EInnoVator™
Software Engineering School

Spring 4 – Annotation-Driven Dependency-Injection

Jorge Simão, Ph.D.

Content

Application-Configuration with Spring

Spring Framework is a comprehensive Java Framework which supports as core service application configuration by **dependency-Injection**. Spring manages application components by creating and initializing them automatically with suitable dependencies. Spring latest versions, 3.2 and more recently 4.x, introduces several new mechanisms that simplify and expand application configuration with annotations.

Defining Components with Annotations

Java classes are declared as Spring managed components – *beans*, by decorating the class with annotation **@Component**. The name of the bean can be specified in the **value()** attribute of the annotation. If omitted, the unqualified uncapitalized name of the class is used (e.g. **myapp.StockService** is named **stockService**). The component is configured by specifying which members should be automatically initialized by Spring using a dependency-injection annotation. **@Autowired** specifies that a dependency to another bean with matching type should be resolved. **@Autowired** annotation can be applied to any class member, including: fields, property setters and other methods, and (at most) one constructor. Failure to resolve a **@Autowired** dependency is fatal, unless attribute **@Autowired.required()** is set **false**. **@Required** annotation can also be used for similar purpose. Bean instance creation can be deferred until a bean is looked-up or injected in other bean with annotation **@Lazy**.

Dependencies having generic types **Collection<T>** (e.g. **List**, and **Set**), are injected with all beans that have a type matching the type of the collection's elements. Dependencies with generic type **Map<String,T>** are injected with an each entry per matching bean – the entry *key* is the name of the bean. Ordering of bean in collections can be controlled with annotation **@Order**.

» Example: A Spring Managed Component

```
@Component
public class MarketServiceImpl
implements MarketService {
  @Autowired
  private StockService stockService;
  …
}
```

Qualifiers

Dependency-injection by type, as performed with annotation **@Autowired**, leads to ambiguity exceptions when multiple beans of the same type are defined. A simple approach to remove ambiguity is by giving precedence to one of the matching bean with annotation **@Primary**. Alternatively, the annotation **@Qualifier** can be used in a field, setter, or parameter, to restrict the bean candidates for injection, by matching to a bean with specific name or a bean with matching qualifier. Qualifiers can also be used to restrict the beans that are injected in a **Collection** or **Map**. Custom qualified annotations can be defined by using **@Qualifier** as meta-annotation. If a custom qualifier annotation has attributes, matching requires that all attributes are equals.

» Example: A Qualified Bean

```
@Component
@Qualified("us")
public class USMarketService implements
             MarketService { … }
```

» Example: Qualified Dependency-Injection in Field

```
@Component
public class TradingService {
  @Qualified("us")
  private MarketService marketService;
  …
}
```

» Example: Custom Qualifier Annotation

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.METHOD})
@Qualifier
public @interface Region {
  String value();
}
```

» Example: Qualified Injection in Argument

```
@Component
public class TradingService {
```

```
  public TradingService(@Region("us") usMarket,
                        @Region("us") euMarket) { … }
}
```

ApplicationContext API

The API of the dependency-injection sub-system in Spring is spear-headed by the services of an **ApplicationContext** – this is the component that manages the life-cycle of other components. Including by performing these steps:

- Load the bean definitions (e.g. by component-scanning classes with annotations)
- Create instances of beans, and initialize them with suitable dependencies
- Call-back the beans instances to notify them about life-cycles events

When using annotation-based dependency-injection the class **AnnotationConfigApplicationContext** can be used.

» Example: Creating an ApplicationContext

```
try (
  AnnotationConfigApplicationContext appContext =
     new AnnotationConfigApplicationContext(
                                "myorg.myapp")) {
  StockService stockService =
    appContext.getBean(StockService.class);

  List<StockInfo> stocks =
         stockService.getBestPerforming(10);
  …
}
```

Notice that **AnnotationConfigApplicationContext** implements the interface **java.lang.AutoCloseable**, and its being created and used in a *try-with-resources* block.

Component Scan

Spring uses a recursive search (scan) on packages to discover classes annotated with **@Component** (or other stereotype annotations – *see below*). Internally, Spring uses the API of Java **ClassLoader** to implement the scanning, so any class in the *classpath* can be loaded. Component scanning is started from some base package(s) specified in the constructor of the **ApplicationContext**. Additionally, component scan be triggered by finding a component (usually a configuration class) with annotation **@ComponentScan**. The base package(s) can be specified by name in attribute **basePackages()** or **value()**. An alternative type-safe approach is to start scanning from the package of the classes specified in attribute **basePackageClasses()**. If no base package is specified, it is assumed to be the package of the class annotated with **@ComponentScan**. A custom strategy for the default name of components can also be set in attribute **nameGenerator()** as a class implementing interface **BeanNameGenerator.**

Inclusion and exclusion filters can be applied on the set of scanned classes with attributes **includeFilters()** and **excludeFilters()**. The details of each filter is specified with annotation **@ComponentScan.Filter**. Attribute **type()** defines the type of the filter and attribute **pattern()** de filter value. Supported filter types include: *regex* on class name (configure also with attribute **resourcePattern()**), type-level annotations (default), type assignment, AOP point-cut expressions, and custom filters. Specified filters are combined such that at least one inclusion filter should be matched, and no exclusion filter is matched. Defaults filters are also considered (to include **@Component** annotated classes), unless attribute **useDefaultFilters()** is set to **false**. A custom filter is defined as a class implementing interface **TypeFilter**.

» Example: Scanning Components From Named Package

```
@ComponentScan(basePackages="myorg.myapp"))
@Configuration
public class AppConfig { … }
```

» Example: Scanning Components From Class Package

```
@ComponentScan(basePackageClasses=AppConfig.class)
public class AppConfig { … }
```

» Example: Component-Scan with Filters

```
@ComponentScan(basePackages="myorg.myapp",
includeFilters={
  @ComponentScan.Filter(type=FilterType.REGEX,
                        pattern="*Service"),
  @ComponentScan.Filter(type=FilterType.ANNOTATION,
                        value=Component.class)
})
public class AppConfig { … }
```

Java Config - Factory Methods

When components require a more complex configuration than is possible to do with annotations, or when classes can not be annotated (e.g. third-party library classes), factory-methods should be used to create bean instances. Factory-methods are marked with annotation **@Bean**, and should use the Java **new** operator and *property setters* to create and initialize the bean instance. The name of the bean is by default the name of the factory-method, but can be override with attribute **name()**. Multiple name can be provided – as *aliases*. Factory method can be located in any class that defines a Spring managed bean, but are usually defined in configuration classes

annotated with stereotype annotation **@Configuration**. The dependencies of a bean can be declared and injected through the parameters of the factory-method, or by using **@Autowired** in the class where the factory-method is defined. If the dependency is defined in the same configuration class as the factory-method, direct invocation of the factory-method defining the dependency is allowed. However, the scope of dependency bean is only respected if the configuration class is annotated with **@Configuration**. (Spring uses code-generation library CGLib to dynamically generate a sub-class of the configuration class that overwrite factory-methods in a way that preserve the bean scope.)

» Example: Spring Java Configuration Class

```
@Configuration
public class AppConfig {
  @Autowired
  private StockRepository stockRepository;

  @Bean
  StockService stockService() {
    return new StockService(stockRepository);
  }
}
```

» Example: Dependency in Method Parameter

```
@Bean
StockService stockService(StockRepository repo) {
  return new StockService(repo);
}
```

» Example: Explicit Name in Bean

```
@Bean(name="dataSource")
public DataSource dataSourceLocal() { … }
```

Bean instances created by a factory-method also have dependencies injected when their classes has members annotated with **@Autowired**. Additionally, a bean instance can be autowired automatically by finding dependencies by matching the name or type of the property to the dependency. This is enabled by setting attribute **autowire()** in annotation **@Bean**. Configuration can be split across several class by using annotation **@Import**.

» Example: Automatic Dependency Injection by Type

```
@Bean(autowire=Autowire.BY_TYPE)
StockService stockService() {
  return new StockService();
}
```

Environment and Property Sources

Each **ApplicationContext** has associated with it an implicit singleton bean of type **Environment** (since Spring 3.1). The **Environment** provides a unified API to access application settings/properties defined in a variety of ways, including: custom properties files loaded with **@PropertySource**, JVM properties (defined with option **-D**), OS defined environment variables, and web app global parameters. Environment variable can be looked up explicitly or injected using annotation **@Value** annotation with property-place-holder expression **${propName}**. SPEL script expressions are also supported with **#{ … }**.

» Example: Importing a Property File

```
@PropertySource("classpath:app-config.properties")
@Configuration
public class Test { … }
```

» Example: Injecting Environment and Setting Lookup

```
@Autowired
public Environment environment;

@Bean
public DataSource dataSource() {
  return new DriverManagerDataSource(
     environment.getProperty("db.url"));
}
```

» Example: Injection of Setting with @Value

```
@Value("${db.url}")
private String url;

@Bean
public DataSource dataSource(String url) {
  return new DriverManagerDataSource(url);
}
```

» Example: @Value Injection in Method Parameter

```
@Bean
public DataSource dataSource(
                      @Value("${db.url}") String url) {
  return new DriverManagerDataSource(url);
}
```

Bean Scopes

The scope of bean defines the context of existence of its instances (e.g. life-cycle duration). By default, Spring beans are defined with scope **singleton** – meaning that a single instance exist per **ApplicationContext**. Alternative scopes can be specified with annotation **@Scope**. Scope **prototype** defines beans whose instances are created every time they are injected. The names of these two provided scopes are defined as constants in class **BeanDefinition**. On a web environment scopes **session**, **request**, and **application** are also available, and backed-up by a Servlet attribute with corresponding scope. The names of these web scopes are defined as constants in class **WebApplicationContext**. The default strategy to assign scopes to scanned beans can be

overridden by setting attribute **scopeResolver()** of annotation **@ComponentScan**.

When beans of a more volatile (short-lived) scope are injected in bean of longer lasting scope (e.g. *session scoped* bean inject in a *singled scoped* bean), a proxy that dynamically resolves to the correct instance depending on the scope context should be injected. Proxy creation is controlled with attribute **@Scope.proxyMode()**. CGLib proxies are created by default. JDK proxies can be configured with mode **ScopedProxyMode.INTERFACES**. The proxy mode can also be specified for scanned components with attribute **@ComponentScan.scopedProxy()**. However, the default behavior here is not to create proxies (unless the **@Scope** annotation specifies otherwise).

» Example: Defining a Prototype-Scoped Bean

```
@Bean
@Scope(value=BeanDefinition.SCOPE_PROTOTYPE)
public ConfigurableSearchStrategy searchStategy() { … }
```

» Example: Defining a Proxyed Session-Scoped Bean

```
@Bean
@Scope(value=WebApplicationContext.SCOPE_SESSION,
       proxyMode=ScopedProxyMode.INTERFACES)
public StockRepository stockRepo() { … }
```

Table below summarize scopes available out-of-the box and registered automatically. Scopes marked with * are available only in a web environment.

Scope	Description
singleton	Single instance per **ApplicationContext**
prototype	Instance created on-demand
session*	Bean backed by session-scoped attribute
request*	Bean backed by request-scoped attribute
application*	Bean backed by application-scoped attribute

Custom scopes can also be defined with a bean that implements interface **Scope**, and registered with a bean of type **CustomScopeConfigurer**. Additional Spring projects may define further scopes, using this mechanism.

Stereotypes

The basic mechanism to define a Java class as Spring bean is the **@Component** annotation. Other annotations can also be used to define a component - known as stereotype annotations. All annotations, annotated themselves with **@Component** are also recognized by the component-scan algorithm. It is also possible to define custom stereotypes by defining an annotation having **@Component** (or another stereotype) as meta-annotation.

The main purpose of stereotype annotations is two-fold: to categorize components (e.g. for description and selection of components); and to automatically "inherit" further annotations.

Several stereotypes annotations are provided out-of-the box by Spring core Framework. Other Spring projects introduce further stereotype annotations.

» Example: A Spring Managed Component

```
@Component
public class MarketServiceImpl implements MarketService
{ … }
```

» Example: A Repository

```
@Repository
public class JpaStockDao implements StockDao { … }
```

» Example: A Controller

```
@Controller
public class StockController { … }
```

» Example: A Rest Controller

```
@RestController
public class RestStockController { … }
```

» Example: Custom Stereotype

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.METHOD})
@Component
@Transactional
public @interface TransactionalService {}
```

Table below summarizes pre-defined stereotypes (non-core are marked with *).

Annotation	Description
@Component	Most generic stereotype for a component
@Service	Service-layer component
@Repository	Data-Access Layer component
@Controller	Web-Layer Component (for Spring MVC)
@RestContoller	A Rest Web-Service Component (for Spring MVC)
@Configuration	Java Config class
@Endpoint *	Spring Web-Service Component
@MessageEndpoint *	Service-Integration Component

Life-Cycle Events Callback Handlers

A bean can be called back to notify them of life-cycle event. Annotation **@PostConstruct** specifies the *callback* handler method that is called after a bean instance is initialized – i.e. all fields and properties are injected. Use cases include: allocation of resources, and completion of the initialization in a non-trivial custom way. Annotation **@PreDestroy** specifies the *callback* method that notifies that the bean instance is about to be destroyed. For singleton beans, the timing of destruction matches the shutdown of the container (or when method **ApplicationContext.reset()** is called). For non-singleton beans, the destruction matches the shutdown of a bean scope context (e.g. when a **Session** is closed for a *session scoped* bean). Prototype scope bean are not called back, since the framework does not keep track of them. When a parent and child bean class defines callback methods, both the methods in the parent and child class are called. Callback method should not have parameters, and should return **void**. In factory-methods annotated with **@Bean**, attribute **initMethod()** can also be used to define an initialization callback (as alternative or in addition to **@PostConstruct**). Likewise, attribute **destroyMethod()** can be used to define an initialization callback (as alternative or in addition to **@PreDestroy**). (Implementing interfaces **InitializingBean** and/or **DisposableBean** is another way to define life-cycle callbacks, but it is not a recommended approach as it couples application components to Spring.)

» Example: Post-Construct and Pre-Destroy Callbacks

```
public class DataServiceImpl implements DataService {
  Cache cache;

  @PostContruct
  public void init() { cache.put(...); }

  @PreDestory
  public void destroy() { cache.close(); }
}
```

» Example: Callbacks in @Bean Factory-Method

```
@Bean(initMethod="init", destroyMethod="destroy")
public DataService dataService() { … }
```

Profiles

Spring beans can be conditionally enabled based on the activation of (at least) one of the *profiles* for which it is defined. Annotation **@Profile** is used to define the profiles in which a bean in defined. This is useful when same type&role beans have different implementation in different application deployments scenarios (e.g. development vs. production). The set of enabled profiles is defined by the value of environment variable **spring.profiles.active**. Alternatively, it can be set programmatically using the API of the **Environment** bean associated with the **ApplicationContext**. Custom profile annotations can be defined using **@Profile** as meta-annotation.

» Example: Profiles in Configuration Class

```
@Profile("dev")
@Configuration
public class AppConfigDev {
  @Bean
  public DataSource dataSource() {
    return new EmbeddedDatabaseBuilder()
    .setType(EmbeddedDatabaseType.HSQL)
    .setName("mydb")
    .addScript("classpath:" + SQLDIR + "schema.sql")
    .addScript("classpath:" + SQLDIR + "test-data.sql")
    .build();
  }
}
```

» Example: Profiles in Bean Factory Methods

```
@Configuration
public class AppConfig {

  @Profile("dev")
  @Bean(name="dataSource")
  public DataSource dataSourceDev() {
    return new EmbeddedDatabaseBuilder().….build();
  }

  @Profile("prod")
  @Bean(name="dataSource")
  public DataSource dataSourceProd()
  throws NamingException {
    Context context = new InitialContext();
    return (DataSource)
       context.lookup("java:comp/env/jdbc/datasource");
  }
}
```

» Example: Activate Profile Programmatically

```
AnnotationConfigApplicationContext context = new
  AnnotationConfigApplicationContext();
context.getEnvironment().setActiveProfiles("dev");
context.register(AppConfig.class);
context.refresh();
```

» Example: Defining Custom Profile Annotation

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.METHOD})
@Profile("jdbc")
public @interface Jdbc {}
```


Conditional Beans

Bean can be conditionally enabled with arbitrary programatically-defined conditions using annotation annotation **@Conditional** (since Spring 4.0). The **value()** attribute is a class that implements interface **Condition**, which is the strategy that decides if a bean should be enabled. (**@Profile** is actually implemented as simply a specific condition.) Method **Condition.match()** takes two input descriptors, including one of the type **AnnotatedTypeMetadata** which allows to perform introspection of types without loading referenced classes and avoiding potential class-loading problems. Custom conditional annotations can also be defined by using **@Conditional** as meta-annotation.

» Example: Conditionally Defined Bean

```
@Conditional(JndiCondition.class)
@Bean(name="dataSource")
public DataSource dataSourceJNDI() throws … {
  return (DataSource) new InitialContext()
    .lookup("java:comp/env/jdbc/datasource");
}
```

» Example: Defining Custom Bean-Enabling Condition

```
public class JndiCondition implements Condition {
  @Override
  public boolean matches(ConditionContext context,
                AnnotatedTypeMetadata typeMetadata) {
    if (Boolean.TRUE.equals(context.getEnvironment()
        .getProperty("jdni.enable", Boolean.class))) {
      return true;
    }
    return false;
  }
}
```

» Example: Defining Custom Conditional Annotation

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.METHOD})
@Conditional(JndiCondition.class)
public @interface Jndi {}
```

» Example: Using Custom Conditional Annotation

```
@Jndi
@Bean(name="dataSource")
public DataSource dataSourceJNDI() throws … { … }
```

Table below summaries the Spring 4 annotations for dependency-injection.

Annotation	Attributes	Description
@Component *(+stereotypes)*	**value()=""**	Define managed component
@Configuration	**value()=""**	Java Config class
@Autowired	**required() =true**	Configured member (injection-point)
@Value	**value()**	Value injection
@Required		Mandatory dependency
@Bean	**value()**	Factory-method
@Qualifier	**value()**	Bean Qualifier
@PostConstruct		Initialization callback
@PreDestroy		Destruction callback
@Profile	**value()**	Bean profile
@Conditional	**value()**	Conditional bean
@Order	**value()**	Bean order
@Lazy		Deferred initialization
@Primary		Higher-Precedence bean
@Import	**value()**	Import configuration classes

JSR-330 & JSR-250 Annotations

JSR-330 dependency-injection annotations are supported since Spring 3.0. Annotation **@javax.inject.Inject** is comparable to **@Autowired**; annotation **@javax.inject.Named** used at type-level takes the role of **@Component**, and used at the parameter or field level takes the role of **@Qualifier**. JSR-330 defines the *prototype* scope as default, although Spring maintains the scope *singleton* as default (i.e. use of JSR-330 annotation **@Singleton** is unnecessary). Annotation **@javax.inject.Scope** is defined in JSR-330 as a mechanism to defined new scopes, but in Spring new scopes are defined by implementing interface **Scope**.

JSR-250 annotation **@javax.annotation.Resource** is also supported (since Spring 2.5) to perform dependency-injection by name at the field-level or in a property setter. The name of the field or property is used as the name to resolve. Attribute **@Resource.value()** can also be used to specify an alternative name. Spring falls-back to bean lookup by type, as with **@Autowired** and **@Inject**, if no bean of matching name is found.

» Example: Dependency-Injection w/ JSR-330 Annotation

```
@Named("marketService")
public class MarketService {
  @Inject
  private StockService stockService;
```


```
}
```

Table below summaries **JSR-330** & **JSR-250** dependency-injection annotations supported by Spring.

Annotation	Spring equiv.	Description
@Inject	**@Autowired**	Injection-Point
@Named	**@Component** **@Qualifier**	Defined Component Qualify Injection-point
@Singleton	**@Scope**(“singleton”)	Singleton scoped bean
@Scope	**Scope** *(interface)*	Define new scope
@Resource	**@Autowired**	Injection-Point
@Priority	**@Order**	Bean order

@Enable* Annotations

In Spring 3.2, several annotation were introduced to enable features from different sub-systems without requiring the use of XML based configuration and importing XML namespaces. Table below summaries these annotations.

Annotation & XML equiv.	Description
@EnableAspectJAutoProxy **<aop:aspectj-autoproxy>**	Enable AOP using AspectJ style pointcuts in @Aspect beans
@EnableTransactionManagement **<tx:annotation-driven>**	Enable AOP based declarative transaction management
@EnableCaching **<cache:annotation-driven>**	Enable AOP based caching
@EnableMBeanExport **<context:mbean-export>**	Enable JMX exporting of @ManagedResource beans
@EnableJms	Register async listener methods annotated with @JmsListener

Resources

- Spring Framework Reference Manual – http://docs.spring.io/spring/docs/4.2.0.BUILD-SNAPSHOT/spring-framework-reference/htmlsingle/
- Spring Framework Project – http://projects.spring.io/spring-framework/
- Spring Framework GitHub Repository – https://github.com/spring-projects/spring-framework

About the Author

Jorge Simão is a software engineer and IT Trainer, with two decades long experience on education delivery both in academia and industry. Expert in a wide range of computing topics, he his an author, trainer, and director (Education & Consulting) at EInnovator. He holds a B.Sc., M.Sc., and Ph.D. in Computer Science and Engineering.

Core Spring Training

Core Spring is **Pivotal**'s official four-day flagship **Spring Framework** training. In this course, students build a Spring-powered Java application that demonstrates the **Spring Framework** and other Spring technologies like Spring AOP and Spring Security in an intensely productive, hands-on setting. Completion of this training prepares participants to take a certification exam and become a **Spring Certified Professional**.
Book now an on-site training for date&location of your choice: **www.einnovator.org/course/core-spring**

Contacts

Training – Bookings & Inquiries
training@einnovator.org

Consultancy – Partnerships & Inquiries
consulting@einnovator.org

General Info
info@einnovator.org