Component Scanning & Configuration

@Component/@Service/@Repository/@Controller stereotypes, @ComponentScan, @Autowired (field/setter/constructor), @Qualifier, and @Primary.

Intermediate 45 min

Last updated: August 18, 2026
Read online: https://www.learnforgex.com/en/topics/component-scanning

---

Component Scanning & Configuration

In the Spring IoC Container lesson, we saw one way to define beans: `@Bean` methods inside `@Configuration` classes (Java Config). This lesson covers the second, much more common way -- putting `@Component` and its meaningful derivatives (`@Service`, `@Repository`, `@Controller`) on classes and letting the container find them on its own. Along the way, we'll see field injection -- which we simulated by hand in plain Java in the Dependency Injection lesson -- inside a real container, and resolve ambiguity between multiple beans with `@Qualifier` / `@Primary`.

What Is Component Scanning?

Component scanning is the container scanning the classpath and finding classes marked with `@Component` (or an annotation derived from it) on its own, registering them as beans -- unlike Java Config, you never have to write a `@Bean` method:

```
// Java Config (from the Spring IoC Container lesson): you define the bean.
@Configuration
class AppConfig {
    @Bean
    OrderService orderService() { return new OrderService(); }
}

// Component scanning: you mark the class itself, the container finds it.
@Service
class OrderService { }
```

The second version has no `@Bean` method at all -- the `@Service` annotation tells `OrderService` itself "find me and register me as a bean." The container scans the packages marked with `@ComponentScan` and discovers classes like this on its own.

Why Does It Exist?

Java Config has a limit: for every bean, you have to write a `@Bean` method that explicitly tells the container how to build it. In an application with dozens of classes, writing a separate `@Bean` method for each one becomes both repetitive and an easy step to forget when a new class is added -- you write the class, but forget to add it to `AppConfig`, and the bean is never registered.

Component scanning flips this responsibility around: it keeps the bean definition in the class itself, not a separate configuration file. When you write a new `@Service`, all you have to do is mark the class itself -- if it's inside a scanned package, the container finds it automatically. This requires far less repetition than Java Config, especially for classes you wrote yourself (classes whose source you own); "Component Scanning vs. Java Config: Which One, When?" covers cases where this isn't always the right choice.

History

Component scanning arrived with Spring 2.5 (2007) -- until then, Spring applications relied almost entirely on XML-based bean definitions (the `ClassPathXmlApplicationContext` era we mentioned in the Spring IoC Container lesson's "History" section). The same release introduced `@Autowired` -- instead of wiring beans together by hand in XML, it let the container automatically find and inject dependencies by type.

`@Qualifier` was added in the same era, to specify which candidate `@Autowired` should pick when more than one exists. In 2009, the JSR-330 standard (`javax.inject`, known today as `jakarta.inject`) brought framework-independent equivalents (`@Inject`, `@Named`) parallel to Spring's own annotations -- Spring supports both, but this project (and most Spring codebases) prefers Spring's own annotations. `@ComponentScan` (together with Java Config, XML-free setup) arrived with Spring 3.0 (2009).

@Component: The Basic Stereotype

The most basic way to turn any class into a bean is putting `@Component` on it and making sure that class is in a package covered by a `@ComponentScan`:

```
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` has no `@Bean` method at all, yet it can still be found with `context.getBean(...)` -- the `@ComponentScan` on `AppConfig` scans the package `AppConfig` itself lives in (the default package, here) and automatically registers every `@Component`-marked class it finds.

Customizing the Bean Name

In the Spring IoC Container lesson's "Bean Naming and Multiple Beans" section, we saw that a `@Bean` method's name defaults to the method's own name -- `@Component` has a similar default, which you can override if you want:

```
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();
    }
}
```

When you give no name at all (`DefaultNamedSender`), the bean name is the class name with its first letter lowercased (`defaultNamedSender`). When you give an explicit name like `@Component("primaryEmailSender")`, that default is completely ignored -- the bean can only be found under the name you gave it.

@Service, @Repository, @Controller: Meaningful Stereotypes

`@Component` on its own says nothing about which layer a class belongs to -- `@Service`, `@Repository`, and `@Controller` are specialized annotations that already carry `@Component`, just under more meaningful names:

```
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();
    }
}
```

As far as the container is concerned, there's no difference in scanning/registration between `@Service` and `@Component` -- the `AnnotationUtils.findAnnotation(...)` call returning `true` proves exactly that. `@Repository`'s one practical extra feature is translating database-library-specific checked exceptions (like `SQLException`) into Spring's own `DataAccessException` hierarchy -- since this project uses JPA, and JPA repositories are registered through a different mechanism (see "This Project's Own Classes: A Real Component Scanning Example"), we don't see this feature directly here, but it kicks in for a hand-written DAO class.

@ComponentScan: Which Packages Get Scanned?

`@ComponentScan` tells the container **where** to look -- if you give it no arguments, it scans the `@Configuration` class's own package (and its subpackages):

```
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();
    }
}
```

In this project itself, the `@SpringBootApplication` on `LearningPlatformApplication` (which carries an implicit `@ComponentScan`) scans `com.cdurgun.learning` and everything under it (`controller`, `service`, `repository`, `config`, `domain`) -- that's why classes like `HomeController` / `TopicController` / `NavigationService` are never registered by hand anywhere. In the example above, `excludeFilters` is used to keep one specific class out of scanning -- real projects

usually prefer `basePackages` to state which packages are included (or, as `@SpringBootApplication` does, specify nothing and just rely on the main class's own package).

Field Injection with @Autowired (Inside a Real Container)

In the Dependency Injection lesson's "Field Injection" section, we simulated by hand, with raw reflection, what a framework does to an `@Autowired` field. Let's have a real container do the same thing now:

```
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();
    }
}
```

There's no `Field.setAccessible(true)` or `Field.set(...)` anywhere in our own code -- the `@Autowired`-marked `notificationSender` field is filled by the container, using exactly that mechanism. The "avoid field injection" advice from the Dependency Injection lesson's "Best

Practices" section still applies here -- this example exists only to show the mechanism, not as the preferred approach.

Setter and Constructor Injection with @Autowired

`@Autowired` can go on constructors and setter methods too, not just fields -- this is the real-container counterpart of the three injection styles from the Dependency Injection lesson:

```
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();
    }
}
```

Writing `@Autowired` on a class with a single constructor is actually **not required** -- Spring understands it automatically (we touched on this briefly in the Spring IoC Container lesson's "Defining Beans: Java Config with @Bean" section). It's still written explicitly here, because it makes the intent clear to anyone reading the code. `@Autowired` on the setter, on the other hand, is required -- since Spring has no way of knowing which setter is meant for injection, an unmarked setter is never called automatically.

Multiple Beans: Resolving Ambiguity with @Qualifier

In the Spring IoC Container lesson's "Bean Naming and Multiple Beans" section, we saw `getBean(Type.class)` throw `NoUniqueBeanDefinitionException` when two beans of the same type exist. `@Qualifier` resolves that same ambiguity at the level of an **injected parameter**:

```
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")` tells Spring "for this parameter, among the `NotificationSender`-typed candidates, pick the one whose name is exactly `emailSender`" -- just like calling `context.getBean("emailSender", NotificationSender.class)` by hand, but expressed declaratively, right in the constructor's signature.

@Primary: Marking a Default Candidate

Instead of writing `@Qualifier` at every injection site separately, there's another way to mark one of the candidates as the **default**:

```
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();
    }
}
```

`EmailNotificationSender`, marked `@Primary`, gets picked automatically at every injection site that specifies no `@Qualifier`. This is ideal for "I want X almost everywhere, and Y only in a few special places" situations -- instead of writing `@Qualifier` everywhere, you only write it at the exceptions (which is exactly what the next section shows).

When @Qualifier and @Primary Are Used Together

When both are used at once, which one wins? An **explicit** `@Qualifier` at the injection site always beats `@Primary`:

```
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` specifies no `@Qualifier` at all, so it gets the `@Primary`-marked `EmailNotificationSender`. `SmsOnlyService`, on the other hand, explicitly asks for `@Qualifier("smsSender")`, so `@Primary`'s presence doesn't matter at all -- the rule "the most specific request wins" applies here too.

Component Scanning vs. Java Config: Which One, When?

Both are valid ways to define beans, but they're each most natural in different situations:

- **Component scanning** (`@Component` and its derivatives) is ideal for classes **you wrote yourself** -- you have access to the source, and keeping the bean definition with the class itself reduces repetition (see "Why Does It Exist?").
- **Java Config** (`@Bean`) is necessary for classes you **don't have source access to** (third-party libraries), or classes whose constructor takes parameters that aren't beans themselves (an API key, a number) -- you can't put an annotation on those.

```
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` is our own class, so `@Service` is the natural choice; `ThirdPartyMailClient`, being both a stand-in for a third-party class and one that takes an API-key constructor parameter, can only be defined with `@Bean`. In real applications, these two approaches are used together almost all the time -- neither replaces the other.

This Project's Own Classes: A Real Component Scanning Example

You can see every mechanism from this lesson in this project's own source code: `HomeController` and `TopicController` are `@Controller`; `NavigationService`, `ContentResolver`, `MarkdownService`, and `CodeExampleResolver` are `@Service` -- all found automatically thanks to the implicit `@ComponentScan` inside `@SpringBootApplication` on `LearningPlatformApplication`.

One interesting exception: repository interfaces like `CourseRepository` and `CategoryRepository` have **no `@Repository` annotation at all**. The reason is that Spring Data JPA uses a different mechanism -- when it sees an interface extending `JpaRepository`, Spring Data (thanks to Spring Boot's auto-configuration) **generates a proxy implementation for that interface at runtime** and registers it as a bean; this is a completely separate path from the classic component scanning we saw in "Field Injection with @Autowired" or "@Component: The Basic Stereotype". We'll take a closer look at which mechanisms Spring Boot triggers "on your behalf" in the Spring Boot Auto-Configuration & Properties lesson.

Best Practices

- **Prefer component scanning for your own classes, and Java Config for third-party or parameterized ones** (see "Component Scanning vs. Java Config: Which One, When?") -- trying to use one in place of the other leads to unnecessary friction.
- **Prefer `@Service` / `@Repository` / `@Controller` over plain `@Component`, since they make the layer clear** -- anyone reading the code can immediately tell which layer a class belongs to just from the annotation (see "@Service, @Repository, @Controller: Meaningful Stereotypes").

- **Use constructor injection instead of field injection** -- every reason covered in the Dependency Injection lesson (testability, `final` fields) still holds inside a real container (see "Field Injection with @Autowired (Inside a Real Container)").
- **Use `@Primary` for "mostly this one" cases, and `@Qualifier` for the exceptions** -- marking a default reduces repetition compared to writing `@Qualifier` at every injection site (see "@Primary: Marking a Default Candidate").
- **Keep what `@ComponentScan` includes/excludes explicit** -- a broad, vague scanning scope makes it harder to track which classes are actually beans (see "@ComponentScan: Which Packages Get Scanned?").

Common Mistakes

1. Writing a class, forgetting to add `@Component` / `@Service`, and then being surprised "why can't the bean be found." Component scanning only finds marked classes -- an unmarked class is never a bean, even if it sits inside a scanned package (see "@Component: The Basic Stereotype").

2. Assuming `@Service` / `@Repository` / `@Controller` use a different scanning mechanism than `@Component`. All three carry `@Component` underneath -- they're completely equivalent as far as the container is concerned (see "@Service, @Repository, @Controller: Meaningful Stereotypes").

3. Assuming `@ComponentScan` scans the entire classpath by default. With no arguments, it only scans the `@Configuration` class's own package (and subpackages) -- a class sitting in a different package is never found (see "@ComponentScan: Which Packages Get Scanned?").

4. Not adding any `@Qualifier` / `@Primary` when multiple beans of the same type exist. This results in a `NoUniqueBeanDefinitionException` at application startup -- the same error we saw in the Spring IoC Container lesson's "Bean Naming and Multiple Beans" section (see "Multiple Beans: Resolving Ambiguity with @Qualifier").

5. Trying to add `@Component` to a third-party class (one you don't own the source of). This isn't possible -- such classes can only be defined with a `@Bean` method (see "Component Scanning vs. Java Config: Which One, When?").

6. Seeing that this project's repository interfaces have no `@Repository` annotation and assuming "they're not registered as beans." Spring Data JPA registers them through a completely separate mechanism (proxy generation) from component scanning (see "This Project's Own Classes: A Real Component Scanning Example").

Summary, Cheat Sheet, and Glossary

Component scanning is the container finding classes marked with `@Component` (and its derivatives) on the classpath and registering them as beans on its own -- an alternative to the Java Config (`@Bean`) approach from the Spring IoC Container lesson that requires far less repetition. Key points:

- `@Component`: the basic stereotype, marks the class itself as a bean; the bean name defaults to the lowercased class name, customizable with `@Component("name")`
- `@Service` / `@Repository` / `@Controller`: meaningful derivatives of `@Component`, indistinguishable to the container
- `@ComponentScan`: determines which package(s) get scanned; with no arguments, scans the `@Configuration` class's own package
- `@Autowired`: can go on a field, setter, or constructor; optional on a single constructor, must be written on each setter individually
- `@Qualifier("name")`: at an injection site, explicitly states which of the same-typed candidates is wanted
- `@Primary`: marks the default candidate to use at every site with no `@Qualifier`, when multiple candidates exist; an explicit `@Qualifier` always wins
- Component scanning is preferred for your own classes, Java Config for third-party or parameterized ones -- the two are used together

Quick reference:

```
@Component                          // the basic stereotype
@Component("customName")            // a custom bean name
@Service / @Repository / @Controller // meaningful stereotypes (all @Component)

@Configuration
@ComponentScan                      // no arguments: scans its own package
// @ComponentScan(basePackages = "com.example")  // specific package(s)
class AppConfig { }

class OrderService {
    @Autowired                      // field injection (not preferred)
    private NotificationSender fieldSender;

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

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

@Component
@Primary                            // default candidate when no @Qualifier is given
class EmailSender implements NotificationSender { }
```

Glossary

Component scanning — The container scanning the classpath and finding classes marked with `@Component` (or a derivative) on its own, registering them as beans.

`@Component` — The annotation that marks a class as a basic bean to be found by component scanning.

Stereotype annotation — A meaningful derivative of `@Component` (`@Service`, `@Repository`, `@Controller`) that expresses a specific layer; indistinguishable from `@Component` as far as the container is concerned.

`@ComponentScan` — The annotation on a `@Configuration` class that states which package(s) get scanned; with no arguments, scans its own package.

`@Autowired` — The annotation that asks for a field, setter, or constructor to be automatically filled in/called by the container.

`@Qualifier` — The annotation that, at an injection site, explicitly states by name which candidate is wanted, when more than one bean of the same type exists.

`@Primary` — The annotation that marks the default bean to use among multiple candidates, when no explicit `@Qualifier` is given.

Java Config — The approach of defining beans with `@Bean` methods inside `@Configuration` classes; the alternative to component scanning.

Appendix: Mini Project — A Multi-Channel Notification Gateway

In the Dependency Injection lesson's "A Multi-Channel Notification Dispatcher" mini project, we injected a `List<NotificationSender>` to broadcast to every channel at once. This time we use another of Spring's special cases: when a `Map<String, T>` is injected, Spring fills that map automatically with a **bean name → bean instance** mapping:

```
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` has no idea which channels exist -- the `sendersByName` map is filled automatically with the names we gave via `@Component("email")` and `@Component("sms")`. If you want to add a new channel (say, `@Component("push")`), you don't have to change a single line of `NotificationGateway` -- just like the naming mechanism from "Customizing the Bean Name", now working in bulk.

> This is a completely different solution to the "Bean Naming and Multiple Beans" (Spring IoC Container lesson) problem's `NoUniqueBeanDefinitionException`: there, we resolved the ambiguity by picking a single bean (`@Qualifier` / `@Primary`); here, we don't resolve the ambiguity at all -- we make every candidate, along with its name, available at once instead.

Appendix: Mini Project — A Book Catalog (Repository/Service /Controller Layers)

The final mini project builds, at a small scale, the three-layer structure (repository/service /controller) mentioned in "This Project's Own Classes: A Real Component Scanning Example", paralleling this project's real architecture:

```
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` depends on `BookService`; `BookService` depends on `BookRepository` -- each layer only knows the one below it, wired together with `@Autowired` constructors. Notice that `BookRepository` (unlike this project's real repositories) carries a real `@Repository` annotation here -- it uses the classic component scanning we saw in "@Service, @Repository, @Controller: Meaningful Stereotypes", not Spring Data JPA's proxy-based mechanism.

> `BookRepository`'s `books` list is singleton state that lives in memory until `context.close()` is called -- that's exactly why the `addBook(...)` call in `BookCatalogAppDemo` shows up in the second `printCatalog()` output (see the Spring IoC Container lesson's "Bean Scope: Singleton (the Default)"). In a real application, this data would of course be persisted in a database (PostgreSQL, as in this project), not in memory.