Spring Annotations

Annotation Tutorial: Contents

JEE Annotations (Abstract Example Application References)

- Part 1: Spring Annotations
- Part 2: Hibernate - JPA Annotations
- Part 3: RESTful Web Service - JAX-RS Annotations
- Part 4: JAXB Annotations
- Part 5: Spring - jUnit Annotations

Spring Annotations: Contents:

Annotation	Package Detail/Import statement
@Service	import org.springframework.stereotype.Service;
@Repository	import org.springframework.stereotype.Repository;
@Component	import org.springframework.stereotype.Component;
@Autowired	Import org.springframework.beans.factory.annotation.Autowired;
@Transactional	import org.springframework.transaction.annotation.Transactional;
@Scope	import org.springframework.context.annotation.Scope;
Spring MVC Annotations	
@Controller	import org.springframework.stereotype.Controller;
@RequestMapping	import org.springframework.web.bind.annotation.RequestMapping;
@PathVariable	import org.springframework.web.bind.annotation.PathVariable;
@RequestParam	import org.springframework.web.bind.annotation.RequestParam;
@ModelAttribute	import org.springframework.web.bind.annotation.ModelAttribute;
@SessionAttributes	import org.springframework.web.bind.annotation.SessionAttributes;
Spring Security Annotations	
@PreAuthorize	import org.springframework.security.access.prepost.PreAuthorize;

For spring to process annotations, add the following lines in your application-context.xml file.

```
<context:annotation-config />
<context:component-scan base-package="...specify your package name..." />
```

Spring supports both Annotation based and XML based configurations. You can even mix them together. Annotation injection is performed before XML injection, thus the latter configuration will override the former for properties wired through both approaches.

@Service

Annotate all your service classes with @Service. All your business logic should be in Service classes.

```
1  @Service
2  publicclassCompanyServiceImpl implementsCompanyService {
3  ...
4  }
```

@Repository

Annotate all your DAO classes with @Repository. All your database access logic should be in DAO classes.

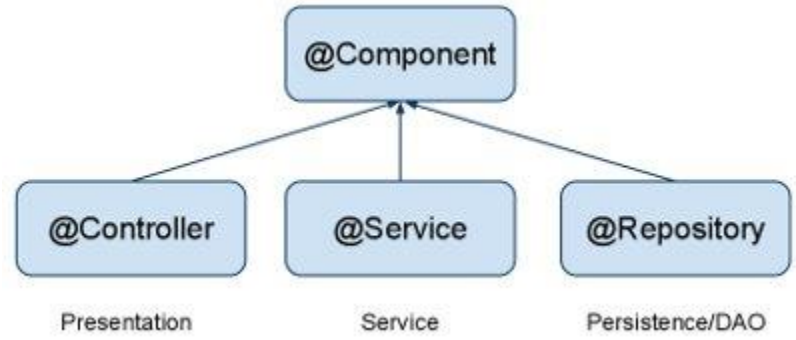
```
1  @Repository
2  publicclassCompanyDAOImpl implementsCompanyDAO {
3  ...
4  }
```

@Component

Annotate your other components (for example REST resource classes) with @Component.

```
1  @Component
2  publicclassContactResource {
3  ...
4  }
```

@Component is a generic stereotype for any Spring-managed component. @Repository, @Service, and @Controller are specializations of @Component for more specific use cases, for example, in the persistence, service, and presentation layers, respectively.



@Autowired

Let Spring auto-wire other beans into your classes using @Autowired annotation.

```
1        @Service
2        publicclassCompanyServiceImpl implementsCompanyService {
3
4         @Autowired
5         privateCompanyDAOcompanyDAO;
6
7         ...
8        }
```

Spring beans can be wired by name or by type.

- @Autowire by default is a type driven injection. @Qualifier spring annotation can be used to further fine-tune autowiring.
- @Resource (javax.annotation.Resource) annotation can be used for wiring by name.

Beans that are themselves defined as a collection or map type cannot be injected through @Autowired, because type matching is not properly applicable to them. Use @Resource for such beans, referring to the specific collection or map bean by unique name.

@Transactional

Configure your transactions with @Transactional spring annotation.

```
1        @Service
2        publicclassCompanyServiceImpl implementsCompanyService {
3
```

```
4           @Autowired
5           privateCompanyDAOcompanyDAO;
6
7           @Transactional
8           publicCompany findByName(String name) {
9
10            Company company = companyDAO.findByName(name);
11            returncompany;
12          }
13          ...
14        }
```

To activate processing of Spring's @Transactional annotation, use the <tx:annotation-driven/> element in your spring's configuration file.

The default @Transactional settings are as follows:

- Propagation setting is PROPAGATION_REQUIRED.
- Isolation level is ISOLATION_DEFAULT.
- Transaction is read/write.
- Transaction timeout defaults to the default timeout of the underlying transaction system, or to none if timeouts are not supported.
- Any RuntimeException triggers rollback, and any checked Exception does not.

These default settings can be changed using various properties of the @Transactional spring annotation.

Specifying the @Transactional annotation on the bean class means that it applies to all applicable business methods of the class. Specifying the annotation on a method applies it to that method only. If the annotation is applied at both the class and the method level, the method value overrides if the two disagree.

@Scope

As with Spring-managed components in general, the default and most common scope for autodetected components is singleton. To change this default behavior, use @Scope spring annotation.

```
1   @Component
2   @Scope("request")
```

```
3    publicclassContactResource {
4    ...
5    }
```

Similarly, you can annotate your component with @Scope("prototype") for beans with prototype scopes.

Please note that the dependencies are resolved at instantiation time. For prototype scope, it does NOT create a new instance at runtime more than once. It is only during instantiation that each bean is injected with a separate instance of prototype bean.

Spring MVC Annotations

@Controller

Annotate your controller classes with @Controller.

```
1    @Controller
2    publicclassCompanyController {
3    ...
4    }
```

@RequestMapping

You use the @RequestMapping spring annotation to map URLs onto an entire class or a particular handler method. Typically the class-level annotation maps a specific request path (or path pattern) onto a form controller, with additional method-level annotations narrowing the primary mapping.

```
1    @Controller
2    @RequestMapping("/company")
3    publicclassCompanyController {
4
5      @Autowired
6      privateCompanyServicecompanyService;
7    ...
8    }
```

@PathVariable

You can use the @PathVariable spring annotation on a method argument to bind it to the value of a URI template variable. In our example below, a request path of /company/techferry will bind companyName variable with 'techferry' value.

```
1      @Controller
```

```
2       @RequestMapping("/company")
3       publicclassCompanyController {
4
5        @Autowired
6        privateCompanyServicecompanyService;
7
8        @RequestMapping("{companyName}")
9        publicString getCompany(Map<String, Object> map,
10              @PathVariableString companyName) {
11        Company company = companyService.findByName(companyName);
12        map.put("company", company);
13        return"company";
14       }
15      ...
16      }
```

@RequestParam

You can bind request parameters to method variables using spring annotation @RequestParam.

```
1       @Controller
2       @RequestMapping("/company")
3       publicclassCompanyController {
4
5        @Autowired
6        privateCompanyServicecompanyService;
7
8        @RequestMapping("/companyList")
9        publicString listCompanies(Map<String, Object> map,
10              @RequestParamintpageNum) {
11        map.put("pageNum", pageNum);
12        map.put("companyList", companyService.listCompanies(pageNum));
13        return"companyList";
14       }
15      ...
16      }
```

Similarly, you can use spring annotation @RequestHeader to bind request headers.

@ModelAttribute

An @ModelAttribute on a method argument indicates the argument should be retrieved from the model. If not present in the model, the argument should be instantiated first and then added to the model. Once present in the model, the

argument's fields should be populated from all request parameters that have matching names. This is known as data binding in Spring MVC, a very useful mechanism that saves you from having to parse each form field individually.

```
1   @Controller
2   @RequestMapping("/company")
3   publicclassCompanyController {
4
5    @Autowired
6    privateCompanyServicecompanyService;
7
8    @RequestMapping("/add")
9    publicString saveNewCompany(@ModelAttributeCompany company) {
10    companyService.add(company);
11    return"redirect:"+ company.getName();
12   }
13  ...
14  }
```

@SessionAttributes

@SessionAttributes spring annotation declares session attributes. This will typically list the names of model attributes which should be transparently stored in the session, serving as form-backing beans between subsequent requests.

```
1   @Controller
2   @RequestMapping("/company")
3   @SessionAttributes("company")
4   publicclassCompanyController {
5
6    @Autowired
7    privateCompanyServicecompanyService;
8   ...
9   }
```

@SessionAttribute works as follows:

- @SessionAttribute is initialized when you put the corresponding attribute into model (either explicitly or using @ModelAttribute-annotated method).
- @SessionAttribute is updated by the data from HTTP parameters when controller method with the corresponding model attribute in its signature is invoked.
- @SessionAttributes are cleared when you call setComplete() on SessionStatus object passed into controller method as an argument.

The following listing illustrate these concepts. It is also an example for pre-populating Model objects.

```
@Controller
@RequestMapping("/owners/{ownerId}/pets/{petId}/edit")
@SessionAttributes("pet")
publicclassEditPetForm {

   @ModelAttribute("types")

   publicCollection<PetType>populatePetTypes() {
      returnthis.clinic.getPetTypes();
   }

   @RequestMapping(method = RequestMethod.POST)
   publicString processSubmit(@ModelAttribute("pet") Pet pet,
         BindingResult result, SessionStatus status) {
      newPetValidator().validate(pet, result);
      if(result.hasErrors()) {
         return"petForm";
      }else{
         this.clinic.storePet(pet);
         status.setComplete();
         return"redirect:owner.do?ownerId="
            + pet.getOwner().getId();
      }
   }
}
```

Spring Security Annotations

@PreAuthorize

Using Spring Security @PreAuthorize annotation, you can authorize or deny a functionality. In our example below, only a user with Admin role has the access to delete a contact.

```
@Transactional
@PreAuthorize("hasRole('ROLE_ADMIN')")
publicvoidremoveContact(Integer id) {
 contactDAO.removeContact(id);
}
```