# 1.Spring dışında dependency injection için kullanabileceğimiz framework’ler / kütüphaneler nelerdir ?

**Ninject** oldukça popüler IoC container'lar dan biri olup bağımlılıkları enjekte etmede kullanılan open source bir kütüphanedir. Dependency injection bizlere loosely coupled dediğimiz birbirlerine gevşek bağlı ve daha kolay test edilebilir geliştirmeler yapmamızı sağlayan bir design pattern dir. IoC ise belli özelliklere sahip ve birbirlerine bağımlı nesnelerin işlevlerini gerçekleştirmek için ihtiyacı olan instance'ları kendilerinin değilde bir IoC container tarafından yönetilmesini söyler.

**Koin** basit ve güçlü dependencies injection kütüphanesidir. Kotlin ile yazılmıştır.

# 2. @SpringBootApplication anotasyonu hangi anotasyonları kapsamaktadır ? Görevlerini kısaca açıklayınız.

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

**@Configuration** : Java tabanlı konfigürasyon işlemi yapan bir anotasyondur.

* **@Bean** tanımlaması içeren fonksiyonlar içerir.
* Tanımlandığı fonksiyon her neyi return ediyorsa, onun context içine ekleneceğini belirtir.
* **@Bean**, genelde 3rd parti sınıflar context içine eklenmek istediğinde kullanılır.
* Dosya içerisinden birden çok bean tanımı olabileceğini belirtir.
* Eskiden config işlemleri xml dosyaları içinde yapılırken, Spring 3’ten sonra **@Configuration** anotasyonuyla java sınıflarından **@Bean** tanımlamasıyla yapabilme özgürlüğü geldi.
* Runtime anında config olarak tanımlanan sınıf içindeki @Bean anotasyonlarının, spring container tarafından işlenebilmesi için belirtilir.
* Aslında **@Configuration** olarak tanımlanan sınıf içindeki **@Bean** tanımlamalarının her biri için app context içinde oluşturulmak üzere otomatik xml tanımlaması yapılmış oluyor.
* Eğer ben tanımı XML dosyasından yapıldıysa, **@ImportResource**(“another-application-context.xml”) tanımlaması ile harici bir xml import edilebilir
* **@Import**(OtherConfiguration.class) tanımlamasıyla harici bir config import edilebilir.

**@Bean Anotasyonu Nedir?**

1. Metod düzeyinde kullanılan bir anotayondur.
2. Bu anotasyon tıpkı @Component anotasyonu gibi çalışır.
3. Bir sınıf @Configuration anotasyonu ile tanımlıysa, o sınıfın fonksiyonları @Bean olarak tanımlanabilir.
4. @Bean olarak tanımlanan bir fonksiyon için, fonksiyon adı geçerli bir id yada bean adı olur.
5. Eğer bir metod @Bean ile tanımlıysa, Spring tarafından yönetilen bir Bean ürettiğini belirtilmiş olur.
6. Spring dokumantasyonunda @Configuration ve @Bean için sunulan örnek kullanım :

@Configuration public class AppConfig { @Bean public TransferService transferService() { return new TransferServiceImpl(); } }

@Bean tanımlaması üstteki gibi Java dosyası içerisindeki bir fonksiyon üzerinde direkt olarak tanımlanacağı gibi şu şekide bir XML dosyası içinde de tanımlanabilir :

<beans> <bean name="transferService" class="com.acme.TransferServiceImpl"/> </beans>

Her iki kullanım da aynıdır ve Spring Application Context içinde, "transferService" adında bir bean nesnesi oluşturulmasını sağlar.

**@ComponentScan :** Projeye dahil edilen komponenetlerin otomatik taranmasını sağlar.

**@Component** Anotasyonu Ne İşe Yarar:

* Class seviyesinde bir anotasyondur.
* Component taraması sırasında bu anotasyona ait sınıflar taranır.
* @Repository, @Service, @Configuration and @Controller anotasyonlarının tamamı **@Component** anotasyonudur.

**@ComponentScan** Nasıl Çalışır

* **@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

**@EnableAutoConfiguration** : Varsayılan konfigürasyonların otomatik gerçekleşmesini sağlar.

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, Thymeleaf classpath içinde yer alıyorsa, tema resolver otomatik aktif edilir.
* Örneğin, projeye bir veritabanı import edilmişse, database konfigürasyonlarının otomatik gerçekleşmesi için JdbcTemplate bean'i otomatik oluşturulur.
* **@Conditional** bazlıdır.
* **@EnableAutoConfiguration**(exclude=DataSourceAutoConfiguration.class) yaparak istediğimiz config sınıfını devredışı bırakabiliriz.
* **@EnableAutoConfiguration** Jar bağımlılığı olan sınıfları konfigüre eder.

**EK NOT**

**@Component ve @Bean Anotasyonları Arasındaki Fark Nedir?**

Her iki anotasyonda aynı işi gerçekleştirirler. Tanımlandıkları yer için Application Context içerisinde bean oluşturmayı sağlarlar. Aralarındaki temel fark, kullanım amaçlarıyla ilgilidir.

Örneğin iki farklı uygulamanız var ve bu iki uygulama da Mail modülünü kullanıyor fakat, uygulamardan bir tanesi mail modülüne ait tüm servisleri kullanırken, diğerinde sadece belli başlı servisler kullanılıyor. Böyle bir durumda Mail modülünü projeye @Component olarak import etmek demek, işe yarayan ve yaramayan tüm Mail servislerini projeye gereksiz yere dahil etmek anlamına gelecektir. Bunu önlemek için ise, @Component yönetimini seçtiysek eğer, @ComponentScan için koşullar yazmamız gerekecektir. Fakat bu koşulları dikkatli yazmazsak servisin düzgün çalışmama ihtimali de var.

Diğer taraftan, @Component kullanmak yerine @Bean ifadesi kullanılabilir. @Bean ifadesini sadece kullanacağımız servisler için tanımlarsak, işe yaramayan servisleri için boş yere projemize import etmemiş oluruz. Bu yüzdendir ki @Bean tanımlaması daha çok, third-pary (üçüncü parti) uygulamaları projemize import ederken tercih edilir. Diğer durumlarda @Component kullanılabilir.

# 3.@Primary, @Qualifier anotasyonlarının kullanım amaçlarını açıklayınız.

**@Qualifier anotasyonı**

Qualifier’ı açıklamadan önce Autowired anotasyondan bahsedelim , Autowired bir bean yönetimidir. Bu yönetim içerisinden değerleri ile birlikte enjekte edilmektedir. Eğer biz beanimize birden fazla değer atama yaparsak yani spring konfigürasyon dosyasında birden fazla bean alanlarına tanımlama yaparsak Spring bu tanımlamalarının hangisinin kullanılacağına ” @Qualifier ” anotasyonu kullanılarak çözülmektedir.

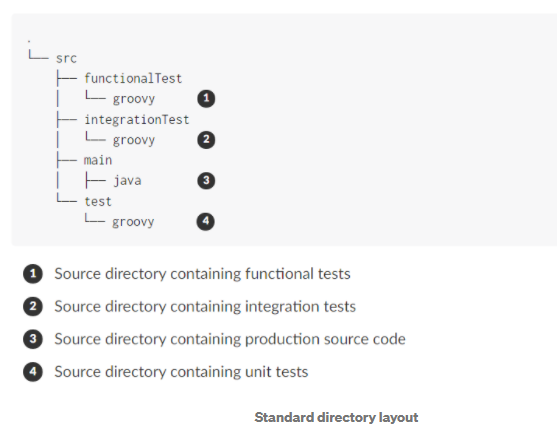
**@Primary anotasyonı**

Spring frameworkünde , aynı türden birden fazla @Bean olduğunda, bir @Bean’a öncelik vermek için @Primary notu kullanılır.

# 4. Convention over configuration kavramını seçtiğiniz bir örnek üzerinden açıklayınız

Konfigürasyon üzerine konvansiyon (aynı zamanda konvansiyonel kodlama olarak da bilinir), yazılım frameworkü tarafından kullanılan ve frameworku kullanan bir geliştiricinin esnekliği kaybetmeden vermesi gereken kararların sayısını azaltmaya çalışan bir yazılım tasarım paradigmasıdır.

Örnek: Maven veya gradle'dan otomatik oluşturmayı kullanarak bir projenin düzenini tanımlamak.



# 5. Aspect Oriented Programlama nedir ? Avantajları ve dezavantajları nelerdir ?

Aspect-Oriented Programming (AOP) ya da Cepheye Yönelik Programlama Xerox PARC tarafından 1990 yıllarında bulunmuş olup, kod geliştiricilere matematiksel işlemler ve kural dışı durumlar gibi içinden çıkılamaz şekilde karışık işleri en iyi şekilde ayırma imkanı sağlayan bir modeldir. Cepheye Yönelik Programlama tamamen yepyeni bir buluş olmayıp, günümüzdeki yazılım geliştirme gereksinimlerini daha modüler karşılayan bir yöntemdir.

Nesneye Yönelimli Programlama’nın ihtiyaçları doğrultusunda gelişen AOP, OOP’nin yerini almak için değil, tam aksine OOP’nin eksik kaldığı noktaları tamamlamak için gelişmiştir.

AOP birbiriyle kesişen ilgilerin (Cross-Cutting Concerns) ayrılması üzerinedir. Uygulama genelinde kullanılacak olan yapıları (logging,exception hand., cache, etc.) core tarafta yazdığımız koddan ayırarak bir çeşit ayrı küçük programcıklar şeklinde yazıp projede kullanmayı hedefler diyebiliriz.

Peki nedir bu kesişen ilgiler diye merak ederseniz aşağıda ki başlıklar örnek gösterilebilir.

Logging and Monitoring

Performance analysis

Debugging and Tracing

Undo Functionality

Validation of inputs and outputs

Morphing the behavior of existing objects

Object Filters

Security Implementation

Managing transactions

**Aspect-Oriented Programming’in Sağladıkları**

* İçi içe yazılmış ve sürekli tekrar eden kodlardan kurtulabiliyoruz,
* Daha temiz ve anlaşılır kodlar yazabiliyoruz,
* Yazmış olduğumuz kodları daha abstract hale getirerek modülerliğini arttırıyoruz,
* Bakım ve geliştirme maliyetlerini azaltıyoruz,
* Uygulamamızı daha yönetilebilir ve daha esnek hale getirebiliyoruz.

# 6.SOLID prensiplerini kısaca açıklayınız. Sizce her koşulda bu prensipler çerçevesinde mi kod yazılmalıdır ? Neden ?

**1. Single responsibility principle (SRP)**

Her class’ın, her mothodun sadece tek bir yaptığı iş vardır, böylece bir değişiklik yapmak için sadece bir nedeniniz olmuş olur. Genişleyebilir, tekrar kullanılabilir ve test edilebilir yapılar kurmak için tek sorumluluk ilkesini dikkate almamız gerekir.

**2.Open/Closed Principle**

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.

**3.Liskov ‘s Substitution Principle**

Liskov’un yerine geçme prensibi alt sınıflardan oluşturulan nesnelerin üst sınıfların nesneleriyle yer değiştirdiklerinde aynı davranışı göstermek zorunda olduklarını söyler. Yani; türetilen sınıflar, türeyen sınıfların tüm özelliklerini kullanmak zorundadır. Eğer kullanmaz ise ortaya işlevsiz, dummy kodlar çıkacaktır. Bu durumda üst sınıfta if else blokları kullanarak tip kontrolü yapmamız gerekebilir ve böylelikle Açık Kapalı prensibine de ters düşmüş oluruz.

**4.Interface Segregation Principle**

Arayüz ayırım prensibi, bir arayüze gerektiğinden fazla yetenek eklemememiz gerektiği söyler.

**5.Dependency Inversion Principle**

Bağımlılığın ters çevirilmesi ilkesine göre üst seviye sınıflar, modüller, methodlar vs. alt seviyeli sınıflara bağımlı olmamalıdır. Alt sınıflarda yapılan değişiklikler üst sınıfları etkilememelidir

# 7. Swagger nedir, ne amaçla kullanılmaktadır ?

Swagger Rest API geliştirmek için gerekli bir sözleşme standardı ve bu çerçevede işlev gören yardımcı araçlar sunan bir teknolojidir. Swagger sunduğu standart ve araçlarla API tasarım, geliştirme, dokümantasyon ve test aşamasında kolaylık sağlamaktadır.

# 8. Richardson Maturity Model’i seviyeleriyle birlikte açıklayınız

Richardson Maturity Model Rest API’lerimizin hangi seviyede olgun olduğunu gösteren bir olgunluk seviyesidir. RMM 4 seviyeden oluşmaktadır ve 0’dan 3’e başlayan seviyeler, yukarı doğru çıktıkca daha etkin kullanıldığını ifade etmektedir.

**Level 0: Swamp of POX**

Servisimizdeki sadece tek metod üzerinden POST olarak erişebildiğimiz seviyedir. Bu seviye transfer protokolü olarak da adlandırılmaktadır.

**Level 1: Resources**

Servimizdeki URI üzerinden bir metot üzerinden erişilebildiğimiz seviyedir. Örnek URI http://localhost/students/1

**Level 2: HTTP verbs**

POST, PUT, GET ve DELETE metotları çağırdığımız seviyedir.

**Level 3: Hypermedia controls**

Servislerimizde URI’nin istek gönderip cevaba göre tekrar URI’ye istek yapabildiğimiz bir seviyedir. Servis üzerinden gelen cevaba göre akıştaki davranışı görebiliriz. HATEOAS kullandığımız seviyedir.

# 9.URL, URI, URN kavramlarını bir örnek üzerinden açıklayınız

**URI (Uniform Resource Identifier)**

URI (Uniform Resource Identifier), kaynağa tam olarak tanımlayıcı (identifier) ile işaret eden (belge, resim, dosya vb.) ve bu işaretleme için standart bir formata sahip karakter dizgisidir. URL kapsamında tutulan alt tanımlardır. Bir sayfa, görsel, dosya vb. URI için örnek olabilir; http://ornekwebsitesi.com/logo.png, ISBN 0-486-27557-4 gibi. URI, URL ve URN olarak 2 kapsama sahiptir. Hem URN’ler (adlar) hem de URL’ler (konumlayıcılar) URI’lardır ve belirli bir URI aynı anda hem bir isim hem de konumlandırıcı olabilir.

**URL (Uniform Resource Locator)**

URL (Uniform Resource Locator), Tekdüzen Kaynak Bulucu ya da Kaynak Konumlayıcı şeklinde ifade edilebilir. İnternet aracılığıyla erişişebilecek kaynakların (dosyalar, dökümanlar vb.) konumu URL ile ifade edilir. URL teknik olarak URI’nın başlangıç kısmını oluşturur. Yapısal olarak, URL’in ardından ise URN gelmektedir. Ancak, bir konum belirtilmek istendiğinde çoğu durumda URI yerine URL ifadesi kullanılmaktadır. URI sözdizimi (syntax) şöyledir:

scheme://domain:port/path?query\_string#fragment\_id

scheme://user:password@host:port/path?query\_string#fragment\_id

Bir kaynağın nerede olduğunu ve ona nasıl ulaşılacağını belirleyen URL http://, ftp:// gibi pek çok protokolü barındırabilir. URL ve URI ilişkisini bir adres tarifi üzerinden gerçekleştirdiğimizde, URL konumu nitelerken, URN konumdaki kişinin adına benzer. Özetle, URL bir konum sağlarken URN bir şeyin kimliğini ifade eder. Örneğin, telnet://192.0.2.16:80/ bir URL iken tel:+1-816-555-1212 bir URN’dir.

**URN (Uniform Resource Name)**

Bir kaynağı benzersiz ve kalıcı bir adla tanımlar, ancak bunu İnternet’te nasıl bulunacağının söylenmesi gerekmez. Belgeleri tanımlamakla sınırlı değildirler. Hatta, URN’ler fikirleri ve kavramları tanımlayabilir. Bir URN genelde urn: prefix’i ile başlar.

urn:isbn:0451450523 ISBN ile bir kitabı işaret eder.

urn:uuid:6e8bc430-9c3a-11d9-9669-0800200c9a66 global olarak benzersiz bir tanımdır.

urn:publishing:book bir belgeyi kitap türü olarak tanımlayan bir XML ad alanıdır.

# 10. Idempotency nedir ? Hangi HTTP metotları idempotent’ tir ?

Bir metodun bir defa çağrıldığında alınan sonuç ile birden fazla kez çağrıldığında alınan sonuç aynı ise bu bir idempotent metottur.

Sunucuya /students/1 şeklinde bir istek gönderdiğimizde, id si 1 olan kayıt dönecektir. Bu isteği birden fazla kez tekrarlamamız sonucu değiştirmeyecektir. Bundan dolayı GET idempotenent bir metottur.

/students/1

body: {‘name’:’Ahmet’, ‘surname’:’Acar’}

Sunucuya yukarıdaki gibi bir istek gönderdiğimizde Metin Alnıaçık ifadesi Ahmet Acar olarak güncellenecektir. İsteği bir kez daha gönderdiğimizde veritabanındaki değer Ahmet Acar olduğunda bir değişiklik olmayacaktır. Bu da idempotenent bir metottur.

Sunucuya yukarıdaki gibi bir istek gönderdiğimizde id si 1 olan kayıt silinecektir. İlgili istediği tekrar çağırdığımızda id si 1 olan bir kayıt olmadığından herhangi bir işlem gerçekleşmeyecektir. Bu da idempotenent bir metottur.

/students

body: {‘name’:’Mehmet’, ‘surname’:’Demirci’}

Sunucuya yukarıdaki gibi bir istek gönderdiğimizde, {‘id’:’4’, ‘name’:’Mehmet’, ‘surname’:’Demirci’} şeklinde bir kayıt oluşacaktır. İlgili isteği bir kez daha çağırırsak {‘id’:’5’, ‘name’:’Mehmet’, ‘surname’:’Demirci’} şeklinde bir kayıt daha oluşacaktır. Yani kaç defa çağırırsak o kadar yeni kayıt oluşacaktır. Bundan dolayı POST non-idempotenent bir metottur

# 11.RFC (Request For Comment) neyi ifade etmektedir ? HTTP hangi RFC dokümanında açıklanmıştır ? Bu dokümanda HTTP hakkında ne tür bilgiler yer almaktadır ?

RFC internet standartları ve protokolleri için teknik yayınlar diyebiliriz. Bu standartlar belirlendikten sonra üzerine teknolojiler geliştirildiği için değişmesi mümkün olmayan yeni talepler sonrasında yeni versiyonları çıkan dökümanlardır. Bir kaç örnek verirsek daha iyi anlatabiliriz

* RFC2616: (HTTP) Hypertext Transfer Protocol — HTTP/1.1
* RFC1034: (DNS) Domain Names
* RFC791: (IP) Internet Protocol
* RFC821 : (SMTP) Simple Mail Transfer Protocol
* RFC6749: The OAuth 2.0 Authorization Framework