Application class, spring will scan that package and all subpackages.

e.g @ComponentScan(basePackages = "com.example")

public class AppConfig { }

@EnableAutoConfiguration tells spring boot to automatically configure beans for libraries on classpath, based on content of pom.xml.

It is used to auto configure third party stuff.

In spring boot, we don’t write them separately, instead use:

@SpringBootApplication

public class DemoApplication {

public static void main(String[] args) {

SpringApplication.run(DemoApplication.class, args);

}

}

It is shortcut for:

@Configuration

@ComponentScan

@EnableAutoConfiguration

public class DemoApplication { }

The practical usecase where both are useful is a simple Spring boot REST API with custom service layer

We need to define following:

@ComponentScan to find custom beans:

@RestController for handling HTTP

@Service for business logic

@Repository for data access

@EnableAutoConfiguration to configure Embedded Tomcat (Spring Web)