# Component Scanning ve Configuration

@Component/@Service/@Repository/@Controller stereotype'ları, @ComponentScan, @Autowired (field/setter/constructor), @Qualifier ve @Primary.

**Orta** 45 dk

Son güncelleme: 18 Ağustos 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/component-scanning

---

# Component Scanning ve Configuration

Spring IoC Container dersinde bean tanımlamanın tek bir yolunu gördük: `@Configuration` sınıfları içindeki `@Bean` metotları (Java Config). Bu derste ikinci, çok daha yaygın kullanılan yolu -- `@Component` ve onun anlamlı türevlerini ( `@Service` , `@Repository` , `@Controller` ) sınıfların üzerine koyup container'ın onları kendiliğinden bulmasını sağlamayı -- işliyoruz. Bunun yanında, Dependency Injection dersinde saf Java ile elle simüle ettiğimiz field injection'ı gerçek bir container içinde göreceğiz, ve birden fazla bean arasındaki belirsizliği `@Qualifier` / `@Primary` ile çözeceğiz.

## Component Scanning Nedir?

Component scanning, container'ın classpath'i tarayıp `@Component` (ya da ondan türeyen bir annotation) ile işaretlenmiş sınıfları kendiliğinden bulup bean olarak kaydetmesidir -- Java Config'in aksine, hiçbir `@Bean` metodu yazmana gerek kalmaz:

```
// Java Config (Spring IoC Container dersinde gördüğümüz): bean'i sen tanımlarsın.
@Configuration
class AppConfig {
    @Bean
    OrderService orderService() { return new OrderService(); }
}

// Component scanning: sınıfın kendisini işaretlersin, container onu bulur.
@Service
class OrderService { }
```

İkinci versiyonda hiçbir `@Bean` metodu yok -- `@Service` annotation'ı, `OrderService` 'in kendisine "beni bul ve bir bean olarak kaydet" diyor. Container, `@ComponentScan` ile işaretlenmiş paketleri tarayıp bu tür sınıfları kendisi keşfediyor.

## Neden Var?

Java Config'in bir sınırı var: her bean için, container'a "bunu nasıl kuracağını" açıkça söyleyen bir `@Bean` metodu yazman gerekir. Onlarca sınıfın olduğu bir uygulamada, her biri için ayrı bir `@Bean` metodu yazmak hem tekrarlayıcı hem de yeni bir sınıf eklendiğinde unutulması kolay bir adım hâline gelir -- sınıfı yazarsın, ama `AppConfig`'e eklemeyi unutursun, ve bean hiç kaydedilmez.

Component scanning bu sorumluluğu tersine çevirir: bean tanımını, ayrı bir yapılandırma dosyasında değil, **sınıfın kendisinde** tutar. Yeni bir `@Service` yazdığında, tek yapman gereken sınıfın kendisini işaretlemek -- taranan bir paketin içindeyse, container onu otomatik bulur. Bu, özellikle kendi yazdığın (kaynak koduna sahip olduğun) sınıflar için Java Config'den çok daha az tekrar gerektirir; "Component Scanning vs Java Config: Ne Zaman Hangisi?" bölümünde bunun her zaman doğru seçim olmadığı durumları göreceğiz.

## Tarihçe

Component scanning, Spring 2.5 (2007) ile geldi -- o zamana kadar Spring uygulamaları neredeyse tamamen XML tabanlı bean tanımlarına dayanıyordu (Spring IoC Container dersindeki "Tarihçe" bölümünde bahsettiğimiz `ClassPathXmlApplicationContext` döneminden). Aynı sürümde `@Autowired` de tanıtıldı -- bean'leri XML'de elle birbirine bağlamak yerine, container'ın bağımlılıkları tipe göre otomatik bulup enjekte etmesini sağladı.

`@Qualifier`, birden fazla aday olduğunda `@Autowired`'ın hangisini seçeceğini belirtmek için aynı dönemde eklendi. 2009'da JSR-330 standardı (`javax.inject`, bugünkü adıyla `jakarta.inject`), Spring'in kendi annotation'larına paralel, çerçeveden bağımsız eşdeğerlerini (`@Inject`, `@Named`) getirdi -- Spring ikisini de destekler, ama bu projede (ve çoğu Spring kod tabanında) Spring'in kendi annotation'ları tercih edilir. `@ComponentScan` (Java Config ile birlikte, XML'siz kurulum) ise Spring 3.0 (2009) ile geldi.

## @Component: Temel Stereotype

Herhangi bir sınıfı bean yapmanın en temel yolu, üzerine `@Component` koymak ve o sınıfın bir `@ComponentScan` kapsamındaki bir pakette olmasını sağlamaktır:

```
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.stereotype.Component;

// @Component marks a class as a bean the container should manage -- unlike the
// @Bean methods from the Spring IoC Container lesson, there's no factory method
// here at all; the class itself IS the bean definition, discovered by scanning.
@Component
class GreetingProvider {
    String greet(String name) {
        return "Hello, " + name + "!";
    }
}

@Configuration
@ComponentScan
class AppConfig {
}

class ComponentAnnotationExample {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext
(AppConfig.class);

        GreetingProvider provider = context.getBean(GreetingProvider.class);
        System.out.println(provider.greet("Ayse"));
        // Hello, Ayse!

        context.close();
    }
}
```

`GreetingProvider`'ın hiçbir `@Bean` metodu yok, ama yine de `context.getBean(...)` ile bulunabiliyor -- `AppConfig` üzerindeki `@ComponentScan`, `AppConfig`'in bulunduğu paketi (burada default package) tarayıp `@Component` işaretli her sınıfı otomatik kaydediyor.

## Bean Adlandırmasını Özelleştirmek

Spring IoC Container dersindeki "Bean Adlandırma ve Birden Fazla Bean" bölümünde `@Bean` metotlarının isminin, varsayılan olarak metot adı olduğunu görmüştük -- `@Component` için de benzer bir varsayılan var, istersen değiştirebilirsin:

```
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.stereotype.Component;

// Without an argument, @Component's bean name defaults to the class name with
// its first letter lowercased ("customBeanNameExample" style). Passing a
// String changes it explicitly.
@Component("primaryEmailSender")
class EmailSender {
    void send(String message) {
        System.out.println("[email] " + message);
    }
}

@Component
class DefaultNamedSender {
    void send(String message) {
        System.out.println("[default] " + message);
    }
}

@Configuration
@ComponentScan
class AppConfig {
}

class CustomBeanNameExample {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext
(AppConfig.class);

        // The explicit name from @Component("primaryEmailSender") -- "emailSender"
        // (the default that would've come from the class name) does not exist.
        EmailSender sender = (EmailSender) context.getBean("primaryEmailSender");
        sender.send("Custom name resolved successfully");
        // [email] Custom name resolved successfully

        // No name given -- defaults to the class name, lowercased at the start.
        System.out.println(context.containsBean("defaultNamedSender")); // true

        context.close();
    }
}
```

Hiçbir isim vermediğinde (`DefaultNamedSender`), bean adı sınıf adının ilk harfi küçük hâle getirilmiş versiyonudur (`defaultNamedSender`). `@Component("primaryEmailSender")` gibi açıkça bir isim verdiğinde, bu varsayılan tamamen yok sayılır -- bean yalnızca verdiğin isimle bulunabilir.

## @Service, @Repository, @Controller: Anlamlı Stereotype'lar

`@Component`'in kendisi hiçbir katman hakkında bir şey söylemez -- `@Service`, `@Repository` ve `@Controller`, üzerlerine zaten `@Component` konmuş, yalnızca daha anlamlı isimler taşıyan özelleşmiş annotation'lardır:

```
import org.springframework.beans.factory.NoSuchBeanDefinitionException;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.annotation.AnnotationUtils;
import org.springframework.stereotype.Component;
import org.springframework.stereotype.Repository;
import org.springframework.stereotype.Service;

// @Service, @Repository, and @Controller are all meta-annotated with
// @Component -- the container treats them identically for scanning and bean
// registration. The difference is purely semantic (readability, and in
// @Repository's case, one extra feature -- exception translation).
@Service
class OrderService {
    void placeOrder(String item) {
        System.out.println("Order placed: " + item);
    }
}

@Repository
class OrderRepository {
    void save(String item) {
        System.out.println("Saved to database: " + item);
    }
}

class OrderController {
    // Plain class -- deliberately NOT annotated, to contrast with the two above.
}

@Configuration
@ComponentScan
class AppConfig {
}

class StereotypeAnnotationsExample {
    public static void main(String[] args) {
        // Confirms @Service really is @Component underneath: AnnotationUtils
        // walks the same meta-annotation chain Spring's own scanner does.
        boolean serviceIsComponent = AnnotationUtils.findAnnotation(Service.class,
Component.class) != null;
        System.out.println(serviceIsComponent); // true

        AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext
(AppConfig.class);

        context.getBean(OrderService.class).placeOrder("Java 21 Book");
        // Order placed: Java 21 Book
```

```
        context.getBean(OrderRepository.class).save("Java 21 Book");
        // Saved to database: Java 21 Book

        // OrderController was never annotated, so it was never scanned -- this
        // line throws NoSuchBeanDefinitionException.
        try {
            context.getBean(OrderController.class);
        } catch (NoSuchBeanDefinitionException e) {
            System.out.println("Not a bean: " + e.getClass().getSimpleName());
            // Not a bean: NoSuchBeanDefinitionException
        }

        context.close();
    }
}
```

Container açısından `@Service` ile `@Component` arasında tarama/kayıt bakımından hiçbir fark yok -- `AnnotationUtils.findAnnotation(...)` çağrısının `true` dönmesi tam olarak bunu kanıtlıyor. `@Repository`'nin tek pratik ek özelliği, veritabanı kütüphanesine özgü checked exception'ları ( `SQLException` gibi) Spring'in kendi `DataAccessException` hiyerarşisine çevirmesidir -- bu proje JPA kullandığı ve JPA repository'leri farklı bir mekanizmayla (bkz. "Bu Projenin Kendi Sınıfları: Gerçek Bir Component Scanning Örneği") kaydedildiği için bu özelliği doğrudan görmüyoruz, ama elle yazdığın bir DAO sınıfında devreye girer.

## @ComponentScan: Hangi Paketler Taranır?

`@ComponentScan`, container'a **nereye** bakacağını söyler -- parametre vermezsen, `@Configuration` sınıfının kendi paketi (ve alt paketleri) taranır:

```
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.ComponentScan.Filter;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.FilterType;
import org.springframework.stereotype.Component;

@Component
class IncludedBean {
    String describe() {
        return "I was scanned and registered.";
    }
}

@Component
class ExcludedBean {
    String describe() {
        return "I should never be registered.";
    }
}

// In a real project (like this one, where @SpringBootApplication on
// LearningPlatformApplication implicitly scans com.cdurgun.learning and
// everything under it), @ComponentScan's basePackages tells the container
// WHERE to look. Here, since AppConfig and the @Component classes above all
// live in the same (default) package, a bare @ComponentScan is enough to
// find them -- the excludeFilter below is what actually keeps ExcludedBean out.
@Configuration
@ComponentScan(excludeFilters = @Filter(type = FilterType.ASSIGNABLE_TYPE, classes =
ExcludedBean.class))
class AppConfig {
}

class ComponentScanConfigExample {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext
(AppConfig.class);

        System.out.println(context.getBean(IncludedBean.class).describe());
        // I was scanned and registered.

        System.out.println(context.containsBean("excludedBean")); // false

        context.close();
    }
}
```

Bu projenin kendisinde, `LearningPlatformApplication`'daki `@SpringBootApplication` (içinde örtük bir `@ComponentScan` barındırır) `com.cdurgun.learning` paketini ve altındaki her şeyi (`controller`, `service`, `repository`, `config`, `domain`) tarar -- bu yüzden `HomeController`/`TopicController`/`NavigationService` gibi sınıflar hiçbir yerde elle kaydedilmez. Yukarıdaki örnekte `excludeFilters`, belirli bir sınıfı taramadan hariç tutmak için kullanılıyor -- gerçek

projelerde genelde `basePackages` ile hangi paketlerin dahil edileceği (ya da `@SpringBootApplication`'da olduğu gibi, hiçbir şey belirtmeyip yalnızca ana sınıfın paketine güvenmek) tercih edilir.

## Field Injection ile @Autowired (Gerçek Container İçinde)

Dependency Injection dersinin "Field Injection" bölümünde, bir framework'ün `@Autowired` bir alana ne yaptığını, elle reflection kullanarak simüle etmiştik. Şimdi aynı şeyi gerçek bir container'a yaptıralım:

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.stereotype.Component;
import org.springframework.stereotype.Service;

interface NotificationSender {
    void send(String to, String message);
}

@Component
class EmailNotificationSender implements NotificationSender {
    @Override
    public void send(String to, String message) {
        System.out.println("[email to " + to + "] " + message);
    }
}

@Service
class OrderService {
    // In the Dependency Injection lesson's "Field Injection" section, we
    // simulated this by hand with raw reflection (Field.setAccessible +
    // Field.set). Here, a real container does exactly that behind @Autowired --
    // no code of ours calls Field.set anywhere.
    @Autowired
    private NotificationSender notificationSender;

    void placeOrder(String customerContact, String item) {
        notificationSender.send(customerContact, "Your order for '" + item + "' has been
placed.");
    }
}

@Configuration
@ComponentScan
class AppConfig {
}

class AutowiredFieldExample {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext
(AppConfig.class);

        OrderService orderService = context.getBean(OrderService.class);
        orderService.placeOrder("ayse@example.com", "Java 21 Book");
        // [email to ayse@example.com] Your order for 'Java 21 Book' has been placed.

        context.close();
    }
}
```

Kendi kodumuzda hiçbir yerde `Field.setAccessible(true)` ya da `Field.set(...)` yok -- `@Autowired` işaretli `notificationSender` alanı, container tarafından, tam olarak o mekanizmayla dolduruluyor. Dependency Injection dersindeki "Best Practices" bölümünde

söylediğimiz "field injection'dan kaçın" tavsiyesi burada da geçerli -- bu örnek yalnızca mekanizmayı göstermek için var, tercih edilen yol değil.

## Setter ve Constructor ile @Autowired

`@Autowired`, field'ların yanı sıra constructor ve setter metotlarına da konabilir -- Dependency Injection dersindeki üç injection stilinin gerçek bir container içindeki karşılığı budur:

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.stereotype.Component;
import org.springframework.stereotype.Service;

interface NotificationSender {
    void send(String to, String message);
}

@Component
class EmailNotificationSender implements NotificationSender {
    @Override
    public void send(String to, String message) {
        System.out.println("[email to " + to + "] " + message);
    }
}

@Service
class ConstructorInjectedOrderService {
    private final NotificationSender notificationSender;

    // @Autowired is optional here -- with a single constructor, Spring uses it
    // automatically. It's written explicitly to keep the intent visible,
    // matching the Dependency Injection lesson's "Constructor Injection" section.
    @Autowired
    ConstructorInjectedOrderService(NotificationSender notificationSender) {
        this.notificationSender = notificationSender;
    }

    void placeOrder(String customerContact, String item) {
        notificationSender.send(customerContact, "Constructor: order for '" + item + "'
placed.");
    }
}

@Service
class SetterInjectedOrderService {
    private NotificationSender notificationSender;

    @Autowired
    void setNotificationSender(NotificationSender notificationSender) {
        this.notificationSender = notificationSender;
    }

    void placeOrder(String customerContact, String item) {
```

```
        notificationSender.send(customerContact, "Setter: order for '" + item + "'
placed.");
    }
}

@Configuration
@ComponentScan
class AppConfig {
}

class AutowiredConstructorSetterExample {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext
(AppConfig.class);

        context.getBean(ConstructorInjectedOrderService.class).placeOrder("ayse@example.
com", "Java 21 Book");
        // [email to ayse@example.com] Constructor: order for 'Java 21 Book' placed.

        context.getBean(SetterInjectedOrderService.class).placeOrder("ayse@example.com",
"Spring Boot Book");
        // [email to ayse@example.com] Setter: order for 'Spring Boot Book' placed.

        context.close();
    }
}
```

Tek constructor'lı bir sınıfta `@Autowired` yazmak aslında **zorunlu değil** -- Spring bunu otomatik anlıyor (Spring IoC Container dersinin "Bean Tanımlama: @Bean ile Java Config" bölümünde bu noktaya kısaca değinmiştik). Yine de burada açıkça yazıldı, çünkü niyeti okuyan biri için netleştiriyor. Setter'daki `@Autowired` ise zorunlu -- Spring, hangi setter'ın enjeksiyon için kullanılacağını bilemeyeceğinden, işaretlenmemiş bir setter asla otomatik çağrılmaz.

## Birden Fazla Bean: @Qualifier ile Belirsizliği Çözmek

Spring IoC Container dersinin "Bean Adlandırma ve Birden Fazla Bean" bölümünde, iki aynı-tipli bean varken `getBean(Type.class)`'in `NoUniqueBeanDefinitionException` fırlattığını görmüştük. `@Qualifier`, aynı belirsizliği **enjekte edilen bir parametre** seviyesinde çözer:

```
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.stereotype.Component;
import org.springframework.stereotype.Service;

interface NotificationSender {
    void send(String to, String message);
}

@Component("emailSender")
class EmailNotificationSender implements NotificationSender {
```

```
    @Override
    public void send(String to, String message) {
        System.out.println("[email to " + to + "] " + message);
    }
}

@Component("smsSender")
class SmsNotificationSender implements NotificationSender {
    @Override
    public void send(String to, String message) {
        System.out.println("[sms to " + to + "] " + message);
    }
}

@Service
class OrderService {
    private final NotificationSender notificationSender;

    // Two NotificationSender beans exist now -- exactly the ambiguity from the
    // Spring IoC Container lesson's "Bean Naming and Multiple Beans"
    // section, but resolved with an annotation at the injection site instead
    // of a manual getBean(name) call.
    OrderService(@Qualifier("emailSender") NotificationSender notificationSender) {
        this.notificationSender = notificationSender;
    }

    void placeOrder(String customerContact, String item) {
        notificationSender.send(customerContact, "Your order for '" + item + "' has been
placed.");
    }
}

@Configuration
@ComponentScan
class AppConfig {
}

class QualifierExample {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext
(AppConfig.class);

        // Without @Qualifier on OrderService's constructor, this line would
        // have failed at startup with a NoUniqueBeanDefinitionException.
        OrderService orderService = context.getBean(OrderService.class);
        orderService.placeOrder("ayse@example.com", "Java 21 Book");
        // [email to ayse@example.com] Your order for 'Java 21 Book' has been placed.

        context.close();
    }
}
```

`@Qualifier("emailSender")`, Spring'e "bu parametre için, `NotificationSender` tipindeki adaylar arasından ismi tam olarak `emailSender` olanı seç" diyor -- tıpkı elle `context.getBean("emailSender", NotificationSender.class)` çağırmak gibi, ama bunu constructor imzasının kendisinde, deklaratif olarak ifade ediyor.

## @Primary: Varsayılan Aday Belirlemek

`@Qualifier` her enjeksiyon noktasına ayrı ayrı yazmak yerine, adaylardan birini **varsayılan** olarak işaretlemenin bir yolu daha var:

```
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Primary;
import org.springframework.stereotype.Component;
import org.springframework.stereotype.Service;

interface NotificationSender {
    void send(String to, String message);
}

@Component
@Primary
// When more than one candidate exists and the injection site has no
// @Qualifier, @Primary breaks the tie -- this bean wins by default.
class EmailNotificationSender implements NotificationSender {
    @Override
    public void send(String to, String message) {
        System.out.println("[email to " + to + "] " + message);
    }
}

@Component
class SmsNotificationSender implements NotificationSender {
    @Override
    public void send(String to, String message) {
        System.out.println("[sms to " + to + "] " + message);
    }
}

@Service
class OrderService {
    private final NotificationSender notificationSender;

    // No @Qualifier here at all -- @Primary on EmailNotificationSender is
    // enough to resolve the ambiguity.
    OrderService(NotificationSender notificationSender) {
        this.notificationSender = notificationSender;
    }

    void placeOrder(String customerContact, String item) {
        notificationSender.send(customerContact, "Your order for '" + item + "' has been
placed.");
    }
}

@Configuration
@ComponentScan
class AppConfig {
}
```

```
class PrimaryExample {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext
(AppConfig.class);

        OrderService orderService = context.getBean(OrderService.class);
        orderService.placeOrder("ayse@example.com", "Java 21 Book");
        // [email to ayse@example.com] Your order for 'Java 21 Book' has been placed.

        context.close();
    }
}
```

`@Primary` işaretli `EmailNotificationSender`, hiçbir `@Qualifier` belirtilmeyen her enjeksiyon noktasında otomatik seçiliyor. Bu, "çoğu yerde X'i istiyorum, yalnızca birkaç özel yerde Y'yi" durumları için idealdir -- her yere `@Qualifier` yazmak yerine, istisnai yerlere yazman yeterli (bir sonraki bölümde tam olarak bunu göreceğiz).

## @Qualifier ve @Primary Bir Arada Kullanıldığında

İkisi aynı anda kullanıldığında hangisi kazanır? Enjeksiyon noktasındaki **açık** `@Qualifier`, her zaman `@Primary`'nin önüne geçer:

```
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Primary;
import org.springframework.stereotype.Component;
import org.springframework.stereotype.Service;

interface NotificationSender {
    void send(String to, String message);
}

@Component("emailSender")
@Primary
class EmailNotificationSender implements NotificationSender {
    @Override
    public void send(String to, String message) {
        System.out.println("[email to " + to + "] " + message);
    }
}

@Component("smsSender")
class SmsNotificationSender implements NotificationSender {
    @Override
    public void send(String to, String message) {
        System.out.println("[sms to " + to + "] " + message);
    }
}

@Service
class EmailOnlyService {
```

```
    // No @Qualifier -- @Primary wins, EmailNotificationSender is injected.
    private final NotificationSender notificationSender;

    EmailOnlyService(NotificationSender notificationSender) {
        this.notificationSender = notificationSender;
    }

    void run() {
        notificationSender.send("ayse@example.com", "via default (@Primary)");
    }
}

@Service
class SmsOnlyService {
    // An explicit @Qualifier at the injection site always wins over @Primary --
    // @Primary only breaks ties when nothing more specific is asked for.
    private final NotificationSender notificationSender;

    SmsOnlyService(@Qualifier("smsSender") NotificationSender notificationSender) {
        this.notificationSender = notificationSender;
    }

    void run() {
        notificationSender.send("+90 555 000 00 00", "via explicit @Qualifier");
    }
}

@Configuration
@ComponentScan
class AppConfig {
}

class QualifierPrimaryTogetherExample {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext
(AppConfig.class);

        context.getBean(EmailOnlyService.class).run();
        // [email to ayse@example.com] via default (@Primary)

        context.getBean(SmsOnlyService.class).run();
        // [sms to +90 555 000 00 00] via explicit @Qualifier

        context.close();
    }
}
```

`EmailOnlyService`, hiçbir `@Qualifier` belirtmediği için `@Primary` işaretli `EmailNotificationSender`'ı alıyor. `SmsOnlyService` ise açıkça `@Qualifier("smsSender")` istediği için, `@Primary`'nin varlığı hiç önemli değil -- "en spesifik talep kazanır" kuralı burada da geçerli.

## Component Scanning vs Java Config: Ne Zaman Hangisi?

İkisi de bean tanımlamanın geçerli yolları, ama en doğal oldukları durumlar farklı:

- **Component scanning** (`@Component` ve türevleri), **kendi yazdığın** sınıflar için idealdir -- sınıfın kaynak koduna erişimin var, bean tanımını sınıfın kendisiyle birlikte tutmak tekrarı azaltır (bkz. "Neden Var?").
- **Java Config** (`@Bean`), kaynak koduna **erişimin olmayan** sınıflar (üçüncü parti kütüphaneler) ya da constructor'ı bean olmayan parametreler (bir API anahtarı, bir sayı) alan sınıflar için gereklidir -- bunlara annotation koyamazsın.

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.stereotype.Service;

// A class we imagine we do NOT own the source of (a third-party library) --
// or, like this project's own domain classes, something that has no reason
// to know Spring exists at all. There's no @Component here: even if we
// wanted one, we couldn't add it to someone else's source file.
class ThirdPartyMailClient {
    private final String apiKey;

    ThirdPartyMailClient(String apiKey) {
        this.apiKey = apiKey;
    }

    void send(String to, String message) {
        System.out.println("[mail-client key=" + apiKey + "] to " + to + ": " + message);
    }
}

// Our own class -- we DO own this one, so @Service (component scanning) is
// the natural choice.
@Service
class NotificationOrchestrator {
    private final ThirdPartyMailClient mailClient;

    @Autowired
    NotificationOrchestrator(ThirdPartyMailClient mailClient) {
        this.mailClient = mailClient;
    }

    void notifyCustomer(String to, String message) {
        mailClient.send(to, message);
    }
}

@Configuration
@ComponentScan
class AppConfig {
    // ThirdPartyMailClient can only be wired via @Bean -- we can't annotate a
    // class whose source we don't own (or shouldn't couple to Spring), and it
    // needs a constructor argument (an API key) that isn't itself a bean.
    @Bean
    ThirdPartyMailClient thirdPartyMailClient() {
        return new ThirdPartyMailClient("demo-api-key");
```

```
    }
}

class MixedConfigExample {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext
(AppConfig.class);

        context.getBean(NotificationOrchestrator.class).notifyCustomer("ayse@example.com",
"Your order has shipped.");
        // [mail-client key=demo-api-key] to ayse@example.com: Your order has shipped.

        context.close();
    }
}
```

`NotificationOrchestrator` kendi sınıfımız olduğu için `@Service`; `ThirdPartyMailClient` ise hem üçüncü parti bir sınıfı temsil ettiği hem de bir API anahtarı parametresi aldığı için yalnızca `@Bean` ile tanımlanabiliyor. Gerçek uygulamalarda bu iki yaklaşım neredeyse her zaman bir arada kullanılır -- biri diğerinin yerine geçmez.

## Bu Projenin Kendi Sınıfları: Gerçek Bir Component Scanning Örneği

Bu dersteki her mekanizmayı, bu projenin kendi kaynak kodunda görebilirsin: `HomeController` ve `TopicController` `@Controller`; `NavigationService`, `ContentResolver`, `MarkdownService`, `CodeExampleResolver` ise `@Service` ile işaretli -- hepsi `LearningPlatformApplication`'daki `@SpringBootApplication`'ın örtük `@ComponentScan`'i sayesinde otomatik bulunuyor.

İlginç bir istisna: `CourseRepository`, `CategoryRepository` gibi repository arayüzlerinde **hiç `@Repository` annotation'ı yok**. Bunun sebebi, Spring Data JPA'nın farklı bir mekanizma kullanması -- `JpaRepository`'yi extend eden bir arayüz gördüğünde, Spring Data (Spring Boot'un auto-configuration'ı sayesinde) bu arayüz için **çalışma zamanında bir proxy implementasyonu üretir** ve onu bean olarak kaydeder; bu, "Field Injection ile @Autowired" ya da "@Component: Temel Stereotype" bölümlerinde gördüğümüz klasik component scanning'den tamamen ayrı bir yol. Spring Boot Auto-Configuration & Properties dersinde, Spring Boot'un hangi mekanizmaları senin yerine "otomatik" tetiklediğine daha yakından bakacağız.

## Best Practices

- **Kendi sınıfların için component scanning'i, üçüncü parti/parametre gerektiren sınıflar için Java Config'i tercih et** (bkz. "Component Scanning vs Java Config: Ne Zaman Hangisi?") -- ikisini birbirinin yerine kullanmaya çalışmak gereksiz zorlanmaya yol açar.

- `@Service` / `@Repository` / `@Controller`**'ı, sadece** `@Component` **yerine, katmanı netleştirdiği için tercih et** -- kodu okuyan biri, bir sınıfın hangi katmanda olduğunu annotation'a bakarak hemen anlar (bkz. "@Service, @Repository, @Controller: Anlamlı Stereotype'lar").
- **Field injection yerine constructor injection kullan** -- bu, Dependency Injection dersinde işlediğimiz gerekçelerin (test edilebilirlik, `final` alanlar) hepsi gerçek bir container içinde de geçerli (bkz. "Field Injection ile @Autowired (Gerçek Container İçinde)").
- `@Primary`**'yi "çoğunlukla bu" durumları için,** `@Qualifier`**'ı istisnalar için kullan** -- her enjeksiyon noktasına `@Qualifier` yazmak yerine bir varsayılan belirlemek, kod tekrarını azaltır (bkz. "@Primary: Varsayılan Aday Belirlemek").
- `@ComponentScan`**'de neyin dahil/hariç tutulduğunu açık tut** -- geniş, belirsiz bir tarama kapsamı, hangi sınıfların gerçekten bean olduğunu takip etmeyi zorlaştırır (bkz. "@ComponentScan: Hangi Paketler Taranır?").

## Yaygın Hatalar

**1. Bir sınıfı yazıp** `@Component` / `@Service` **eklemeyi unutmak, sonra "neden bean bulunamadı" diye şaşırmak.** Component scanning yalnızca işaretlenmiş sınıfları bulur -- işaretlenmemiş bir sınıf, taranan pakette olsa bile asla bean olmaz (bkz. "@Component: Temel Stereotype").

**2.** `@Service` / `@Repository` / `@Controller`**'ın** `@Component`**'ten farklı bir tarama mekanizması kullandığını sanmak.** Üçü de altında `@Component` taşır, container açısından tamamen eşdeğerdirler (bkz. "@Service, @Repository, @Controller: Anlamlı Stereotype'lar").

**3.** `@ComponentScan`**'in varsayılan olarak tüm classpath'i tarayacağını sanmak.** Parametre verilmezse yalnızca `@Configuration` sınıfının kendi paketi (ve altları) taranır -- farklı bir pakette kalan bir sınıf hiç bulunmaz (bkz. "@ComponentScan: Hangi Paketler Taranır?").

**4. Birden fazla aynı-tipli bean varken hiçbir** `@Qualifier` / `@Primary` **eklememek.** Bu, uygulama başlangıcında `NoUniqueBeanDefinitionException` ile sonuçlanır -- Spring IoC Container dersindeki "Bean Adlandırma ve Birden Fazla Bean" bölümünde gördüğümüz hatanın aynısı (bkz. "Birden Fazla Bean: @Qualifier ile Belirsizliği Çözmek").

**5. Üçüncü parti bir sınıfa (kaynak kodun olmayan)** `@Component` **eklemeye çalışmak.** Bu mümkün değildir -- böyle sınıflar yalnızca bir `@Bean` metoduyla tanımlanabilir (bkz. "Component Scanning vs Java Config: Ne Zaman Hangisi?").

**6. Bu projedeki repository arayüzlerinin** `@Repository` **annotation'ı olmadığını görüp "bean olarak kaydedilmemiş" sanmak.** Spring Data JPA, bunları component scanning'den tamamen ayrı bir mekanizmayla (proxy üretimi) kaydeder (bkz. "Bu Projenin Kendi Sınıfları: Gerçek Bir Component Scanning Örneği").

## Özet, Cheat Sheet ve Terimler Sözlüğü

Component scanning, container'ın `@Component` (ve türevleri) ile işaretlenmiş sınıfları classpath'te bulup kendiliğinden bean olarak kaydetmesidir -- Spring IoC Container dersindeki Java Config'e (`@Bean`) alternatif, çok daha az tekrar gerektiren bir bean tanımlama yolu. Öne çıkan noktalar:

- `@Component`: temel stereotype, sınıfın kendisini bean işaretler; bean adı varsayılan olarak sınıf adının küçük harfli hâli, `@Component("isim")` ile özelleştirilebilir
- `@Service` / `@Repository` / `@Controller`: `@Component`'in anlamlı türevleri, container için birbirinden farksız
- `@ComponentScan`: hangi paket(ler)in taranacağını belirler; parametresiz kullanımda `@Configuration` sınıfının kendi paketi taranır
- `@Autowired`: field, setter ya da constructor'a konabilir; tek constructor'da isteğe bağlı, birden fazla setter'da her birine ayrı ayrı yazılmalı
- `@Qualifier("isim")`: enjeksiyon noktasında, aynı tipteki adaylardan hangisinin isteneceğini açıkça belirtir
- `@Primary`: birden fazla aday varken, `@Qualifier` belirtilmeyen her yerde kullanılacak varsayılan adayı işaretler; açık bir `@Qualifier` her zaman kazanır
- Component scanning kendi sınıfların için, Java Config üçüncü parti/parametreli sınıflar için tercih edilir -- ikisi bir arada kullanılır

Hızlı referans:

```
@Component                          // temel stereotype
@Component("customName")            // özel bean adı
@Service / @Repository / @Controller // anlamlı stereotype'lar (hepsi @Component)

@Configuration
@ComponentScan                      // parametresiz: kendi paketini tarar
// @ComponentScan(basePackages = "com.example")  // belirli paket(ler)
class AppConfig { }

class OrderService {
    @Autowired                      // field injection (tercih edilmez)
    private NotificationSender fieldSender;

    @Autowired                      // constructor injection (tercih edilen)
    OrderService(@Qualifier("emailSender") NotificationSender sender) { }

    @Autowired                      // setter injection
    void setSender(NotificationSender sender) { }
}

@Component
@Primary                            // @Qualifier yoksa varsayılan aday
class EmailSender implements NotificationSender { }
```

### Terimler Sözlüğü

**Component scanning** — Container'ın classpath'i tarayıp `@Component` (ya da türevi) ile işaretlenmiş sınıfları kendiliğinden bulup bean olarak kaydetmesi.

`@Component` — Bir sınıfı, component scanning tarafından bulunacak temel bir bean olarak işaretleyen annotation.

**Stereotype annotation** — `@Component`'in, belirli bir katmanı ifade eden (`@Service`, `@Repository`, `@Controller`) anlamlı türevi; container açısından `@Component`'ten farksızdır.

`@ComponentScan` — Bir `@Configuration` sınıfına, hangi paket(ler)in taranacağını söyleyen annotation; parametresiz kullanımda kendi paketini tarar.

`@Autowired` — Bir alanın, setter'ın ya da constructor'ın, container tarafından otomatik doldurulmasını/çağrılmasını isteyen annotation.

`@Qualifier` — Aynı tipte birden fazla bean adayı olduğunda, bir enjeksiyon noktasında hangisinin isteneceğini isimle açıkça belirten annotation.

`@Primary` — Birden fazla aday arasından, açık bir `@Qualifier` verilmediğinde kullanılacak varsayılan bean'i işaretleyen annotation.

**Java Config** — `@Configuration` sınıfları içindeki `@Bean` metotlarıyla bean tanımlama yaklaşımı; component scanning'in alternatifi.

## Ek: Mini Proje — Çok Kanallı Bildirim Ağ Geçidi

Dependency Injection dersindeki "Çok Kanallı Bildirim Dağıtıcısı" mini projesinde `List<NotificationSender>` enjekte ederek tüm kanallara aynı anda yayın yapmıştık. Bu kez Spring'in bir başka özel durumunu kullanıyoruz: `Map<String, T>` enjekte edildiğinde, Spring bu haritayı **bean adı → bean nesnesi** eşlemesiyle otomatik dolduruyor:

```
import java.util.Map;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.stereotype.Component;
import org.springframework.stereotype.Service;

interface NotificationSender {
    void send(String to, String message);
}

@Component("email")
class EmailNotificationSender implements NotificationSender {
    @Override
    public void send(String to, String message) {
        System.out.println("[email to " + to + "] " + message);
    }
}

@Component("sms")
class SmsNotificationSender implements NotificationSender {
    @Override
    public void send(String to, String message) {
        System.out.println("[sms to " + to + "] " + message);
    }
}

@Service
class NotificationGateway {
    private final Map<String, NotificationSender> sendersByName;

    // Spring has a special case for Map<String, T> parameters: it injects
    // EVERY bean of type T, keyed by bean name -- no @Qualifier, no manual
    // registry needed. NotificationGateway never has to change to learn
    // about a new channel; adding a third @Component is enough.
    @Autowired
    NotificationGateway(Map<String, NotificationSender> sendersByName) {
        this.sendersByName = sendersByName;
    }

    void sendVia(String channel, String to, String message) {
        NotificationSender sender = sendersByName.get(channel);
        if (sender == null) {
            throw new IllegalArgumentException("Unknown channel: " + channel);
        }
        sender.send(to, message);
    }
}

@Configuration
@ComponentScan
class AppConfig {
}
```

```
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

class NotificationGatewayDemo {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext
(AppConfig.class);

        NotificationGateway gateway = context.getBean(NotificationGateway.class);
        gateway.sendVia("email", "ayse@example.com", "Your order has shipped.");
        // [email to ayse@example.com] Your order has shipped.
        gateway.sendVia("sms", "+90 555 000 00 00", "Your order has shipped.");
        // [sms to +90 555 000 00 00] Your order has shipped.

        try {
            gateway.sendVia("push", "ayse@example.com", "Not registered");
        } catch (IllegalArgumentException e) {
            System.out.println("Failed: " + e.getMessage());
            // Failed: Unknown channel: push
        }

        context.close();
    }
}
```

`NotificationGateway`, hangi kanalların var olduğunu hiç bilmiyor -- `sendersByName` haritası, `@Component("email")` ve `@Component("sms")` ile verdiğimiz isimlerle otomatik dolduruluyor. Yeni bir kanal eklemek ( `@Component("push")` gibi) istersen, `NotificationGateway`'in tek satırını bile değiştirmen gerekmez -- tıpkı "Bean Adlandırmasını Özelleştirmek" bölümünde gördüğümüz isim mekanizmasının, bu kez toplu hâlde çalışması gibi.

> Bu, "Bean Adlandırma ve Birden Fazla Bean" (Spring IoC Container dersi) bölümündeki `NoUniqueBeanDefinitionException` probleminin bambaşka bir çözümü: orada belirsizliği tek bir bean seçerek ( `@Qualifier` / `@Primary` ) çözüyorduk, burada ise belirsizliği hiç çözmüyoruz -- tüm adayları, isimleriyle birlikte, aynı anda kullanılabilir hâle getiriyoruz.

## Ek: Mini Proje — Kitap Kataloğu (Repository/Service /Controller Katmanları)

Son mini proje, "Bu Projenin Kendi Sınıfları: Gerçek Bir Component Scanning Örneği" bölümünde bahsettiğimiz üç katmanlı yapıyı (repository/service/controller) küçük ölçekte, bu projenin gerçek mimarisine paralel şekilde kuruyor:

```
import java.util.ArrayList;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.stereotype.Component;
import org.springframework.stereotype.Repository;
```

```
import org.springframework.stereotype.Service;

// A miniature version of this project's own three-layer structure
// (repository/service/controller) -- see "This Project's Own Classes:
// A Real Component Scanning Example" for how NavigationService,
// TopicController, and the JPA repositories are wired the same way in
// practice.
@Repository
class BookRepository {
    private final List<String> books = new ArrayList<>(List.of("Java 21 Book", "Spring Boot
Book"));

    List<String> findAll() {
        return books;
    }

    void save(String title) {
        books.add(title);
    }
}

@Service
class BookService {
    private final BookRepository bookRepository;

    @Autowired
    BookService(BookRepository bookRepository) {
        this.bookRepository = bookRepository;
    }

    List<String> listBooks() {
        return bookRepository.findAll();
    }

    void addBook(String title) {
        bookRepository.save(title);
    }
}

@Component
class BookController {
    private final BookService bookService;

    @Autowired
    BookController(BookService bookService) {
        this.bookService = bookService;
    }

    void printCatalog() {
        for (String title : bookService.listBooks()) {
            System.out.println("- " + title);
        }
    }
}

@Configuration
@ComponentScan
```

```
class AppConfig {
}
```

```
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

class BookCatalogAppDemo {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext
(AppConfig.class);

        BookController controller = context.getBean(BookController.class);
        controller.printCatalog();
        // - Java 21 Book
        // - Spring Boot Book

        // BookRepository is a singleton, so this change is visible to every
        // later call -- exactly the "Bean Scope: Singleton (Default)"
        // behavior from the Spring IoC Container lesson.
        context.getBean(BookService.class).addBook("Reflection Book");
        controller.printCatalog();
        // - Java 21 Book
        // - Spring Boot Book
        // - Reflection Book

        context.close();
    }
}
```

`BookController`, `BookService`'e; `BookService` de `BookRepository`'ye bağımlı -- her katman yalnızca bir altındakini tanıyor, `@Autowired` constructor'larla birbirine bağlanıyor. `BookRepository`'nin (bu projenin gerçek repository'lerinin aksine) burada gerçek bir `@Repository` annotation'ı taşıdığına dikkat et -- bu, Spring Data JPA'nın proxy tabanlı mekanizmasını değil, "@Service, @Repository, @Controller: Anlamlı Stereotype'lar" bölümünde gördüğümüz klasik component scanning'i kullanıyor.

> `BookRepository`'nin `books` listesi, `context.close()` çağrılana kadar bellekte yaşayan bir singleton durumdur -- `BookCatalogAppDemo`'daki `addBook(...)` çağrısının ikinci `printCatalog()` çıktısında görünmesi tam olarak bu yüzden (bkz. Spring IoC Container dersindeki "Bean Scope: Singleton (Varsayılan)"). Gerçek bir uygulamada bu veri elbette bir veritabanında (bu projede olduğu gibi PostgreSQL'de) kalıcı olurdu, bellekte değil.