Spring MVC

By

Apparao G

Advantages of Spring MVC:

- Web application framework that takes advantage of spring design principles
- Dependency Injection
- Interface-driven design
- Supports Restful URLs
- Supports to plug with other MVC frameworks like Struts, web works etc.
- POJO without being tied up with a framework
- Testing through Dependency Injection
- Binding of request data to domain objects
- Form validation
- Error handling
- Page workflow
- Flexible in supporting different view technologies like
    - JSP, Velocity, Excel, PDF

Spring MVC follows MVC 2 design pattern.

MVC (Model-View-Controller):

Advantages of MVC design pattern:

1. Clearly separates business, navigation and presentation logic.
2. Proven mechanism for building a thin, clean web-tier.

Controller

- Handles navigation logic and interacts with the service tier for business logic

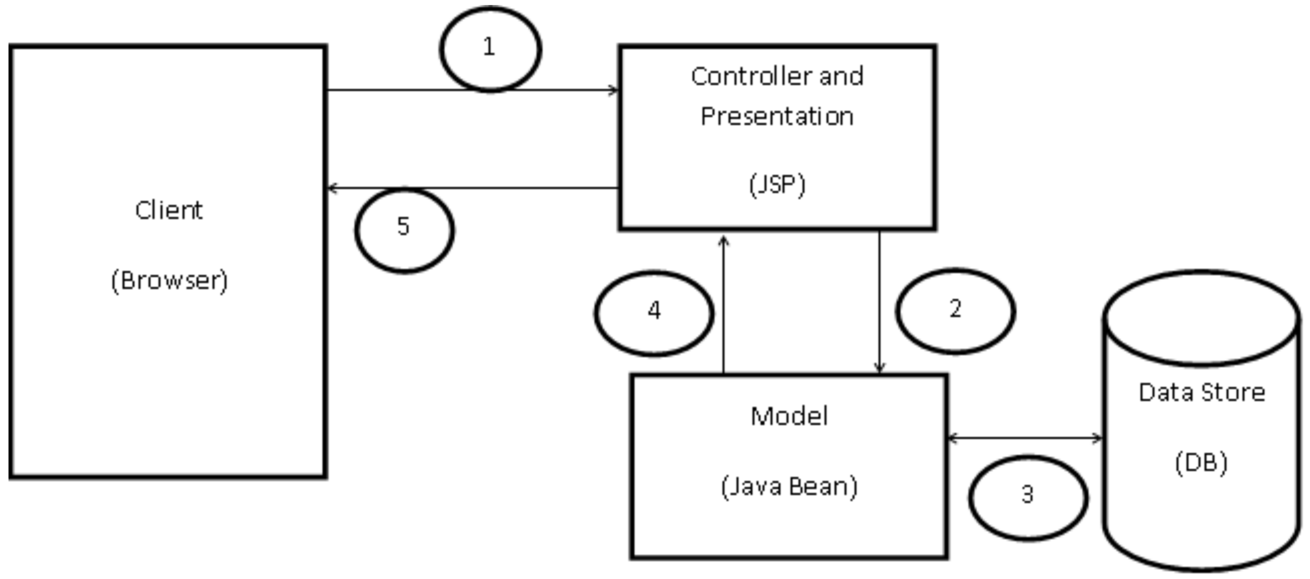
Model

- The contract between the Controller and the View
- Contains the data needed to render the View
- Populated by the Controller

View

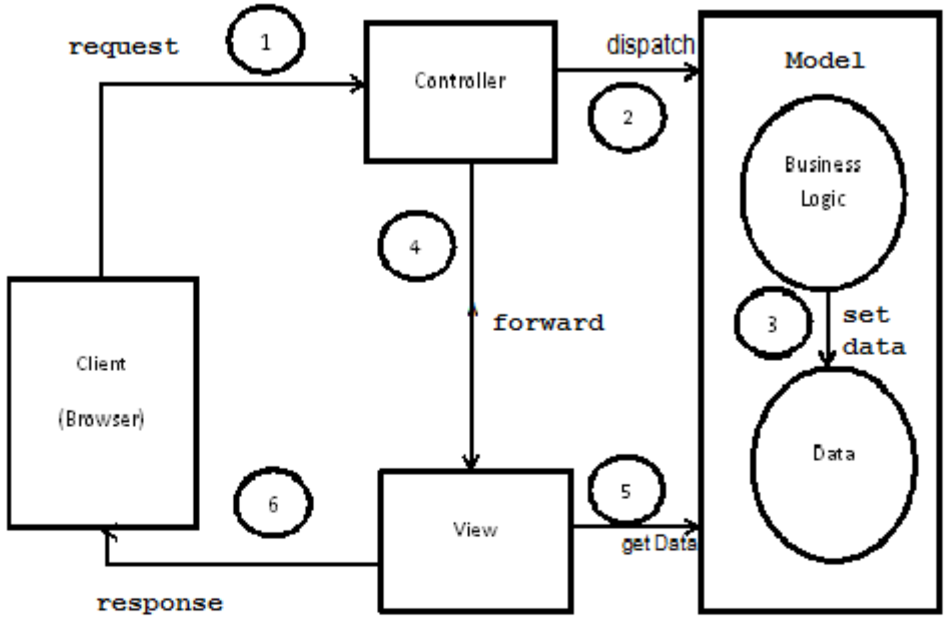
- Renders the response to the request
- Pulls data from the model

Model-1 Architecture:



- In this model, the client directly accesses the pages in the web container, and these pages service the entire request and send the response back.
- While servicing the client requests these pages generally use a model which represents the business logic for the application.
- These pages are usually implemented using JSP pages, sometimes servlets, and model as Beans.
- In this architecture, controlling and presentation are mixed into a single component, and hence this model architecture is also called as page-centric architecture.

Model-2 Architecture:



- In this model, the client cannot directly access the page, instead all the requests are made to a controller component generally implemented as a servlet.
- The controller component then uses the model (which is generally java bean or EJB) to process the client request, and then dispatches the request to view pages (implemented using JSP).
- These pages prepare the response and send it to the client.

Front Controller Design Pattern :

Context:

- We have decided to use the Model-View-Controller (MVC) pattern to separate the user interface logic from the business logic of a web application.
- We have reviewed the Page Controller pattern, but page controller classes have complicated logic, or our application determines the navigation between pages dynamically based on configuration rules.
- The Page Controller pattern describes a separate controller per logical page.

Problem:

We want to structure the controller for every complex Web applications in the best possible manner so that we can achieve reuse and flexibility while avoiding code duplication and decentralization problems in page controller.

Solution:

- Use a Front Controller as the initial point of contact for handling all related request.
- The Front Controller solves the decentralization problem present in the Page Controller by channeling all requests through a single controller, that is, it centralizes control logic otherwise duplicated.

The following functionalities need to be implemented into Front Controller:

- **Protocol Handling** is a process of handling protocol-specific request, which involves in resolving the request given in a specific protocol format and preparing a message in a specific protocol format.
- **Context Transformation** is a process of converting the protocol specific data into a more general form, like into our system-defined java bean object or into a general collection object.

- **Navigation** is a process that chooses the object for handling a particular request that can perform the core processing for this request and the view that can present the response to the client.

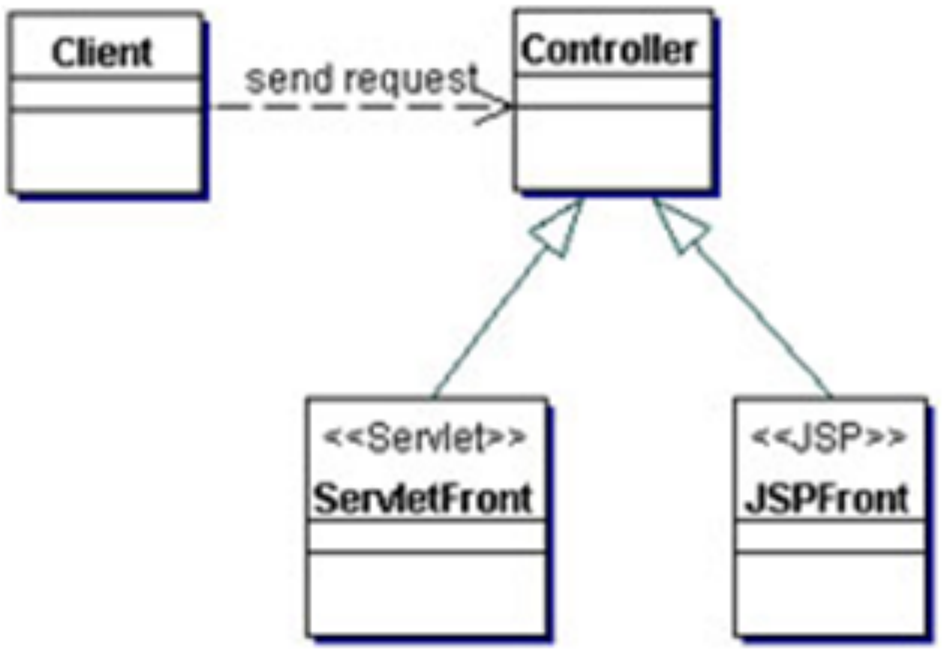
- **Dispatch** involves in dispatching the request from one part of the application to another, like from request handling object to view processing components.

Benefits:

- **Centralized control**: *Front Controller* coordinates all of the requests that are made to the Web application. The solution describes using a single controller instead of the distributed model used in *Page Controller*. This single controller is in the perfect location to enforce application-wide policies, such as security and usage tracking.
- **Thread-safety:** Because each request involves creating a new command object, the command objects themselves do not need to be thread safe. This means that you avoid the issues of thread safety in the command classes. This does not mean that you can avoid threading issues altogether, though, because the code that the commands act upon, the model code, still must be thread safe .
- **Configurability:** Only one front controller needs to be configured into the Web server; the handler does the rest of the dispatching. This simplifies the configuration of the Web server. Some Web servers are awkward to configure. Using dynamic commands enables you to add new commands without changing anything.

Class Diagram of Front Controller:

A servlet component implementing front controller is referred as Servlet Front, similarly with JSP it is known as JSP Front.



Issue in Using this Pattern:

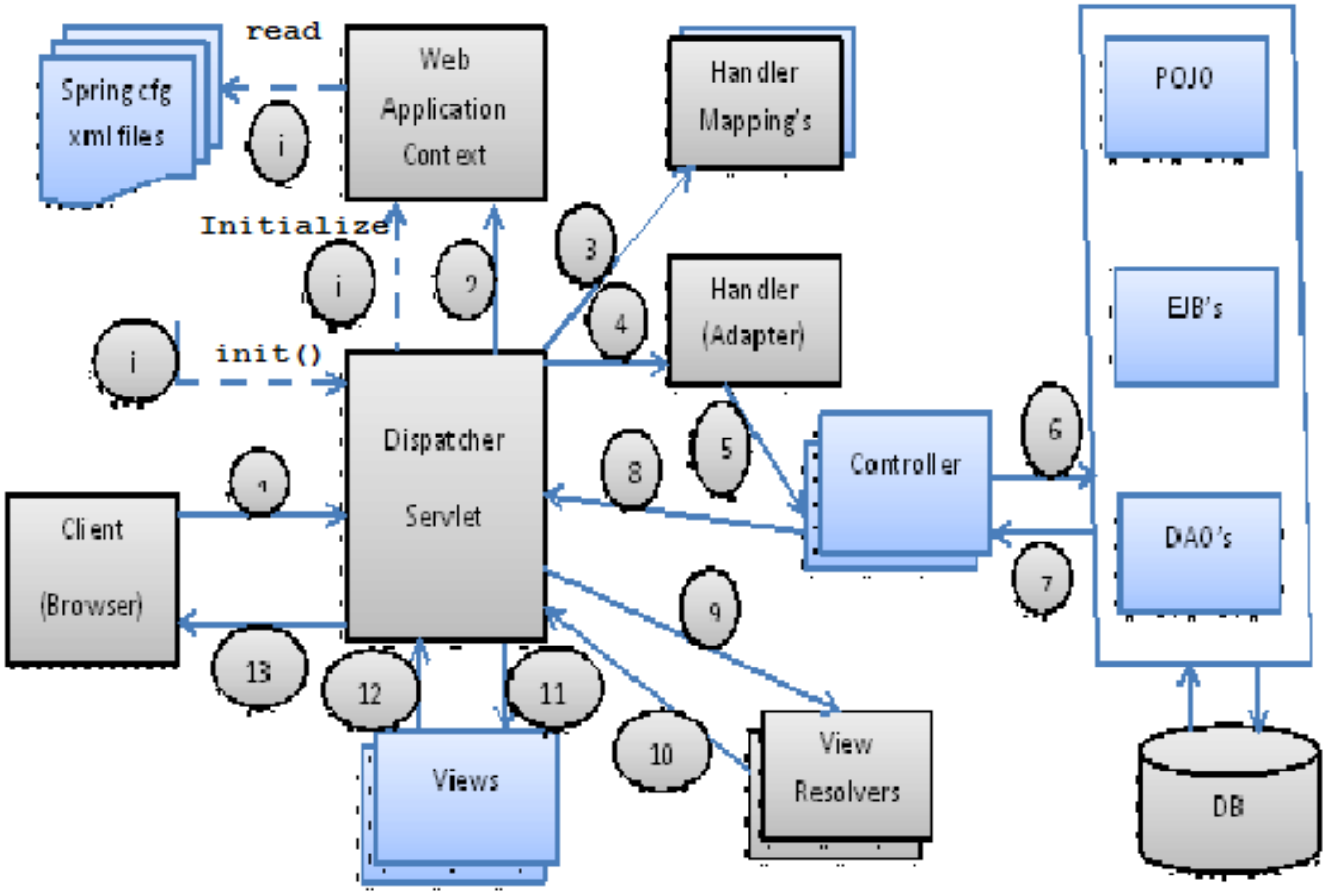
1. Performance considerations: Front Controller is a single controller that handles all requests for the web application. Here it requires determining the type of command that processes the request.

   If it must retrieve data from XML document (Configuration files) to make the decision, performance could be very slow as a result.

   However, this can be minimized by loading the configurations into memory at the time of initializing the application and avoiding the reading of the XML document multiple numbers of times.

2. **Increased complexity:** *Front Controller* is more complicated than *Page Controller.* It often involves replacing the built-in controller with a custom built *Front Controller*. Implementing this solution increases the maintenance costs and the time it takes for developers to orient themselves to the solution.

Spring MVC Architecture:



Note: In the below architecture the dotted arrows shows the initialization process and solid arrows shows the request process. And the sky colour elements are the elements which we need to program.

DispatcherServlet:

- *DispatcherServlet* acts as the Front Controller in spring web mvc.
- It is Configured in *web.xml*
- Loads Spring application context from XML configuration file
  - */WEB-INF/[servlet-name]-servlet.xml*
- Initializes *WebApplicationContext*
  - *WebApplicationContext* is bound into *ServletContext*
  - It is bound by default under the key DispatcherServlet.WEB_APPLICATION_CONTEXT_ATTRIBUTE.

- The locale resolver is bound to the request to enable elements in the process to resolve the locale to use when processing the request (rendering the view, preparing data, and so on). If you do not need locale resolving, you do not need it.
- The theme resolver is bound to the request to let elements such as views determine which theme to use. If you do not use themes, you can ignore it.
- If you specify a multipart file resolver, the request is inspected for multiparts; if multiparts are found, the request is wrapped in a MultipartHttpServletRequest for further processing by other elements in the process.

- The **DispatcherServlet** then invokes the HandlerChain which will execute the following:
  - Checks if there are any interceptors mapped and invokes the Pre Processing logic.
  - The controllers handler method will be invoked where the request is processed and the result is returned.
  - The mapped interceptors post processing logic will be invoked
- The **DispactherServlet** based on the result returned by the Controllers handlers method, the ResutlToViewNameTranslator component is invoked to generate the view name.
- The **view resolver** will then decide on what view needs to be rendered (JSP/XML/PDF/ VELOCITY etc.,) and then the result will be dispatched to the client.

Spring MVC Components:

HandlerMapping:

– Routing of requests to handlers

HandlerAdapter:

– Adapts to handler interface. Default utilizes *Controller*s

Helps the DispatcherServlet to invoke a handler mapped to a request regardless of the handler is actually invoked. For example, invoking an annotated controller requires resolving various annotations. Thus the main purpose of a HandlerAdapter is to shield the DispatcherServlet from such details.

HandlerExceptionResolver:

– Maps exceptions to error pages

– Similar to standard Servlet, but more flexible

Note: Handler exception resolvers that are declared in the WebApplicationContext pick up exceptions that are thrown during processing of the request. Using these exception resolvers allows you to define custom behaviors to address exceptions.

ViewResolver:

– Used to map logical View names to actual View implementations

MultipartResolver:

– Handling of file upload

LocaleResolver:

– Default uses HTTP accept header, cookie, or session

ModelAndView

- Created by the Controller
- Stores the Model data
- Associates a View to the request
  - Can be a physical View implementation or a logical View name

Handler Mapping:

- When the Dispatcher servlet receives the request it delegates the request to the HandlerMapping.
- HandlerMapping is responsible for mapping the incoming request to the handler that can handle the request.
- Spring provides built-in navigation strategies as determining the handler based on the request URL mapping which is again based on bean name.
- Apart from the built-in navigation strategies spring allows defining system-specific built strategy, which can be done by writing a class implementing the HandlerMapping interface.

The spring built-in HandlerMapping implementations are:

1. BeanNameUrlHandlerMapping
2. SimpleUrlHandlerMapping
3. ControllerClassNameHandlerMapping
4. CommonsPathMapHandlerMapping

BeanNameUrlHandlerMapping:

- The org.springframework.web.servlet.handler.BeanNameHandlerMapping is one of the implementation of HandlerMapping interface.This implementation defines the navigation strategy the maps the request URL's servlet-path to the bean names.
- The BeanNameHandlerMapping is the default handler mapping when no handler mapping is configured in the application context.

```
<bean id="/ login.spring"
      class="com.apparao.controller.LoginControllers"/>
<bean id="/ logout.spring"
      class="com.apparao.controller.LogoutControllers"/>
<bean id="handleMappingClass"
      class="org.springframework.web.servlet.handler.BeanNameUrlHandl
      erMapping" />
```

- According to the above configuration the request with the URL <context path>/login.spring will be handled by the LoginController and the request with the URL <context path>/logout.spring will be handled by the LogoutController.

SimpleUrlHandlerMapping:

- The org.springframework.web.servlet.handler.SimpleUrlHandlerMapping is one of the implementation of HandlerMapping interface. This implementation defines the navigation strategy the maps the request URL's servlet-path to the mapping configured. That is, it locates the handler(controller) by matching the request URL's servlet path with the key of the given properties or map.
- In this case need to inject properties or map object that describes the mappings between the request URL path and handler beans(controllers).The SimpleUrlHandlerMapping supports two options to configure the mapping-one to bean names and other to bean instances.

Option 1:

```
<bean id="loginController" class="com.apparao.controller.LoginControllers"/>
<bean id="logoutController" class="com.apparao.controller.LoginControllers"/>

<bean id="urlmappingClass" class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
        <property name="mappings">
                                        <props>
                            <prop key="/login.spring">loginController</prop>
                                              <prop key="/logout.spring">logoutController</prop>
                      </props>
        </property>
  </bean>
```

Option 2 :

```
<bean id="loginController" class="com.apparao.controller.LoginControllers"/>
<bean id="logoutController" class="com.apparao.controller.LogoutControllers"/>
<bean id="urlmapping" class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
        <property name="urlMap">
                      <map>
                              <entry key="/login.spring">
                                        <ref local="loginController" />
                              </entry>
                             <entry key="/logout.spring">
                                        <ref local="logoutController" />
                             </entry>
                      </map>
        </property>
        </bean>
```

ControllerClassNameHandlerMapping:

- The org.springframework.web.servlet.handler.ControllerClassNameHandlerMapping is one of the implementation of HandlerMapping interface.
- This handler mapping implementation is newly introduced in spring 2.0. The ControllerClassNameHandlerMapping follows a simple convention for generating URL path mappings.
- The convention for simple Controller implementations (those that handle a single request type) is to take the short name of the controller class, remove the 'Controller' suffix if it exists and return the remaining text, lowercased, as the mapping, with a leading /.
- For example if the controller class name is com.apparao.controller.LoginController then the path is /login*

```
<bean id="login" class="com.apparao.controller.LoginControllers"/>
<bean id="logout" class="com.apparao.controller.LogoutControllers"/>
<bean id="urlmapping"
    class="org.springframework.web.servlet.mvc.support.ControllerClassNameHandlerMapping" />
```

Controllers:

- Receives requests from DispatcherServlet and interacts with business tier. Implements the Controller interface
  - ModelAndView handleRequest(HttpServletRequest req, HttpServletResponse resp) throws Exception
- Returns ModelAndView object
- ModelAndView contains the model (a Map) and either a logical view name,
  - or implementation of View interface

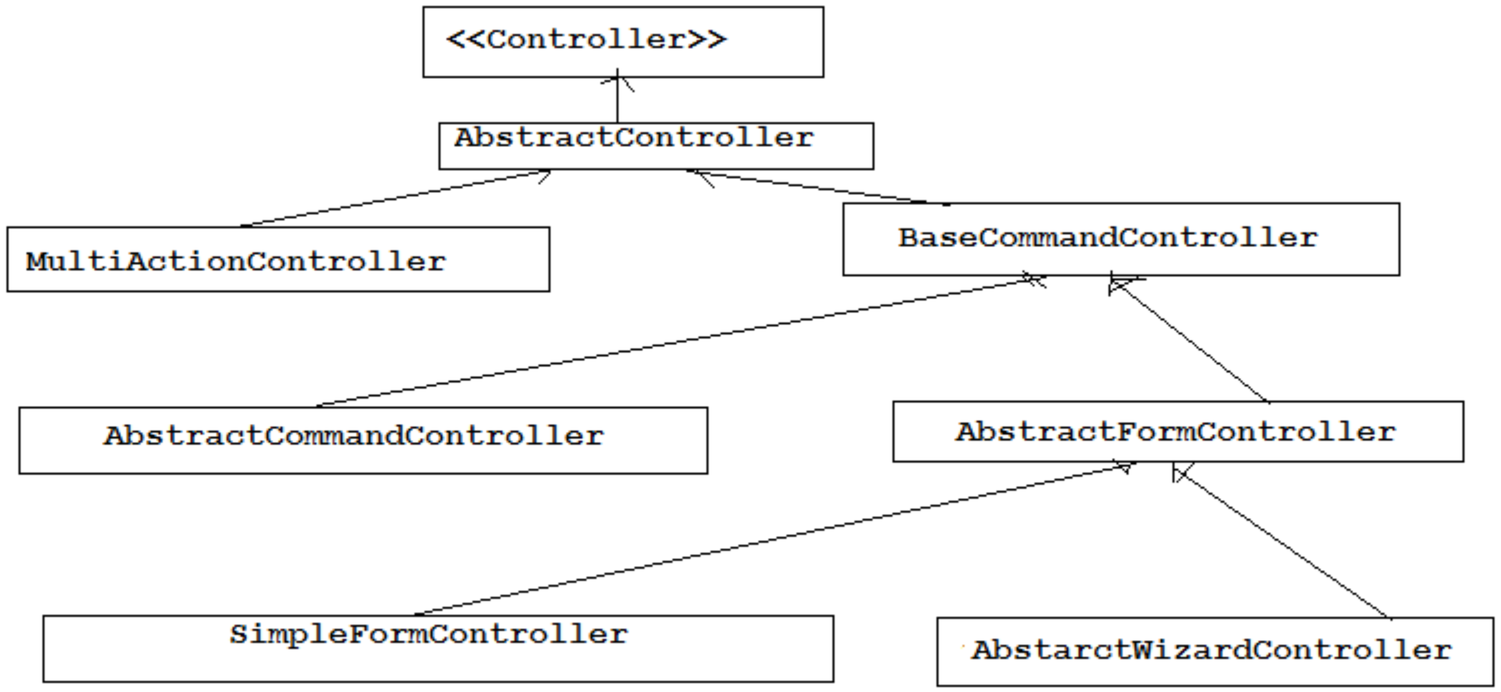
Spring provides different types of controller implementation classes.

AbstractConroller

- BaseCommandController
  - AbstractCommandController
  - AbstractFormController
    - SimpleFormController
    - AbstractWizardController

MultiActionController

ParameterizableViewController



<<Controller>>
AbstractController
MultiActionController
BaseCommandController
AbstractCommandController
AbstractFormController
SimpleFormController
AbstarctWizardController

AbstractController

Convenient superclass for controller implementations

Workflow

– handleRequest() of the controller will be called by the DispatcherServlet

– the located Controller is then responsible for handling the actual request and - if applicable - returning an appropriate ModelAndView

– handleRequest() method calls abstract method handleRequestInternal() , which should be implemented by extending classes to provide actual functionality to return ModelAndView objects.

BaseCommandController

- Controller implementation which creates an object (the command object) on receipt of a request and attempts to populate this object with request parameters
- Workflow

  – Since this class is an abstract base class for more specific implementation, it does not override the *handleRequestInternal*() method and also has no actual workflow.
- This controller is the base for all controllers wishing
  - – to populate JavaBeans based on request parameters
  - – to validate the content of such JavaBeans using Validators
  - – to use custom editors (in the form of PropertyEditors) to transform objects into strings and vice versa.

AbstractFormController

- Form controller that auto-populates a form bean from the request. This, either using a new bean instance per request, or using the same bean when the *sessionForm* property has been set to true.

- This class is the base class for both framework subclasses like SimpleFormController and AbstractWizardFormController.

SimpleFormControllers:

- Handles single page from input
- Most commonly used command controller
- Split into two workflows
  - Form request
    - Load form backing object and reference data
    - Show form view
  - Form submission
    - Load from backing object
    - Bind and validate from backing object
    - Execute submission logic
    - Show success view

MultiActionController

1. Controller implementation that allows multiple request types to be handled by the same class.
2. Subclasses of this class can handle several different types of request with methods of the form
   - (ModelAndView | Map | void) actionName(HttpServletRequest request, HttpServletResponse response);
3. Request to actionName mapping is resolved via methodNameResolver property in the configuration file.

MethodNameResolver

Implementations

InternalPathMethodNameResolver

- The method name is taken from the last part of the path
  /servlet/login.html -> login(...)
- Default behavior

ParameterMethodNameResolver

- The method name is taken from the specified request parameter
- The default parameter name is action

PropertiesMethodNameResolver

- The method name is resolved via <prop>

Example:

Step1: Configure the DispatcherServlet in web.xml

```
<servlet>
   <servlet-name>ds</servlet-name>
   <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
   <load-on-startup>10</load-on-startup>
 </servlet>
 <servlet-mapping>
   <servlet-name>ds</servlet-name>
   <url-pattern>*.spring</url-pattern>
 </servlet-mapping>
```

Step2: Configure the *ParameterMethodNameResolver* in ds-servlet.xml file.

```
<bean id="myMethodResolver"
      class="org.springframework.web.servlet.mvc.multiaction.ParameterMethodNameResolver">
<property name="paramName" value="submit" />
</bean>
```

Step3: Register your controller into ds-servlet.xml file.

```
<bean name="/mypath.spring" class="com.apparao.controller.MathsController" >
<property name="methodNameResolver" ref="myMethodResolver"/>
</bean>
```

Step 4: Develop the JSP Pages.

```
<html>
<body>
<%if(request.getAttribute("result")!=null){ %>
Result of previous request(<%=request.getParameter("submit") %>):
<b><%=request.getParameter("operand1") %>,<%=request.getParameter("operand2") %>
is <%=request.getAttribute("result") %></b>
<%}%>
<form action="mypath.spring" method="post">
Operand1 :<input type="text" name="operand1" /><br />
Operand2 :<input type="text" name="operand2"/><br />
<input type="submit" name="submit" value="add"/>
<input type="submit" name="submit" value="subtract"/>
<input type="submit" name="submit" value="product"/>
<input type="submit" name="submit" value="division"/>
</form>
  </body>
</html>
```

Step5: Implement logic in your controller class

```
public class MathsController extends MultiActionController{
   public ModelAndView add(HttpServletRequest request,HttpServletResponse response)
      throws Exception{
      String operand1 = request.getParameter("operand1");
      String operand2 = request.getParameter("operand2");
      int result = 0;
      if(operand1 != null && !operand1.equalsIgnoreCase("")&& operand2 != null && !operand2.equalsIgnoreCase("")){
         int op1 = Integer.parseInt(operand1);
         int op2 = Integer.parseInt(operand2);
         result = op1+op2;
      }
      return new ModelAndView("/Home.jsp","result",result+"");
   }

   public ModelAndView subtract(HttpServletRequest request,HttpServletResponse response)
   throws Exception{

      String operand1 = request.getParameter("operand1");
      String operand2 = request.getParameter("operand2");
      int result = 0;
      if(operand1 != null && !operand1.equalsIgnoreCase("")&& operand2 != null && !operand2.equalsIgnoreCase("")){
         int op1 = Integer.parseInt(operand1);
         int op2 = Integer.parseInt(operand2);
         result = op1-op2;
      }
   return new ModelAndView("/Home.jsp","result",result+"");
   }
}
```

View:

Renders the output of the request to the client

- Implements the View interface
- Built-in support for
  - JSP, XSLT, Velocity, Freemaker
  - Excel, PDF, JasperReports

View Resolvers

- Resolves logical view names returned from controllers into View objects
- Implements ViewResolver interface
  - View resolveViewName(String viewName, Locale locale) throws Exception
- Spring provides several implementations
  - UrlBasedViewResolver
  - InternalResourceViewResolver
  - BeanNameViewResolver
  - ResourceBundleViewResolver
  - XmlViewResolver

UrlBasedViewResolver

Simple implementation of the ViewResolver interface that effects the direct resolution of logical view names to URLs, without an explicit mapping definition. This is appropriate if your logical names match the names of your view resources in a straightforward manner, without the need for arbitrary mappings.

As an example, with JSP as a view technology, you can use the UrlBasedViewResolver. This view resolver translates a view name to a URL and hands the request over to the RequestDispatcher to render the view.

```
 <bean id="viewResolver"
class="org.springframework.web.servlet.view.UrlBasedViewResolver">
    <property name="viewClass" value="org.springframework.web.servlet.view.JstlView"/>
    <property name="prefix" value="/WEB-INF/jsp/"/>
    <property name="suffix" value=".jsp"/>
</bean>
```

When returning login as a logical view name, this view resolver forwards the request to the RequestDispatcher that will send the request to /WEB-INF/jsp/login.jsp.

This view resolver implementation additionally supports the feature of specifying the forward URLs and redirects the URLs. Forward URLs can be specified using ‘forward:’ as a prefix to the URL and redirect URLs can be specified using ‘redirect:’ as a prefix to the URL.

InternalResourceViewResolver:

This is a subclass of UrlBasedViewResolver that supports InternalResourceView (in effect, Servlets and JSPs) and subclasses such as JstlView and TilesView.

```
<bean id="viewResolver"
class="org.springframework.web.servlet.view. InternalResourceViewResolver ">
<property name="viewClass"
    value="org.springframework.web.servlet.view.InternalResourceView"/>
<property name="prefix" value="/WEB-INF/jsp/"/>
<property name="suffix" value=".jsp"/>
</bean>
```

The preceding configuration looks almost same as the UrlBasedViewResolver, the strategy used is also the same, but as described earlier this is a convenient class to use InternalResourceView and its subtypes, supporting to locate the internal resources (that is, resources in the web application)

Note: 1. Configuring a viewClass other than the InternalResourceView or its subclass to the InternalResourceViewResolver will throw an exception.

2. Here if the viewClass is not configured then it defaults to InternalResourceView.

BeanNameViewResolver

- The org.springframework.web.servlet.view.BeanNameViewResolver is a ViewResolver implementation that defines the view navigation strategy which maos the logical view name to the bean names in the application context to resolve the view.

  ```
  <bean id=”viewResolver”
  class=”org.springframework.web.servlet.vew.BeanNameViewResolver” />
  ```

- As discussed above , the BeanNameViewResolver locates the view by matching the logical view name with the bean name in the application context, but in case we have many views it is generally not a better practice to mix the views with the main application context beans.To manage these kinds of situations we can use XmlViewResolver.

XmlViewResolver:

This view resolver defines the view navigation strategy which maps the logical viewname to the bean names in the spring Beans XML file configured to its ‘location’ property for resolving the view.

```
<bean id=”viewResolver” class=”org.springframework.web.servlet.view.XmlViewResolver”>
<property name=”location” value=”/WEB-INF/views/MyViews.xml” />
</bean>
```

The ‘location’ attribute specifies the spring Beans XML configuration file in which the views have to be located. If this property is not configured then the XmlViewResolver by defaults looks for the definitions in ‘WEB-INF/views.xml’ file.

ResourceBundleViewResolver:

- The View definitions are kept in a separate configuration file
  - – You do not have to configure view beans in the application context file
  - _ You should configure the following code in <servlet-name>-servlet.xml file.
- Supports internationalization (I18N)

```
<bean id="viewResolver"
   class="org.springframework.web.servlet.view.ResourceBundleViewResolver">
<property name="basename" value="views"/>
</bean>
```

Example: views.properties

```
user.(class)=org.springframework.web.servlet.view.JstlView
user.url=/WEB-INF/views/user.jsp
userList.(class)=org.springframework.web.servlet.view.JstlView
userList.url=/WEB-INF/views/userList.jsp
```

In case of Tiles:

Configure the following code in <servlet-name>-servlet.xml file.

```
<bean id="tilesViewResolver" class="org.springframework.web.servlet.view.ResourceBundleViewResolver"
        p:basename="views" />
 <bean id="tilesConfigurer"
       class="org.springframework.web.servlet.view.tiles2.TilesConfigurer" p:definitions="/WEB-INF/context/tiles.xml"/>
```

views.properties

```
user.(class)=org.springframework.web.servlet.view.tiles2.TilesView
user.url=user
```

Following configuration is required in tiles.xml file

```
<tiles-definitions>
    <definition name="baseLayout" template="/WEB-INF/views/layout.jsp">
        <put-attribute name="title" value="" />
        <put-attribute name="header" value="/WEB-INF/views/header.jsp" />
        <put-attribute name="body" value="" />
        <put-attribute name="footer" value="/WEB-INF/views/footer.jsp" />
    </definition>
    <definition name="user" extends="baseLayout">
        <put-attribute name="title" value="User Login Page" />
        <put-attribute name="body"
                value="/WEB-INF/views/user.jsp" />
        <put-list-attribute name="styleSheets">
                <add-attribute value="css/style.css" />
        </put-list-attribute>
    </definition>
</tiles-definitions>
```

Configuring Multiple ViewResolver:

```
<bean  class=”org.springframework.web.servlet.view.XmlViewResolver”>
<property name=”location” value=”/WEB-INF/views/MyViews.xml” />
<property name=”order” value=”0” />
</bean>

<bean  class=”org.springframework.web.servlet.vew.BeanNameViewResolver” >
<property name=”order” value=”1” />
</bean>

<bean class="org.springframework.web.servlet.view. InternalResourceViewResolver ">
<property name="viewClass" value="org.springframework.web.servlet.view.InternalResourceView"/>
<property name="prefix" value="/WEB-INF/jsp/"/>
<property name="suffix" value=".jsp"/>
<property name=”order” value=”2” />
</bean>
```

The preceding code snippet shows how to define three view resolvers where the first preference is given to the XmlViewResolver. If it cannot resolve the view name then the next view resolver (that is BeanNameViewResolver) is used, and so on.

Note: If the ‘detectAllViewResolvers’ initialization parameter of DispatcherServlet is configured to false then multiple view resolvers are not detected instead only the bean with a name viewResolver is located as a viewResolver.

Spring MVC annotations:

@Controller

The @Controller annotation indicates that a particular class serves the role of a *controller*. There is no need to extend any controller base class or reference the Servlet API.

@RequestMapping

The @RequestMapping annotation is used to map URLs like ‘/login.html' onto an entire class or a particular handler method. Typically the type-level annotation maps a specific request path (or path pattern) onto a form controller, with additional method-level annotations 'narrowing' the primary mapping for a specific HTTP method request method ("GET"/"POST") or specific HTTP request parameters.

@RequestParam annotated parameters for access to specific Servlet request parameters. Parameter values will be converted to the declared method argument type.

The @RequestParam annotation is used to bind request parameters to a method parameter in your controller.

For example:

```
public String setupForm(@RequestParam("userId") int userId, ModelMap
    model) {
    // Implementation code
}
```

@ModelAttribute

Case1:

@ModelAttribute has two usage scenarios in controllers. When placed on a method parameter, @ModelAttribute is used to map a model attribute to the specific, annotated method parameter (see the newUser() method below).

```
@RequestMapping(value="/user")
public ModelAndView newUser(@ModelAttribute("userForm") UserForm
userForm,HttpServletRequest request){
  System.out.println("new User method");
  return new ModelAndView("user", "userForm", new UserForm());
}
```

Case2:

@ModelAttribute is also used at the method level to provide *reference data* for the model (see the addUser() method below). For this usage the method signature can contain the same types as documented above for the @RequestMapping annotation.

```
@ModelAttribute("userForm")
public ModelAndView addUser(BindingResult result, HttpServletRequest
    request){
        userList = userService.addUser(userForm);
      return new ModelAndView("userList", "userForm", userForm);
  }
```

@SessionAttributes

- The type-level @SessionAttributes annotation declares session attributes used by a specific handler. This will typically list the names of model attributes which should be transparently stored in the session or some conversational storage, serving as form-backing beans between subsequent requests.
- The following code snippet shows the usage of this annotation:

```
@Controller
 @RequestMapping(“/login.html")
@SessionAttributes(“userForm")
public class EditPetForm {
//  implementation logic
}
```

@InitBinder

Annotating controller methods with @InitBinder allows you to configure web data binding directly within your controller class. @InitBinder identifies methods which initialize the WebDataBinder which will be used for populating command and form object arguments of annotated handler methods.

The following example demonstrates the use of @InitBinder for configuring a CustomDateEditor for all java.util.Date form properties.

```
@Controller
public class MyFormController {
 @InitBinder
public void initBinder(WebDataBinder binder) { SimpleDateFormat dateFormat =
    new SimpleDateFormat("yyyy-MM-dd"); dateFormat.setLenient(false);
    binder.registerCustomEditor(Date.class, new CustomDateEditor(dateFormat,
    false));
}
    // implementation logic
}
```

Annotation-based controller configuration:

Step1: Configure the Dispatcher servlet in web.xml file

```
 <servlet>
    <servlet-name>dcma</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</
     servlet-class>
              <load-on-startup>10</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>dcma</servlet-name>
    <url-pattern>*.html</url-pattern>
   </servlet-mapping>
```

Step2: Create the file with name <servlet-name>-servlet.xml

Example: dcma-servlet.xml

Step3: Activates post-processors for annotation-based controller.

```
<context:annotation-config />
<context:component-scan base-package="com.ctl.dcma.controller" />
```

Step4: Configure the MultiactionController

```
<bean
class="org.springframework.web.servlet.mvc.multiaction.MultiActionControll
er" />
```

Step5: Configure the Handler

```
 <bean
class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodH
andlerAdapter" />
```

Step6: Configure the view resolver

```
<bean id="viewResolver"


    class="org.springframework.web.servlet.view.InternalResourceViewResolver"
    >
     <property name="prefix" value="/WEB-INF/views/" />
     <property name="suffix" value=".jsp" />
  </bean>
```

(or)

```
<bean
    class="org.springframework.web.servlet.view.InternalResourceViewResolver"
    p:prefix="/WEB-INF/jsp/“  p:suffix=".jsp"/>
```

Step7: If we are using different views configure the
ResourceBundleViewResolver

```
<bean id="jstlViewResolver"
    class="org.springframework.web.servlet.view.ResourceBundleViewResolver"
    p:basename="views" />
```

Step8: Create file with name views.properties and configure the url mapping

```
user.(class)=org.springframework.web.servlet.view.JstlView
user.url=WEB-INF/views/user.jsp
userList.(class)=org.springframework.web.servlet.view.JstlView
userList.url=WEB-INF/views/userList.jsp
```

Write your own jsp’s like user.jsp

Example: web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/javaee" xmlns:web="http://
        java.sun.com/xml/ns/javaee/web-app_2_5.xsd" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/
        javaee/web-app_2_5.xsd" id="WebApp_ID" version="2.5">
  <display-name>DCMA TOOL</display-name>
        <context-param>
                    <param-name>contextConfigLocation</param-name>
                    <param-value>/WEB-INF/context/applicationContext.xml</param-value>
  </context-param>
<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
  </listener>
  <servlet>
    <servlet-name>dcma</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
                    <load-on-startup>10</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>dcma</servlet-name>
    <url-pattern>*.html</url-pattern>
    <url-pattern>/dcma/*</url-pattern>
  </servlet-mapping>
        <welcome-file-list><welcome-file>index.jsp</welcome-file></welcome-file-list>
        <session-config><session-timeout>5</session-timeout> </session-config>
</web-app>
```

dcma-servlet.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:context="http://www.springframework.org/schema/context"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:p="http://www.springframework.org/schema/p"
   xmlns:mvc="http://www.springframework.org/schema/mvc"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context-2.5.xsd
       http://www.springframework.org/schema/mvc
       http://www.springframework.org/schema/mvc/spring-mvc.xsd">

  <bean id="jstlViewResolver" class="org.springframework.web.servlet.view.ResourceBundleViewResolver" p:basename="views" />
   <bean id="configurationLoader" class="org.springmodules.validation.bean.conf.loader.annotation.AnnotationBeanValidationConfigurationLoader"/>
   <bean id="validator" class="org.springmodules.validation.bean.BeanValidator" p:configurationLoader-ref="configurationLoader"/>

   <context:annotation-config />
   <context:component-scan base-package="com.ctl.dcma.controller" />

    <bean class="org.springframework.web.servlet.mvc.multiaction.MultiActionController" />

   <bean class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter" />

   <bean id="viewResolver"
      class="org.springframework.web.servlet.view.InternalResourceViewResolver">
      <property name="prefix" value="/WEB-INF/views/" />
      <property name="suffix" value=".jsp" />
   </bean>
  <bean id="multipartResolver" class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
      <property name="maxUploadSize" value="10000000"/>
  </bean>
  <bean id="localeChangeInterceptor" class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor">
           <property name="paramName" value="locale"/>
   </bean>
   <bean id="localeResolver" class="org.springframework.web.servlet.i18n.SessionLocaleResolver"/>

  <bean id="messageSource" class="org.springframework.context.support.ReloadableResourceBundleMessageSource">
      <property name="basenames">
         <list>
            <value>classpath:properties/messages</value>
            <value>classpath:properties/ValidationMessages</value>
         </list>
      </property>
      <property name="cacheSeconds">
         <value>10</value>
      </property>
  </bean>
 </beans>
```

```
<%@taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
</head>
<body>
<script type="text/javascript">
function addUser(){
		//alert("adduser");
   document.getElementById("userForm").action="userList.html";
   document.getElementById("userForm").target="_self";
   document.getElementById("userForm").method = "post";
   document.getElementById("userForm").submit();
}

</script>
<form:form method="post" id="userForm" name="userForm"
		modelAttribute="userForm">
		<div>
		<table border="0" cellspacing="0" cellpadding="0">
					<tr>
								<td ></td>
								<td >Enter User Id:</td>
								<td >&nbsp;</td>
								<td ><form:input path="userId" id="userId"
											style="width:255px"></form:input></td>
								<td class="error">&nbsp;<form:errors path="userId" /></td>
					</tr>
					<tr>
								<td ></td>
								<td >User Name:</td>
								<td >&nbsp;</td>
								<td ><form:input path="userName" id="userName"
											style="width:255px"></form:input></td></td>
								<td class="error">&nbsp;<form:errors path="userName" /></td>
					</tr>
					<tr>
								<td ></td>
								<td >User Role:</td>
								<td >&nbsp;</td>
								<td ><form:input path="role" id="role"
											style="width:255px"></form:input></td></td>
								<td class="error">&nbsp;<form:errors path="role" /></td>
					</tr>
					<tr><td colspan="1"><input type="button" value="Add User" onclick="javascript:addUser();"/></td>
					<td colspan="2">&nbsp;</td></tr>
		</table>
		</div>
</form:form>
</body>
</html>
```

userList.jsp

```
<%@taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>User List</title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
<script type="text/javascript">
function addUser(){
		//alert("adduser");
   document.getElementById("userForm").action="user.html";
   document.getElementById("userForm").target="_self";
   document.getElementById("userForm").method = "post";
   document.getElementById("userForm").submit();
}
</script>
</head>
<body>

<form:form method="post" id="userForm" name="userForm"
		modelAttribute="userForm">
		<h1><b>List of User Details</b></h1>
		<div>
		<table border="1" cellspacing="0" cellpadding="0" width="400px">
			<tr>
				<th width="100px">User ID</th>
				<th width="150px">User Name</th>
				<th width="150px">Role</th>
			</tr>
			<c:choose>
				<c:when test="${userForm.userList ne null}">
					<c:forEach items="${userForm.userList}" var="userObject">
						<tr >
							<td width="100px"><c:out value="${userObject.userId}" /></td>
							<td width="150px"><c:out value="${userObject.userName}" /></td>
							<td width="150px"><c:out value="${userObject.role}" /></td>
						</tr>

					</c:forEach>
				</c:when>
				<c:otherwise>
					<tr>
						<td  colspan="4" height="60" align="center"><font color="red">No User are available</font></td>
					</tr>
				</c:otherwise>
			</c:choose>

		</table>
		</div>
		<div class="clear"></div>
		<div><input type="button" value="Add New User" onclick="javascript:addUser();"/></div>
</form:form>
</body>
</html>
```

```
@Controller
public class UserController {
    @Autowired
    public UserService userService;

    @RequestMapping(value="/userList",method=RequestMethod.POST)
    public ModelAndView addUser(@ModelAttribute("userForm") UserForm userForm,
            BindingResult result, HttpServletRequest request){
        List<User> userList = null;
        try {
            if(userForm != null && userForm.getUserId()!= null){

                userService.addUser(getUser(userForm));
             }
            userList = userService.getAllUsers();
            userForm.setUserList(userList);
        } catch (Exception e) {
            e.printStackTrace();
        }

        return new ModelAndView("userList", "userForm", userForm);
    }

    @RequestMapping(value="/user")
    public ModelAndView newUser(@ModelAttribute("userForm") UserForm userForm,HttpServletRequest request){
        System.out.println("new User method");
        return new ModelAndView("user", "userForm", new UserForm());
    }

    private User getUser(UserForm userForm) {
        User user = new User();
        if(userForm != null){
            user.setUserId(userForm.getUserId());
            user.setUserName(userForm.getUserName());
            user.setRole(userForm.getRole());
        }
        return user;
    }

}
```

Service Layer:

UserService.java

```
package com.ctl.dcma.service;
import java.util.List;
import com.ctl.dcma.exception.DcmaServiceException;
import com.ctl.dcma.model.User;
public interface UserService {
    void addUser(User user)throws DcmaServiceException;
    List<User> getAllUsers()throws DcmaServiceException;
}
```

UserServiceImpl.java

```
@Service
public class UserServiceImpl implements UserService {
    @Autowired
    public UserDao userDao;

    @Transactional(propagation = Propagation.REQUIRED,isolation = Isolation.DEFAULT,readOnly = false )
    public void addUser(User user) throws DcmaServiceException {
        try {
            userDao.addUser(user);
        } catch (DcmaDAOException dse) {
           throw new DcmaServiceException(dse.getMessage(), dse);
        }

    }

    @Override
    public List<User> getAllUsers() throws DcmaServiceException {

        try {
            return userDao.getAllUsers();
        } catch (DcmaDAOException dse) {
            throw new DcmaServiceException(dse.getMessage(), dse);
        }
    }

}
```

Dao Layer:

BaseDao.java

```
package com.ctl.dcma.commons;
import javax.annotation.PostConstruct;
import org.hibernate.SessionFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.orm.hibernate3.support.HibernateDaoSupport;
import org.springframework.stereotype.Repository;

@Repository
public class BaseDao extends HibernateDaoSupport{

    @Autowired
    public SessionFactory sessionFactory;

    @PostConstruct
     public void init(){
        System.out.println("in side init in base dao:"+sessionFactory);
        setSessionFactory(sessionFactory);
    }

}
```

UserDao.java

```
package com.ctl.dcma.dao;

import java.util.List;

import com.ctl.dcma.exception.DcmaDAOException;
import com.ctl.dcma.model.User;

public interface UserDao {

    void addUser(User user)throws DcmaDAOException;

    List<User> getAllUsers()throws DcmaDAOException;

}
```

UserDaoImpl.java

```
package com.ctl.dcma.dao;

import java.util.List;

import org.springframework.dao.DataAccessException;
import org.springframework.stereotype.Repository;

import com.ctl.dcma.commons.BaseDao;
import com.ctl.dcma.exception.DcmaDAOException;
import com.ctl.dcma.model.User;

@Repository
public class UserDaoImpl extends BaseDao implements UserDao {

    @Override
    public void addUser(User user) throws DcmaDAOException {

        try {
            getHibernateTemplate().save(user);
        } catch (DataAccessException dae) {
           throw new DcmaDAOException(dae.getMessage(), dae);
        }
    }

    @Override
    public List<User> getAllUsers() throws DcmaDAOException {
        List<User> userList = null;
        try {
            userList =  getHibernateTemplate().find("from com.ctl.dcma.model.User");
        } catch (DataAccessException dae) {
            throw new DcmaDAOException(dae.getMessage(), dae);
        }
       return userList;
    }

}
```

Model object: User .java

```
package com.ctl.dcma.model;
import java.io.Serializable;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;
@Entity
@Table(name = "RX.user_details")
public class User implements Serializable {
    @Id
    @Column(name = "user_id", length = 10)
    private int userId;

    @Column(name = "user_name", length = 20)
    private String userName;

    @Column(name = "role", length = 20)
    private String role;

    public int getUserId() {
        return userId;
    }

    public void setUserId(int userId) {
        this.userId = userId;
    }

    public String getUserName() {
        return userName;
    }

    public void setUserName(String userName) {
        this.userName = userName;
    }

    public String getRole() {
        return role;
    }

    public void setRole(String role) {
        this.role = role;
    }

}
```

DcmaServiceException.java

```
package com.ctl.dcma.exception;

public class DcmaServiceException extends Exception {


    public DcmaServiceException(String message) {
        super(message);
    }
    public DcmaServiceException(String message, Throwable cause) {
        super(message, cause);
    }
    public DcmaServiceException(Throwable cause) {
        super(cause);
    }


}
```

DcmaDAOException.java

```
package com.ctl.dcma.exception;
public class DcmaDAOException extends Exception {
    public DcmaDAOException(String message) {
        super(message);
    }
    public DcmaDAOException(String message, Throwable cause) {
        super(message, cause);
    }
    public DcmaDAOException(Throwable cause) {
        super(cause);
    }
}
```

applicationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans --->
        <context:component-scan
                base-package="com.ctl.dcma.service,com.ctl.dcma.dao,com.ctl.dcma.commons" />
        <bean id="propertyConfigurer"
                class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"
                p:location="/WEB-INF/classes/properties/application.properties" />

        <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
                <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver" />
                <property name="url" value="${db.url}" />
                <property name="username" value="${db.username}" />
                <property name="password" value="${db.password}" />
        </bean>
        <bean id="sessionFactory"
                class="org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean"
                p:dataSource-ref="dataSource"
                p:configurationClass="org.hibernate.cfg.AnnotationConfiguration"
                p:packagesToScan="com.ctl.dcma.model" >

                <property name="hibernateProperties">
                        <props>
                                <prop key="hibernate.dialect">org.hibernate.dialect.Oracle9Dialect</prop>
                                <prop key="hibernate.show_sql">true</prop>
                                <prop key="hibernate.format_sql">false</prop>
                        </props>
                </property>
        </bean>
        <bean id="txManager" class="org.springframework.orm.hibernate3.HibernateTransactionManager">
                <property name="sessionFactory" ref="sessionFactory" />
        </bean>
</beans>
```

I18N:

Step 1:create messages.properties and messages_en.properties files and put the form lables as key ,value pairs

Example: user.list=List of User Details
            form.label.user.id=Enter User Id

Step 2: Configure the ReloadableResourceBundleMessageSource in <servlet-name>-servlet.xml file.

```
 <bean id="messageSource" class="org.springframework.context.support.ReloadableResourceBundleMessageSource">
     <property name="basenames">
        <list>
           <value>classpath:properties/messages</value>
           <value>classpath:properties/ValidationMessages</value>
        </list>
     </property>
     <property name="cacheSeconds">
        <value>10</value>
     </property>
 </bean>
```

Step 3: Use the following tag in jsp files.

```
<spring:message code="form.label.user.id" />
```

Step4: If u want localized message in controllers class use one of the following method.

```
getMessage(String key, Object[] msgArgs, String defaultMessage,Locale locale)
getMessage(String key, Object[]msgArgs, Locale locale)
getMessage(MessageSourceResolvable resolvable, Locale locale)
```

Spring Validation:

Step 1: For enable the Annatation based validation ,use the following configuration in <servlet-name>-servlet.xml

```
<bean id="configurationLoader"
    class="org.springmodules.validation.bean.conf.loader.annotation.AnnotationBeanValidation
    ConfigurationLoader"/>
  <bean id="validator" class="org.springmodules.validation.bean.BeanValidator"
    p:configurationLoader-ref="configurationLoader"/>
```

Step 2: create ValidationMessages_en.properties files and put the form lables as key ,value pairs

Example: userId.mandatory=User Id is Mandatory field.
 userId.invalid=Invaild User Id.

Step 3: Set the validation messages in Validation class.

```
public class UserValidator {
    public static void validate(UserForm userForm,Errors errors){
        if (userForm.getUserId() == null) {
            errors.rejectValue("userId","userId.mandatory", "User Id is Mandatory field");
        }
        if (userForm.getUserId() != null && userForm.getUserId() <= 0) {
            errors.rejectValue("userId","userId.invalid", "Invaild User Id.");
        }
        if (userForm.getUserName() == null || "".equalsIgnoreCase(userForm.getUserName())) {
            errors.rejectValue("userName", "userName.mandatory","User NAme is Mandatory Field");
        }
        if (userForm.getRole() == null || "".equalsIgnoreCase(userForm.getRole())) {
            errors.rejectValue("role", "role.mandatory","Role is Mandatory Field");
        }
    }
}
```

Step 4: To display the error messages in jsp use the following tag.

```
<form:errors path="userId" />
```

<u>File Upload :</u>

Step1: Configure the CommonsMultipartResolver

```
 <bean id="multipartResolver"
       class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
        <property name="maxUploadSize" value="10000000"/>
  </bean>
```

Step 2: Add the enctype=*"multipart/form-data“ in jsp .And sample code look like this.*

```
<body>
<h1>Please upload a file</h1>
<form method="post" action="saveAttachments.html" enctype="multipart/form-data">
       Upload Attachment : <input type="file" name="UploadDocumentFile" /> <br/>
       <input type="submit" value="Save"/>
</form>
</body>
```

Step3: Add the following code in controller class.

```
@RequestMapping(value = "/saveAttachments", method = RequestMethod.POST)
   public ModelAndView saveSCdocRequiredAttachment(
         @RequestParam("UploadDocumentFile") MultipartFile file, HttpServletRequest request) {

      Attachment attachment = new Attachment();
      UploadForm uploadForm = new UploadForm();
      try {
         System.out.println("File name:"+file.getOriginalFilename());
         System.out.println("File type:"+file.getContentType());
         System.out.println("File size:"+file.getSize());
         System.out.println("File size:"+file.getBytes());

         InputStream fileStreamContent = file.getInputStream();
         attachment.setDocumentName(file.getOriginalFilename());
         attachment.setFileContent(FileCopyUtils.copyToByteArray(fileStreamContent));
         attachment.setContentType(file.getContentType());
         attachment.setDocumentSize(file.getSize());

         // Invoke the service layer and save attachment.
         userService.saveDocument(attachment);

         // Get the list of attachments.
         List<Attachment> attachments = userService.getListofDocuments();
         uploadForm.setAttachmentList(attachments);
      } catch (IOException e) {
         e.printStackTrace();
      }
      return new ModelAndView("saveAttachments","uploadForm",uploadForm);
   }
```

EXTENDING SPRING MVC WITH SPRING MOBILE

Introduction

The modern web no longer is limited to desktop browsers. Smart phones and tablets have become an integral part of our daily lives. Web sites that may look good on a 22" monitor usually do not format and display well on a much smaller screen. Additionally, network speeds can limit the performance of a web site on mobile devices. Because of these reasons many developers and organizations are considering how to make their web sites accessible to all the various devices and screen sizes for which people are using.

Device detection is useful when requests by mobile devices need to be handled differently from requests made by desktop browsers. The Spring Mobile Device module provides support for server-side device detection. This support consists of a device resolution framework, site preference management, and site switcher.

Device resolution:

Device resolution is the process of introspecting a HTTP request to determine the device that originated the request. It is typically achieved by analyzing the User-Agent header and other request headers.

In Spring Mobile, the DeviceResolver interface defines the API for device resolution:

```
public interface DeviceResolver {
        Device resolveDevice(HttpServletRequest request);
}
```

The returned Device models the result of device resolution:

```
public interface Device {
    /** * True if this device is not a mobile or tablet device. */
    boolean isNormal();

    /** * True if this device is a mobile device such as an Apple iPhone or an Nexus
    One Android. * Could be used by a pre-handle interceptor to redirect the user to a
    dedicated mobile web site. * Could be used to apply a different page layout or
    stylesheet when the device is a mobile device. */

     boolean isMobile();

    /** * True if this device is a tablet device such as an Apple iPad or a Motorola
    Xoom. * Could be used by a pre-handle interceptor to redirect the user to a
    dedicated tablet web site. * Could be used to apply a different page layout or
    stylesheet when the device is a tablet device. */

    boolean isTablet();
}
```

As shown above, Device.isMobile() can be used to determine if the client is using a mobile device, such as a smart phone. Similarly, Device.isTablet() can be used to determine if the client is running on a tablet device. Depending on the DeviceResolver in use, a Device may support being downcast to access additional properties.

Reference Links :

http://msdn.microsoft.com/en-us/library/ff648617

http://static.springsource.org/spring/docs/2.0.x/reference/mvc.html

http://static.springsource.org/spring/docs/3.0.x/reference/mvc.html

Reference Books:

Spring in Action

Pro Spring