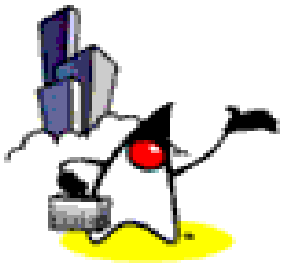
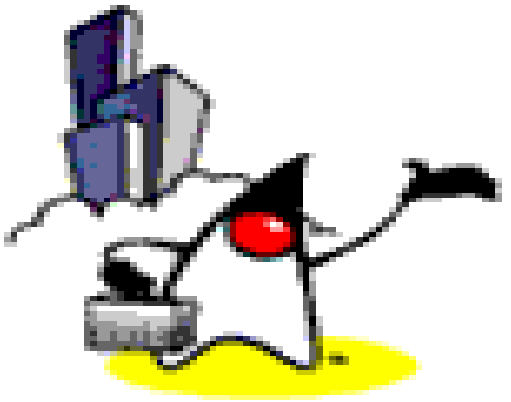
Spring MVC



Topics

- Spring MVC features
- Request life-cycle
- DispatcherServlet
- URL Handler mapping
- Controllers
- View & View Resolvers
- Validation
- Interceptors



Spring MVC Features

What is Spring MVC?

- Web application framework that takes advantage of design principles of Spring framework
  - Dependency Injection
  - Interface-driven design
  - POJO without being tied up with a framework

Features of Spring MVC

- Clear separation of roles:
  - Controller, validator,form object, model object, DispatcherServlet, handler mapping, view resolver, etc.
  - Each role can be fulfilled by a specialized class.
- Powerful and straightforward configuration of both framework and application classes as JavaBeans
  - Including easy referencing across contexts, such as from web controllers to business objects and validators.

source:/static.springframework.org/spring/docs/2.5.x/reference/mvc.html

Features of Spring MVC (Cont.)

- Adaptability, non-intrusiveness
  - Use whatever controller subclass you need (plain, command, form, wizard, multi-action, or a custom one) for a given scenario instead of deriving from a single controller for everything.
- Reusable business code
  - No need for duplication.
  - You can use existing business objects as command or form objects instead of mirroring them in order to extend a particular framework base class.

source:/static.springframework.org/spring/docs/2.5.x/reference/mvc.html

Features of Spring MVC (Cont.)

- Customizable binding and validation
  - type mismatches as application-level validation errors that keep the offending value, localized date and number binding, etc instead of String-only form objects with manual parsing and conversion to business objects.
- Customizable handler mapping and view resolution
  - handler mapping and view resolution strategies range from simple URL-based configuration, to sophisticated, purpose-built resolution strategies. This is more flexible than some web MVC frameworks which mandate a particular technique.

source:/static.springframework.org/spring/docs/2.5.x/reference/mvc.html

Features of Spring MVC (Cont.)

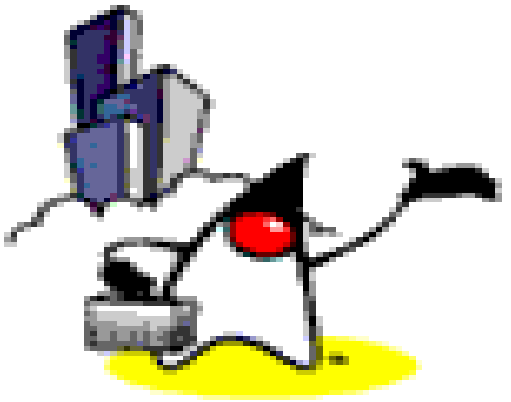
- Flexible model transfer
  - model transfer via a name/value Map supports easy integration with any view technology.
- Customizable locale and theme resolution, support for JSPs with or without Spring tag library, support for JSTL, support for Velocity
- A simple yet powerful JSP tag library known as the Spring tag library
  - provides support for features such as data binding and themes.
  - The custom tags allow for maximum flexibility in terms of markup code.

source:/static.springframework.org/spring/docs/2.5.x/reference/mvc.html

Features of Spring MVC (Cont.)

- A JSP form tag library, introduced in Spring 2.0, that makes writing forms in JSP pages much easier.
- Beans whose lifecycle is scoped to the current HTTP request or HTTP Session.
  - This is not a specific feature of Spring MVC itself, but rather of the *WebApplicationContext* container(s) that Spring MVC uses.

source:/static.springframework.org/spring/docs/2.5.x/reference/mvc.html

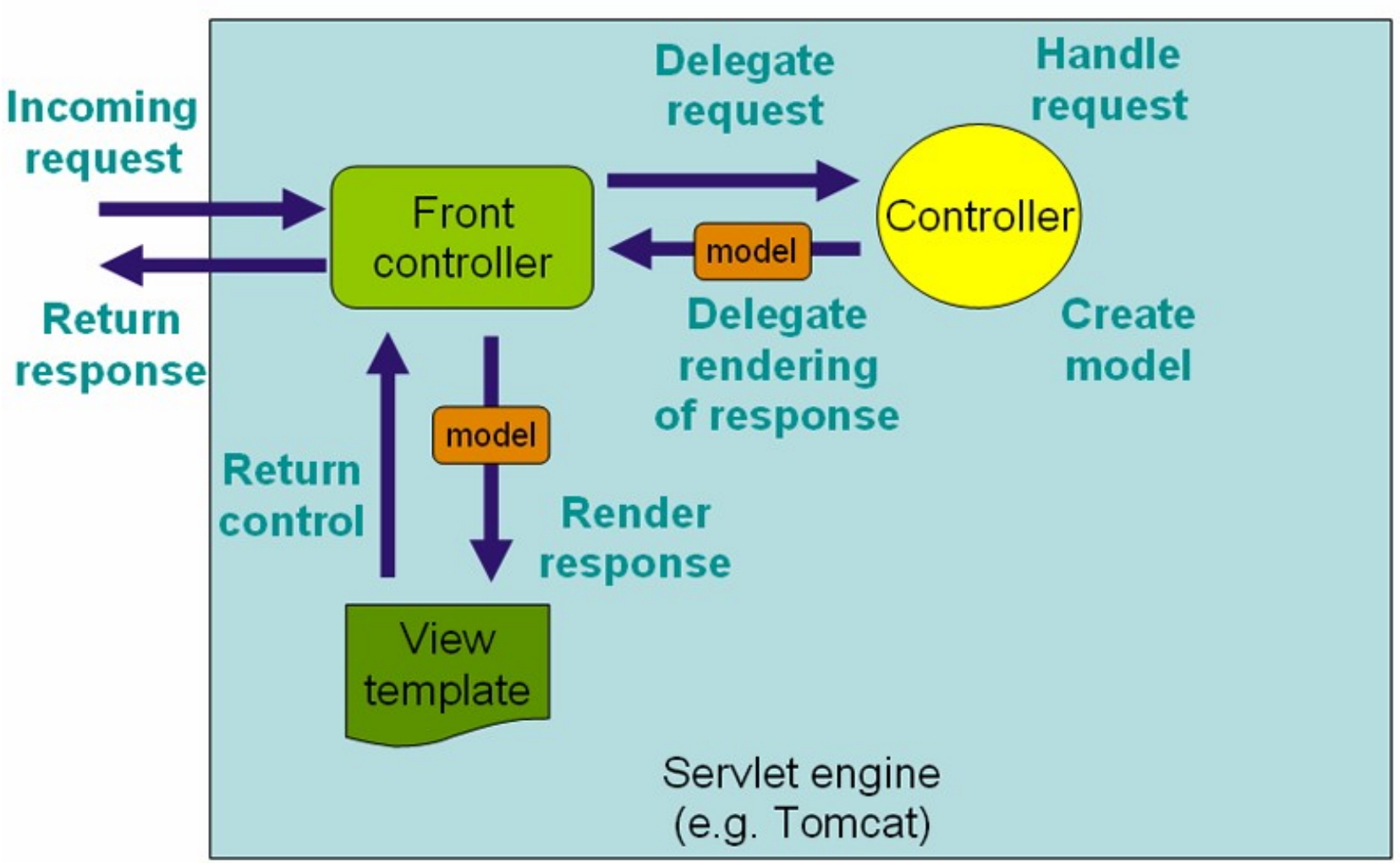


Request Life-cycle in Spring MVC

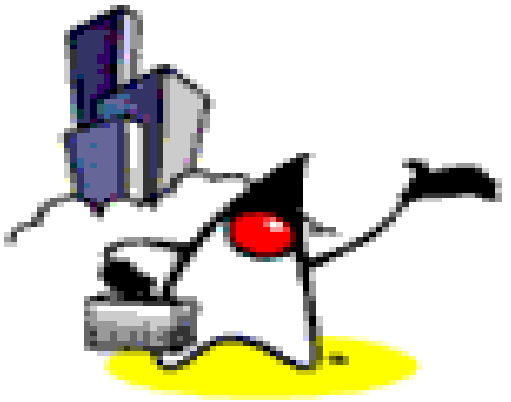
Request Life-cycle

1. *DispatchServlet* receives a HTTP request
2. *DispatchServlet* selects a Controller based on the URL Handler mapping and pass the request to the Controller
3. *Controller* performs the business logic (and set values of Model objects)
4. *Controller* returns *ModelAndView* object
5. *ViewResolver* selects a view
6. A selected view gets displayed (using values of Model objects)

The requesting processing workflow in Spring Web MVC



source:/static.springframework.org/spring/docs/2.5.x/reference/mvc.html



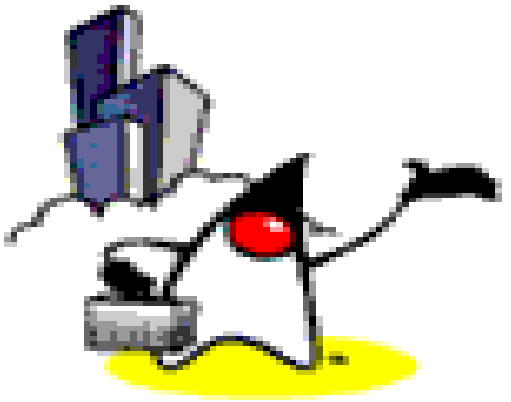
DispatcherServlet

DispatcherServlet

- The *DispatcherServlet* is the Front Controller
- Coordinates the request life-cycle
- Loads Spring application context from XML configuration file
  - */WEB-INF/[servlet-name]-servlet.xml*
- Initializes *WebApplicationContext*
  - *WebApplicationContext* is bound into *ServletContext*
- Configured in *web.xml*

Configuration of DispatcherServlet in web.xml

```
<web-app>

   <servlet>
      <servlet-name>example</servlet-name>
      <servlet-class>org.springframework.web.servlet.DispatcherServlet
      </servlet-class>
      <load-on-startup>1</load-on-startup>
   </servlet>

   <servlet-mapping>
      <servlet-name>example</servlet-name>
      <url-pattern>*.form</url-pattern>
   </servlet-mapping>

</web-app>
```



Spring MVC Interfaces

Spring MVC Interfaces

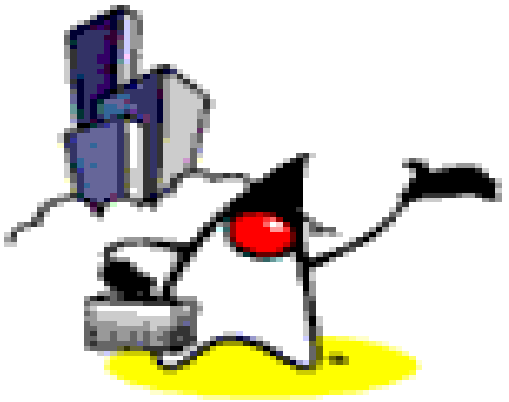
- *HandlerMapping*
  - Routing of requests to handlers (controllers)
- *HandlerAdapter*
  - *Adapts to handler interface*
  - *Default utilizes Controllers*
- *HandlerExceptionResolver*
  - Maps exceptions to error pages
  - *Similar to standard Servlet, but more flexible*
- *ViewResolver*
  - Maps symbolic name to view

Spring MVC Interfaces (Continued)

- *MultipartResolver*
  - Handling of file upload
- *LocaleResolver*
  - Default uses HTTP accept header, cookie, or session

Implementations of Spring MVC Interfaces

- Custom implementations can be created and configured
  - This is how Spring framework itself can be extended and customized
- Spring framework comes with built-in implementations (with default implementation selected) of the interfaces



URL Handler Mapping (URL to Controller Mapping)

Url Handler Mappings

- Instructs *DispatcherServlet* regarding which Controller to invoke for a request
- Implements *HandlerMapping* interface
- Spring MVC comes with two implementation classes of *HandlerMapping* interface
  - *BeanNameUrlHanlderMapping* (default)
  - *SimpleUrlHandlerMapping* (prefered)
- Implementation will be dependency-injected

BeanNameUrlHanlderMapping

- Maps URLs to beans with names or aliases that start with a slash ("/")
- Example: an incoming URL "/foo" would map to a handler named "/foo", or "/foo /foo2 /mypage"

```
<bean id=”..” name=”/foo /foo2 /mypage”
        class=”...”
</bean>
```

- Not recommended since it couples the name of the bean to the location of the controller
- Used as a default if no other HandlerMapping bean is registered in the application context.

Example: BeanNameUrlHanlderMapping (from springMVCTravel sample app)

```
<bean id="handlerMapping"
class="org.springframework.web.servlet.handler.BeanNameUrlHand
  lerMapping">

</bean>

<bean name="/home"
  class="com.apress.springMVCBooks.flight.web.HomeController">
        <property name="flightService" ref="flightService" />
</bean>
```

SimpleUrlHanlderMapping

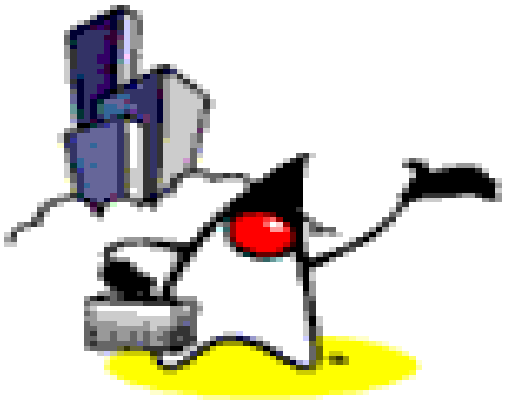
- Allows you to create complex mapping
- Mappings to bean names can be set via the "*mappings*" property
- More widely used (and preferred) than *BeanNameUrlHandlerMapping*

Example: SimpleUrlHanlderMapping (from PetClinic sample app)

```
<bean id="urlMapping"
  class="org.springframework.web.servlet.handler.SimpleU
  rlHandlerMapping">
 <property name="mappings">
   <props>
     <prop key="/welcome.htm">clinicController</prop>
     <prop key="/vets.htm">clinicController</prop>
     <prop key="/findOwners.htm">findOwnersForm</prop>
     <prop key="/owner.htm">clinicController</prop>
     <prop key="/addOwner.htm">addOwnerForm</prop>
     <prop key="/editOwner.htm">editOwnerForm</prop>
     <prop key="/addPet.htm">addPetForm</prop>
     <prop key="/editPet.htm">editPetForm</prop>
     <prop key="/addVisit.htm">addVisitForm</prop>
   </props>
 </property>
</bean>
```

Mapping Patterns (key="<mapping-pattern>")

- /myapp/*.foo (pattern)
  - /myapp/x.foo, /myapp/yz.foo (examples that match the pattern)
- /myapp/p*ttern
  - /myapp/p1ttern, /yapp/pxttern
- /**/example
  - /myapp/example, /youapp/yourdir/example
- /myapp/**/mydir/foo.*
  - /myapp/yourapp/yourdir/mydir/foo.x
- /**/*.jsp
  - /yourapp/yourdir/x.jsp, /myapp/mydir/yz.jsp



Controllers

What is a Controller?

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

Controller Class Hierarchy

- AbstractConroller
  - BaseCommandController
    - AbstractCommandController
    - AbstractFormController
      - SimpleFormController
      - AbstractWizardController
  - MultiActionController
  - ParameterizableViewController

AbstractController

- Convenient superclass for controller implementations
- Workflow
  - *handleRequest*() of the controller will be called by the *DispatcherServlet*
  - *handleRequest*() method calls abstract method *handleRequestInternal*(), which should be implemented by extending Controllers to provide actual functionality to return *ModelAndView* objects. In other words, the extending Controller is responsible for handling the actual request and returning an appropriate *ModelAndView*

BaseCommandController

- Controller implementation which
  - Creates an object (the command object) on receipt of a request
  - Populates command object with request parameters

BaseCommandController

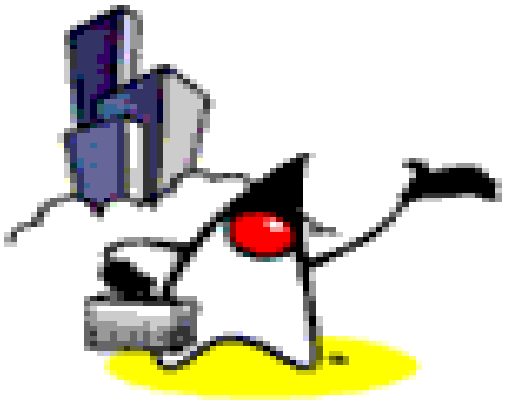
- This controller is the base for all controllers wishing
  - to populate JavaBeans based on request parameters
  - to validate the content of such JavaBeans using Validators
  - to use custom editors (in the form of *PropertyEditors*) to transform strings into objects of some type and vice versa, for example

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*, and custom form controllers you can provide yourself

AbstractFormController

- A form-input view and an after-submission view have to be provided programmatically.
- Subclasses need to override
  - *showForm* to prepare the form view, and *processFormSubmission* to handle submit requests.
- To provide those views using configuration properties, use the *SimpleFormController*.
  - The *SimpleFormController* provides implementation of the *showForm* and processFormSubmission methods



SimpleFormController

SimpleFormController

- Concrete FormController implementation that provides configurable form and success views
- Automatically resubmits to the form view in case of validation errors, and renders the success view in case of a valid submission.
- The *onSubmit()* method is overidden by the application

SimpleFormController

- Handles single page form input
- Split into two workflows
  - Form view
    - 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

SimpleFormController: Form View

- Controller receives a request for a new form (typically GET)
- *formBackingObject*()
  - to load or create an object edited by the form
- *initBinder*()
  - to register custom editors for fields in the command object
- *showForm*()
  - to return a view to be rendered
- *referenceData*()
  - to add data needed by the form (select list) to the model

SimpleFormController: Form Submission

- Controller receives a form submission (typically a POST)
- *formBackingObject*()
  - to load or create an object edited by the form
- Request data is bound to the form backing object
- *onBind*()
  - to perform custom processing after binding but before validation
- Validator is invoked
- onBindAndValidate()
  - to do custom processing after binding and validation
- onSubmit()
  - to do custom submission processing

SimpleFormController Class (from springMVCSimpleFormController app)

```
public class LoginBankController extends SimpleFormController {

 protected ModelAndView onSubmit(Object command) throws Exception{

   LoginCommand loginCommand = (LoginCommand) command;

   authenticationService.authenticate(loginCommand);
   AccountDetail accountdetail =
      accountServices.getAccountSummary(loginCommand.getUserId());
      return new
        ModelAndView(getSuccessView(),"accountdetail",accountdetail);
 }
```

Form view

- Indicates what view to use when the user asks for a new form or when validation errors have occurred on form submission
- Configurable

```
<property name="formView">
    <value>login</value>
</property>
```

Success view

- Indicates what view to use when successful form submissions have occurred.
  - Could display a submission summary.
  - More sophisticated actions can be implemented by overriding one of the *onSubmit()* methods.
- Configurable

```
<property name="successView">
    <value>accountdetail</value>
</property>
```

SimpleFormController Configuration

```
<bean id="loginBankController"
class="springexample.contoller.LoginBankController">
  <property name="sessionForm">
    <value>true</value>
  </property>
  <property name="commandName">
    <value>loginCommand</value>
  </property>
  <property name="commandClass">
    <value>springexample.commands.LoginCommand</value>
  </property>
```

SimpleFormController Configuration

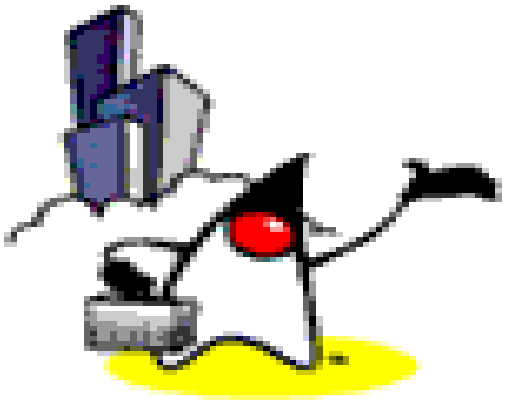
```
<property name="authenticationService">
    <ref bean="authenticationService" />
</property>
<property name="accountServices">
    <ref bean="accountServices" />
</property>
<property name="formView">
    <value>login</value>
</property>
<property name="successView">
    <value>accountdetail</value>
</property>

</bean>
```

SimpleFormController Configuration

- *commandClass*—the class of the object that will be used to represent the data in this form.
- *commandName*—the name of the command object.
- *sessionForm*—if set to false, Spring uses a new bean instance (i.e. command object) per request, otherwise it will use the same bean instance for the duration of the session.

SimpleFormController Configuration

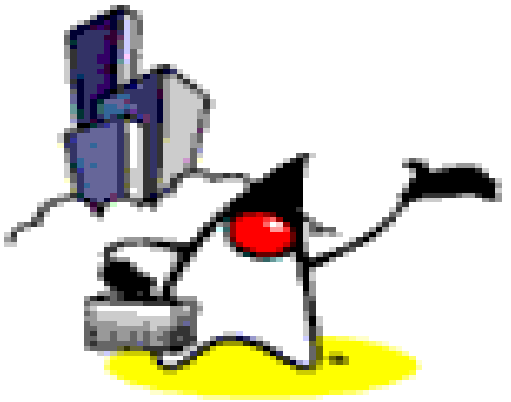
- *validator*—a class that implements Spring's Validator interface, used to validate data that is passed in from the form.
- *formView*—the JSP for the form, the user is sent here when the controller initially loads the form and when the form has been submitted with invalid data.
- *successView*—the JSP that the user is routed to if the form submits with no validation errors.



AbstractWizardFormController

AbstractWizardFormController

- Abstract class - you have to implement
  - validatePage()
  - processFinish()
  - processCancel()
- You probably also want to write a contractor, which should at the very least call setPages() and setCommandName(). The former takes as its argument an array of type String. This array is the list of views which comprise your wizard.



MultiActionController & MethodNameResolver

MultiActionController

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

MethodNameResolver Implementations

- *InternalPathMethodNameResolver*
- *ParameterMethodNameResolver*
- *PropertiesMethodNameResolver*

InternalPathMethodNameResolver

- Simple implementation of *MethodNameResolver* that maps URL to method name.
- Although this is the default implementation used by the *MultiActionController* class (because it requires no configuration), it's bit naive for most applications.
  - In particular, we don't usually want to tie URL to implementation methods.
- Maps the resource name after the last slash, ignoring an extension.
  - E.g. "/foo/bar/baz.html" to "baz"

ParameterMethodNameResolver

- Interprets a request parameter as the name of the method that is to be invoked.
- For example, *'http://www.sf.net/index.view?method=testIt'* will result in the method *testIt(HttpServletRequest, HttpServletResponse)* being invoked.
- The '*paramName*' property specifies the name of the request parameter that is to be used.

ParameterMethodNameResolver

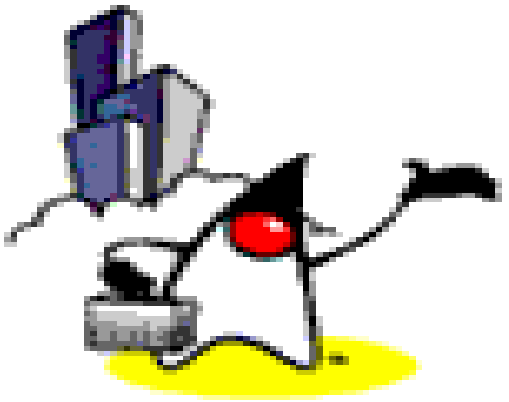
```
<bean id="paramMultiController"

   class="org.springframework.web.servlet.mvc.multiaction.MultiActionContr
   oller">

     <property name="methodNameResolver">
         <bean
   class="org.springframework.web.servlet.mvc.multiaction.ParameterMethodN
   ameResolver">
             <property name="paramName" value="method"/>
         </bean>
     </property>

     <property name="delegate">
         <bean class="samples.SampleDelegate"/>
     </property>

</bean>
}
```

PropertiesMethodNameResolver

- Uses a user-defined Properties object with request URLs mapped to method names.
- For example, when the Properties contain '/index/welcome.html=doIt' and a request to /index/welcome.html comes in, the doIt(HttpServletRequest, HttpServletResponse) method will be invoked.
- This particular MethodNameResolver uses the Spring PathMatcher class internally, so if the Properties contained '/**/welcom?.html', the example would also have worked.

PropertiesMethodNameResolver

```
<!-- This bean is a MultiActionController that manages general View
   rendering. It uses the "clinicControllerResolver" bean below for method
   name resolution.-->
<bean id="clinicController"
   class="org.springframework.samples.petclinic.web.ClinicController">
 <property name="methodNameResolver" ref="clinicControllerResolver"/>
 <property name="clinic" ref="clinic"/>
</bean>

<!-- This bean is a MethodNameResolver definition for a
   MultiActionController. It maps URLs to methods for the
   "clinicController" bean.-->
<bean id="clinicControllerResolver"
   class="org.springframework.web.servlet.mvc.multiaction.PropertiesMethod
   NameResolver">
 <property name="mappings">
   <props>
      <prop key="/welcome.htm">welcomeHandler</prop>
      <prop key="/vets.htm">vetsHandler</prop>
      <prop key="/owner.htm">ownerHandler</prop>
   </props>
 </property>
</bean>
```



View & View Resolvers

View

- Renders the output of the request to the client
- Implements the *View* interface
- Built-in support for
  - JSP, XSLT, Velocity, FreeMarker
  - 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

- Direct resolution of symbolic view names to URLs, without explicit mapping definition
- Example:
  - prefix="/WEB-INF/jsp/", suffix=".jsp"
  - viewname="test" -> "/WEB-INF/jsp/test.jsp"

InternalResourceViewResolver

- Convenient subclass of *UrlBasedViewResolver* that supports *InternalResourceView* (i.e. Servlets and JSPs) and subclasses such as *JstlView* and *TilesView*.
- The view class for all views generated by this resolver can be specified via *UrlBasedViewResolver.setViewClass(java.lang.Class)*

InternalResourceViewResolver

- “myview” -> /WEB-INF/views/myview.jsp

```
<bean id="viewResolver"
class="org.springframework.web.servlet.view.InternalResourceV
  iewResolver">

   <property name="prefix">
      <value>/WEB-INF/views/</value>
   </property>
   <property name="suffix">
      <value>.jsp</value>
   </property>
</bean>
```

ResourceBundleViewResolver

- The View definitions are kept in a separate configuration file
  - You do not have to configure view beans in the application context file
- Supports internationalization (I18N)

ResourceBundleViewResolver

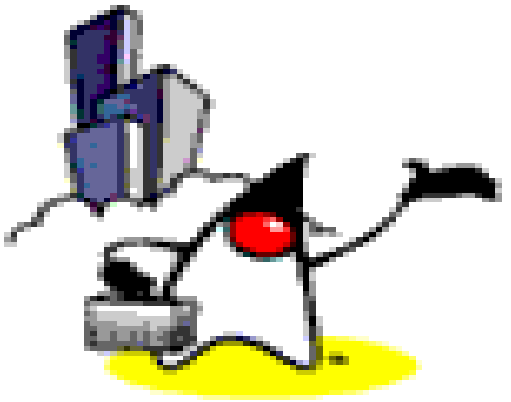
```
<!-- This bean provides explicit View mappings in a resource bundle
   instead of the default InternalResourceViewResolver. It fetches
   the view mappings from localized "views_xx" classpath files, i.e.
   "/WEB-INF/classes/views.properties" or "/WEB-
   INF/classes/views_de.properties". Symbolic view names returned by
   Controllers will be resolved by this bean using the respective
   properties file, which defines arbitrary mappings between view
   names and resources. -->
<bean id="viewResolver"
   class="org.springframework.web.servlet.view.ResourceBundleViewRes
   olver">
  <property name="basename" value="views"/>
</bean>
```

Example: views.properties

```
# This is from petclinic sample application

welcomeView.(class)=org.springframework.web.servlet.view.JstlView
welcomeView.url=/WEB-INF/jsp/welcome.jsp

vetsView.(class)=org.springframework.web.servlet.view.JstlView
vetsView.url=/WEB-INF/jsp/vets.jsp

# A lot more are defined
```

Example: Returning a View

```
public class ClinicController extends MultiActionController
   implements InitializingBean {

  public ModelAndView welcomeHandler(HttpServletRequest request,
   HttpServletResponse response) throws ServletException {
      return new ModelAndView("welcomeView");
   }

  public ModelAndView vetsHandler(HttpServletRequest request,
   HttpServletResponse response) throws ServletException {
      return new ModelAndView("vetsView", "vets",
                              this.clinic.getVets());
   }
```



Spring Tags

<spring:bind path=”..”> tag

- Used to connect one of the form’s input fields with the corresponding field in the command object
  - `<spring:bind path="loginCommand.userId">`
- Within this tag, the *status* object is available.
- *path=”..”* attribute
  - The path to the bean or bean property to bind status information for.
  - The *status* object will exported to the page scope, specifically for this bean or bean property
  - For instance *account.name*, *company.address.zipCode* or just *employee*.

${status.value} with <spring:bind ..>

- Is used to get the current value of the field.
- Either have been set up in the controller bean servlet (in the *formBackingObject()* method), or will be the previously submitted value if the form has already been filled in, submitted, and rejected by the validator bean servlet.

${status.value} with <spring:bind ..> (login.jsp of springMVCSimpleFormController)

```
## bind on the userId field of the loginCommand
<spring:bind path="loginCommand.userId">
    <td width="20%">
        <input type="text"
               name='<c:out value="${status.expression}"/>'
               value='<c:out value="${status.value}"/>'>
    </td>
</spring:bind>
```

${status.errorMessages} with <spring:bind ..>

- *${status.errorMessages}* contains a list of error messages linked to the field, localised if necessary.
- *${status.errorMessage}* contains the first error message, and can safely be used if you are expecting a maximum of one error per form field.

${status.errorMessages} with <spring:bind ..> (login.jsp of springMVCSimpleFormController)

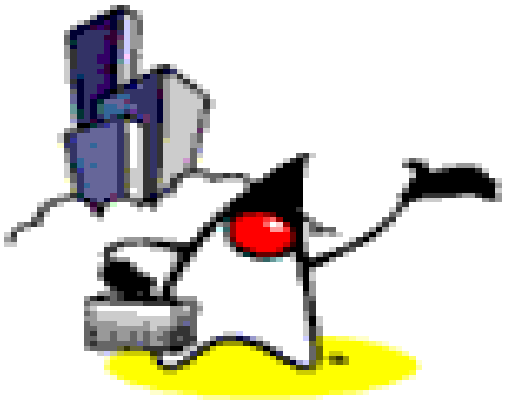
```
<form method="post" action="">
    ## first bind on the object itself to display global
    ## errors - if available
    <spring:bind path="loginCommand">
        <c:forEach items="${status.errorMessages}"
                   var="errorMessage">
            <font color="red"> <c:out value="${errorMessage}" /> <br>
            </font>
        </c:forEach>
    </spring:bind>
```

<spring:hasBindErrors name="..." > tag

- Provides you with support for binding the errors for an object.
- If they are available, an *Errors* object gets bound in the page scope, which you can inspect.
- Basically it's a bit like the <*spring:bind*> tag, but this tag does not feature the status object, it just binds all the errors for the object and its properties.

web.xml Configuration for Spring tags (login.jsp of springMVCSimpleFormController)

```
<web-app>

    ...
    <taglib>
        <taglib-uri>/spring</taglib-uri>
        <taglib-location>/WEB-INF/spring.tld</taglib-location>
    </taglib>
</web-app>
```



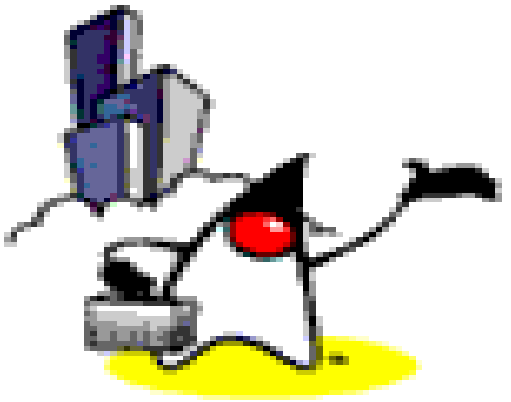
Validation

Validation Configuration (tradingapp-servlet.xml from springMVCStockTrading)

```
<!--Validator for logon forms, implementing Spring's Validator interface. -->
 <bean id="logonValidator"
       class="com.devx.tradingapp.web.LogonValidator"/>
```

Validation Class (LoginValidator.java from springMVCStockTrading)

```
public class LogonValidator implements Validator {

    private final Log logger = LogFactory.getLog(getClass());

    public boolean supports(Class clazz) {
        return clazz.equals(Credentials.class);
    }

    public void validate(Object obj, Errors errors) {
        Credentials credentials = (Credentials) obj;
        if (credentials == null) {
            errors.rejectValue("username", "error.login.not-
  specified", null,
                    "Value required.");
        } else {
        ....
```



Interceptors

Interceptors

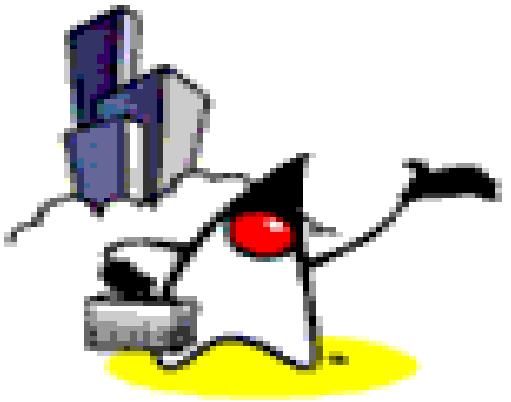
- You can specify a list of interceptors that are called for each mapping
- An interceptor can process each request before or after the appropriate controller is called
- Implements *HandlerInterceptor* interface or extend *HandlerInterceptorAdaptor*

Example: Interceptor Configuration

```
<bean id="urlMapping"
      class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="interceptors">
      <list>
        <ref bean="officeHoursInterceptor"/>
      </list>
    </property>
    <property name="mappings">
      <props>
        <prop key="/actionNa*">dispatchController</prop>
        <prop key="/o*">dispatchController</prop>
      </props>
    </property>
</bean>

<bean id="officeHoursInterceptor"
      class="samples.TimeBasedAccessInterceptor">
    <property name="openingTime"><value>17</value></property>
    <property name="closingTime"><value>18</value></property>
</bean>
```

Example: Interceptor

```
public class TimeBasedAccessInterceptor extends HandlerInterceptorAdapter {

    ...
    public boolean preHandle(
            HttpServletRequest request,
            HttpServletResponse response,
            Object handler)
            throws Exception {

        Calendar cal = Calendar.getInstance();
        int hour = cal.get(Calendar.HOUR_OF_DAY);
        if ((openingTime <= hour) && (hour < closingTime)) {
            return true;
        } else {
            response.sendRedirect("outsideOfficeHours.jsp");
            return false;
        }
    }

    public void postHandler(..) {}
}
```



Themes

What is theme?

- The theme support provided by the Spring web MVC framework enables you to further enhance the user experience by allowing the look and feel of your application to be themed.
- A theme is basically a collection of static resources affecting the visual style of the application, typically style sheets and images.

Spring MVC