

Spring Tutorial Notes

xml should be placed outside the com packages

notes:

- **3 ways of config:**
 - **full xml**
 - **xml component**
 - **<context: component - scan base-package="name of the package" />**
 - **java config**

Annotation: (no need to use xml to define an object)

1. @Component — on top of object class
2. In client code, use lowercased version of the object class name

or

1. @Component("name")
2. In client code, specify the object using "name"

Autowiring: (use method without specifying, but look for it in components)

(constructor injection, setter injection, field injection)

- **Constructor injection:**

1. @Component — on top of the object class where method desired is defined
2. @Autowired — on top of the constructor where we use unspecified method

- **Setter injection: (inject dependencies by calling setter methods on class)**

1. same but in setter

as a matter of fact, any method can use tech

- **Field injection: (inject dependencies by setting field values on your class directly)**

1. same but on top of field

- **autowiring and qualifiers**

1. @Qualifier(class name with lowercase) under @Autowired

Bean scope with annotations

- @Scope("desired_scope") — on top of the object class. (singleton,

prototype, etc.)

Bean life cycle

- `@PostConstruct` (will execute after constructor is called)
- and
- `@PreDestroy` (will execute before bean is destroy)
- on methods

Spring config using Java code (no xml)

- **General define process**
 1. Create a java class and annotated as `@Configuration`
 2. Add component scanning support: `@ComponentScan("package_name")`
 3. Read spring java configuration class
`(AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(SportConfig.class);`
 4. Retrieve bean from spring container (same as xml version)
- **Define beans in spring**
 1. Define method to expose bean (`@Bean` — on top of method)
 2. Inject bean dependencies
 3. Read spring java config class
 4. Retrieve bean from spring container
- **Java config props**
 1. Create properties file
 2. Load properties file in spring config [e.g. `@PropertySource("classpath:sport.properties")`]
 3. Reference values from properties file [e.g. `@Value("${foo.email}")`]

Spring MVC

framework for developing web application

- **configuration:**
 1. Add config to file: WEB-INF/web.xml
 1. Configure spring MVC dispatcher servlet
 2. Set up URL mappings to spring mvc dispatcher servlet
 2. Add config to file: WEB-INF/spring-mvc-demo-servlet.xml
 1. Add support for spring component scanning
 2. Add support for conversion, formatting and validation
 3. Configure spring mvc view resolver
- **Controller**
 1. `@Controller` — on controller class
 2. `@RequestMapping("/url_section")` — on controller method
- **HTML hyperlink**

e.g. `Hello World form`

- **MVC Model (container for application data)**

In controller, can put anything in the model (string, database, object, etc.)

JSP can access data from the model

1. Two params for controller method: HttpServletRequest and Model
2. HttpServletRequest.getParameter("param_name")
3. Model.addAttribute("param_name", param_value)

- **Binding request param (bind value of "studentName" to theName)**

e.g. public String processFormVersionThree(
 @RequestParam("studentName") String theName,
 Model model) {...}

- **Request Mapping for Controller**

Serves as parent mapping for controller

All request mapping on methods in the controller are relative

Similar to folder directory structures

@RequestMapping("/root") — on object

@RequestMapping("/branch") — on methods

Form tags (generate HTML)

make use of data binding

automatically setting / retrieving data from a java object / bean

- **Test-Field:**

<form:input.../>

add model attribute

in controller:

```
@RequestMapping("/showForm")  
public String showForm(Model theModel) {
```

```
    // create a student object  
    Student theStudent = new Student();
```

```
    //add student object to the model  
    theModel.addAttribute("student", theStudent);
```

```
    return "student-form";  
}
```

```
@RequestMapping("/processForm")  
public String processForm(@ModelAttribute("student") Student theStudent) {
```

```
    // log the input data  
    System.out.println("theStudent: " + theStudent.getFirstName() + " " +
```

```
theStudent.getLastName());

    return "student-confirmation";
}
```

in jsp:

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<form:form action="processForm" modelAttribute="student">
```

```
    First name: <form:input path="firstName" />
```

```
    <br><br>
```

```
    Last name: <form:input path="lastName" />
```

```
    <br><br>
```

```
    <input type="submit" value="Submit" />
```

```
</form:form>
```

```
The student is confirmed: ${student.firstName} ${student.lastName}
```

- **Drop down list:**
 <**form:select** .../>

in jsp

```
<form:select path="country">
    <form:option value="China" label="China" />
    <form:option value="USA" label="USA" />
    <form:option value="Russia" label="Russia" />
    <form:option value="UK" label="UK" />
    <form:option value="France" label="France" />
</form:select>
```

OR

in object

```
    LinkedHashMap
```

in jsp

```
<form:options items="${student.countryOptions}" />
```

- **Radio Buttons (multiple choice)**

`<form:radiobuttons.../>`

Java `<form:radiobutton path="favoriteLanguage" value="Java" />`

C# `<form:radiobutton path="favoriteLanguage" value="C" />`

PHP `<form:radiobutton path="favoriteLanguage" value="PHP" />`

Ruby `<form:radiobutton path="favoriteLanguage" value="Ruby" />`

- **Check boxes (choose multiple items)**

`<form:checkbox.../>`

in jsp

Linux `<form:checkbox path="operatingSystems" value="Linux" />`

Mac OS `<form:checkbox path="operatingSystems" value="Mac OS" />`

Spring and Validation

@NotNull — checks that the annotation value is not null

@Min — must be a number \geq value

@Max — must be a number \leq value

@Size — size must match the given size

@Pattern — must match the given size

@Future/@Past — data must be in future or past of given data

...

- **Validation required**

1. **Add validation rule to Customer class**
 2. **Display error messages on HTML form**
 3. **Perform validation in the Controller class**
-

@RequestMapping("/processForm")

public String processForm(

 @Valid @ModelAttribute("customer") Customer theCustomer,
 BindingResult theBindingResult) {

 System.out.println("last name: |" + theCustomer.getLastName() + "|");

 if (theBindingResult.hasErrors()) {

 return "customer-form";

 } else {

 return "customer-confirmation";

 }

}

4. Update confirmation page

- **Validation initBinder (deal with white space i.e. make it invalid)**

1. register custom editor in controller

@InitBinder

@InitBinder

```
public void initBinder(WebDataBinder dataBinder) {  
    // true: if all whitespaces trim to null  
    StringTrimmerEditor stringTrimmerEditor = new StringTrimmerEditor(true);  
    dataBinder.registerCustomEditor(String.class, stringTrimmerEditor);  
}
```
