**HM#2** **Doğukan Bayrak**

## 1 – IOC and DI means?

IoC (Inversion Of Control) is a software development principle that aims to create objects that are loose coupling throughout the lifecycle of the application. It is responsible for the life cycle of objects, provides their management.

Dependency injection is basically providing the objects that an object needs (its dependencies) instead of having it construct them itself. It's a very useful technique for testing, since it allows dependencies to be mocked or stubbed out. Dependencies can be injected into objects by many means (such as constructor injection or setter injection). One can even use specialized dependency injection frameworks (e.g. Spring) to do that, but they certainly aren't required. You don't need those frameworks to have dependency injection. Instantiating and passing objects (dependencies) explicitly is just as good an injection as injection by framework.

## 2 – Spring Bean Scopes?

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. Otherwise, a bean is simply one of many objects in your application. Beans, and the dependencies among them, are reflected in the configuration metadata used by a container.

- **Singleton:** It returns a single bean instance per Spring IoC container. This single instance is stored in a cache of such singleton beans, and all subsequent requests and references for that named bean return the cached object. If no bean scope is specified in bean configuration file, **default** to singleton.
- **Prototype**: It returns a new bean instance each time when requested. It does not store any cache version like singleton.
- **Request**: It returns a single bean instance per HTTP request.
- **Session:** It returns a single bean instance per HTTP session (User level session).
- **GlobalSession:** It returns a single bean instance per global HTTP session. It is only valid in the context of a web-aware Spring ApplicationContext (Application level session).

## 3 – What does @SpringBootApplication do ?

Spring Boot @SpringBootApplication annotation is used to mark a configuration class that declares one or more @Bean methods and also triggers auto-configuration and component scanning. It’s same as declaring a class with @Configuration, @EnableAutoConfiguration and @ComponentScan annotations.

**4 – What is Spring AOP? Where and How to use it?**

AOP is an approach that reduces the complexity of the software and increases the modularity. What is meant by modularity here is the division of non-functional code, that is, intersecting interests, used in many parts of the system during the application (Seperation of Cross Cutting Concerns). In this way, the structures to be used throughout the application are abstracted from the system and encapsulated, allowing them to be used in many places. In general, it can be said that AOP helps to improve the existing system rather than solving a problem.

**5 – What is Singleton and where to use it?**

Singleton pattern is a design pattern which restricts a class to instantiate its multiple objects. It is nothing but a way of defining a class. Class is defined in such a way that only one instance of the class is created in the complete execution of a program or project. It is used where only a single instance of a class is required to control the action throughout the execution. A singleton class shouldn't have multiple instances in any case and at any cost. Singleton classes are used for logging, driver objects, caching and thread pool, database connections.

**6 – What is Spring Boot Actuator and Where to use it?**

Spring Boot Actuator is a sub-project of the Spring Boot Framework. It includes a number of additional features that help us to monitor and manage the Spring Boot application. It contains the actuator endpoints (the place where the resources live). We can use HTTP and JMX endpoints to manage and monitor the Spring Boot application. If we want to get production-ready features in an application, we should use the Spring Boot actuator.

**7 -What is the primary difference between Spring and Spring Boot?**

- Spring is an open-source lightweight framework widely used to develop enterprise applications. Spring Boot is built on top of the conventional spring framework, widely used to develop REST APIs.
- The most important feature of the Spring Framework is dependency injection. The most important feature of the Spring Boot is Autoconfiguration.
- It helps to create a loosely coupled application. It helps to create a stand-alone application.
- To run the Spring application, we need to set the server explicitly. Spring Boot provides embedded servers such as Tomcat and Jetty etc.
- To run the Spring application, a deployment descriptor is required. There is no requirement for a deployment descriptor.
- To create a Spring application, the developers write lots of code. It reduces the lines of code.
- It doesn't provide support for the in-memory database. It provides support for the in-memory database such as H2.

**8 – Why to use VCS?**

Version control is important to keep track of changes and keep every team member working on the right version. You should use version control software for all code, files, and assets that multiple team members will collaborate on. It needs to do more than just manage and track files. It should help you develop and ship products faster. This is especially important for teams practicing DevOps. That's because using the right one:

- Improves visibility.
- Helps teams collaborate around the world.
- Accelerates product delivery.

**9 – What are SOLID Principles? Give sample usages in Java?**

SOLID refers to five design principles in object-oriented programming, designed to reduce code rot and improve the value, function, and maintainability of software. The SOLID principles help the user develop less coupled code. If code is tightly coupled, a group of classes are dependent on one another. This should be avoided for better maintainability and readability. SOLID design is an acronym for the following five principles:

- Single Responsibility Principle
- Open-Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

### 9.1.Single responsibility principle

Every class in Java should have a single job to do. To be precise, there should only be one reason to change a class. Here's an example of a Java class that does not follow the single responsibility principle (SRP):

```java
public class Vehicle {
    public void printDetails() {}
    public double calculateValue() {}
    public void addVehicleToDB() {}
}
```

The Vehicle class has three separate responsibilities: reporting, calculation, and database. By applying SRP, we can separate the above class into three classes with separate responsibilities.

### 9.2. Open-closed principle

Software entities (e.g., classes, modules, functions) should be *open* for an extension, but *closed* for modification.

Consider the below method of the class VehicleCalculations:

```java
public class VehicleCalculations {
    public double calculateValue(Vehicle v) {
        if (v instanceof Car) {
            return v.getValue() * 0.8;
        if (v instanceof Bike) {
            return v.getValue() * 0.5;

    }
}
```

Suppose we now want to add another subclass called Truck. We would have to modify the above class by adding another if statement, which goes against the Open-Closed Principle. A better approach would be for the subclasses Car and Truck to override the calculateValue method:

```java
public class Vehicle {
    public double calculateValue() {...}
}
public class Car extends Vehicle {
    public double calculateValue() {
        return this.getValue() * 0.8;
}
public class Truck extends Vehicle{
    public double calculateValue() {
        return this.getValue() * 0.9;
}
```

Adding another Vehicle type is as simple as making another subclass and extending from the Vehicle class.

### 9.3. Liskov substitution principle

**(LSP)** applies to inheritance hierarchies such that derived classes must be completely substitutable for their base classes.

Consider a typical example of a Square derived class and Rectangle base class:

```java
public class Rectangle {
    private double height;
    private double width;
    public void setHeight(double h) { height = h; }
    public void setWidht(double w) { width = w; }
    ...
}
public class Square extends Rectangle {
    public void setHeight(double h) {
        super.setHeight(h);
        super.setWidth(h);
    }
    public void setWidth(double w) {
        super.setHeight(w);
        super.setWidth(w);
    }
```

The above classes do not obey LSP because you cannot replace the Rectangle base class with its derived class Square. The Square class has extra constraints, i.e., the height and width must be the same. Therefore, substituting Rectangle with Square class may result in unexpected behavior.

### 9.4. Interface segregation principle

The **Interface Segregation Principle (ISP)** states that clients should not be forced to depend upon interface members they do not use. In other words, do not force any client to implement an interface that is irrelevant to them.

Suppose there's an interface for vehicle and a Bike class:

```
public interface Vehicle {
    public void drive();
    public void stop();
    public void refuel();
    public void openDoors();
}
public class Bike implements Vehicle {

    // Can be implemented
    public void drive() {...}
    public void stop() {...}
    public void refuel() {...}

    // Can not be implemented
    public void openDoors() {...}
}
```

As you can see, it does not make sense for a Bike class to implement the openDoors() method as a bike does not have any doors! To fix this, ISP proposes that the interfaces be broken down into multiple, small cohesive interfaces so that no class is forced to implement any interface, and therefore methods, that it does not need.

### 9.5. Dependency inversion principle

The **Dependency Inversion Principle (DIP)** states that we should depend on abstractions (interfaces and abstract classes) instead of concrete implementations (classes). The abstractions should not depend on details; instead, the details should depend on abstractions. Consider the example below. We have a Car class that depends on the concrete Engine class; therefore, it is not obeying DIP.

```
public class Car {
    private Engine engine;
    public Car(Engine e) {
        engine = e;      }
    public void start() {
        engine.start();
    } }
public class Engine {
   public void start() {...}
}
```

The code will work, for now, but what if we wanted to add another engine type, let's say a diesel engine? This will require refactoring the Car class. However, we can solve this by introducing a layer of abstraction. Instead of Car depending directly on Engine, let's add an interface:

```
public interface EngineInterface {
    public void start();
}
```

Now we can connect any type of Engine that implements the Engine interface to the Car class:

```
public class Car {
    private EngineInterface engine;
    public Car(EngineInterface e) {
        engine = e;
    }
    public void start() {
        engine.start();
    }
}
public class PetrolEngine implements EngineInterface {
   public void start() {...}
}
public class DieselEngine implements EngineInterface {
   public void start() {...}
```

**10 - What is RAD model?**

RAD Model or Rapid Application Development model is a software development process based on prototyping without any specific planning. In RAD model, there is less attention paid to the planning and more priority is given to the development tasks. It targets at developing software in a short span of time.

- Business Modeling
- Data Modeling
- Process Modeling
- Application Generation
- Testing and Turnover

**11 - What is Spring Boot starter? How is it useful?**

Before Spring Boot was introduced, Spring Developers used to spend a lot of time on Dependency management. Spring Boot Starters were introduced to solve this problem so that the developers can spend more time on actual code than Dependencies. Spring Boot Starters are dependency descriptors that can be added under the <dependencies> section in pom.xml. There are around 50 Spring Boot Starters for different Spring and related technologies. These starters give all the dependencies under a single name. For example, if you want to use Spring Data JPA for database access, you can include spring-boot-starter-data-jpa dependency.

The advantages of using Starters are as follows:

- Increase productivity by decreasing the Configuration time for developers.
- Managing the POM is easier since the number of dependencies to be added is decreased.
- Tested, Production-ready, and supported dependency configurations.
- No need to remember the name and version of the dependencies.

**12 – What is Caching? How can we achieve caching in Spring Boot?**

Caching is a part of temporary memory (RAM). It lies between the application and persistence database. It stores the recently used data that reduces the number of database hits as much as possible. In other words, caching is to store data for future reference. The primary reason for using cache is to make data access faster and less expensive. When the highly requested resource is requested multiple times, it is often beneficial for the developer to cache resources so that it can give responses quickly. Using cache in an application enhances the performance of the application. Data access from memory is always faster in comparison to fetching data from the database. It reduces both monetary cost and opportunity cost.

We can achieve caching in Spring Boot if we use the annotation which is @EnableCaching. It is defined in org.springframework.cache.annotation package. It is used together with @Configuration class.

**13 – What & How & Where & Why to logging?**

Data logging is the process of collecting and storing data over a period of time in different systems or environments. It involves tracking a variety of events. Put simply, it is collecting data about a specific, measurable topic or topics, regardless of the method used. While data logging generally is associated with devices, even looking at a thermometer daily at a set time and writing down the temperature on a piece of paper with a pen is a rudimentary way to "log data". In other words, it's methodical collection of information.

**14 - What is Swagger? Have you implemented it using Spring Boot?**

Swagger allows you to describe the structure of your APIs so that machines can read them. The ability of APIs to describe their own structure is the root of all awesomeness in Swagger. Why is it so great? Well, by reading your API's structure, we can automatically build beautiful and interactive API documentation. We can also automatically generate client libraries for your API in many languages and explore other possibilities like automated testing.