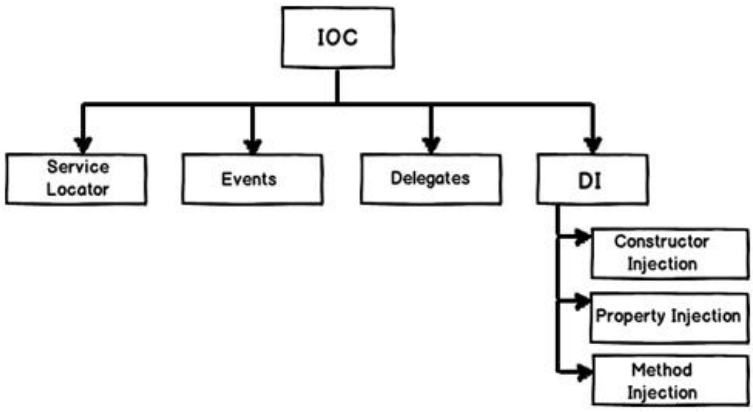
HW#2

1 – IOC and DI means ?



IoC (Kontrolün tersine çevrilmesi **)** : **-** bir yazılım tasarım prensibidir. Nesnelerin işlerini yapmak için güvendikleri başka nesneler yaratmadığı anlamına gelir. Bunun yerine ihtiyaç duydukları nesneleri bir dış hizmetten alırlar çeşitli şekillerde uygulanır (xml dosyası veya tek uygulama hizmeti v.b.)

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.

Örneğin; Spring ile basit bir not defteri uygulaması geliştirdik. Spring bir framework olduğu için bütün kaynakları kendisi ayarlayacak ve yönetecektir.Yani projeyi Spring başlatacak ve hazır olduğunda sizin kodunuzu çalıştıracaktır. Not defteri için bir kayıt oluşturduğunuzda sizin kodunuzu çalıştıracak ve sizin kodunuzun çalışması bittiğinde kontrolü yeniden Spring Framework'ü devralacaktır. İşte tam olarak bu olaya Inversion of Control denmektedir.

Inversion of Control'ün getirdiği avantajlar;

1. Bir methodun implementasyonundan izole bir şekilde çalıştırılabilmesini sağlar.
2. Farklı implementasyonlar arasında, kolayca geçiş yapabilmenizi sağlar.
3. Program modülerliğini artırır.
4. Bağımlılıklar en aza indiği için test etmeyi/yazmayı kolaylaştırır.

DI , bağımlı nesne almanın IoC ilkesinin somut nesneler kullanmadan, ancak soyutlamalar (arayüzler) kullanılarak yapıldığı anlamına gelir. Bu, tüm bileşenler zincirini test edilebilir hale getirir, çünkü daha yüksek seviyeli bileşen, daha düşük seviyeli bileşene bağlı değildir, sadece arayüzden. Mock'lar bu arayüzleri uygular. Dependency injection kaba tabir ile bir sınıfın/nesnenin bağımlılıklardan kurtulmasını amaçlayan ve o nesneyi olabildiğince bağımsızlaştıran bir programlama tekniği/prensibidir.
DI, IoC'nin bir alt türüdür ve *constructor injection, setter injection ve ya Interface injection* ile uygulanır .

Ancak Spring yalnızca aşağıdaki iki türü destekler:

- **Setter injection**
    - Setter tabanlı DI, çekirdeklerini başlatmak için argümansız bir kurucu veya argümansız statik fabrika yöntemini çağırdıktan sonra kullanıcının çekirdeklerinde ayarlayıcı yöntemleri çağırarak gerçekleştirilir.
- **Constructor injection**
    - Yapıcı tabanlı DI, her biri bir işbirlikçiyi temsil eden bir dizi argümana sahip bir yapıcıyı çağırarak gerçekleştirilir. Bunu kullanarak, enjekte edilen çekirdeklerin boş olmadığını ve hızlı başarısız olduğunu doğrulayabiliriz (derleme zamanında ve çalışma zamanında değil), bu nedenle uygulamanın kendisini başlatırken elde ederiz NullPointerException: bean does not exist. Yapıcı enjeksiyonu, bağımlılıkları enjekte etmek için en iyi uygulamadır.

Dependency Injection uygulamanın avantajları nelerdir ?

- Bağımlılık oluşturacak nesneleri direkt olarak kullanmak yerine, bu nesneleri dışardan verilmesiyle sistem içerisindeki bağımlılığı minimize etmek amaçlanır. Böylece bağımlılık bulunan sınıf üzerindeki değişikliklerden korunmuş oluruz.

- Unit testlerin yazımını kolaylaştırırken doğruluğunu da artırır. Yazılım geliştirmedeki en önemli konulardan biri de yazılım içerisinde bulunan *component*lerin **“loosely coupled”** (gevşek bağlı) olmasıdır. Dependency Injection’da bunu sağlayabilen önemli tekniklerden birisidir. Böylece bağımsızlığı sağlanan sınıflar tek başına test edilebilir.

---

2 – Spring Bean Scopes ?

Bir bean scope(kapsam), o çekirdeğin yaşam döngüsünü ve onu kullandığımız bağlamlardaki görünürlüğünü tanımlar.

Spring çerçevesinin en son sürümü 6 tür scope tanımlar:

a. **singleton**
b. **prototype**
c. **request**
d. **session**
e. **application**
f. **websocket**

Bahsedilen son dört scope, *request, session, application* ve *websocket* yalnızca web uyumlu bir uygulamada kullanılabilir.

a. **Singleton scope :**

    Örneğin, Spring'i her ihtiyaç duyulduğunda yeni bir bean örneği üretmeye zorlamak için, bean kapsam niteliğini **prototip** olarak ilan etmelisiniz . Benzer şekilde, Spring'in her ihtiyaç duyulduğunda aynı bean örneğini döndürmesini istiyorsanız, bean'in kapsam niteliğini **singleton** olarak bildirmelisiniz .**Bu, bean tanımını Spring IoC konteyneri (varsayılan) başına tek bir örneğe dahil eder.**

Kapsam kavramını örneklemek için bir *Kişi varlığı oluşturalım:*

```
public class Person {
   private String name;

   // standard constructor, getters and setters
}
```

Daha sonra, **@Scope** notunu kullanarak *bean'i singleton* kapsamıyla tanımlarız :

```
@Bean
@Scope("singleton")
public Person personSingleton() {
   return new Person();
}
```

String değeri yerine bir sabiti aşağıdaki şekilde de kullanabiliriz :

```
@Scope(value = ConfigurableBeanFactory.SCOPE_SINGLETON)
```

Şimdi, aynı bean atıfta bulunan iki nesnenin, her ikisi de aynı bean örneğine atıfta bulunduğundan, yalnızca birinin **durumunu değiştirse bile aynı değerlere sahip olacağını gösteren bir test yazmaya devam edebiliriz:**

```
private static final String NAME = "John Smith";

@Test
public void givenSingletonScope_whenSetName_thenEqualNames() {
   ApplicationContext applicationContext =
     new ClassPathXmlApplicationContext("scopes.xml");

   Person personSingletonA = (Person) applicationContext.getBean("personSingleton");
   Person personSingletonB = (Person) applicationContext.getBean("personSingleton");

   personSingletonA.setName(NAME);
   Assert.assertEquals(NAME, personSingletonB.getName());

   ((AbstractApplicationContext) applicationContext).close();
}
```

Bu *örnekteki kapsamlar.xml* dosyası, kullanılan çekirdeklerin xml tanımlarını içermelidir:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans.xsd">

   <bean id="personSingleton" class="org.baeldung.scopes.Person" scope="singleton"/>
</beans>
```

b. Prototype Scope

Prototip kapsamına sahip bir bean , konteynerden her istendiğinde farklı bir örnek döndürür. Bean tanımındaki *@Scope* ek açıklamasına değer *prototipi ayarlanarak tanımlanır:*

```
@Bean
@Scope("prototype")
public Person personPrototype() {
  return new Person();
}
```

Singleton kapsamı için yaptığımız gibi bir sabit de kullanabiliriz :

```
@Scope(value = ConfigurableBeanFactory.SCOPE_PROTOTYPE)
```

Şimdi, *prototip* kapsamı ile aynı bean adını isteyen iki nesneyi gösteren öncekine benzer bir test yazacağız. Artık **aynı bean örneğine atıfta bulunmadıkları için farklı durumlara sahip olacaklar:**

```
private static final String NAME = "John Smith";
private static final String NAME_OTHER = "Anna Jones";

@Test
public void givenPrototypeScope_whenSetNames_thenDifferentNames() {
  ApplicationContext applicationContext =
    new ClassPathXmlApplicationContext("scopes.xml");

  Person personPrototypeA = (Person) applicationContext.getBean("personPrototype");
  Person personPrototypeB = (Person) applicationContext.getBean("personPrototype");

  personPrototypeA.setName(NAME);
  personPrototypeB.setName(NAME_OTHER);

  Assert.assertEquals(NAME, personPrototypeA.getName());
  Assert.assertEquals(NAME_OTHER, personPrototypeB.getName());

  ((AbstractApplicationContext) applicationContext).close();
}
```

Scops.xml dosyası, *prototip* kapsamıyla bean için xml tanımını eklerken önceki bölümde sunulan dosyaya benzerdir *:*

```
<bean id="personPrototype" class="org.baeldung.scopes.Person" scope="prototype"/>
```

WEB UYUMLU SCOPE'LAR

Yalnızca web'i tanıyan bir uygulama bağlamında kullanılabilen dört ek kapsam vardır. Bunları pratikte daha az kullanıyoruz. Request *scope,* tek bir HTTP isteği için bir bean örneği oluştururken, *oturum* kapsamı bir HTTP Oturumu için bir beean örneği oluşturur.Uygulama kapsamı, bir **ServletContext** yaşam döngüsü için bean örneğini oluşturur ve **websocket** kapsamı, belirli bir **WebSocket** oturumu için *onu oluşturur.*Bean'leri başlatmak için kullanılacak bir sınıf oluşturalım:

```
public class HelloMessageGenerator {
  private String message;

  // standard getter and setter}
```

c. Request Scope

Bean'i @Scope ek açıklamasını kullanarak *istek* kapsamıyla tanımlayabiliriz :

```
@Bean
@Scope(value = WebApplicationContext.SCOPE_REQUEST, proxyMode =
ScopedProxyMode.TARGET_CLASS)
public HelloMessageGenerator requestScopedBean() {
   return new HelloMessageGenerator();
}
```

ProxyMode özniteliği, web uygulaması bağlamının somutlaştırıldığı anda etkin bir istek olmadığı için gereklidir . Spring, bağımlılık olarak enjekte edilecek bir proxy oluşturur ve bir istekte ihtiyaç duyulduğunda hedef bean başlatır.

Yukarıdaki tanım için bir kısayol görevi gören *@RequestScope* oluşturulmuş bir açıklama da kullanabiliriz :

```
@Bean
@RequestScope
public HelloMessageGenerator requestScopedBean() {
   return new HelloMessageGenerator();
}
```

Daha sonra requestScopedBean öğesine enjekte edilmiş bir referansı olan bir denetleyici tanımlayabiliriz . Web'e özel kapsamları test etmek için aynı isteğe iki kez erişmemiz gerekiyor.

İstek her çalıştırıldığında *mesajı* görüntülersek , daha sonra yöntemde değiştirilse bile değerin *null olarak sıfırlandığını görebiliriz.* Bunun nedeni, her istek için farklı bir bean örneğinin döndürülmesidir.

```
@Controller
public class ScopesController {
   @Resource(name = "requestScopedBean")
   HelloMessageGenerator requestScopedBean;

   @RequestMapping("/scopes/request")
   public String getRequestScopeMessage(final Model model) {
      model.addAttribute("previousMessage", requestScopedBean.getMessage());
      requestScopedBean.setMessage("Good morning!");
      model.addAttribute("currentMessage", requestScopedBean.getMessage());
      return "scopesExample";
   }
}
```

d. Session Scope

Bean'ı yukarıdaki gibi benzer bir şekilde tanımlayabiliriz:

```
@Bean
```

```
@Scope(value = WebApplicationContext.SCOPE_SESSION, proxyMode =
ScopedProxyMode.TARGET_CLASS)
public HelloMessageGenerator sessionScopedBean() {
   return new HelloMessageGenerator();
}
```

bean tanımını basitleştirmek için kullanabileceğimiz özel bir oluşturulmuş açıklama da var:

```
@Bean
@SessionScope
public HelloMessageGenerator sessionScopedBean() {
   return new HelloMessageGenerator();
}
```

Daha sonra **sessionScopedBean** referansıyla bir controller tanımlıyoruz . *Yine* session *için* mesaj alanının değerinin aynı olduğunu göstermek için iki istek çalıştırmamız gerekiyor .

Bu durumda istek ilk kez yapıldığında değer mesajı null olur . Ancak, bir kez değiştirildiğinde, tüm oturum için aynı çekirdeğin aynı örneği döndürüldüğü için sonraki istekler için bu değer korunur.

```
@Controller
public class ScopesController {
   @Resource(name = "sessionScopedBean")
   HelloMessageGenerator sessionScopedBean;

   @RequestMapping("/scopes/session")
   public String getSessionScopeMessage(final Model model) {
      model.addAttribute("previousMessage", sessionScopedBean.getMessage());
      sessionScopedBean.setMessage("Good afternoon!");
      model.addAttribute("currentMessage", sessionScopedBean.getMessage());
      return "scopesExample";
   }
}
```

e. Application Scope

Application scope, bir *ServletContext'in* yaşam döngüsü için bean örneğini oluşturur .

Bu, *singleton* kapsamına benzer, ancak bean'nın scope açısından çok önemli bir fark vardır.

Çekirdekler Application *scope* alındığında, aynı çekirdeğin aynı örneği,
aynı ServletContext içinde çalışan birden çok sunucu uygulaması tabanlı uygulama arasında paylaşılırken , *tekil* kapsamlı çekirdekler yalnızca tek bir uygulama bağlamına göre kapsamlandırılır.

Application scope ile bean'i oluşturalım :

```
@Bean
@Scope(
 value = WebApplicationContext.SCOPE_APPLICATION, proxyMode =
ScopedProxyMode.TARGET_CLASS)
public HelloMessageGenerator applicationScopedBean() {
```

```
    return new HelloMessageGenerator();
}
```

İstek ve *oturum* scopelarına benzer şekilde daha kısa bir sürüm kullanabiliriz:

```
@Bean
@ApplicationScope
public HelloMessageGenerator applicationScopedBean() {
    return new HelloMessageGenerator();
}
```

Şimdi bean iiçin referans veren bir controller oluşturalım:

```
@Controller
public class ScopesController {
    @Resource(name = "applicationScopedBean")
    HelloMessageGenerator applicationScopedBean;

    @RequestMapping("/scopes/application")
    public String getApplicationScopeMessage(final Model model) {
        model.addAttribute("previousMessage", applicationScopedBean.getMessage());
        applicationScopedBean.setMessage("Good afternoon!");
        model.addAttribute("currentMessage", applicationScopedBean.getMessage());
        return "scopesExample";
    }
}
```

Bu durumda, applicationScopedBean içinde bir kez
ayarlandığında , aynı *ServletContext'te* çalışıyor olması koşuluyla, sonraki tüm istekler, oturumlar ve hatta bu bene erişecek farklı servlet uygulamaları için değer *mesajı korunacaktır.*

f. Websocket scope:

Son olarak, *websocket* scope ile bean'i oluşturalım:

```
@Bean
@Scope(scopeName = "websocket", proxyMode = ScopedProxyMode.TARGET_CLASS)
public HelloMessageGenerator websocketScopedBean() {
    return new HelloMessageGenerator();
}
```

İlk erişildiğinde, WebSocket kapsamındaki çekirdekler WebSocket oturum özniteliklerinde depolanır. Bean'in aynı örneği, tüm WebSocket oturumu boyunca o bean'e her erişildiğinde döndürülür .

Tekil davranış sergilediğini de söyleyebiliriz, ancak yalnızca bir **WebSocket** *oturumu* ile sınırlıdır.

---

3 – What does @SpringBootApplication do ?

Bir SpringBoot uygulamasının çalışması için 3 temel anotasyon bulunmaktadır :

- **@SpringBootApplication**
- **@EnableAutoConfiguration**
- **@ComponentScan**

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

@EnableAutoConfiguration anotasyonu ile uygulama bağlılıkları (dependencies) içerisinde belirtilen yapılandırmaları otomatik olarak çalıştırır. Örnek olarak veritabanı ayarlamasını yapar, REST yapılandırmasını hazırlar ve gerekli olan herşey uygulama için hazır hale gelir.

@ComponentScan anotasyonu ise proje içerisinde tanımlanan bütün komponentleri tarar. Bu sayede modeller, repositoryler servisler, controllerlar vs. hepsi kullanıma hazır hale getirilir.

---

4 – Why Spring Boot over Spring?

Spring Boot'un Spring'e göre avantajlarından bazıları şunlardır:

- Gömülü kapsayıcı desteği sağlar
- *Java -jar* komutunu kullanarak bağımsız olarak çalıştırmayı sağlayın
- Harici bir kapsayıcıya dağıtırken olası jar çakışmalarını önlemek için bağımlılıkları hariç tutma seçeneği
- Dağıtım sırasında aktif profilleri belirtme seçeneği
- Entegrasyon testleri için rastgele bağlantı noktası oluşturma
- Maven konfigurasyonunu basitlestirmek icin varsayilan starter bir pom.xml dosyasi ile gelmesi

---

5 – What is Singleton and where to use it ?

Singleton, bir sınıfın yalnızca bir nesneye sahip olmasını sağlayan bir tasarım desenidir.

Bir singleton sınıfı oluşturmak için bir sınıfın aşağıdaki özellikleri uygulaması gerekir:

- `private`Sınıf dışında nesne oluşturmayı kısıtlamak için sınıfın bir yapıcısını oluşturun.
- `private`Tek nesneye başvuran sınıf türünde bir öznitelik oluşturun .

- `public static`Oluşturduğumuz nesneyi oluşturmamıza ve erişmemize izin veren bir yöntem oluşturun. Yöntemin içinde, birden fazla nesne oluşturmamızı kısıtlayan bir koşul oluşturacağız.

Örnek: Java Singleton Sınıf Sözdizimi

```
class SingletonExample {
  private static SingletonExample singleObject;

  private SingletonExample() {

  }

  public static SingletonExample getInstance() {
  }
}
```

Yukarıdaki örnekte,

- `private static SingletonExample singleObject`- sınıfın nesnesine bir referans.
- `private SingletonExample()`- sınıfın dışında nesne oluşturmayı kısıtlayan özel bir kurucu.
- `public static SingletonExample getInstance()`- bu yöntem, sınıfın tek nesnesine başvuruyu döndürür. yöntem beri`statik`, sınıf adı kullanılarak erişilebilir.

Java'da Singleton Kullanımı

Veritabanları ile çalışırken Singleton'lar kullanılabilir. Tüm istemciler için aynı bağlantıyı yeniden kullanırken veritabanına erişmek için bir bağlantı havuzu oluşturmak için kullanılabilirler. Örneğin,

```
class Database {
  private static Database dbObject;

  private Database() {
  }
```

```
  public static Database getInstance() {

    // create object if it's not already created
    if(dbObject == null) {
      dbObject = new Database();
    }

    // returns the singleton object
    return dbObject;
  }

  public void getConnection() {
    System.out.println("You are now connected to the database.");
  }
}

class Main {
  public static void main(String[] args) {
    Database db1;

    // refers to the only object of Database
    db1= Database.getInstance();

    db1.getConnection();
  }
}
```

---

6 – Explain @RestController annotation in Sprint boot?

@RestController İki Anotasyonun Bileşimidir

Bu iki anotasyon @Controller ve @ResponseBody anotasyonudur. Yani şu iki kod aynıdır.

```
@Controller
@ResponseBody
public class MyController { }

@RestController
public class MyRestController { }
```

@ResponseBody ise cevabın JSON olarak gönderimesini sağlar.

Eğer cevabın JSON olarak değil String olarak gitmesini istersek şöyle yaparız.

```
@PostMapping("/reverse")
public String reverseList(@RequestBody String string) {
  ...}
```

RestController Sınıfın Metodlarında Kullanılan Anotasyonlar Nelerdir?

1. Get isteği için @GetMapping veya onun uzun hali olan @RequestMapping ile birlikte kullanılır.
2. Post isteği için @PostMapping veya onun uzun hali olan @RequestMapping ile birlikte kullanılır.
3. Delete isteği için @DeleteMapping veya onun uzun hali olan @RequestMapping ile birlikte kullanılır.
4. Metodlara geçilen parametreler için @PathVariable kullanılır.

RestController'da Kullanılan Metod Parametreleri

Metod parametresi olan nesneler

- controller altındaki **dto/request** ve **dto/response** dizinlerinde saklayabiliriz.

- bir projede **incoming** ve **outgoing** şeklinde kullanmıştık ancak bence güzel olmamıştı

RestController İçin Path Nasıl Verilir
RestController'a bir path vermek gerekir. Eğer vermezsek controller'ın path'i boş string kabul edilir.

Dolayısıyla base path **"api"** ise ve sınıfımızda get için **"status"** metodu varsa şöyle erişebiliriz.
http://localhost:8080/status

Örnek
RestController'a bir path vermek için şöyle yaparız

```
@RestController
@RequestMapping("/api")
public class HelloWorldController {

 //URI: http://localhost:8080/api/hello
 @RequestMapping(value = "/hello", method = RequestMethod.GET)
 public ResponseEntity<String> get() {
  return new ResponseEntity<>("Hello World", HttpStatus.OK);
 }

}
```

value Alanı

@RestController anotasyonu sınıfı Spring Bean haline getirir. Bu alana verilen değer bean'in ismi olur.

---

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. Neredeyse hiçbir konfigürasyon yapmadan tek başına bir uygulama oluşturulmasına yardımcı olur.

Oto konfigürasyon Spring Boot için özel bir özelliktir.

Spring kütüphanesinin yararları

- Spring kütüphanesi bir web uygulamasının tüm katmanlarına uygulanabilir.
- Gevşek bağlılık (Loose Coupling) ve kolay test edilebilirlik sağlar.
- XML ve Annotation konfigürasyonlarını destekler.
- Singleton ve Factory sınıflarının ortadan kaldırılması için gerekli yeteneğe sahiptir.
- Bildirimsel (Declarative) programlamayı destekler.

Spring Boot'un yararları

- Bağımsız (stand-alone) uygulamalar oluşturur.
- Gömülü olarak Tomcat, Jetty veya Undertow birlikte gelir.
- XML konfigürasyonuna ihtiyaç duymaz.
- LOC (Lines of Code) 'u azaltmayı hedefler.
- Başlatması kolaydır.
- Özelleştirme ve yönetim basittir.

Bu nedenle 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.

---

8 – Why to use VCS ?

Bir dosyanın değişik sürümlerini korumak için Sürüm Kontrol Sistemi (VCS) kullanmanız çok akıllıca olacaktır. VCS, dosyaların ya da bütün projenin geçmişteki belirli bir sürümüne erişmenizi, zaman içinde yapılan değişiklikleri karşılaştırmanızı, soruna neden olan şeyde en son kimin değişiklik yaptığını, belirli bir hatayı kimin, ne zaman sisteme dahil ettiğini ve daha başka pek çok şeyi görebilmenizi sağlar. Öte yandan, VCS kullanmak, bir hata yaptığınızda ya da bazı dosyaları yanlışlıkla sildiğinizde durumu kolayca telâfi etmenize yardımcı olur. Üstelik, bütün bunlar size kayda değer bir ek yük de getirmez.

---

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

SOLID, yazılımınızın modüler ve bakımı, anlaşılması, hata ayıklaması ve yeniden düzenlemesi kolay olmasını sağlayan yapılandırılmış bir tasarım yaklaşımıdır. SOLID'i takip etmek ayrıca hem geliştirme hem de bakımda zamandan ve emekten tasarruf etmenize yardımcı olur. SOLID, kodunuzun katı ve kırılgan hale gelmesini önleyerek uzun ömürlü yazılımlar oluşturmanıza yardımcı olur.

Single Responsibility Principle	Her sınıf, sistemin tek bir parçasından veya işlevselliğinden sorumlu olmalıdır.
Open-Closed Principle	Yazılım bileşenleri, genişletmeye açık olmalı, ancak değiştirilmeye açık olmamalıdır.
Liskov Substitution Principle	Bir üst sınıfın nesneleri, sistemi bozmadan alt sınıflarının nesneleri ile değiştirilebilir olmalıdır.
Interface Segregation Principle	Hiçbir müşteri, kullanmadığı yöntemlere bağımlı olmaya zorlanmamalıdır.
Dependency Inversion Principle	Yüksek seviyeli modüller düşük seviyeli modüllere bağlı olmamalı, her ikisi de soyutlamalara bağlı olmalıdır.

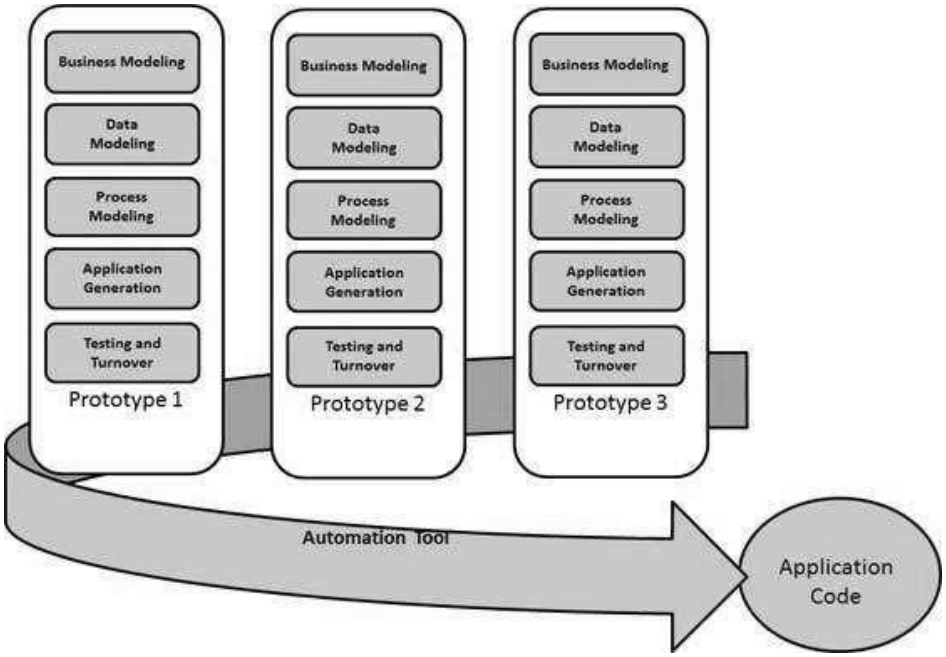
---

10 - What is RAD model ?

RAD (Hızlı Uygulama Geliştirme) modeli , belirli bir planlama gerektirmeden prototip oluşturmaya ve yinelemeli geliştirmeye dayanır. Yazılımın kendisini yazma süreci, ürünü geliştirmek için gerekli planlamayı içerir.

Hızlı Uygulama Geliştirme, atölyeler veya odak grupları aracılığıyla müşteri gereksinimlerinin toplanmasına, prototiplerin müşteri tarafından yinelemeli konsept kullanılarak erken test edilmesine, mevcut prototiplerin (bileşenlerin) yeniden kullanılmasına, sürekli entegrasyona ve hızlı teslimata odaklanır.

Aşağıdaki çizim, RAD Modelini ayrıntılı olarak açıklamaktadır.



---

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

Spring Boot Starters, pom.xml'deki **< dependencies>** bölümünün altına eklenebilen bağımlılık tanımlayıcılarıdır . Farklı Spring ve ilgili teknolojiler için yaklaşık 50'den fazla Spring Boot Starter vardır. Bu başlatıcılar, 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.

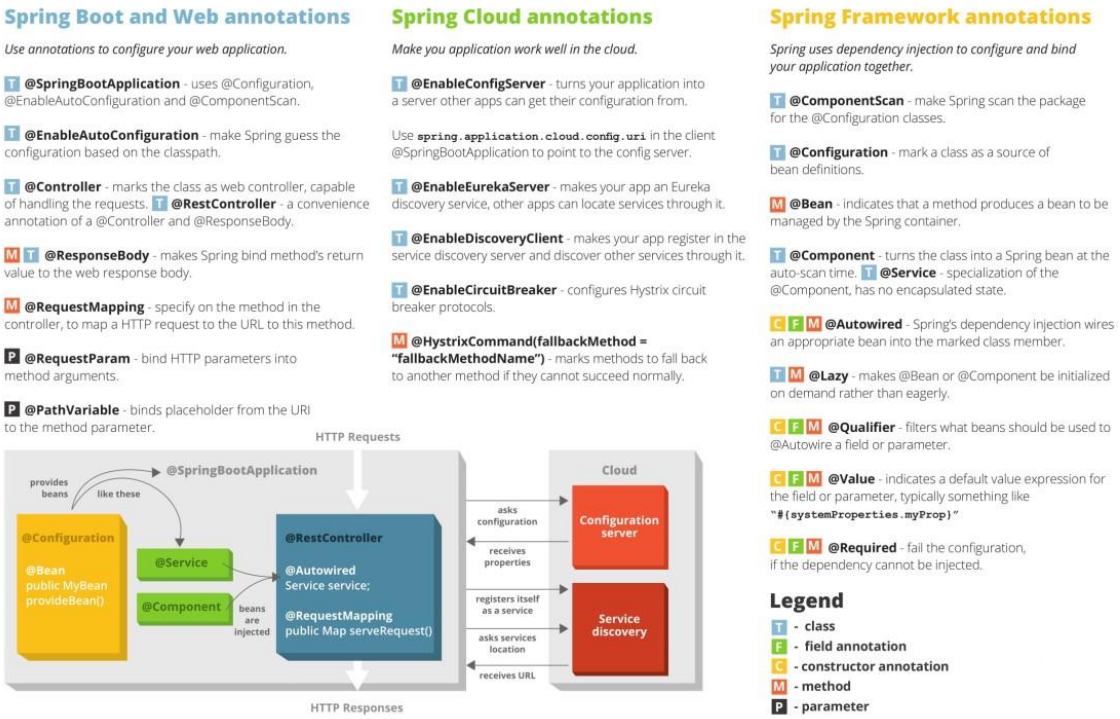
Aşağıdaki tabloda örnek starter yer almaktadır.

İsim	Tanım
spring-boot-starter	Otomatik yapılandırma desteği, günlük kaydı ve YAML dahil olmak üzere çekirdek başlatıcı
spring-boot-starter-activemq	Apache ActiveMQ kullanarak JMS mesajlaşması için Başlatıcı
spring-boot-starter-amqp	Spring AMQP ve Rabbit MQ kullanımı için başlangıç
spring-boot-starter-aop	Spring AOP ve AspectJ ile en-boy odaklı programlama için başlangıç

spring-boot-starter-artemis	Apache Artemis kullanarak JMS mesajlaşması için Başlatıcı
spring-boot-starter-batch	Spring Batch'i kullanmak için başlangıç

---

12 – What are the Spring Boot Annotations?



---

13 – What is Spring Boot dependency management?

Spring Boot, dependency ve yapılandırmayı otomatik olarak yönetir. Spring Boot'un her sürümü, desteklediği dependency bir listesini sağlar.
dependency listesi, Maven ile kullanılabilecek Malzeme Listelerinin (yay-önyükleme dependency) bir parçası olarak mevcuttur. Bu nedenle, yapılandırmamızda dependency sürümünü belirtmemize gerek yoktur.
Spring Boot kendini yönetir. Spring Boot sürümünü güncellediğimizde Spring Boot, tüm dependency tutarlı bir şekilde otomatik olarak yükseltir.

Bağımlılık Yönetiminin Avantajları

- Spring Boot sürümünü tek bir yerde belirterek dependency bilgilerinin merkezileştirilmesini sağlar. Bir sürümden diğerine geçtiğimizde yardımcı olur.
- Spring Boot kitaplıklarının farklı sürümlerinin uyumsuzluğunu önler.
- Yalnızca sürümü belirten bir kitaplık adı yazmamız gerekiyor. Çok modüllü projelerde yardımcı olur.

Not: Spring Boot, gerekirse dependency sürümünün geçersiz kılınmasına da izin verir.

Maven dependency Yönetim Sistemi

Maven projesi, Spring-önyükleme-başlangıç-üst öğesinden aşağıdaki özellikleri devralır:

- Varsayılan Java derleyici sürümü
- UTF-8 kaynak kodlaması
- Spring-boot-dependency-pom'dan bir dependency Bölümünü devralır. Ortak bağımlılıkların sürümünü yönetir. Bu bağımlılıklar için <version> etiketini yok sayar.
- Mantıklı eklenti yapılandırması

---

14 - What is Spring Boot Actuator?

Spring Boot Actuator, Spring Boot Framework'ün bir alt projesidir. Çalışan herhangi bir uygulama hakkında operasyonel bilgileri ortaya çıkarmak için HTTP uç noktalarını kullanır.

Bu kitaplığı kullanmanın temel yararı, üretime hazır uygulamalardan sistem durumu ve izleme ölçümleri almamızdır. Ayrıca, metriklerin toplanması, trafiğin anlaşılması veya veritabanının durumunun bilinmesi Actuator ile son derece kolay hale gelir.

Kaynak:

1. https://gokhana.medium.com/inversion-of-control-ioc-nedir-ve-avantajlar%C4%B1-nelerdir-cf05e42c16e4
2. https://stackoverflow.com/questions/6550700/inversion-of-control-vs-dependency-injection
3. https://stackoverflow.com/questions/130794/what-is-dependency-injection/47090662#47090662
4. https://stackoverflow.com/questions/26884881/difference-between-inversion-of-control-dependency-injection
5. https://github.com/eugenp/tutorials/tree/master/spring-core-2
6. https://www.educative.io/answers/what-are-the-solid-principles-in-java
7. https://www.codingninjas.com/codestudio/library/spring-boot-dependency-management