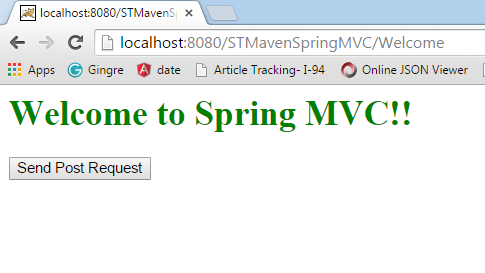
**Introduction**

|  |
| --- |
| The Spring web MVC framework provides model-view-controller architecture and ready components that can be used to develop flexible and loosely coupled web applications.  The MVC pattern results in separating the different aspects of the application (input logic, business logic, and UI logic), while providing a loose coupling between these elements.  [MVC](http://www.google.com/url?q=http%3A%2F%2Fwww.raystec.com%2Fcase-study%2Fmvc-architecture&sa=D&sntz=1&usg=AFQjCNF92DTla_Wtk4uO-jDNUez5BMwVCQ)is a framework methodology that separates an application's code implementation into three components model, view, and controller. Each component contains different application logics.  Spring MVC is built on the Spring Inversion of control(IoC) framework. Spring decouples the MVC components and simplifies the whole MVC configuration.  Figure: Spring MVC Component interaction diagram   * User requests are sent to Controller. Controller communicates with Service to perform business operations and will set return data  or other data into Model object and forward control to View. * View gets data from Model object and displays at Page. * Model is a carrier of data between Controller and View. * Service communicates to DAO to make database operations. Service performs business operations and handles transactions. * DAOs are implemented in ORM or JDBC and communicate to database.  DAO performs CRUD operations.   The DispatcherServlet  The Spring Web model-view-controller (MVC) framework is designed around a DispatcherServlet that handles all the HTTP requests and responses. The request processing workflow of the Spring Web MVC DispatcherServlet is illustrated in the following diagram:    Following is the sequence of events corresponding to an incoming HTTP request to DispatcherServlet:   * After receiving an HTTP request, DispatcherServlet consults the HandlerMapping to call the appropriate Controller. * Controller is called with the help of URL mapped to GET or POST request. Controller calls object of Service class to perform required business operation. After finishing services call, Controller stores data into model and returns View name to DispatcherServlet. * The DispatcherServlet will take help from ViewResolver to pickup the defined view for the request. * Once view is finalized, The DispatcherServlet passes the model data to the view which is finally rendered on the browser. View takes data from model to display at view page.   All the above mentioned components ie. HandlerMapping, Controller and ViewResolver are parts of WebApplicationContext which is an extension of the plain ApplicationContext with some extra features necessary for web applications.  Components  Defining Controller  Controllers are defined with help of @Controller annotation. The @Controller annotation indicates that a particular class serves the role of a controller. The @RequestMapping annotation is used to map a URL to either an entire class or a particular handler method.  **@Controller**  **@RequestMapping(value = "/Welcome")**  **public class [WelcomeCtl](https://sites.google.com/site/sunraysspringmvc/source-code/src/in-co-sunrays-ctl/welcomectl-java" \t "_blank) {**  **@RequestMapping(method = RequestMethod.GET)**  **public String display(Model model) {**  **System.out.println("I am in display");**  **model.addAttribute("message", "Welcome to Spring MVC!!");**  **return "**[**Welcome**](https://sites.google.com/site/sunraysspringmvc/source-code/webapp/pages/welcome-jsp)**";**  **}   ..**  **}**  The @Controller annotation defines the class as a Spring MVC controller. Here, annotation @RequestMapping before class maps \Welcome url to this controller.  Next annotation @RequestMapping(method = RequestMethod.GET) before login() method is used to declare the display() method as the controller's default service method that handles HTTP GET request.  Last @RequestMapping(method = RequestMethod.POST) before submit() method is used to declare the submit() method as the controller's  service method that handles HTTP POST request.  The @Controller mechanism also allows you to create RESTful Web sites and applications by using  @PathVariable annotation and other features.  Creating JSP Views  Spring MVC supports different types of views for different presentation technologies that includes JSPs, HTML, PDF, Excel worksheets, XML, Velocity templates, XSLT, JSON, Atom and RSS feeds, JasperReports etc.  But most commonly we use JSP templates written with JSTL.  Here is sample code that created JSP view for [Welcome](https://sites.google.com/site/sunraysspringmvc/source-code/webapp/pages/welcome-jsp) page. As per view resolver Welcome page will be located as  /WEB-INF/pages/Welcome.jsp:  <html>     <head>       <title>Hello Spring MVC</title>     </head>     <body>  **<h2>${message}</h2>**     </body>  </html>  Here ${message} is the model attribute which we have added from controller using model.addAttribute() method. You may add more attributes in model to display at View.  Model  Model is a map object that contains data in form of key and value pair. Model carries data from Controller to View. View gets data from Model using keys and displays them using JSTL and other Spring tags. Here Model carries "message"  from Controller to View. Model is passed as parameter to a Controller method.  Following are the two model maps provided by Spring:   1. org.springframework.ui.Model 2. org.springframework.web.servlet.ModelAndView   **Q: Can you override DispatcherServlet ?**  A: Yes we can override it to implement Front Controller.  **Q:  What is the function of Handler Mapping object?**  A: It routes mapping url to respective controller class and its method.  **Q:  What is the function of View Resolver ?**  A: Is resolves the view path and render view.  **Q:  Which View resolver you have used in your application?**  A:  UrlBasedViewResolver  **Q: What is the function of Model?**  A: It carries data from Controller to View and View to Controller. Model object is injected in a method of controller as a parameter.  **Q:  What is the naming convention of configuration file ?**  A:  servletname-dispatcher.xml  **Q:  How do you inject Model object into method of controller?**  A:  Using the method argument we will pass model object to method of controller.  **Q: If you create dispatcher-servlet.xml file in SpringMVC then do you create ApplicationContext.xml file ?**  A: No ApplicationContext.xml is created for non-web standalone application. |

### Getting Started

|  |
| --- |
| Defining Controller Define your controller class with help of @Controller annotation. The @Controller annotation indicates that a particular class serves the role of a controller. Use  @RequestMapping annotation to map controller class and its methods to a URL.  The @RequestMapping annotation is used to map a URL to either an entire class or a particular handler method.  This mapped urls will be used to send request from browser to the controller.  Here [Welcome](https://sites.google.com/site/sunraysspringmvc/source-code/src/in-co-sunrays-ctl/welcomectl-java) class is annotated by @Controller annotation and mapped with \Welcome url.   Method display() is mapped with GET method and receives Model object as the parameter. Method display() is called when GET request is sent to URL \Welcome.  Model is a map object that contains data in form of key and value pair. Model carries data from Controller to View. View gets data from Model using keys and displays them using JSTL and other Spring tags. Here Model carries "message"  from Controller to View.  Method  display() returns View name. View name is resolved by ViewResolver as per bean definition in [sunraysweb-servlet.xml](https://sites.google.com/site/sunraysspringmvc/source-code/webapp/web-inf/sunraysweb-servlet-xml).  View name "Welcome" will be resolved as [\WEB-INF\pages\Welcome.jsp](https://sites.google.com/site/sunraysspringmvc/source-code/webapp/pages/welcome-jsp).  Likewise submit() method is mapped with POST method.  When POST request is sent submit() method will be called.  **@Controller**  **@RequestMapping(value = "/Welcome")**  **public class WelcomeCtl {**  **@RequestMapping(method = RequestMethod.GET)**  **public String display(Model model) {**  **System.out.println("I am in display");**  **model.addAttribute("message", "Welcome to Spring MVC!!");**  **return "Welcome";**  **}**  **@RequestMapping(method = RequestMethod.POST)**  **public String submit(Model model) {**  **System.out.println("I am in submit");**  **model.addAttribute("message", "You sent post request!");**  **return "Welcome" }}** Creating JSP View Spring MVC supports different types of views for different presentation technologies that includes JSPs, HTML, PDF, Excel worksheets, XML, Velocity templates, XSLT, JSON, Atom and RSS feeds, JasperReports etc.  But most commonly we use JSP templates written with JSTL.  Here is Welcome.jsp.  This View is resolved by ViewResolver as [/WEB-INF/pages/Welcome.jsp](https://sites.google.com/site/sunraysspringmvc/source-code/webapp/pages/welcome-jsp). View gets  value stored against  "message" key in Model object and displays value using JSTL ( ${..} ) tag.  <html>     <head>       <title>Hello Spring MVC</title>     </head>     <body>  **<h2>${message}</h2>**     </body>  </html>  Here ${message} is the model attribute which we had added from controller using model.addAttribute() method. You may add more attributes in model to display at View. Send a Request to Controller Now your Controller is ready. Open a browser and send a request to controller using http://localhost:8080/STMavenSpringMVC/Welcome URL and you will see following screen. |



### Path Binding with Controller

|  |
| --- |
| The Controller is mapped with URL with the help of @RequestMapping annotation. Annotation @RequestMapping can map Controller class and its methods as well.   Mapping can be further bound with HTTP POST and GET methods.  Typically the class-level annotation maps a specific request path (or path pattern) to the class controller, with additional method-level annotations narrowing the primary mapping for a specific HTTP method request method ("GET", "POST", etc.) or an HTTP request parameter condition. Bind Request Path to the Controller and GET and POST requests to its methods Here is sample code that binds [UrlMappingCtl](https://sites.google.com/site/sunraysspringmvc/source-code/src/in-co-sunrays-ctl/urlmappingctl-java) class with path "/UrlMapping" .  Further it binds GET request to display() method and POST to submit() method. You can access this controller using GET request by making a call at url http://localhost:8080/STMavenSpringMVC/UrlMapping.  @Controller  @RequestMapping(value = "/UrlMapping")  public class UrlMappingCtl {  //Map GET request with display() method.  @RequestMapping(method = RequestMethod.GET)  public String display() {  System.out.println("I am in display method");  return "UrlMappingView";  }  //Map GET request with display() method.  @RequestMapping(method = RequestMethod.POST)  public String submit() {  System.out.println("I am in submit method");  return "UrlMappingView";  }  This View is resolved by ViewResolver as [/WEB-INF/pages/UrlMappingView.jsp](https://sites.google.com/site/sunraysspringmvc/source-code/webapp/pages/urlmappingview-jsp). Here is form that sends POST request.  <f:form action="UrlMapping" method="post">     <input type="submit" value="Send Post Request" name="operation"> </f:form> Bind sub-path to method of the Controller You can map sub-path to the method. In order to make a call to path bound method, sub-path will be prefixed by class level path.  Here is sample code that maps path \search  to method search().  Further method is bound to GET and POST requests both.  @RequestMapping(value = "/search", method = { RequestMethod.POST, RequestMethod.GET })  public String search() {  System.out.println("I am in search method, it handles GET and POST methods.");  return "TestSearchView";  }  Request can be made to this method by calling  http://localhost:8080/STMavenSpringMVC/UrlMapping/search using either GET and POST request. Bind request parameters to method parameters You can bind request parameters to method parameters with the help of @RequestParam annotation. In order to bind a request parameter, its name must be same as method parameter. You can bind one or multiple request parameters to method.  Here is the example code we are binding "message" request parameter to method parameter with help of @RequestParam annotation.  @RequestMapping(value = "/param", method = RequestMethod.GET)  public String requestParam(@RequestParam String message) {  System.out.println("Request Parameter : " + message);  return "UrlMappingView";  }  FAQ  Q: How do you bind request parameters with method parameters ?  Q: How do you bind request parameters with Form bean ?  Q: How do you bind URI  variables with method parameters ?  Q: How do you bind JSON request body object to Form bean ? |

### Data Biding

|  |
| --- |
| Controller.  Controller uses   1. annotation @RequestParam to bind request parameters with method arguments. 2. annotation @ModelAttribute to bind request parameters to a bean object and store bean into Model object.     Figure: request parameter binding Method Parameter Binding Annotation @RequestParam to bind request parameters with method arguments.  Here is example code that binds a "message" request parameters to method parameter "message".  This is equivalent to  String message = request.getParameter("message");  @RequestMapping(value = "/param", method = RequestMethod.GET)  public String requestParam(**@RequestParam** String message) {    System.out.println("Request Parameter : " + message);    return "UrlMappingView";  }  See   * [ReqBindingCtl](https://sites.google.com/site/sunraysspringmvc/source-code/src/in-co-sunrays-ctl/reqbindingctl-java) * [ReqBindingView.jsp](https://sites.google.com/site/sunraysspringmvc/source-code/webapp/pages/reqbindingview-jsp)   Above code  binds request parameter named "message" to the method parameter named "message".   Request parameter's name can be passed to annotation as parameter. Above method signature can be rewritten as  public String requestParam(**@RequestParam("message")** String message) {..} Different Names If request parameter name  and method parameter name is different then you can write annotation as  public String requestParam(**@RequestParam("msg")** String message) {..}  Here request parameter named "msg" will be bound with method parameter named "message". Type Conversion If type of method parameter is other than String then type conversion will be automatically done by Spring MVC.  public String requestParam(**@RequestParam("id")** **long**id) {..}  Here "id" request parameter will be converted into long and stored in "id" method parameter. Optional Parameter By default request parameter is required. If it is not sent in request then you will get an exception. To make it optional you can set "required" attribute of @RequestParam to false.  @RequestParam(value="id", required=false)). Form bean Binding Request parameters can be populated into a Model attribute form-bean using @ModelAttribute annotation. This concept is called Data Binding in Spring MVC.  Here is the example [LoginSimpleForm](https://sites.google.com/site/sunraysspringmvc/source-code/src/in-co-sunrays-form/loginsimpleform" \t "_blank) bean that is populated from request parameters in the controller [FormBindingCtl](https://sites.google.com/site/sunraysspringmvc/source-code/src/in-co-sunrays-ctl/formbindingctl-java" \t "_blank).   Form bean is further bind to HTML form  of [LoginBindingView.jsp](https://sites.google.com/site/sunraysspringmvc/source-code/webapp/pages/loginbindingview-jsp" \t "_blank).  **Form bean**  It is simple POJO that has attributes and its getter and setter methods.  public class [LoginSimpleForm](https://sites.google.com/site/sunraysspringmvc/source-code/src/in-co-sunrays-form/loginsimpleform" \t "_blank) {  private String login = null;  private String password = null;  private String message = null;  private String operation = null;  public String getLogin() {  return login;  }  public void setLogin(String login) {  this.login = login;  }  //Other accessor  }  **Controller**  Populates Request parameters into [LoginSimpleForm](https://sites.google.com/site/sunraysspringmvc/source-code/src/in-co-sunrays-form/loginsimpleform" \t "_blank) bean and make it Model attribute named "loginForm".  public class FormBindingCtl {  @RequestMapping(value = "/login", method = RequestMethod.GET)  public String loginDisplay(**@ModelAttribute("loginForm")**LoginSimpleForm form, Model model) {  return "LoginBindingView";  }  @RequestMapping(value = "/login", method = RequestMethod.POST)  public String loginSubmit(@ModelAttribute("loginForm") LoginSimpleForm form, Model model) {  String message = "You entered Login: " + form.getLogin()  + " and Password: " + form.getPassword();  form.setMessage(message);  return "LoginBindingView";  }  ...  **HTML form biding in View**  Spring tag f:form and commandName="loginForm" is used to bind model attribute "loginForm" to HTML form elements.  <BODY>  <H1>Login Form</H1>  <H3 style="color: green">${loginForm.message }</H3>  <f:form action="../FormBinding/login" method="post" **commandName="loginForm"**>  <table>  <tr>  <td>Login</td>  <td><f:input path="login" /></td>  </tr>  <tr>  <td>Password</td>  <td><f:password path="password" /></td>  </tr>  <tr>  <td colspan="2"><input type="submit" value="SignIn"  name="operation"> | <input type="submit" value="SignUp"  name="operation"></td>  </tr>  </table>  </f:form>  </BODY> |

**Input Validation**

|  |
| --- |
| Validations are applied using two libraries  :   1. **Business Validation**: Checks the business conditions that may need database communication. 2. **Input Validation:** Checks data entered by User. It does not need database communication. It can be further categorized as    1. Programmatic or Manual validation: Validation code are written manually by Programmer    2. Declarative validation :  Validation are declared using XML configuration or Annotations.   Spring supports both kind of Input validations Programmatic and Declarative. |

### Declarative Input Validation

|  |
| --- |
| Spring 3 supports JSR-303 Bean Validation API to perform declarative input validations.  You can apply declarative validation with help of   1. Bean Validation API 1.1 annotations 2. Hibernate Validator 5.0.1.Final annotations.   Here is example code of [UserValidationForm](https://sites.google.com/site/sunraysspringmvc/source-code/src/in-co-sunrays-form/uservalidationform-java" \t "_blank) bean that applies Basic and Hibernate validations. Form Bean public class [UserValidationForm](https://sites.google.com/site/sunraysspringmvc/source-code/src/in-co-sunrays-form/uservalidationform-java" \t "_blank) {      @Min(value = 1, message = "Id must be a positive number.")     private long id = 0L;      @NotNull(message = "First Name can not be Null.")     @Size(min = 5, max = 50)     private String firstName = null;      @NotNull(message = "Last Name can not be Null.")     @Size(min = 5, max = 50)     private String lastName = null;      @NotNull(message = "Gender can not be Null.")     @Size(min = 1, max = 6)     private String gender = null;      @Min(value = 13, message = "Age must be beatween 13-90.")     @Max(value = 90, message = "Age must be beatween 13-90.")     int age = 0;      ...  Annotation@NotNull, @Min, @Max and @Size are the basic validation annotations.   Attribute values in an annotation set the error message and other restrictions. For example min and max attributes of @Size annotation, validate the minimum and maximum size of a string.  Additional validations can be applied by Hibernate validation annotations.  Here is example of applying @Email and  @URL validation of hibernate annotation.  @Email(message = "Invalid Email ID.")  private String email = null;  @URL(message = "Invalid URL")  private String url = null;  Controller  Annotation @Valid is used to enable validation on a bean.  If validation is failed, error messages are  set into BindingResult object.  You can call hasErrors() method of BindingResult  to check if there is any validation error and take corrective path.  Here is sample code that will print all error messages at console.  @Controller @RequestMapping(value = "/FormBinding") public class [FormBindingCtl](https://sites.google.com/site/sunraysspringmvc/source-code/src/in-co-sunrays-ctl/formbindingctl-java" \t "_blank) {     @RequestMapping(value = "/user", method = RequestMethod.POST)     public String userSubmit( @ModelAttribute("userForm") @Valid UserValidationForm userFrom, BindingResult result,                              Model model){          if (result.hasErrors()) {             model.addAttribute("error", "Input validation error!");             List<ObjectError> l = result.getAllErrors();             for (ObjectError e : l) {                 System.out.println(e.getObjectName() + "-" + e.getCode() + "-" + e.getDefaultMessage());             }         } else {             String message = "Entered user name is " + userFrom.getFirstName()+ " " + userFrom.getLastName();             model.addAttribute("message", message);         }         System.out.println("I am in post" + userFrom.getDateOfBirth());         return "UserBindingView";     } }  Order of controller method parameters should be Validation bean, BindingResult and then Model object. View You can display error messages at View using  <f:errors  /> tag.  Set bean's field name to path attribute and CSS error class to  cssClass attribute.  <tr>  <td>ID</td>  <td><f:input path="id" /> <f:errors path="id" cssClass="error" /></td>  </tr>  <tr>  <td>First Name</td>  <td><f:input path="firstName" /> <f:errors path="firstName" cssClass="error" /></td>  </tr> Configuration In order enable MVC validation annotation your [configuration xml](https://sites.google.com/site/sunraysspringmvc/source-code/webapp/web-inf/sunraysweb-servlet-xml) must have following entry   <mvc:annotation-driven /> Maven Dependency You need to include following Maven dependencies for annotation in pom.xml:  <dependency>      <groupId>javax.validation</groupId>      <artifactId>validation-api</artifactId>      <version>1.1.0.Final</version>  </dependency>  <dependency>      <groupId>org.hibernate</groupId>      <artifactId>hibernate-validator</artifactId>      <version>5.1.0.Final</version>  </dependency> AnnotationsBasic Annotations  1. **@Size**(min=2, max=30) will only allow names between 2 and 30 characters long 2. **@NotNull** will not allow a null parameter. It checks the name of attribute in the request parameter list, if does not exist then fail validation. By default if parameter is received with empty value then Spring MVC set empty string instead of null to its attribute. For check non empty value then go with @NonEmpty annotation. 3. **@Min**(18) will not allow allow value less than 18 4. **@Max**(75) will not allow allow value greater than 75 5. **@Pattern**(regexp="\\d{6}") will apply a regular expression.  Here it will accept 6 digits. 6. **@Digits**(integer = 15, fraction = 2) will validate decimal values.  Hibernate Annotations  1. @NotEmpty : allow non empty string 2. @Email allow valid email formatted string 3. @Length (min=3,max=25): checks length of string 4. @Range: range of a value 5. @Range(min = 1, max = 150): checks length of a number.   Ex:@Range(min = 1, max = 150) //age need between 1 and 150  int age;   1. @URL: allow valid URL  Format Annotations @NumberFormat(style=Style.CURRENCY)  @DateTimeFormat(pattern="MM/dd/yyyy") Custom Messages (i18n) You can specify error messages in annotation itself.  @NotEmpty (message="Name is required")  private String name null;  To achieve i18n you can create message\_en.properties file can make following entries.  You can override default error messages here. Error messages are resolved using the following pattern:  {ValidationClass}.{modelObjectName}.{field}  For example is "password" field of our “loginForm” model object fails the “NotEmpty” validation, the “NotEmpty.loginForm.password" message will be searched.  If message is not found then it will search   “NotEmpty.loginForm".  If message is not found then finally it will search “NotEmpty".  If message is not found then default messages will be rendered.  NotEmpty=value can not be empty Size=Size=value must be between {2} and {1} characters long Email=Not a valid email.  NotEmpty.form.password=Password is required. Email.form.emailId=Login should be an email id. |

### Pre-load Data

|  |
| --- |
| Pre loaded data the data that is displayed at View HTML Forms  irrespective of any error.  ***All  drop-down lists on html forms are  pre-loaded data.***  Annotation @ModelAttribute can be used on a method of controller.  When @ModelAttribute is applied on a method of controller then this method will be invoked before any method mapped with @RequestMapping annotation.  Here is example of [UserCtl](https://sites.google.com/site/sunraysspringmvc/source-code/src/in-co-sunrays-ctl/userctl-java" \t "_blank)that pre-loads  role list, displayed at User Form while added a User.  @Controller @RequestMapping(value = "/User") @SessionAttributes("userContext") public class [UserCtl](https://sites.google.com/site/sunraysspringmvc/source-code/src/in-co-sunrays-ctl/userctl-java" \t "_blank) {  **@ModelAttribute     public void prepare(@RequestParam(required = false) String operation,Model model) {**          System.out.println("Loding preloaded data Role List");          List<RoleDTO> l = new ArrayList<RoleDTO>();          RoleDTO admin = new RoleDTO();         admin.setId(1);         admin.setName("Admin");         l.add(admin);          RoleDTO member = new RoleDTO();         admin.setId(2);         admin.setName("Member");         l.add(member);          RoleDTO guest = new RoleDTO();         admin.setId(3);         admin.setName("Guest");         l.add(guest);          System.out.println("Size or Role List " + l.size());         model.addAttribute("roleList", l);      }     ..  **Q: When is preload method called?**  A: When @ModelAttribute is applied on a method of controller then this method will be invoked before any method mapped with @RequestMapping annotation. |

### Container Objects

|  |
| --- |
| Container objects HttpServletRequest, HttpServletResponse  and HttpSession can be sent to a Controller's method using method arguments.  Here is an example controller [ContainerObjectCtl](https://sites.google.com/site/sunraysspringmvc/source-code/src/in-co-sunrays-ctl/containerobjectctl-java" \t "_blank) that passes container objects to a controller method.  @Controller @RequestMapping(value = "/ContainerObj") public class [ContainerObjectCtl](https://sites.google.com/site/sunraysspringmvc/source-code/src/in-co-sunrays-ctl/containerobjectctl-java" \t "_blank) {      @RequestMapping(value = "/Request", method = RequestMethod.GET)     public String injectRequest(**HttpServletRequest request, HttpServletResponse response**, Model model) {         //...         return "[ContainerObjectView](https://sites.google.com/site/sunraysspringmvc/source-code/webapp/pages/containerobjectview-jsp" \t "_blank)";     }      @RequestMapping(value = "/Session", method = RequestMethod.GET)     public String injectSession(**HttpSession session**, Model model) {         //...         return "[ContainerObjectView](https://sites.google.com/site/sunraysspringmvc/source-code/webapp/pages/containerobjectview-jsp" \t "_blank)";     }      @RequestMapping(value = "/All", method = RequestMethod.GET)     public String injectAll(**HttpServletRequest request,HttpServletResponse response, HttpSession session**, Model model) {         //...         return "[ContainerObjectView](https://sites.google.com/site/sunraysspringmvc/source-code/webapp/pages/containerobjectview-jsp" \t "_blank)";     }  } |

### Interceptors

|  |
| --- |
| An Interceptors is the object that intercepts an incoming request and performs operation before sending request to the target controller and perform some operations after receiving response from target controller.  An Interceptor is mapped with a URI pattern. All requests, matching URI pattern will be intercepted by the Interceptor.  Multiple interceptors can be mapped with the single URI pattern. |

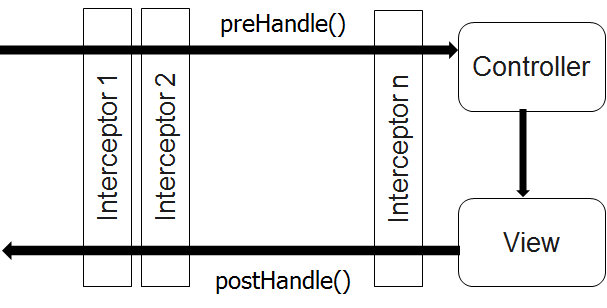


Figure: Interceptor communication

You can create an Interceptor by two ways

1. Implement HandlerInterceptor interface
2. Inherit  HandlerInterceptorAdapter abstract class. HandlerInterceptorAdapter  implements HandlerInterceptor interface.

This HandlerInterceptor  declares three methods:

1. preHandle(..) is called before the actual target controller is executed
2. postHandle(..) is called after the target controller  is executed. and
3. afterCompletion(..) is called after the complete request has executed.

The preHandle(..) method returns a boolean value. You can use this method to break or continue the processing of the interceptor execution chain. When this method returns true, the handler execution chain will continue; when it returns false, the DispatcherServlet stops further chain execution and transfers control to the desired page.

Here is an example interceptor [TimeLoggerInt](https://sites.google.com/site/sunraysspringmvc/source-code/src/in-co-sunrays-ctl/timeloggerint-java" \t "_blank) that intercepts  incoming requests at URI patteren /UrlMapping/\*  and log the pre and post processing time-stamp.  
  
  
public class [TimeLoggerInt](https://sites.google.com/site/sunraysspringmvc/source-code/src/in-co-sunrays-ctl/timeloggerint-java" \t "_blank) extends HandlerInterceptorAdapter {  
  
    /\*\*  
     \* Called before request is executed by target controller.  
     \*/  
    @Override  
    public boolean preHandle(HttpServletRequest request,  
            HttpServletResponse response, Object handler) throws Exception {  
  
        System.out.println("TimeLoggerInt:  preHandle()  : " + new Date());  
        System.out.println("request uri is :: " + request.getRequestURI());  
  
        return true;  
    }  
  
    /\*\*  
     \* Called after request is executed by target controller.  
     \*/  
  
    @Override  
    public void postHandle(HttpServletRequest request,HttpServletResponse response, Object handler, ModelAndView modelAndView) throws Exception {  
        System.out.println("TimeLoggerInt: postHandle() ! " + new Date());  
    }  
  
    /\*\*  
     \* Called when request processing is completely done.  
     \*/  
  
    @Override  
    public void afterCompletion(HttpServletRequest request,HttpServletResponse response, Object handler, Exception ex)  
            throws Exception {  
        System.out.println("TimeLoggerInt: after Completion() ! " + new Date());  
    }  
}

Interceptor will be configured in [configuration](https://sites.google.com/site/sunraysspringmvc/source-code/webapp/web-inf/sunraysweb-servlet-xml)file with the help of <mvc:interceptor> tag.   
  
Here in below example configuration mapping, Interceptor [TimeLoggerInt](https://sites.google.com/site/sunraysspringmvc/source-code/src/in-co-sunrays-ctl/timeloggerint-java" \t "_blank) will intercept all requests coming to URI pattern "/UrlMapping/\*".  Interceptor [FrontCtlInt](https://sites.google.com/site/sunraysspringmvc/source-code/src/in-co-sunrays-ctl/frontctlint-java" \t "_blank) will intercept all incoming requests since it is mapped to "/\*" URI Pattern.

<mvc:interceptors>  
    <mvc:interceptor>  
        <mvc:mapping path="/\*" />  
        <bean class="in.co.sunrays.ctl.FrontCtlInt" />  
    </mvc:interceptor>  
  
    <mvc:interceptor>  
        <mvc:mapping path="/UrlMapping/\*" />  
        <bean class="in.co.sunrays.ctl.TimeLoggerInt" />  
    </mvc:interceptor>  
</mvc:interceptors>

## Front Controller

This is the special kind of design pattern that intercepts all request and check authentication and authorization.  It is responsible to stop request those are not authorized or authenticated.

Authentication is mainly to check whether user is logged in or not. If user is logged in then only allow user to access application functionality. Authorization is basically to provide role based access.  
  
  
public class [FrontCtlInt](https://sites.google.com/site/sunraysspringmvc/source-code/src/in-co-sunrays-ctl/frontctlint-java" \t "_blank) extends HandlerInterceptorAdapter {  
    @Override  
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {  
  
        HttpSession session = request.getSession();  
  
        if (session.getAttribute("user") == null) {  
  
            System.out.println("FrontCtlInt: User is not logged in ");  
  
            /\*  
             \* If user is not logged then forward it Login page and return false  
             \* value to interceptor. I  
             \*/  
  
            response.sendRedirect("Login.html");  
            return false;  
  
        }  
  
        return true;  
    }  
    //..

Q: How do you create interceptor ?

Q: Which method will you override in interceptor ?

Q: What are the difference between preHandle() and postHandle()  methods?

Q: Can you map one interceptor to one controller?

Q: Can you map one interceptor to multiple controllers?

Q: Can you map multiple interceptors to one controller?

Q: Can you map multiple interceptors to mutiple controllers?

**Q: Why did you created interceptor in your application?**

A: We implemented Front Controller in our application using Interceptor. Front Controller does authentication and authorization for our application.

### Internationalization & Localization

|  |
| --- |
| Introduction Internationalization (i18n) and localization  (L10n) are means of supporting multi-languages for multiple countries  or multiple regions within a country without changing code in the application.  The terms are frequently abbreviated to the numeronyms i18n (where 18 stands for the number of letters between the first  i and last n in internationalization) and L10n respectively, due to the length of the words. The capital L in L10n helps to  distinguish it from the lowercase i in i18n. Create Message resource files Create message resource files, one for each language in the root class path.  These files are suffixed by 2 character language  code say "en" for English, "hi" for Hindi, and "sp" for Spanish.  **1. [messages\_en.properties:](https://sites.google.com/site/sunraysspringmvc/source-code/src/messages_en-properties" \t "_blank)**  *message.view=Message from View*  *message.ctl=Message from Controller*  *message.error=Error Message*  **2. [messages\_hi.properties](https://sites.google.com/site/sunraysspringmvc/source-code/src/messages_hi-properties" \t "_blank):**  *message.view=Message from View \u0939\u093F\u0928\u094D\u0926\u0940*  *message.ctl=Message from Controller \u0939\u093F\u0928\u094D\u0926\u0940*  *message.error=Error Message \u0939\u093F\u0928\u094D\u0926\u0940*  Code suffixed by \u (*\u0939, \u093F, \u0928*) are representing unicode values of Hindi fonts. "हिन्दी" string is converted into  unicode values as  "*\u0939\u093F\u0928\u094D\u0926\u0940".*  Configuration  We have created message resource properties for our application. We need to declare these files in spring  configuration file. We will use class org.springframework.context.support.  ReloadableResourceBundleMessageSource to define the message resources.  <bean id="messageSource"   class="org.springframework.context.support.**ReloadableResourceBundleMessageSource**">      <property name="basename" value="**classpath:messages**" />      <property name="defaultEncoding" value="UTF-8" />  </bean>  Now we will provide a feature where user will be able to select language for the application. This is implemented using org.springframework.web.servlet.i18n.LocaleChangeInterceptor class. The LocaleChangeInterceptor class will intercept request and make i18n, L10n changes.  <bean id="localeChangeInterceptor" class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor">     <property name="paramName" value="lang" />  </bean>  User can pass request parameter ?lang=hi or ?lang=en to change the Locale and in turn his regional language.  User's selected Locale is saved in cookies for future subsequent requests. Class org.springframework.web.servlet.i18n.CookieLocaleResolver is used to store the locale changes in cookies.  Due to security reasons,  in few applications cookies are not supported in this case Locale is stored in user's Session object using class org.springframework.web.servlet.i18n.SessionLocaleResolver.  <bean id="localeResolver" class="org.springframework.web.servlet.i18n.SessionLocaleResolver">      <property name="defaultLocale" value="en" />  </bean>  Make following entries on \*-servlet.xml.  <!-- Defines the message resources --> <bean id="messageSource"    class="org.springframework.context.support.ReloadableResourceBundleMessageSource">    <property name="basename" value="classpath:messages" />    <property name="defaultEncoding" value="UTF-8" /> </bean>  <!-- Stores User locale in cookie for subsequent requests --> <bean id="localeResolver"    class="org.springframework.web.servlet.i18n.CookieLocaleResolver">    <property name="defaultLocale" value="en"></property> </bean>  <!--Intercepts request and make i18n, L10n changes. --> <mvc:interceptors>    <bean id="localeChangeInterceptor"       class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor">       <property name="paramName" value="lang" />    </bean> </mvc:interceptors> Change the View : The JSPs Now as we have created two message resources files and configured it in Spring MVC, we will use these messages in the JSP files.  Create a [LocaleCtl.java](https://sites.google.com/site/sunraysspringmvc/source-code/src/in-co-sunrays-ctl/localectl-java) that will forward to view [LocaleView.jsp](https://sites.google.com/site/sunraysspringmvc/source-code/webapp/pages/localeview-jsp" \t "_blank) that will contain following code:  <%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>  <%@taglib uri="http://www.springframework.org/tags" prefix="spring"%>  <%@ page isELIgnored="false"%>  <HTML>  <BODY>      <a href="${pageContext.request.contextPath}"> Home </a>        <form>           Change Language:          <select name="lang">              <option value="en">English</option>              <option value="hi">Hindi</option>          </select> <input type="submit" value="Go">      </form>      <H1 style="color: green">${message }</H1>  **<spring:message code="message.view" />**  </BODY>  </HTML>  We have used <spring:message> tag to display the message from resource bundle in above JSP.  One thing that you must note here in [LocaleView.jsp](https://sites.google.com/site/sunraysspringmvc/source-code/webapp/pages/localeview-jsp" \t "_blank) , we have defined <select name="lang"> list where you can select either Hindi or English language. The Form sets a request parameter ?lang=hi when user submit the Form. Note that spring identifies this request parameter by using LocaleChangeInterceptor interceptor and change the local accordingly. Also note that while configuring LocaleChangeInterceptor in spring-servlet.xml file, we have specified property “paramName” with value “lang”.  <bean id="localeChangeInterceptor" class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor">  **<property name="paramName" value="lang" />**  </bean>  Thus the Spring framework will look for a parameter called “lang” from request. Get Locale Messages in the Controller You may also need Locale messages in a Controller when you want to sent some notification messages to View from Controller in case of Business Validation fail or Success operation.  You nay do it by Auto wring MessageSource object and pass Locale parameter in controller method. Here is the example of [LocaleCtl.java](https://sites.google.com/site/sunraysspringmvc/source-code/src/in-co-sunrays-ctl/localectl-java) that gets a message for resource and set into Model object.  @Autowired  **private MessageSource messageSource;**  @RequestMapping(method = RequestMethod.GET)  public String display(Model model, **Locale locale**) {  **String msg = messageSource.getMessage("message.ctl", null, locale);**      model.addAttribute("message", msg);      return "LocaleView";  } Input validation error locale messages There is a very simple way to get error messages of a validated form-bean in localized language. Here we will have an example form-bean [LocaleForm](https://sites.google.com/site/sunraysspringmvc/source-code/src/in-co-sunrays-form/localeform-java" \t "_blank) that will apply input validations on an attribute and in case of validation fail error messages will be localized from resource bundle files.  Lets start applying localization on input validation error messages:  **1. Validation Form**  Here is an example [LocaleForm](https://sites.google.com/site/sunraysspringmvc/source-code/src/in-co-sunrays-form/localeform-java" \t "_blank) that is applying input validation @NotEmpty and @Size at "name" attribute.  public class LocaleForm {      @NotEmpty      @Size(min=5, max=20)      private String name =null;  **2. Controller**  Controller validate the form and stores form-bean in the Model by name "localeForm".  public class [LocaleCtl](https://sites.google.com/site/sunraysspringmvc/source-code/src/in-co-sunrays-ctl/localectl-java" \t "_blank){  @RequestMapping(method = RequestMethod.GET)  public String display(@ModelAttribute("**localeForm**") @Valid [LocaleForm](https://sites.google.com/site/sunraysspringmvc/source-code/src/in-co-sunrays-form/localeform-java" \t "_blank) form,BindingResult bindingResult, Model model, Locale locale) {  ..  **3. Entry in Resource files**  Now time to make entries for locale messages in resource files [messages\_en.properties](https://sites.google.com/site/sunraysspringmvc/source-code/src/messages_en-properties" \t "_blank) and [messages\_hi.properties](https://sites.google.com/site/sunraysspringmvc/source-code/src/messages_hi-properties" \t "_blank). Keys  of validation error messages must follow a name pattern (*Validation.formNameInModel.attributeName )* that consists of (1) Input Validation Type (2) Name of form-bean object in the Model (3) Attribute name on which validation is applied*.*  Here keys are formatted for @Size and @NonEmpty validation on "name" attribute.  NotEmpty.localeForm.name=Name is required.  Size.localeForm.name=Name must be between 5 to 20 characters.  **4. View JSP**  Here is JSP [LocaleView.jsp](https://sites.google.com/site/sunraysspringmvc/source-code/webapp/pages/localeview-jsp" \t "_blank) that does not need any special entry for i18n / L10n messages.  Name : <form:input path="name" /><form:errors path="name" cssClass="error" />  That's it, Now go ahead and  enjoy i18n/L10n messages in your application.  **FAQ**  Q: How do you configure i18n?  Q: What parameter will you pass in request object to change the language?  A: lang parameter.  Q: How do configure 'lang'  parameter?  Q: How many .properties file will you create?  A: It depends upon number of languages support.  Each language .properties file is suffix by language 2 character code.  Q: What are the location of .properties file?  Q: Root class path  Q: How do you configure in locale messages Form bean?  A:  By passing message key in validation annotation.  Q: How do you get language specific messages in a method of Controller?  A: Using MessageResource class.  Q: What is MessageResource class? |

### Tags

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Form Tag Library Spring 2 and above  provides a set of data binding-aware tags for handling form elements when using JSP and Spring Web  MVC. Each tag provides support for the set of attributes of its corresponding HTML tag counterpart. The tag-generated HTML  is HTML 4.01/XHTML 1.0 compliant.  Spring Tags binds the HTML form elements with Form bean attributes.  The tag library descriptor (TLD) is included in the  spring-webmvc.jar.  Configuration  The form tag library comes bundled in spring-webmvc.jar. The library descriptor is called spring-form.tld. Form  tag library can be imported by taglib directive in JSP using following syntax:  <%@taglib uri="http://www.springframework.org/tags/form" prefix="form"%>  where "form" is the tag name prefix that you will use to access tags of this library.  Here is a simple example that uses Spring's form tags. It binds HTML elements with LoginBean class attributes.  Controller  defines form-bean  "loginForm" as model attribute and <form:form commandName="**loginForm**">  tag binds this form-bean  with HTML elements using its "commandName" attribute.  <form:form action="login" method="post" commandName="**loginForm**">      Login: <form:input path="login" /><br>      Password: <form:password path="password" /><br>      <input type="submit" value="SignIn" name="operation">  </form:form>  @RequestMapping(value = "/login", method = RequestMethod.GET)  public String loginDisplay(@ModelAttribute("loginForm") LoginForm form, Model model)  {      return "LoginBindingView";  }  public class LoginForm {      private String login = null;      private String password = null;      private String operation = null;      //Setter and Getter methods  } The form tag This tag renders an HTML form and exposes the form-bean object's binding path to inner tags to bind html  elements and form-bean attributes. It puts the command object in the PageContext so that the command  object can be accessed by inner tags. All the other tags in this library are nested tags of the form tag.  In above example form tag exposes command object loginForm to its inner tags and binds its login and password attributes to the html form elements.  <form:form action="login" method="post" commandName="**loginForm**"> The input tag This tag renders an HTML input tag using the bound value and default type value text (type=text). Starting with Spring 3.1  you can use other types such HTML5-specific types like email, tel, date, and others. In above example we have used input elements:  <form:input path="login" /> The checkbox and checkboxes tag This tag renders an HTML input tag using "checkbox"  type.  A checkbox can be bound  to an attribute or  multiple checkboxes can be bound to an array of elements.   Checkboxes values can be coded in HTML or can be rendered  by an array, collection or collection map.  To explain this tag we are assuming [TagForm](https://sites.google.com/site/sunraysspringmvc/source-code/src/in-co-sunrays-form/tagform-java" \t "_blank) bean is containing attributes, those are bound to HTML form  elements in [TagView.jsp](https://sites.google.com/site/sunraysspringmvc/source-code/webapp/pages/tagview-jsp" \t "_blank). Controller [TagCtl](https://sites.google.com/site/sunraysspringmvc/source-code/src/in-co-sunrays-ctl/tagctl-java" \t "_blank) will transfer control to the [TagView.jsp](https://sites.google.com/site/sunraysspringmvc/source-code/webapp/pages/tagview-jsp" \t "_blank).  public class [TagForm](https://sites.google.com/site/sunraysspringmvc/source-code/src/in-co-sunrays-form/tagform-java" \t "_blank) {       private String name = null;       private boolean agreed = false;       private String[] selectedSkills = null;       private ArrayList<String> selectedCourses = null;       private String[] selectedLanguages = null;  public class [TagCtl](https://sites.google.com/site/sunraysspringmvc/source-code/src/in-co-sunrays-ctl/tagctl-java" \t "_blank) {       @ModelAttribute       public void preLoad(@ModelAttribute("tagForm") TagForm form, Model model) {            // Set Boolean value            form.setAgreed(true);            // Set list of elements as Array            String[] str = { "Java", "PHP", "Android" };            model.addAttribute("skillArray", str);            // Create Collection            ArrayList<String> courses = new ArrayList<String>();            courses.add("BE");            courses.add("MCA");            courses.add("MBA");            model.addAttribute("coursesList", courses);            // Create Map            HashMap<String, String> languages = new HashMap<String, String>();            languages.put("en", "English");            languages.put("hi", "Hindi");            languages.put("sp", "Spanish");            model.addAttribute("languagesMap", languages);    //..       }  **1) Render simple checkbox and bind with form-bean attribute:**  Below tag will render HTML checkbox. If checkbox is selected then it will set value "Yes" to name attribute  otherwise value will be null.  Want Java? <form:checkbox path="name" value="Yes"/>  Below tag will bind boolean value. If selected then value of agreed attribute in form bean will be true  otherwise false.  Agreed:<form:checkbox path="agreed" />  **2) Bind multiple checkboxes of same name to a String array:**  Below example will set selected values to selectedSkills[] array attributes in  form bean.  <form:checkbox path="selectedSkills" value="Java" /> Java  <form:checkbox path="selectedSkills" value="PHP" /> PHP  <form:checkbox path="selectedSkills" value="Android" /> ANDROID  **3) Render checkboxes from String Array or Collection:**  You can render multiple checkboxes  from a string array, stored as Model attribute.  Here model attribute  skillArray is used to render checkbox items.  <form:checkboxes path="selectedSkills" items="${skillArray}" />  It will create checkboxes just like example # 2  Similarly you render checkbox items from a Collection  <form:checkboxes path="selectedCourses" items="${coursesList}" />  **4) Render checkboxes from a collection map:**  You can also render checkbox items from a collection map.  Collection Keys will become the checkbox values  to be select. Collection values will become label of checkboxes.  <form:checkboxes path="selectedLanguages" items="${languagesMap}" /> The select tag This tag renders an HTML select element. It binds selected value to the the form-bean attribute. It uses nested option and options tags.  Select elements can be rendered  from an array, collection and collection map. If select list is rendered from array or  collection then label and selected value will be same. If select list is rendered from collection map then map  key will become selected value and map value will become label of list.  **1) Render Select element from array, collection or map:**  <form:select path="skill" items="${skillArray}"/>  <form:select path="course" items="${coursesList}"/>  <form:select path="lang" items="${languagesMap}"/>  **2) Use nested tag option and options:**  Here is a simple list:  <form:select path="gender">       <form:option value="Male"/>       <form:option value="Female"/>  </form:select>  Here list is created from a collection with an additional option value "--Please Select":  <form:select path="course">       <form:option value="-" label="--Please Select"/>       <form:options items="${coursesList}" />  </form:select>  **3) Render Select element from a collection of bean:**  List items can be rendered from a java bean collection. Tag attributes  itemValue and itemLabel simply refer  to bean properties used to make values and labels of list.  Here is an example that renders list from a collection of java-bean [RoleDTO](https://sites.google.com/site/sunraysspringmvc/source-code/src/in-co-sunrays-dto/roledto-java" \t "_blank)   Collection is set to Model from  [TagCtl](https://sites.google.com/site/sunraysspringmvc/source-code/src/in-co-sunrays-ctl/tagctl-java) and list is shown at [TagView](https://sites.google.com/site/sunraysspringmvc/source-code/webapp/pages/tagview-jsp" \t "_blank).  <form:select path="role" items="${roleList}" itemValue="id"  itemLabel="name" />  public class TagCtl {       @ModelAttribute       public void preLoad(@ModelAttribute("tagForm") TagForm form, Model model) {    //....            ArrayList<RoleDTO> roleList = new ArrayList<RoleDTO>();            RoleDTO role1 = new RoleDTO();            role1.setId(1);            role1.setName("Admin");            roleList.add(role1);            RoleDTO role2 = new RoleDTO();            role2.setId(2);            role2.setName("Manger");            roleList.add(role2);            RoleDTO role3 = new RoleDTO();            role3.setId(2);            role3.setName("Customer");            roleList.add(role3);            model.addAttribute("roleList", roleList);        }     //..  }  The items attribute is typically populated with a collection or array of item objects. The radio tag This tag renders an HTML input tag with type radio. It binds radio element's value to the the form-bean attribute.  Radio HTML elements can be rendered  from an array, collection or collection map. If radio input tags are  rendered from an array or a collection then label and input element's value will be same. If radio input tags are  rendered from a collection map then map key will become value of radio input tag and map value will become  label of radio input tag.  **1) Render simple radio HTML input element**  Below tag will render a simple radio HTML element and bind it with form-beam attribute "gender":  <form:radiobutton path="gender" value="Male" /> Male  <form:radiobutton path="gender" value="Female" /> Female  **2) Render radio elements from array or collection**  You can render multiple radio elements from a string array or collection, stored as Model attribute. Here model  attribute skillArray and coursesList are used to render radio elements:  <form:radiobuttons path="skill" items="${skillArray}"/>  <form:radiobuttons path="course" items="${coursesList}"/>  **3) Render radio elements from collection map**  You can also render radio input elements from a collection map.  Collection Keys will become the value of radio  tag. Collection values will become label of radio input elements.  <form:radiobuttons path="lang" items="${languagesMap}"/>  See [TagView.jsp](https://sites.google.com/site/sunraysspringmvc/source-code/webapp/pages/tagview-jsp" \t "_blank) for complete example. The textarea tag This tag renders an HTML textarea. <form:textarea path="address" rows="3" cols="20" /> The hidden tag This tag renders an HTML input tag with type hidden using the bound value. <form:hidden path="house" /> The errors tag This tag renders field errors in an HTML span tag. It provides access to the errors created in your controller or those were  created by any validators associated with your controller.  <form:input path="lastName" />  <form:errors path="lastName" />  You can display entire list of errors messages for the current view page using wildcard mapping in path attribute:  <form:errors path="\*" />  Attribute path may have following values:   * path="\*" - displays all errors * path="lastName" - displays all errors associated with the lastName field * if path is omitted - object errors only are displayed  JSTL TJSTL Tags  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | The JavaServer Pages Standard Tag Library (JSTL) is a collection of useful JSP tags which encapsulates core functionality common to many JSP applications.  Spring MVC support JSTL.  JSTL has support for control statements  such as iteration and if condition. It supports tags for manipulating XML documents, internationalization tags, and SQL tags. It also provides a framework for integrating existing custom  tags with JSTL tags.  JSTL tags can be categorized into following libraries:   1. Core Tags 2. Formatting tags 3. SQL tags 4. XML tags 5. JSTL Functions  Core Tag Library These tags are most frequently used and contains tags for control statements.  Using these tags you can  implement conditional  and iterative presentation logic. Iteration is mostly used to render list data on Views.  You can import Core tag library using following JSP taglib directive:  <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>  Followings are JSTL tags and their description:   |  |  | | --- | --- | | **Tag** | **Description** | | **<c:out >** | Like <%= ... >, but for expressions. | | <c:set > | Sets the result of an expression evaluation in a 'scope' | | <c:remove > | Removes a scoped variable (from a particular scope, if specified). | | <c:catch> | Catches any Throwable that occurs in its body and optionally exposes it. | | **<c:if>** | Simple conditional tag which evaluates its body if the supplied condition is true. | | **<c:choose>** | Simple conditional tag that establishes a context for mutually exclusive conditional operations, marked by <when> and <otherwise> | | **<c:when>** | Subtag of <choose> that includes its body if its condition evaluates to 'true'. | | **<c:otherwise >** | Subtag of <choose> that follows <when> tags and runs only if all of the prior conditions evaluated to 'false'. | | <c:import> | Retrieves an absolute or relative URL and exposes its contents to either the page, a String in 'var', or a Reader in 'varReader'. | | **<c:forEach >** | The basic iteration tag, accepting many different collection types and supporting subsetting and other functionality . | | <c:forTokens> | Iterates over tokens, separated by the supplied delimeters. | | <c:param> | Adds a parameter to a containing 'import' tag's URL. | | <c:redirect > | Redirects to a new URL. | | <c:url> | Creates a URL with optional query parameters |   Most frequently tags are marked **bold** in table. The <c:out > Tag The <c:out> tag displays the result of an expression, similar to the way <%= %> works with a difference  that <c:out> tag lets you use the simpler "." notation to access properties.  For example, to access customer.address.street just use tag is  <s:out value="${customer.address.street}" />  <s:out value="${marksheet.name}" /> The <c:if> Tag The <c:if> tag evaluates an expression and displays its body content only if the expression evaluates to true.   |  |  |  |  | | --- | --- | --- | --- | | **Attribute** | **Description** | **Required** | **Default** | | test | Condition to evaluate | Yes | None | | var | Name of the variable to store the condition's result | No | None | | scope | Scope of the variable to store the condition's result | No | page |   Example # 1:  <c:set var="price" scope="session" value="${10\*110}"/>  <c:if test="${price < 200}">     <p>You can buy pizza that's price is : <c:out value="${price}"/><p>  </c:if>  Example # 2:  List<CollegeDTO> list = list = service.search();  <c:if test="${**empty**list}">    <H2 style="color: red">Records not found</H2>  </c:if> The <c:forEach> Tag This tag iterates over a collection of objects. This is the alternate of Java  loop (for, while, or do-while) statements.  This has following attributes:   |  |  |  |  | | --- | --- | --- | --- | | **Attribute** | **Description** | **Required** | **Default** | | items | Information to loop over | No | None | | begin | Element to start with (0 = first item, 1 = second item, ...) | No | 0 | | end | Element to end with (0 = first item, 1 = second item, ...) | No | Last element | | step | Process every step items | No | 1 | | var | Name of the variable to expose the current item | No | None | | varStatus | Name of the variable to expose the loop status | No | None |   Example:  List<CollegeDTO> list = list = service.search();  <c:forEach items="${list}" var="college" varStatus="loop">       <tr>            <td><input type="checkbox" name="ids" value="${college.id}"></td>            <td>${loop.index}</td>            <td>${college.name}</td>            <td>${college.address}</td>       </tr>  </c:forEach> The <c:choose> Tag This tag works like a Java switch statement, it allows you to choose between a number of alternatives.  Where the switch statement has case statements, the <c:choose> tag has <c:when> tags. A a switch  statement has default clause to specify a default action and similar way <c:choose> has <c:otherwise> as  default clause.  **Attributes:**  The <c:choose> tag does not have any attribute.  The <c:when> tag has one attributes which is listed below.  The <c:otherwise> tag does not have any attribute.   |  |  |  |  | | --- | --- | --- | --- | | **Attribute** | **Description** | **Required** | **Default** | | test | Condition to evaluate | Yes | None |   <c:set var="money" scope="request" value="100"/>  <c:choose>      <c:when test="${money< 100}">          <B>You can buy regular Pizza</b>      </c:when>      <c:when test="${money> 100 }">          <B>You can buy regular Pizza with Cheese </b>      </c:when>      <c:otherwise>          <B>Would you like to have Bada Pav?</b>      </c:otherwise>  </c:choose> The <c:url> Tag This tag formats a URL into a string and stores it into a variable. This tag automatically performs URL rewriting  when necessary. The var attribute specifies the variable that will contain the formatted URL.  The JSTL url tag is just an alternative method of writing the call to the response.encodeURL() method. The only  real advantage the url tag provides is proper URL encoding, including any parameters specified by children param tag.  It has following attributes:   |  |  |  |  | | --- | --- | --- | --- | | **Attribute** | **Description** | **Required** | **Default** | | value | Base URL | Yes | None | | context | / followed by the name of a local web application | No | Current application | | var | Name of the variable to expose the processed URL | No | Print to page | | scope | Scope of the variable to expose the processed URL | No | Page |   Example:  <a href="<c:url value="/index.jsp"/>" >Home</a> | |

**RESTFul Web Services**

|  |
| --- |
| RESTFul API can be implemented using   1. Spring 3.0    1. @PathVariable and @ResponseBody annotation in Spring 3.0 2. Spring 4 and above    1. @RestController  in Spring 4 and above 3. JSON objects are return by JSON Controller   **Q: How do you create RestFul web services?**  A: Using rest controllers.  **Q: Which data is returned by RestFul web services?**  A: JSON data.  **Q: What are the differences between Normal and Rest controllers?**  A: Rest controllers always return response in JSON format whereas normal controllers may return  response into xml, html, tiles of any other format.  **Q: What are the differences between SOAP and Restfull services?**  A: Rest controllers always return response in JSON format whereas SOAP return XML.  **Q: How do you test restful controllers**  A: Using POSTMAN  **Q: How do map request JSON object to a Form bean?**  A: We use @RequestBody annotation  **Q: What is the function of @RequestBody annotation**  A: It maps JSON data from request body to Java bean.  **Q: What is the function of @ResponseBody annotation ?**  A: It converts Java bean into JSON object and send to Response body. |