Spring Layout

Reference Guide

Sponsored By:

groupware Switch on.

Document Authors: Matthew Tomlinson, Rob Monie

Document Version: 0.5

Spring Layout Version: 0.7.3

Table of Contents

Introduction....4
- Overview....4
- Architecture....4
- Getting Started....5
  - Configuring Layout Resources....5
    - Using the LayoutResources servlet....5
    - Managing Layout Resources Manually....6
    - Referencing Layout Resources in html files....6

Spring Layout Controllers....7
  - Single Page Forms 7
    - AbstractBaseFormController....8
  - Method delegation....8
  - Convenience Javascript Method for method delegation....8
  - Default Property Editor Registration....8
  - Localised Messages....9
    - Saving Messages for Display....9
    - Messages and Redirects....9
  - Page Modes....10
  - Page Access Resolvers....10
  - Specifying Read Only Fields....11
  - Controller Configuration Reference....11
  - Multi Page Forms....12
    - MultiPanelFormController....12
    - Multi-page Form State Management....12
    - Form Page Navigation....13
    - Validation....13
      - Validation Mode....14
    - Displaying all Form Pages at Once....15
  - Controller Configuration Reference....15
  - Controller Lifecycle....16

Security....20
- Page Access Resolvers....20
  - Implementing a PageAccessResolver....21
  - Configuration....21
  - Page Level Security....22
  - Field Level Security....23
  - Function Permissions....24
  - PageAccessHolder....24
- JSP Tags....25

Form and Field Layout Configuration....26
- LayoutForm Attributes....27
- LayoutField Attributes....27
- Default Field Attribute....28
- Panel Forms / Form Groups....28
  - LayoutPanelForm Attributes....29
- Validation Configuration....29
  - Simple Mandatory Field Validation....29
  - Conditional Mandatory Field Validation....30
    - Validating Against Multiple Mandatory Conditions....30
    - Cross Panel Validation Dependencies 31

Raising Errors on Alternate Fields....31
PathBasedCondition attributes....32
Validation Rules....32
Writing Custom Validation Rules....33
JSP Tags....34
Form and Field Tags....34
Field Tags....34
Form Tags 36
Panel Tab & Footer Renderers....36
Security Tags....37
Other Tags....37
User Interface Behaviour....38
Overriding the Default Look and Feel....39
Javascript Reference....40

Introduction

Overview

Spring Layout is an Open Source Spring MVC extension that simplifies the development of data-centric web applications. It is designed to service a number of common requirements of web applications enabling the rapid development of form based web applications. This is achieved through the following set of features:

- JSP Tag framework with integrated declarative client & server-side validation
- Enables the use of a single JSP for edit / read modes
- Pluggable security with page and form field level granularity
- Rich multi-page / wizard form framework
- Editable Data Grid

Architecture

Spring Layout consists of a number of key components including:

- Controller Extensions – Extend the core Spring MVC Controllers providing specific Spring Layout functionality.
- Security Framework – Optionally used by most components within the framework for authorisation at the page and field level. The security framework deals specifically with authorization concerns and should be used in conjunction with a third party authentication framework. Acegi Security is a very popular and highly recommended choice.
- JSP Tag Library – Provides rich form tags with built in security and validation capabilities.
- Form & Field Definitions – Configuration of layout for forms and fields for referencing by JSPs.
- Validation Framework – Provides rich declarative validation rules via field definitions. Easily Extended for specific business validation requirements.
- Layout Resources – Javascript, CSS and images allowing rich front end behaviours and formatting based around validation, and user interaction.

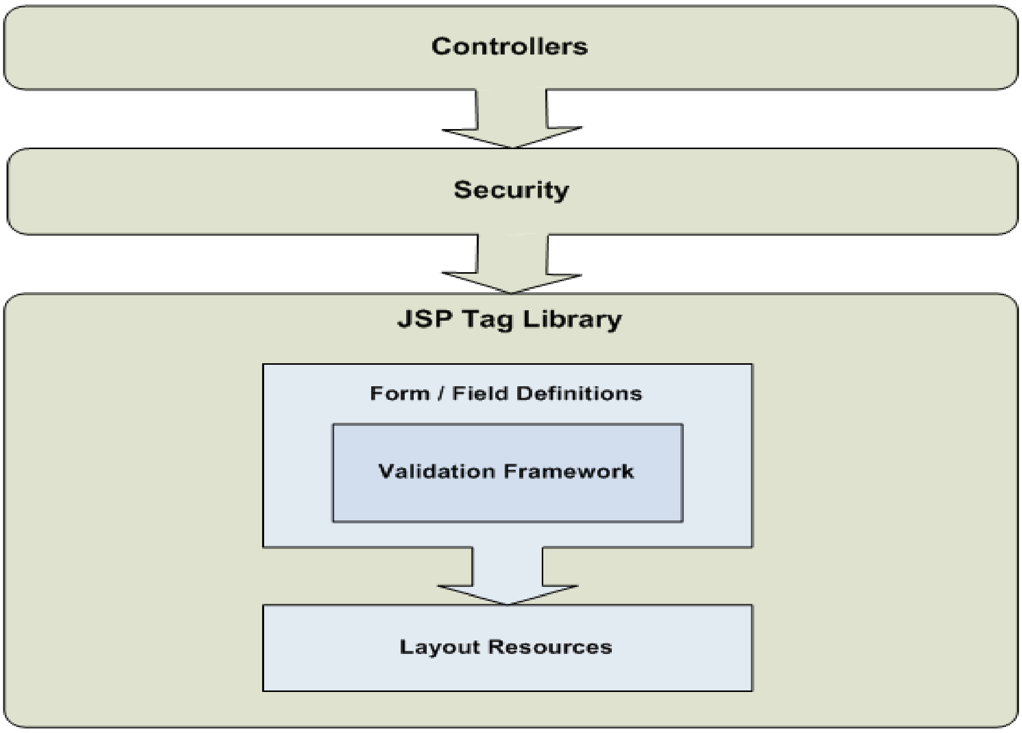


Illustration 1: Overall Spring Layout Architecture

Getting Started

To get started using Spring Layout there are some general configuration requirements and options.

Configuring Layout Resources

Spring Layout comes with a comprehensive Javascript library and style sheets that must be imported into JSP pages. This can be done in either of two ways:

Using the LayoutResources servlet

The Spring Layout jar contains all layout resources and dependencies including Javascript, CSS and images. The simplest way to configure Spring Layout is to map the Layout Resources servlet in web.xml. This method ensures an easy upgrade and simple configuration. You will always have layout resources compatible with the jar file in your project.

To configure the Layout Resources servlet, add the following to your web.xml file.

```
<servlet>
  <servlet-name>layoutResourceServlet</servlet-name>
  <servlet-class>net.sf.springlayout.web.layout.servlet.LayoutResourcesServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>layoutResourceServlet</servlet-name>
  <url-pattern>/layoutResources/*</url-pattern>
```

```
</servlet-mapping>
```

Using this configuration, the Layout Resources servlet will serve up all JS, CSS and image files from within the Spring Layout jar.

While this method is the simplest, it may be desirable to manage these files manually within your project. One reason for doing this might be to manage the javascript libraries that Spring Layout depends on such as prototype and behaviour.js. Obviously there might be compatibility issues with using different versions of these libraries so we don't recommend it.

It should be noted that it is possible to extend and override the CSS supplied by Spring Layout. This is detailed in the “Overriding the default Look and Feel” chapter.

Managing Layout Resources Manually

To enable the manual management of the Layout Resources files firstly extract the layoutResources folder from the Spring Layout jar using Winrar or another archive management tool.

This folder should be placed under your projects “WebRoot” folder or equivalent depending on your project structure. Ultimately it should be placed in such a location that it can be referenced via the path */[app root]/layoutResources/.*

Remember that now whenever you upgrade Spring Layout, you must repeat this process. If you modified or replaced any files here you must now manage this in upgrades.

Referencing Layout Resources in JSPs or HTML files

To simplify referencing all the Spring Layout dependencies, a single JSP tag can be used in the html <head> element.

Use of this tag in it's most simple form is shown in the following example:

```
<head>
     <layout:config />
     ...
</head>
```

The result of this is to output all the relevant js and css imports and setup some Javascript variables.

Any Javascript and CSS specific to the page should be placed after this tag. This is especially important for any CSS that overrides Spring Layout defaults as CSS overrides existing style declarations as it renders.

Displaying Loading Message

The layout:config tag provides an attribute called “showLoadingMessage” that can be used to turn on or off a loading message displayed to users when a page is being refreshed or submitted. While providing feedback to the user that something is happening, it has the added benefit of preventing any clicking on the page. This is a great way to prevent double posting of forms.

Further details of the <layout:config /> tag can be found in the Tag Reference.

Spring Layout Controllers

Being a web application framework the Spring Layout controllers are the centre of the Spring layout extensions and provide the underlying lifecycle for most aspects of the framework including:

- data binding
- validation
- security
- page flow

The Spring Layout controllers extend the base Spring form controller hierarchy to support two page or form types.

- Single page forms
- Multi-page forms for a tabbed or wizard style form

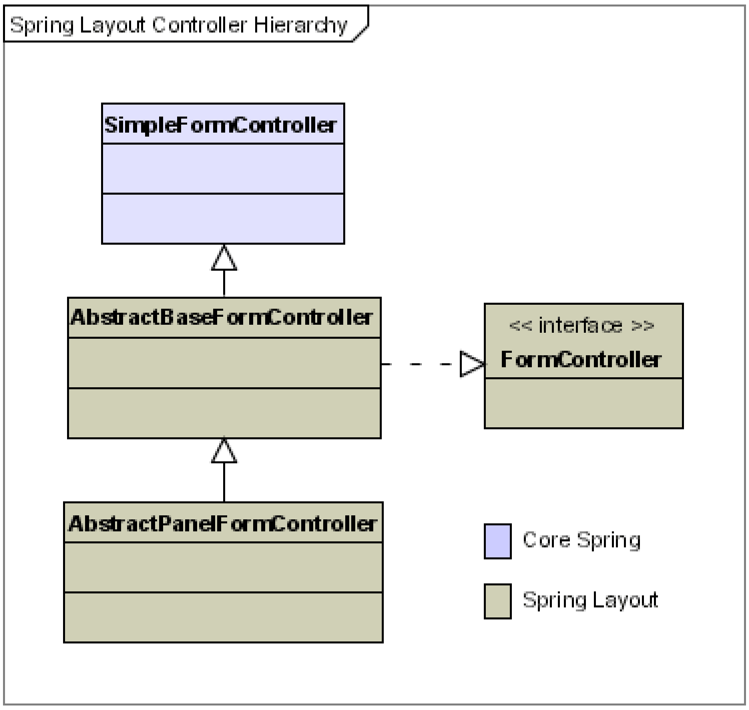

Spring Layout Controller Hierarchy
SimpleFormController
AbstractBaseFormController
<< interface >>
FormController
AbstractPanelFormController
Core Spring
Spring Layout

Figure 1: Controller Hierarchy

The intended use of these controllers in an application follows what would generally be considered to be best practices for controllers in a Spring application so previous experience in using Spring MVC would be advantageous to understanding Spring Layout.

Single Page Forms

Data centric web applications most commonly use a single form per transaction model where a html form is presented showing existing data for reading or editing or providing the ability to add new data. Commonly this data would be submitted to a servlet or in Spring's case a controller, validated against business rules before being bound to a model object graph and saved to a database. Springs core Dispatcher Servlet and Controller framework handles this elegantly and provides an excellent base for extension.

AbstractBaseFormController

At the top of the Spring Layout Controller hierarchy is AbstractBaseFormController. This controller provides all the functionality for a single page controller and the common base functionality for multi page controllers.

Method delegation

Method delegation is a concept used commonly in MVC style frameworks where a controller method can be identified through a request parameter and called automatically using Java reflection. This approach allows multiple functions for a page such as *save*, *delete* etc to be rolled into a single controller reducing the number of controllers required and consequently reducing configuration. Spring layout uses the request parameter “method” to specify the name of the controller method to call. A controller can contain either one or two methods with the name specified in the “method” parameter with either of two method signatures depending on what point of the controller lifecycle they should be called.

The first method signature that is potentially called occurs pre-binding enabling data manipulation to be achieved that is not practical to do using regular property editors.

```
public void methodName(HttpServletRequest request,
                                   Object command,
                                   BindException errors) throws Exception
```

The second method signature is called after binding and validation has taken place by an overridden SimpleFormController.onSubmit method

```
public ModelAndView methodName(HttpServletRequest request,
                                          HttpServletResponse response,
                                          Object command,
                                                 BindException errors) throws Exception
```

Convenience Javascript Method for method delegation

As a convenience a Javascript method exists for transparently adding the 'method' parameter and it's value and submitting the current form. This exists in layout.js and contains the following method signature:

```
function performAction(formName, action, unpositioned)
```

Editing the Command Object before binding

At times some manipulation of the command object may be required prior to binding and invoking property editors. On these occasions this can be achieved by overriding the following controller method.

```
protected void editCommandBeforeBind(HttpServletRequest request, Object command)
```

Default Property Editor Registration

A default behaviour of Spring is to register default property editors for numeric values to raise an error on binding when the value is null. While we have never found this to be a desirable default behaviour, rather than changing this without warning, Spring Layout allows a controller level switching of this via the controller property *defaultPropertyEditorsAllowEmpty.*

```
<property name="defaultPropertyEditorsAllowEmpty" value="true" />
```

Using this configuration in each controller or an abstract parent controller will register all numeric property editors so that they do not raise a binding error when they encounter a null value.

Localised Messages

Simplified localised message retrieval is available from within the controller via the provided *getMessage* methods. Properties files for messages are setup through the regular MessageSource configuration similar to the following configuration:

```
<bean id="messageSource"
class="org.springframework.context.support.ResourceBundleMessageSource">
     <property name="basenames">
            <list>
                   <value>messages</value>
                   <value>labels</value>
            </list>
     </property>
     <property name="useCodeAsDefaultMessage" value="true" />
</bean>
```

Saving Messages for Display

A basic convenience method is supplied for saving messages into the request under known request attribute (WebConstants.*MESSAGES)*

```
protected void saveMessages(Messages messages, HttpServletRequest request)
```

A simple Messages class exists that is useful for grouping a list of messages with a title. This resides at *net.sf.springlayout.web.message.Messages* and may be useful in some situations rather than using a List object.

Messages and Redirects

Due to the well known issues around back buttons / page refreshes and using server side forwards, we strongly encourage the use of redirects following the successful execution of a POST. The implication of this is that anything added to the request will be lost following the redirect. The solution to this provided by Spring Layout is to configure a Messages Interceptor. The MessagesInterceptor is responsible for temporarily storing any request attribute stored in the request into the session and then moving them back into the request on the GET following the redirect.

The following configuration is an example of how the publicUrlMapping bean can be used with a messages interceptor to intercept all controller invocations:

```
<bean id="messagesInterceptor" class="net.sf.springlayout.web.interceptor.MessagesInterceptor"
/>

<bean id="publicUrlMapping"
class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
     <property name="interceptors">
                    <list>
                           <ref local="messagesInterceptor" />
                    </list>
     </property>
     ...
</bean>
```

For single page forms it is still practical to use a forward after validation failure or any other error that requires the current form to be returned to the user with error messages. For this reason there is no errors equivalent to the MessagesInterceptor. Errors should be handled in a typical Spring fashion using the errors object and it's methods.

Page Modes

One benefit of using Spring Layout is the ability to open a page in read or edit mode. How a page is opened can be set in two ways. Firstly a default setting for a controller can be specified in the controller configuration using the property *defaultEditMode*. Setting this to *true* will open the page in edit mode. Setting it to false (the default) will open a page in read mode.

Alternatively, or in combination with this configuration, the url can be used to open a page in read or edit mode. Appending the parameter *e=1*, will open the page in edit mode whereas *e=0,* will open the page in read mode. Specifying the mode in the url will always override the default.

Used in combination with Page Access Resolvers which is touched on in the next section and detailed in the security chapter, page modes can be requested and either granted or downgraded to the user's maximum access level.

For example, consider a user registration system users are only allowed to edit their own profile but can view other's profiles. If a user appends *e=1* to the url of another user's profile, they will only be granted read access.

Page Access Resolvers

Spring layout provides a flexible security authorization layer which is primarily based around but not limited to the Controller Command Object. This layer of security can be combined with any authentication strategy such as Acegi Security and comes in the form of PageAccessResolvers. Page Access Resolvers are optionally injected into Spring Layout Controllers and enable record and field level security to be applied to a controller and it's associated page. The specifics of implementing Page Access Resolvers are covered in detail in the Security Chapter.

Once a Page Access Resolver has been written it can be injected into a controller in the following manner:

```
<bean id="userPageAccessResolver"
class="net.sf.springlayout.sampleapp.web.security.UserPageAccessResolver" />

<bean id="userController" parent="abstractItrackController"
            class="net.sf.springlayout.sampleapp.web.controller.UserController" >

     <property name="pageAccessResolver">
            <ref bean="userPageAccessResolver" />
     </property>

     ...

</bean>
```

The Page Access Resolver will be called at the appropriate time in the controller lifecycle and it's effects on the security of the current request applied.

Specifying Read Only Fields in the Controlle

It would usually be desirable to programatically configure specific fields as read only using a Page Access Resolver in order to keep all security concerns in a central location. It is possible however for simple cases where a Page Access Resolver might not be required, to configure read only fields in the controller itself. This can be achieved by overriding the following controller method and returning a List of strings representing the field path to be treated as read only:

```
public List defineReadOnlyFields(HttpServletRequest request)
```

Controller Configuration Reference

The following table details all Spring Layout settable properties for controllers that extend AbstractBaseFormController:

Attribute	**Description**
form	Reference to the Form Spring bean used to configure rules relating to validation and layout for fields on the page.
defaultEditMode	true / false. Determines whether the page should open in edit or

	read mode by default if not specified in the url
defaultProperty-EditorsAllowEmpty	True / false. If set to true will register all default property editors to allow empty values to be seen as null and valid. Default Spring behaviour will see a null value for an integer as a binding error.
pageAccessResolver	Reference to an optional PageAccessResolver used for determining a user's access to the current page, its fields and functions within it.
redirectFormView	The name of the view (defined in views.properties or equivalent) that redirects to the current controller. This is required so that a redirect can be used after all form submission / binding.
formView	The name of the view used to render the current page. Used by showForm as in core SpringMVC.
commandName	Name by which to store and reference the command object.

Key: Shaded cells indicate properties important to Spring Layout but part of the core Spring MVC framework

Multi Page Forms

Applications that are required to gather large sets of data in a single transaction or use a wizard styled approach for gathering data may require forms that span multiple pages. This method of data gathering and editing presents a set of problems that are not a consideration for single page forms. Spring Layout provides a framework to solve many of the complexities of multi-page or wizard style forms development.

Spring Layout allows for the use of different controllers for different pages within the multi-page form. This allows for far greater flexibility than using a single controller but also introduces it's own set of problems that need to be handled.

MultiPanelFormController

The MultiPanelFormController should be used for any controller that is intended to be part of a multi-page form group. The MultiPanelFormController controller extends AbstractBaseFormController so all of the common functionality is inherited while a number of features specific to a multi-page form are added. Individual controllers that work together in a multi-page environment do not need to be made explicitly aware of each other at the controller level but are tied together by configuration in the PanelForms and PanelFormGroup to which they belong. This configuration is detailed in the LayoutConfiguration Section .

Multi-page Form State Management

As with a single page form, multi page form requires a single command object to hold the object graph for editing. This object graph may be as complex as your business logic requires but ultimately a single object graph is viewed / edited and potentially updated. To store the changes a user makes to the object graph as they edit and navigate between pages, binding of form data against the command object needs to take place but not be persisted to the database until a user explicitly saves the form. Spring Layout does not handle this directly but puts the responsibility in the hands of the developer. We can however offer our experience and suggest a strategy that may work for you and also warn of any complexities that may be encountered.

We've found the easiest way to handle this is to store the command object in the session. This seems like a pretty obvious solution but it can have it's problems when combined with ORM tools such as Hibernate. We found it impractical to work with lazily loaded object graphs when the command object is stored in the session if using a Hibernate Session per Request strategy such as the OpenSessionInViewFilter. This is the case without reattaching the command object to the hibernate session on every request which can present problems. This method therefore requires the entire object graph that could be worked on to be fetched when the initial load of the top level object is performed and the object to remain in a detached state until a save is performed.

Obviously this approach is not feasible for all data models,so it may be necessary to employ a different approach such as using Long Hibernate Sessions that span multiple http requests.

It is not within the scope of this document to discuss Hibernate or other ORM strategies in detail but rather we thought it useful to highlight the potential problems in using multi-page forms.

Form Page Navigation

Navigating between pages works differently depending on whether the user is in read or edit mode. When in read mode, it is as simple as providing links to the specific controller. When in edit mode things are different because any changes made to the form need to be bound to the command object backing the page before navigating to the target form page. As part of the Spring Layout JSP Tag library a default panel (page) tab renderer is supplied to output and format tabs for moving between pages in read and edit mode. These are detailed in the JSP / LayoutForm section.

From a controller perspective the main thing to be aware of is the controller method for navigating between pages.

```
public ModelAndView changePanel(HttpServletRequest request,
                                                  HttpServletResponse response,
                                                  Object command,
                                                  BindException errors) throws Exception
```

This controller method is used for changing panels when a form is in edit mode. It checks the request for the *WebConstants.TARGET_PANEL_PARAM* parameter, binds the current form data to the command object and redirects the browser to the target page.

Validation

Multi-page controllers have increased complexity in validation due to the fact that they need to validate not only the current page but all pages within the Panel Form Group. When the *validate* method is called on the panelForm group from the controller lifecycle as shown below, the validation rules for each page need to be executed in order to validate the entire form. Due to potential differences in the command objects between controllers within a multi-page form group, such as adapted or wrapped / nested command objects , the validation for each page calls upon the getCommandObject method for the page's controller to resolve the object as it is used on that page.

More often than not, each page within a panel form group will return identical command objects but in the case of an adapted or wrapped object this may be necessary.

A simple and fairly trivial example of a user registration form is as follows. Note that it's unlikely we'd implement this use case in this fashion but it serves to illustrate the point.

Page one of the form collects the user's personal details such as name and address. The command object for this form is the user object so the paths to form fields against the command object could be something like the following.

```
user.firstName
user.lastName
user.streetAddress1
```

... and so on.

On page two of the form you want to capture the login details including the username and the user's password. Part of this process is to add a password confirmation field which is not part of the model but to make use of validation rules and Spring binding you decide to nest this command object under a form object which contains the User and passwordConfirm properties. This wrapping or nesting of the mode in the command object is performed by the controller for form page two. Subsequently the path to the fields on this form are a level deeper:

```
userForm.user.userName
userForm.user.password
userForm.passwordConfirm
```

Having two different objects as the command object would present a problem here if the user object was used to validate all form pages. However, the validation object asks each controller to serve up it's specific command object so the object graph for each page reflects the actual paths through the object graph used in the form field definitions and their validation rules.

The capabilities and configuration of Spring Layout validation is documented in greater detail in the validation chapter.

Validation Mode

Multi-page forms require some special behaviour when it comes to page validation. As form data is effectively submitted when navigating between form pages, it is necessary to differentiate between a form submission that requires validation such as a *save*, and form submission that is simply part of navigation. On top of this there are also situations when navigating between pages require validation.

For example, if on a two page form a user submits the form causing validation failures on both form pages. The user is returned to the current form with errors highlighted to be fixed. They are also shown through highlighting of the inactive form page that there are errors on that page. The user fixes all the validation failure on the current form and then clicks on the 'tab' to go to the second form page. At this point, the validation is still required so that when the next page opens it still shows the errors for that page.

To handle this, a special validation mode exists. Setting of the validation mode is simple and can be achieved through setting the hidden form field *validateMode*. The *validateMode* field is rendered for every form by the Spring Layout form tags and is used to transport the validation mode between requests. Turning the validation mode on or off is as simple as setting this field to '1' or '0'.

For example, on saving of a form, validation is required. The following Javascript could be used to firstly set the field and secondly submit the form to the controller calling the save method:

```
function saveUser(formName)
{
     setFieldValue("validateMode", "1");
     performAction(formName, "saveUser");
}
```

The two javascript functions shown here are part of the Spring Layout Javascript Library and are fairly self explanatory by name. More details about these can be found in the Javascript chapter. The point to notice here is that the field *validateMode* has been set to '1' which puts the form into validation mode. The form will now stay in validation mode while the user moves between form pages until validation mode is turned off. Through setting the *validateMode* field to '0'.

A case where this might be particularly useful is a form where two different types of 'save' are possible.

- Save as Draft – no validation required.
- Submit for Approval– requires validation before moving to approval stage.

In this case, saving as draft turns validation mode off as part of it's submission process, whereas submitting for approval turns it on.

Displaying all Form Pages at Once

A common requirement of web applications is providing printing of application content. Printing forms that span multiple pages presents problems if a server request is required to navigate between pages.

Multi-page controllers can easily be used to print all or a select group of pages at once. The main consideration that needs to be noted is that for multi-page forms that are not based on a common command object like in the previous section where one or more are wrapped / nested or adapted, the command object needs to be swapped out during page rendering. If every controller in the group returns the same command object this is not a concern but for the same reason as for validation, if form fields are to bind correctly against their command object the command object must be of the correct type.

To handle swapping of the command object within a JSP page Spring Layout provides the SwapCommandObject tag. This tag can be used inline within a JSP to swap the command object for that returned by the named controller. More detail about this tag and its use is in the JSP tag reference chapter.

Controller Configuration Reference

The following table details all Spring Layout settable properties for controllers that extend AbstractPanelFormController:

Attribute	**Description**
panelFormGroup	Reference to the PanelFormGroup Spring bean this controller is to be used with. The panelFormGroup bean holds references to all panelForm beans that configure rules relating to validation and layout for fields on the page.
defaultEditMode	true / false. Determines whether the page should open in edit or read mode by default if not specified in the url
defaultProperty-EditorsAllowEmpty	True / false. If set to true will register all default property editors to allow empty values to be seen as null and valid. Default Spring behaviour will see a null value for an integer as a binding error. Also registers custom editors for Calendar and FieldAccessibleCalendar classes.
pageAccessResolver	Reference to an optional PageAccessResolver used for determining a user's access to the current page and functions within it.
redirectFormView	The name of the view (defined in views.properties or equivalent) that redirects to the current controller. This is required so that a redirect can be used after all form submission / binding. This includes changing panels / tabs or any form of page refresh.
formView	The name of the view used to render the current page. Used by showForm as in core SpringMVC.
commandName	Name by which to store and reference the command object.

Key: Shaded cells indicate properties important to Spring Layout but part of the core Spring MVC framework

Controller Lifecycle

The Spring Layout controller lifecycle extends that of SimpleFormController. The following activity diagrams shows in detail the flow of a controller for a GET request.

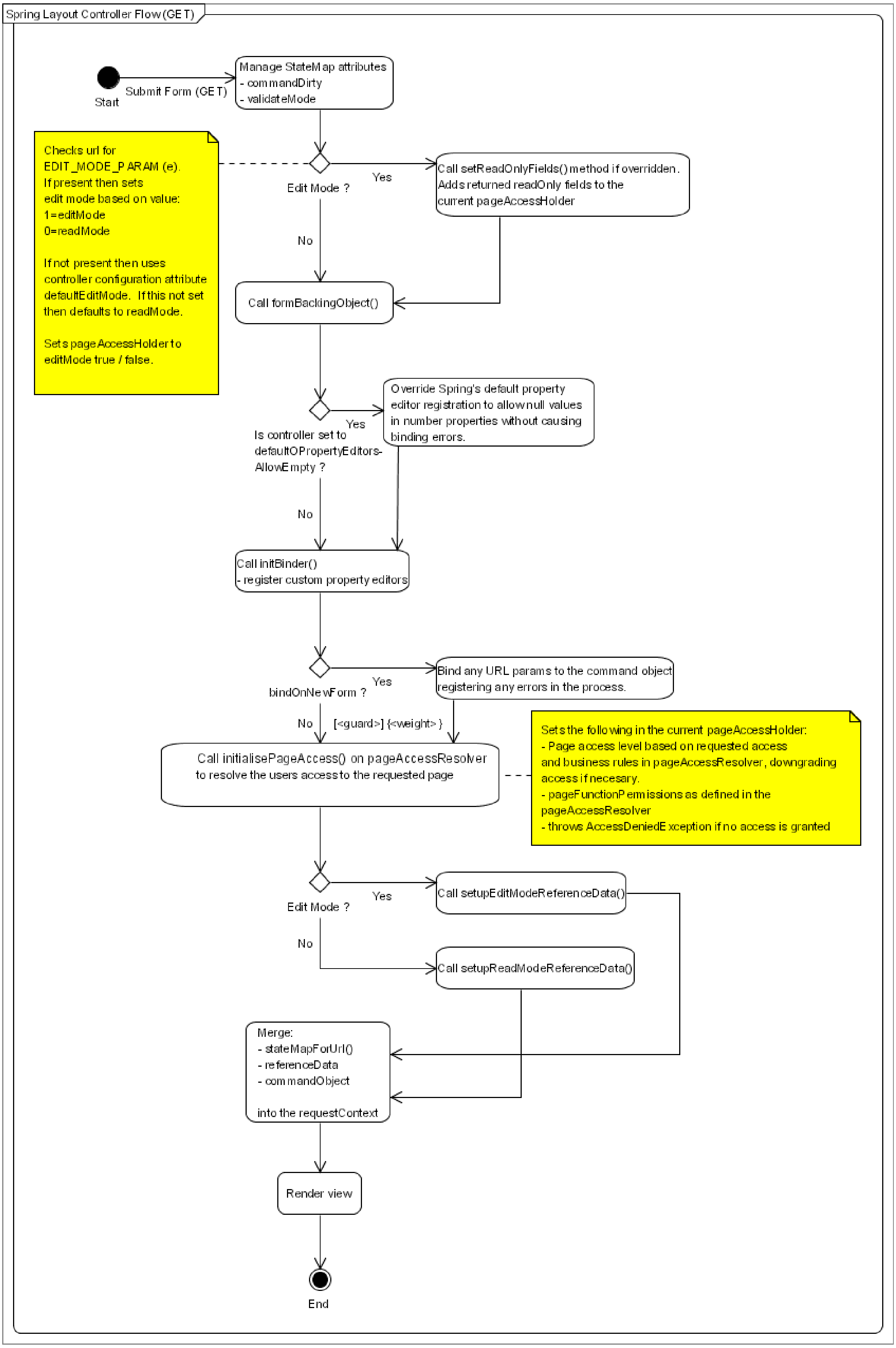
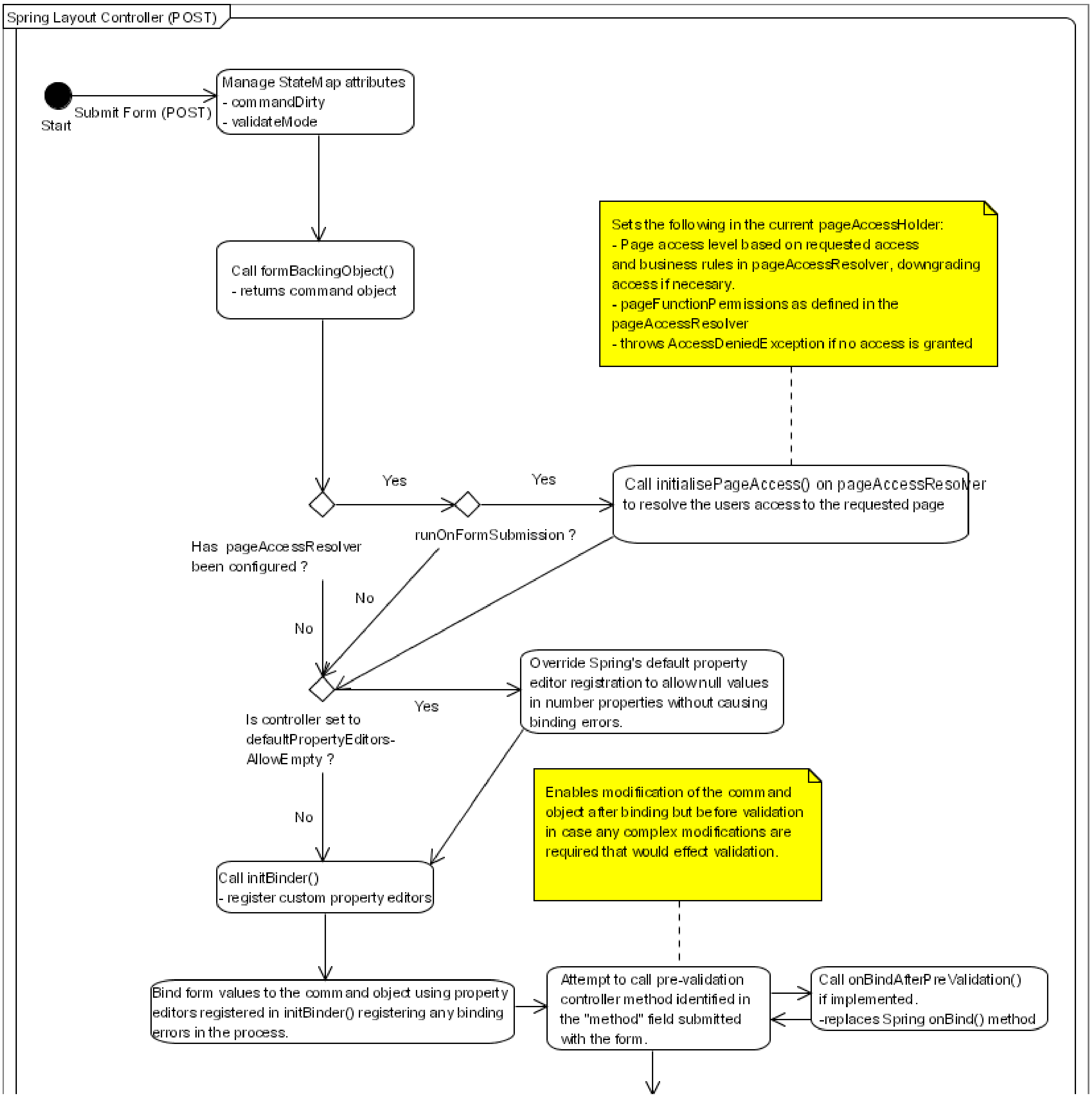

Spring Layout Controller Flow (GET)
Start
Submit Form (GET)
Manage StateMap attributes
- commandDirty
- validateMode
Checks url for
EDIT_MODE_PARAM (e).
If present then sets
edit mode based on value:
1=editMode
0=readMode
If not present then uses
controller configuration attribute
defaultEditMode. If this not set
then defaults to readMode.
Sets pageAccessHolder to
editMode true / false.
Edit Mode ?
Yes
Call setReadOnlyFields() method if overridden.
Adds returned readOnly fields to the
current pageAccessHolder
No
Call formBackingObject()
Override Spring's default property
editor registration to allow null values
in number properties without causing
binding errors.
Yes
Is controller set to
defaultOPropertyEditors-
AllowEmpty ?
No
Call initBinder()
- register custom property editors
bindOnNewForm ?
Yes
Bind any URL params to the command object
registering any errors in the process.
No
[<guard>] {<weight>}
Call initialisePageAccess() on pageAccessResolver
to resolve the users access to the requested page
Sets the following in the current pageAccessHolder:
- Page access level based on requested access
and business rules in pageAccessResolver, downgrading
access if necesary.
- pageFunctionPermissions as defined in the
pageAccessResolver
- throws AccessDeniedException if no access is granted
Edit Mode ?
Yes
Call setupEditModeReferenceData()
No
Call setupReadModeReferenceData()
Merge:
- stateMapForUrl()
- referenceData
- commandObject
into the requestContext
Render view
End

Figure 2: Controller flow for GET requests

The following activity diagrams shows in detail the flow of a controller for a POST request.


Spring Layout Controller (POST)
Start
Submit Form (POST)
Manage StateMap attributes
- commandDirty
- validateMode
Call formBackingObject()
- returns command object
Sets the following in the current pageAccessHolder:
- Page access level based on requested access
and business rules in pageAccessResolver, downgrading
access if necesary.
- pageFunctionPermissions as defined in the
pageAccessResolver
- throws AccessDeniedException if no access is granted
Yes
Yes
Call initialisePageAccess() on pageAccessResolver
to resolve the users access to the requested page
Has pageAccessResolver
been configured ?
runOnFormSubmission ?
No
No
Override Spring's default property
editor registration to allow null values
in number properties without causing
binding errors.
Yes
Is controller set to
defaultPropertyEditors-
AllowEmpty ?
Enables modification of the command
object after binding but before validation
in case any complex modifications are
required that would effect validation.
No
Call initBinder()
- register custom property editors
Bind form values to the command object using property
editors registered in initBinder() registering any binding
errors in the process.
Attempt to call pre-validation
controller method identified in
the "method" field submitted
with the form.
Call onBindAfterPreValidation()
if implemented.
-replaces Spring onBind() method

cont rued on next page

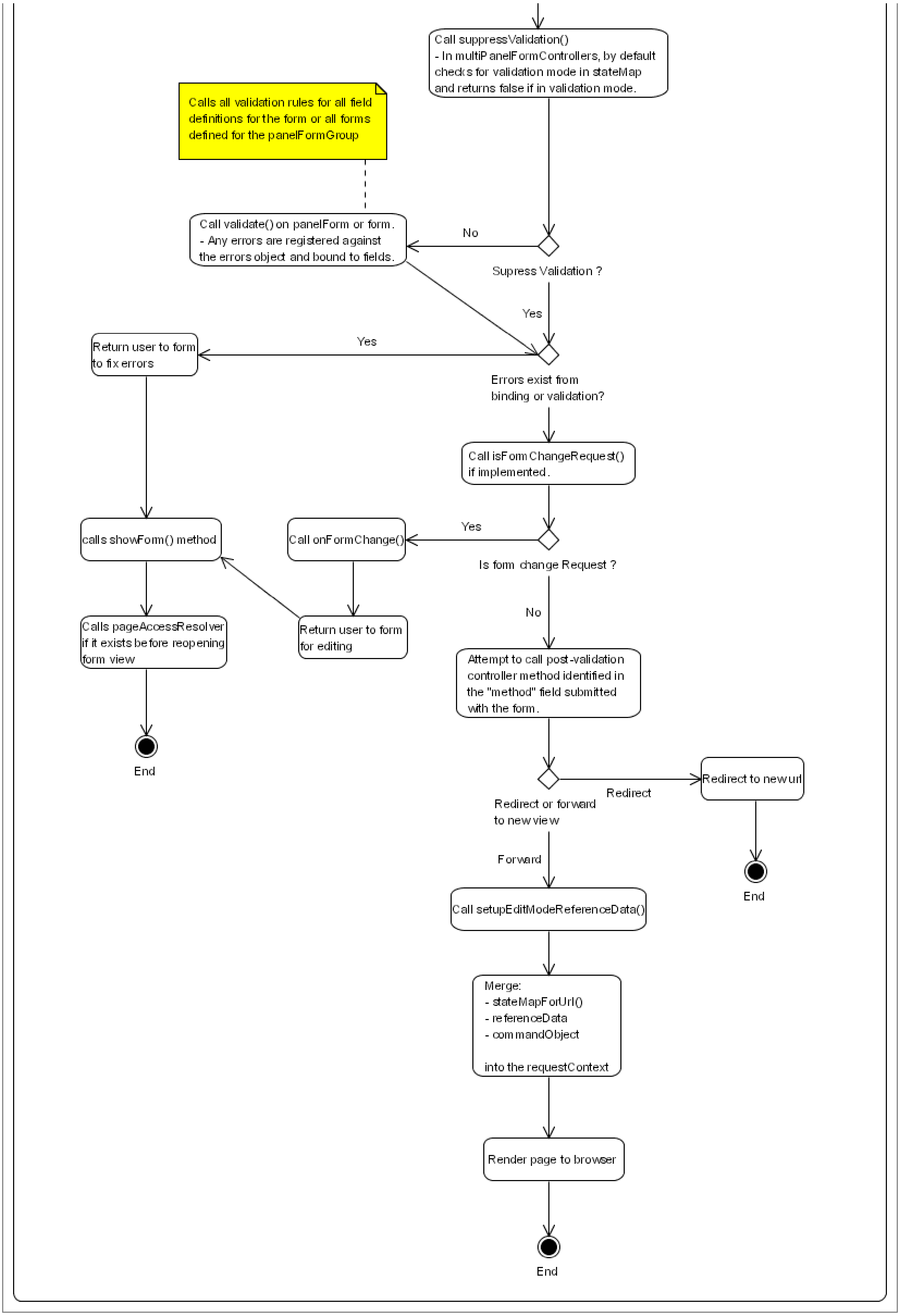

Call suppressValidation()
- In multiPanelFormControllers, by default checks for validation mode in stateMap and returns false if in validation mode.
Calls all validation rules for all field definitions for the form or all forms defined for the panelFormGroup
Call validate() on panelForm or form.
- Any errors are registered against the errors object and bound to fields.
No
Supress Validation ?
Yes
Return user to form to fix errors
Yes
Errors exist from binding or validation?
Call isFormChangeRequest() if implemented.
calls showForm() method
Call onFormChange()
Yes
Is form change Request ?
Calls pageAccessResolver if it exists before reopening form view
Return user to form for editing
No
Attempt to call post-validation controller method identified in the "method" field submitted with the form.
End
Redirect to new url
Redirect
Redirect or forward to new view
Forward
End
Call setupEditModeReferenceData()
Merge:
- stateMapForUrl()
- referenceData
- commandObject
into the requestContext
Render page to browser
End

Figure 3: Controller flow for POST requests

Security

As touched on in the Controllers chapter, Spring layout provides a flexible security authorization layer which is primarily based around but not limited to the Controller Command Object. This layer of security can be combined with any authentication strategy such as Acegi Security and comes in the form of PageAccessResolvers.

Page Access Resolvers allow security at three levels:

1. Page Level Security – allows runtime evaluation of rules written in Java to allow, downgrade or reject the requested page.
2. Field Level Security – allows the conditional identification of fields to be rendered in read mode for the current request.
3. Page Function Security – allows the specification of Page Functions and their associated permissions based on the current user and context.

Page Access Resolvers

Page Access Resolvers are optionally injected into Spring Layout Controllers and called following the resolution of the command object through *formBackingObject* and, for controllers with *bindOnNewForm,* following binding of request parameters to the command object. The following simplified illustration of the flow shows where in the controller lifecycle, pageAccessResolvers are executed.

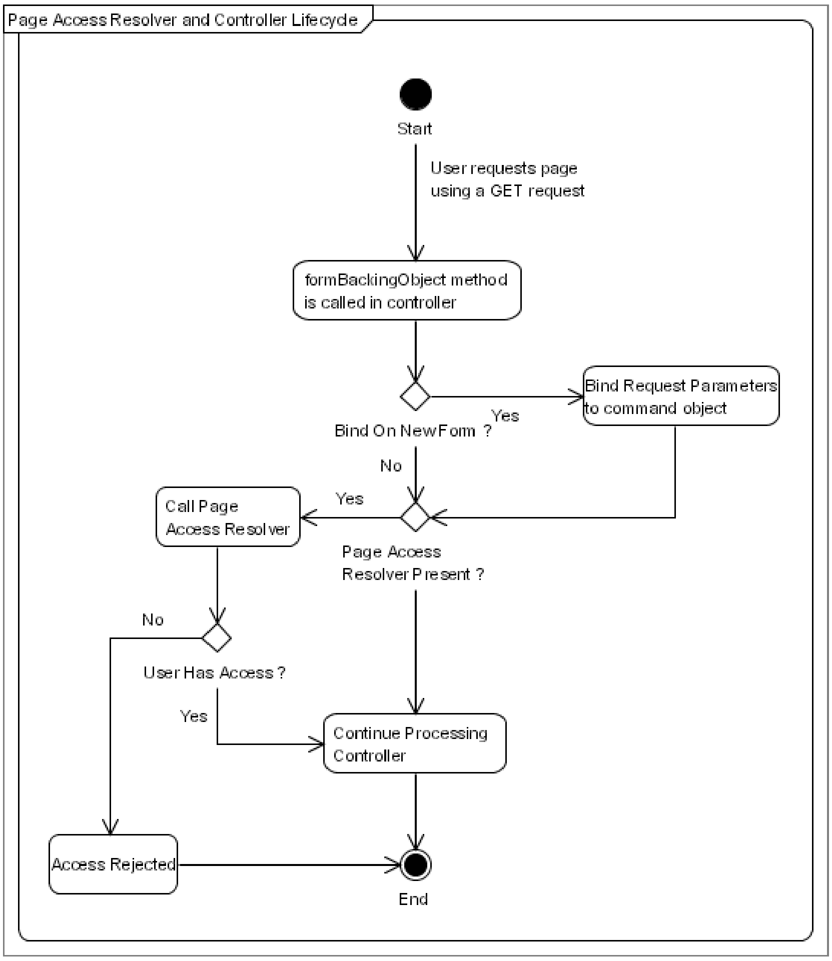

Page Access Resolver and Controller Lifecycle
Start
User requests page using a GET request
formBackingObject method is called in controller
Bind Request Parameters to command object
Yes
Bind On NewForm ?
No
Yes
Call Page Access Resolver
Page Access Resolver Present ?
No
User Has Access ?
Yes
Continue Processing Controller
Access Rejected
End

Figure 4: Page Access Resolver and Controller Lifecycle

By default Page Access Resolvers are only run during a GET request and not during a POST. For many applications this is acceptable but it should be considered when designing a security strategy. If pageAccessResolvers are to be run on a POST, the *runOnFormSubmission* property of the Page Access Resolver should be set to 'true'.

Implementing a PageAccessResolver

An implementation of a Page Access Resolver should extend the net.sf.springlayout.web.security.AbstractPageAccessResolver class. This class provides the following abstract methods to be implemented by the concrete PageAccessResolver.

```
protected abstract AccessLevel getPageAccessLevel(
                                                  HttpServletRequest request,
                                                  Object object);

protected abstract PageFunctionStateMap getPageFunctionPermissions(
                                   HttpServletRequest request,
                                                  Object object,
                                                  boolean editMode);
```

AbstractPageAccessResolver also provides an optional method that can be overridden if there is a requirement for conditional read-only fields.

```
protected List defineReadOnlyFields(HttpServletRequest request, Object object)
```

These methods and related object types will be explained in more detail in the following sections.

Configuration

Page Access Resolvers are configured as Spring beans and injected into controllers using the *pageAccessResolver* property. The following example demonstrates this:

```
<bean id="userPageAccessResolver"
class="net.sf.springlayout.sampleapp.web.security.UserPageAccessResolver" />

<bean id="userController" parent="abstractItrackController"
             class="net.sf.springlayout.sampleapp.web.controller.UserController" >

     <property name="pageAccessResolver">
             <ref bean="userPageAccessResolver" />
     </property>

     ...

</bean>
```

Page Level Security

The first of the two methods that must be implemented in a Page Access Resolver is responsible for determining the page access level.

```
protected abstract AccessLevel getPageAccessLevel(HttpServletRequest request, Object object);
```

As can be seen from inspection of the method signature, this method provides the command object and request for use in determining the access level for the current request. It would commonly be a requirement to evaluate a user's access based on their identity but due to the varying implementation of the user, or authentication object in different systems this has been left out. Implementing applications should provide their own strategy for resolving the current user if it is required at this point such as storing them in the request / session or in ThreadLocal.

An example of a simple implementation of a Page Access Resolver might look something like the following:

```
protected AccessLevel getPageAccessLevel(HttpServletRequest request,
                                                         Object object)
{

      User user = (User) object;

      if (user.isNew())
      {
         return AccessLevel.EDIT_ACCESS;

      }
      else
      {
         Authentication authentication =
SecurityContextHolder.getContext().getAuthentication();

         if (authentication.getName().equals(user.getUsername())
         {
            return AccessLevel.EDIT_ACCESS;
         }
         else
         {
            return AccessLevel.READ_ACCESS;
         }
      }

}
```

In this example, the command object is firstly cast to it's correct type of *User*. If the user is a new object then we want to allow anyone access so the Access Level returned is *AccessLevel.EDIT_MODE.* Access levels are a type safe enumeration with possible values of

- Access Level.EDIT_ACCESS
- Access Level.READ_ACCESS
- Access Level.NO_ACCESS

The second part to this example checks to see if the current user is the same user as is being opened and returns, Access Level.EDIT_ACCESS if they match and Access Level.READ_ACCESS if they don't. The implementation for authentication in this case is Acegi Security, so the authentication object is retrieved using the Acegi SecurityContextHolder.

Field Level Security

Spring Layout provides a concept of Modes for fields rendered using JSP pages. This enables a single form to be used for both read only and editable views of a form. Field level security provides the added benefit of being able to conditionally identify particular fields based on their path in relation to the command object, as read only. This can be particularly useful if you want certain parts of a form to be locked down from specific users or groups of users.

Say for instance you wanted to extend the previous example to allow the administrator of the application access to edit all user's details but prevent them from changing key personal data such as the user's first and last name.

Along with some minor modifications to the getPageAccessLevel method from the previous example, you would need to override the following method:

```
protected List defineReadOnlyFields(HttpServletRequest request, Object object)
```

This method allows a list of Strings to be returned that represent the paths to the fields that should be displayed as read only.

Similarly to the *getPageAccessLevel* method example, the *defineReadOnlyFields* method should evaluate the current context based around the command object if necessary and return it's list of fields. In this case the rules are quite simple in that it should check to see if the current user is an Administrator (by whatever role based security is implemented) and return the read only fields if this is the case.

There would be no point evaluating this code if the page was in read mode so Spring Layout does not call the *defineReadOnlyFields* method unless the page is in edit mode.

```
protected List defineReadOnlyFields(HttpServletRequest request, Object object)
{
      List list = new ArrayList();
      Authentication authentication = SecurityContextHolder.getContext().getAuthentication();

      if (ArrayUtils.contains(authentication.getAuthorities(),
            new GrantedAuthorityImpl(Role.ROLE_GLOBAL_ADMINISTRATOR)))
      {
         list.add("firstName");
        list.add("lastName");
      }

      return list;
}
```

Function Permissions

Function Permissions represent permissions a user has to functions or actions within a page such as *Save, Edit, Delete* etc. Most applications have functions of some kind and many require these functions to be selectively enabled, disabled or hidden depending on the current user and various other factors. As with the other Page Access Resolver features, the *getPageFunctionPermissions* method includes the command object and request as part of it's method signature. On top of these it also provides as a convenience, a boolean to indicate whether the page is in edit mode or not.

```
protected abstract PageFunctionStateMap getPageFunctionPermissions(
                                   HttpServletRequest request,
                                                  Object object,
                                                  boolean editMode);
```

A simple example of defining Page Function Permissions would be the alternate display of the *save* and *edit* buttons depending on the page mode.

```
protected PageFunctionStateMap getPageFunctionPermissions(
                                                  HttpServletRequest request,
                                                  Object object,
                                                  boolean editMode)
{

      PageFunctionStateMap functionMap = new PageFunctionStateMap();

      User user = (User) object;

      if(editMode)
      {
         functionMap.put("save", PageFunctionState.ENABLED);
      }
      else
      {
         functionMap.put("edit", PageFunctionState.ENABLED);
      }

      return functionMap;
}
```

Obviously far more complex conditions can be implemented based around the authenticated user and other contextual information if necessary.

PageAccessHolder

The PageAccessHolder is used to store state about the access levels for the current request. It is used throughout SpringLayout by the Controllers, PageAccessResolvers and JSP tags to determine how the application should behave and display.

The PageAccessHolder is evaluated and set for each request at appropriate times within the controller lifecycle and stored in ThreadLocal using the PageAccessManager. If the pageAccessHolder is to be referenced within application code it can be done using the static method *PageAccessManager.getPageAccessHolder()*.

The Page Access Holder contains various methods for querying the current access levels. It is advisable not to set access levels directly in the pageAccessHolder unless you are completely familiar with the lifecycle of Spring Layout. For the vast majority of cases it is more intuitive and easier to use the methods described in previous sections to achieve security related behaviour.

It is also important to to note that the PageAccessHolder should be considered the absolute authority on page access levels. Do not reference the url parameter directly to determine the page access level as this will not necessarily accurately represent the current situation. It is quite possible that although a user has requested a page for edit access, their access has been downgraded to read access through a Page Access Resolver. Obviously using the url parameter would be invalid in this situation.

JSP Tags

Security is integrated in some way into most of the the Spring Layout JSP form and field tags. These tags are detailed in the JSP Tags section. There are however some convenience tags that can be used within a page for security related concerns that are worth mentioning here.

Tag	Description
<layout:ifEditMode>	Use to wrap content to be displayed when the page is in edit mode
<layout:ifReadMode>	Use to wrap content to be displayed when the page is in read mode
<layout:hasPageFunction>	Use to wrap content to be displayed if the page has the specified function

Form and Field Layout Configuration

Central to Spring Layout is the concept of Form and Field definitions. These are used to configure rules relating to form fields for layout and validation purposes. When a Spring Layout JSP form or field tag is rendered, it firstly looks up it's corresponding definition to determine how it should be rendered and and how it should behave.

One of the first steps to creating a form in Spring Layout is to create a form definition with it's corresponding field definitions. To achieve this a Spring Application Context file is created in the “WEB-INF/layout” folder. The location of Spring Layout configuration is up to the developer although this is a recommended location for consistency and clarity. Following is an example file for a user form called “userFormDefinition.xml”:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <bean id="commentForm" class="net.sf.springlayout.web.layout.LayoutForm">
     <property name="formAction" value="userDetails.html" />
     <property name="formName" value="userDetailsForm" />
     <property name="controllerBeanName" value="userDetailsController" />
        <property name="fieldDefinitions">
            <map>
                <entry key="id">
                    <bean class="net.sf.springlayout.web.layout.LayoutFieldDefinition">
                        <property name="fieldKey" value="id" />
                        <property name="mandatory" value="false" />
                    </bean>
                </entry>
                <entry key="firstName">
                    <bean class="net.sf.springlayout.web.layout.LayoutFieldDefinition">
                        <property name="fieldKey" value="firstName />
                        <property name="mandatory" value="true" />
                    </bean>
                </entry>
                <entry key="lastName">
                    <bean class="net.sf.springlayout.web.layout.LayoutFieldDefinition">
                        <property name="fieldKey" value="lastName" />
                        <property name="mandatory" value="true" />
                    </bean>
                </entry>
                <entry key="email">
                    <bean class="net.sf.springlayout.web.layout.LayoutFieldDefinition">
                        <property name="fieldKey" value="email" />
                        <property name="mandatory" value="true" />
                    </bean>
                </entry>

             ...

            </map>
        </property>
    </bean>
</beans>
```

The field definitions serve as a central location for defining the field attributes including field size and mandatory requirements including mandatory rules (covered later in this section).

One of the major benefits of the SpringLayout extensions is the generation of both front-end Javascript validation and back-end server side validation utilising the same field definition.

Using the above as an example the properties are explained below:

LayoutForm Attributes

Property	Description
id	This property is used to identify the bean
class	net.sf.springlayout.web.layout.LayoutFom
formAction	The definition of the form action generated into the HTML form tag <form name="commentForm" id="commentForm" action="comment.html" method="post">
formName	The definition of the form name and id generated into the HTML form tag <form name="commentForm" id="commentForm" action="comment.html" method="post">
controllerBeanName	The id of the controller bean name defined in controller-servlet.xml
fieldDefinitions	A map of field definitions used on the form see below (LayoutField Attributes)

LayoutField Attributes

Property	Description
class	net.sf.springlayout.web.layout.LayoutFieldDefintion
key	Key to the map of field definitions the actual field name
fieldKey	Unique field name, the same value as the key property
mandatory	Simple true/false attribute to indicate if field is mandatory or not
minLength	The minimum length of the field
maxLength	The maximum length of the field
mandatoryConditions	Set of mandatory conditions for the field
mandatoryCondition	An individual expression based mandatory condition
mandatoryListType	and/or the type of condition, the conditions are “anded” or “ored” together (default = “and”)
validationRules	Validation rules to identify incorrect format of input data.
dependantValidationFieldKeys	Errors can be placed on other labels in the UI if required, Errors are mapped to all keys in this list.
masterFieldId	Id of the field on the page that the Javascript will be attached too

Default Field Attribute

To ease the field configuration in the layout xml file it is possible to set up a default field attribute as follows:

```
<entry key="DEFAULT_FIELD">
   <bean class="net.sf.springlayout.web.layout.LayoutFieldDefinition">
      <property name="fieldKey" value="DEFAULT_FIELD" />
      <property name="mandatory" value="false" />
   </bean>
</entry>
```

This has the effect of being used by any field that is not defined in the XML explicitly, and giving it the properties specified in this default entry.

Panel Forms / Form Groups

SpringLayout provides the ability to support a group of form pages that work together as a 'panel group', allowing for multiple forms to be used in cross form validation and for the creation of 'tabbed' or 'wizard' style step-by-step forms.

Configuring a group of forms that work together in a form group requires an additional bean definition commonly maintained in a special XML file for organisational clarity, namely “layoutPanelFormGroups.xml”. This bean is configured as follows:

```
<bean id="userFormGroup" class="net.sf.springlayout.web.layout.LayoutPanelFormGroup">
     <property name="panelForms">
            <list>
                   <ref bean="userDetailsPanelForm" />
                   <ref bean="loginDetailsPanelForm" />
            </list>
     </property>
</bean>
```

In this example a LayoutPanelFormGroup has been set up with user details and login details panels each of which has its own panel form. Each panel form has similar configuration to the standard layout form with a few additions. Following is an excerpt of the “userDetailsPanelForm” properties:

```
<bean id="userDetailsPanelForm" class="net.sf.springlayout.web.layout.LayoutPanelForm">

     <property name="panelLabel" value="User Details" />
     <property name="index" value="0" />
     <property name="formAction" value="userDetails.html" />
     <property name="formName" value="userDetailsForm" />
     <property name="controllerBeanName" value="userDetailsController" />
     <property name="redirectFormViewName" value="user_userDetails-redirect" />
     <property name="panelFormTabStateManager">
            <bean class="net.sf.springlayout.web.layout.UserDetailsPanelTabStateManager" />
     </property>
     <property name="fieldDefinitions">
     ...
     </property>
</bean>
```

LayoutPanelForm Attributes

Property	Description
id	This property is used to identify the bean
class	net.sf.springlayout.web.layout.LayoutPanelForm
panelLabel	A text label that can be used on the UI as for example a text label on the tab for the panel that is used for navigation
index	Order in the list of the panel group 0 = first panel
formAction	The definition of the form action generated into the HTML form tag
formName	The definition of the form name and id generated into the HTML form tag
controllerBeanName	The id of the controller bean name defined in controller-servlet.xml
redirectFormView	The name of the view that the form redirects to after a post, this ensures that if a client refreshes the browser after a post, information is not posted a second time
panelFormTabStateManager	This allows for a manager to be used to control the edit state of the specific panel form
fieldDefinitions	A map of field definitions used on the form

Validation Configuration

Validation is one of the key aspects of the SpringLayout extensions. Out of the box, Spring Layout provides a rich declarative validation framework for common validation requirements. It provides the ability to easily write custom validation rules for specific validation requirements. Validation is incorporated into the tag library to visually display errors to the UI as well as generate Javascript for validation that is executed on the browser. It is also incorporated into the Controller Framework to validate on the server side to ensure that only valid data is sent through to the service layer for further processing.

The correct configuration carried out in the field definitions is all that is needed to achieve this two level validation, the configuration options available for validation offer a great deal of flexibility and power.

This section goes through the configuration options and shows some examples of some common validation configurations.

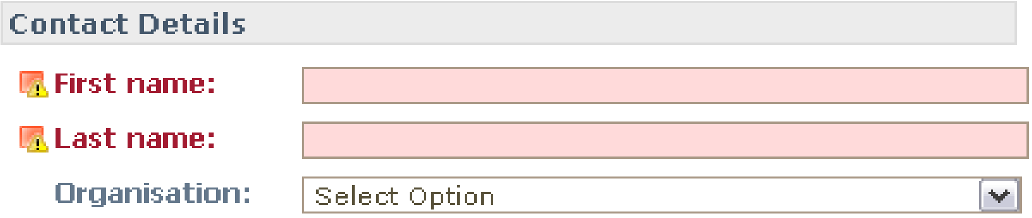
Simple Mandatory Field Validation

In it's most basic form, mandatory fields can be configured without a condition. Often a field may be mandatory always so an easy configuration is provided. The following example shows a basic mandatory field configuration where the firstName field is always mandatory:

```
<entry key="firstName">
  <bean class="net.sf.springlayout.web.layout.LayoutFieldDefinition">
     <property name="fieldKey" value="firstName" />
     <property name="mandatory" value=”true”>
  </bean>
</entry>
```

The effect of this simple configuration is as follows:

1. The field and it's label are rendered as mandatory using appropriate CSS classes
2. On submission of the form (in validation mode), the field is validated as mandatory. If the field fails mandatory field validation an error is bound to the field in the *errors* object. Note that if mandatory field validation fails, there are no further validation rules executed against the field. Validation rules for other fields are still evaluated.
3. The *showForm* method is called in AbstractBaseFormController returning the form to the user.
4. The form renders with appropriate CSS to highlight fields with errors.



Contact Details

First name:

Last name:

Organisation: Select Option

Figure 5: Example of basic form error highlighting

**Note: Spring Layout provides a default look and feel which can be overridden for individual applications.

Conditional Mandatory Field Validation

The *mandatoryCondition* property allows for the definition of a single condition, to determine if a field should be made mandatory or not.

```
<entry key="email">
  <bean class="net.sf.springlayout.web.layout.LayoutFieldDefinition">
     <property name="fieldKey" value="email" />
     <property name="mandatoryCondition">
           <bean class="net.sf.springlayout.web.validator.condition.PathBasedCondition">
               <property name="operator" value="==" />
               <property name="path" value="receiveEmail" />
               <property name="value" value="true" />
           </bean>
     </property>
  </bean>
</entry>
```

Referring to the above LayoutFieldDefinition the “email” field will be made mandatory if the “receiveEmail” has a value of “true”. For depended validation to work in this example both of these fields must exist on the current form page. It is also possible to validate dependencies across form pages, the details of which will be covered later.

Validating Against Multiple Mandatory Conditions

The *mandatoryConditions* property allows for the definition of a list of conditions to determine if a field should be mandatory or not. Multiple conditions can be executed using the *AND* or *OR* operator. With *AND* being the default it is not necessary to define this for AND conditions. The property *mandatoryListType* must be used if the conditions are to be evaluated together using *OR.*

```
<entry key="email">
  <bean class="net.sf.springlayout.web.layout.LayoutFieldDefinition">
     <property name="fieldKey" value="email" />
      <property name="mandatoryListType" value="OR" />
      <property name="mandatoryConditions">
         <list>
           <bean class="net.sf.springlayout.web.validator.condition.PathBasedCondition">
              <property name="operator" value="==" />
              <property name="path" value="subscribeToTechMailList" />
              <property name="value" value="true" />
           </bean>
           <bean class="net.sf.springlayout.web.validator.condition.PathBasedCondition">
              <property name="operator" value="==" />
              <property name="path" value="subscribeToBusinesMailList" />
              <property name="value" value="true" />
           </bean>
        </list>
     </property>
  </bean>
</entry>
```

Referring to the above LayoutFieldDefinition the “email” field will be made mandatory if the user has elected to subscribe to the technical OR business mailing lists.

Cross Panel Validation Dependencies

The onThisForm property defaults to *true* and indicates that the field identified in the path property is on the current form, setting this property to *false* indicates that the validation is across panels / form pages.

```
<entry key="email">
  <bean class="net.sf.springlayout.web.layout.LayoutFieldDefinition">
     <property name="fieldKey" value="email" />
     <property name="mandatoryCondition">
           <bean class="net.sf.springlayout.web.validator.condition.PathBasedCondition">
               <property name="operator" value="==" />
               <property name="path" value="receiveEmail" />
               <property name="value" value="true" />
               <property name="onThisForm" value="true" />
           </bean>
           <bean class="net.sf.springlayout.web.validator.condition.PathBasedCondition">
               <property name="operator" value="&lt" />
               <property name="path" value="age" />
               <property name="value" value="100" />
               <property name="onThisForm" value="false" />
           </bean>
     </property>
  </bean>
</entry>
```

Referring to the above LayoutFieldDefinition the “email” field is mandatory based on the “receiveEmail” field being equal to “true” and the user's age being less than 100. We figure anyone older than 100 can be excused for not having an email address and accidently requesting to receive email.

Raising Errors on Alternate Fields

Errors in Spring are lodged against the errors object and can be either global or bound to a specific field or path of the command object. There are times when it is desirable to raise errors against a field other than the field with the error such as when a single label is used for multiple fields and any of those fields need to show errors in the label for those fields.

```
<entry key="subType">
   <bean class="net.sf.springlayout.web.layout.LayoutFieldDefinition">
      <property name="fieldKey" value="subType" />
      <property name="mandatoryConditions">
         <list>
           <bean class="net.sf.springlayout.web.validator.condition.PathBasedCondition">
              <property name="operator" value="!=" />
              <property name="path" value="type" />
              <property name="value" value="10" />
           </bean>
           <bean class="net.sf.springlayout.web.validator.condition.PathBasedCondition">
              <property name="operator" value="!=" />
              <property name="path" value="identified" />
              <property name="value" value="" />
           </bean>
         </list>
      </property>
      <property name="dependentValidationFieldKeys">
         <list>
           <value>type</value>
         </list>
      </property>
    </bean>
</entry>
```

Referring to the above LayoutFieldDefinition the “subType” field is mandatory based on the “type” field not being equal (!=) to 10 and the “identified” field not being an empty string (“”). When the validation error occurs put the error onto the “type” field as subtype does not have a label on the UI.

PathBasedCondition attributes

Property	**Description**
class	net.sf.springlayout.web.validator.condition.PathBasedCondition
operator	Comparison operator “==”, “!=”, “&lt;”, “&lt;=”, “&gt;”, “&gt;=” Equals, Not Equals, Less Than, Less Than and Equals, Greater Than, Greater Than and Equals.
path	Path to the field being evaluated for the condition
value	The value the field is checked against using the defined operator
onThisForm	Specifies if the path field being referenced is on this form/panel – default is 'true'

Validation Rules

Spring Layout comes with several validation rules for common requirements. It is envisaged that this set will grow with time and contributions. The validation rules currently available include:

- Email Validator
- Range Validator (Number / Date) ??
- TBA

Writing Custom Validation Rules

Spring Layout makes it easy to write custom validation rules and provides the framework for implementing both Java and Javascript validation in the one place. Once the validation rule is written and implemented against a field definition, Spring Layout takes care of executing the validation rules in the front and back end at the appropriate times.

The easiest way to demonstrate a custom validation rule is to look at an existing one. To start with a validation rule must implement the *ValidationRule* interface.

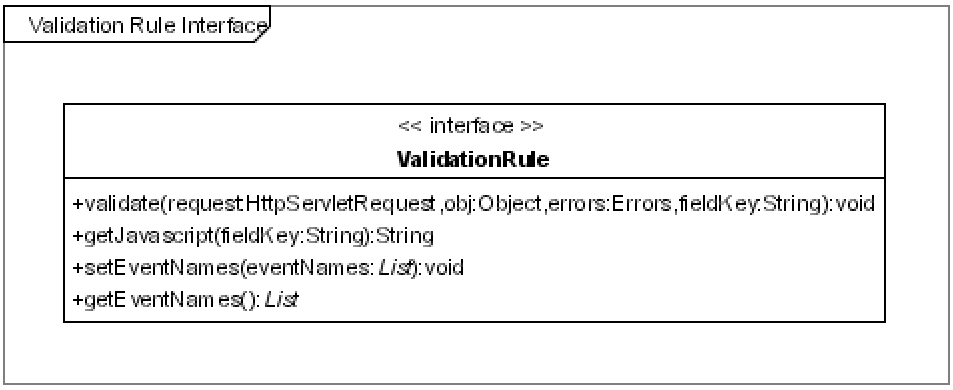

Validation Rule Interface
<< interface >>
ValidationRule
+validate(request:HttpServletRequest,obj:Object,errors:Errors,fieldKey:String):void
+getJavascript(fieldKey:String):String
+setEventNames(eventNames:List):void
+getEventNames():List

Figure 6: Validation Rule Interface

JSP Tags

The JSP Tag library in Spring Layout, offers a rich set of functional control tags, mode or status tags, security tags, validation tags and configuration tags to ease the development process and help quickly layout a form. The tags tightly integrate into the rest of the library as well as the Spring Framework itself to provide a rapid forms development platform.

The Tags provided by Spring Layout can be broken into the following types:

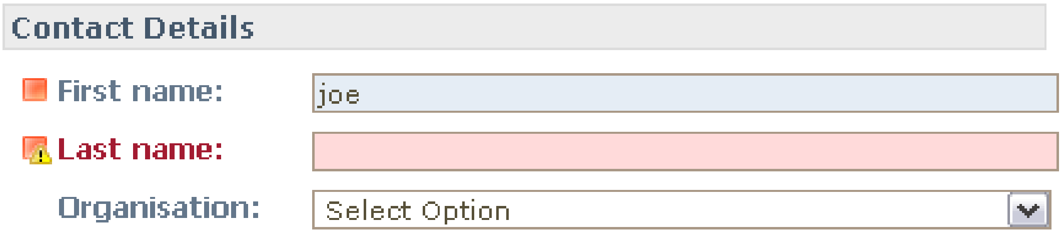
Form and Field Tags

Form and Field tags facilitate the building of html forms and integrate both validation and security functionality. As much as possible, they closely resemble their html tag equivalents and allow the pass through of all regular html attributes.

Field Tags

Alongside the tags used for rendering html form elements, Spring Layout also provides a field label tag that is bound to it's associated field. Besides the fact that it encourages correct use of html for forms creation, it also enables various front end behaviours such as highlighting of field labels on error and disabling of field labels when a field is disabled. There are various form behaviours such as this that can be picked up 'for free' simply by using the appropriate Javascript functions in place of direct DOM manipulation. This is detailed in the Javascript Chapter.

To illustrate some basic front end behaviour, the following two screen shots show a form after a submission that failed validation due to a mandatory field not being filled in. The first screen shot shows the highlighting of the “Last Name” field due to it failing validation. Notice change in the mandatory field image as well. This is all achieved through CSS classes applied to each element so is completely customisable on a per application basis.

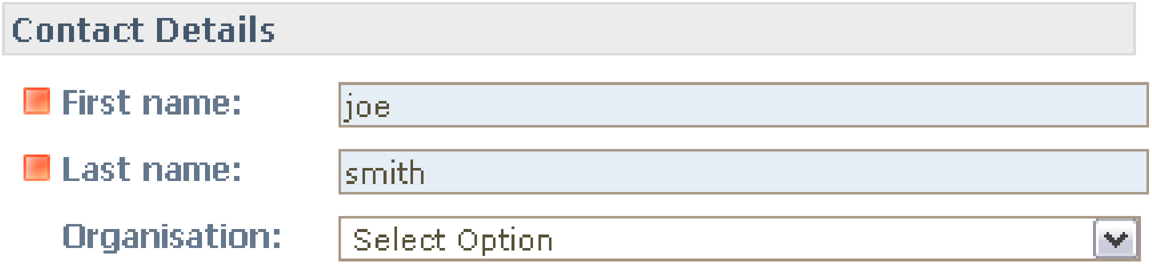


Contact Details

First name:	joe
Last name:	
Organisation:	Select Option

Illustration 2: Demonstrates highlighting of form errors after failed submission

The second screen shot shows the result of the user filling in the “Last Name” field and leaving the field, triggering the *onblur* field event. No submission of the form was required for this as it is taken care of in the front end.



Contact Details

First name: joe

Last name: smith

Organisation: Select Option

Illustration 3: Demonstrates the result of validation errors being corrected

It is important to highlight that this type of front-end behaviour is not unique to mandatory fields. All validation rules provide both front and back-end validation. It is also possible for custom validation rules to achieve the same level of front-end behaviour.

The code snippet for the above example looks as follows:

```
…

<h2>
     <spring:message code="textLabel.userContactDetails.heading" />
</h2>
<table class="formTable" summary="form layout table">
     <tr>
            <th>
                   <layout:label path="user.firstName" />
            </th>
            <td>
                   <layout:input path="user.firstName" />
            </td>
     </tr>
     <tr>
            <th>
                   <layout:label path="user.lastName" />
            </th>
            <td>
                   <layout:input path="user.lastName" />
            </td>
     </tr>
     <tr>
            <th>
                   <layout:label path="user.organisation" />
            </th>
            <td>
                   <layout:select path="user.organisation">
                          <layout:options items="${organisationList}" label="title"
value="id"
                                 defaultLabelKey="textLabel.selectOption" defaultValue="" />
                   </layout:select>
            </td>
     </tr>
</table>

...
```

Those using the new Spring 2.0 tags or who have used Struts tags in the past would be familiar with style used for the Spring Layout tags.

Note: A full reference guide detailing all the attributes of the Spring Layout Tags can be found in the TagLibDoc documentation – see http://springlayout.sourceforge.net/tlddoc/index.html

Handling Null Values in Nested Paths

Anyone familiar with Spring MVC will likely have encountered the NullValueInNestedPathException. This exception is thrown when BeanWrapper tries to traverse a path that has a null value somewhere in the path except for the last position. For example if the path *user.address.suburb* had a null value at *suburb,* BeanWrapper would safely return null. If however the *address* object in this graph was null, BeanWrapper would throw a NullValueInNestedPathException as it could not get past *address* to *suburb*.

Without justifying the reasons for this behaviour, this is a source of frustration for some users, especially when it comes to rendering values through Spring's field tags. The approach chosen to get around this problem in Spring Layout is to catch this exception and render the field as a disabled / read only field. This has two benefits. Firstly, it renders the page without error, and secondly it prevents any fields from being posted with the form data thus preventing the exception from occurring on binding to the model when submitted. If the field should be editable, then the solution, as always is to instantiate blank objects for binding against in formBackingObject.

Form Tags

Field tags must be wrapped in one of two types of form tag.

- <layout:form>
- <layout:multiPanelForm>

The form tags are mainly responsible for binding the form with it's appropriate Form Definition and fields / validation rules. In it's simplest use, the <layout:form> tag provides a singles attribute which identifies the name of the Spring Bean for the form Definition.

The more complex of the two form tags, the <layout:multiPanelForm> requires a little more configuration due to the added behaviour that it is responsible for.

Attribute	**Description**
panelFormGroupBeanName	Identifies the name of the Spring Bean configured for this Panel Form Group
currentPanelBeanName	Identifies the Spring Bean configured for this page / panel of the panel form group.
panelTabRendererBeanName	Identifies the Spring Bean configured for rendering the panel group tabs for this form group (see below).
panelFooterRendererBeanName	Identifies the Spring Bean configured for rendering the footer for this form group (see below).

Panel Tab & Footer Renderers

Multi Panel Forms require navigation to switch between form panels (pages). The MultiPanelForm Tag provides the ability to inject instances of the default panel tab renderer or a custom renderer for added flexibility.

The simplest method is to inject either or both of the default renderers which provide a tabbed style header for the form and a wizard (next / back) styled footer.

The tag in the JSP would look similar to the following:

```
<layout:multiPanelForm panelFormGroupBeanName="userPanelFormGroup"
   currentPanelBeanName="userPanelForm"
   panelTabRendererBeanName="defaultPanelTabRenderer"
   panelFooterRendererBeanName="defaultPanelFooterRenderer">
```

The following default panel tab renderer beans would also be required. A good place for these to exist is within the same application context files that the panelFormGroup beans are in.

```
<bean name="defaultPanelTabRenderer"
class="net.sf.springlayout.web.layout.taglib.renderer.DefaultPanelTabRenderer" />
     <bean name="defaultPanelFooterRenderer"
class="net.sf.springlayout.web.layout.taglib.renderer.DefaultPanelFooterRenderer" />
```

<<Add Screen Shots Here>>

Security Tags

The Security tags in Spring Layout provide simple conditional JSP processing based on the current user's access. This enables the hiding of certain parts of a page if a user does not have a particular page function or the page is in read or edit mode. These tags include:

- <layout:hasPageFunction>
- <layout:ifEditMode>
- <layout:ifReadMode>

Other Tags

Some other tags provided for different purposes include:

- <layout:swapCommandObject> - Used for switching between command objects returned by different controllers if multiple pages of a multiPanelForm is being displayed at once.
-

User Interface Behaviour

Describe things like using:

- setFieldValue to track changes in the 'commandDirty' field
- How validation works on the client side – registering events etc.
-

Overriding the Default Look and Feel

Spring Layout comes with a

Javascript Reference

Point to JS Docs if we can generate them

Detail all dependencies / current versions etc.

How to setup the layoutResources Servlet

How to set up so JS dependencies can be overridden