Annotation-based Configuration

Sang Shin
www.javapassion.com
"Learn with JPassion!"

Disclaimer

- Many slides of this presentation are based on the Spring Framework Reference Documentation
  - http://static.springsource.org/spring/docs/3.0.x/reference/index.html

Topics

- Annotation-based Dependency Injection
  - *@Autowired, @Required*
- Qualifier
  - @Qualifier, Custom qualifier
- JSR 330 (Dependency Injection for Java)
  - @Inject
- JSR 250 (Common Annotations)
  - *@PostConstruct & @PreDestroy, @Resource*
- Java-based Spring configuration (instead of XML configuration file)
  - *@Configuration, @Bean*
- *@Component* and further stereotyped annotations
  - *@Service, @Repository, @Controller*
- Auto scanning

Annotation-based Dependency Injection (DI)

Annotation-based Dependency Injection

- An alternative to XML based dependency injection
  - Bean definitions and wiring can be specified inside the Java class
- You can use both (XML and annotation-based)
  - Annotation-based injection is performed before XML-based injection
  - XML-based injection will override Annotation-based injection

Annotations Introduced in Spring

- Spring 2.0
  - *@Required*
- Spring 2.5
  - *@Autowired*
  - JSR-250 (Common Annotation for Java Platform 1.0) annotations: *@Resource, @PostConstruct, @PreDestroy*
- Spring 3.0
  - JSR 330 (Dependency Injection for Java) annotations: *@Inject, @Qualifier, @Named, and @Provider*
  - *@Configuration, @Bean, @Value*

<context:annotation-config/>

- It automatically register *AutowiredAnnotationBeanPostProcessor*, *CommonAnnotationBeanPostProcessor*, *PersistenceAnnotationBeanPostProcessor*, as well as the *RequiredAnnotationBeanPostProcessor*

```
<?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-3.0.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context-3.0.xsd">

   <context:annotation-config/>

</beans>
```

@Autowired

@Autowired

- Can be used in the Java source code for specifying DI requirement (instead of in XML file)
- Places where *@Autowired* can be used
  - Fields
  - Setter methods
  - Constructor methods
  - Arbitrary methods
- By default, the autowiring fails whenever zero candidate beans are available

@Autowired at Field

```
public class MovieRecommender {

  // @Autowired at the field
  @Autowired
  private MovieCatalog movieCatalog;



  // ...
}
```

@Autowired at Setter method

```
public class SimpleMovieLister {

  private MovieFinder movieFinder;

  // MovieFinder object gets injected automatically
  @Autowired
  public void setMovieFinder(MovieFinder movieFinder) {
      this.movieFinder = movieFinder;
  }

  // ...
}
```

@Autowired at Constructor method

```
public class MovieRecommender {

  private CustomerPreferenceDao customerPreferenceDao;

  // @Autowired at the constructor
  @Autowired
  public MovieRecommender(
      CustomerPreferenceDao customerPreferenceDao) {
      this.customerPreferenceDao = customerPreferenceDao;
  }

  // ...
}
```

@Autowired at arbitrary methods

- You can also apply @Autowired annotation to methods with arbitrary names and/or multiple arguments:

```
public class MovieRecommender {

  private MovieCatalog movieCatalog;
  private CustomerPreferenceDao customerPreferenceDao;

  // MovieCatalog and CustomerPreferenceDao objects are
  // injected automatically
  @Autowired
  public void prepare(MovieCatalog movieCatalog,
                      CustomerPreferenceDao customerPreferenceDao) {
     this.movieCatalog = movieCatalog;
     this.customerPreferenceDao = customerPreferenceDao;
  }

  // ...
}
```

@Required

- The *@Required* annotation applies to bean property setter methods
- It throws an exception it if has not been set – you can check if the field has been set or not without using the field in the business logic code

```
public class SimpleMovieLister {

  @Autowired
  private MovieFinder movieFinder;

  // @Required annotation indicates that the affected bean
  // property must be populated at configuration time, through
  // an explicit property value in a bean definition or through
  // autowiring.
  @Required
  public void setMovieFinder(MovieFinder movieFinder) {
      this.movieFinder = movieFinder;
  }

  // ...
}
```

Lab:

Exercise 1: Autowiring with "@Autowired" annotation

4939_spring3_di_annotation.zip

@Qualifier

Fine-tuning @Autowired with Qualifiers

- Because 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

```
public class MovieRecommender {

  // Among the multiple candidates of MovieCatalog type, select
  // the one that has the bean name “main”.
  @Autowired
  @Qualifier("main")
  private MovieCatalog movieCatalog;

  // ...
}
```

Fine-tuning @Autowired with @Qualifier

- 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;
  }

  // ...
}
```

Qualifier name is usually bean name

```
<?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-3.0.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context-3.0.xsd">

  <context:annotation-config/>

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

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

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

</beans>
```

Qualifier name can be from <qualifier value="xx">

```
<?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-3.0.xsd
     http://www.springframework.org/schema/context
     http://www.springframework.org/schema/context/spring-context-3.0.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>
```

Custom Qualifier

Creating Custom Qualifier Annotation

- You can create your own custom qualifier annotations.
- Simply define an annotation and provide the @Qualifier annotation within your definition:

```
// Create custom qualifier annotation called “Genre”
@Target({ElementType.FIELD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface Genre {

  String value();
}
```

@Autowired with Custom Qualifier

- Then you can provide the custom qualifier annotation on autowired fields and parameters:

```
public class MovieRecommender {

  @Autowired
  @Genre("Action")
  private MovieCatalog actionCatalog;

  private MovieCatalog comedyCatalog;

  @Autowired
  public void setComedyCatalog(
      @Genre("Comedy") MovieCatalog comedyCatalog) {
      this.comedyCatalog = comedyCatalog;
  }

  // ...
}
```

@Autowired with Custom Qualifier

```
<?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-3.0.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context-3.0.xsd">

  <context:annotation-config/>

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

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

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

</beans>
```

Lab:

Exercise 2: Fine-tuning with @Qualifier annotation and custom annotation
4939_spring3_di_annotation.zip

@Inject Annotation from JSR 330 (Dependency Injection for Java)

JSR 330's @Inject

- JSR 330 – Dependency Injection for Java
- JSR 330's @Inject annotation can be used in place of Spring's @Autowired
- JSR 330's @Inject annotation, however, does not have a required property unlike Spring's @Autowired annotation which has a required property to indicate if the value being injected is optional

@JSR 330 Maven Dependency

```
<!-- JSR 330 Dependency Injection for Java -->
<dependency>
    <groupId>javax.inject</groupId>
    <artifactId>javax.inject</artifactId>
    <version>1</version>
</dependency>
```

Lab:

Exercise 3: JSR 330 Annotations - @Inject
4939_spring3_di_annotation.zip

@PostConstruct & @PreDestroy & @Resource from JSR 250 (Common Annotations for Java)

@PostConstruct and @PreDestroy

- Offers an alternative to initialization callbacks and destruction callbacks

```
public class CachingMovieLister {

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

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

@Resource

- Spring also supports injection using the JSR-250 *@Resource* annotation on fields or bean property setter methods. This is a common pattern found in Java EE 5 and Java 6, which Spring supports for Spring-managed objects as well.
- *@Resource* takes a 'name' attribute, and by default Spring will interpret that value as the bean name to be injected. In other words, it follows by-name semantics as demonstrated in this example:

```
public class SimpleMovieLister {

    private MovieFinder movieFinder;

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

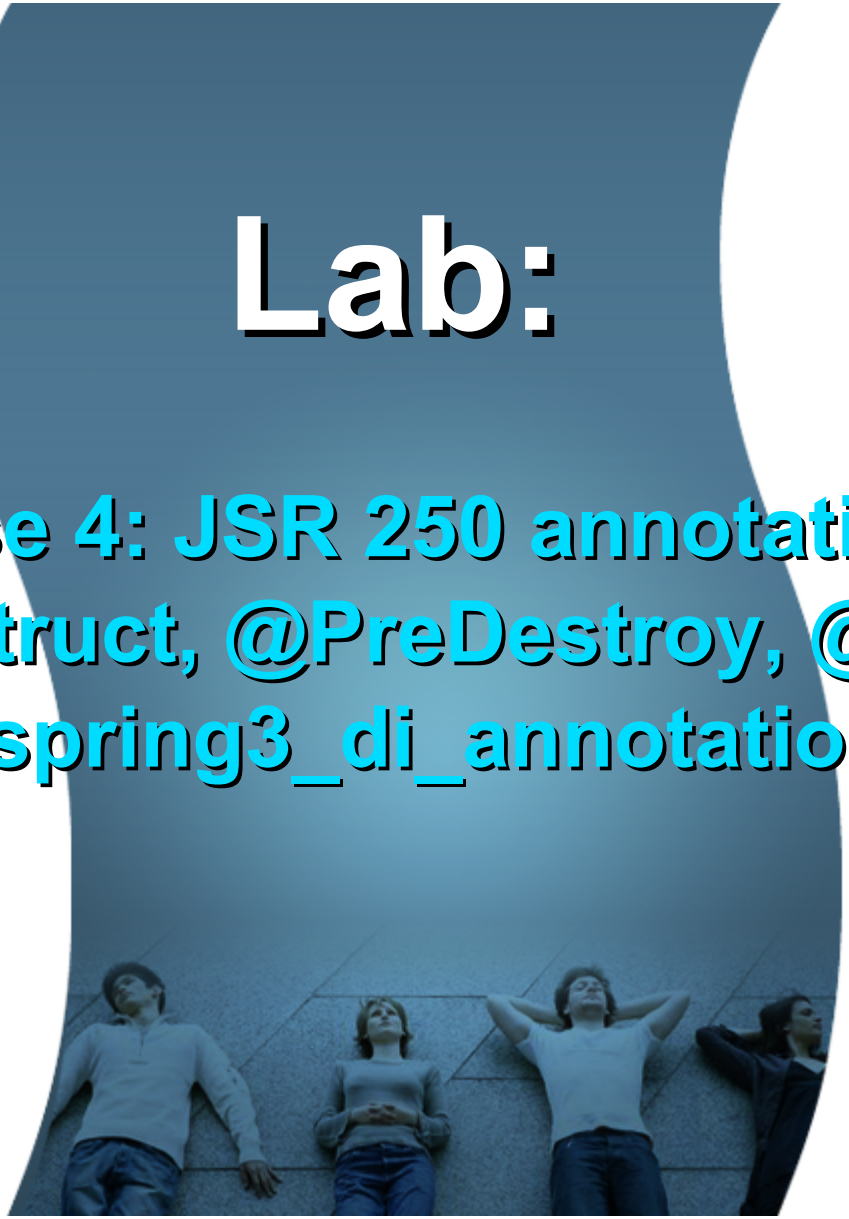
@JSR 250 Maven Dependency

```
<!-- JSR-250 Common Annotations for the Java Platform -->
<dependency>
<groupId>javax.annotation</groupId>
<artifactId>jsr250-api</artifactId>
<version>1.0</version>
</dependency>
```

Lab:

Exercise 4: JSR 250 annotations - @PostConstruct, @PreDestroy, @Resource

4939_spring3_di_annotation.zip



Java-based Container Configuration

@Configuration and @Bean

- Annotating a class with the *@Configuration* indicates that the class can be used by the Spring IoC container as a source of bean definitions (as opposed to from XML file)

```
import com.acme.services.MyServiceImpl;
@Configuration
public class AppConfig {
    // @Bean annotation plays the same role as the
    // <bean/> element in XML configuration
    @Bean
    public MyService myService() {
        return new MyServiceImpl();
    }
}
```

The above is the same as

```
<beans>
  <bean id="myService"
            class="com.acme.services.MyServiceImpl"/>
</beans>
```

AnnotationConfigApplicationContext

- Like Spring XML files are used as input when instantiating a *ClassPathXmlApplicationContext*, *@Configuration* classes may be used as input when instantiating an *AnnotationConfigApplicationContext*.

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

    MyService myService = ctx.getBean(MyService.class);
    myService.doStuff();
}
```

@Configuration and @Bean

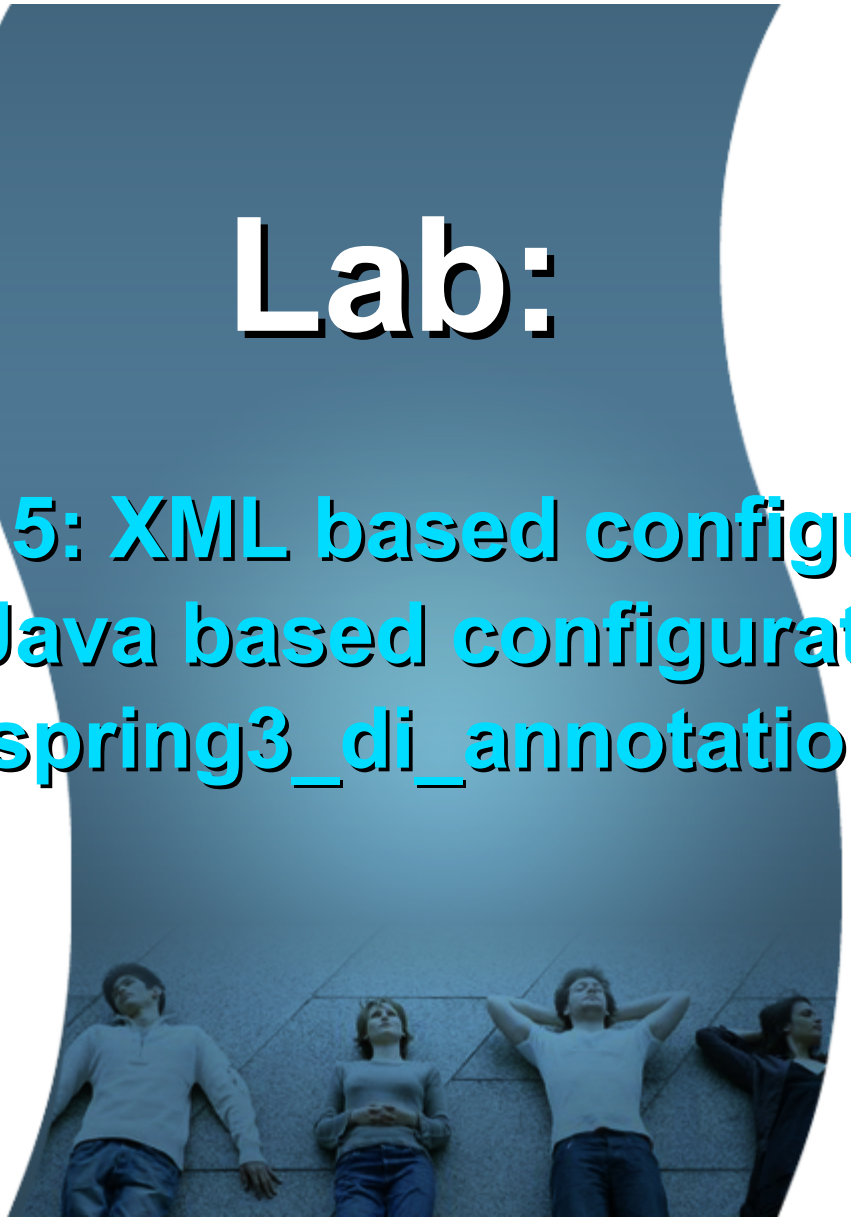
- A case where a bean has a dependency bean

```
@Configuration
public class AppConfig {
   @Bean
   public TransferService transferService() {
      return new TransferServiceImpl(accountRepository());
   }
   @Bean
   public AccountRepository accountRepository() {
      return new InMemoryAccountRepository();
   }
}

The above is the same as
<bean id = "transferService"
          class = "com.javapassion.examples.account.service.TransferServiceImpl">
        <property name="accountRepository" ref="accountRepository"/>
</bean>
<bean id = "accountRepository"
          class = "com.javapassion.examples.account.repository.InMemoryAccountRepository">
</bean
```

Lab:

Exercise 5: XML based configuration and Java based configuration
4939_spring3_di_annotation.zip



@Component & Further Stereotype Annotations (@Repository, @Service, @Controller)

@Component, @Repository, @Service, @Controller

- @Component is a generic stereotype for any Spring-managed component
- @Repository, @Service, and @Controller are specializations of @Component for more specific use cases
  - @Repository – for persistence
  - @Service – for service
  - @Controller – for controller

Why use @Repository, @Service, @Controller over @Component?

- You can annotate your component classes with @Component, but by annotating them with @Repository, @Service, or @Controller instead, your classes are more properly suited for processing by tools or associating with aspects
- It is also possible that @Repository, @Service, and @Controller may carry additional semantics in future releases of the Spring Framework.

Why Use @Repository, @Service, @Controller

- @Repository
  - A class that is annotated with "@Repository" is eligible for Spring org.springframework.dao.DataAccessException translation.
- @Service
  - A class that is annotated with "@Service" plays a role of business service
- @Controller
  - A class that is annotated with "@Controller" plays a role of controller in the Spring MVC application

Lab:

Exercise 6: Service class configured with XML and annotation

4939_spring3_di_annotation.zip

Component Scanning

Component Scanning Declaration

- The specified package via base-package attribute – *com.acme.mypackage* package in the example below - will be scanned, looking for any *@Component*-annotated (and its stereo-typed annotations - *@Service, @Repository, @Controller*) classes, and those classes will be registered as Spring bean definitions within the container.

```
<beans>
    <context:component-scan base-package="com.acme.mypackage"/>
</beans>
```

Filters to Customize Scanning

- Example: Ignore all @Repository annotations and using "stub" repositories instead.

```
<beans>

  <context:component-scan base-package="org.example">
    <context:include-filter type="regex"
                            expression=".*Stub.*Repository"/>
    <context:exclude-filter type="annotation'
                    expression="org.springframework.stereotype.Repository"/>
  </context:component-scan>

</beans>
```

Learn with Passion!
JPassion.com