IOC :

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. (IOC bizim yerimize new leme işlemini yapar sarı renkli yerli Spring bizim için yapar)

IoC kullanmanın avantajlarını şöyle sıralayabiliriz:

- Loosely coupled (bağımlılığı az) sınıflar oluşturma
- Kolay test yazma
- Yönetilebilirlik
- Farklı implementasyonlar arası kolay geçiş

CustomerManager manager = new CustomerManager(new CustomerDal)

DI :

Dependency injection kaba tabir ile bir sınıfın/nesnenin bağımlılıklardan kurtulmasını amaçlayan ve o nesneyi olabildiğince bağımsızlaştıran bir programlama tekniği/prensibidir.Bir sınıfının bağımlı olduğu nesneden bağımsız hareket edebilmesini sağlayabilir ve kod üzerinde olası geliştirmelere karşın değişiklik yapma ihtiyacını ortadan kaldırabilirsiniz.

Bağımlılığa örnek kod:

```
public
class
Car {

          DieselEngine engine;

          public void drive() {
              String engineStart = engine.start();
          }
      }
```

```
 public class DieselEngine {

     public String start() {
         return "DieselEngine started ";
     }

 }
```

DI kullanımına örnek kod :

```
public interface Car {
void drive();
}

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

```
public class DieselEngine implements Engine {

@Override
public String start() {
return "Diesel Engine started.";
}
}
```

```
public class ElectricEngine implements Engine {

    @Override
    public String start() {
        return "Electric Engine started.";
    }
}public class GasolineEngine implements Engine {

    @Override
    public String start() {
        return "Gasoline Engine started.";
    }
}
```

```
public class AutoCar implements Car {

Engine engine;

public AutoCar(Engine engine) {
this.engine = engine;
}

@Override
public void drive() {

String engineStart = engine.start();

}
```

Engine sınıfı constructor ile Autocar sınıfına inject edildi.

```
Engine engine = new ElectricEngine();
Car car = new AutoCar(engine);
car.drive();
```

Spring Bean Scope:

Bean : Oluşturulmuş çalışabilen bir objedir. Spring Framework uygulamamızın omurgasını oluşturan ve Spring IOC container tarafından yönetilen nesnelere *BEAN* denir. Yeniden kullanılabilir objeler olarak kabul düşünebiliriz.

Singleton* **: Her Bean varsayılan olarak singleton olmakla ile birlikte sadece bir kez üretilir. Singleton Design Pattern' deki gibi düşünebiliriz.

```
@Bean
@Scope("singleton")
//@Scope(value = ConfigurableBeanFactory.SCOPE_SINGLETON)
public MyBean myBean() {
   return new MyBean();
```

**Prototype* : Söz konusu olan bean için her bir istek yapıldığında oluşturulur. Her oluşturmada farklı bir instance üretilir.

```
@Bean
@Scope("prototype")
//@Scope(value = ConfigurableBeanFactory.SCOPE_PROTOTYPE)
public MyBean myBean() {
   return new MyBean();
}
```

**Request* : Request bean'i adından da yola çıkarak HTTP isteği geldiğinde oluşturulur. HTTP request düzeyinde etkin bir şekil kapsama alınır.

```
@Bean
@Scope(value = WebApplicationContext.SCOPE_REQUEST, proxyMode =
ScopedProxyMode.TARGET_CLASS)
//@RequestScope
public MyBean myBean() {
   return new MyBean();
}
```

**Session : Session Scope Web Uygulamalarında HTTP isteği geldiğinde oluşturulur. Request Scope'a benzer bir yaklaşım.*

```
@Bean
@Scope(value = WebApplicationContext.SCOPE_SESSION, proxyMode =
ScopedProxyMode.TARGET_CLASS)
//@SessionScope
public MyBean myBean() {
   return new MyBean();
}
```

** Application Scope*

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.

```
@Bean
@Scope(value = WebApplicationContext.SCOPE_APPLICATION, proxyMode =
ScopedProxyMode.TARGET_CLASS)
//@ApplicationScope
public MyBean myBean() {
return new MyBean();
}
```

**WebSocket Scope*

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.

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

@SpringBootApplication:

Birçok Spring Boot geliştiricisi, uygulamalarının otomatik yapılandırma, bileşen taraması kullanmasını ve "uygulama sınıflarında" ekstra yapılandırma tanımlayabilmek ister. Bu üç özelliği etkinleştirmek için tek bir @SpringBootApplication açıklaması kullanılabilir:

@EnableAutoConfiguration: Spring Boot'un otomatik yapılandırma mekanizmasını etkinleştirir

@ComponentScan: uygulamanın bulunduğu pakette @Component taramasını etkinleştirir

@Configuration: bağlamda ekstra bean kaydetmeye veya ek yapılandırma sınıflarını içe aktarmaya izin verir.

@SpringBootApplication annotation'ı, aşağıdaki örnekte gösterildiği gibi, varsayılan öznitelikleriyle @Configuration, @EnableAutoConfiguration ve @ComponentScan kullanımına eşdeğerdir:

```
package com.example.myapplication;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication // @Configuration @EnableAutoConfiguration ve @ComponentScan gibi
public class Application {

	public static void main(String[] args) {
		SpringApplication.run(Application.class, args);
	}

}
```

Spring Boot over Spring:

- Spring Boot spring'in üzerine geliştirilmiş bir frameworkdür.
- Spring XML tabanlı configration ile yönetiliyotdu, Spring Boot'da configration lar application properties ve annotation lar la yönetilmektedir.

- Spring mikro servis mimarisine uyum sağlamakta zorlandığından Spring Boot yaygınlaşmıştır.
- Spring Boot daha genel kullanılan REST API ler üzerinde çalışır,Springde ise Enterprise Edition framework ü daha çok kullanılır.
- Spring ile daha özel alanlarda çalışmalar yaparken, Spring boot ile daha geniş alanlarda geliştirmeler yapabilabilmektedir.
- Spring Boot 'da Anotation'lar ve default configrationlar sayesinde otomatik olarak birçok class ve dependecy'ler configure edilebilmekte
- Spring Boot'da mikro servis mimarisine daha stand alone uygulamalar yazılabildiği için Spring 'e göre daha loose coupled uygulamalar yazalıbilmesini sağlar.
- Sprin Boot ,boiler plate ve gereksiz configurationları elimine eder.
- Spring Boot Jety ve Tomcat gibi gömülü sunucular sunar.
- Spring Boot starter projeler sağlıyor
- Spring Boot bir çok gereksiz boiler plate kısmı ,configration 'ı atarak productivity arttırırken geliştirme süresini azaltmaktadır

Singleton nedir nerede kullanılır ?

Genel olarak bir sınıftan kaç tane obje üretilidğini kontrol etmek istendiğinde singleton yapısı kullanılmış olur.Tek bir nesneye ihtiyaç duyulan durumlarda (veri tabanı bağlantıları, port bağlantıları, dosya işlemleri, loglama işlemleri,bildirimlerde, iş katmanı servislerimizde) kullanılır.

- Singleton pattern bir sınıftan birden fazla instance oluşmasını engeller ve sanal makinada oluşturulan tek instance'ın kullanılmasını garanti eder.
- Singleton sınıfı, sınıfa ait instance'a ulaşmak için global erişim sağlamalıdır.
- Singleton sınıfının constructor'ı private olmalıdır. Böylece o sınınftan yeni instance oluşturmak engellenmiş olur.
- Singleton sınıfın instance'ı içeride private static olarak tutulmalıdır.
- Tutulan instance'ı döndüren bir public static metoda sahip olmalıdır.

Singleton Patterni çeşitli yöntemlerle gerçekleştirilebilir;

- Eager Initialization
- Static Block Initialization
- Lazy Initialization
- Thread Safe Initialization
- Bill Pugh Singleton Implementation

@RestController Annotation

```
import org.springframework.web.bind.annotation.RestController;
```

- @Controller ve @ResponseBody annotationlarının yerine kullanılan, RESTFul web servis methodları yazarken kullanılan işleri kolaylaştıran, Spring MVC 4.0'dan sonra hayatımıza girmiş olan bir annotationdır. @RestController kullandığımız sınıflarda ilgili adreslere cevap verecek methodlar için ayrıca @ResponseBody annotationu eklenmez. Sadece mapping işleminin yapılması yeterli olacaktır.
- Bu anotasyon ilgili sınıftaki bütün metodların birer REST servis noktası olmasını sağlar. Bu anotasyon aslında @Controller ve @ResponseBody anotasyonlarının bileşimi.

- **@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 { }
```

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.

Spring ve SpringBoot arasındaki temel fark nedir?

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.

Versiyon Kontrol Sistemi Neden Kullanılır ?

Versiyon Kontrol Sistemi Nedir?

Kaynak kod yönetimi olarak da adlandırılan versiyon kontrolü, zaman içinde dosyalarda yapılan değişiklikleri yönetmenize ve bu değişiklikleri bir veritabanında depolamanıza olanak tanır.

Versiyon kontrol yazılımını kullanmak, yüksek performanslı yazılımlar geliştirmek ve DevOps süreçlerini doğru şekilde çalıştırmak için büyük faydalar sağlar.

Versiyon kontrol sistemlerinin sağladığı bazı faydalar aşağıdaki şekildedir;

- 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.

Özetle versiyon kontrol sisteminin önemini şu 3 maddeyle açıklayabiliriz;

- Görünürlüğü arttırır.
- Ekiplerin dünya çapında işbirliği yapmasına yardımcı olur.
- Ürün geliştirme sürecini hızlandırır.

SOLID PRENSIPLERI NELERDIR?

S **— Single-responsibility principle**

- Bir sınıf (nesne) yalnızca bir amaç uğruna değiştirilebilir, o da o sınıfa verilen görevdir, yani bir sınıfın(fonksiyona da indirgenebilir) yapması gereken yalnızca bir işi olması gerekir.

```
public class Vehicle {
    public void details() {}
    public double price() {}
    public void addNewVehicle() {}
}
-------------------------------------------------------------------------------
----------------------------------------
public class VehicleDetails {
    public void details() {}
}

public class CalculateVehiclePrice {
    public double price() {}
}

public class AddVehicle {
    public void addNewVehicle() {}
}
```

O— Open-closed principle

- Bir sınıf ya da fonksiyon var olan özellikleri korumalı ve değişikliğe izin vermemelidir. Yani davranışını değiştirmiyor olmalı ve yeni özellikler kazanabilme yeteğine sahip olmalıdır.

- Bu prensibi bir bildirim hizmeti örneği ile anlayalım.

```
public class NotificationService{
    public void sendNotification(String medium) {
        if (medium.equals("email")) {}
    }
}
```

- Burada e-posta dışında yeni bir ortam tanıtmak istiyorsanız, diyelim ki bir cep telefonu numarasına bildirim gönderin, o zaman NotificationService sınıfında kaynak kodunu değiştirmeniz gerekiyor.
  Bunu aşmak için kodunuzu, herkesin özelliğinizi genişleterek yeniden kullanabileceği ve herhangi bir özelleştirmeye ihtiyaç duyarsa sınıfı genişletip üzerine özelliklerini ekleyebileceği şekilde tasarlamanız gerekir.

- Aşağıdaki gibi yeni bir arayüz oluşturabilirsiniz:

```
public interface NotificationService {
    public void sendNotification(String medium);
}
```

- Eposta bildirimi:

```
public class EmailNotification implements NotificationService {
    public void sendNotification(String medium){
        // write Logic using for sending email
    }
}
```

- Mobil Bildirim:

```
public class MobileNotification implements NotificationService {
    public void sendNotification(String medium){
        // write Logic using for sending notification via mobile
    }
}
```

L **— Liskov substitution principle**

- Kodlarımızda herhangi bir değişiklik yapmaya gerek duymadan alt sınıfları, türedikleri(üst) sınıfların yerine kullanabilmeliyiz.

- Rectangle temel sınıfından türetilen bir kare sınıf örneğini ele alalım:

```
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);
    }
}
```

- Rectangle Sınıfta genişlik ve yükseklik ayarı son derece mantıklı görünüyor . Ancak square sınıfında SetWidth() ve SetHeight() anlamlı değildir çünkü birini ayarlamak diğerini ona uyacak şekilde değiştirir.
- Bu durumda, Rectangle temel sınıfını türetilmiş Square sınıfıyla değiştiremeyeceğiniz için Square, Liskov ikame testinde başarısız olur. Square sınıfının ekstra kısıtlamaları vardır, yani yükseklik ve genişlik aynı olmalıdır. Bu nedenle, Rectangle sınıfının Square sınıfıyla değiştirilmesi beklenmeyen davranışlara neden olabilir.

I — Interface segregation principle

- Sorumlulukların hepsini tek bir arayüze(interface) toplamak yerine daha özelleştirilmiş birden fazla arayüz(interface) oluşturmalıyız.

- **Bunu bir araç arayüzü örneği ile anlayalım:**

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

- **Bike, Şimdi bu Araç arayüzünü kullanarak bir sınıf oluşturduğumuzu varsayalım.**

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

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

- **Bisikletin kapıları olmadığı için son işlevi uygulayamıyoruz.**
- **Bunu düzeltmek için, hiçbir sınıfın ihtiyaç duymadığı herhangi bir arabirimi veya yöntemi uygulamak zorunda kalmaması için arabirimleri küçük çoklu arabirimlere ayırmanız önerilir.**

```
public interface Vehicle {
    public void drive();
    public void stop();
    public void refuel();
}
public interface Doors{
    public void openDoors();
}
```

- **İki sınıf oluşturma - Car ve Bike**

```
public class Bike implements Vehicle {
    public void drive() {}
    public void stop() {}
   public void refuel() {}
}
public class Car implements Vehicle, Door {
    public void drive() {}
    public void stop() {}
    public void refuel() {}
    public void openDoors() {}
}
```

D **— Dependency Inversion Principle**

- Sınıflar arası bağımlılıklar olabildiğince az olmalıdır özellikle üst seviye sınıflar alt seviye sınıflara bağımlı olmamalıdır.
- Müşterilerin en sevdikleri kitapları belirli bir rafa koymalarını sağlayan bir kitapçı olduğunu varsayalım.

- Bu işlevi uygulamak için bir Book sınıf ve bir Shelf sınıf oluşturuyoruz. Kitap sınıfı, kullanıcıların raflarda depoladıkları her kitabın incelemelerini görmelerine olanak tanır. Raf sınıfı, raflarına bir kitap eklemelerine izin verecektir. Örneğin,

```
public class Book {
    void seeReviews() {}
}


public class Shelf {
     Book book;
     void addBook(Book book) {}
}
```

- Her şey yolunda görünüyor, ancak high-level Shelf sınıf low-level'e bağlı Book olduğundan, yukarıdaki kod Dependency Inversion Principle'ı ihlal ediyor. Bu, mağaza bizden müşterilerin kendi yorumlarını da raflara eklemelerini sağlamamızı istediğinde açıkça ortaya çıkıyor. Talebi karşılamak için yeni bir UserReview sınıf oluşturuyoruz:

```
public class UserReview{
     void seeReviews() {}
}
```

- Şimdi, Kullanıcı İncelemelerini de kabul edebilmesi için Shelf sınıfını değiştirmeliyiz. Ancak bu, Açık/Kapalı İlkesini de açıkça bozacaktır.
- Çözüm, alt düzey sınıflar (Kitap ve UserReview) için bir soyutlama katmanı oluşturmaktır. Bunu Ürün arabirimini tanıtarak yapacağız, her iki sınıf da onu uygulayacak. Örneğin, aşağıdaki kod kavramı gösterir.

```
public interface Product {
    void seeReviews();
}

public class Book implements Product {
    public void seeReviews() {}
}

public class UserReview implements Product {
    public void seeReviews() {}
}
```

- Artık Raf, uygulamaları yerine Ürün arayüzüne başvurabilir (Kitap ve Kullanıcı İncelemesi). Yeniden düzenlenmiş kod ayrıca, müşterilerin raflarına da koyabilecekleri yeni ürün türlerini (örneğin, Dergi) daha sonra tanıtmamıza da olanak tanır.

```
public class Shelf {
    Product product;
    void addProduct(Product product) {}
    void customizeShelf() {}
}
```

RAD Modeli nedir?

RAD Modeli veya Hızlı Uygulama Geliştirme modeli, herhangi bir özel planlama olmaksızın prototiplemeye dayalı bir yazılım geliştirme sürecidir. RAD modelinde, planlamaya daha az dikkat edilir ve geliştirme görevlerine daha fazla öncelik verilir. Kısa sürede yazılım geliştirmeyi hedefler.

- SDLC RAD modellemesinin aşağıdaki aşamaları vardır
- İş modeli
- Veri Modelleme
- Süreç Modelleme
- Uygulama Üretimi
- Test ve Ciro

Bilginin girdi-çıktı kaynağına ve hedefine odaklanır. Projeleri küçük parçalar halinde teslim etmeye vurgu yapar; daha büyük projeler bir dizi küçük projeye bölünmüştür. RAD modellemenin temel özellikleri, şablonların, araçların, süreçlerin ve kodun yeniden kullanımına odaklanmasıdır.

RAD Modelinin Farklı Aşamaları

Hızlı Uygulama Geliştirme Modelinin aşağıdaki beş ana aşaması vardır

RAD Modeli Aşamaları	RAD Modellemede gerçekleştirilen faaliyetler
İş modeli	▪ Ürün, çeşitli iş kanalları arasındaki bilgi akışı ve dağıtım temelinde tasarlanır.
Veri Modelleme	▪ İş modellemesinden toplanan bilgiler, işletme için önemli olan bir dizi veri nesnesi olarak rafine edilir.
Süreç Modelleme	▪ Veri modelleme aşamasında bildirilen veri nesnesi, bir iş fonksiyonunu uygulamak için gerekli bilgi akışını sağlamak için dönüştürülür.
Uygulama Üretimi	▪ Yazılımın oluşturulması, süreç ve veri modellerinin prototiplere dönüştürülmesi için otomatik araçlar kullanılır.
Test ve Ciro	▪ Prototipler her yineleme sırasında ayrı ayrı test edildiğinden, genel test süresi RAD'de kısalır.

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

Hızlı Uygulama Geliştirme Avantajları ve Dezavantajları

RAD Modelinin Avantajları	RAD Modelinin Dezavantajları
▪ Esnek ve değişikliklere uyarlanabilir	▪ Daha küçük projeler için kullanılamaz
▪ Genel proje riskini azaltmanız gerektiğinde kullanışlıdır	▪ Tüm uygulamalar RAD ile uyumlu değildir
▪ Değişikliklere uyarlanabilir ve esnektir	▪ Teknik risk yüksek olduğunda uygun değildir
▪ Komut dosyaları, üst düzey soyutlamalar ve ara kodlar kullanıldığı için çıktıları aktarmak daha kolaydır	▪ Geliştiriciler yazılımı zamanında teslim etmeye kararlı değilse, RAD projeleri başarısız olabilir
▪ Kod oluşturucular ve kodun yeniden kullanılması nedeniyle, manuel kodlamada bir azalma var	▪ Bir sürümü kısa sürede bitirmek için özelliklerin daha sonraki bir sürüme itildiği zaman kısıtlaması nedeniyle azaltılmış özellikler
▪ Doğadaki prototipleme nedeniyle, daha az kusur olasılığı vardır.	▪ Azaltılmış ölçeklenebilirlik, RAD tarafından geliştirilen bir uygulamanın bir prototip olarak başlayıp bitmiş bir uygulamaya dönüşmesi nedeniyle oluşur.
▪ RAD'deki her aşama, istemciye en yüksek öncelikli işlevselliği sunar	▪ İlerleme ve alışılmış sorunları izlemek zordur, çünkü ne yapıldığını gösteren herhangi bir belge yoktur.
▪ Daha az insanla üretkenlik kısa sürede artırılabilir	▪ Çok yetenekli tasarımcılar veya geliştiriciler gerektirir

Özet

- RAD tam form veya RAD şu anlama gelir: Hızlı Uygulama Geliştirme
- Hızlı Uygulama Geliştirme tanımı: Hızlı Uygulama Geliştirme modeli, herhangi bir özel planlama olmaksızın prototip oluşturmaya dayalı bir yazılım geliştirme sürecidir.

Spring Boot Starter.

Starterlar kısaca uygulamanıza ekleyebileceğiniz bir dizi bağımlılık tanımlayıcısıdır. Sizi kullanmak istediğiniz teknolojilerin her biri için arama yapıp teker teker bağımlılık olarak ekleme zahmetinden kurtarır. **Starter**lar sayesinde ihtiyacınız olan Spring ve ilgili teknolojileri kolayca uygulamanıza ekleyebilirsiniz. Örnek olarak Spring ve JPA kullanmak istiyorsanız spring-boot-starter-data-jpa bağımlılığını projenize eklemeniz yeterli olacaktır.

Staterların isimlendirilmesi

Resmi **starter**lar sprint-boot-starter-* kalıbını kullanırlar. spring-boot ismni resmi Spring Boot startırları için ayrılmıştır. Eğer kendi **starter**ınızı oluşturacaksanız spring-boot şeklinde başlamaması gerekiyor. Kendi oluşturacağınız starter örnek olarak benimprojem-spring-boot-starter şeklinde olabilir.

Resmi Spring Boot Starterları

Aşağıdaki tabloda Spring ekibi tarafından hazırlanmış bazı Spring Boot **starter**larını, açıklamalarını ve pom.xml dosyalarını aşağıdaki tablodan bulabilirsiniz.

Starter İsmi	Açıklama
spring-boot-starter	Oto-konfigirasyon, loglama ve YAML desteği içeren ana starter
spring-boot-starter-data-jdbc	Spring Data JDBC kullanmak için oluşturulmuş starter
spring-boot-starter-data-jpa	Hibernate ile beraber Spring Data JPA kullanmak için oluşturulmuş starter
spring-boot-starter-security	Spring Security kullanmak için oluşturulmuş starter
spring-boot-starter-web	Spring Mvc kullanarak, Tomcat'in varsayılan olarak içinde gömülü geldiği starter

Spring Boot Annotations:

- Annotation kelimesinin Türkçe karşılığını, kimi yerlerde "notasyon" kimi yerlerde ise "anotasyon" ifadesini görebilirsiniz. Peki kısaca nedir bu Annotations denilen kavram.
- Developlement anında IDE veya compiler tarafından yada run-time anında framework tarafından yorumlanan ifadelerdir.

- Bir öğenin tanımını yapar, ne yapması gerektiğini açıklar ve yazılım geliştirme sürecini hem hızlandırır hem de kolaylaştırır.
- Anotasyonlar kodun içerisinde tanımlandıktan sonra, işlevsel hale gelebilmeleri için Spring tarafından 2 defa taranırlar.
- İlk önce yalnızca anotasyonları (spring tarafından yönetilen bean) taranır ve yapılmas gereken görev eşleştirmeleri yapılır.
- İkinci taramada ise anotasyon tanımlamasına göre işlemini yapar.
- Tüm Spring Bean'leri "App Context" yada "Spring Context" (IoC container) adı verilen bir container içinde yaşarlar.
- Anotasyonlar çok güçlü kullanımlardır ve çok farklı şekillerde kullanılabilirler. Birkaç örnek vermek gerekirse:
- Kısıtlamaları ve kullanımı tanımlayan anotasyonlar: @Deprecated, @Override, @NotNull
- Bir öğenin çalışma yapısını belirten notasyonlar: @Entity, @TestCase, @WebService
- Bir öğenin davranışını belirten notasyonlar: @Statefull, @Transaction
- Bir öğenin nasıl işleneceğinin belirten notasyonlar: @Column, @XmlElement
- Özetle, her durumda bir öğeyi tanımlamak ve anlamını netleştirmek için kullanılan notasyon türleri vardır. Burada öğe olarak bahsedilen şey bir değişken olabilir, bir fonksiyon veya bir sınıf olabilir.

Temel Spring Boot Anotasyonları

- @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.

Spring Boot Dependecy Management

Spring Boot, bağımlılıkları ve yapılandırmayı otomatik olarak yönetir. Spring Boot'un her sürümü, desteklediği bağımlılıkların bir listesini sağlar. Bağımlılık listesi, Maven ile kullanılabilecek Malzeme Listelerinin (Spring Boot bağımlılıklarının) bir parçası olarak mevcuttur. Bu yüzden konfigürasyonumuzdaki bağımlılıkların versiyonunu belirtmemize gerek yoktur. Spring Boot kendini yönetir. Spring Boot sürümünü güncellediğimizde Spring Boot, tüm bağımlılıkları tutarlı bir şekilde otomatik olarak yükseltir.

Bağımlılık Yönetimi Avantajları:

- Spring Boot sürümünü tek bir yerde belirterek bağımlılık bilgilerinin merkezileştirilmesini sağlar. Bir sürümden diğerine geçtiğimizde yardımcı olur.
- Spring Boot kütüphanelerinin farklı versiyonlarının uyumsuzluklarını önler.
- Sadece sürüm belirterek bir kitaplık adı yazmamız yeterlidir.Multi modül projelerde çok yardımcı olur.

Spring Boot Actuator nedir:

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