Spring Bean Annotations

Here we'll discuss the most common Spring bean annotations used to define different types of beans.

There're several ways to configure beans in a Spring container. We can declare them using **XML configuration**. We can declare beans using the **@Bean** annotation in a configuration class.

Or we can mark the class with one of the annotations from the org.springframework.stereotype package and leave the rest to component scanning.

Component Scanning

Spring can automatically scan a package for beans if component scanning is enabled.

@ComponentScan configures which packages to scan for classes with annotation @Configuration. We can specify the base package names directly with one of the **basePackages** or value arguments (value is an alias for **basePackages**):

```
@Configuration
@ComponentScan(basePackages = "com.spring.training")
class AppConfig {}
```

Also, we can point to classes in the base packages with the basePackageClasses argument:

```
@Configuration
@ComponentScan(basePackageClasses = BookFactory.class)
class AppConfig {}
```

Both arguments are arrays so that we can provide multiple packages for each.

If no argument is specified, **the scanning happens from the same package where the @ComponentScan annotated class is present**.

@ComponentScan leverages the Java 8 repeating annotations feature, which means we can mark a class with it multiple times:

```
@Configuration
@ComponentScan(basePackages = "com.spring.training")
@ComponentScan(basePackageClasses = BookFactory.class)
class AppConfig {}
```

Alternatively, we can use **@ComponentScans** to specify multiple **@ComponentScan** configurations:

```
@Configuration
@ComponentScans({
  @ComponentScan(basePackages = "com.spring.training"),
  @ComponentScan(basePackageClasses = BookFactory.class)
})
class AppConfig {}
```

@Component

@Component is a class level annotation. During the component scan, Spring Framework automatically detects classes annotated with **@Component**.

For example:

```
@Component
class BookFactory {
    // ...
}
```

By default, the bean instances of this class have the same name as the **class name with a lowercase initial**. On top of that, we can specify a different name using the optional value argument of this annotation.

Since @Repository, @Service, @Configuration, and @Controller are all meta-annotations of @Component, they share the same bean naming behavior. Also, Spring automatically picks them up during the component scanning process.

@Repository

DAO or Repository classes usually represent the database access layer in an application, and should be annotated with @Repository:

```
@Repository
class UserDAOImpl {
    // ...
}
```

One advantage of using this annotation is that it **has automatic persistence exception translation enabled.** When using a persistence framework such as **Hibernate**, native exceptions thrown within classes annotated with **@Repository** will be automatically translated into subclasses of Spring's DataAccessExeption.

@Service

The business logic of an application usually resides within the service layer – so we'll use the @Service annotation to indicate that a class belongs to that layer:

```
@Service
public class UserService {
    // ...
}
```

@Controller

@Controller is a class level annotation which tells the Spring Framework that this class serves as a controller in Spring MVC:

```
@Controller
public class UserController {
    // ...
}
```

@Configuration

Configuration classes can contain bean definition methods annotated with **@Bean**:

```
@Configuration
class AppConfig {

    @Bean
    Book myBook() {
        return new Book();
    }

}
```

@Autowired

We can use the **@Autowired** to mark a dependency which Spring is going to resolve and inject. We can use this annotation with a constructor, setter, or field injection.

Constructor injection:

```
class Car {
    Engine engine;

    @Autowired
    Car(Engine engine) {
        this.engine = engine;
    }
}
```

Setter injection:

```
class Car {
    Engine engine;

    @Autowired
    void setEngine(Engine engine) {
        this.engine = engine;
    }
}
```

Field injection:

```
class Car {
    @Autowired
    Engine engine;
}
```

@Autowired has a **boolean argument called required** with a **default value of true**. It tunes Spring's behavior when it doesn't find a suitable bean to wire. When true, an exception is thrown, otherwise, nothing is wired.

Note, that if we use constructor injection, all constructor arguments are mandatory.

Starting with version 4.3, we don't need to annotate constructors with @Autowired explicitly unless we declare at least two constructors.

@Bean

@Bean marks a factory method which instantiates a Spring bean:

```
@Bean
Engine engine() {
    return new Engine();
}
```

Spring calls these methods when a new instance of the return type is required.

The resulting bean has the same name as the factory method. If we want to name it differently, we can do so with the name or the value arguments of this annotation (the argument value is an alias for the argument name):

```
@Bean("engine")
Engine getEngine() {
    return new Engine();
}
```

Note, that all methods annotated with @Bean must be in @Configuration classes.

@Qualifier

We use **@Qualifier along with @Autowired** to provide the bean id or bean name we want to use **in ambiguous situations**.

For example, the following two beans implement the same interface:

```
class Bike implements Vehicle {}

class Car implements Vehicle {}
```

If Spring needs to inject a Vehicle bean, it ends up with multiple matching definitions. In such cases, we can provide a bean's name explicitly using the @Qualifier annotation.

Using constructor injection:

```
@Autowired
Biker(@Qualifier("bike") Vehicle vehicle) {
    this.vehicle = vehicle;
}
```

Using setter injection:

```
@Autowired
void setVehicle(@Qualifier("bike") Vehicle vehicle) {
    this.vehicle = vehicle;
```

```
}
```

Alternatively:

```
@Autowired
@Qualifier("bike")
void setVehicle(Vehicle vehicle) {
    this.vehicle = vehicle;
}
```

Using field injection:

```
@Autowired
@Qualifier("bike")
Vehicle vehicle;
```

@Required

@Required on setter methods to mark dependencies that we want to populate through XML:

```
@Required
void setColor(String color) {
    this.color = color;
}
```

```
<bean class="com.spring.annotations.Bike">
    <property name="color" value="green" />
</bean>
```

Otherwise, BeanInitializationException will be thrown.

@Value

We can use @Value for injecting property values into beans. It's compatible with constructor, setter, and field injection.

Constructor injection:

```
Engine(@Value("8") int cylinderCount) {
    this.cylinderCount = cylinderCount;
}
```

Setter Injection:

```
@Autowired
void setCylinderCount(@Value("8") int cylinderCount) {
    this.cylinderCount = cylinderCount;
}
```

Alternatively:

```
@Value("8")
void setCylinderCount(int cylinderCount) {
    this.cylinderCount = cylinderCount;
}
```

Field Injection:

```
@Value("8")
int cylinderCount;
```

Of course, injecting static values isn't useful. Therefore, we can use **placeholder strings in @Value** to wire **values defined in external sources**, for example, in .properties or .yaml files.

Let's assume the following .properties file:

```
engine.fuelType=petrol
```

We can inject the value of engine.fuelType with the following:

```
@Value("${engine.fuelType}")
String fuelType;
```

We can use @Value even with SpEL