Annotation-based Configuration

Summary

Description

Spring 2.5 can use annotation to inject dependency of the Spring. Although @Autowired annotation provides functions as auto wiring, it provides more precise control and wide usability. Spring 2.5 also adds support for JSR-250 annotations such as @Resource, @PostConstruct, and @PreDestroy. Of course, these options are only available if you are using at least Java 5 (Tiger) and thus have access to source level annotations. Use of these annotations also requires that certain BeanPostProcessors be registered within the Spring container.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:context="http://www.springframework.org/schema/context"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
            http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
            http://www.springframework.org/schema/context
            http://www.springframework.org/schema/context/spring-context-2.5.xsd">

      <context:annotation-config/>

</beans>
```

(AutowiredAnnotationBeanPostProcessor, CommonAnnotationBeanPostProcessor, PersistenceAnnotationBeanPostProcessor, and RequiredAnnotationBeanPostProcessor are registered using above <context:annotation-config/> tag.)

@Required

The @Required annotation is applied to the bean property setter method.

```
public class SimpleMovieLister {

    private MovieFinder movieFinder;

    @Required
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }

    // ...
}
```

This annotation simply indicates that the affected bean property must be populated at configuration time: either through an explicit property value in a bean definition or through autowiring. The container will throw an exception if the affected bean property has not been populated; this allows for eager and explicit failure, avoiding NullPointerExceptions or the like later on.

@Autowired

@Autowired annotation is applied to the “traditional” setter method.

```
public class SimpleMovieLister {

    private MovieFinder movieFinder;

    @Autowired
    public void setMovieFinder(MovieFinder movieFinder) {
```

```
        this.movieFinder = movieFinder;
    }

    // ...
}
```

The annotation may also be applied to methods with arbitrary names and/or multiple arguments:

```
public class MovieRecommender {

    private MovieCatalog movieCatalog;

    private CustomerPreferenceDao customerPreferenceDao;

    @Autowired
    public void prepare(MovieCatalog movieCatalog, CustomerPreferenceDao customerPreferenceDao)
{
        this.movieCatalog = movieCatalog;
        this.customerPreferenceDao = customerPreferenceDao;
    }

    // ...
}
```

The @Autowired annotation may even be applied on constructors and fields:

```
public class MovieRecommender {

    @Autowired
    private MovieCatalog movieCatalog;

    private CustomerPreferenceDao customerPreferenceDao;

    @Autowired
    public MovieRecommender(CustomerPreferenceDao customerPreferenceDao) {
        this.customerPreferenceDao = customerPreferenceDao;
    }

    // ...
}
```

It is also possible to provide *all* beans of a particular type from the ApplicationContext by adding the annotation to a field or method that expects an array of that type:

```
public class MovieRecommender {

    @Autowired
    private MovieCatalog[] movieCatalogs;

    // ...
}
```

The same applies for typed collections

```
public class MovieRecommender {

    private Set<MovieCatalog> movieCatalogs;

    @Autowired
    public void setMovieCatalogs(Set<MovieCatalog> movieCatalogs) {
        this.movieCatalogs = movieCatalogs;
    }
```

```
    // ...
}
```

Even typed Maps may be autowired as long as the expected key type is String. The Map values will contain all beans of the expected type, and the keys will contain the corresponding bean names:

```
public class MovieRecommender {

    private Map<String, MovieCatalog> movieCatalogs;

    @Autowired
    public void setMovieCatalogs(Map<String, MovieCatalog> movieCatalogs) {
        this.movieCatalogs = movieCatalogs;
    }

    // ...
}
```

By default, the autowiring will fail whenever *zero* candidate beans are available; the default behavior is to treat annotated methods, constructors, and fields as indicating *required* dependencies. This behavior can be changed as demonstrated below.

```
public class SimpleMovieLister {

    private MovieFinder movieFinder;

    @Autowired(required=false)
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }

    // ...
}
```

@Autowired may also be used for well-known "resolvable dependencies": the BeanFactory interface, the ApplicationContext interface, the ResourceLoader interface, the ApplicationEventPublisher interface and the MessageSource interface. These interfaces (and their extended interfaces such as ConfigurableApplicationContext or ResourcePatternResolver) will be automatically resolved, with no special setup necessary.

Fine-tuning Annotation-based Autowiring with Qualifiers

Since autowiring by type may lead to multiple candidates, it is often necessary to have more control over the selection process. One way to accomplish this is with Spring's @Qualifier annotation. This allows for associating qualifier values with specific arguments, narrowing the set of type matches so that a specific bean is chosen for each argument.

```
public class MovieRecommender {

    @Autowired
    @Qualifier("main")
    private MovieCatalog movieCatalog;

    // ...
}
```

The @Qualifier annotation can also be specified on individual constructor arguments or method parameters:

```
public class MovieRecommender {

    private MovieCatalog movieCatalog;

    private CustomerPreferenceDao customerPreferenceDao;

    @Autowired
    public void prepare(@Qualifier("main") MovieCatalog movieCatalog, CustomerPreferenceDao
customerPreferenceDao) {
        this.movieCatalog = movieCatalog;
        this.customerPreferenceDao = customerPreferenceDao;
    }

    // ...
}
```

The corresponding bean definitions would look like as follows. The bean with qualifier value "main" would be wired with the constructor argument that has been qualified with the same value.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
        http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-2.5.xsd">

    <context:annotation-config/>

    <bean class="example.SimpleMovieCatalog">
        <qualifier value="main"/>
        <!-- inject any dependencies required by this bean -->
    </bean>

    <bean class="example.SimpleMovieCatalog">
        <qualifier value="action"/>
        <!-- inject any dependencies required by this bean -->
    </bean>

    <bean id="movieRecommender" class="example.MovieRecommender"/>

</beans>
```

For a fallback match, the bean name is considered as a default qualifier value. This means that the bean may be defined with an id "main" instead of the nested qualifier element, leading to the same matching result. However, note that while this can be used to refer to specific beans by name, @Autowired is fundamentally about type-driven injection with optional semantic qualifiers. This means that qualifier values, even when using the bean name fallback, always have narrowing semantics within the set of type matches; they do not semantically express a reference to a unique bean id. Good qualifier values would be "main" or "EMEA" or "persistent", expressing characteristics of a specific component - independent from the bean id.

Qualifiers also apply to typed collections.

@Resource

The Spring utilizes JSR-250 @Resource annotation applied in the field and bean property setter method to support the dependency injection.

@Resource takes a 'name' attribute, and by default Spring will interpret that value as the bean name to be injected.

```
public class SimpleMovieLister {

    private MovieFinder movieFinder;

    @Resource(name="myMovieFinder")
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }
}
```

If no name is specified explicitly, then the default name will be derived from the name of the field or setter method: In case of a field, it will simply be equivalent to the field name; in case of a setter method, it will be equivalent to the bean property name. So the following example is going to have the bean with name "movieFinder" injected into its setter method:

```
public class SimpleMovieLister {

    private MovieFinder movieFinder;

    @Resource
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }
}
```

Similar to @Autowired, @Resource may fall back to standard bean type matches (i.e. find a primary type match instead of a specific named bean) as well as resolve well-known "resolvable dependencies" So the following example will have its customerPreferenceDao field looking for a bean with name "customerPreferenceDao" first, then falling back to a primary type match for the type CustomerPreferenceDao. The "context" field will simply be injected based on the known resolvable dependency type ApplicationContext.

```
public class MovieRecommender {

    @Resource
    private CustomerPreferenceDao customerPreferenceDao;

    @Resource
    private ApplicationContext context;

    public MovieRecommender() {
    }

    // ...
}
```

@PostConstrutor and @PreDestroy

The CommonAnnotationBeanPostProcessor recognizes @Resource annotation and JST-250 *lifecycle* annotation.

```
public class CachingMovieLister {

    @PostConstruct
    public void populateMovieCache() {
        // populates the movie cache upon initialization...
    }
```

```
    @PreDestroy
    public void clearMovieCache() {
        // clears the movie cache upon destruction...
    }
}
```

Reference

- Spring Framework - Reference Document / 3.11. Annotation-based configuration