1 – IOC and DI means ?

IOC ve DI anlamı?

IoC(Inversion Of Control), uygulamanın yaşam döngüsü boyunca birbirine bağımlılığı az (loose coupling) olan nesneler oluşturmayı amaçlayan bir yazılım geliştirme prensibidir. Nesnelerin yaşam döngüsünden sorumludur, yönetimini sağlar. IoC kullanan sınıfa bir interface inject edildiğinde, ilgili interface metotları kullanılabilir olur. Böylece IoC kullanan sınıf sadece kullanacağı metotları bilir, sınıf içerisinde daha fazla metot olsa bile interface'de belirtilen metotlara erişebilecektir.

Sınıf içerisinde yapılacak herhangi bir değişiklikte IoC kullanan sınıf etkilenmeyeceği için yeniden yazılabilir ve test edilebilir yazılım geliştirmemizi sağlar. IoC nesne bağlamalar genellikle uygulama başlangıcında yapılandırılmaktadır. Bu anlamda tek bir yerden yapılan IoC yapılandırmalarının değiştirilmesi ve yönetimi de oldukça kolaydır.

Dependency injection bir sınıfın/nesnenin bağımlılıklardan kurtulmasını amaçlayan ve o nesneyi olabildiğince bağımsızlaştıran bir programlama tekniği/prensibidir.

Dependency Injection uygulayarak; bir sınıfının bağımlı olduğu nesneden bağımsız hareket edebilmesini sağlayabilir ve kod üzerinde olası geliştirmelere karşın değişiklik yapma ihtiyacını ortadan kaldırabilirsiniz.

2 – Spring Bean Scopes ?

Bir bean'in context'i, kullandığımız bağlamlarda o bean'in yaşam döngüsünü ve oluşumunu tanımlar. Spring framework çatısında 6 adet bean scope bulunmaktadır:

- Singleton

Bir bean default olarak singleton scope'a sahiptir. Bean singleton scope ile tanımlandığı zaman mevcut application context'imiz içerisinde o bean'den yalnızca ve yalnızca tek bir adet initialize edileceğini garanti ederiz. Bu bean ile yapılacak olan tüm request'ler cache'lenmiş olan aynı nesne üzerinden yapılır. Bean nesnemiz üzerinde yapılan bir değişiklik bean'i kullanan tüm yerlerde etkilecektir.

- Prototype

Prototype ile belirlenmiş bir bean, container içerisinde çağırıldığı her request'te yeniden oluşturulacaktır.

Yalnızca web uyumlu uygulamalar ile kullanılabilen scope'lar ise:

- Request

Request scope değeri her bir http request'i bir adet bean oluştururur.

- Session

Session scope değeri her bir http session'ı için bir adet bean oluşturur.

- Application

Bir application scope, ServletContext'in yaşam döngüsü için bean örneğini oluşturur. Bu singleton scope'a benzer ancak aralarında farklılıklar mevcuttur. Bir bean application scope değerine sahipken bu bean çoklu servlet tabanlı uygulamalar ile de paylaşılabilirken, singleton scope değerine sahip bir bean yalnızca mevcut application context'i içerisinde tanımlıdır.

- Websocket

WebSocket scope'a sahip bean'ler tipik olarak singleton scope yapısındadır ve herhangi bir WebSocket oturumundan daha uzun yaşar. Bu nedenle, WebSocket scope'una sahip bean'ler için bir proxyMode tanımı gerekir. Singleton davranış sergilediğini, ancak yalnızca bir WebSocket sesion'ı ile sınırlı olduğunu da söyleyebiliriz.

3 – What does @SpringBootApplication do ?

@SpringBootApplication ne yapar?

@SpringBootApplication anotasyonu uygulamanın giriş metodunu belirtir. Yani halk arasındaki tabir ile main fonksiyondur. Uygulama bu metod ile başlar.

4 – Why Spring Boot over Spring?

Neden Spring Boot Spring'e tercih edilir?

Spring kütüphanesi bize esneklik uygulamaya odaklanırken, Spring Boot kod uzunluğunu kısaltmayı ve bir web uygulaması geliştirmenin en kolay yolunu bize sunmayı amaçlamaktadır. Spring Boot, uygulama geliştirme için gerekli olan süreyi bir hayli kısaltır. Neredeyse hiçbir konfigürasyon yapmadan tek başına bir uygulama oluşturulmasına yardımcı olur.

Spring Boot bir kütüphane olmayıp, Spring tabanlı hazır bir proje başlatıcıdır. Otomatik yapılandırma gibi özelliklerle sizi uzun kod yazmaktan kurtarır ve gereksiz yapılandırmalardan kurtulmanızı sağlar.

5 – What is Singleton and where to use it ?

Singleton nedir ve nerede kullanılır?

Spring, nesne oluştururken varsayılan olarak singleton yapısı ile nesneleri oluşturur. Yani her sınıftan yalnızca bir tane nesne oluşturur.

Bu tasarım örüntüsündeki amaç, bir class'tan sadece bir instance yaratılmasını sağlar. Yani herhangi bir class'tan bir instance yaratılmak istendiğinde, eğer daha önce yaratılmış bir instance yoksa yeni yaratılır. Daha önce yaratılmış ise var olan instance kullanılır.

6 – Explain @RestController annotation in Sprint boot?

Spring Boot @RestController anotasyonunu açıklayın.

Temel amacı adından da anladığımız gibi REST endpoint'leri üretmeyi sağlamak. Aslında bunun bize sağladığı şey @Controller ve @ResponseBody notasyonlarını tek seferde vermesi. Bu sayede REST controller olmasını isteyeceğimiz her metoda @ResponseBody yazmaktan da kurtulmuş oluyoruz.

RestController @Controller ve@ ResponseBody'nin birleşiminden oluşur. Dolaylı olarak da @Controller üzerinden @Component notasyonunu da barındırıyor. @Controller notasyonunun aksine @RestController datanın kendisini JSON veya XML formatı ile direkt olarak sunabiliyor.

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

Spring ve Spring Boot arasındaki temel fark nedir?

Spring Boot, geleneksel Spring çerçevesinin üzerine inşa edilmiştir. Bu nedenle, Spring'in tüm özelliklerini sağlar ve kullanımı Spring'den daha kolaydır. Spring Boot, mikroservice tabanlı bir çerçevedir ve çok daha kısa sürede üretime hazır bir uygulama yapar. Spring Boot'da her şey otomatik olarak yapılandırılır. Belirli bir işlevi kullanmak için sadece uygun konfigürasyonu kullanmamız gerekiyor. Spring Boot, Tomcat ve Jetty gibi gömülü server sağlar. REST API geliştirmek istiyorsak Spring Boot çok kullanışlıdır.

8 – Why to use VCS ?

VCS neden kullanılır?

- Her dosyanın tam bir uzun vadeli değişiklik geçmişi tutulur. Buda, dosya üzerinde yıllar içinde birçok kişi tarafından yapılan her değişikliğin tutulması anlamına gelir.
- Ekip üyelerinin eşzamanlı olarak aynı kod üzerinde çalışmasına imkan verir. Alt sürümler oluşturarak yazılım üzerinde farklı çalışmaları yürütüp, sonrasında ana yazılıma bunu entegre etmek mümkündür.
- Yazılımda yapılan her değişikliği takip edip proje yönetimine bağlayabilme imkanı sağlar. Yazılım üzerindeki sorunların, sürümler ile ilişkilendirilebilmesine ve takip edilebilmesine olanak sağlar.

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

SOLID prensipleri nedir. Javadan örnek kullanımlar gösterin.

1. Single Responsibility Principle

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

Java'da her bir classın tek bir sorumluluğu olması gerekir. Yukarıdaki örnekte verilen class Single ResponsibilVehicle sınıfının üç ayrı sorumluluğu vardır. SRP uygulayarak yukarıdaki sınıfı ayrı sorumluluklarla üç sınıfa ayırabiliriz.

2. Open/Closed Principle

```
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;

    }
}
```

Truck adında başka bir alt sınıf eklemek istediğimizi varsayalım. Open-Closed İlkesine aykırı olan başka bir if ifadesi ekleyerek yukarıdaki sınıfı değiştirmemiz gerekir. Car ve Truck alt sınıflarının calculateValue yöntemini override etmesi daha iyi bir yaklaşım olacaktır:

```
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;
}
```

3. Liskov ‘s Substitution Principle

```
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);
    }
}
```

Yukarıdaki sınıflar LSP'ye uymaz çünkü Rectangle temel sınıfını türetilmiş Square sınıfıyla değiştiremezsiniz. Square sınıfının ekstra kısıtlamaları vardır, yani "yükseklik" ve "genişlik"

aynı olmalıdır. Bu nedenle, Rectangle sınıfının Square sınıfıyla değiştirilmesi beklenmeyen davranışlara neden olabilir.

4. Interface Segregation Principle

ISP, clientlerin kullanmadıkları interface üyelerine bağımlı olmaya zorlanmaması gerektiğini belirtir. Başka bir deyişle, herhangi bir clienti, kendileriyle ilgisi olmayan bir interface uygulamaya çalışılmamalıdır.

```
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() {...}
}
```

Bike sınıfının openDoors() methodunu implemente etmesi bir bisikletin kapısı olmadığı için mantıklı değildir. Bunu düzeltmek için ISP, interfacelerin çoklu, küçük uyumlu interfacelere ayrılmasını önerir. Böylece hiçbir sınıf herhangi bir interface' i ve dolayısıyla ihtiyaç duymadığı methodları uygulamak zorunda kalmaz.

5. Dependency Inversion Principle

DIP, somut uygulamalar (sınıflar) yerine abstractionlara (interface ve abstract classlar) güvenmemiz gerektiğini belirtir. Soyutlamalar ayrıntılara bağlı olmamalıdır; bunun yerine, ayrıntılar soyutlamalara dayanmalıdır. Aşağıdaki örneği düşünün. Somut Engine sınıfına bağlı olan bir Car sınıfımız var; bu nedenle, DIP'ye uymuyor.

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

Kod şimdilik işe yarayabilir ama bir motor tipi eklemek istersek bu Car sınıfının yeniden düzenlenmesini gerektirecektir. Ancak, bunu bir abstraction katmanı ekleyerek çözebiliriz. Doğrudan Engine'e bağlı Car yerine bir arayüz ekleyelim:

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

Artık Engine interface'ini uygulayan herhangi bir Engine türünü Car sınıfına bağlayabiliriz.

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

10 - What is RAD model ?

RAD Model nedir?

RAD(Rapid Application Development) yani **Hızlı Uygulama Geliştirme** bir yazılım geliştirme yöntemidir. Çok fazla detaya girilmeden, hızlı şekilde çalışan bir uygulama oluşturma amacıyla benimsenen bu yöntem için kullanılan birçok araç ve kütüphane bulunmaktadır.

RAD Metodolojisi ne zaman kullanılır?

- Kısa sürede (2-3 ay) bir sistem üretilmesi gerektiğinde
- Gereksinimler bilindiğinde
- Kullanıcı tüm yaşam döngüsü boyunca dahil olduğunda
- Teknik risk daha az olduğunda
- 2-3 ay gibi kısa bir sürede modüler hale getirilebilen bir sistem oluşturma zorunluluğu olduğunda
- Bir bütçe, kod oluşturma için otomatik araçların maliyeti ile birlikte modelleme için tasarımcıları karşılayacak kadar yüksek olduğunda

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

Spring Boot starter nedir? Nasıl kullanışlı olur?

Spring Boot tanıtılmadan önce Spring Developerları, dependency yönetimine çok zaman harcıyordu. Spring Boot Starters, bu sorunu çözmek için geliştirildi. Böylece geliştiriciler, bağımlılıklardan ziyade gerçek kod üzerinde daha fazla zaman harcayabilir.

Spring Boot Starters, pom.xml'deki **<dependencies>** bölümünün altına eklenebilen bağımlılık tanımlayıcılarıdır. Farklı Spring ve ilgili teknolojiler için yaklaşık 50 Spring Boot Starter vardır. Bu starterlar, tüm bağımlılıkları tek bir ad altında verir. Örneğin, veritabanı erişimi için Spring Data JPA'yı kullanmak istiyorsanız, spring-boot-starter-data-jpa bağımlılığını dahil edebilirsiniz.

12 – What are the Spring Boot Annotations?

Spring Boot Annotationları nelerdir?

@Bean - Bir metodun Spring tarafından yönetilen bir Bean ürettiğini belirtir

@Service - Belirtilen sınıfın bir servis sınıfı olduğunu belirtir.

@Repository - Veritabanı işlemlerini gerçekleştirme yeteneği olan yapıldığı repository sınıfını belirtir.

@Configuration - Bean tanımlamaları gibi tanımlamalar için bir Bean sınıfı olduğunu belirtir

@Controller - Requestleri yakalayabilme yeteneği olan bir web controller sınıfını belirtir.

@RequestMapping - controller sınıfının handle ettiği HTTP Requestlerin path eşleştirmesini yapar

@Autowired - Constructor, Değişken yada setter metodlar için dependency injection işlemi gerçekleştirir

@SpringBootApplication - Spring Boot autoconfiguration ve component taramasını aktif eder.

13 – What is Spring Boot dependency management?

Spring Boot dependency management nedir?

Dependency Management, gerekli tüm bağımlılıkları tek bir yerde yönetmenin ve bunlardan verimli bir şekilde yararlanmanın bir yoludur.

Dependency listesi, Maven ile kullanılabilecek Bills of Materials (spring-boot-dependencies) bir parçası olarak mevcuttur.

14 - What is Spring Boot Actuator?

Spring Boot Actuator nedir?

Spring Boot Actuator, uygulamaların production ortamına hazır özellikleri (health check, disk usage, heap dump vs.) otomatik aktifleştirir ve farklı HTTP endpoint' ler ile etkileşimde bulunmayı sağlayan bir yapı sunar.

Spring Boot Actuator’ ün projede aktif olması istenirse aşağıdaki Maven dependency bloğunun pom dosyasına eklenmesi gerekir.

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