Annotation-based Configuration

"Code with Passion!"

Topics (page 1)

- Annotation-based Dependency Injection
  - *@Autowired*
- Qualifier
  - @Qualifier, Custom qualifier
- JSR 330 (Dependency Injection for Java)
  - @Inject
- JSR 250 (Common Annotations)
  - *@PostConstruct & @PreDestroy, @Resource*
- *@Component and further stereotyped annotations*
  - *@Service, @Repository, @Controller*
- Auto scanning
  - *@ComponentScan*

Topics (page 2)

- Java-based Spring configuration (instead of XML configuration file)
  - *@Configuration, @Bean*
- Profile
  - *@Profile*
- Spring Boot
  - *@SpringBootApplication*
  - *@EnableAutoConfiguration*

Annotation-based Dependency Injection (DI)

Annotation-based DI specification

- An alternative to XML based DI specification
  - Bean definitions and wiring are specified in the Java source code
- You can use both XML and annotation-based DI specifications
  - Annotation-based injection is performed before XML-based injection
  - XML-based injection will override Annotation-based injection
- Annotation-based DI specification is usually preferred over XML-based DI specification
  - Typing checking is possible at compile time
  - No need to have separate XML-file

DI related Annotations Introduced in Spring

- 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*
- Spring 3.1
  - *@ComponentScan, @Profile*
- Spring 4
  - *@SpringBootApplication*
  - *@EnableAutoConfiguration*
  - *@Conditional*

@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 (setter injection)
  - Constructor methods (constructor injection)
  - Arbitrary methods

@Autowired at Field

```
public class MovieRecommender {

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


  // ...
}
```

@Autowired at Setter method

```
public class SimpleMovieLister {

  private MovieFinder movieFinder;

  // MovieFinder object gets created and injected by Spring DI container
  @Autowired
  public void setMovieFinder(MovieFinder movieFinder) {
     this.movieFinder = movieFinder;
  }

  // ...
}
```

@Autowired at Constructor method

```
public class MovieRecommender {

  private CustomerPreferenceDao customerPreferenceDao;

  @Autowired // used at the constructor – optional from Spring 4.3
  public MovieRecommender(
      CustomerPreferenceDao customerPreferenceDao) {
      this.customerPreferenceDao = customerPreferenceDao;
  }

  // ...
}
```

@Autowired at arbitrary methods

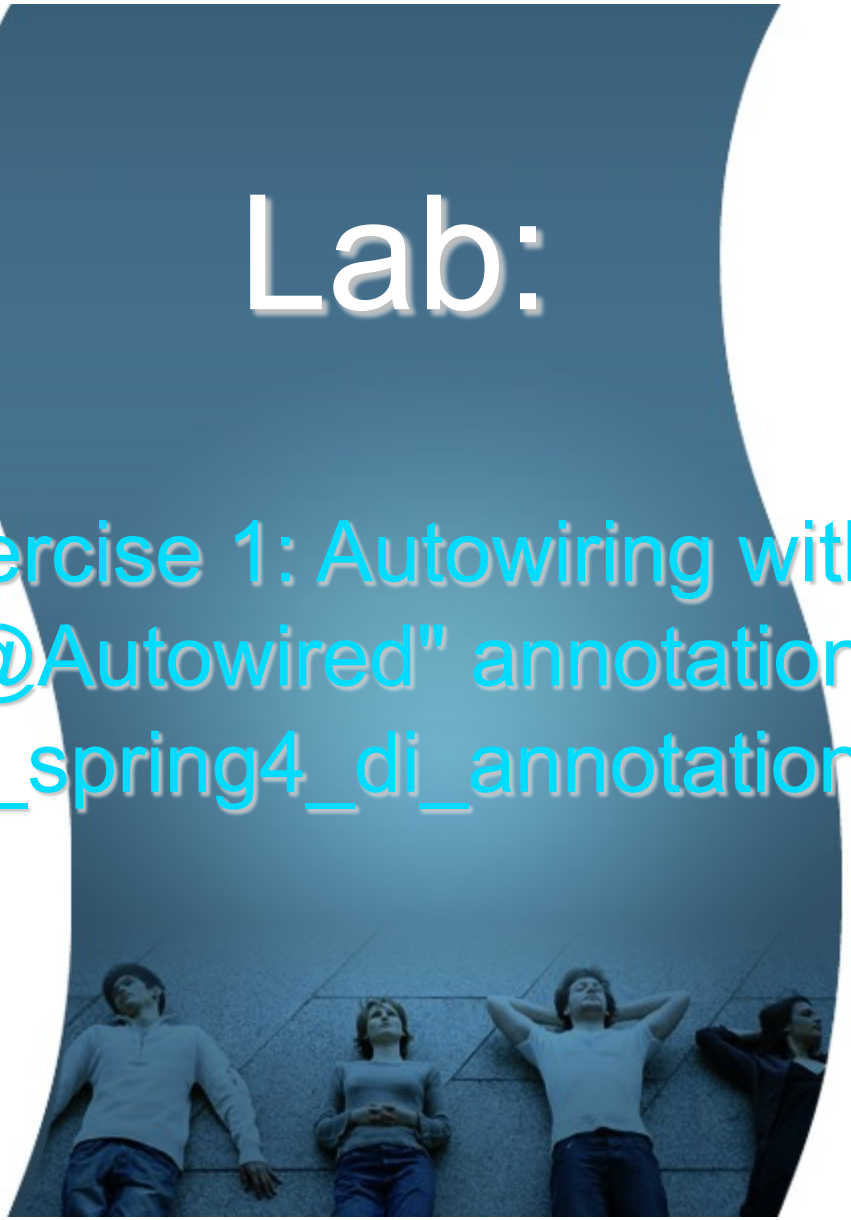
- You can also apply @Autowired annotation to methods with arbitrary names and/or multiple arguments:

```
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;
  }

  // ...
}
```

Lab:

Exercise 1: Autowiring with "@Autowired" annotation

4939_spring4_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 arguments

```
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

```
@Configuration
public class BeanConfiguration {

    @Bean(name = "myaddress")
    public AddressInterface getMyAddress() {
        AddressInterface address = new MyAddress();
        return address;
    }

    @Bean(name = "youraddress")
    public AddressInterface getYourAddress() {
        AddressInterface address = new YourAddress();
        return address;
    }

    @Bean
    public Person getPerson() {
        Person person = new Person();
        return person;
    }

}
```

There are two candidates to AddressInterface type: MyAddress and YourAddress.

Give name to these candidates so that they can be specified in the selection process

Custom Qualifier

Creating Custom Qualifier Annotation

- You can create your own custom qualifier annotations.

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

  String value();
}
```

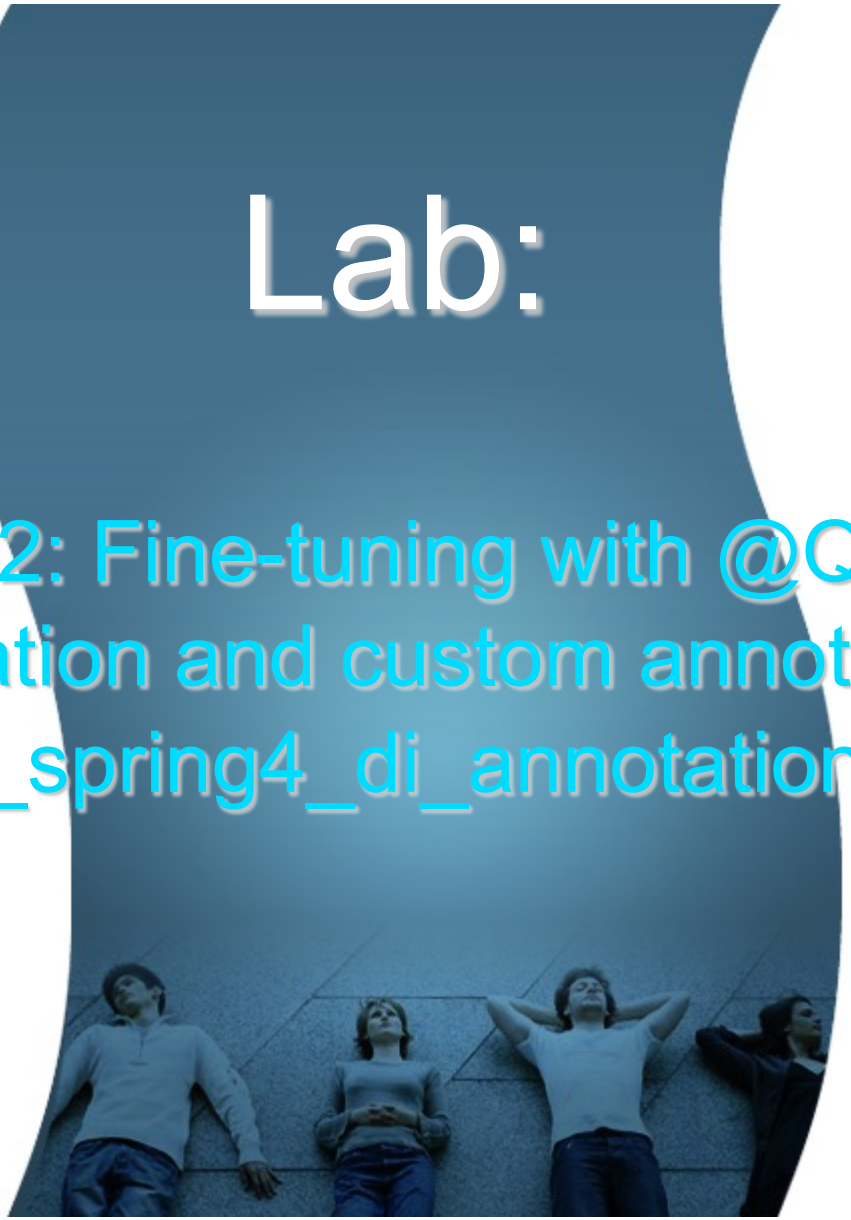
@Autowired with Custom Qualifier

- 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;
  }

  // ...
}
```

Lab:

Exercise 2: Fine-tuning with @Qualifier annotation and custom annotation
4939_spring4_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 annotation

@JSR 330 Maven Dependency

```
<!-- JSR 330 Dependency Injection for Java -->
<dependency>
        <groupId>javax.enterprise</groupId>
       <artifactId>cdi-api</artifactId>
       <version>1.2</version>
</dependency>
```

Lab:

Exercise 3: JSR 330 Annotations - @Inject

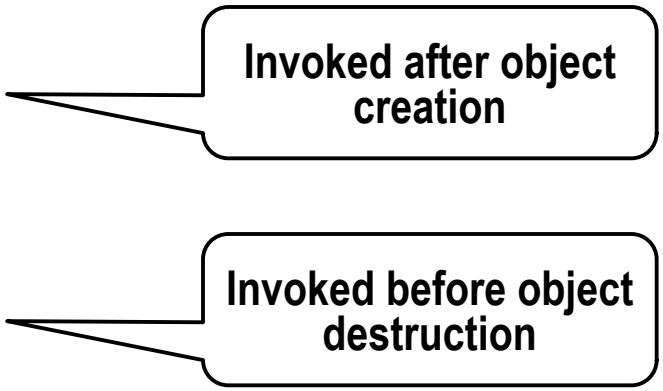
4939_spring4_di_annotation.zip

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

@PostConstruct and @PreDestroy

- Offers an post-initialization callback and an pre-destruction callback

```
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

```
public class SimpleMovieLister {

    private MovieFinder movieFinder;

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

Lab:

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

4939_spring4_di_annotation.zip

@Profile

@Profile

- Spring Profiles provide a way to segregate parts of your application configuration and make it only available in certain environments
- Any @Component or @Configuration can be marked with @Profile to limit when it is loaded

```
@Configuration
@Profile("production")
public class ProductionConfiguration {

    // ...

}
```

- You can then set a spring.profiles.active Environment property to specify which profiles are active
- You can also specify the property in *application.properties* file

```
spring.profiles.active=production,mysql
```

Lab:

Exercise 7: @Profile

4939_spring4_di_annotation.zip

@SpringBootApplication
@EnableAutoConfiguration

@SpringBootApplication

- Composite annotation (Stereo annotation)
- Introduced as part of Spring Boot

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@Configuration
@EnableAutoConfiguration
@ComponentScan
public @interface SpringBootApplication {

/**
 * Exclude specific auto-configuration classes such that they will never be applied.
 * @return the classes to exclude
 */
Class<?>[] exclude() default {};

}
```

@EnableAutoConfiguration

- Enable auto-configuration of the Spring Application Context, attempting to guess and configure beans that you are likely to need
- Introduced as part of Spring Boot
- Auto-configuration classes are usually applied based on your classpath and what beans you have defined
  - If you have tomcat-embedded.jar on your classpath, you are likely to want a TomcatEmbeddedServletContainerFactory (unless you have defined your own EmbeddedServletContainerFactory bean)
- Auto-configuration tries to be as intelligent as possible and will back-away as you define more of your own configuration
  - You can always manually exclude() any configuration that you never want to apply
  - Auto-configuration is always applied after user-defined beans have been registered.

Code with Passion!