**Spring MVC in 5 minutes:**

Process flow:

* DispatcherServlet receives a request whose URL pattern is “/home.htm”.
* DispatcherServlet consults **HandlerMapping** to find a controller whose bean name is “/home.htm”, finding the HomeController bean.
* DispatcherServlet dispatches the request to HomeController for processing.
* HomeController returns a **ModelAndView** object with a logical view name of home.
* DispatcherServlet consults its **view resolver** (configured as InternalResourceViewResolver) to find a view whose logical name is home.
  + InternalResourceViewResolver returns the path to /WEB-INF/jsp/home.jsp.
* DispatcherServlet forwards the request to the JSP at /WEB-INF/jsp/home.jsp to render the home page to the user.

DispatcherServlet is configured into web.xml.

When **DispatcherServlet is loaded**, By default, it loads the **Spring application context** from an XML file. If application needs to load more than one spring application context file, we need to configure a **context loader** in **web.xml** file

Following three important info is configured in the context file:

* Handler-mapping used - BeanNameUrlHandlerMapping
* View resolver used - InternalResourceViewResolver
* Controller class of the application - HelloController

Controller class:

* Controller class has to extend some base controller class.
* All of Spring’s controllers return a **ModelAndView** object from their execution methods
* Controller class cannot have same name even if different package. otherwise
  + Unexpected exception parsing XML document from ConflictingBeanDefinitionException. with caused controller class name.

A ModelAndView object holds both

* view information ("HelloPage") and
  + ModelAndView modelandview = **new** ModelAndView("HelloPage");
* model data (Message) that will be used when rendering the output.
  + modelandview.**addObject**("Message", "welcome to Spring- MVC");

Spring MVC Annotations:

@Controller annotation

* Must put base-package entry into Spring application context file to use this annotation.
  + <context:component-scan base-package="app.controller" />
* You needn’t to provide handler-mapping class entry along with controller class entry in xml file.

@RequestMapping annotation:

* RequestMapping annotation value for each method present in each controller class are mapped with request url pattern.
* Two controller methods(even in different packake) can't have same value of @RequestMapping annotations, otherwise Exception while deploying into container
  + org.springframework.beans.factory.BeanCreationException:
* If present at class level then @RequestMapping annotations present at method level is now relative to class-level.

@PathVariavle :

* Used in method argument, to bind place-holder provided in url-pattern/Request-mapping with variable.
* To use @pathVariable annotation provide below entry in spring-dispatcher-servlet.xml file.
  + <mvc:annotation-driven/>

@RequestParam annotation is used to bind the request parameters with given variables.

Againg used in method argument.

@ModelAttribute

As method argument:

Spring provides ModelAndView annotation to achieve two task automatically.

If you use @ModelAttribute annotation in method argument on student object then

* Binding of all request parameter( i.e. form element ) with the corresponding properties of the model object(Student) happens automatically.
* Corresponding JSP file for this request handler method can simply use the model reference passed as argument of @ModelAttribute.

on method of controller class

then whatever object you add to the model object of this method spring mvc automatically adds that object to the modelAndView object for each request handler method which is present in this controller class.

**Data Binding feature of spring:**

**AutoDatabinding** is feature of ModelAttribute where JSP data is bind to model(student) data automatically.

For catching all data binding related error, **bindingresult** is used at method argument which mostly redirect to the same page where error occurred.

Spring MVC internally uses **propertyeditor** classes to perform type conversion while performing data binding. Spring MVC has many built-in propertyeditor classes.

Initbinder And WebDataBinder are provided for Customizing data-binding feature

Like ignoring(not bind) one of the field (Mobile), even if provided while data binding.

When Spring provided property editor classes are not enough to meet the requirement, you can write your own property editor class.

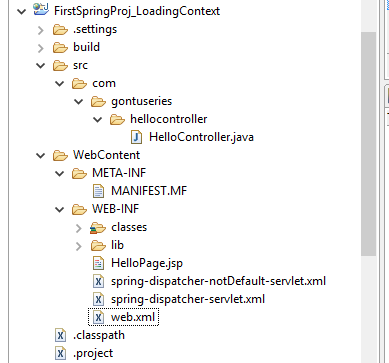
When one spring-class is deployed into tomcat not only that class but entire project is deployed.

* Url formation:
  + <http://localhost:8070/SpringAnnotProj/submitAdmissionForm3.html>
  + http://{**Machine**}:{**port**}/{**project-name**}/{valueOfRequestMapping}
* Two controller methods(even in different package) can't have same value of @RequestMapping annotations in same project. This leads to following exception
  + java.lang.IllegalStateException: Ambiguous mapping found.
  + Idea is to uniquely form URL.
* Controller class name cannot be same even in different package. Very unusual!! If it founds, then throws exception
  + May 15, 2016 3:47:13 PM org.springframework.web.servlet.FrameworkServlet initServletBean

SEVERE: Context initialization failed

In Details:

Spring-MVC directory structure:



**Startup-configuration:**

In Spring MVC, **DispatcherServlet** is the **front controller**. Before developing any app, we need to configure DispatcherServlet into web.xml.

A **front controller** is a common web-application pattern where a **single servlet** delegates responsibility for a request to other components of an application to perform the actual processing.

<servlet>

<servlet-name>spring-dispatcher</servlet-name>

<servlet-class>

org.springframework.web.servlet.DispatcherServlet

</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>spring-dispatcher</servlet-name>

<url-pattern>/</url-pattern>

</servlet-mapping>

When **DispatcherServlet is loaded**, By default, it will load the **Spring application context** from an XML file Whose **name** is based on the **name of the servlet**(spring-dispatcher-servlet.xml).

If your application has more than one **spring application context** file, then to ensure all of these configuration files are loaded, we need to configure a **context loader** in **web.xml** file as below.

<listener>

<listener-class>org.springframework.web.context.

ContextLoaderListener</listener-class>

</listener>

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/Fst-servlet.xml, Sec-servlet.xml</param-value>

</context-param>

A context loader loads context configuration files **in addition** to the one that DispatcherServlet loads. That means **spring-dispatcher-servlet.xml** must be present otherwise application will throw error (see below) while deploying into container(Tomcat).

SEVERE: Context initialization failed

org.springframework.beans.factory.BeanDefinitionStoreException: IOException parsing XML document from ServletContext resource [/WEB-INF/spring-dispatcher-servlet.xml]; nested exception is **java.io.FileNotFoundException**: Could not open ServletContext resource [/WEB-INF/spring-dispatcher-servlet.xml]

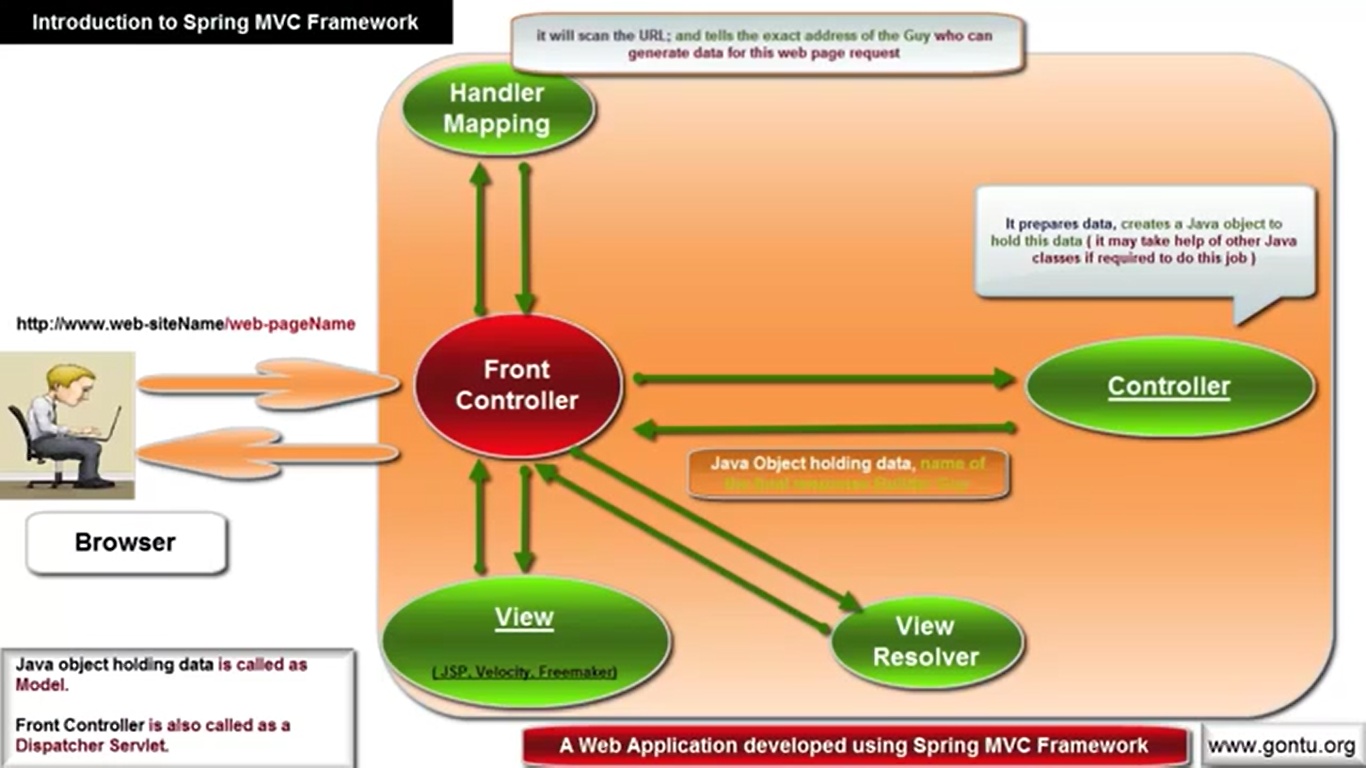
Caused by: java.io.FileNotFoundException: Could not open ServletContext resource **[/WEB-INF/spring-dispatcher-servlet.xml]**

at org.springframework.web.context.support.ServletContextResource.getInputStream(ServletContextResource.java:141)

**Basic-Flow**:

Process flow:

* DispatcherServlet receives a request whose URL pattern is “/home.htm”.
* DispatcherServlet consults **HandlerMapping** to find a controller whose bean name is “/home.htm”, finding the HomeController bean.
* DispatcherServlet dispatches the request to HomeController for processing.
* HomeController returns a **ModelAndView** object with a logical view name of home.
* DispatcherServlet consults its **view resolver** (configured as InternalResourceViewResolver) to find a view whose logical name is home.
  + InternalResourceViewResolver returns the path to /WEB-INF/jsp/home.jsp.
* DispatcherServlet forwards the request to the JSP at /WEB-INF/jsp/home.jsp to render the home page to the user.



**Spring MVC-Architecture**

**These all components of Spring-MVC are configured in spring application context** file (spring-dispatcher-servlet.xml).

<beans xmlns=*"http://www.springframework.org/schema/beans"*>

*<bean id="HandlerMapping" class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"/>*

*<bean name="/welcomeMost.html"*

*class="com.gontuseries.hellocontroller.HelloController" />*

*<bean id="viewResolver"*

*class="org.springframework.web.servlet.view.InternalResourceViewResolver" >*

*<property name="prefix">*

*<value>/WEB-INF/</value>*

*</property>*

*<property name="suffix">*

*<value>.jsp</value>*

*</property>*

*</bean>*

</beans>

**Following three important info is configured in the file:**

* **Handler-mapping used -** *BeanNameUrlHandlerMapping*
* **View resolver used -** *InternalResourceViewResolver*
* **Controller class of the application -** *HelloController*

When a request comes to DispatcherServlet with a URL that ends with “/*welcome*.htm”, DispatcherServlet will dispatch the request to *welcome* Controller for handling.

The default handler mapping used by DispatcherServlet is BeanNameUrlHandlerMapping, which uses the base name by default as the URL pattern.

HandlerMapping class match the incoming request url pattern with the beans name of the controller classes provided in this file.

Once found it suggest front controller to make a call to that controller class.

Spring provides other handler mappings that let you decouple a controller’s bean name from its URL pattern.

**Controller class:**

* Controller class has to extend some base controller class.
  + There are many base classes provided by spring.
* All of Spring’s controllers return a ModelAndView object from their execution methods.

**protected** ModelAndView handleRequestInternal(HttpServletRequest request,

HttpServletResponse response) **throws** Exception {

ModelAndView modelandview = **new** ModelAndView("HelloPage");

modelandview.addObject("Message", "welcome to Spring- MVC");

**return** modelandview;

}

A ModelAndView object holds both

* view information ("HelloPage") and
* model data (Message) that will be used when rendering the output.

DispatcherServlet uses view-resolver to find a view whose logical name (HelloPage) is returned by ModelAndView object.

Spring MVC comes with several view resolvers and simplest is **InternalResourceViewResolver**.

InternalResourceViewResolver prefixes the view name returned in the ModelAndView with the value of its prefix property and suffixed it with the value from its suffix property.

For *HelloController*, InternalResourceViewResolver will find the view at /WEB-INF/HelloPage.jsp.

Finally, DispatcherServlet forwards the request to the JSP at /WEB-INF/HelloPage.jsp to render the page to the user.

<html>

<body>

<h1>First Spring MVC Application Demo</h1>

<h2>${Message}</h2>

</body>

</html>

**Spring-MVC Annotation:**

**@Controller annotation:** use for the java class which is controller-class.

* Controller class need not required to extend base spring-MVC controller class.
* No need of putting entry of controller class in Spring application context file.
* You must put base-package entry into Spring application context file.

<context:component-scan base-package=*"app.controller"* />

In the absence of this entry you need to put entry of every single controller class in this file even if they are annotated with @controller.

* You needn’t to provide handler-mapping class entry as well.

**@RequestMapping annotation:**

* When request reaches to the web.xml file from browser the request is mapped to the dispatcherServlet (Front-contller) in web.xml.
* front controller reads spring-dispatcher-servlet.xml file for further processing.
* After seeing context:component-scan front-controller understand that annotation based controllers has been used and loads all the controller classes under the package app.controller in memory.
* Then it looks at RequestMapping annotation value of each method present in each controller class and compare their values by incoming request url pattern and whichever matches is then called.

@Controller

@RequestMapping("/greet")

**public** **class** MultiActionController {

@RequestMapping("/welcome")

**protected** ModelAndView helloWorld() {

ModelAndView modelandview = **new** ModelAndView("HelloPage");

modelandview.addObject("msg", "Hello world!!");

**return** modelandview;

}

**}**

**When** @RequestMapping annotations is present at class level then @RequestMapping annotations present at method level is now relative to class-level.

**url will be http://localhost:8070/SpringAnnotProj/greet/welcome**

**What happens if two controller methods are having same value of** @RequestMapping annotations:

Exception while deploying into container

org.springframework.beans.factory.BeanCreationException: Error creating bean with name 'org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping#0': Invocation of init method failed; nested exception is java.lang.IllegalStateException: **Ambiguous mapping found**. Cannot map 'testController' bean method

public org.springframework.web.servlet.ModelAndView app.controller.Z\_Doubts.TestController.helloWorld()

to {[/welcome],methods=[],params=[],headers=[],consumes=[],produces=[],custom=[]}: There is already 'helloController' bean method

public org.springframework.web.servlet.ModelAndView app.controller.A\_hellocontroller.HelloController.helloWorld() mapped.

at org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.initializeBean(AbstractAutowireCapableBeanFactory.java:1568)

**Controller class name cannot be same even if different package. Very unusual!!**

**If it founds, then throws exception**

May 15, 2016 3:47:13 PM org.springframework.web.servlet.FrameworkServlet initServletBean

SEVERE: Context initialization failed

org.springframework.beans.factory.BeanDefinitionStoreException: Unexpected exception parsing XML document from ServletContext resource [/WEB-INF/spring-dispatcher-servlet.xml]; nested exception is org.springframework.context.annotation.ConflictingBeanDefinitionException: Annotation-specified bean name **'helloController'** for bean class [app.controller.C\_PathVariable.HelloController] conflicts with existing, non-compatible bean definition of same name and class [app.controller.A\_hellocontroller.HelloController]

at org.springframework.beans.factory.xml.XmlBeanDefinitionReader.doLoadBeanDefinitions(XmlBeanDefinitionReader.java:413)

**looks like, this is because of same bean name generated through annotation.**

**Generated bean name is having the class name - helloController. For both class.**

**So confliction.**

RequestMapping has two parameters

* URL Pattern
* HTTP request or method (GET, POST, DELETE, PUT)

NOTE: - by default GET request is made by browser

@RequestMapping(value="/admissionForm.html", method = RequestMethod.***GET***)

**public** ModelAndView getAdmissionForm() { }

**@PathVariable**

Part of value of RequestMapping can be used as a placeholder.

For this include placeholder name inside curly braces {}.

@RequestMapping("/welcome/countryname/{user1}")

**public** ModelAndView helloWorld() {

}

To display/use placeholder in method, it needs to be bind with some method argument using **@PathVariavle** annotation.

@RequestMapping("/welcome1/{country}/{userName}")

**public** ModelAndView helloWorld(@PathVariable("country") String country,

@PathVariable("userName") String name ) {

ModelAndView model = **new** ModelAndView("HelloPage");

model.addObject("msg","hello " + name + country - " +country);

**return** model;

}

provide the pathVariable annotation on a map to bind more than one request param placeholders.

@RequestMapping("/welcome2/{country}/{userName}")

**public** ModelAndView helloWorld(@PathVariable Map<String,String> pathVars) {

}

**To use @pathVariable annotation provide following entry in** spring-dispatcher-servlet.xml file.

<mvc:annotation-driven/>

**RequestParam**

**@RequestParam** annotation is used to bind the request parameters with given variables.

* Request parameters are the field name set at html/jsp page.
* They are provided as the value/argument of **@RequestParam.**

@RequestMapping("/submitAdmissionForm.html")

**public** ModelAndView submitAdmissionForm(

@RequestParam("studentName") String name, @RequestParam("studentHobby") String hobby) {

ModelAndView model = **new** ModelAndView("AdmissionSuccess");

model.addObject("msg","Details submitted by you::

Name: "+name+

", Hobby: " + hobby);

**return** model;

}

Just like **@PathVariable**, **@RequestParam** on map can be used to bind more than one request parameters.

@RequestMapping(value="/submitAdmissionForm2.html", method = RequestMethod.***POST***)

**public** ModelAndView submitAdmissionForm(@RequestParam

Map<String,String> reqPar) {

String name = reqPar.get("studentName");

String hobby = reqPar.get("studentHobby");

}

Providing default value for request parameters.

When form is submitted without value for any field then Default value serve as value.

@RequestMapping("/submitAdmissionForm3.html")

**public** ModelAndView submitAdmissionForm(

@RequestParam(value="studentName", defaultValue="Manish") String name,

@RequestParam("studentHobby") String hobby) { }

In above example default value for "**studentName**" is "**Manish**", Which is accessed by **name** in this method.

**ModelAndView**

**You can add any model object to the MaodelAndView object. That model object can be used in JSP file.**

ModelAndView model = **new** ModelAndView("jsp/AdmissionSuccess");

model.addObject("student", student);

in jsp you can access same object by simply referring that name (key).

student.studentName = ${student.studentName}

@ModelAttribute

As method argument:

Spring provides ModelAndView annotation to achieve two task automatically.

If you use **@ModelAttribute** annotation in method argument on student object then

\* - binding of all request parameter( i.e. form element ) with the corresponding properties of the model object(Student) happens automatically.

\* - Corresponding JSP file for this request handler method can simply use the model reference passed as argument of **@ModelAttribute.**

\*

**public** ModelAndView submitAdmissionForm(@ModelAttribute("JSP\_student") Student student1) {

ModelAndView model1 = **new** ModelAndView("jsp/AdmissionSuccess");

**return** model1;

}

@ModelAttribute on method :

When ModelAttribute annotation is applied on method of controller class then whatever object you add to the model object of this method spring mvc automatically adds that object to the modelAndView object for each request handler method which is present in this controller class.

@ModelAttribute

**public** **void** addingCommonObjects(Model model1) {

model1.addAttribute("headerMessage", "@ModelAttribute EX);

}

Use Case : Common code of view like header can be put here.

Concept: Spring MVC calls method having annotation modelAttribute first, before making call any of it’s request handler method.

Adv : Less code.

**AutoDatabinding**

* AutoDatabinding is feature of ModelAttribute
* It works with any data type
* primitive and corresponding wrapper class type,
* Collection object or
* user defined type.

AutoDatabindig happens from JSP to model(student) object.

* JSP has just two type **text**, **multipleselect/option**
* This text type can be mapped to string, long, int date etc type of model.
* multipleselect/option can be mapped to collection type of model.

It’s user responsibility to provide correct data-type in JSP page, otherwise exception is thrown.

DOB in JSP should be provided Date format (even though it’s type is text in JSP) otherwise while binding to model date data type it would throw exception.

**Data binding error handling:**

BindingResult: For catching all data binding related error, use bindingresult at method argument.

**Redirect to the same page where error occurred.**

**In jsp file provide errors tag provided by Spring-MVC and include corresponding tag library.**

<%@ taglib prefix=*"form"* uri=*"http://www.springframework.org/tags/form"*%>

<form:errors path=*"student1.\*"*/>

**public** ModelAndView submitAdmissionForm(@ModelAttribute("student1") Student student, **BindingResult** result) {

**if** (**result**.**hasErrors**()) {

ModelAndView model1 = **new** ModelAndView("JSP/AdmissionForm");

**return** model1;

}

ModelAndView model1 = **new** ModelAndView("JSP/AdmissionSuccess");

**return** model1;

}

Customizing data-binding feature: **@Initbinder And WebDataBinder**

Use Case :

We need to ignore(not bind) one of the field say student Mobile, even if provided.

i.e. if we do not provide any value for studentMobile then

SringMVC must bind all form elements except studentMobile form element value with corresponding studentMobile property.

You can customize any field with below steps:

* Write a method for customizing logic
* On top of this method place @initbinder
* In method argument provide the reference of **WebDatabinder**
* With the help of WebDatabinder reference customize the field.

@InitBinder

**public** **void** myInitBinder(WebDataBinder webBinder){

webBinder.setDisallowedFields(**new** String[]{"studentMobile", "studentSkills"});

}

PropertyEditor Class

Use Case :

We need to provide the date in exactly in following format. 2010/10/10

If we use some different format say 2010---10---10, we will get exception.

How can we provide in above format?

This can be done with the help of webDataBinder reference and PropertyEditor class.

Concept:

Spring MVC uses propertyeditor classes internally to perform type conversion while performing data binding.

Spring MVC has many built-in propertyeditor classes.

FileEditor

ClassEditor

CustomDateEditor

CustomNumberEditor

you can use particular propertyeditor class in order to customize data binding for a particular data-type.

In this example we use CustomDateEditor class in order to customize data binding for date type.

@InitBinder

**public** **void** myInitBinder(WebDataBinder webBinder){

webBinder.setDisallowedFields(**new** String[]{"studentMobile"});

SimpleDateFormat dateFormat = **new** SimpleDateFormat("yyyy---dd--MM");

webBinder.registerCustomEditor(Date.**class**, "studentDOB",

**new** CustomDateEditor(dateFormat, **false**));

}

**Custom propertyeditor class**

Sometimes Spring provided property editor classes are not enough to meet the requirement. In these cases you can write your own property editor class.

* Write a class which extends PropertyEditorSupport class.
* Override the method public void setAsText(String name){ }
* Register this class with webBinder.

**Form-Validation:**

Prerequisites for form-validation:

A third part library is required which implements JSR unit 3 or JSR unit 49.

JSR – Java specification request. Provided by community JCP.

JCP – Java community process. This community has given out two standards related to bean validation - unit 3 or JSR unit 49

Hibernet-validator library is most popular among such Library.

Download following four jars from below link –

* hibernate-validator-5.1.0.Final.jar
* validation-api-1.1.0.Final.jar
* classmate-1.0.0.jar
* jboss-logging-3.1.1.GA.jar

<http://stackoverflow.com/questions/22541598/what-are-the-jars-necessary-to-bean-validation>

and put them in project lib folder: /SpringAnnotProj/WebContent/WEB-INF/lib

Now you will able to use all the validator annotations.

**Some imp validator Annotations**

* **Size**
  + **min:** required minimum size
  + **max:** required maximum size
  + **message(optional)**: custom message to be displayed when size in not in range.

**Example** : @Size(min=2, max=20, message="field between {min} and {max} characters.")

* **@Past : Date must be in past compare to today.**
* **@Future : Date must be in future compare to today.**
* **@Pattern : takes the regular expression for validation.**
  + **@pattern(regexp=***"[^0-9]\*" – Field shouldn’t contain any digit.*
* **@Max(2222) : field value should not cross 2222.**
* **@NotNull**
* **@NotEmpty**

**Spring message resource:**

**Include below code snippet in spring-dispatcher-servlet.xml**

<bean id=*"messageSource"* class=*"org.springframework.context.support.*

*ReloadableResourceBundleMessageSource"* >

<property name=*"basename"*>

<value>/WEB-INF/resource/studentMessages</value>

</property>

</bean>

**Create a file :**

WEB-INF/resource/studentMessages.properties

Key-Pattern : [validationAnnotationName].[ObjectReferenceName].[FieldName]

Ex: Size.student.studentHobby="Enter a value for **{0}** field between **{2}** and **{1}** characters--> From rssource bundle : Size.student.studentHobby"

* **{0}** – Field Name i.e. studentHobby
* **{1} – max value**
* **{2}** - **min value**

**Place Holder Concepts:**

**{0} is for name of filed for which violation occurs**

**{1}, {2} etc are for argument values passed to the constraint annotation in alphabetical order. As max comes before (alphabetically) min so {1} is for max and {2} is for max.**

**You need to restart the server to reflect the changes made in** studentMessages.properties.

If you avoid restarting the server make below change into **spring-dispatcher-servlet.xml** under bean id=*"messageSource"*

<property name=*"cacheSeconds"* value=”1”/>

This feature may degrade the performance of application so not recommended into production env.

**Custom Annotation:**

**public** **class** HobbyValidatorParam **implements** ConstraintValidator{

@Override

**public** **void** initialize(IsValidHobby arg0) {

// **TODO** Auto-generated method stub

}

@Override

**public** **boolean** isValid(String studentHobby, ConstraintValidatorContext ctx) {

**return** **false**;

}

}

@Documented

@Constraint(validatedBy = HobbyValidatorParam.**class**)

@Target(ElementType.***FIELD***)

@Retention(RetentionPolicy.***RUNTIME***)

**public** **@interface** IsValidHobby {

String message() **default** "Provide a valid hobby.";

Class<?>[] groups() **default**{};

Class<? **extends** Payload>[] payload() **default**{};

}

**You can crate your own custom from validation annotation.**

**Steps:**

**- Create a annotation class having complete definition of the annotation.**

**Name of the cutom annotation**

**Default error message**

**- Create the java class having validation logic.**

**This class is used by annotation class.**

**Now this custom validation annotation can be used as any other built-in validation annotation.**

**Interceptor**

**Interceptor is used**

**Include a java class which extends** HandlerInterceptorAdapter and override its methods

* **preHandle**: this method is called before handling the web-url. If this returns true then only further processing(appropriate handler method is called) is done otherwise application will send back the response from this method itself.
* **postHandle**: This method is called after spring-mvc executes the request handler method for a request.
* **afterCompletion**: This method is called after view file produces a response object for a request.

Register this class by putting it’s entry into Spring’s configuration file.

For every web page request

<mvc:interceptors>

<bean class = *"fully.qualified.ClassName"* />

</mvc:interceptors>

For specific URL request

<mvc:interceptors>

<mvc:interceptor>

<mvc:mapping path=””

<bean class = *"fully.qualified.ClassName"* />

</mvc:interceptor>

</mvc:interceptors>

**I18n**

**Steps:**

* **Replace hard-wired labelling with corresponding placeholder in JSP file.**
* **Create property files for each supported language using placeholders as key and value is language supported values.**
* **Set the language preference setting for browser using windows/settings.**
* **Create the supported language links at web page.**
  + **Pass the parameter sitelanguage=”lang”**
* **Register the interceptor localeChangeInterceptor in the spring configuration file.**
  + **This interceptor extracts the parameter sitelanguage and accordingly loads appropriate property file.**
  + **These all happens before processing the request (property of interceptor).**
* **Include an entry of cookieLocaleResolver bean in the spring configuration file.**

**Exception-Handling:**

**Exception handler methods**

* **Create custom web-page for each errors.**
* **Create exception handler methods.**
  + **Annotated by @ExceptionHandler()**
  + **Returned by symbolic view name(i.e. custom web-page).**

**If controller class throws any exception then that exception is handled by corresponding exception handler methods.**

**If it doesn’t find any such exception handler then it returns default error web-page.**

**Exception handler methods are applicable only for those controller class where it is present.**

**This leads to code duplication (redundancy).**

**@controllerAdvice**

**This annotation is applied to at the class level.**

**Exception handler methods placed in this class is applicable for the entire application.**

**So all the exception handler are placed globally at one place.**

**Exception handling in bean configuration file:**

**Class – SimpleMappingExceptionResolver**

**ExceptionMappings - Then map property is used to do the mapping for known exception and symbolic view.**

**defaultErrorView - Then property is used to do the map other exceptions with symbolic view.**

**Then using map property do the mapping for exception and symbolic view.**

Quick Concepts in 5 minutes: