1. IOC and DI means ?

Inversion of control bir yazılım tasarım prensibidir. *Ioc* ile Uygulama içerisindeki obje instance'larının yönetimi sağlanarak, bağımlılıklarını en aza indirgemek amaçlanmaktadır. Projeniz deki bağımlılıkların oluşturulmasını ve yönetilmesini geliştiricinin yerine, framework'ün yapması olarak da açıklanabilir.

Yazdığımız kod bloğu çalışacağı zaman, framework bizim kodumuzu çağırır ve çalıştırır daha sonra kontrol yeniden framework'e geçmesi olayının tümüne Inversion Of Control adı verilmektedir.

Dependency Injection'ı özetle anlatmak gerekirse; bağımlılık oluşturacak parçaların ayrılıp, bunların dışardan verilmesiyle sistem içerisindeki bağımlılığı minimize etme işlemidir.

Temel olarak oluşturacağınız bir sınıf içerisinde başka bir sınıfın nesnesini kullanacaksanız new anahtar sözcüğüyle oluşturmamanız gerektiğini söyleyen bir yaklaşımdır. Gereken nesnenin ya Constructor'dan ya da Setter metoduyla parametre olarak alınması gerektiğini vurgulamaktadır. Böylece iki sınıfı birbirinden izole etmiş olduğumuzu savunmaktadır

Yazılımı oluşturan yapılar kaçınılmaz olarak birbirleri ile ilişkilidir. Lakin bu ilişkinin bir bağa ve sınırlandırmaya sebep olmaması için mümkün mertebe ilişkiyi gevşek tutmak önemlidir. Biz buna Loosely Coupled yani Gevşek Bağlılık diyoruz.

2. Spring Bean Scopes ?

Beanlerimizin bir yaşam döngsü vardır. Bu yaşam döngüsü çerçevesinde istediğimiz işlemleri yapması için Beanimizin kapsamını yani scope'unu belirlememiz gerekmektedir. Spring Beanlerimizdeki scopeleri Spring IoC container tarafından yönetilir ve beanlerimizdeki nesnelerin ne zaman ve nasıl oluşturulacağını belirler.

Spring'e oluşturduğumuz beanlerin Scope'lerini belirterek yönetmemiz ve Spring'in bu belirtmemize göre oluşturmasını sağlamaktayız. Bu scope göre Beanimizin kullanım alanını bir bakıma belirtmiş olmaktayız.

Scope Çeşitleri

singleton

Varsayılan olarak her bean Singleton'dur. Bu Bean'den sadece bir tane üretilir.

prototype

Bean'e istek geldiğinde oluşturulur. Her istekte farklı bir instance oluşturulur.

request

Web uygulamaları için kullanılır. Her HTTP isteği geldiğinde instance oluşturulur.

session

Web uygulamaları için kullanılır. Her HTTP session oluştuğunda instance oluşturulur.

globalSession

Web uygulamaları için kullanılır. Her HTTP isteği geldiğinde sadece bir tane instance oluşturulur.

3. What does @SpringBootApplication do ?

Bu anotasyon @Configuration, @EnbaleAutoConfiguration, @ComponentScan anotasyonlarının üçünü de içeren temel bir anotasyondur.

@Configuration

Configuration anotasyonu bulunduğu sınıfın Spring için bir konfigürasyon sınıfı olduğunu, bu sınıf içinde bir veya birden fazla bean tanımlaması yapılabileceğini belirtir

@Bean tanımlaması içeren fonksiyonlar içerir

Tanımlandığı fonksiyon her neyi return ediyorsa, onun context içine ekleneceğini belirtir.

Runtime anında config olarak tanımlanan sınıf içindeki @Bean anotasyonlarının, spring container tarafından işlenebilmesi için belirtilir.

@EnableAutoConfiguration

Uygulamaya eklenen Jar dosyalarının otomatik olarak konfigüre edilmesini sağlar. Örneğin, projeniz için herhangi bir veritabanı konfigürasyonu yapmadıysanız ve sadece HSQL veritabanını uygulamanıza import ettiyseniz, bu durumda @EnableAutoConfiguration sizin için otomatik olarak in-memory bir veritabanı oluşturup kullanımınıza sunacaktır.

@EnableAutoConfiguration, module bazlı class ve classpath'leri config yapar.

Örneğin, projeye bir veritabanı import edilmişse, database konfigürasyonlarının otomatik gerçekleşmesi için JdbcTemplate bean'i otomatik oluşturulur.

@ComponentScan

@ComponentScan, proje içerisinde bean olarak tanımlanan sınıfları tarar.

@Configuration içeren tüm Spring sınıflarını otomatik olarak tarar.

@Configuration ile kullanılan @Import(OtherConfiguration.class) tanımlamasının alternatifi olarak kullanılabilir.

@Repository, @Service, @Configuration ve @Controller anotasyonlarının tamamı @Component anotasyonu olduğundan, bu anotasyonları içeren sınıfları da otomatik olarak tarayıp Spring Application Context içine ekler.

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

Birbirinden farklı işlevler yürüten ancak kesişen davranışların birbirlerinden ayrışması metodolojisine AOP denilmektedir.

- Loglama
- Transaction Management
- Pooling
- Security → Güvenlik
- Caching → Kaynak Yönetimi
- Performans Ölçümü → kaynak yönetimi
- Exception Handling → Hata yönetimi
- Operations → İş kuralları

AOP, yukarıda sıralanan birçok işlevin ayrı ayrı tasarlanarak sadece gerektiğinde uygulama ile ahenk içinde çalıştırılabilmesi yöntemidir. Burada her uygulama birimi, Aspect olarak tasarlanmış herhangi bir modülü istediği zaman edinip kullanabilir.

Spring'in AOP bağımlılığını pom.xml'ye ekleyerek başlayalım:

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

```
public Ship saveShip(Ship ship) {
      try {
         // Kullanıcı yetkilerini kontrol et.
         // Yapılan işlemi logla.
         // Transactional baslat

         Ship _ship = shipService.saveShip(ship);

         // Ship'ı cache'e ilgili mmsiNo ile ekle.
         // transaction commit
         return _ship;
      }
      catch (Exception ex) {
         // Exception handling işlemleri.
         // Roll back
         throw;
      }
      finally {
         // Logla
      }
 }
```

Bir önceki sayfadaki gibi bir kod bloğunu AOP ile aşağıdaki şekle çevirebiliriz.

```
[ExceptionHandling]
[CheckUserAuthorization]
[LogProcess]
[Cache(TimeOut = 10000)]
[Transactional]
public Ship saveShip(Ship ship) {
        Ship _ship = shipService.saveShip(ship);
        return _ship;
}
```

5. What is Singleton and where to use it ?

Singleton: Yalnızca bir instance oluşturulacak ve Nesne her çağrıldığında aynı instance (sınıf örneği) vermesi anlamına gelmektedir.

Varsayılan scope her zaman singleton'dur. Ancak, bir bean'in yalnızca bir örneğine ihtiyacınız olduğunda, aşağıdaki kod parçacığında gösterildiği gibi, bean yapılandırma dosyasında kapsam özelliğini singleton olarak ayarlayabilirsiniz .

```
<!-- A bean definition with singleton scope -->
<bean id = "..." class = "..." scope = "singleton">
   <!-- collaborators and configuration for this bean go here -->
</bean>
```

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

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. Yani çalışan uygulamalarda kullanılır.

autoconfig	Tüm autoconfig tanımlamalarını gösterir	True
beans	Spring tarafından yönetilen tüm beanleri gösterir	True
dump	Thread dump almayı sağlar.	True
env	Spring's ConfigurableEnvironment değerlerini gösterir.	True
health	Uygulama sağlığını gösterir.	False
info	Uygulama bilgilerini gösterir.	False
loggers	Uygulamada kullanılan log bilgileri gösterir.	True
mappings	@RequestMapping tanımlaması yapılan değerleri gösterir.	True
trace	En son kullanılan 100 HTTP isteklerini listeler.	True

Spring Boot uygulamanızın sonuna actuator/beans , actuator/health gibi yukarıda belirlediğimiz parametreleri yazarak endpointlere ulaşıp bilgi almamız mümkün olmaktadır.

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

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.

8. Why to use VCS ?

Bir dosyanın değişik sürümlerini korumak istiyorsak, Sürüm Kontrol Sistemi (VCS) kullanmamız çok akıllıca olacaktır. VCS, dosyaların ya da bütün projenin geçmişteki belirli bir sürümüne erişmemizi, zaman içinde yapılan değişiklikleri karşılaştırmamızı, soruna neden olan şeyde en son kimin değişiklik yaptığını, belirli bir hatayı kimin, ne zaman sisteme dahil ettiğini ve daha başka pek çok şeyi görebilmemizi sağlar. Öte yandan, VCS kullanmak, bir hata yaptığımızda ya da bazı dosyaları yanlışlıkla sildiğimizde durumu kolayca telâfi etmemize yardımcı olur. Üstelik, bütün bunlar bize kayda değer bir ek yük de getirmez.

Birden fazla kişi aynı proje hatta aynı kod üzerinde çalışabilir ve bu değişiklikler tek bir kod dosyasına entegre edilebilir. Farklı gereksinimler için farklı dallar oluşturup aynı anda birden fazla geliştirme yapabilir ve bu geliştirmeleri sonradan tek bir projede toplayabiliriz

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

SOLID adı, her harfin bir yazılım tasarım ilkesini temsil ettiği, aşağıdaki gibi hatırlatıcı bir kısaltmadır:

1. **S**ingle Responsibility Principle
2. **O**pen Closed Principle
3. **L**iskov's Substitution Principle
4. **I**nterface Segregation Principle
5. **D**ependency Inversion Principle

Single Responsibility Principle (Tek Sorumluluk İlkesi)

"Bir sınıfın yalnızca bir sorumluluğu olmalı"

Başka bir deyişle, bir sınıfı yalnızca bir amaç için yazmalı, değiştirmeli ve korumalıyız. Eğer bu model sınıfı ise yalnızca bir oyuncuyu ya da varlığı temsil etmelidir. Bu, bize başka bir varlığa ait değişikliklerinin etkilerinden endişe etmeden gelecekte değişiklik yapma esnekliği sağlayacaktır.

Benzer şekilde, eğer servis / yönetici sınıfı yazıyorsak, yöntem çağrıları sadece bu kısmı içermeli ve hiçbir şey içermemelidir. Modül ile ilgili global fonksiyonlar bile kullanılmaz. Onları başka bir global olarak erişilebilir sınıf dosyasında ayırmak daha iyi olur. Bu, belirli bir amaç için sınıfın korunmasına yardımcı olacaktır ve sınıfın sadece belirli bir modüle görünürlüğüne karar vermiş oluruz.

Diyelim ki bir kitapçı için bir Java uygulaması yazıyoruz. Kullanıcıların her bir kitabın adını ve yazarını alıp ayarlamasına ve envanterde kitabı aramasına izin veren bir Kitap sınıfı oluşturuyoruz.

```
1
2  public class Kitap {
3      String baslik;
4      String yazar;
5
6      String getBaslik() {
7              return baslik;
8      }
9      void setBaslik(String baslik) {
10             this.baslik = baslik;
11     }
12     String getYazar() {
13             return yazar;
14     }
15     void setYazarString yazar) {
16             this.yazar = yazar;
17     }
18     void kitapAra() {...}
19 }
20
```

Yukarıdaki Kitap sınıfında Tek Sorumluluk İlkesini ihlal eder şekilde iki sorumluluk mevcuttur. İlk olarak, kitaplarla ilgili verileri (başlık ve yazar) belirler. İkincisi, envanterdeki kitabı arar. Ayarlayıcı yöntemleri, aynı kitabı envanterde aramak istediğimizde sorunlara yol açabilecek olan Kitap nesnesini değiştirebilir.

Bununla birlikte, Kitap sınıfı iki sorumluluğa sahip olduğundan yukarıdaki kod Tek Sorumluluk İlkesini ihlal eder. İlk olarak, kitaplarla ilgili verileri (unvan ve yazar) belirler. İkincisi, envanterdeki kitabı arar. Ayarlayıcı yöntemleri, aynı kitabı envanterde aramak istediğimizde sorunlara yol açabilecek olan Kitap nesnesini değiştirir.

Tek Sorumluluk İlkesini uygulamak için iki sorumluluğu çözmemiz gerekir. Refactor kodunda, Kitap sınıfı yalnızca Kitap nesnesinin verilerini almaktan ve ayarlamaktan sorumlu olmalıdır.

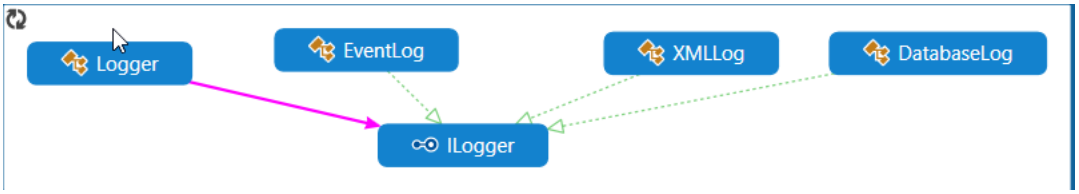
```
1
2  public class Kitap {
3      String baslik;
4      String yazar;
5
6      String getBaslik() {
7              return baslik;
8      }
9      void setBaslik(String baslik) {
10             this.baslik = baslik;
11     }
12     String getYazar() {
13             return yazar;
14     }
15     void setYazar(String yazar) {
16             this.yazar = yazar;
17     }
18 }
19
```

Ardından, envanteri kontrol etmekten sorumlu olacak Envanter adlı başka bir sınıf oluşturuyoruz. KitapAra() yöntemini buraya taşır ve kurucudaki Kitap sınıfına başvururuz.

```
1
2  public class Envanter{
3       Kitap kitap;
4
5      Envanter(Kitap kitap) {
6          this.kitap = kitap;
7      }
8
9      void KitapAra() {...}
10 }
11
```

Open Closed Principle (Açık Kapalı İlkesi)

Açık kapalı prensibi, yazılım geliştirirken kullandığımız varlıkların (class, method vs.) gelişime açık, kodların ise değişime kapalı olması ilkesidir. Örneğin; bir loglama altyapısı oluşturduğunuzu düşünün, Veritabanına ve XML'e kayıt tutuyorsunuz. Daha sonradan Eventloglara da log tutma ihtiyacı hissettiğinizde, sadece Eventloglara kayıt tutan kodları yazmanız yetecek, kodunuzda hiçbir değişiklik yapmadan bu yapı sisteme entegre olacak. Bunun için uygulayacağımız çözüm şu şekilde olabilir:




```
public interface ILogger
{
    4 references
    string LogTut();
}

0 references
public class DatabaseLog : ILogger
{
    4 references
    public string LogTut() {
        return "Veritabanına log tutuldu.";
    }
}

0 references
public class XMLLog : ILogger
{
    4 references
    public string LogTut() {
        return "XML'e log tutuldu.";
    }
}
```

```
references
public class Logger
{
    private ILogger logger;

    1 reference
    public void LogTo(ILogger logger)
    {
        this.logger = logger;
    }

    1 reference
    public string Log()
    {
        return this.logger.LogTut();
    }
}
```

Burada LogTo methoduna ILogger interface'inden implemente olmuş bir class veriyoruz. Yani; daha sonrasında gene ILogger interface'i üzerinden implemente olmuş EventLog isimli bir class yazarsak Logger sınıfı üzerinde hiçbir değişiklik yapmamız gerekmeden, sisteme n sayıda log tutan yapı entegre edebileceğiz.

```
3 references
public class EventLog : ILogger
{
    4 references
    public string LogTut() {
        return "Eventlog'lara kayıt edildi.";
    }
}
```

Liskov Substitution Principle (LSP)

Liskov Substitution Principle'a göre alt sınıflardan oluşturulan nesnelerin üst sınıfların nesneleriyle yer değiştirdiklerinde aynı davranışı göstermek zorundadır. Yani; türetilen sınıflar, türeyen sınıfların tüm özelliklerini kullanmak zorundadır.

```
public class Rectangle {

        private double width;
        private double height;



        public Rectangle(double width, double height) {
                super();
                this.width = width;
                this.height = height;
        }



        public double getWidth() {
                return width;
        }



        public double getHeight() {
                return height;
        }


        public double area(){
                return width*height;
        }


public class Square extends Rectangle{


    public Square(double side) {
                super(side, side);
        }
}
```

Interface Segregation Principle (ISP)

Interface Segregation prensibine göre, her interface'in belirli bir amacı olmalıdır. Tüm metodları kapsayan tek bir interface kullanmak yerine, herbiri ayrı metod gruplarına hizmet veren birkaç interface tercih edilmektedir.

Aşağıdaki interface birden fazla iş yapmaktadır. Bu arayüzden türetilen sınıflar tüm metodları kullanmak zorunda kalacaktır. Bunun yerine bu arayüzler daha küçük iş birimlerine ayrılmalıdır.

Aşağıdaki interface birden fazla iş yapmaktadır. Bu arayüzden türetilen sınıflar tüm metodları kullanmak zorunda kalacaktır. Bunun yerine bu arayüzler daha küçük iş birimlerine ayrılmalıdır.

```
interface IPost {

    void CreatePost();

    void ReadPost();
}
```

Küçük parçalara ayrılmış interface'ler sınıflara daha kolay eklenirler. Bu sayede bu arayüzlerden türetilen sınıflar kullanmadıkları metodları almamış olurlar.

```
interface IPostCreate {

    void CreatePost();
}
interface IPostRead {

    void ReadPost();
}
```

Dependency Inversion Principle (DIP)

- Üst seviye (High-Level) sınıflar, alt seviye (Low-Level) sınıflara bağlı olmamalıdır, ilişki abstraction veya interface kullanarak sağlanmalıdır.
-
- Abstraction(soyutlama) detaylara bağlı olmamalıdır, tam tersi detaylar abstraction(soyutlama)'lara bağlı olmalıdır.

Aşağıdaki Blog sınıfı içerisindeki Create metodu, Post sınıfı içerisindeki CreatePost isimli metoda bağımlıdır. Bunun sebebi CreatePost method'unun Create içerisinde kullanılmasıdır. Bu metotda yapılacak tüm değişiklikler Create method'unda da değişiklik gerektirecektir.

```
class Blog {

    //  High Level Class
    public final void Create() {
        Post post = new Post();
        post.CreatePost(true);
    }
}
class Post {

    //  Low Level Class
    public final void CreatePost(boolean picture) {
        //  Process
    }
}
```

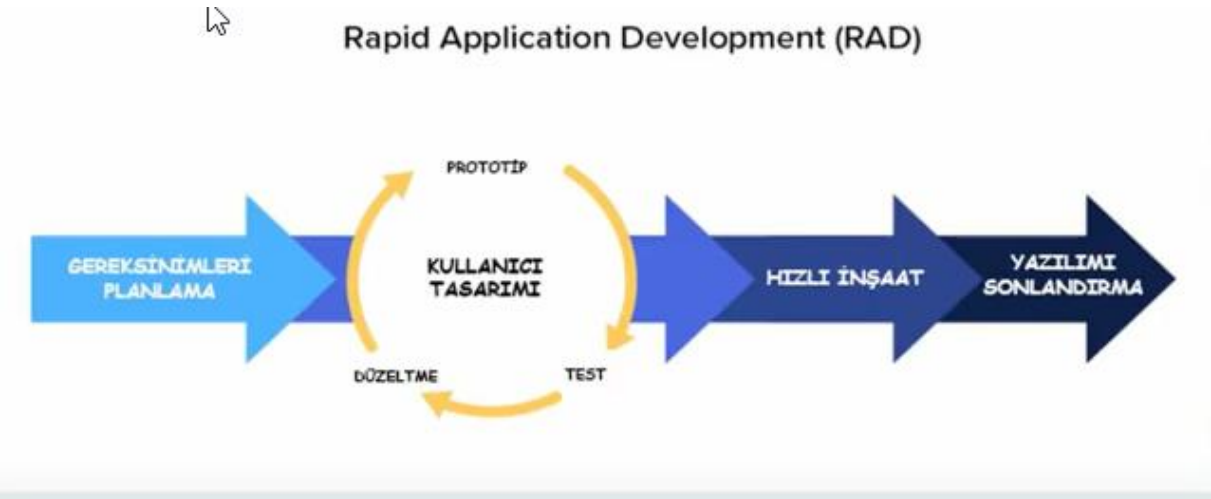
Bu bağımlılıktan kurtulabilmek için aşağıdaki kodu uygulayabiliriz.

```
interface IContent {

    void CreatePost(boolean picture);
}
class Blog {

    // High Level Class
    IContent content;

    public Blog() {
        this.content = new Post();
    }

    public final void Create() {
        this.content.CreatePost(true);
    }
}
class Post extends IContent {

    // Low Level Method
    public final void CreatePost(boolean picture) {
        // Process
    }
}
```

10. What is RAD model ?

Türkçesi ‘Hızlı Uygulama Geliştirme’ olarak çevrilen Rapid Application Development (RAD), uzun süren geliştirme ve test döngülerine göre hızlı prototip oluşturmaya ve hızlı geri bildirime öncelik veren bir geliştirme modelidir. Hızlı uygulama geliştirme ile geliştiriciler, her seferinde sıfırdan bir geliştirme programı başlatmaya gerek kalmadan bir yazılıma birden çok yineleme ve güncelleme yapabilir.

RAD modelinin 4 temel adımı aşağıdaki şemadaki gibidir.



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

Spring Boot tanıtılmadan önce Spring Developers, Bağımlılık yönetimine çok zaman harcıyordu. Spring Boot Starters, bu sorunu çözmek için tanıtıldı, böylece geliştiriciler, Bağımlılıklardan ziyade gerçek kod üzerinde daha fazla zaman harcayabilir. Spring Boot Starters, pom.xml'deki <bağımlılıklar> bölümünün altına eklenebilen bağımlılık tanımlayıcılarıdır.

Starter kullanmanın avantajları şunlardır:

- Geliştiriciler için Yapılandırma süresini azaltarak üretkenliği artırır
- Eklenecek bağımlılık sayısı azaldığı için POM'u yönetmek daha kolaydır.
- Test edilmiş , üretime hazır ve desteklenen bağımlılık yapılandırmaları sağlar.
- Bağımlılıkların adını ve sürümünü hatırlamamıza gerek kalmaz.

12. What is Caching ? How can we achive caching in Spring Boot ?

Caching sık kullanılan dataları kaydetme tekniğine verilen isimdir. Kaydetme işlemi uygulamayı host eden sunucunun ram belleğinde(In Memory Caching) ve harici bir caching sunusunda(Distributed Caching) gerçekleştirilebilir.

Cacheleyeceğimiz datanın iki özelliği taşıyor olması gerekir. Birincisi çok güncellenmemeli, İkincisi çok sık erişiliyor olmalı.

Spring Boot uygulamasına Spring Caching özelliği kazandırmak için maven projesi yapıyorsak bağımlılığımız aşağıdaki gibi olması gereklidir.

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

Controllerımızda ilgili metodumuzdan çıkacak sonucun caching özelliği verebilmek için caching **@Cacheable** anotasyonu ile işaretleriz.

Main methodumuza **@EnableCaching** yazarak uygulamımıza bu özelliği kazandırmış oluyoruz.

İstediğimiz zaman bu özelliği devre dışı bırakmak istersek Caching ile işaretlediğimiz tüm anotasyonları silmemize gerek yok. Spring Boot uygulamamızın **application.properties** dosyasına; **spring.cache.type=none** yazmak yeterli olacaktır.

13. What & How & Where & Why to logging ?

Loglama, bir uygulamanın(application) çalışma zamanındaki(runtime) durumunu sistematik kontrol edilebilir, okunabilir şekilde saklama yöntemidir. Loglamayı development, debug ve test süreçlerinde kullanabiliriz.

Loglama kavramında 3 önemli kavramı inceleyecek olursak;

Loglama, sistematik olmalıdır. Loglama, kontrol edilebilir olmalıdır. Loglama, bir uygulamanın durumunu açıklamalıdır.

Hızlı hata ayıklamak(debug): Bir problemle karşılaştırdığımızda, log kayıtları bize sorunun nereden kaynaklandığını gösterecektir. İyi yazılmış bir loglama kodu,sorunun asıl kaynağını nereden çıktığını daha kısa surede bulmamızı sağlayacaktır bu da bize debug işleminde vakit kazandıracaktır.

Kolay bakım: İyi yazılmış loglama yapısı ile uygulamamızın bakimi/devam edilebilirliği kolay olacaktır. Loglama kodlarının sadece debug için tutulmadığını biliyorsak bu log kayıtları bize sistem hakkında bilgi verecektir ve bu bilgiler ışığında sistem bakimi daha kolay olacaktır.

Geçmiş: Loglama bilgileri istenilen bir dizinde , istenilen bir dosya isminde örneğin tarihsel bir ek olarak ufuk_log_29_12_2013 seklinde tutulabilir. İçerik olarak da farklı formatlara uygun şekilde tutabileceğimizi belirtmiştik. Geriye donuk olarak bu loglanan dosyalar tutulur herhangi bir soruna veya farklı bir duruma karşı tutulabilir.

Maliyet ve Zaman kazancı: Iyi yazilmis loglama yapisi ile , hizli hata ayiklama, kolay bakim gibi zaman ve maliyet kazanci saglanabilir.

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

Swagger2, RESTful web hizmetleri için REST API belgeleri oluşturmak için kullanılan açık kaynaklı bir projedir. RESTful web servislerimize web tarayıcısı üzerinden erişmek için bir kullanıcı arayüzü sağlar.

Spring Boot uygulamasında Swagger2'yi etkinleştirmek için, derleme yapılandırmaları dosyamıza aşağıdaki bağımlılıkları eklemeniz gerekir.

```
<dependency>
   <groupId>io.springfox</groupId>
   <artifactId>springfox-swagger2</artifactId>
   <version>2.7.0</version>
</dependency>
<dependency>
   <groupId>io.springfox</groupId>
   <artifactId>springfox-swagger-ui</artifactId>
   <version>2.7.0</version>
</dependency>
```

Ana Spring Boot uygulamanıza @EnableSwagger2 anotasyonu ekleriz. @EnableSwagger2 anotasyonu, Spring Boot uygulamanız için Swagger2'yi etkinleştirmek için kullanılır.

Ardından, Spring Boot uygulamanız için Swagger2'yi yapılandırmak için Docket Bean oluşturulur. Swagger2 için REST API'lerini yapılandırmak için temel paketi tanımlamamız gerekiyor.

Son olarak oluşturulan Docket Bean'ı ana Spring Boot uygulama sınıfı dosyasının kendisine ekleriz.

Nihai olarak bu arayüze sahip olur ve işlemlerimizi bu arayüz üzerinden gerçekleştirebiliriz.