Cobia™ v0.3

UI Developer's Guide

October 18, 2007

(rev-a)

StillSecure
100 Superior Plaza Way
Suite 200
Superior, CO 80027
P (303) 381-3800
F (303) 381-3880
www.stillsecure.com

Table of contents

Purpose

What the Cobia™ UI Developer's Guide is

The Cobia UI Developer's Guide is a comprehensive document describing the following:

- The relationship between the user interface (UI) design and *screen* components
- The model and application programming interface (API) of screen components
- The relationship between configuration screens and the Domain model
- How to create JavaServer™ Faces (JSF) converters
- How to develop screen backing beans
- How to create JSF validation methods
- How to create client-side screen behavior
- How to integrate with AJAX in Dashboards
- Including raw HTML content in screen JSPX files

The Cobia UI Developer's Guide has been designed so that a Cobia module developer will be able to understand screen construction and code manipulation methods easily once they review the basic code structure and the cascading style sheet (CSS) that controls the layout.

For this release of the Style Guide, StillSecure® is focusing on the following:

- Explaining screen construction using the Cobia UI components
- Explaining how to connect screen components to the server-side backing beans and domain model

Who Should Use the UI Developer's Guide

This guide is intended for any developer who is looking to create the user interface for a Cobia module. You must have a strong understanding of the Java EE web tier model, specifically the JSF framework. You should also have a strong understanding of JavaScript (JS) and a moderate (but not detailed) understanding of XHTML and CSS.

How this Guide Fits in the Cobia Architecture

Figure 1: The Cobia Architecture shows a high-level view of the Cobia architecture.

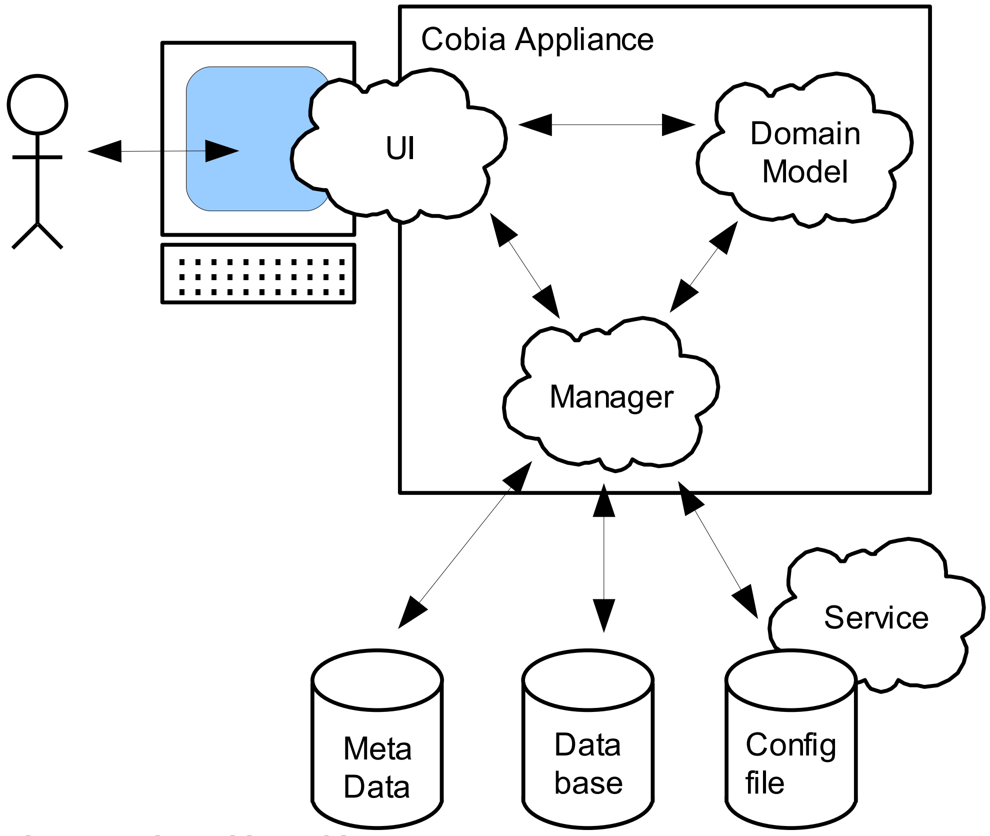


Figure 1: The Cobia Architecture

There are four major tiers to a Cobia module:

- UI
- Domain model
- Manger
- Service

The service tier is not formally part of the Cobia module; usually a service is an existing operating system (OS) service (such as dhcpd, firewall, and so on) or third-part utility (such as WiFi or Snort™). The other three tiers reside within the Java™ application that makes up your module code. However, you will note that the UI bubble extends between the Cobia appliance box and the user's desktop; this indicates that some of the functionality of the UI exists on the user's web browser, such as any JavaScript™ code required by the UI for your module.

This document focuses on the UI tier of a Cobia module. In the future there will likely be Developer's Guides for the other tiers, but for now please read the Cobia-Module-SDK document for information about the Domain model and Manager tier development.

Current Cobia documents can be downloaded from the Web site (http://cobia.stillsecure.org/?q=node/29) or you can request them by following the instructions on the Web site.

Glossary

This section describes the terms used in this document.

Term	**Definition**
screen	A screen is a conceptual unit of work for the user. One screen generates one breadcrumb.
multi-content screen	Is a unit of work with multiple content pages. Navigation between pages is performed using the tabs in the left column of a two-column layout.
page	A single conceptual view into some chunk of information.
component	A programming unit. A component might be large such as a complete screen or small such as a single UI element.
widget	A synonym for *component*, but typically refers to a single UI element such as a text field.
module	A cohesive collection of functionality that is usually associated with a single service. A module contains one or more managers.
manager	A cohesive collection of functionality that executes Cobia Jobs.
entity	A complex object that has some form of *fixed identity* that is defined by the context of the domain itself.
value object	A relatively simple object that does not have identity; that is, the equality of two value objects is solely defined by the value(s) contained in the two objects.
job	A single activity of a manager. Typical jobs include, retrieving the Domain model for a manager, setting a new model configuration, starting and stopping a service, performing searches on dynamic data of the service (such as log file entries), and so on.

Relationship with UI Design

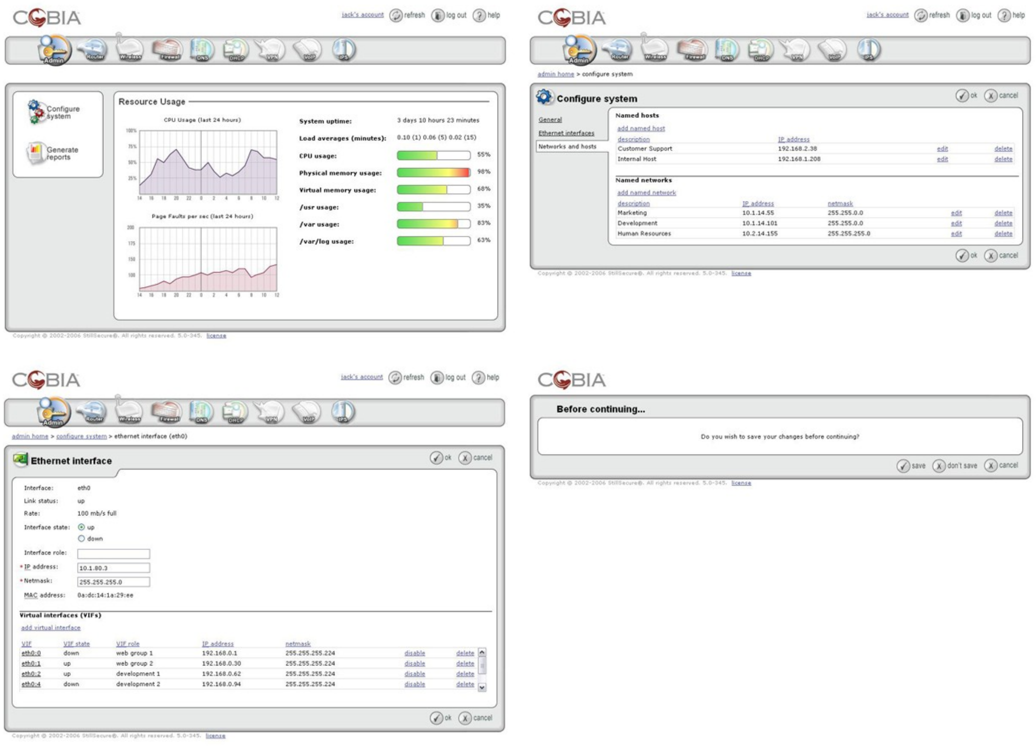
This section describes the relationship between the Cobia UI Design and UI Development. In particular, this section provides an overview of the three primary UI design screen types and how these are implemented using the Cobia UI Framework.

Design Review

As described in the Cobia_Style_Guide document, there are currently four screen types:

- Dashboard
- Two-column screen (also called a Multi-Content Screen in the UI Framework)
- Full screen
- Confirmation screen

These screen types define the templates you will use to create your module and help with your module architectural flow. Figure 2 shows the four types of screens that have been designed for Cobia.



Resource Usage

System uptime:	3 days 10 hours 23 minutes
Load averages (minutes):	0.10 (1) 0.06 (5) 0.02 (15)
CPU usage:	55%
Physical memory usage:	98%
Virtual memory usage:	68%
/usr usage:	35%
/var usage:	83%
/var/log usage:	63%

Copyright © 2002-2006 StillSecure®. All rights reserved. 5.0-345.

admin home > configure system

Configure system

Named hosts

add named host

description	IP address		
Customer Support	192.168.2.38	edit	delete
Internal Host	192.168.1.208	edit	delete

Named networks

add named network

description	IP address	netmask		
Marketing	10.1.14.55	255.255.0.0	edit	delete
Development	10.1.14.101	255.255.0.0	edit	delete
Human Resources	10.2.14.155	255.255.255.0	edit	delete

admin home > configure system > ethernet interface (eth0)

Ethernet interface

Interface:	eth0
Link status:	up
Rate:	100 mb/s full
Interface state:	[✓ up] down
Interface role:	
* IP address:	10.1.80.3
* Netmask:	255.255.255.0
MAC address:	0a:dc:14:1a:29:ee

Virtual interfaces (VIFs)

add virtual interface

VIF	VIF state	VIF role	IP address	netmask		
eth0:0	down	web group 1	192.168.0.1	255.255.255.224	disable	delete
eth0:1	up	web group 2	192.168.0.30	255.255.255.224	disable	delete
eth0:2	up	development 1	192.168.0.62	255.255.255.224	disable	delete
eth0:4	down	development 2	192.168.0.94	255.255.255.224	disable	delete

Before continuing...

Do you wish to save your changes before continuing?

Figure 2: The Four Cobia Screen Types

As a module developer, you will only create screens of the first three types. You will not be responsible for creating Confirmation screens (and other internal screens, such as the login screen); these are provided by the Cobia framework.

Module Screen Flows

The entry point to any module is the Dashboard and there most be one and only one of these screens. From the Dashboard the user can navigate to configuration, reporting, and monitoring screens.

Figure 3 provides a screen flow from the Firewall module (version 1.0).

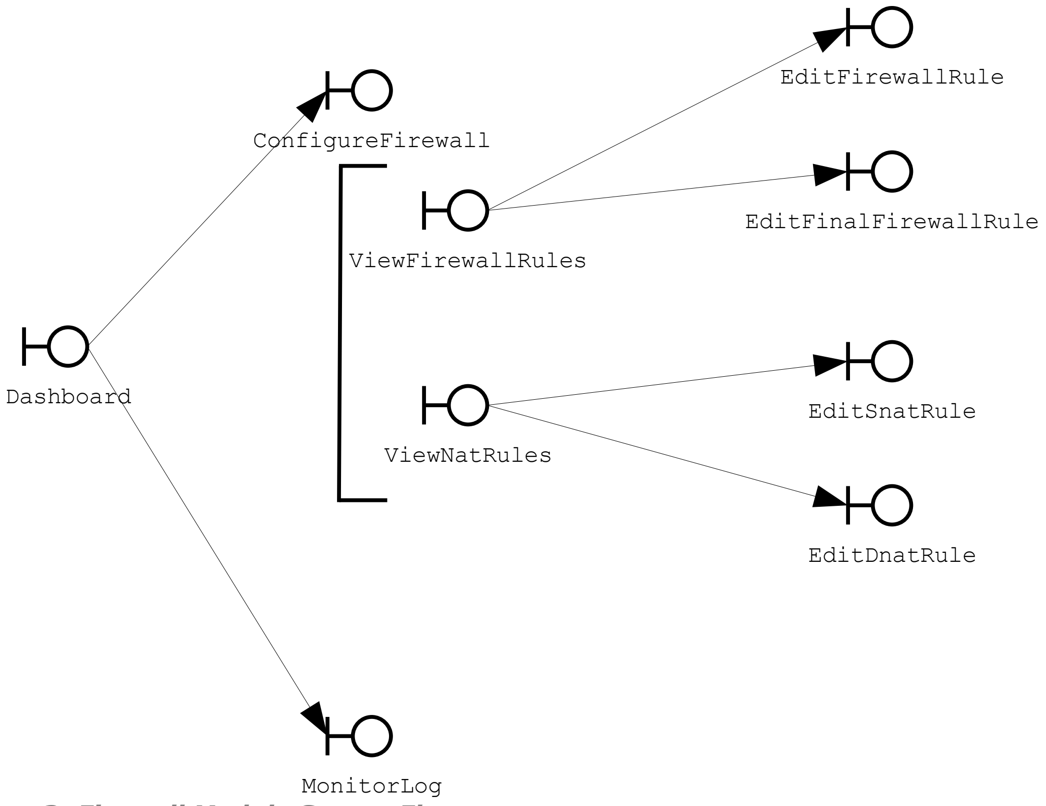


Figure 3: Firewall Module Screen Flow

From the Dashboard, the user may select to configure the firewall service or to monitor the firewall logs. The ConfigureFirewall screen is a multi-content screen which has two content pages:

- ViewFirewallRules
- ViewNatRules

From these two pages you can add or edit specific rules.

Figure 4 provides another example; this time from the Admin module.

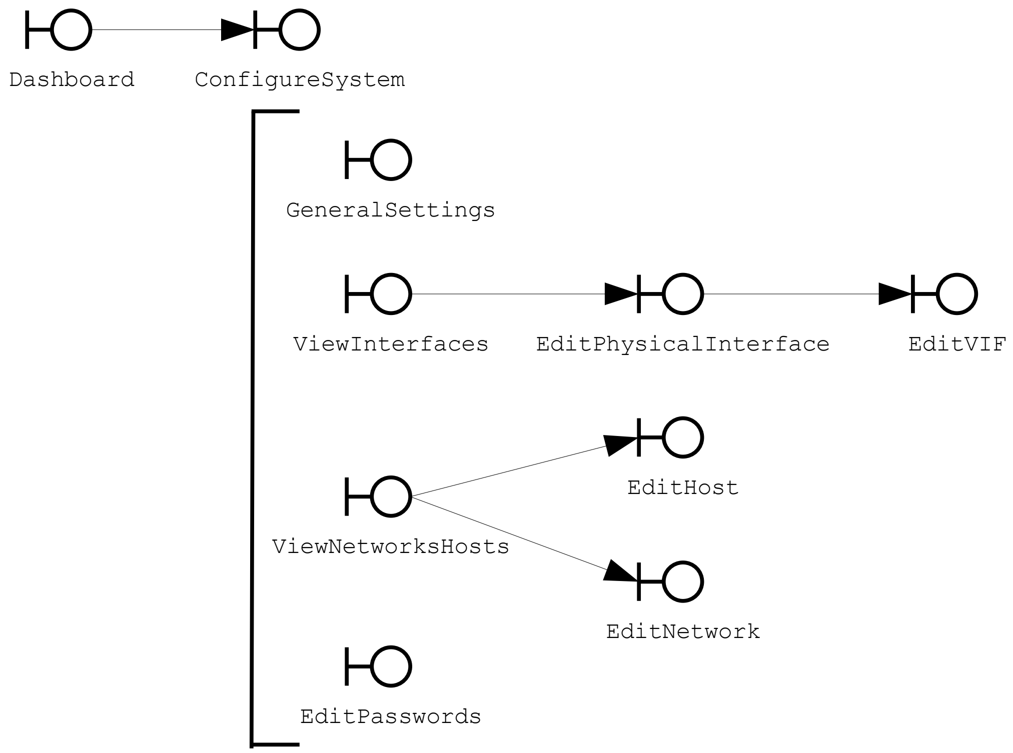


Figure 4: Admin Module Screen Flow

From the Dashboard, the user can select only the main configuration screen: ConfigureSystem. This is a multi-content screen with four content pages:

- GeneralSettings
- ViewInterfaces
- ViewNetworksHosts
- EditPasswords

From the ViewInterfaces page, the use can edit an Ethernet interface using the EditPhysicalInterface screen (a Full screen type); from this configuration screen the user can add and edit a virtual interface (VIF) using the EditVIF screen, and so on.

Screen Components (Overview)

The term *component* is overused in the software industry. In this document, I will use the term in two ways. First, complete screens (or content pages) are components. Second, individual UI elements are also called components. Where the context is not obvious I will qualify the term: either *screen component* or *UI component*. Another common term for a UI component is *widget*. Widgets are typically atomic elements; screens are never referred to as widgets.

Each Cobia screen is made up of two programming elements: a JSPX page and its JavaServer Faces (JSF) backing bean (see Figure 5).

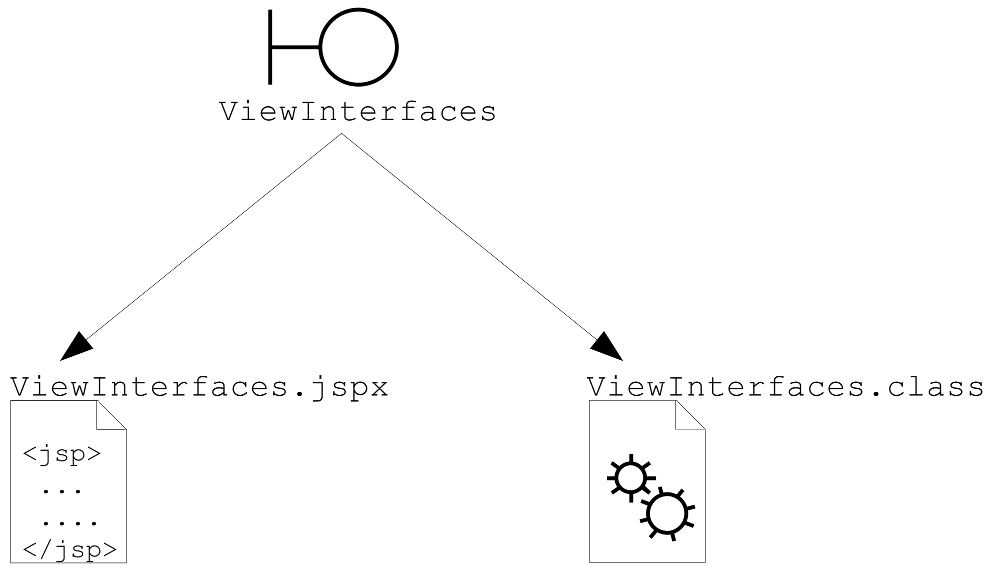


Figure 5: Screen Component Composition

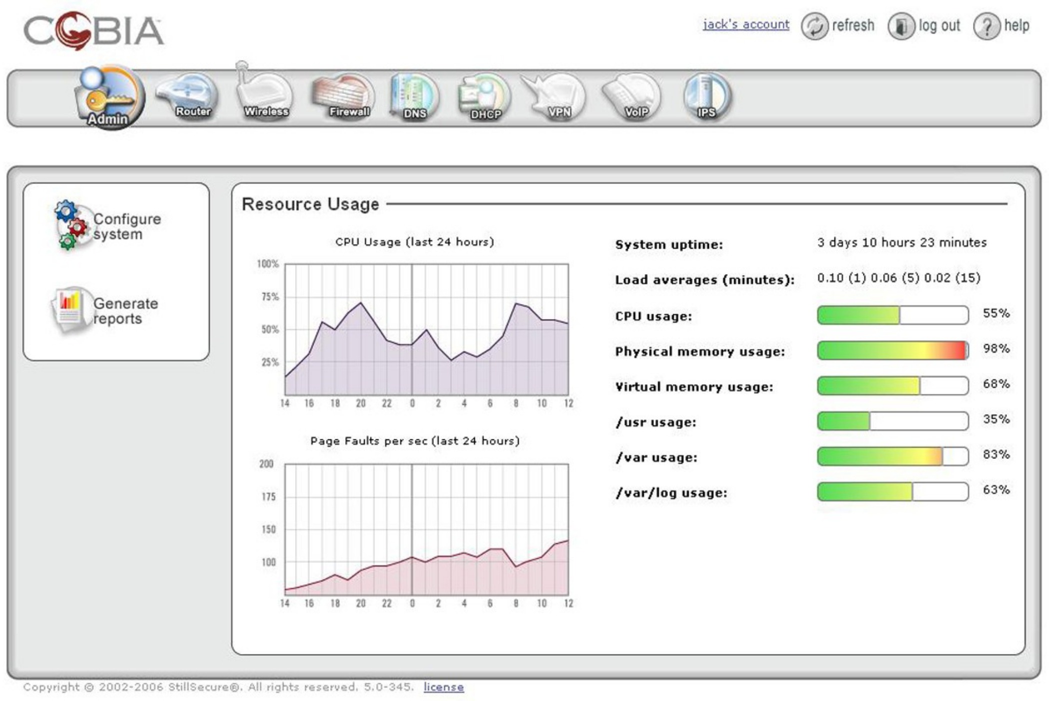
This UI Developer's Guide describes the fundamentals of building both of these elements. The JSPX is created with a set of UI components (entered as JSP tags) from the Cobia UI component library. The JSF backing bean is a Java class that implements the Cobia Screen interface; typically, you would use one of the built-in abstract classes to subclass. More on this in the following sections.

This document assumes that you have a fundamental knowledge of JavaServer Faces technology which is the basis of building Cobia screens.

Dashboard Screens (Overview)

The primary purpose of a Dashboard screen is to present the status of the module's service. This is usually done with summary data and graphs.

Figure 6 shows an example Dashboard screen from the Admin module.


jack's account
refresh
log out
help
Admin
Router
Wireless
Firewall
DNS
DHCP
VPN
VoIP
IPS
Configure system
Generate reports
Resource Usage
CPU Usage (last 24 hours)
Page Faults per sec (last 24 hours)
Copyright © 2002-2006 StillSecure®. All rights reserved. 5.0-345. license

System uptime:	3 days 10 hours 23 minutes
Load averages (minutes):	0.10 (1) 0.06 (5) 0.02 (15)
CPU usage:	55%
Physical memory usage:	98%
Virtual memory usage:	68%
/usr usage:	35%
/var usage:	83%
/var/log usage:	63%

Figure 6: Example Dashboard Screen (Admin Module)

The section frame on the left includes iconic links to the configuration screen and reports manager. The section frame on the right contains the summary information for the Admin module: usage graphs and summary data. These UI components provide dynamic data using AJAX controls to make periodic requests to the Cobia appliance to retrieve up-to-date information and graphs.

Below is a template of the JSPX structure of a typical Dashboard:

```
<jsp:root
    xmlns:jsp='http://java.sun.com/JSP/Page' version='2.0'
   xmlns:f='http://java.sun.com/jsf/core'
   xmlns:h='http://java.sun.com/jsf/html'
   xmlns:c='http://cobia.stillsecure.org/jsf'>
<jsp:directive.page contentType='text/html; charset=ISO-8859-1' />
<f:view>
<c:screen value='#{BEAN-NAME}'>
<h:form id='x'>

  <c:division id='header'>
    <c:logo />
    <c:applicationNavigation />
    <c:screenNavigation id='accessNav' />
  </c:division>

  <c:division id='container'>

    <c:screenAnchor name='modules' for='accessNav' />
```

```
20.     <c:interModuleNavigation />
21.
22.     <c:screenAnchor name='breadcrumbs' for='accessNav' />
23.     <c:breadcrumbTrail rendered='false' />
24.
25.     <c:frame>
26.
27.       <c:screenMessages />
28.
29.       <c:division id='dashboard'>
30.
31.         <c:division id='dashboardLeft'>
32.           <c:sectionFrame id='dashboardNav'>
33.             <c:screenAnchor name='submodule' for='accessNav' />
34.             <c:intraModuleNavigation id='routerNavigation'>
35.               <!-- PUT MODULE NAVIGATION COMMAND LINKS HERE -->
36.               <!-- h:commandLink action="#{routerConfig.open}"
styleClass="routerMod" value="Configure Router Protocols"/ -->
37.             </c:intraModuleNavigation>
38.           </c:sectionFrame>
39.         </c:division>
40.
41.         <c:division id='dashboardRight'>
42.           <c:sectionFrame id='dashboardContent'>
43.
44.             <c:screenAnchor name='content' for='accessNav' />
45.
46.           <!-- PUT YOUR PAGE CONTENT HERE -->
47.
48.           </c:sectionFrame>
49.         </c:division>
50.
51.       </c:division>
52.     </c:frame>
53.
54.   </c:division>
55.
56.   <c:division id='footer' addParagraph='true'>
57.     <c:copyright />
58.     <c:versionNumber />
59.     <c:license />
60.   </c:division>
61.
62. </h:form>
63. </c:screen>
64. </f:view>
65. </jsp:root>
```

Like the UI design, the JSP file includes three major divisions:

- Header (lines 11-15)
- Container (lines 17-54)
- Footer (lines 56-60)

Typically, you will only need to modify a few lines in this template. Line 8 must include the bean name for your backing bean for the Dashboard screen. This name must be the same as what you will declare in the module-faces-config.xml file; usually, I use the module's ID and the word "Home"; for example, adminHome for the Admin module, dhcpHome for the DHCP module, and so on.

The next area that you will need to modify is the IntraModuleNavigation tag (lines 34-37). Here you will insert one or more JSF CommandLink components to navigate to the second-tier screens in your module. In this example, there is a link to the ConfigureSystem screen.

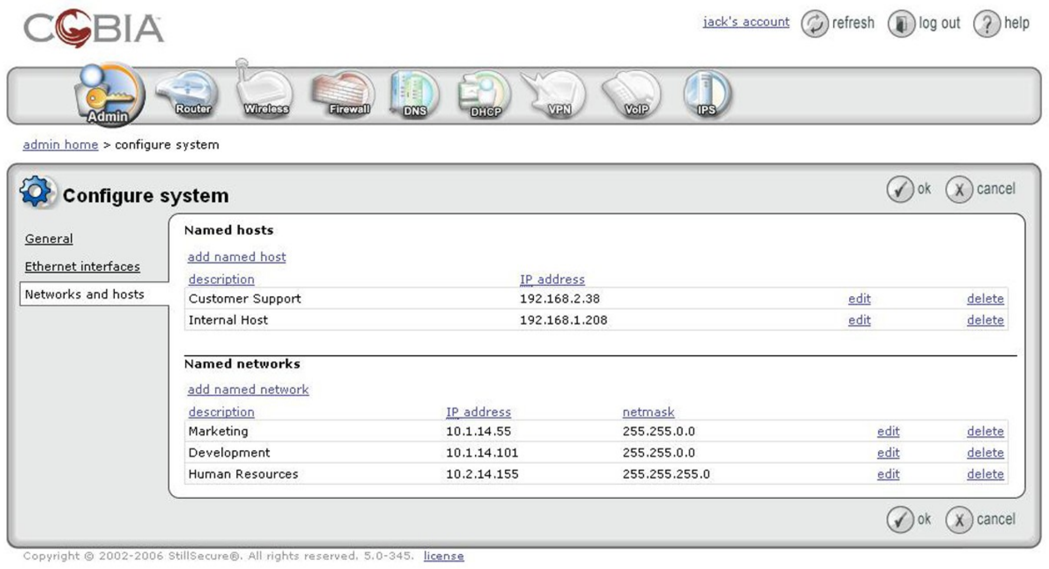
Lastly, you will place the main content of your Dashboard within the SectionFrame (lines 42-48) after the ScreenAnchor component (line 44). There are many ways to represent information on a Dashboard screen. We will talk more about this in Dashboard Screens section on page 101.

The rest of this template should be remain as-is in order to maintain the Cobia look-and-feel. This template is provided in the Cobia project under the directory: `docs/SDK/examples/ui_devel/Dashboard_TEMPLATE.jspx`.

Multi-Content Screens (Overview)

Multi-Content (also called "two column" screens in the UI design) screens have a set of navigation tabs in the left column. The typical use of this type of layout is for complex configuration screens which require multiple, large blocks of configuration data.

Figure 7 shows an example multi-content screen for the Admin module.



Named hosts

description	IP address
Customer Support	192.168.2.38
Internal Host	192.168.1.208

Named networks

description	IP address	netmask
Marketing	10.1.14.55	255.255.0.0
Development	10.1.14.101	255.255.0.0
Human Resources	10.2.14.155	255.255.255.0

Figure 7: Admin Module ConfigureSystem Screen

This screen has three content pages:

- General
- Ethernet interfaces
- Networks and hosts

The tabs on the left column allow the user to navigate from one configuration page to the next. Conceptually all of these pages are part of a single screen: ConfigureSystem. The are treated as a unit relative to the breadcrumb and validation mechanisms.

The difficult part about Multi-Content screens is that the **screen** itself does not have a JSP page; it only has a backing bean. A Multi-Content screen includes several content pages. Each content page has a JSP page and its own backing bean.

NOTE: I will repeat this again because it is confusing. The one exception to the rule that a *screen* is composed of a JSP page and a backing bean is Multi-Content screens. These only have backing beans. The Multi-Content screen's *pages* have both elements. Don't panic; I will talk about this in more detail later.

Below is a template of the JSPX structure of a typical Multi-Content screen **page**:

```
<jsp:root
    xmlns:jsp='http://java.sun.com/JSP/Page' version='2.0'
   xmlns:f='http://java.sun.com/jsf/core'
   xmlns:h='http://java.sun.com/jsf/html'
   xmlns:c='http://cobia.stillsecure.org/jsf'>
<jsp:directive.page contentType='text/html; charset=UTF-8' />
<f:view>
<c:screen value='#{BEAN-NAME}'>
<h:form id='x'>

  <c:division id='header'>
    <c:logo />
    <c:applicationNavigation />
    <c:screenNavigation id='accessNav' />
  </c:division>

  <c:division id='container'>

    <c:screenAnchor name='modules' for='accessNav' />
    <c:interModuleNavigation />

    <c:screenAnchor name='breadcrumbs' for='accessNav' />
    <c:breadcrumbTrail />

    <c:frame>

      <c:screenMessages />

      <c:screenButtons location='TOP' />

      <c:division id='twoColumnLayout'>

        <c:division id='twoColumnLeft'>
          <c:screenAnchor name='contentNav' for='accessNav' />
          <c:contentNavigation />
        </c:division>

        <c:division id='twoColumnRight'>

          <c:sectionFrame id='twoColumnContent'>

            <c:screenAnchor name='content' for='accessNav' />

            <!-- PUT YOUR PAGE CONTENT HERE -->

          </c:sectionFrame>

          <c:screenButtons location='BOTTOM' />

        </c:division>
      </c:division>

    </c:frame>
  </c:division>

  <c:division id='footer' addParagraph='true'>
    <c:copyright />
    <c:versionNumber />
    <c:license />
  </c:division>

</h:form>
</c:screen>
</f:view>
```

```
65. </jsp:root>
```

Like the UI design, the JSP file include three major divisions:

- Header (lines 11-15)
- Container (lines 17-54)
- Footer (lines 56-60)

Typically, you will only need to modify a few lines in this template. Line 8 must include the bean name for you backing bean for the Multi-Content screen; *not* for the page. The specific page will also have a backing bean and it will be used in the content area of the page. I will talk more about this in the Multi-Content Screen section.

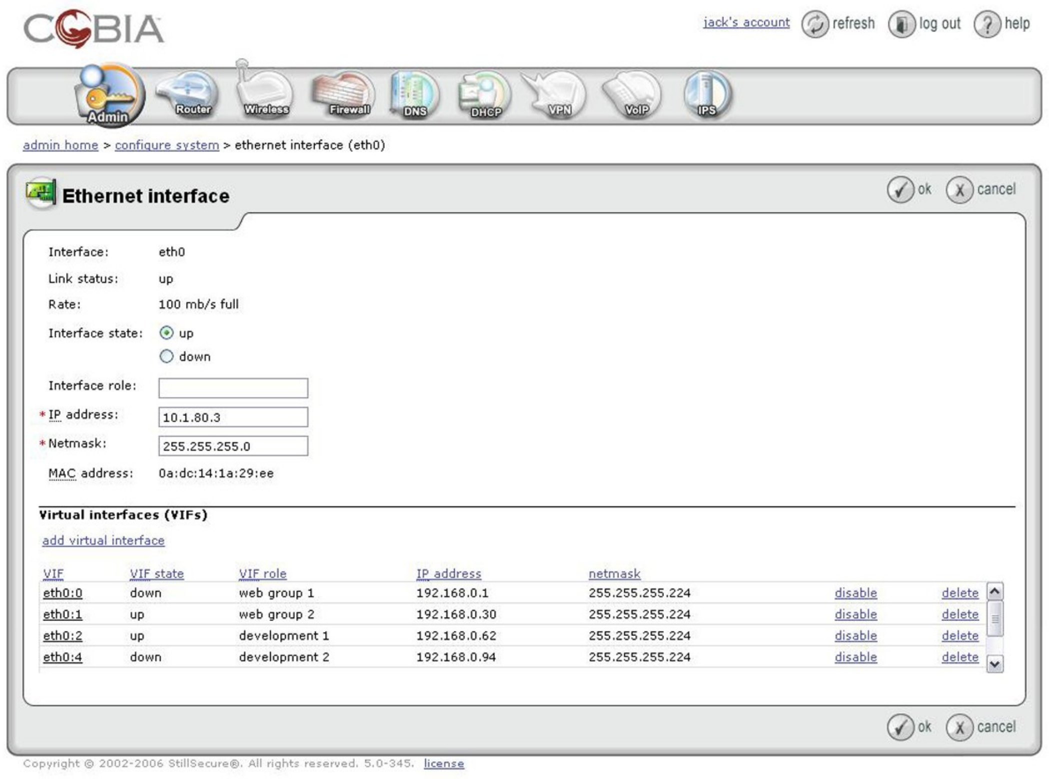
The only other area of this JSP page you will need to modify is the main content area (line 44). This is where you will place the form widgets for this page's configuration. We will talk more about this in Multi-Content Screens section on page 42.

The rest of this template should be remain as-is in order to maintain the Cobia look-and-feel. This template is provided in the Cobia project under the directory: `docs/SDK/examples/ui_devel/MultiContentScreen_TEMPLATE.jspx`.

Fullscreen Screens (Overview)

Fullscreen screens do not include any tab navigation; the content area takes up the full screen width; hence the name. Fullscreens are primarily used as configuration screens; although they are also used for search screens, such as the Firewall's MonitorLog screen.

Figure 8 shows an example Fullscreen from the Admin module.



admin home > configure system > ethernet interface (eth0)

Ethernet interface

Interface:	eth0
Link status:	up
Rate:	100 mb/s full
Interface state:	[✓ up] down
Interface role:	
* IP address:	10.1.80.3
* Netmask:	255.255.255.0
MAC address:	0a:dc:14:1a:29:ee

Virtual interfaces (VIFs)

add virtual interface

VIF	VIF state	VIF role	IP address	netmask		
eth0:0	down	web group 1	192.168.0.1	255.255.255.224	disable	delete
eth0:1	up	web group 2	192.168.0.30	255.255.255.224	disable	delete
eth0:2	up	development 1	192.168.0.62	255.255.255.224	disable	delete
eth0:4	down	development 2	192.168.0.94	255.255.255.224	disable	delete

Copyright © 2002-2006 StillSecure®. All rights reserved. 5.0-345. license

Figure 8: Admin Module Fullscreen: EditPhysicalInterface

Below is a template of the JSPX structure of a typical Fullscreen screen type:

```
1.  <jsp:root
2.      xmlns:jsp='http://java.sun.com/JSP/Page' version='2.0'
3.     xmlns:f='http://java.sun.com/jsf/core'
4.     xmlns:h='http://java.sun.com/jsf/html'
5.     xmlns:c='http://cobia.stillsecure.org/jsf'>
6.  <jsp:directive.page contentType='text/html; charset=UTF-8' />
7.  <f:view>
8.  <c:screen value='#{BEAN-NAME}'>
9.  <h:form id='x'>
10.
11.   <c:division id='header'>
12.     <c:logo />
13.     <c:applicationNavigation />
14.     <c:screenNavigation id='accessNav' />
15.   </c:division>
16.
17.   <c:division id='container'>
18.
19.     <c:screenAnchor name='modules' for='accessNav' />
20.     <c:intraModuleNavigation />
21.
22.     <c:screenAnchor name='breadcrumbs' for='accessNav' />
23.     <c:breadcrumbTrail />
24.
25.     <c:frame>
26.
27.       <c:screenMessages />
28.
29.       <c:screenButtons location='TOP' />
30.
31.       <c:fullscreenFrame id='DIV_NAME'>
32.       <!-- The DIV_NAME is important as screen-specific styles usually
    depend on it. -->
33.
34.         <c:screenAnchor name='content' for='accessNav' />
35.
36.         <!-- PUT YOUR PAGE CONTENT HERE -->
37.
38.       </c:fullscreenFrame>
39.
40.       <c:screenButtons location='BOTTOM' />
41.
42.     </c:frame>
43.   </c:division>
44.
45.   <c:division id='footer' addParagraph='true'>
46.     <c:copyright />
47.     <c:versionNumber />
48.     <c:license />
49.   </c:division>
50.
51. </h:form>
52. </c:screen>
53. </f:view>
54. </jsp:root>
```

Like the UI design, the JSP file includes three major divisions:

- Header (lines 11-15)
- Container (lines 17-43)
- Footer (lines 45-49)

Typically, you will only need to modify a few lines in this template. Line 8 must include the bean name for you backing bean for this screen.

You will also have to provide an ID for the FullscreenFrame UI widget on line 31. I usually use the name of the screen with a lower case letter. For example, if the screen is called EditHost, then I use `editHost` in the `id` attribute. This is used when you create the CSS styles for this page. More on this latter.

The only other area of this JSP page you will need to modify is the main content area (line 44). This is where you place the form widgets for this page's configuration. We will talk more about this in Configuration Pages section on page 47.

The rest of this template should be remain as-is in order to maintain the Cobia look-and-feel. This template is provided in the Cobia project under the directory: `docs/SDK/examples/ui_devel/FullScreen_TEMPLATE.jspx`.

Module Icons

One of the first things you must do when build the UI for a module is the create the module icon set. Figure 9 shows an example icon set from the Admin module.

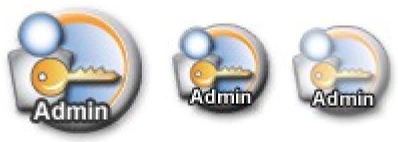


Figure 9: Icon Set

The Icon Set includes:

- A large icon to indicate that the module is selected and being viewed. (`iconLgCo_`*`<moduleID>`*`.gif`)
- A small faded icon to show a module that can be selected from the list. (`iconSmGy_`*`<moduleID>`*`.gif`)
- A small full color icon to show the module is being selected when hovered over. (`iconSmCo_`*`<moduleID>`*`.gif`)

Where:

`<moduleID>` is the ID of the module you are creating. For example, the Admin module uses ID of `admin`; so the large icon would be called: `iconLgCo_admin.gif`.

For more details about how to create module icons read the Cobia_Style_Guide document.

After creation, these icon files must be placed in the images directory of the web directory, for example: `cobia/modules/admin/web/images/` directory for the Admin module.

Next you must define the CSS styles that allow the InterModuleNavigation UI component to locate these icons. Create the file `moduleNavStyle.css` file in your module's `web/styles/` directory and include these three style definitions:

```
div#moduleNavigation ul li a.admin {
    background:url(/admin/images/iconSmGy_admin.gif) top left no-repeat;
    margin:0;
    padding: 85px 0 0 0;
    width: 55px;
}
div#moduleNavigation ul li a.admin:hover {
    background:url(/admin/images/iconSmCo_admin.gif) top left no-repeat;
    width: 55px;
}
div#moduleNavigation ul li a.adminSelect {
    background:url(/admin/images/iconLgCo_admin.gif) top left no-repeat;
    margin: 0 5px 0 0;
    padding: 85px 0 0 0;
    width: 71px;
}
```

Replace the word 'admin' with your module's ID. You might also need to adjust the width and padding attributes if your icons are larger than standard Cobia icons.

Planning Screens

This section provides a brief discussion of how to plan the screen flow for your module. This will be brief because we cannot provide detailed advice on what is, because by its very nature it is domain specific. However, there are a few general guidelines that have been created by the UI design team which describes how UI development should be done.

Basic UI Design Guidelines

The dashboard should present summary statistics about the service (or services) provided by your module. Let us call this the first tier screen.

The second tier screens should focus on high-level operation of your module's service. Typically this means: configuration, monitoring, and reporting. Links to these screens should be made available on the dashboard in the left-hand column of links; the IntraModuleNavigation UI widget. Figure 10 shows an example of the buttons/links in the Firewall dashboard.

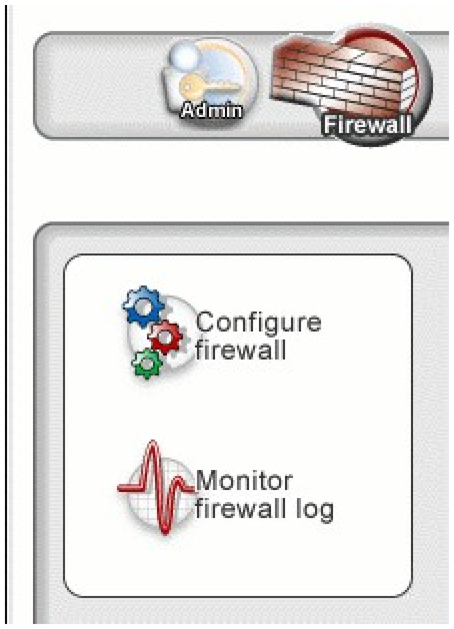


Figure 10: Second Tier Buttons on the Firewall Dashboard

The configuration screens should follow the Domain model from the root of the model toward the leaf entities of the model. Each type of entity should have its own screen (or content page) to configure that specific entity.

Example: Firewall Module

We will begin with a relatively simple domain: Firewall. Figure 11 shows the domain model for the Firewall module. The FirewallState entity is composed of zero or more FirewallRule entities, zero or more SnatRule entities, and zero or more DnatRule entities. These rule entity classes inherit from abstract classes: Rule and NatRule. However, this inheritance is purely a convenience for the Domain model developer and is not reflected in the UI. Also notice that the FirewallState includes an additional link for the *final* firewall rule; there is always one of these.

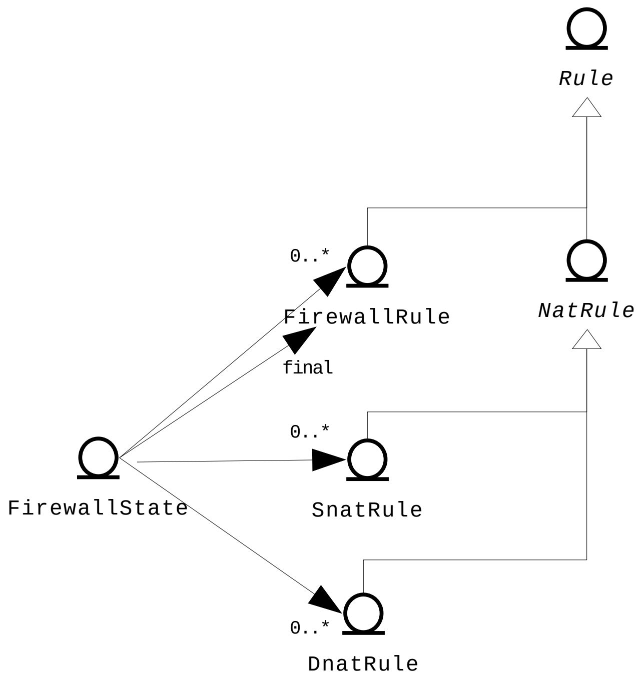


Figure 11: Firewall Domain Model

Figure 12 shows the screen flow for the Firewall module.

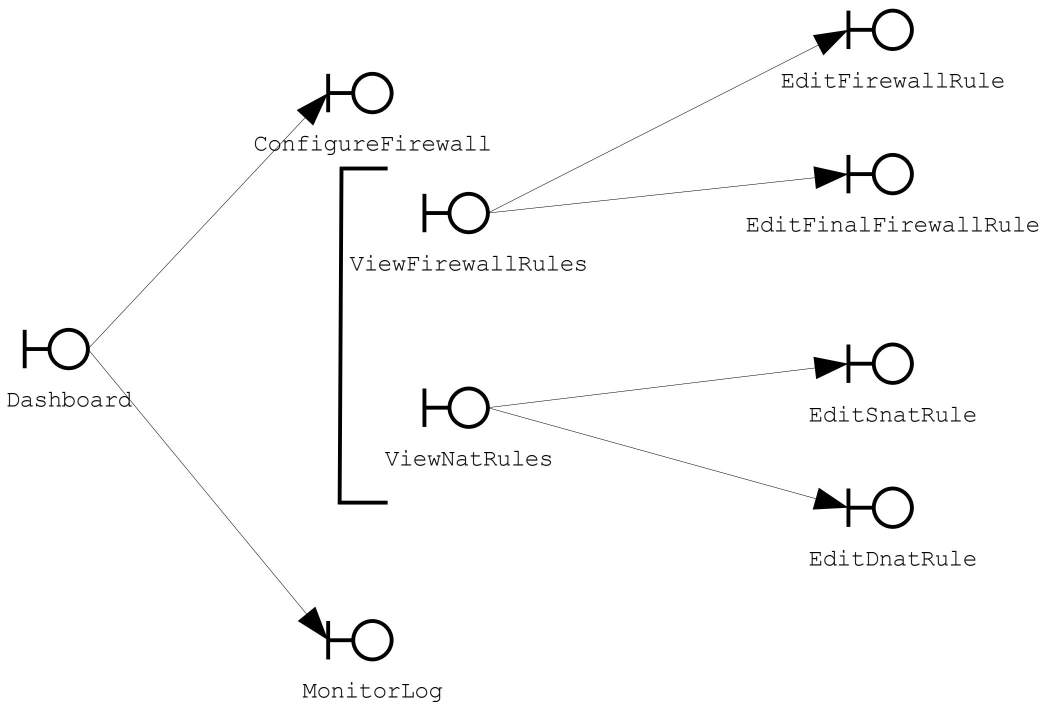


Figure 12: Firewall Screen Flow

From the Dashboard the user navigates to the ConfigureFirewall Mulit-Content screen which includes two content pages: ViewFirewallRules and ViewNatRules. From the ViewFirewallRules page the user can add or edit a user-defined FirewallRule entity; both the add and edit operations are handled by the same screen definition because there are no concrete differences between the UI widgets used in the add or edit operation. There is also a link to edit the final firewall rule; this does require a new screen because the set of operations (properties to change) are much more limited than in the user-defined rules.

From the ViewNatRules page, the user may add/edit either an SNAT rule or a DNAT rule. These have unique screens because there are differences in the type of data that can be configured in their properties; specifically, SNAT rules allow masquerading but DNAT rules do not.

Example: DHCP Module

Now let us look at a more complex module. Figure 13 shows the domain model for the DHCP module. The DhcpServer entity class is the root of the model. It contains zero or more Scope entities and it contains an instance of the OptionsMap which is a complex Map-based data structure to hold server options (also called parameters). A Scope entity contains zero or more DynamicPool and ReservationPool entities; it also contains a collection of Lease objects which is for viewing only (not configuration). Both pool entities inherit from an abstract Pool entity class. This class also contains a single instance of the OptionsMap data structure. This Cobia data structure is based upon the InheritableMap interface which allows one InheritableMap to inherit mapped values from a parent. In the current DHCP model, pool options inherit from server options. Pool options can override server options and define unique local options. Lastly, the ReservationPool entity contains zero or more HostReservation entities.

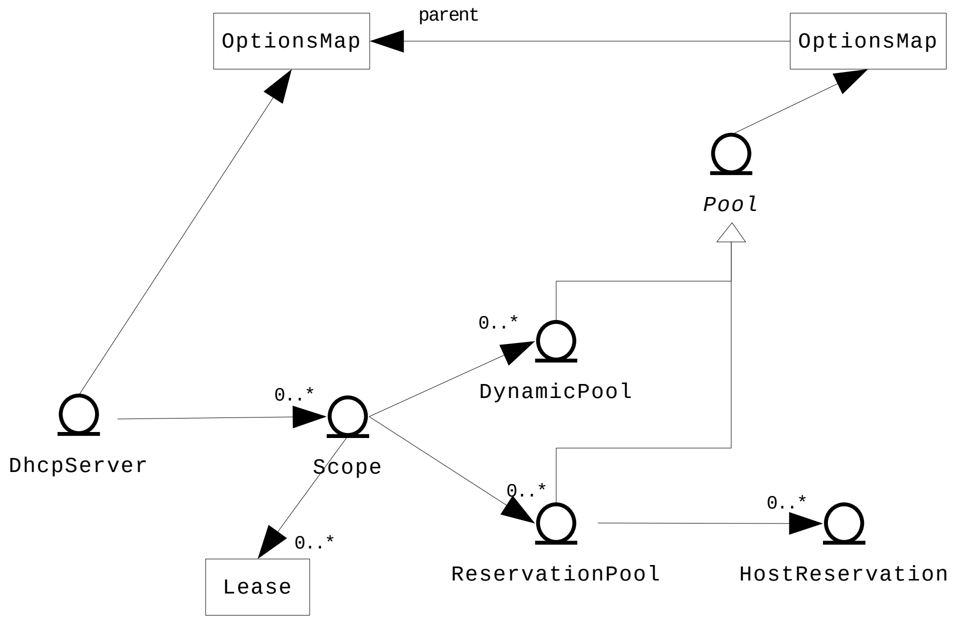


Figure 13: DHCP Domain Model

Figure 14 shows the DHCP module screen flow. Notice the complex use of Multi-Content screens. The ConfigureDHCP screen contains two pages: the ConfigureServerSettings page which allows the user to start and stop the DHCP service as well as add and edit scopes and the ConfigureServerOptions page which allows the user to edit the server options.

The ConfigureScope screen has two pages: the EditScope page allows the user to add and edit pools (both dynamic and reservation) and the ViewLeasse page allows the user to see the current set of leases issued to DHCP clients.

The ConfigurePool screen has two pages: the Edit*Xyz*Pool page allows the user to manipulate the IP ranges for the pool and the ConfigurePoolOptions page allows the user to edit the pool options. There are actually four variations: AddDynamicPool, EditDynamicPool, AddReservationPool, and EditReservationPool. The last two screens allow the user to add and edit HostReservation entities.

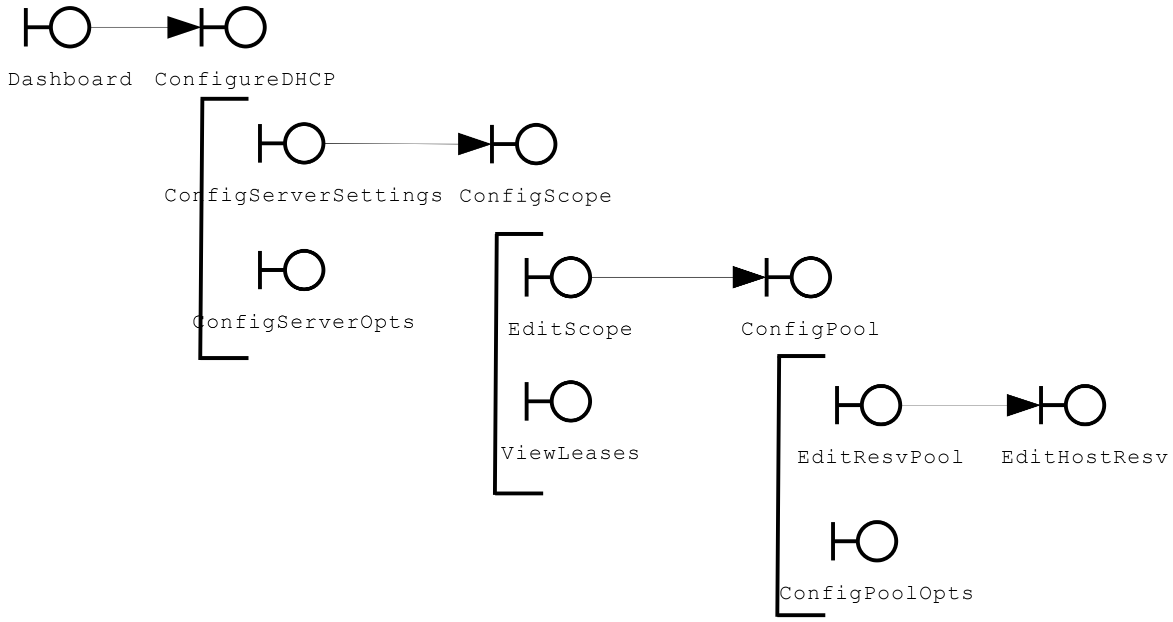


Figure 14: The DHCP Screen Flow

Screen design is as fluid as Domain modeling and your module will likely have different needs then what is shown here. However, the principles are basically the same:

- Each domain entity should have it's own page or screen.
- Start at the root of the model and work towards the leafs of the model.
- Complex entities should be split into multiple content pages.
- Complex data structures should have their own content page.

UI Development Steps

It is beyond the scope of this document to define a development process; however, the following sketch is based on my experience building the first four modules in Cobia:

1. Create a sketch of the Dashboard

   This is critical because the dashboard is the first screen that the user visits when you navigate to a module. Do not attempt to add content such as graphs or charts to this screen because usually you will not have that data in the Domain model yet.
2. Create the second tier screen for the configuration of your service. Add the link from the dashboard to this screen.
3. Iteratively add screens and pages as you (or your team) develop the Domain model.
4. After the configuration screens are complete, then create any monitoring screens.
5. Finally, complete the Dashboard by adding graphs and charts.

Of course this is just a suggestion, but this order usually makes sense because until you can configure your backend service it is usually not possible to generate the data necessary to populate the search (monitor) screens and the dashboard's graphs.

Screen Model and API

This section provides details about the screen programming model and the APIs of the various screen types and modes. This is your starting point for creating Cobia screens. There is a lot of information to absorb; do not feel that you have to understand all of this immediately. It will take building several different types of screens before you get all of this. *Don't panic!*

Screen Components

As described in the Screen Components (Overview) section on page 10, a screen component is composed of two elements: a JSPX file and a backing bean (a Java object). Figure 15 illustrates the connection between these two elements.

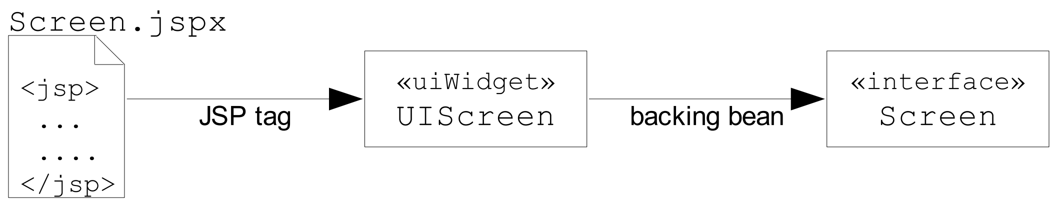


Figure 15: Generic Elements of a Screen Component

The JSPX file includes a `<c:screen>` tag which creates a UIScreen object. This UI widget include a JSF value binding which refers to the backing bean. This backing bean *must* be an object that implements the Screen interface.

Figure 16 provides a concrete example from the Admin module.

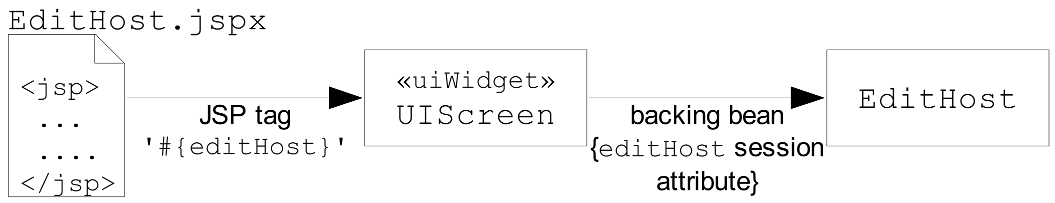


Figure 16: The EditHost Screen Component Elements

The EditHost.jspx file includes a `<c:screen value='#{editHost}'>` tag. The value binding `#{editHost}` refers to a session-scoped attribute called `editHost`. This session attribute holds the EditHost object. The EditHost class implements the Screen interface. In fact, it extends the AbstractUpdateScreen class and we will see this class in the The Update Screen API section on page 36.

Screen Types

The Screen interface includes a method to specify the layout type of the screen. Figure 17 shows the API for this.

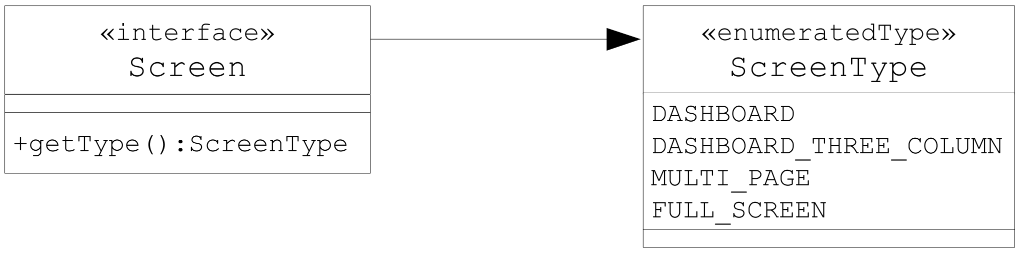


Figure 17: Screen Type API

The ScreenType enum includes values for the three primary UI design layout types:

- Dashboard
- Mulit-Content
- Full screen.

The three-column Dashboard is included for a new type of layout for Dashboards which is still being developed and is not described in this version of the UI Developer's Guide.

Screen Rendering Process

The Screen UI component handles the basic HTML rendering. Here is the basic structure of all Cobia screens:

```
1.  <jsp:root
2.      xmlns:jsp='http://java.sun.com/JSP/Page' version='2.0'
3.     xmlns:f='http://java.sun.com/jsf/core'
4.     xmlns:h='http://java.sun.com/jsf/html'
5.     xmlns:c='http://cobia.stillsecure.org/jsf'
6.     xmlns:ctags='http://cobia.stillsecure.org/tags'>
7.  <jsp:directive.page contentType='text/html; charset=ISO-8859-1' />
8.  <f:view>
9.  <c:screen value='#{BEAN-NAME}'>
10. <ctags:loadScript src='PATH-TO-JAVASCRIPT-FILE' />
11. <ctags:loadStyle src='PATH-TO-CSS-FILE' />
12. <h:form id='x'>
13.   <!-- SCREEN CONTENT HERE -->
14. </h:form>
15. </c:screen>
16. </f:view>
17. </jsp:root>
```

The `jsp:root` tag (line 1-6) declares the JSP file is a JSP Document; the XML version of a JSP file. This is where you usually will declare all of the tag libraries that you will use in this JSP page. The `jsp:directive.page` tag (line 7) declares the content type of the page to be HTML. The `f:view` tag (line 8) declares that this JSP is a JSF page. The `c:screen` tag (line 9) declares the Cobia screen which creates a Screen component in the JSP component hierarchy for the page. The `ctags:loadScript` tag (line 10) allows you to declare a JavaScript file to include in the screen. This Cobia custom tag (part of the ctags tag library) may be used as many times as you like, but must exist inside of the screen tag. The `ctags:loadStyle` tag (line 11) allows you to declare a CSS file to include in the screen. This Cobia custom tag may be used as many times as you like, but must exist inside of the screen tag. Finally, the `h:form` tag (line 12) declares a JSF form component. The rest of the page is represented abstractly by the XML comment on line 13.

The Screen component renders the `html`, `head`, `title`, and `body` tags. It also renders the `script` and `link` tags within the head to include the JavaScript and CSS files required by the screen. Here is the basic structure of the HTML content rendered by the JSP structure listed above:

```
<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.0 Transitional//EN\" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd\">
<html xmlns="http://www.w3.org/1999/xhtml">

<head>
  <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
  <title>SCREEN-TITLE - PRODUCT-NAME</title>
  <script language="javascript" type="text/javascript" src="/common/scripts/prototype.js"></script>
  <script language="javascript" type="text/javascript" src="/common/scripts/json.js"></script>
  <script language="javascript" type="text/javascript" src="/common/scripts/firebugx.js"></script>
  <script language="javascript" type="text/javascript" src="/common/scripts/helper_functions.js"></script>
  <script language="javascript" type="text/javascript" src="/common/scripts/addDomLoadEvent.js"></script>
  <script language="javascript" type="text/javascript" src="/common/scripts/browser_sniffer.js"></script>
  <script language="javascript" type="text/javascript" src="/common/scripts/cookies.js"></script>
  <script language="javascript" type="text/javascript" src="/common/scripts/navigation.js"></script>
  <script language="javascript" type="text/javascript" src="/common/scripts/css.js"></script>
  <script language="javascript" type="text/javascript" src="/common/scripts/object.js"></script>
  <script language="javascript" type="text/javascript" src="/common/scripts/enable_disable.js"></script>
  <script language="javascript" type="text/javascript" src="/common/scripts/util.js"></script>
  <script language="javascript" type="text/javascript" src="/common/scripts/select.js"></script>
  <!-- JAVASCRIPT FILES FROM <loadScript> TAG INCLUDED HERE -->
  <!-- COMPONENT JAVASCRIPT FILES INCLUDED HERE -->
  <!-- SCREEN-TYPE JAVASCRIPT FILE INCLUDED HERE -->
  <!-- SCREEN JAVASCRIPT FILE INCLUDED HERE -->
  <link rel="stylesheet" type="text/css" href="/common/styles/common.css" media="screen">
  <link rel="stylesheet" type="text/css" href="/common/styles/forms.css" media="screen">
  <link rel="stylesheet" type="text/css" href="/components/styles/label.css" media="screen">
  <!-- SS FILES FROM <loadStyle> TAG INCLUDED HERE -->
  <!-- COMPONENT CSS FILES INCLUDED HERE -->
  <!-- SCREEN-TYPE CSS FILE INCLUDED HERE -->
  <!-- MODULE CSS FILE INCLUDED HERE -->
</head>

<body id="Dashboard">
<form target="" id="x" name="x" method="post" action="/admin/Dashboard.jsf" enctype="application/x-www-form-urlencoded">
  <!-- SCREEN CONTENT HERE -->
</form>
</body>

</html>
```

The text in RED is dynamic content. The `title` tag (line 5) outputs the title of the screen, followed by a dash, and then the name of the product, currently Cobia; however, this will be configurable in the future.

The `script` tags in lines 6-18 are all of the standard JavaScript files that are always available to all Cobia screens. This is followed by all of the JavaScript files specified by the `loadScript` tags. This is followed by all of the JavaScript files required by the Cobia components used in the page. This is followed by the JavaScript file used by the screen type; currently the only screen type with a JS file is the FULL_SCREEN type. This is followed by the JS file for *this* screen with the path: `/<moduleID>/scripts/SCREEN_ID.js`; for example, the Admin EditPassword screen would by: `/admin/scripts/EditPassword.js`.

The `link` tags in lines 23-25 are all of the standard CSS files that are always available to all Cobia screens. This is followed by all of the CSS files specified by the `loadStyle` tags. This is followed by all of the CSS files required by the Cobia components used in the page. This is followed by the CSS file used by the screen type. This is followed by the CSS file for *this* module with the path: `/<moduleID>/styles/<moduleID>.css`; for example, the Admin module would by: `/admin/styles/admin.css`. This means that the styles for all screens within a module must be placed within this file.

Lastly, the `h:form` tag renders to an HTML form tag (line 32).

Note: The above description is not the full story. Also included in the rendering of the `h:form` tag includes some additional content used by the AJAX JSF library we include with Cobia: RichFaces. For the most part, you can ignore these details.

Screen Modes

The screen type defines the layout structure, but not the purpose or *mode* of the screen. Figure 18 illustrates the interface relationships of the screen modes.

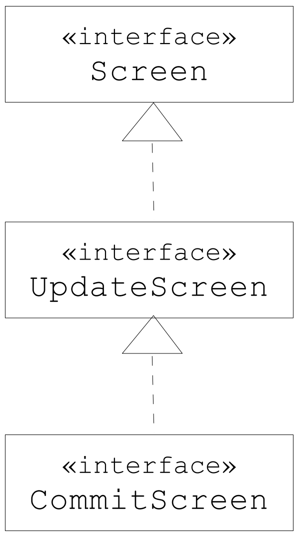


Figure 18: The Three Screen Mode Interfaces

Currently there are only two fully defined screen modes:

- View-based
- Configuration-based.

The Screen interface is responsible for basic view behavior. The UpdateScreen interface extends the Screen interface and provides configuration-based behaviors; this is also called the "update" mode because you can update or change the configuration of an entity in your Domain model.

There is another screen mode "commit" that is reserved for future use.

The Screen API

Figure 19 provides a UML diagram of the fundamental Screen interface and the AbstractScreen class that provides a base implementation of that interface. Typically, you will never have to override the methods provided in the base class.

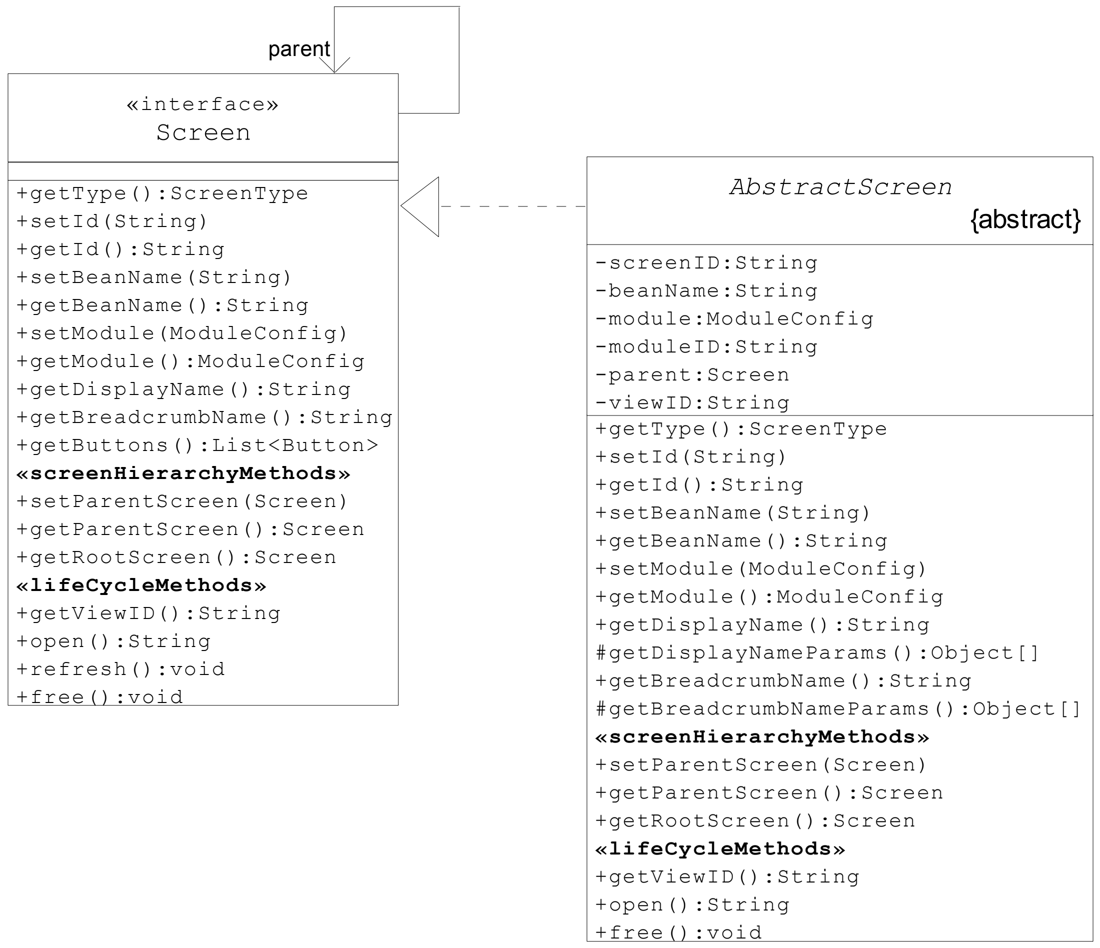


Figure 19: The Screen Interface and Abstract Base Class

The Screen interface is the center of the Cobia UI Framework.

Basic Information

The `getType` method provides the type of layout. The base class returns `FULL_SCREEN` from this method, but may be overriden by other subclasses.

The screen ID, `getId`/`setId`, must be the name of the screen and must match both the JSPX file name and the backing bean's file name. For example, the EditHost screen ID matches the EditHost.jspx and the EditHost.java file names. The base class provides this object attribute, `screenID`, and the setter and getter methods. This attribute may only be set once; an assertion error is thrown if the application attempts to set it twice.

The bean name, `getBeanName`/`setBeanName`, must be the name of the session-scoped attribute that holds the backing bean object. This name is at the discretion of the UI developer, but it is best if it has a name that is reminiscent of the screen ID. For example, the EditHost screen has the bean name of `editHost`. The base class provides this object attribute, `beanName`, and the setter and getter methods. This attribute may only be set once; an assertion error is thrown if the application attempts to set it twice.

The module that this screen belongs to is stored in the screen object. The base class provides this object attribute, `module`, and the setter and getter methods. This attribute may only be set once; an assertion error is thrown if the application attempts to set it twice.

The `getDisplayName` method provides the title of the screen. The base provides an implementation that looks the screen title in the module's text.properties file using the following key template:

```
<moduleID>.<screenID>.Title
```

For example, the key for the Admin module's EditHost screen would be:

```
admin.EditHost.Title
```

Some screen titles may choose to augment the title with domain-specific information. For example, the EditHost screen might display the title “Edit Host (Fappy)”. To accomplish this the title text property would use a message format structure like this:

```
admin.EditHost.Title=Edit Host ({0})
```

The {0} part of the string is replaced with a parameter passed to the Message object. This parameter is provided by the protected helper method called, `getDisplayNameParameters`, which returns an Object array. Here is what the EditHost class method might look like:

```
class EditHost extends AbstractScreen {
...
  protected Object[] getDisplayNameParameters() {
    return new Object[] { host.getDescription() };
  }
...
}
```

The `getBreadcrumbName` method provides the name of the breadcrumb for this screen. It is usually the same as the display name by in all lowercase.

The `getButtons` method provides a list of `Button` objects which are meant to display in the main content section of the screen. Not all screens need this so the base class does not provide an implementation of the `getButtons` method. This is left to subclasses.

Screen Hierarchy Methods

Every screen keeps a reference back to its parent screen. Every screen must have a parent except for Dashboards which are the root of the screen hierarchy for every module. The `setParent`/`getParent` methods manage this information.

The `getRootParent` method returns the root screen in the hierarchy. This must always be the Dashboard of the screen's module.

Screen Life Cycle Methods

The screen life cycle methods provide the connection to the JSF framework that the Cobia UI Framework is built on top of.

The `getViewID` method returns the String which represents the JSF navigation outcome name for this screen. The base implementation of this method creates an ID using the following template: `<moduleID>.<screenID>`; for example, the Admin module's EditHost screen has the view ID of `admin.EditHost`. These symbolic names are used in the JSF configuration file to match the screen with the JSPX file that implements the view for that screen. Here is a snippet from the Admin module's `web/WEB-INF/module-faces-config.xml` file:

```
<navigation-rule>
   <from-view-id>*</from-view-id>
   <navigation-case>
      <from-outcome>admin.EditHost</from-outcome>
      <to-view-id>/admin/EditHost.jspx</to-view-id>
   </navigation-case>
</navigation-rule>
```

The `open` method is a JSF action method that instructs the JSF framework to display this screen. The default implementation simply returns the viewID for this screen. This method may be called several times during the life of the screen. If you need to create any resources for this screen you should probably do it in the screen's constructor.

The `free` method tells this screen object that it is no longer in the breadcrumb trail. The default implementation removes the bean name from the session scope. The `free` method may also be used to release any resources used by this screen. If you override this method make sure that you call the `super.free()` method to free the bean from the session scope.

The `refresh` method is called with the Refresh button in the application-wide menu in the header division of the screen is clicked by the user. The purpose of this action is to refresh the data on the screen or in the case of a configuration screen to reset the data back to its original state. This method is application-specific and thus the base class does not provide an implementation of this method.

The Dashboard API

Figure 20 provides a UML diagram of the API for the Dashboard base class. All Cobia dashboards should extend the `AbstractDashboard` class.

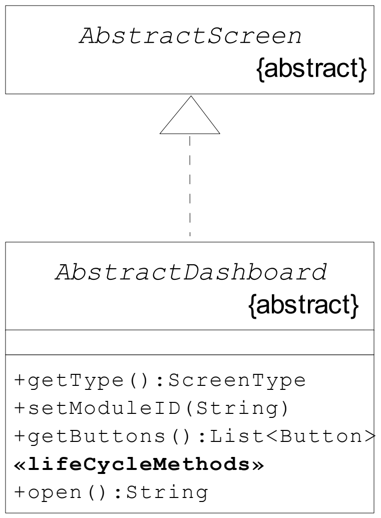


Figure 20: The Dashboard API

The `AbstractDashboard` class inherits all of the base methods from the `AbstractScreen` class.

The `AbstractDashboard` class overrides the `getType` method and returns `DASHBOARD`.

The `AbstractDashboard` class adds the `setModuleID` method and allows the module configuration to be assigned using the module's string-based ID. Dashboard backing beans are created using the JSF managed beans facility and the module ID is declared in the faces configuration file. Here is a snippet from the Admin module's `web/WEB-INF/module-faces-config.xml` file:

```
<faces-config>
   <managed-bean>
      <managed-bean-name>adminHome</managed-bean-name>
      <managed-bean-class>
         org.stillsecure.cobia.module.admin.web.Dashboard
      </managed-bean-class>
      <managed-bean-scope>session</managed-bean-scope>
      <managed-property>
         <property-name>id</property-name>
         <value>Dashboard</value>
      </managed-property>
      <managed-property>
         <property-name>moduleID</property-name>
         <value>admin</value>
      </managed-property>
      <managed-property>
         <property-name>beanName</property-name>
         <value>adminHome</value>
      </managed-property>
   </managed-bean>
...
</faces-config>
```

Note that the screen ID, the module's ID, and the bean name are configured directly by this declaration. The `AbstractScreen.getModule` method knows how to use the module ID to retrieve the ModuleConfiguration object from the Appliance class.

The `AbstractDashboard` class overrides the `getButtons` method and returns an empty list because dashboards do not have screen manipulation buttons.

The `AbstractDashboard` class overrides the `open` method and calls the `refresh` method before returning the dashboard's view ID.

The Update Screen API

Figure 21 shows the UML diagram of the `UpdateScreen` interface API. Fullscreen layout, configuration screens should subclass the `AbstractUpdateScreen` class.

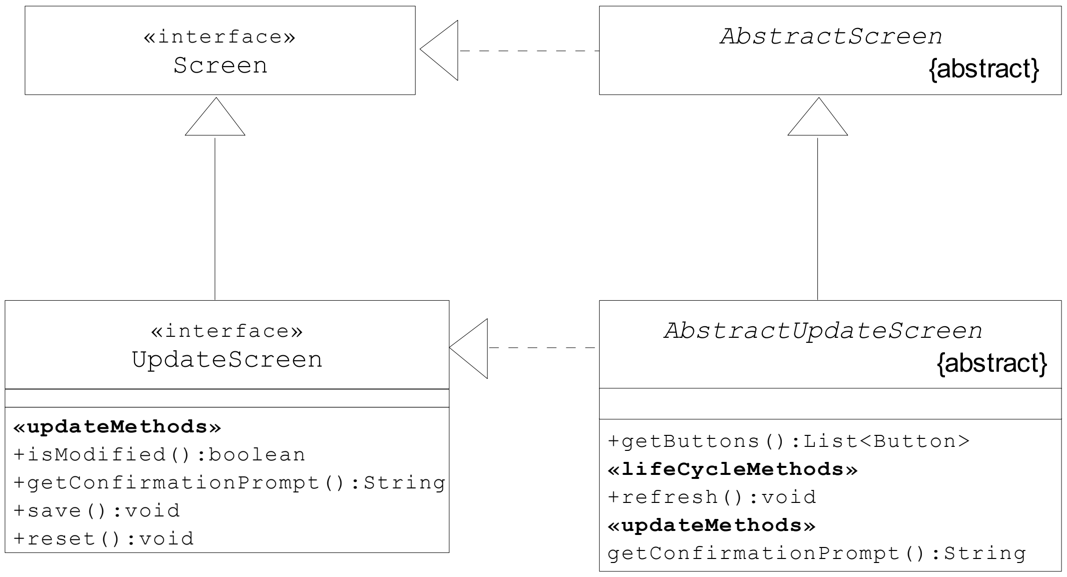


Figure 21: The UpdateScreen Interface and Abstract Base Class

The UpdateScreen interface extends the Screen interface and provides additional methods to handle configuration content.

The `isModified` method queries the screen object to determine if any form data has changed. The base class does not provide a default implementation because this method is always screen-specific.

The `save` method saves the form data changes. Depending on the screen and your module, this could mean simply saving data into the local entities of the Domain model, or it could save data to a database, or it could execute a Job to one of the module's Manager objects.

The `reset` method resets the form data.

The `getConfirmationPrompt` method returns a user message. This is used when the user attempts to navigate away from a configuration screen before they have saved any changes on the screen. The default implementation looks up the message in the module's text.properties file using the key based on the template: `<moduleID>.<screenID>.confirmationPrompt`; for example, the Admin module EditHost key is: `admin.EditHost.confirmationPrompt`.

The AbstractUpdateScreen also provides default implementation for several Screen methods. The `getButtons` method returns a list of two buttons: the “OK” and “Cancel” buttons. The behavior of the OK button is to call the `save` method on the current screen and pop back to the parent screen. The behavior of the Cancel button is to call the `reset` method on the current screen and pop back to the parent screen.

The Multi-Content Screen API

Figure 22 shows the UML diagram of the MultiContentScreen interface and the abstract base class.

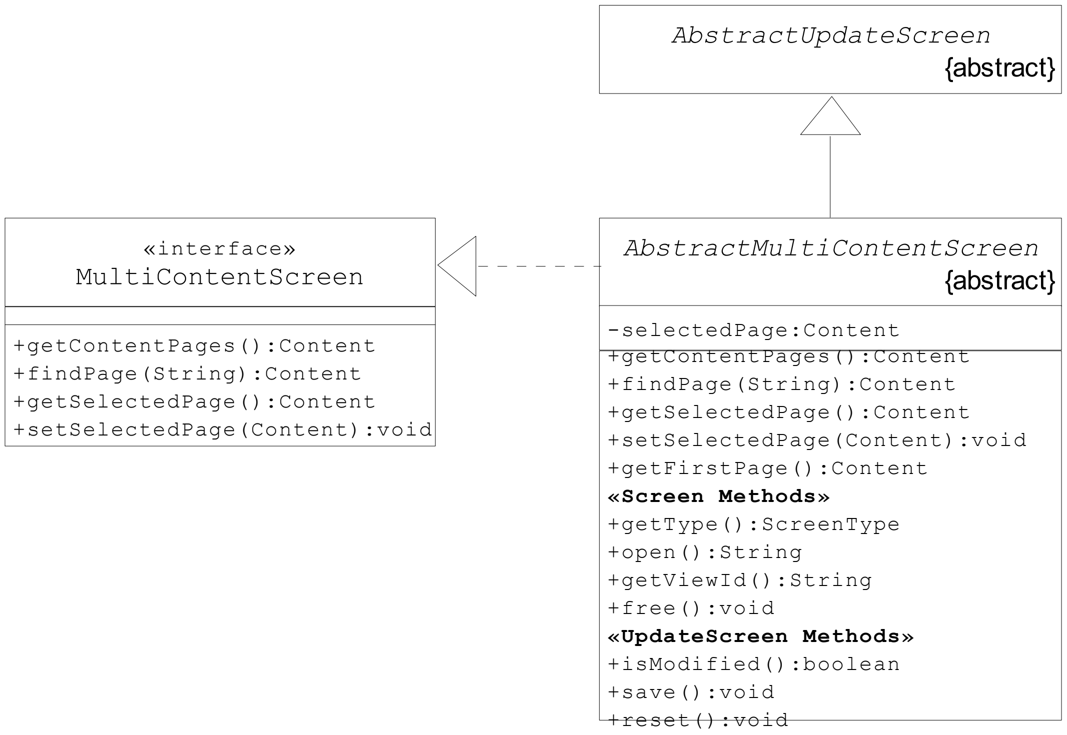


Figure 22: The MultiContentScreen Interface and Base Class

A Multi-Content screen contains one or more pages of content that make up a single, conceptual screen. The interface provides methods to access these pages. At any given time, there is only one content page visible to the user; this is called the *selected page*. A content page is very similar to screens in most regards; they have JSPX pages and backing beans which implement the Content interface.

The MultiContentScreen Methods

The AbstractMultiContentScreen is a base implementation of this interface and it also extends the AbstractUpdateScreen. This decision was made because the typical use of Multi-Content screens is organize complex configuration information. However, it could be possible that you might want a Multi-Content screen whose pages are only for viewing; this is also possible and is not prohibited by this implementation.

The `getContentPages` method must return the complete list of content pages. Typically, this method is used to construct the pages and store the list for later use. Because this method is highly dependent upon the actual screen, this method does not have a default implementation.

The `findPage` method finds the content page based on the String parameter, which is the ID of the page.

The `getSelectedPage` method returns the current selected page. The default implementation returns the `selectedPage` attribute if it has been set; otherwise, this method returns the first in the list of pages.

The `setSelectedPage` method sets the `selectedPage` attribute. This method is called by the HTML links generated by the MultiContentNavigation UI widget. This widget is included in the Multi-Content layout by default. It generates the left-hand column menu of tabs that allow the user to move between content pages.

The Screen Methods

The AbstractMultiContentScreen class also provides default implementations for several Screen methods.

The `getType` method return the `MULTI_PAGE` ScreenType value.

The `getViewId` and `open` methods return the view ID for the current selected page. This is because Multi-Content screens themselves to not have a JSPX file, but rather depend upon the content page JSPX files to render the specific pages of this screen.

The `free` method calls the `free` method for each content page and then removes the session-scoped attribute for this screen.

The UpdateScreen Methods

The AbstractMultiContentScreen class also provides default implementations for several UpdateScreen methods.

The `isModified` method iterates over each content page and returns true if any of the pages have been modified.

The `save` method iterates over each content page and calls the `save` method on the page if it has been modified.

The `reset` method iterates over each content page and calls the `reset` method on each page if it is a UpdateContent page.

Content Page API

Figure 23 shows the UML diagram of the relationship between the MultiContentScreen interface and the Content interfaces.

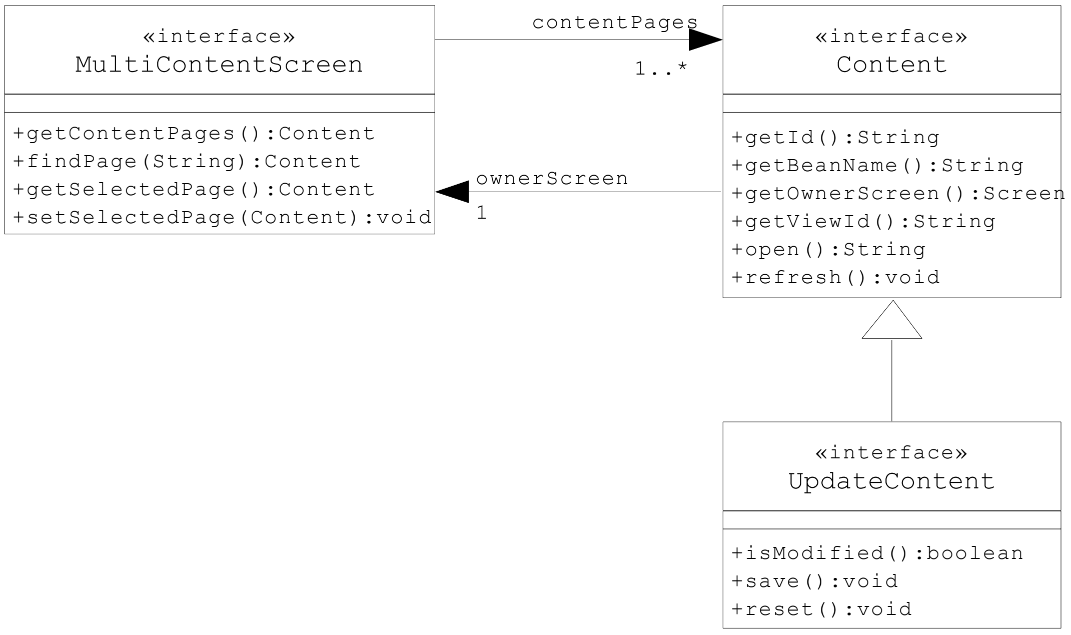


Figure 23: The Multi-Content Screen and the Relationship to the Content

The API for the two Content interfaces mimics the API for the two primary Screen interfaces; so I will not discuss the individual methods. The only addition is the `getOwnerScreen` method. This method returns a reference to the owner Multi-Content screen object.

These interfaces also have default implementations called: AbstractContent and AbstractUpdateContent. The default method implementations are basically the same as those for the base Screen classes. In addition, there is a protected constructor in the base classes that takes three arguments:

- Page ID (the name of the page class which is also the name of the JSPX file)
- Bean name
- Owner screen.

All subclasses of either of these base classes must provide a constructor which calls the super and pass in these three pieces of information.

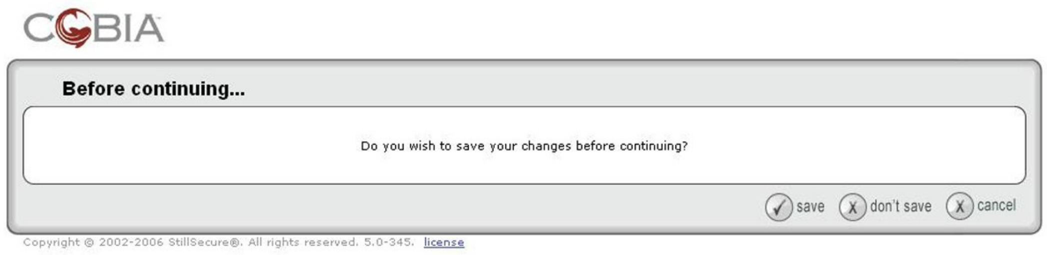
Breadcrumbs and Confirmation

Every page includes the BreadcrumbTrail UI widget which appears between the inter-module navigation bar and the gray frame which constitutes the main content of the page.

NOTE: Dashboard screens do not render this widget because dashboards are always the root of the breadcrumb trail.

The breadcrumb trail shows the screens that have been traversed to get to the current screen. Each ancestor screen renders as an HTML link which the user can click on to navigate back to that screen. Doing so pops all of the screens off of the breadcrumb stack from the current screen up to the selected screen.

However, if the current screen and any other intermediate screens have unsaved configuration changes, the user will be immediately sent to the Confirmation screen (see Figure 24) and asked to either: Save, Don't save, or Cancel the operation. The message in the center of the screen is generated from the screen that is modified, by calling the `getConfirmationPrompt` method, or a generic message if multiple screens require confirmation.


Before continuing...

Do you wish to save your changes before continuing?

save
don't save
cancel
Copyright © 2002-2006 StillSecure®. All rights reserved. 5.0-345. license

Figure 24: Confirmation Screen

The **Save** button tells the Cobia UI Framework to call the `save` methods on all screens that have configuration modifications; starting with current screen and working backwards to the screen just before the selected screen. Each of these screens are popped off of the breadcrumb stack. Finally, the `open` method of the selected screen is called providing the JSF framework with the view ID to navigate to.

The **Don't save** button tells the Cobia UI Framework to call the `reset` methods on all screens between the current screen and the screen just before the selected screen. Each of these screens are popped off of the breadcrumb stack. Finally, the `open` method of the selected screen is called providing the JSF framework with the view ID to navigate to.

The **Cancel** button returns the user to the current screen by calling the `open` method on the current screen object.

Furthermore, clicking on any module icon in the intra-module navigation menu also invokes the confirmation process.

Multi-Content Screens

This section describes the programming effort involved in building a multi-content screen.

Purpose of Multi-Content Screens

Multi-Content screens are used to present complex data or configuration forms in multiple pages of content but are treated as a single unit; a single screen. This should only be used when each page of content is truly related to a single, coherent, but large chunk of content.

API Review

The MultiContentScreen interface provides a list of content pages each of which has its own JSPX file and usually its own backing bean; although that need not be the case. For more details read the The Multi-Content Screen API section on page 37.

The API allows the user to select a specific page to be viewed. This is done using the MultiContentNavigation UI widget which is built into the Multi-Content screen template. See the Multi-Content Screens (Overview) section on page 14.

Creating Multi-Content Screen Backing Beans

Creating the backing beans for Multi-Content screens is usually very easy. Simply extend the AbstractMultiContentScreen class and implement the getContentPages method. This requires that you have at least the first cut at an implementation of each content page class. Of course, you can always just start with one content page at a time and build up the complete screen iteratively.

Here is an example of a Multi-Content screen from the Admin module:

```
public class ConfigureSystem extends AbstractMultiContentScreen
{
    public ConfigureSystem()
    {
        // do nothing
    }

    private List<Content> pages;

    //
    // MultiContentScreen methods
    //

    public List<Content> getContentPages()
    {
        if ( this.pages == null )
        {
            this.pages = makePageList();
        }
        return this.pages;
    }

    protected List<Content> makePageList()
    {
        List<Content> pages = new ArrayList<Content>();
        pages.add(new GeneralSettings(this));
        pages.add(new ViewInterfaces(this));
        pages.add(new ViewNetworksHosts(this));
        pages.add(new EditPassword(this));
        return pages;
   }

    //
    // Screen methods
    //

    public void refresh()
    {
        // do nothing
    }
}
```

Commit Points

This section describes the current strategy for committing configuration data to a module's backend manager. This strategy is likely to change and evolve over time as we gain more experience building modules for a variety of different services.

The fundamental element of the Cobia commit point strategy is to save all configuration changes for the whole module and save the whole model at one single point within the module. The rationale is that most module's deal with a single service which has a single configuration file. In these cases it makes sense to save all of the changes at one point.

Saving occurs at the second-tier configuration screen, which is usually a Multi-Content screen. Figure 25 shows the commit point for the Firewall module.

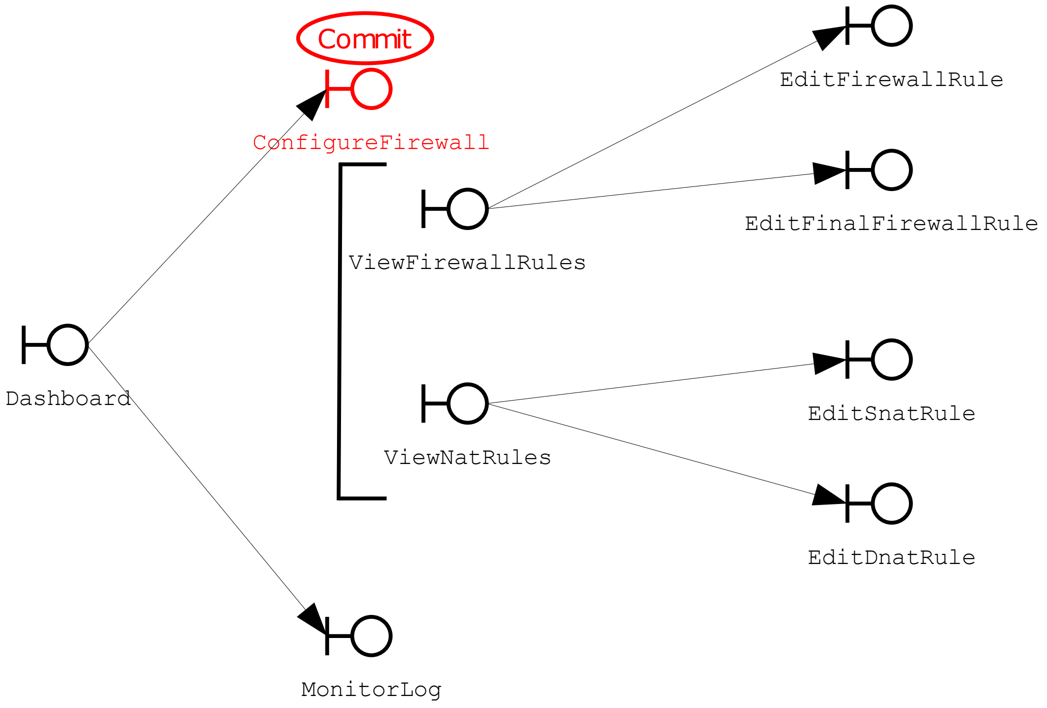


Figure 25: Commit Point for the Firewall Module

Any changes for individual firewall or NAT rules in screens such as EditFirewallRule, EditSnatRule and so on, are saved in the domain model in memory. The complete FirewallState model object is saved to the backend service when the user clicks the OK button on the ConfigureFirewall Multi-Content screen (either on the ViewFirewallRules or ViewNatRules page).

```
public void save()
{
    // Save each content page
    super.save();

    // Commit the complete model to the backend
    // This might require additional work,
    // but not in the firewall module.

    // Create the Job
```

```
        Job job = new Job(FirewallConstants.MODULE_ID,
                FirewallConstants.MANAGER_NAME,
                FirewallConstants.SET_STATE_METHOD,
                WebContext.getCurrentUser());

        // Store the model in the job
        job.addParamByName(FirewallConstants.SET_STATE_METHOD_IN_PARAM_MODEL
,
                this.dashboard.getModel());

        // Execute the job
        job = appliance.injectDependentJobs(job);
        JobResult jr = appliance.executeJob( job );

        // Inform the user of any messages from the job
        for( FeedbackMessage usermessage : jr.getUserMessages() )
        {
            MessageUtils.addMessage( usermessage );
        }
    }
```

Some times a module might include multiple managers with independent domain models. In this case, each content page within the second-tier Multi-Content screen should be responsible for performing the individual commit points by using the `UpdateContent.save` method for each page's backing bean.

Strategies to Get the Domain Model

In order to make changes to a configuration, you must also be able to retrieve the configuration in the form of a Domain model from the backend. This is also done using the Cobia Job facility.

For modules that use a single Domain model, like Firewall, it usually makes sense to have the Dashboard screen object retrieve the model and then allow the configuration screens to manipulate that model. Here is the code for the Firewall `Dashboard.refresh` method:

```
    public void refresh()
    {
        // Create the Get Job
        Appliance appliance = Appliance.instance();
        Job job = new Job(FirewallConstants.MODULE_ID,
                FirewallConstants.MANAGER_NAME,
                FirewallConstants.GET_STATE_METHOD,
                WebContext.getCurrentUser());

        // Perform the job
        job = appliance.injectDependentJobs(job);
        JobResult result = appliance.executeJob(job);
        this.model = (FirewallState)
result.getResult(FirewallConstants.GET_STATE_METHOD_OUT_PARAM_MODEL);
        this.serverState = (ServiceState)
result.getResult(FirewallConstants.GET_STATE_METHOD_OUT_PARAM_SERVICE_STATE)
;

        // Process the user messages for this job
        List< FeedbackMessage > userMessages = result.getUserMessages();
        for( FeedbackMessage usermessage : userMessages )
        {
            MessageUtils.addMessage( usermessage );
        }

    }
```

Other modules might require multiple Domain models in different configuration screens (or content pages). In this case, the individual page should perform the “get model” Job in the `reset` method.

For the rest of this document, we will not discuss the use of “get” and “set” jobs in order to keep the code examples as simple as possible.

Configuration Pages

This section describes how to create configuration pages.

The Purpose of Configuration Pages

A configuration page is a Fullscreen or page of a Multi-Content screen that manipulates data in the Domain model for the purpose of configuring the backend service.

A configuration page may also have buttons on the page that performs actions on the managers of your module, which ultimately affects the operation of the service.

Elements of a Form Field

One of the main goals of the Cobia UI Framework is to provide a higher level of abstraction on top of the standard JSF form component. In Cobia, form fields usually include five elements. Figure 26 illustrates these elements for the ImageList component.

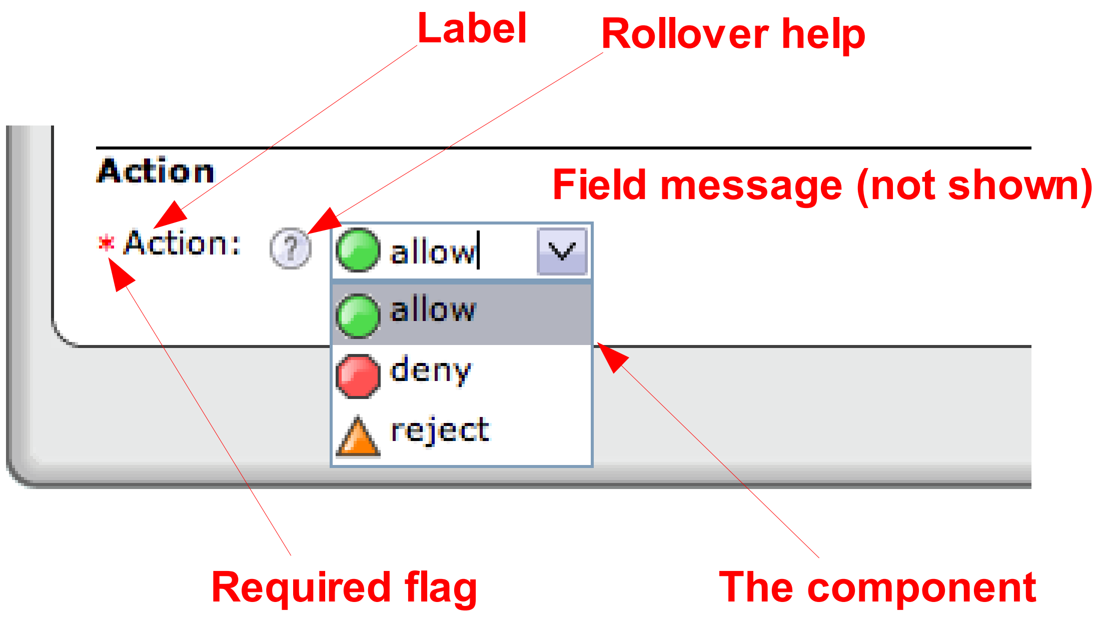


Figure 26: Elements of a Form Field Component

The five elements are:

- The field component itself: a text field, a drop-down list, and so on
- The required flag; missing if the field is not required
- The field label
- The optional rollover help text
- The field message, which is displayed if a data conversion or field validation error occurs

These five elements are arranged in different positions based upon the UI design of the specific component. The component's JSF renderer decides this placement and so

The UI Component library includes several of these individual elements (FieldLabel, RolloverHelp, FieldMessage) as unique components but these are not usually used in isolation.

Screen Text and Internationalization Support

The Cobia Framework has a rich set of utilities for supporting internationalization (often abbreviated as I18N). In general, all user-visible text is stored in the module's `text.properties` resource bundle file located in the module's `src/properties/` directory. The Cobia UI Framework takes advantage of these facilities and you will quickly notice that no user-visible text will ever appear in the JSPX files. For example, the labels and help text for components are stored in this resource bundle. For example, the label for the component in Figure 2 on page 8 is defined in the resource bundle like this:

```
firewall.EditFirewallRule.ruleAction.Label=Action
```

Take a close look at the structure of the resource key. The first element is the module ID: `firewall`. The second element is the screen ID: `EditFirewallRule`. The third element is the comonent ID: `ruleAction`. The last element is the attribute of the component: in this case `Label`. Some components have several text attributes and therefore may have multiple entries in the text.properties file with different attribute. Here is the entry for the ruleAction component's help text:

```
firewall.EditFirewallRule.ruleAction.helpText=The action that will be taken
upon...
```

Other Text Messages

Beyond the support for component labels, the resource bundle should be used for all text messages that will be shown to the user (and ideally even those that are sent to the log file). Again, the common format of message keys is: `<moduleID>.<screenID>.<messageKey>`; for example, the message when a RipInterface entity is saved uses the key: `router.EditRipInterface.saveMessage`.

Sometimes a message may be used by many screens or is generic; in that case you may use the key structure: `<moduleID>.<messageKey>`. What is most important is that all of your module's keys start with the module ID. The reason for this is that the `text.properties` files of all modules are merged together and this first element of the key is used as a namespace to prevent message key collisions.

Supporting Substitution

Just like in software development, your resource bundles should follow the Don't Repeat Yourself principle. The most obvious example of this is that some screens may have the same fields; this is very common with two screens that manipulates a single type of entity, such as add and edit screens. For example, the Router module has two screens to manipulate the RipInterface entity: AddRipInterface and EditRipInterface. The add screen includes a drop-down list to select the Ethernet interface for the RipInterface, but the edit screen does not allow the user to change this attribute. However, both screens share all other fields, such as the “Passive mode on” checkbox. Here is the component from the JSPX file:

```
<c:booleanCheckbox id='passive' value='#{addRip.rip.passive}' />
```

The label for the edit screen is:

```
router.EditRipInterface.passive.Label=Passive mode on
```

The label for the add screen is identical. We could repeat this text, but if you decided to change it in edit screen you might forget to change it in the add screen which could lead to confusion for your users. The Cobia I18N facilities support the ability to include a substitution using the syntax: `{!KEY}` to reference another KEY defined in the resource bundle. Here is the label definition in the add screen:

```
router.AddRipInterface.passive.Label={!router.EditRipInterface.passive.Label
}
```

Another use of substitutions is that frequently help text will refer to the field label. Here is the extended version of the ruleAction help text in the Firewall module:

```
firewall.EditFirewallRule.ruleAction.helpText=The <span
class='reference'>{!firewall.EditFirewallRule.ruleAction.Label}</span> is
the action that will be taken upon...
```

Notice the label substitution within the <span> element in the help text. If the ruleAction label ever changes, then the help text will automatically be updated as well.

Another common use is to define a set of common terms or acronyms used frequently within the text of the UI. Here is an example from the Admin module:

```
admin.IP.acronym=<acronym title='Internet protocol'>IP</acronym>
admin.IPv4.acronym=<acronym title='Internet protocol version
4'>IPv4</acronym>
admin.IPs.acronym=<acronym title='Internet protocol (IP)
addresses'>IPs</acronym>
admin.TCP.acronym=<acronym title='transmission control
protocol'>TCP</acronym>
admin.MAC=MAC
admin.MAC.acronym=<acronym title='media access control'>MAC</acronym>
admin.VIF=virtual interface
admin.VIF.acronym=<acronym title='virtual interface'>VIF</acronym>
```

You can then use these generic terms embedded in a variety of label, help text, and user messages for your module's GUI.

In general, it is not a good idea to use terms defined in other modules (although it is legal to do so); however, you may use keys defined in the Admin and Firewall modules because these will always exist in a Cobia appliance.

Supporting Multiple Languages

If you are not familiar with Java's I18N support, you might want to read the on-line documentation and javadocs for the ResourceBundle class. In general, a properties file can be translated into multiple languages and at runtime the "locale" of the user's web browser determines which resource bundle is used to generate the text for the GUI. So for example, if the user speaks German, then the `text_de.properties` file will be used to retrieve the text based on the common set of keys.

Currently, Cobia modules have not been translated, but the framework for supporting internationalization is in place.

Lastly, you should read through the base `text.properties` file in the `appliance/src/properties/` directory. There are many generic terms that have been pre-defined that you can use for substitution in your text messages.

Screen Buttons

By default, every configuration page has two screen-level buttons, OK and Cancel, which appear at the top and bottom of the main content frame on the page. Figure 27 shows an example of the EditRipInterface screen.

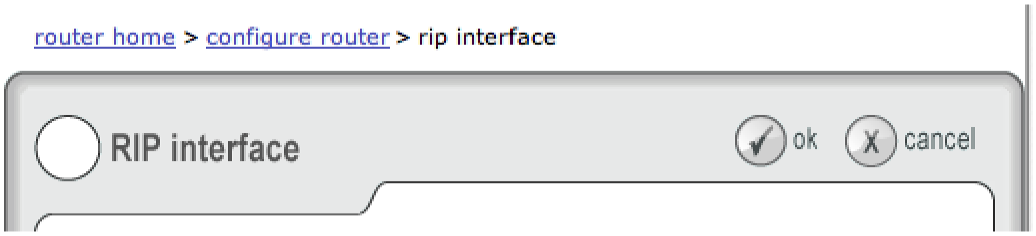


Figure 27: Screen Butons

When the user clicks on the OK button, the page's `save` method is called and this page is popped off of the breadcrumb stack and the parent screen is displayed. If there are any field conversion or validation errors, then these are displayed so the user can fix these before proceeding with the save.

When the user clicks on the Cancel button, the page's `reset` method is called and this page is popped off of the breadcrumb stack and the parent screen is displayed. Any field conversion or validation errors are ignored.

Content Buttons

Content buttons are special UI buttons that allow the user to execute actions on either the UI or on the module's manager. Figure 28 shows two types of content buttons that currently exist in the UI component library.

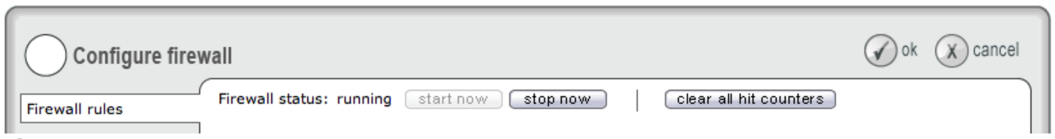


Figure 28: Content Buttons

The first example (stop now) is called a ToggleButtons component and in this case it is used to start and stop the firewall service. Here is the JSP code to generate this component:

```
<c:toggleButtons id='firewallStatus' condition='#{firewallBean.running}' >
  <c:contentButton id='startCommand' action='#{firewallBean.startServer}'
type='StartNow' />
  <c:contentButton id='stopCommand' action='#{firewallBean.stopServer}'
type='StopNow' />
</c:toggleButtons>
```

The ToggleButtons component must contain two, and only two, ContentButton subcomponents. The `condition` attribute must be a value binding that returns a Boolean value. The first button is active when the `condition` attribute is false; otherwise, the second button is active. Here is the code for this value binding:

```
    public boolean isRunning()
    {
        return (this.dashboard.getServerState() == ServiceState.RUNNING);
    }
```

Each ContentButton must have a JSF action method binding in the `action` attribute. This is called when user clicks on that button. Note that only the enabled button can be clicked by the user. Typically, these methods will execute some Job against a Manager of your module. Here is the code for the Firewall `startServer` action method:

```
    public String startServer()
    {
        Appliance appliance = Appliance.instance();
        User user = (User) WebContext.getCurrentUser();
        Job job = new Job(FirewallConstants.MODULE_ID,
                          FirewallConstants.MANAGER_NAME,
                          BaseConstants.START_SERVICE_METHOD,
                          user);
        job = appliance.injectDependentJobs(job);
        JobResult result = appliance.executeJob(job);
        if ( ! result.hasError() )
        {
            ServiceState serverState = (ServiceState)
result.getResult(BaseConstants.START_SERVICE_METHOD_OUT_PARAM);
            this.dashboard.setServerState(serverState);
        }

        // Process the user messages for this job
        List< FeedbackMessage > userMessages = result.getUserMessages();
        for( FeedbackMessage usermessage : userMessages )
        {
            MessageUtils.addMessage( usermessage );
        }

        // Return back to this screen
        return getViewId();
    }
```

Note that the value returned is the view ID of this screen bean; typically, ContentButtons should return to the same screen, but this is not a requirement and your action methods could choose to open another screen.

The ToggleButton component requires the following additional CSS styles that you should include in the module's CSS file:

```
div#firewallButtons {
   border-bottom: solid 1px #666;
   margin-bottom: 1em;
   padding-bottom: 1em;
   width: 98%;
}
div#firewallStatus div {
   float: left;
}
div#firewallStatus span {
   float: left;
   margin: 0 .5em;
}
```

Lastly, the icons associated with each ContentButton are defined in a CSS file and identified in the tag by the `type` attribute. In this example, the first ContentButton has the type of StartNow. Here is the CSS styles for that button:

```
a.contentStartNowButton {
   background: url(../images/button_start_now.gif) top left no-repeat;
   width: 76px;
}
```

```
a.contentStartNowButton:hover {
   background: url(../images/button_start_now_hover.gif) top left no-repeat;
   width: 76px;
}
a.contentStartNowButtonInactive {
   background: url(../images/button_start_now_inactive.gif) top left no-
repeat;
   width: 76px;
   cursor: default;
}
```

There are four predefined ContentButton types:

- Reset
- Search
- StartNow
- StopNow

Their icons are already available in the UI Framework; however, you may create your own ContentButton types and iconsby performing the following steps:

1. Create three icons: active, hover, and inactive.
2. Place them in your module's `web/images/` directory.
3. Create three styles in the `web/styles/<moduleID>.css` file that mimic the above styles but replaces “StartNow” with the type name of your button

You may also use ContentButton components in isolation; not just inside of the ToggleButton component.

Form Layout Techniques

All configuration forms must be contained with a FormWrapper component. This creates a `<div>` structure in the rendered HTML that is then styled to define the widths of the field labels; thus providing alignment to the field components themselves. Here is a simple example from the Admin module's GeneralSetting page:

```
<c:formWrapper id='generalSettings'>
  <c:textField id='hostName' value='#{bean.hostName}' required='false' />
  <c:textField id='domainName' value='#{bean.domainName}' required='false'
/>
  <c:textField id='defaultGateway' value='#{bean.defaultGateway}'
required='false'>
    <f:converter converterId='converter.Ipv4Address' />
  </c:textField>
</c:formWrapper>
```

This form contains three simple TextField components contained in a FormWrapper. Here is the style in the web/styles/admin.css file to define the label widths within this div structure:

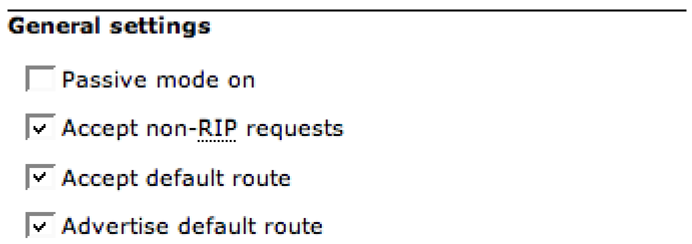
```
div#generalSettings label { /* Form label width */
   width: 12em;
}
```

The width value depends on the length of the longest label in the page's form. Play with this value until the labels no longer wrap in the three supported browsers: FireFox v2, InternetExplorer (IE) v6 and IE v7.

Creating Sections

Sometimes it is important to group fields that have a common purpose.

The easiest way to do this is to separate a group of fields using a SectionHeading component. Figure 29 shows an example heading from the Router module's EditRipInterface screen:


General settings

- [ ] Passive mode on
- [✓] Accept non-RIP requests
- [✓] Accept default route
- [✓] Advertise default route

Figure 29: Example SectionHeading in the EdtRipInterface Screen

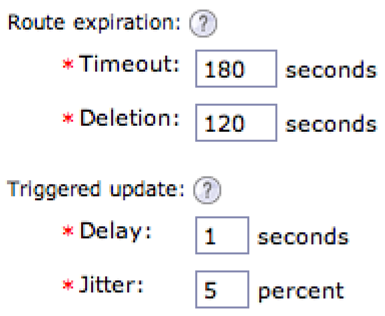
Here is the JSPX code to generate the section heading:

```
<c:sectionHeading id='ripGenSetting' />
<c:booleanCheckbox id='passive' value='#{addRip.rip.passive}' />
<c:booleanCheckbox id='acceptNonRipRequests'
value='#{addRip.rip.acceptNonRipRequests}' />
```

The text for the heading is defined in the text.properties file using the Label component attribute. The RIP heading above is defined by this line:

```
router.EditRipInterface.ripGenSetting.Label=General settings
```

If a small group of fields have high cohesion, you may also group them into a SubSection component. Figure 30 shows an example from the Router module's EditRipInterface screen:


Route expiration:

* Timeout: 180 seconds

* Deletion: 120 seconds

Triggered update:

* Delay: 1 seconds

* Jitter: 5 percent

Figure 30: Example SubSection in the EditRipInterface Screen

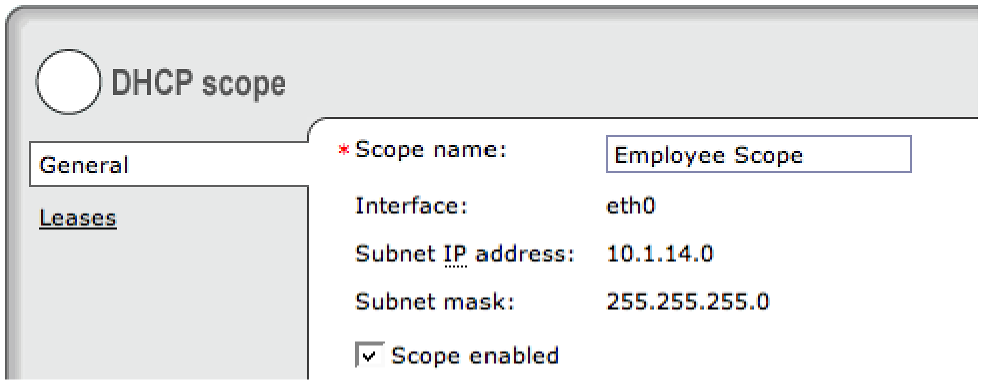
The SubSection creates a heading (with an option help text icon) and then indents the child components. In this example, the `routerExpiration` SubSection contains two TextField components. Here is the JSPX tags for this content:

```
<c:subSection id='routeExpiration'>
  <c:textField id='routeExpirationTimeout'
value='#{addRip.rip.routeTimeout}' size='4' required='true'>
    <f:converter converterId='converter.Integer' />
    <f:validateLongRange minimum='0' />
  </c:textField>
  <c:textField id='routeExpirationDelay' value='#{addRip.rip.deletionDelay}'
size='4' required='true'>
     <f:converter converterId='converter.Integer' />
     <f:validateLongRange minimum='0' />
   </c:textField>
</c:subSection>
```

The SubSection component must have a `Label` text attribute and an optional `helpText` attribute in the text.properties file.

Displaying Content in Forms

Sometimes you need to display data in a form to provide necessary information to the user to properly fill out the form. Figure 31 shows an example from the DHCP module:



* Scope name: Employee Scope

Interface: eth0

Subnet IP address: 10.1.14.0

Subnet mask: 255.255.255.0

☑ Scope enabled

Figure 31: Example Form Output

Here is the JSPX code for this chunk of the form:

```
<c:formWrapper id='editDhcpScope'>
  <c:textField id='scopeName' required='true'
value='#{editScope.scope.name}' validator='#{editScope.validateScopeName}'>
    <f:converter converterId='converter.TrimString' />
  </c:textField>
  <c:formOutput id='scopeInterface' styleClass='formTwoColumnInfo'
      value='#{editScope.scope.ethInterface.device}' />
  <c:formOutput id='subnetIpAddress' styleClass='formTwoColumnInfo'
      value='#{editScope.scope.ethInterface.networkAddress}' />
  <c:formOutput id='subnetNetMask' styleClass='formTwoColumnInfo'
      value='#{editScope.scope.ethInterface.netmask}' />
```

The text for the labels is defined in the text.properties file:

```
dhcp.EditScope.scopeInterface.Label=Interface
dhcp.EditScope.subnetIpAddress.Label=Subnet {!admin.IP.acronym} address
dhcp.EditScope.subnetNetMask.Label=Subnet mask
```

Lastly, the labels are actually HTML `<dt>` elements which need to be styled to be the same width as the rest of the labels in the form. This code needs to be in the `dhcp.css` file:

```
div#editDhcpScope label, div#editDhcpScope dt { /* Form label width */
   width: 12em;
}
```

The second CSS selector "div#editDhcpScope dt" identifies the FormOutput components in the editDhcpScope FormWrapper.

Domain Modeling Review

This section provides a quick review of the domain modeling techniques used in Cobia. There are two fundamental types of domain objects: entities and value objects. Figure 32 illustrates the general types of domain model elements used in Cobia.

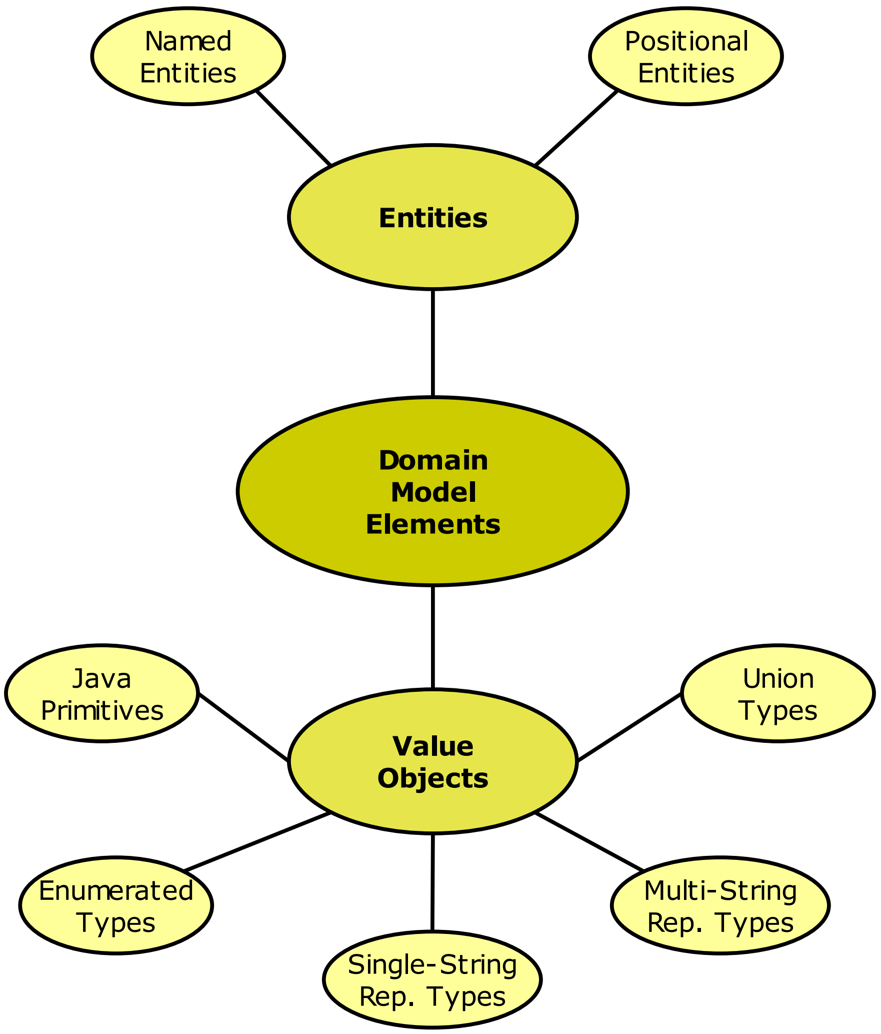


Figure 32: Types of Domain Model Elements

Entity Model Elements

An *entity* is a complex object that has some form of *fixed identity*. The identity is usually defined by the domain itself. For example, an Ethernet interface identity is defined by the symbolic device name given to it by the operating system, such as "eth0" or "lo". Another example, a RIP interface is identified by the Ethernet interface assigned to the RIP interface. The concept of an entity is directly supported in Cobia with the `Entity` interface and `AbstractEntity` base class in the `org.stillsecure.cobia.base.domain` package.

There are two special subtypes of entities: named entity and positional entity.

A named entity is, as it sounds, an entity that has a user-defined name. Generally speaking, a named entity does not have an identity that is based upon the name because in most cases we allow the user to change these names; thus the names are not **fixed** and cannot be used in an `equals` method to defined the identity of the entity class. The named entity concept is also supported in Cobia with the `NamedEntity` interface and `AbstractNamedEntity` base class.

A positional entity is an entity that exists within an ordered list within another entity class. For example, a `UiFirewallRule` is a positional entity within the `FirewallState` entity because the order of firewall rules is significant. At this time, there is no direct support for the positional entity concept in Cobia.

Entities are mutable objects; the data attributes of an entity instance can be changed. Whenever a change occurs the state of the entity should reflect this change, by calling the `setState` method. Typically, the `UpdateScreen.save` method changes the state of an entity before leaving the screen that configures that entity type. The `UpdateScreen.isModified` method is called before the save. If this method returns false, then save is not called; thus the entity state is not changed. The `Entity` interface includes an inner enum called `State` that has four values:

- `UNMODIFIED`
- `MODIFIED`
- `ADDED`
- `DELETED`

The latter two states are for entities that are within a collection (set or list) within another entity.

Configuration screens are used to edit (modify) and add entities. The EntitySet and EntityList components are used to manipulate collections of entities. The Dealing with Entities section on page 65 describes how to build UI components that manipulate collections of entities.

Value Object Model Elements

A *value object* is a relatively simple object that does not have identity; that is, the equality of two value objects is solely defined by the value(s) contained in the objects. A corollary of this definition is that value objects are immutable. You cannot change the internal data of a value object. If you need new data, simply create a new instance of the value object class.

There are five special subtypes of value objects:

- Java primitives
- Enumerated types (Java `enum` classes)
- Single-string representation types
- Multi-string representation types
- Union types

There are eight *Java primitives*:

- Boolean
- byte
- short
- int
- long
- float
- double
- char

Typically, a Cobia module will likely only need Boolean, integer, and float-point primitive types. The Dealing with Primitive Data section on page 70 describes how to build field components that manipulate primitive data.

Enumerated types are types that have a finite set of values that are given symbolic names. Java SE v5 now directly supports enumerated types using the `enum` construct. For example, the Firewall module needs to distinguish between several IP protocols and has defined the Protocol enumerated type with these values:

- `ANY`
- `TCP`
- `UDP`
- `TCP_AND_UDP`
- `ICMP`

Enumerated types are frequently used in drop-down lists and displayed in summary data tables. The Dealing with Enumerated Types section on page 74 describes how to build field component that manipulate enum data.

Single-string representation types are any data types that can be represented with a single, or simple, string of information. This is a very loose definition, so let me show some examples to demonstrate this definition:

- An IPv4 address can be represented by a dotted-quad of bytes (integers between 0 and 255); for example, "192.168.1.2" and "10.0.47.255" are both valid IPv4 addresses. This data type is so useful that the Cobia team has created a class to represent this data: the `Ipv4Address` in the `org.stillsecure.cobia.base.domain.values` package.
- A date is another example of a data type that can be represented by a single string.
- Other examples include: NetMask and MacAddress.

Of course, you could always just store this information as raw String objects in your Domain model. *The question you need to ask yourself is,* "Does the Domain model or module manager classes need to manipulate this data?" Stated as an object-oriented question, "Does the data have behavior?" If the answer is yes, then it is probably a good idea to encapsulate that data in a Java class. The Dealing with Single-String Representation Types section on page 79 describes how to build field components that manipulate this type of data.

Multiple-string representation types are any data types that cannot be represented with a single, or simple, string of information but must be represented with several chunks of information. Again, an example will help you understand this loose definition:

- An IPv4 subnet is a combination of an IPv4 address with a netmask (or CIDR number); thus, it takes two pieces of information to construct such an object. However, who is to say that "192.168.1.2/24" is **not** a simple string representation? Clearly, it could be.

The difference between single-string rep and multi-string rep is one of preference or UI usability. Ultimately, the answer to this question is whether or not it requires two (or more) data fields to represent a single Domain model attribute. The Dealing with Multiple-String Representation Types section on page 82 describes how to build field components that manipulate this type of data.

A special subtype of multi-string-rep types is collections of simple data. The Java language defines two simple collection types:

- sets - unordered data
- lists – ordered data

Cobia defines another collection type in the `Range` interface and `RangeImpl` class in the `org.stillsecure.cobia.util` package: a range. A *range* is a collection of two values that define a start and end point. Currently, this data type implements an inclusive range. The range type is a generic class meaning it can be applied to any data types that implement Java's `Comparable` interface. For example, you can have a `Range<Integer>` or a `Range<Ipv4Address>` because both integer and IPv4 addresses understand the concept of before and after: 42 is before 47 and 192.168.1.2 is before 192.168.47.1. The Dealing with Data Collections section on page 86 describes how to build field components that manipulate collections of data.

Union types are data types that have multiple ways of representing a single piece of information. This is analogous to the C language concept of the `union` structure; hence the name. Again, an example is the best way to describe this concept. In the Firewall module, we needed to represent the concept of an IP address matching specification, which can be a single IP address, an IP subnet, a range of IP addresses, a user-defined host (a named IP address) or user-defined network (a named subnet). Take a look at the code in `IpAddressMatchingSpec` in the `org.stillsecure.cobia.base.domain.value` package. In general, a union type satisfies the following critieria:

- There is an internal enum of the different types of representation; and a `getType` method which returns the instance's representation type.
- There is a constructor for each type of representation.
- There are getter methods for each type of representation, which is only allowed if the instance is of the correct type (all calls on other getter methods should throw an AssertionError).
- No setter methods are defined.

The last criteria is relevant to all value object classes, value objects are immutable. The data of a value object must not be manipulated, ever. If you need to change the data of an attribute that is a value object, then create a new instance of the value object class and store that in the attribute.

Other Data Structures

Entities and Value Objects represent maybe 80% of the data/information representation needs of a Domain model. It is possible that you will need more complex data structures as well, such as maps, inheritable maps, queues, stacks, trees, networks, and so on. There currently are no UI components that are designed to help with these data structures.

How to Create Configuration Screen Backing Beans

Typically, a single entity type will have its own configuration screen. This screen will include form fields to change the attributes of a single entity at a time. There are two basic strategies for creating configuration screens:

- using the backing bean to store the modified data
- using the `StateInterceptor` utility to store the modified data

For the code examples in this section, imagine you have an entity class like this:

```java
public class HostAndDomainName extends AbstractEntity {

   private String hostName;
   private String domainName;

   public String getDomainName() {
      return domainName;
   }

   public void setDomainName(String domainName) {
```

```
11.         this.domainName = domainName;
12.     }
13. 
14.     public String getHostName() {
15.         return hostName;
16.     }
17. 
18.     public void setHostName(String hostName) {
19.         this.hostName = hostName;
20.     }
21. 
22.      public void resetState()
23.      {
24.          setState(Entity.State.UNMODIFIED);
25.      }
26. }
```

Using the Backing Bean

In some ways, using the backing bean object to store the modified data is the most straight-forward approach to creating configuration screens; however, as you will soon see, it requires a lot of coding.

With this strategy, the page object (backing bean) would also contain the same data attributes as the entity:

```
1.  public class EditHostDomainNames extends AbstractUpdateContent
2.  {
3.      //
4.      // Screen data
5.      //
6. 
7.      private String hostNameOld;
8.      private String hostName;
9.      private String domainNameOld;
10.     private String domainName;
11. 
12.     public String getHostName()
13.     {
14.         return hostName;
15.     }
16.     public void setHostName(String hostName)
17.     {
18.         this.hostName = hostName;
19.     }
20. 
21.     public String getDomainName()
22.     {
23.         return domainName;
24.     }
25.     public void setDomainName(String domainName)
26.     {
27.         this.domainName = domainName;
28.     }
29. // more methods
30. }
```

This class defines the backing bean for a content page used within a Multi-Content screen. Ignore that detail for now. Concentrate on lines 15-36. Notice that we have defined one set of properties: `hostName` and `domainName`, but we also maintain an "old" value for each. This will be used to determine if the user has changed any values in the UI. The getter and setter methods are used by the JSF components to modify the data in the backing bean. Here is the snippet of JSPX code that interacts with this backing bean:

```
1.  <c:formWrapper id='generalSettings'>
2.    <c:textField id='hostName' value='#{generalBean.hostName}'
required='false' />
3.    <c:textField id='domainName' value='#{generalBean.domainName}'
required='false' />
4.  </c:formWrapper>
```

Line 2 calls the `getHostName` method (line 20) to display the value in the text field and when the user submits the form JSF calls the `setHostName` method (line 24) to store the new value.

Finally, the backing bean must implement the UpdateContent methods: `isModified`, `reset`, `save`, and `refresh`. Remember, the `isModified` method determines if the user has made any changes to the data on the screen. The `reset` method is used to retrieve the data from the Domain model. The `save` method is used to store the modified data into the Domain model. And the `refresh` method is used to get the screen data back into the last saved state. Here is that code:

```
1.  public class EditHostDomainNames extends AbstractUpdateContent
2.  {
3.  // Screen data code from above (lines 3-28)
4.  
5.      public EditHostDomainNames(Screen ownerScreen, HostAndDomainName entity)
6.      {
7.          super(PAGE_NAME, BEAN_NAME, ownerScreen);
8.          this.entity = entity;
9.          reset();
10.     }
11.     private static final String PAGE_NAME =
EditHostDomainNames.class.getSimpleName();
12.     private static final String BEAN_NAME = "generalBean";
13.     private final HostAndDomainName entity;
14. 
15.     //
16.     // UpdateContent methods
17.     //
18. 
19.     @Override
20.     public boolean isModified()
21.     {
22.         if ( ! domainName.equals(domainNameOld) ) return true;
23.         if ( ! hostName.equals(hostNameOld) ) return true;
24.         return false;
25.     }
26. 
27.     @Override
28.     public void reset()
29.     {
30.         // Restore the entity data into the backing bean
31.         this.domainName = entity.getDomainName();
32.         this.domainNameOld = this.domainName;
33.         this.hostName = entity.getHostName();
34.         this.hostNameOld = this.hostName;
35.     }
36. 
37.     @Override
38.     public void save()
39.     {
40.         // Save the user data into the entity
41.         entity.setHostName(this.hostName);
42.         entity.setDomainName(this.domainName);
43.         // Modify the entity's state
44.         entity.setState(Entity.State.MODIFIED);
45.     }
```

```

    @Override
    public void refresh()
    {
        // Simply reset the backing bean data
        this.domainName = this.domainNameOld;
        this.hostName = this.hostNameOld;
    }
}
```

This code is pretty easy to understand. The most important aspect is that the save method must set the entity's state to the MODIFIED state. This tells the parent page that the entity has been changed.

Using the StateInterceptor Utility

NOTE: This facility is experimental; use at your own risk.

If your entities have many attributes, you can see that the using the backing bean screen object to hold this data quickly becomes rather tedious. The StateInterceptor utility is designed to hold the attribute changes to an object without actually changing the original object. Figure 33 Illustrates how the StateInterceptor utility works.

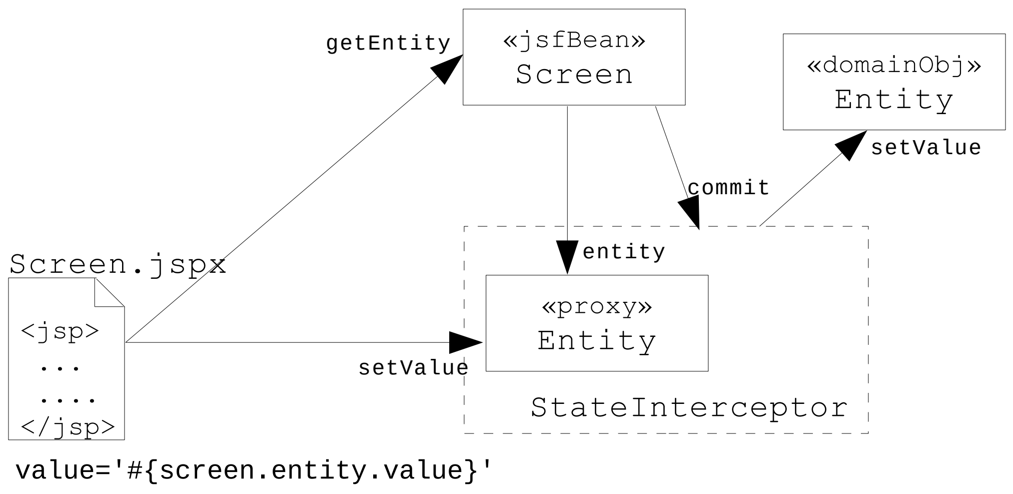


Figure 33: How the StateInterceptor Works

The screen object is created with an instance of the entity object. The screen creates an interceptor from the original entity object. The interceptor creates a proxy object which behaves just like the original entity. It starts with all of the same property data. The JSP value binding accesses this proxy object to retrieve the data and to set the data when the user submits a change. When the screen's save method is called, the screen object calls the commit method on the interceptor object which then stores the changes to the original entity object.

Using the StateInterceptor utility does not require any changes to the domain entity class; however, the screen backing bean will be coded as follows:

```
public class EditHostDomainNames extends AbstractUpdateContent
{
    public EditHostDomainNames(Screen ownerScreen, HostAndDomainName entity)
    {
        super(PAGE_NAME, BEAN_NAME, ownerScreen);
        this.originalEntity = entity;
        this.interceptor = new StateInterceptor<HostAndDomainName>(entity);
        this.proxyEntity = this.interceptor.getProxy();
    }
    private static final String PAGE_NAME =
EditHostDomainNames.class.getSimpleName();
    private static final String BEAN_NAME = "generalBean";

    private final HostAndDomainName originalEntity;
    private final StateInterceptor<HostAndDomainName> interceptor;
    private final HostAndDomainName proxyEntity;

    //
    // Model methods
    //

    public HostAndDomainName getEntity()
    {
        return this.proxyEntity;
    }

    //
    // UpdateContent methods
    //

    @Override
    public boolean isModified()
    {
        return this.interceptor.isModified();
    }

    @Override
    public void reset()
    {
        // Restore the entity data into the backing bean
        this.interceptor.rollback();
    }

    @Override
    public void save()
    {
        // Save the user data into the entity
        this.interceptor.commit();
        // Modify the entity's state
        this.originalEntity.setState(Entity.State.MODIFIED);
    }

    @Override
    public void refresh()
    {
        // Simply reset the backing bean data
        reset();
    }
}
```

The JSPX file also needs to reference the entity proxy. Here are the changes to the main content of this example page:

```
<c:formWrapper id='generalSettings'>
  <c:textField id='hostName' value='#{generalBean.entity.hostName}'
required='false' />
  <c:textField id='domainName' value='#{generalBean.entity.domainName}'
required='false' />
</c:formWrapper>
```

The great thing about using the StateInterceptor utility is that no matter how many properties (JavaBean-style get/set method properties) you add to entity object, the backing bean code need not change.

There are limitations to the StateInterceptor utility:

- one-to-one relationships to other entity objects are not directly handled

  To make changes to entities referred to by the proxy, you should use another screen to make those changes.
- one-to-many relationships to other entity objects are not yet handled
- complex data structures such as maps, trees, and graphs are not yet handled
- derived properties are handled but require special coding

Here is an example of how to code a derived property:

```
1.     @StateInterceptorMethod (type = InterceptorMethodType.DERIVED_PROPERTY,
   propertyName = "")
2.     public int getDomainNameLength()
3.     {
4.         return getDomainName().length();
5.     }
```

This method determines the number of characters in the entity's domainName property. There are two coding requirements to make this derived property to reflect any changes to the domainName property in the proxy.

- First, the method must be annotated to declare that this method is recognized by the StateInterceptor utility (line 1).
- Second, the code must **not** use the private attributes directly, but must use methods to access all data in the object (line 4).

Using a Hybrid Strategy

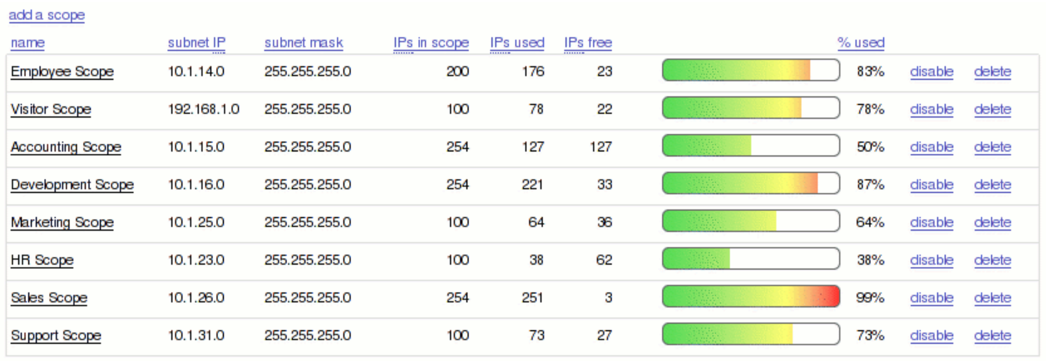
Ideally, the StateInterceptor utility would handle all Domain modeling constructs, but because it does not frequently you will have to use a combination of these two techniques.

Dealing with Entities

The Domain model for a module is usually a complex arrangements of relationships between entities. Often these relationships are one-to-many. Review the models for the Firewall (see page 21) and DHCP (see page 24) modules. The root of a Domain model is typically a single entity object, in DHCP this is the DhcpServer object. This object contains zero or more Scope entity objects. Each Scope object may contain zero or more Pool entity objects, dynamic and reservation pools are subtypes. Lastly, a ReservationPool object may contain zero or more HostReservation entity objects.

Entity Sets

When one entity contains an *unordered, one-to-many collection* of other entities, the Java `Set` collection should be used. The ConfigureServerSettings page includes an EntitySet to manipulate DHCP Scope entities. Figure 34 shows this table.



add a scope

name	subnet IP	subnet mask	IPs in scope	IPs used	IPs free		% used		
Employee Scope	10.1.14.0	255.255.255.0	200	176	23		83%	disable	delete
Visitor Scope	192.168.1.0	255.255.255.0	100	78	22		78%	disable	delete
Accounting Scope	10.1.15.0	255.255.255.0	254	127	127		50%	disable	delete
Development Scope	10.1.16.0	255.255.255.0	254	221	33		87%	disable	delete
Marketing Scope	10.1.25.0	255.255.255.0	100	64	36		64%	disable	delete
HR Scope	10.1.23.0	255.255.255.0	100	38	62		38%	disable	delete
Sales Scope	10.1.26.0	255.255.255.0	254	251	3		99%	disable	delete
Support Scope	10.1.31.0	255.255.255.0	100	73	27		73%	disable	delete

Figure 34: Example EntitSet from DHCP Module

Every element in this figure is rendered by a single UI widget, the EntitySet component, plus embedded Column component that generate the data columns in the table. Here is that JSP code:

```
<c:entitySet id='scopesTable' value='#{dhcpMgr.scopes}' var='scope'
        allowsAdd='true' allowsEdit='true' allowsDelete='true'>
  <c:column id='name'            value='#{scope.name}' />
  <c:column id='subnetIpAddress' value='#{scope.ethInterface.ipAddress}' />
  <c:column id='subnetMask' value='#{scope.ethInterface.netmask}' />
  <c:column id='ipsInScope' value='#{scope.totalIPs}' type='Numeric' />
  <c:column id='ipsUsed'    value='#{scope.usedIPs}'  type='Numeric' />
  <c:column id='ipsFree'    value='#{scope.freeIPs}'  type='Numeric' />
</c:entitySet>
```

This component is similar to the dataTable component in the standard JSF library. The purpose is to generate an HTML table that displays a collection of objects one in each row of the table. Furthermore data columns in the table display specific information from each object.

The EntitySet component takes that concept on step further by providing links that allow the user to add, edit, and delete (and undelete) entities. These features are enabled by special attributes on the `entitySet` tag:

- `allowsAdd` – determines if new entities can be added
- `allowsEdit` – determines if the entities can be edited
- `allowsDelete` – determines if the entities can be deleted and undeleted

The Column component must have `id` and `value` attributes. Optionally, this component may include a type attribute which provides styling hints. The possible values are:

- String – the value is a string (the default)
- Numeric – the value is a number
- Action – the value is a JSP link for an action (this is used internally)
- Image – the value is an image (not yet supported)
- Date – the value is a date

The EntitySet component includes these text attributes:

- itemName – the name of the type of entity; this is used to create the "add a XYZ" link
- emptyMessage – the text to be displayed when the table is empty; a default message is generated if this attribute is not specified

The Column component includes this attribute:

- columnLabel – the label for this column

Here is the complete set of text attributes for this example:

```
dhcp.ConfigureServerSettings.scopesTable.itemName=scope
dhcp.ConfigureServerSettings.scopesTable.emptyMessage=No DHCP scopes have
been added. Click <span class='reference'>add a scope</span> to add a scope
to the DHCP server.
dhcp.ConfigureServerSettings.scopesTable.name.columnLabel={!BASE.name}
dhcp.ConfigureServerSettings.scopesTable.subnetIpAddress.columnLabel=subnet
{!admin.IP.acronym}
dhcp.ConfigureServerSettings.scopesTable.subnetMask.columnLabel=subnet mask
dhcp.ConfigureServerSettings.scopesTable.ipsInScope.columnLabel={!admin.IPs.
acronym} in scope
dhcp.ConfigureServerSettings.scopesTable.ipsUsed.columnLabel={!admin.IPs.acr
onym} used
dhcp.ConfigureServerSettings.scopesTable.ipsFree.columnLabel={!admin.IPs.acr
onym} free
```

The EntitySet component requires the following backing bean members:

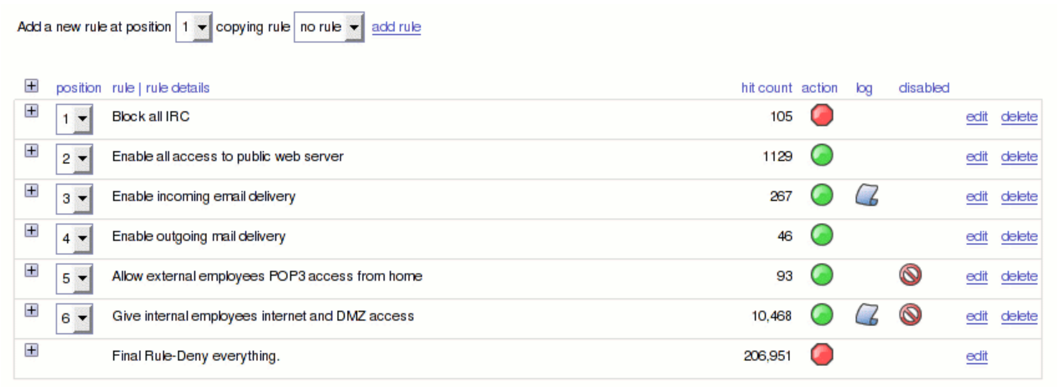
- `Set<XYZ> XYZsSet` – the instance variable that holds the Set collection of XYZ entities

  For example, if the entity type is `Scope`, then this might be named `scopesSet`.
- `List<XYZ> XYZs` – the instance variable that holds the sorted collection of XYZ entities; this is calculated from the set above
- `int XYZsIndex` – the instance variable that holds the index into the above List; this is used by the edit and delete JSF actions and is set by a hidden field
- `Set<XYZ> getXYZsSet()` – this method retrieves the set of entities
- `void setXYZsSet(Set<XYZ> var)` – this method stores the set of entities
- `void clearXYZs()` – this method clears the List instance variable
- `List<XYZ> getXYZs()` – this method retrieves the order list of entities
- `int getXYZsIndex()` – this method retrieves the index attribute
- `void setXYZsIndex(int var)` – this method set the index attribute; this is called automatically by the JSF hidden component
- `XYZ getXYZ()` – this method retrieves the entity at the specified index
- `String editXYZ()` – this method is the JSF action method used when the edit button is clicked by the user; it should use the `getXYZ` method to determine which entity to edit and should navigate to a newly created edit screen
- `String deleteXYZ()` – this method is the JSF action method used when the delete button is clicked by the user; it should use the `getXYZ` method to determine which entity to delete and should navigate back to the same screen
- `String undeleteXYZ()` – this method is the JSF action method used when the undelete button is clicked by the user; it should use the `getXYZ` method to determine which entity to undelete and should navigate back to the same screen
- `String addXYZ()` – this method is the JSF action method used when the add link is clicked by the user; it should create a new entity, set its state to ADDED, and then create an add (or edit) screen object and navigate to that screen
- `void addXYZ(XYZ var)` – this method is used by the add/edit screen when adding a new entity; this should only be called if the user saves the new entity (and not when the screen is canceled)

Take a look at the ConfigureServerSettings backing bean for an example of how these methods are implemented.

Entity Lists

When one entity contains an *ordered, one-to-many collection* of other entities, the Java `List` collection should be used. Lists keep positional information. The ViewFirewallRules page uses an EntityList component to manipulate the list of Firewall rules. Figure 35 shows this component.



Add a new rule at position 1 copying rule no rule add rule

	position	rule | rule details	hit count	action	log	disabled		
⊞	1	Block all IRC	105	🔴			edit	delete
⊞	2	Enable all access to public web server	1129	🟢			edit	delete
⊞	3	Enable incoming email delivery	267	🟢	▨		edit	delete
⊞	4	Enable outgoing mail delivery	46	🟢			edit	delete
⊞	5	Allow external employees POP3 access from home	93	🟢		🚫	edit	delete
⊞	6	Give internal employees internet and DMZ access	10,468	🟢	▨	🚫	edit	delete
⊞		Final Rule-Deny everything.	206,951	🔴			edit	

Figure 35: Example EntityList Component in the Firewall Module

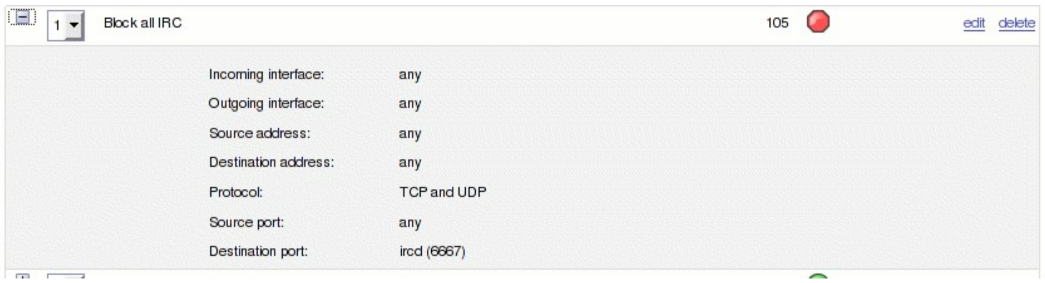
An EntityList is very similar to an EntitySet. Here is the JSP code for this example:

```
1.  <c:entityList id='fwRules' value='#{firewallBean.firewallRules}' var='rule'
2.       lastEntity='#{firewallBean.model.finalFirewallRule}'
3.       lastEntityEditAction='#{firewallBean.editLastRule}'
4.       allowsAdd='true' allowsEdit='true' allowsDelete='true' >
5.    <c:column id='description' value='#{rule.description}' />
6.    <c:column id='hitCount'    value='#{rule.hitCount}' type='Numeric' />
7.    <c:column id='action'      value='#{rule.action}'
8.       converter='converter.firewall.FirewallRuleAction' escape='false' />
9.    <c:column id='logEnabled'  value='#{rule.loggingEnabled}'
10.      converter='converter.firewall.FirewallRuleLogEnabled' escape='false' />
11.   <c:column id='status'      value='#{rule.enabled}'
12.      converter='converter.firewall.FirewallRuleDisabled' escape='false' />
13.   <f:facet name='entityDetails'>
14.     <c:formWrapper id='details'>
15.       <c:formOutput id='incomingInterface'  value='#{empty
rule.incomingInterface ? "any" : rule.incomingInterface}' />
16.       <c:formOutput id='outgoingInterface'  value='#{empty
rule.outgoingInterface ? "any" : rule.outgoingInterface}' />
17.       <c:formOutput id='sourceAddress'
value='#{rule.sourceAddress.displayString}' />
18.       <c:formOutput id='destinationAddress'
value='#{rule.destinationAddress.displayString}' />
19.       <c:formOutput id='protocol'
value='#{rule.protocol.displayString}' />
20.       <c:formOutput id='sourcePort'
value='#{rule.sourcePort.displayString}' />
21.       <c:formOutput id='destinationPort'
value='#{rule.destinationPort.displayString}' />
22.      </c:formWrapper>
23.    </f:facet>
24. </c:entityList>
```

The entityList tag has two attribute unique to lists:

- lastEntity – (optional) an entity that is always at the bottom of the list
- lastEntityEditAction – (optional) the JSF action method binding to edit this unique item

Also notice lines 13 through 23; this creates a JSF facet which alters the EntityList component. The `entityDetails` facet allows the UI developer to provide more details about each individual entity. These details are displayed in the hidden row below the entity row. Notice the plus symbol in the first column; this allows the user to open and close the entity details row. Figure 36 shows what this looks like.



1	Block all IRC	105	🔴	edit delete

Incoming interface:	any
Outgoing interface:	any
Source address:	any
Destination address:	any
Protocol:	TCP and UDP
Source port:	any
Destination port:	ircd (6667)

Figure 36: The `entityDetail` ***Facet***

This facet may be used by both EntityList and EntitySet components and it behaves the same way.

The EntityList component also requires a complex set of backing bean methods. Review the ViewFirewallRules backing bean code in the Firewall module to see what you need to implement.

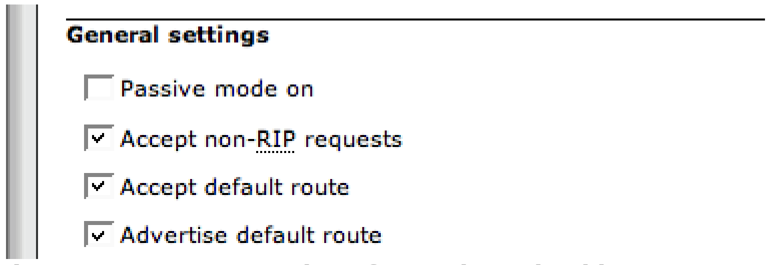
Dealing with Primitive Data

As I mentioned previously, there are eight primitive data types in Java: Boolean, byte, short, int, long, float, double, and char (character). In this section, we will look at two cases that are common in Cobia configuration pages: Boolean and integers.

Dealing with Boolean Values

There are two form widgets that input with Boolean values: BooleanCheckbox and BooleanRadio.

Figure 37 shows several examples of a BooleanCheckbox.



General settings

- [ ] Passive mode on
- [x] Accept non-RIP requests
- [x] Accept default route
- [x] Advertise default route

Figure 37: Four Examples of a BooleanCheckbox

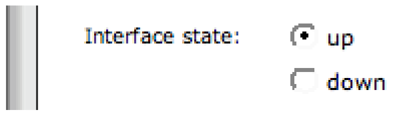
The component is a single HTML checkbox followed by a label. Here is the JSPX code for this portion of the EditRipInterface screen:

```
<c:booleanCheckbox id='passive' value='#{editRip.rip.passive}' />
<c:booleanCheckbox id='acceptNonRipRequests'
value='#{editRip.rip.acceptNonRipRequests}' />
<c:booleanCheckbox id='acceptDefaultRoute'
value='#{editRip.rip.acceptDefaultRoute}' />
<c:booleanCheckbox id='advertiseDefaultRoute'
value='#{editRip.rip.advertiseDefaultRoute}' />
```

This component includes two text attributes:

- Label – the text label for the widget
- helpText – (optional) the text used in a rollover help icon

Figure 38 shows an example of a BooleanRadio widget.



Interface state: (● up) (down)

Figure 38: An Example of a BooleanRadio

The component is a form label followed by two radio buttons stacked one on top of the other; the true state is the top button. Here is the JSPX code for this portion of the EditPhysicalInterface screen:

```
<c:booleanRadio id="interfaceStatus" value="#{editIntf.interface.status}"/>
```

This component requires three text attributes. Here are the attributes for this example:

```
admin.EditPhysicalInterface.interfaceStatus.Label=Interface state
admin.EditPhysicalInterface.interfaceStatus.true.Label=up
admin.EditPhysicalInterface.interfaceStatus.false.Label=down
```

Sometimes you need to display a Boolean value to the user, but you do not want to use the standard Java names: true and false. For example, you might want to display: up or down (the state of an Ethernet interface), enabled or disabled, on or off, and so on. Cobia provides a built-in JSF converter that allows you to display alternate text.

Here is a snippet of the JSP code in the ViewInterfaces page of the Admin module:

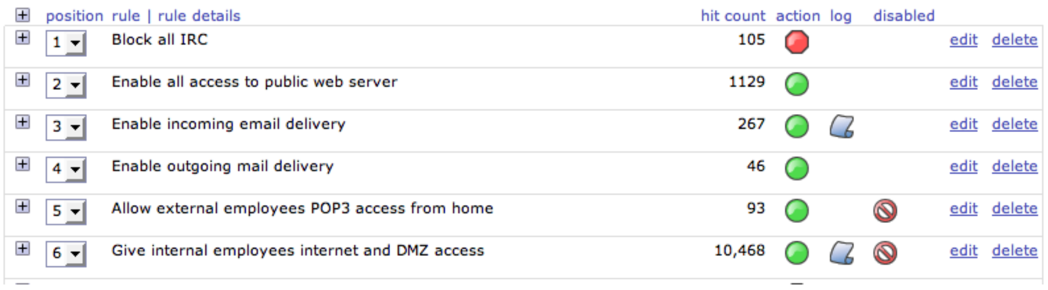
```
<c:entitySet id='intfTable' value='#{intfMgr.interfaces}' var='ethIntf'
            allowsAdd='false' allowsEdit='true' allowsDelete='false'>
  <c:column id='name'          value='#{ethIntf.name}'      sort='true' />
  <c:column id='state'         value='#{ethIntf.status}'    sort='true'>
```

```
      <ctags:booleanConverter type='UP/DOWN' />
   </c:column>
...
```

The status attribute is a Boolean value and the booleanConverter tag is converting this internal value to the string "up" or "down". There are four predefined *types* that you may use:

- STANDARD – the standard Java values: true or false
- YES/NO – answers a question: yes or no
- OP-MODE – the optionational mode: enabled or disabled
- UP/DOWN – the status of a device: up or down

You can also create your own converters to display icons that reflect a Boolean state that is unique the situation. Figure 39 shows an example in which an icon is displayed (or not displayed) for a firewall rule to be logged.



position	rule | rule details	hit count	action	log	disabled		
1	Block all IRC	105	🔴			edit	delete
2	Enable all access to public web server	1129	🟢			edit	delete
3	Enable incoming email delivery	267	🟢	✓		edit	delete
4	Enable outgoing mail delivery	46	🟢			edit	delete
5	Allow external employees POP3 access from home	93	🟢		🚫	edit	delete
6	Give internal employees internet and DMZ access	10,468	🟢	✓	🚫	edit	delete

Figure 39: Example Boolean Converters to Icons

Here is the JSP code for this column in the firewall rules EntityList table:

```
<c:column id='logEnabled'  value='#{rule.loggingEnabled}'
   converter='converter.firewall.FirewallRuleLogEnabled' escape='false' />
```

Here is the code for this converter:

```
public class RuleLogEnabledConverter implements Converter
{
    public final static String CONVERTER_ID =
"converter.firewall.FirewallRuleLogEnabled";

    public Object getAsObject(FacesContext ctx, UIComponent comp, String
displayString)
            throws ConverterException
    {
        assert false : "This converter is for display only.";
        return null;
    }

    public String getAsString(FacesContext ctx, UIComponent comp, Object
object) throws ConverterException
    {
        String result = null;

        if ( object == null )
        {
            result = HTML.NBSP_ENTITY;
        }
        else
        {
            Boolean ruleLogEnabled = (Boolean) object;
            if ( ruleLogEnabled )
                result = "<span class=\"logIcon\" title=\"Logging for this rule is enabled\">log</span>";
            else
                result = HTML.NBSP_ENTITY;
        }

        return result;
    }

}
```

You will need to declare this converter in your module's faces configuration file and you will need to defined the CSS styles for this class of span which identifies the icon. Here is the style definition for this example:

```
.logIcon {
   background: url(../images/icon_log.gif) top left no-repeat;
   display: block;
   height: 0;
   overflow: hidden;
   padding: 25px 7px 0 0;
   width: 18px;
}
```

Dealing with Integer Values

Figure 30 on page 55 shows a set of text fields that take integers as input values. Here are the JSP tags for the first field:

```
1. <c:textField id='routeExpirationTimeout' value='#{addRip.rip.routeTimeout}' size='4' required='true'>
2.   <f:converter converterId='converter.Integer' />
3.   <f:validateLongRange minimum='0' />
4. </c:textField>
```

The TextField component has three text attributes:

- Label – the label before the field itself
- helpText – the text displayed by the rollover help icon
- unitsLabel – the text after the field which is usually used to display the units of the value entered

Here are the text properties for this example:

```
router.EditRipInterface.routeExpirationTimeout.Label={!CAP:BASE.timeout}
router.EditRipInterface.routeExpirationTimeout.unitsLabel={!BASE.seconds}
```

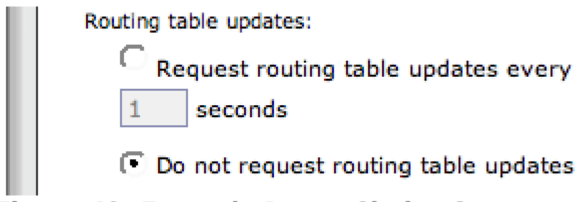
where these substitution variables are defined in the base text.properties file as:

```
BASE.seconds=seconds
BASE.timeout=timeout
```

The key to using a text field to store an integer to the domain model is using a JSF converter. Line 2 in the JSP code example above declares that this TextField component must use the Cobia built-in `converter.Integer` converter. JSF has its own built-in integer converter, but I had trouble with it in the past so I created one for Cobia and have not had any more trouble.

Occasionally, you might want to have an entity attribute that you can specify with a single text field, but you also want to give the user a choice of using your value or some predefined default value that have a semantic difference than specifying a value;

that is some values change the operational mode of the service. For example, in a RIP interface there is a parameter called a "routing table updates" which is usually specified in the number of seconds between updates, but if the value is set to zero then that tells RIP not to do updates at all. In this situation it is not very helpful to simply provide a text field and force the user to remember that zero is a special value. Instead Cobia provides the DatumChoice component which handles this situation. Figure 40 shows the RIP form for this field.


Routing table updates:

( ) Request routing table updates every

1 seconds

[✓ Do not request routing table updates]

Figure 40: Example DatumChoice Component

Here is the JSP code for this field:

```
<c:datumChoice id='requestInterval' value='#{editRip.rip.requestInterval}'
default='0' size='3'>
  <f:converter converterId='converter.Integer' />
  <f:validateLongRange minimum='0' />
</c:datumChoice>
```

The tag structure for this component is almost identical to a TextField component except that there is a `default` attribute which specifies the value to be used if the user selects the second radio button (the default).

The TextField component has three text attributes:

- Label – the label for the field itself
- helpText – (optional) the text displayed by the rollover help icon
- MAIN.Label – the text after the first radio button and before the field
- MAIN.unitsLabel – (optional) the text after the field which is usually used to display the units of the value entered
- DEFAULT.Label – the text for the second radio button

Here are the text properties for this example:

```
router.EditRipInterface.requestInterval.Label=Routing table updates
#router.EditRipInterface.requestInterval.helpText=
router.EditRipInterface.requestInterval.MAIN.Label=Request routing table
updates every
router.EditRipInterface.requestInterval.MAIN.unitsLabel={!BASE.seconds}
router.EditRipInterface.requestInterval.DEFAULT.Label=Do not request routing
table updates
```

Dealing with Enumerated Types

Enumerated types are used for a wide variety of situations. Here are a few examples in existing Cobia modules:

- subset of Internet protocols
- the action of a firewall rule
- the severity of an event

There are three HTML menu (or drop-down list) style components in Cobia that work well with enumerated types: DropDownList, ImageList and ColorList.

The DropDownList Component

Figure 41 shows a DropDownList component from the EditFirewallRule screen.

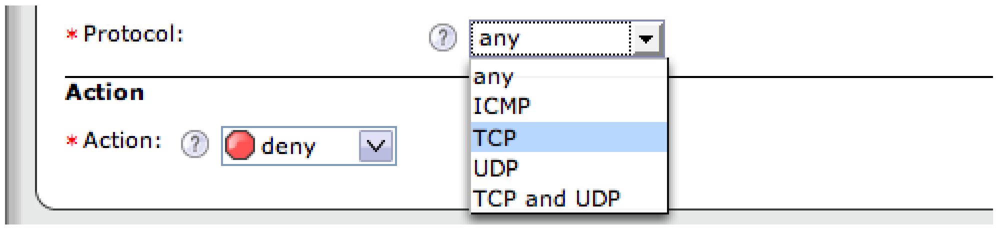


Figure 41: DropDownList with an Enumerated Type

The DropDownList component has these text attributes:

- Label – the label for the field itself
- helpText – (optional) the text displayed by the rollover help icon

Here is the JSP code for this example:

```
<c:dropDownList id='protocol' value='#{editRule.rule.protocol}'>
  <ctags:enumConverter
type='org.stillsecure.cobia.base.domain.values.Protocol' />
  <c:enumSelectItems
type='org.stillsecure.cobia.base.domain.values.Protocol' />
</c:dropDownList>
```

The DropDownList component is purpose component in the spirit of the UISelectOne JSF component. Therefore, the DropDownList component accepts an UISelectItems component. In this case, the Cobia UI Framework provides a built-in EnumSelectItems component which takes a `type` attribute to declare the fully-qualified Java class name for the enumerated type. This subcomponent generates the items that populate the drop-down list; it creates a mapping between the enum value and a user-friendly string. To accomplish this, the enum type must implement the `UiLocalizable` interface in the `org.stillsecure.cobia.web` package.

Here is the (abbreviated) code the Protocol enum type:

```
public enum Protocol implements UiLocalizable
{
    ANY, TCP, UDP, TCP_AND_UDP, ICMP;

    public String getLocalizedString()
    {
        return EnumUtils.getLocalizedString(this, BaseConstants.MODULE_ID);
    }
}
```

Cobia provides a `EnumUtils` class in the `org.stillsecure.cobia.web.util` package which has a generic implementation of the `getLocalizedString` method. This method performs a resource look-up for each enum value in a special resource bundle file which should be stored in the module's `src/properties/` directory called, `enum.properties`. The resource keys are based on the following template:

```
<moduleID>.<enumClassName>.<enumValue>
```

Here are the resource keys for the Protocol enum type in the Cobia base:

```

# org.stillsecure.cobia.base.domain.values.Protocol enum
```

```
BASE.Protocol.ANY=any
BASE.Protocol.TCP=TCP
BASE.Protocol.UDP=UDP
BASE.Protocol.TCP_AND_UDP=TCP and UDP
BASE.Protocol.ICMP=ICMP
```

Occasionally and enum type will be nested in another class that uses that type. Here is an example from the Router module:

```
public class RipInterface extends AbstractEntity
{
    public static enum Horizon implements UiLocalizable
    {
        NO_HORIZON,
        SPLIT_HORIZON_POISON_REVERSE,
        SPLIT_HORIZON;

        public String getLocalizedString()
        {
            return EnumUtils.getLocalizedString(this,
RouterConstants.MODULE_ID);
        }
    }
    // more code
}
```

In Java's notation the enum type class name is RipInterface$Horizon. The dollar sign is converted to a period when determining the resource key. Therefore, the enum.properties file would include these entries:

```
# RipInterface$Horizon enum
router.RipInterface.Horizon.NO_HORIZON=none
router.RipInterface.Horizon.SPLIT_HORIZON_POISON_REVERSE=Split Horizon
Poison Reverse
router.RipInterface.Horizon.SPLIT_HORIZON=Split Horizon
```

The ImageList Component

Figure 42 shows a ImageList component from the EditFirewallRule screen.

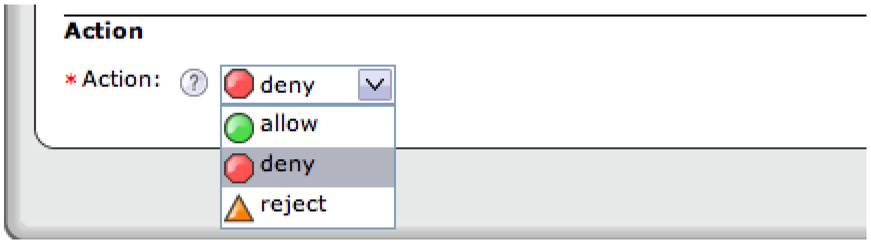


Figure 42: An Example ImageList Component

The ImageList component has these text attributes:

- Label – the label for the field itself
- helpText – (optional) the text displayed by the rollover help icon

Here is the JSP code for this example:

```
<c:imageList id='ruleAction' required='true'
     type='org.stillsecure.cobia.module.firewall.domain.Action'
     value='#{editRule.rule.action}' />
```

Unlike the DropDownList component, the ImageList only works with enum types and does not use SelectItems. The enum type must implement the `UiColor` interface in the `org.stillsecure.cobia.web` package.

Here is the code the Action enum type:

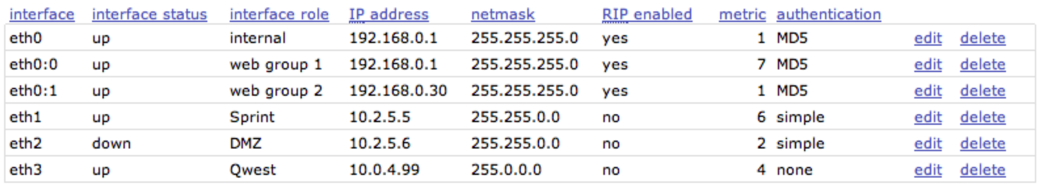
```java
public enum Action implements UiImage, UiLocalizable
{
    ALLOW, DENY, REJECT;

    public ImageData getDisabledImage()
    {
        return this.disabledImage;
    }

    public ImageData getHighlightedImage()
    {
        return this.highlightedImage;
    }

    public ImageData getPrimaryImage()
    {
        return this.primaryImage;
    }

    public String getLocalizedString()
    {
        return EnumUtils.getLocalizedString(this,
FirewallConstants.MODULE_ID);
    }

    //
    // Private
    //

    private Action()
    {
        String image_name = this.name().toLowerCase();
        String image_file = null;

        image_file = String.format("/%s/images/icon_%s_dropdown.gif",
                                   FirewallConstants.MODULE_ID, image_name);
        this.primaryImage = new UiImage.ImageData(image_file, 15, 15);

        image_file = String.format("/%s/images/icon_%s_dropdown_hi.gif",
                                   FirewallConstants.MODULE_ID, image_name);
        this.highlightedImage = new UiImage.ImageData(image_file, 15, 15);

        image_file = String.format("/%s/images/icon_%s_dropdown_dis.gif",
                                   FirewallConstants.MODULE_ID, image_name);
        this.disabledImage = new UiImage.ImageData(image_file, 15, 15);
    }

    private ImageData primaryImage;
    private ImageData highlightedImage;
    private ImageData disabledImage;
}
```

Data conversion is also handled directly by the component and does not require a converter.

Displaying Enum Values as Text and Icons

Occasionally you will need to display an enum value as either text or as an icon.

Figure 43 shows an example of an entity table that displays an enum as text.



interface	interface status	interface role	IP address	netmask	RIP enabled	metric	authentication		
eth0	up	internal	192.168.0.1	255.255.255.0	yes	1	MD5	edit	delete
eth0:0	up	web group 1	192.168.0.1	255.255.255.0	yes	7	MD5	edit	delete
eth0:1	up	web group 2	192.168.0.30	255.255.255.0	yes	1	MD5	edit	delete
eth1	up	Sprint	10.2.5.5	255.255.0.0	no	6	simple	edit	delete
eth2	down	DMZ	10.2.5.6	255.255.0.0	no	2	simple	edit	delete
eth3	up	Qwest	10.0.4.99	255.0.0.0	no	4	none	edit	delete

Figure 43: Displaying Enum Values as Text

Here is the JSP code for the authentication column in this table:

```
<c:column id='authentication' value='#{rip.authentication.type}'>
  <ctags:enumConverter forDisplay='true'
type='org.stillsecure.cobia.module.router.domain.RipAuthentication$Type' />
</c:column>
```

The critical element is the use of the `forDisplay` attribute in the `enumConverter` tag. This tells the converter to use the localized string of the enum value when converting from the enum value to text.

Figure 39 on page 72 shows an example of an entity table that displays an enum as an icon. Here is the JSP code for the action column in this table:

```
<c:column id='action' value='#{rule.action}'
    converter='converter.firewall.FirewallRuleAction' escape='false' />
```

This column specifies a module-specific converter and the escape attribute is false which indicates that the rendered text is HTML code that should not be XML-escaped. Here is the code for this converter:

```
public class RuleActionConverter implements Converter
{
    public final static String CONVERTER_ID =
"converter.firewall.FirewallRuleAction";

    public Object getAsObject(FacesContext ctx, UIComponent comp, String
displayString)
            throws ConverterException
    {
        assert false : "This converter is for display only.";
        return null;
    }

    public String getAsString(FacesContext ctx, UIComponent comp, Object
object) throws ConverterException
    {
        String result = null;

        if ( object == null )
        {
            return "";
        }

        try
        {
            Action action = (Action) object;
            switch (action)
            {
            case ALLOW:
            {
```

```
28.                 result = "<span class=\"allowIcon\" title=\"Allow packets
    that match this rule\">allow</span>";
29.                 break;
30.             }
31.             case DENY:
32.             {
33.                 result = "<span class=\"denyIcon\" title=\"Deny packets that
    match this rule\">deny</span>";
34.                 break;
35.             }
36.             case REJECT:
37.             {
38.                 result = "<span class=\"rejectIcon\" title=\"Reject packets
    that match this rule\">reject</span>";
39.                 break;
40.             }
41.             default:
42.                 throw new ClassCastException();
43.             }
44.         }
45.         catch (ClassCastException e)
46.         {
47.             FacesMessage msg
48.                 =
    MessageUtils.createFacesMessage(RULE_ACTION_CONVERTER_ERROR2,
    object.toString());
49.             throw new ConverterException(msg);
50.         }
51.         return result;
52.     }
53.     private static final String RULE_ACTION_CONVERTER_ERROR2 =
    "firewall.RuleActionConverter.getAsStringFailed";
54. }
```

Like the icon-converter for boolean values, you will need to specify the CSS styles for these span classes (on lines 28, 33 and 38) and create the necessary icons. Typically, these icons will be the same as the "active" icons defined by the ImageData elements in the UiColor implementation within the enum type itself.

Dealing with Single-String Representation Types

Strings are used for a wide variety of purposes: names, descriptions, passwords and so on. There are three widgets in the Cobia UI Framework for dealing with strings: TextField, SecretField and TextArea. Here are code snippets from JSP pages for each of these three component types:

```
<c:textField id='description' required='true' size='25'
  converter='converter.TrimString'
  value='#{editRule.rule.description}'
  validator='#{editRule.validateRuleName}' />
```

```
<c:textArea id='description' cols='40' rows='3'
   value='#{editReservation.reservation.description}' />
```

```
<c:secretField id='currentUIPwd' required='true'
  value='#{editPassword.currentUIPwd}'
  validator="#{editPassword.validateCurrentUIPassword}" />
```

These components have these text attributes:

- Label – the label for the field itself

- helpText – (optional) the text displayed by the rollover help icon

The TextField component has one more text attribute:

- unitsLabel – the text after the field

In the first example, notice the use of the TrimString converter. This converter takes a string input and calls the trim function on the string to remove any blank characters surrounding any non-blank characters. Using this converter prevents a user from entering blank characters for a name.

Strings are also used to provide a user-friendly representation of a simple value object; see the Value Object Model Elements section on page 58. Two UI widgets are useful to handle the input of these types of value objects: TextField and ChoiceTextField. Here is an example of a TextField that input an IPv4Address object:

```
<c:textField id='ipAddress' required='true'
  converter='converter.Ipv4Address' size='15'
  value='#{editReservation.reservation.ipAddress}'
  validator='#{editReservation.validateIpAddress}' />
```

The critical element of using either of these components with a value object in the domain model is the use of a converter. These converter can be pretty easy to build if the value object class includes a constructor that takes a `String` object as a parameter and has a `toString` method to recreates the string presentation.

Here is the (partial) code for the Ipv4Address value object class:

```
public class Ipv4Address extends IpAddress
{
    public final long address;

    public Ipv4Address( String address ) throws IllegalArgumentException
    {
        int shift = 24;
        long addr = 0;
        for( String token : address.split( "\\." ) )
        {
            long next = Integer.parseInt( token );
            if( next < 0 || next > 255 )
                throw new IllegalArgumentException( "Invalid IPv4 address: "
+ address );
            addr |= next << shift;
            shift -= 8;
        }
        if( shift != -8 || address.endsWith( "." ) )
            throw new IllegalArgumentException( "Invalid IPv4 address: " +
address );

        this.address = addr;
    }

// MORE CODE

    @Override
    public String toString()
    {
        return ( ( address >> 24 ) & 0xff ) + "." + ( ( address >> 16 ) &
0xff ) + "."
                + ( ( address >> 8 ) & 0xff ) + "." + ( address & 0xff );
    }
}
```

Here is the code for the Ipv4Address converter:

```
1.  public class Ipv4AddressConverter implements Converter
2.  {
3.      /** The JSF ID for this converter.  The ID must be the same as
4.       * the ID defined in the framework-faces-config.xml file. */
5.      public final static String CONVERTER_ID = "converter.Ipv4Address";
6.  
7.      /**
8.       * This method converts the string representation to an Ipv4Address object.
9.       */
10.     public Object getAsObject(FacesContext ctx, UIComponent comp, String displayString) throws ConverterException
11.     {
12.         Ipv4Address result = null;
13.         try
14.         {
15.             if (displayString != null && displayString.trim().length() > 0)
16.             {
17.                 result = new Ipv4Address(displayString);
18.             }
19.         }
20.         catch (IllegalArgumentException e)
21.         {
22.             FacesMessage message
23.                 = MessageUtils.createFacesMessage(IPv4ADDRESS_CONVERTER_TO_OBJECT_FAILED, displayString);
24.             throw new ConverterException(message);
25.         }
26.         return result;
27.     }
28.     private static final String IPv4ADDRESS_CONVERTER_TO_OBJECT_FAILED = "BASE.Ipv4AddressConverter.getAsObjectFailed";
29. 
30.     /**
31.      * This method converts the Ipv4Address object to the string representation.
32.      */
33.     public String getAsString(FacesContext ctx, UIComponent comp, Object object) throws ConverterException
34.     {
35.         String result = null;
36.         if ( object == null ) return "";
37.         try
38.         {
39.             Ipv4Address i = (Ipv4Address) object;
40.             result = i.toString();
41.         }
42.         catch (ClassCastException e)
43.         {
44.             assert false : e;
45.         }
46.         return result;
47.     }
48. }
```

When creating your own value object converters it is important that you consider the case of `null` values. The conditional statement on line 15 prevents the converter from attempting to converter an empty string, which fails in the object constructor. Similarly, line 36 returns an empty string of the `object` parameter is `null`.

Lastly, notice the `getAsObject` method executes the Ipv4Address constructor in a try-catch block. This is used to catch any `IllegalArgumentException`s that might be thrown by the constructor. The catch block throws a JSF `ConverterException` and uses the Cobia `MessageUtils` facility to create the necessary JSF message object using a resource key. This key is looked up in the `text.properties` resource bundle. Here is the text for this example:

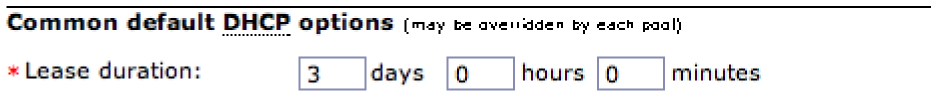
```
BASE.Ipv4AddressConverter.getAsObjectFailed=Converter Error: Invalid IPv4
address ''{0}''.
```

The `{0}` argument is the value entered by the user.

Dealing with Multiple-String Representation Types

Sometimes a value object cannot easily be represented with a single string in a TextField. Therefore, we created the MultiTextField component. This component creates a field that contains two or more text fields. The combined data is used to construct a single value object.

Figure 44 shows an example from the DHCP ConfigureServerOptions screen.



Common default DHCP options (may be overridden by each pool)

* Lease duration: 3 days 0 hours 0 minutes

Figure 44: Example MultiTextField Component

Here is the JSP code for this example:

```
1. <c:multiTextField id='leaseDuration' elements='days:3,hours:3,minutes:3'
2.      value='#{globalOptionsMgr.leaseDuration}'
3.      required='true' converter='converter.dhcp.Time'>
4.   <f:validateLongRange minimum='1' />
5. </c:multiTextField>
```

The critical part of this component is the `elements` tag attribute. This is a comma-delimited list of field IDs and the sizes of each field. These field IDs are critical for the data conversion which we will discuss in detail below.

The `leaseDuration` attribute of the backing bean is stored as a single integer, the number of seconds. Here are the accessor methods in the backing bean:

```
    public int getLeaseDuration()
    {
        return leaseDuration;
    }
    public void setLeaseDuration(int leaseDuration)
    {
        this.leaseDuration = leaseDuration;
    }
```

It is the responsibility of the `converter.dhcp.Time` converter to split the integer value into three elements, days, hours and minutes; and also convert the three elements back into a single integer value. The key to this type of conversion is that JSON (the JavaScript Object Notation) is used to pass multiple values to the client. For example, 1 day / 2 hours / 3 minutes is represented as "`{days:1 , hours:2 , minutes:3 }`" which is converted to 93780 seconds. The MultiTextField component generates JavaScript code to parse this JSON object and place each data element into the appropriate field based upon the field identifiers in the `elements` tag attribute (line 1). There is also JavaScript code that reconstructs the JSON string which is sent back to the web container to be handled by the JSF framework.

Here is the code for the time converter:

```java
public class TimeConverter implements Converter
{
    /** The JSF id for this converter.  Must be the same as defined in the module-faces-config.xml file. */
    public final static String CONVERTER_ID = "converter.dhcp.Time";

    public final static int SECONDS_IN_MINUTE = 60;
    public final static int SECONDS_IN_HOUR = SECONDS_IN_MINUTE * 60;
    public final static int SECONDS_IN_DAY = SECONDS_IN_HOUR * 24;

    public Object getAsObject(FacesContext ctx, UIComponent comp, String displayString) throws ConverterException
    {
        int result;

        // Extract time elements from the JSON object
        JSONObject jsonObject = JSONObject.fromString(displayString);
        int days = getValue(jsonObject, DAYS_KEY, "BASE.days", -1);
        int hours = getValue(jsonObject, HOURS_KEY, "BASE.hours", 23);
        int minutes = getValue(jsonObject, MINUTES_KEY, "BASE.minutes", 59);

        // Create result in seconds
        result = (days * SECONDS_IN_DAY) + (hours * SECONDS_IN_HOUR) + (minutes * SECONDS_IN_MINUTE);

        return (Integer) result;
    }

    private int getValue(JSONObject jsonObject, String fieldKey, String fieldNameKey, int max)
    {
        int value = 0;
        try
        {
            value = jsonObject.getInt(fieldKey);
            if ( value < 0 )
            {
                String fieldName = new Resource(fieldNameKey).toString();
                FacesMessage message
                    = MessageUtils.createFacesMessage(TIME_CONVERTER_VALUE_NOT_NEGATIVE, fieldName);
                throw new ConverterException(message);
            } else if ( max != -1 && value > max )
            {
                String fieldName = new Resource(fieldNameKey).toString();
                FacesMessage message
                    = MessageUtils.createFacesMessage(TIME_CONVERTER_VALUE_NOT_IN_RANGE, fieldName, max);
                throw new ConverterException(message);
            }
        }
        catch (JSONException e)
        {
            String fieldName = new Resource(fieldNameKey).toString();
            FacesMessage message
                = MessageUtils.createFacesMessage(TIME_CONVERTER_VALUE_NOT_AN_INT, fieldName);
            throw new ConverterException(message);
        }
        return value;
    }
    private static final String TIME_CONVERTER_VALUE_NOT_AN_INT = "dhcp.TimeConverter.valueNotAnInt";
    private static final String TIME_CONVERTER_VALUE_NOT_NEGATIVE = "dhcp.TimeConverter.valueIsNegative";
    private static final String TIME_CONVERTER_VALUE_NOT_IN_RANGE = "dhcp.TimeConverter.valueNotInRange";

    /**
     * This method converts from the Integer object to the string representation.
     */
    public String getAsString(FacesContext ctx, UIComponent comp, Object object) throws ConverterException
    {
        Integer seconds = (Integer) object;

        // Calculate time elements
        int days = seconds / SECONDS_IN_DAY;
        int daysRemainder = seconds - (days * SECONDS_IN_DAY);
        int hours = daysRemainder / SECONDS_IN_HOUR;
        int minutes = (daysRemainder - (hours * SECONDS_IN_HOUR)) / SECONDS_IN_MINUTE;

        // Create the JSON object with these elements
        JSONObject jsonObject = new JSONObject();
        jsonObject.put(DAYS_KEY, days);
        jsonObject.put(HOURS_KEY, hours);
        jsonObject.put(MINUTES_KEY, minutes);

        return jsonObject.toString();
    }

    static final String DAYS_KEY = "days";
    static final String HOURS_KEY = "hours";
    static final String MINUTES_KEY = "minutes";
}
```

The text attributes of this component are a little more complex than other components. It does include the Label and helpText attributes, but it also includes optional pre- and post-labels for each element. Here are the text attributes for this example:

```
dhcp.ConfigureServerOptions.leaseDuration.Label=Lease duration
dhcp.ConfigureServerOptions.leaseDuration.days.postLabel={!BASE.days}
dhcp.ConfigureServerOptions.leaseDuration.hours.postLabel={!BASE.hours}
dhcp.ConfigureServerOptions.leaseDuration.minutes.postLabel={!BASE.minutes}
```

Here is a another example. Figure 45 shows an example from the Firewall module.

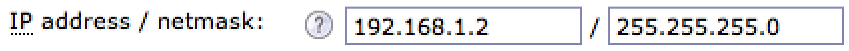

IP address / netmask: 192.168.1.2 / 255.255.255.0

Figure 45: Another MultiTextField Example

This example has two elements: `ipAddress` and `netmask`. The converter used for this example is the built-in `converter.IpNetworkAddress` converter. Here is the code for this converter:

```
public class IpNetworkAddressConverter implements Converter
{
    /** The JSF ID for this converter.  The ID must be the same as
     * the ID defined in the module-faces-config.xml file. */
    public final static String CONVERTER_ID = "converter.IpNetworkAddress";
```

```java

    public Object getAsObject(FacesContext ctx, UIComponent comp, String displayString) throws ConverterException
    {
        if ( displayString.trim().length() == 0 ) return null;

        JSONObject jsonObject = JSONObject.fromString(displayString);

        // Convert IP address
        String ipAddrStr = jsonObject.getString(IP_ADDRESS_KEY);
        IpAddress ipAddress;
        try
        {
            ipAddress = IpAddress.parse(ipAddrStr);
        }
        catch (IllegalArgumentException e)
        {
            FacesMessage message
                = MessageUtils.createFacesMessage(BAD_IP_ADDRESS_MSG, ipAddrStr);
            throw new ConverterException(message);
        }

        // Convert netmask
        String netmaskStr = jsonObject.getString(NETMASK_KEY);
        NetMask netmask;
        try
        {
            netmask = new NetMask(netmaskStr);
        }
        catch (IllegalArgumentException e)
        {
            FacesMessage message
                = MessageUtils.createFacesMessage(BAD_NETMASK_MSG, netmaskStr);
            throw new ConverterException(message);
        }

        return new IpNetworkAddress(ipAddress, netmask);
    }
    private static final String BAD_IP_ADDRESS_MSG = "BASE.Ipv4AddressConverter.getAsObjectFailed";
    private static final String BAD_NETMASK_MSG = "BASE.NetmaskConverter.getAsObjectFailed";

    public String getAsString(FacesContext ctx, UIComponent comp, Object object) throws ConverterException
    {
        if ( object == null ) return "";

        IpNetworkAddress network = (IpNetworkAddress) object;

        // Create the JSON object with these elements
        JSONObject jsonObject = new JSONObject();
        jsonObject.put(IP_ADDRESS_KEY, network.getAddress().toString());
        jsonObject.put(NETMASK_KEY, network.getNetmask().toString());

        return jsonObject.toString();
    }

    static final String IP_ADDRESS_KEY = "ipAddress";
    static final String NETMASK_KEY = "netmask";
}
```

Dealing with Data Collections

If your Domain model includes an attribute that contains multiple values of some primitive or value object, there are special UI widgets to deal with these.

Dealing with Lists or Sets of Values

Figure 46 shows the ValueAccumulator component.

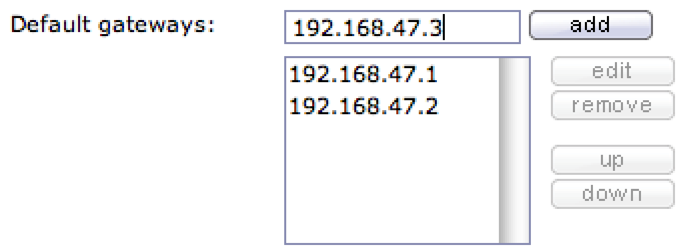


Figure 46: Example ValueAccumulator Component

The component uses a lot of JavaScript code to manipulate an internal HTML select element using the add, edit, remove, up and down buttons. Values are added from the text field either when the user types the Enter key or when the add button is clicked. The user can edit a specific item either by double-clicking the item or by single-clicking the item and then clicking the edit button; this removes the item from the list and places it into the text field which can then be edited and re-added. The user can remove an item by clicking the item and clicking the remove button.

If the list is ordered, the up/down buttons are displayed. The user single-clicks the element to move and then clicks either the up or down button to move the item within the list.

Here is the JSP code for this example from the DHCP ConfigureServerOptions screen:

```
<c:valueAccumulator id='defaultGateways' listHeight='6'
     listOrderable='true' textFieldSize='15'
     value='#{globalOptionsMgr.defaultGatewaysString}'
     validator='#{globalOptionsMgr.validateDefaultGateways}' />
```

By default, this component transfers the list of items using a comma-delimited string. Conversion must be done by the setter method in the backing bean or domain entity. The validation of the items themselves must be done by a developer-supplied validator method which must also parse the comma-delimited string.

Dealing with Ranges

Figure 47 shows an example of a MultiTextField used to input a value range.

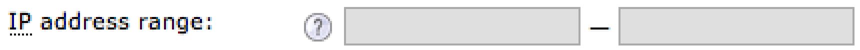
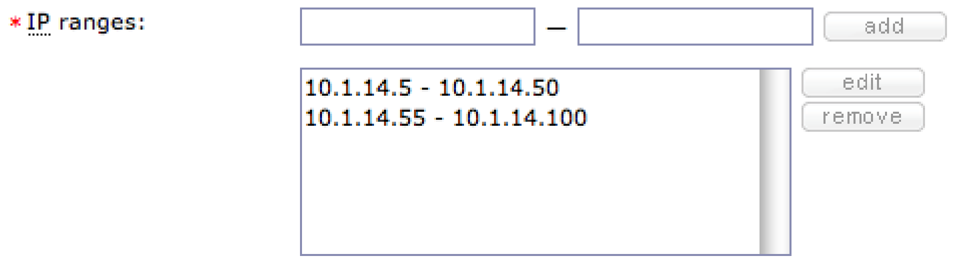


Figure 47: Example MultiTextField for a Range Value

The Cobia UI Framework provides a built-in converter to handle this generic type of data collection: `converter.Range`. When using this converter the elements of the MultiTextField must be named `start` and `end`. This converter must include an embedded converter that is used to converter the individual elements.

Dealing with Lists of Ranges

Figure 48 shows an example of a RangeAccumulator component.



10.1.14.5 - 10.1.14.50
10.1.14.55 - 10.1.14.100

Figure 48: Example RangeAccumulator Component

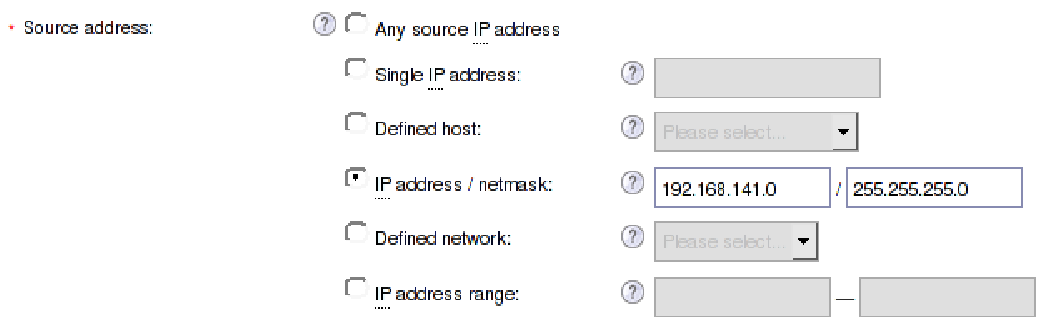
Here is the JSP code for this example from the DHCP ConfigureServerOptions screen:

```
<c:rangeAccumulator id='ipRanges' required='true' dataType='string'
     listHeight='5' textFieldSize='16'
     value='#{editReservationPool.ipRangesString}'
     validator='#{editReservationPool.validateIpRanges}'/>
```

Like the ValueAccumulator component, this component transfers the list of items using a comma-delimited string of start/end values separated by the dash (-) character. Conversion must be done by the setter method in the backing bean or domain entity. The validation of the items themselves must be done by a developer-supplied validator method which must also parse the comma-delimited string.

Dealing with Union Types

Union types are the most complex of the value object types used in Cobia. Figure 49 shows an example union type field in the Firewall module. Union type widgets are also the most complicated widgets to use because it requires additional work to create the specification of the mapping between the UI widget and the actual union type value object. The details of this mapping are beyond the scope of this document; you must analyze the code of the examples discussed below to figure out how to use this component.



Source address:

( ) Any source IP address
( ) Single IP address:
( ) Defined host: Please select...
(•) IP address / netmask: 192.168.141.0 / 255.255.255.0
( ) Defined network: Please select...
( ) IP address range: —

Figure 49: Example UnionTypeField Component

Here is the JSP code for this example:

```
<c:unionTypeField id='sourceAddress' required='true'
     value='#{editRule.rule.sourceAddress}'
     type='firewall.IpAddressMatchingSpec'
     elementOrder='ANY, HOST, USER_DEFINED_HOST, NETWORK,
USER_DEFINED_NETWORK, RANGE' />
```

The type attribute declares the symbolic name for the union type specification for this field and the elementOrder declares the order in which the union type "types" are presented on the page.

The Union Type Specification

Every UnionTypeField component requires a *union type specification* which is an object that maps between the UI widget and the actual union type object. The type attribute is a symbolic name that maps to a Java class name to implements this specification object. Figure 50 shows an abstract representation of the relationship between the UI widget, the specification, and the actual union type value object.

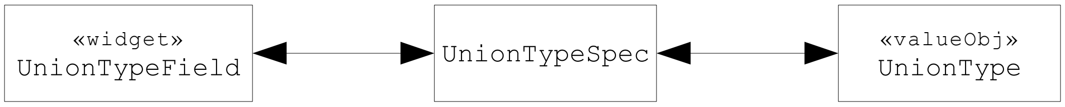


Figure 50: The Abstract Relationship between the Widget and the Value Object

Any module that requires a UnionTypeField must supply the unionTypeSpec.properties file in the modules src/properties/ directory. Here is this file for the Firewall module:

```
firewall.IpAddressMatchingSpec=org.stillsecure.cobia.module.firewall.web.IpA
ddressMatchingSpecification
firewall.IpPortMatchingSpec=org.stillsecure.cobia.module.firewall.web.IpPort
MatchingSpecification
firewall.UiAddressTranslation=org.stillsecure.cobia.module.firewall.web.Addr
essTranslationSpecification
firewall.UiPortTranslation=org.stillsecure.cobia.module.firewall.web
.PortTranslationSpecification
```

This is a mapping between the symbolic name used by the widget's type attribute and the specification class. The specification class must extend the UnionTypeSpecification abstract class from the org.stillsecure.cobia.web package. Figure 51 shows a specific example of this relationship.

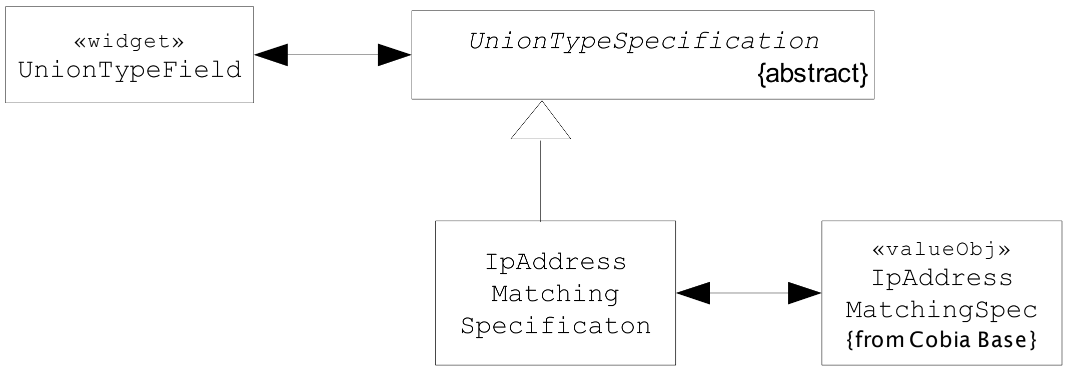


Figure 51: Specific Example of the UnionTypeSpecification Mapping

Every union type specification class must implement these methods:

- `getTypes` – this method returns the complete set of enum values of the union type's `Type` enum
- `createUIComponent` – this method creates the UI component used to input the value for a specific "type" of the union type object
- `makeSetFunction` – this method generates the JavaScript code to set the UnionTypeField's hidden input field from the selected radio button and its individual data entry component
- `makeInitFunction` – this method generates the JavaScript code to initialize the radio button and its data entry component from the UnionTypeField's hidden input field; these two JS functions are the inverse of each other
- `convertToObject` – this method converts a JSON string from the UnionTypeField's hidden input field to an actual union type object
- `convertToString` – this method converts an actual union type object to the JSON string which is then stored in the UnionTypeField's hidden input field
- `validateObject` – this method allows generic validation of the converted union type object; frequently this method is not required if the `convertToObject` method handles all of the validation

The best way to learn how to build one of these is to analyze an example. There are four examples in the Firewall module. The `IpAddressMatchingSpecification` and the `PortTranslationSpecification` classes are the best examples.

The UnionTypeField Component

Figure 49 on page 90 shows an example union type field in the Firewall module.

The text attributes of this field require labels for each union type's type. Here is the example from the Firewall module:

```
firewall.EditFirewallRule.sourceAddress.Label=Source address
firewall.EditFirewallRule.sourceAddress.helpText=help text...
firewall.EditFirewallRule.sourceAddress.ANY.Label=Any source
{!admin.IP.acronym} address
#firewall.EditFirewallRule.sourceAddress.ANY.helpText=
firewall.EditFirewallRule.sourceAddress.HOST.Label=Single
{!admin.IP.acronym} address
firewall.EditFirewallRule.sourceAddress.HOST.host.helpText=help text...
firewall.EditFirewallRule.sourceAddress.USER_DEFINED_HOST.Label=Defined host
firewall.EditFirewallRule.sourceAddress.USER_DEFINED_HOST.definedHost.helpTe
xt=help text...
```

```
firewall.EditFirewallRule.sourceAddress.NETWORK.Label={!admin.IP.acronym}
address / netmask
firewall.EditFirewallRule.sourceAddress.NETWORK.network.helpText=help
text...
firewall.EditFirewallRule.sourceAddress.NETWORK.network.ipAddress.postLabel=
/
firewall.EditFirewallRule.sourceAddress.USER_DEFINED_NETWORK.Label=Defined
network
firewall.EditFirewallRule.sourceAddress.USER_DEFINED_NETWORK.definedNetwork.
helpText=help text...
firewall.EditFirewallRule.sourceAddress.RANGE.Label={!admin.IP.acronym}
address range
firewall.EditFirewallRule.sourceAddress.RANGE.range.helpText=help text...
firewall.EditFirewallRule.sourceAddress.RANGE.range.ipStart.postLabel=&mdash
;
```

Notice that there are three layers of the component ID portion of the text key. For example, the `sourceAddress` is the ID of the UnionTypeField itself; the `NETWORK` is the ID of the UnionTypeElement component is the second layer; and the `network` is the ID of the MulitTextField component is the third layer. The IDs of the second layer are the names of the values of the union type's `Type` enum. The IDs of the third layer are the IDs of the subcomponents defined in the `createUIComponent` method in the specification class.

UnionTypeField components also require special CSS styles. Here are the style definitions for this example:

```
div#editFirewallRule div#sourceAddressOptions { /**/
   margin: 0 0 0 18em;
   padding: 0;
}
div#editFirewallRule div#sourceAddressOptions label { /**/
   width: 15em;
}
```

The first definition declares the margin that places the radio buttons in-line with the other form fields on this page. The second definition declares the width of the labels after the radio buttons and before the embedded form widgets for each radio button. The “Options” portion of the `div` ID is added by the UnionTypeField component when the component is rendered to HTML.

The UnionType Component

There is another variation on this component, called the UnionType or *union type section*. Figure 52 shows the only existing example of this component from the Router module, in the EditRipInterface screen.

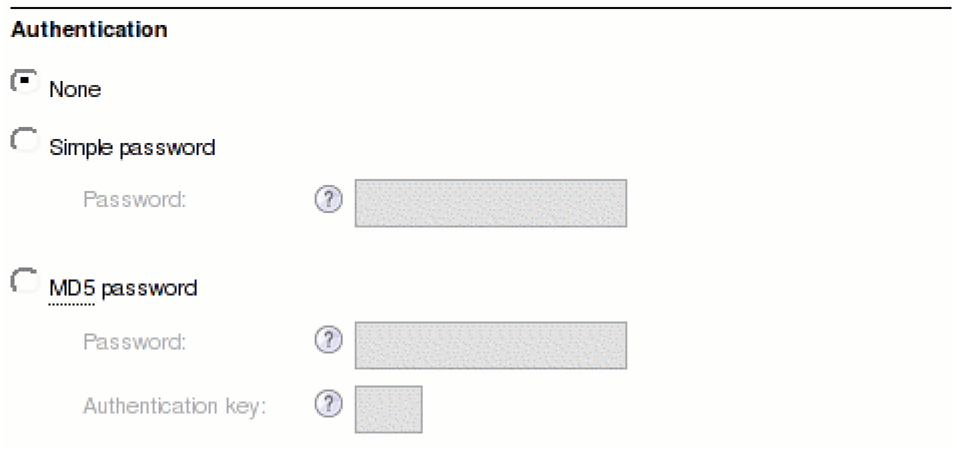


Figure 52: Example UnionType Component

This component is similar to a UnionTypeField except that it is not rendered as a field but rather as a separate section in the form. It includes its own SectionHeading (see Creating Sections on page 53). After each radio button (one for each type of the union type) a SubSection component is used. This component also uses a union type specification but the code for this type of component is a little more complex than specifications for UnionTypeField components. **NOTE**: specifications for these two types of union type widgets are not compatible.

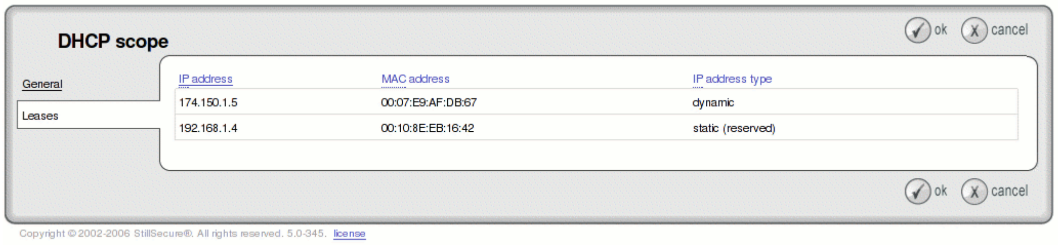
Review the code in the `RipAuthenticationSpecification` class in the Router module for an example of this type of specification.

View and Search Pages

This section describes how to create view and search screens.

The Purpose of View Pages

View pages are used to display dynamic data. This data usually comes from a database. Figure 53 shows an example view page from the DHCP module.



DHCP scope

General

Leases

IP address	MAC address	IP address type
174.150.1.5	00:07:E9:AF:DB:67	dynamic
192.168.1.4	00:10:8E:EB:16:42	static (reserved)

Copyright © 2002-2006 StillSecure®. All rights reserved. 5.0-345. license

Figure 53: Example View Page from the DHCP Module

View pages are like search pages, but the search criterion is predefined by the context of the page. For example, the DHCP scope leases is the search for leases for a specific DHCP scope. View screens should only be used when the expected result set is relatively small because the complete result set will be displayed on one page.

The Structure of View Pages

Typically, a view page can use an EntitySet UI widget to display the data. Here is the JSP code for this example:

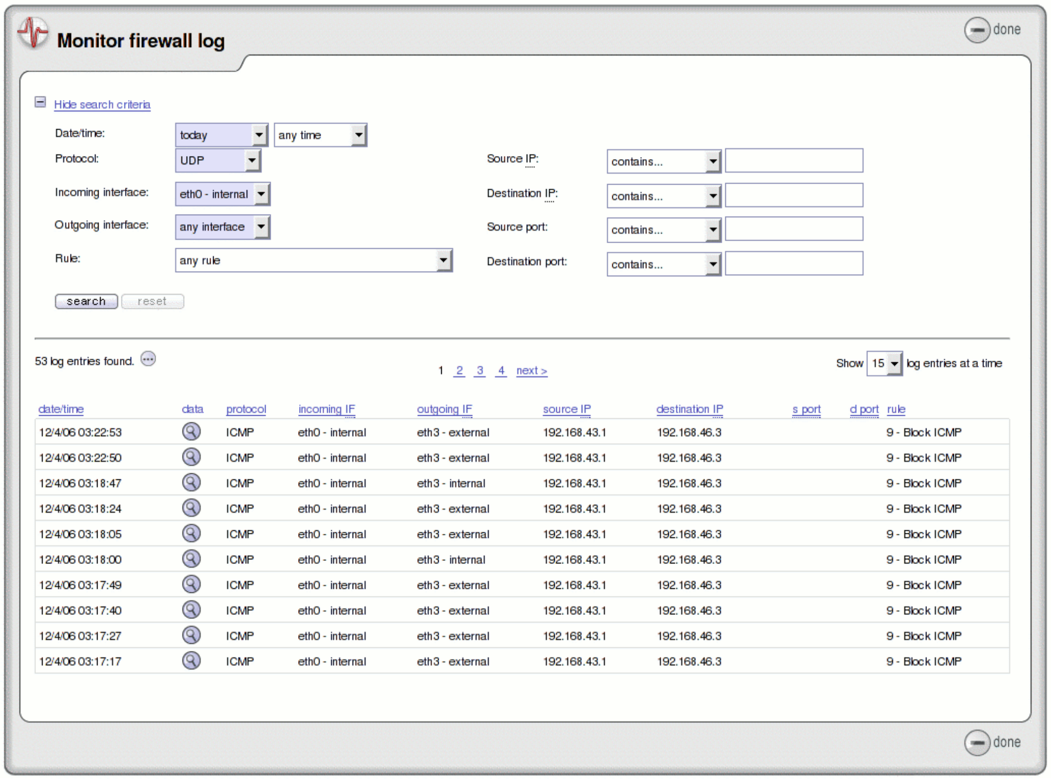
```
<c:entitySet var="dhcpCurrent" value="#{editScope.scopedel.leases}"
     id="leases" allowsAdd="false" allowsDelete="false" allowsEdit="false">
  <c:column value="#{dhcpCurrent.app_dhcp_cur_ipaddress}" id="ipAddress" />
  <c:column value="#{dhcpCurrent.app_dhcp_cur_macaddress}" id="macAddress"/>
  <c:column value="#{'IPv4'}" id="protocol" />
</c:entitySet>
```

Notice that the `allowsXyz` attributes are all set to false. This means that the EntitySet component does not allow the user to add, edit or delete any item in the table.

Finally, the objects supplied in the value list must implement the `Entity` interface even though they will not be manipulated by configuration screens.

The Purpose of Search Pages

Search pages are used to search on and display dynamic data. This data usually comes from a database. Figure 54 shows an example search page from the Firewall module.



Date/time:	today / any time		
Protocol:	UDP	Source IP:	contains...
Incoming interface:	eth0 - internal	Destination IP:	contains...
Outgoing interface:	any interface	Source port:	contains...
Rule:	any rule	Destination port:	contains...

53 log entries found.

1 2 3 4 next >

Show 15 log entries at a time

date/time	data	protocol	incoming IF	outgoing IF	source IP	destination IP	s port	d port	rule
12/4/06 03:22:53		ICMP	eth0 - internal	eth3 - external	192.168.43.1	192.168.46.3			9 - Block ICMP
12/4/06 03:22:50		ICMP	eth0 - internal	eth3 - external	192.168.43.1	192.168.46.3			9 - Block ICMP
12/4/06 03:18:47		ICMP	eth0 - internal	eth3 - internal	192.168.43.1	192.168.46.3			9 - Block ICMP
12/4/06 03:18:24		ICMP	eth0 - internal	eth3 - external	192.168.43.1	192.168.46.3			9 - Block ICMP
12/4/06 03:18:05		ICMP	eth0 - internal	eth3 - external	192.168.43.1	192.168.46.3			9 - Block ICMP
12/4/06 03:18:00		ICMP	eth0 - internal	eth3 - internal	192.168.43.1	192.168.46.3			9 - Block ICMP
12/4/06 03:17:49		ICMP	eth0 - internal	eth3 - external	192.168.43.1	192.168.46.3			9 - Block ICMP
12/4/06 03:17:40		ICMP	eth0 - internal	eth3 - external	192.168.43.1	192.168.46.3			9 - Block ICMP
12/4/06 03:17:27		ICMP	eth0 - internal	eth3 - external	192.168.43.1	192.168.46.3			9 - Block ICMP
12/4/06 03:17:17		ICMP	eth0 - internal	eth3 - external	192.168.43.1	192.168.46.3			9 - Block ICMP

Figure 54: Example Search Page from the Firewall Module

Search screens allow the user to dynamically specify the search criteria for the result set. Search screens should be used when the expected result set is relatively large because the result set will be truncated. The user will then be allowed to page through the result set one chunk at a time.

The Structure of Search Pages

Search pages are split into two section:

- The criterion section at the top of the page provides a form that allows the user to enter the data for the database search as well as the reset and search buttons.
- The result set table at the bottom of the page provides the table of the data plus controls to move through the pages of the result set as well as the size of each page chunk.

Both of these sections are contained within a FormWrapper component to provide proper CSS styles for the form elements.

The Criterion Section

The criterion section provides:

- a toggle button to show and hide the criterion section
- the search form fields
- the search and reset buttons

Here is a sketch of the criterion section:

```
<c:showHideToggle id='logSearchCriteriaShowHide'
```

```
      onElement='firewallLogSearchCriteria'
      value='#{monitorLog.showCriteria}' />

<c:division id='firewallLogSearchCriteria'>

   <c:division id='searchColumnOneCriteria'>
      <!-- LEFT SIDE SEARCH FIELDS -->
   </c:division>

   <c:division id='searchColumnTwoCriteria'>
      <!-- RIGHT SIDE SEARCH FIELDS -->
   </c:division>

   <c:division id='searchButtons' styleClass='searchNavigation'>
      <!-- HTML CONTENT FOR THE RESET/SEARCH BUTTONS -->
   </c:division>
   <f:verbatim><br clear="left" /></f:verbatim>

</c:division>
<f:verbatim><hr /></f:verbatim>
```

The ShowHideToggle component is used to create the plus/minus icon (with corresponding text) which is used to show or hide an HTML division specified by the `onElement` attribute. So far this component is has only been used in search screens, but it is a generic component that can be used in a variety of screens.

The main `firewallLogSearchCriteria` division is organized into three divisions:

- `searchColumnOneCritieria`, which provides the criteria form fields on the left side
- `searchColumnTwoCritieria`, which provides the criteria form fields on the right side
- `searchButtons`, which provides the reset and search buttons

Figure 55 shows the data objects for entering the search criteria.

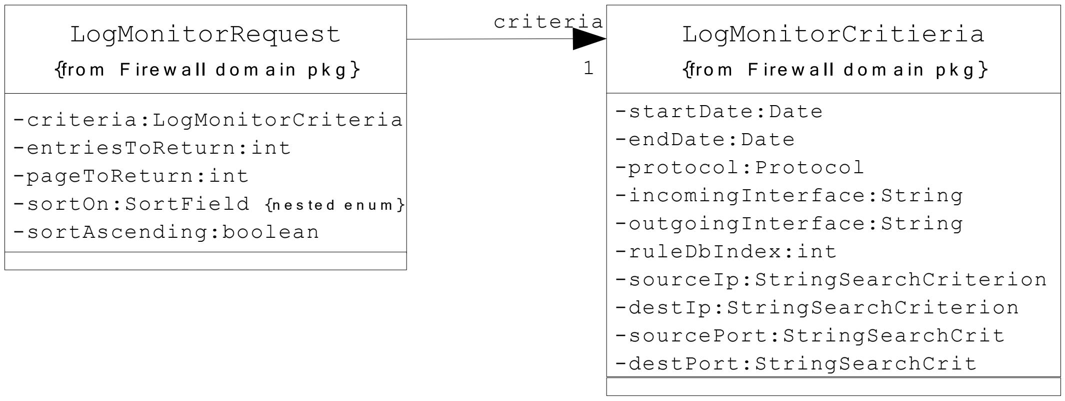

LogMonitorRequest
{from Firewall domain pkg}
-criteria:LogMonitorCriteria
-entriesToReturn:int
-pageToReturn:int
-sortOn:SortField {nested enum}
-sortAscending:boolean
criteria
1
LogMonitorCritieria
{from Firewall domain pkg}
-startDate:Date
-endDate:Date
-protocol:Protocol
-incomingInterface:String
-outgoingInterface:String
-ruleDbIndex:int
-sourceIp:StringSearchCriterion
-destIp:StringSearchCriterion
-sourcePort:StringSearchCrit
-destPort:StringSearchCrit

Figure 55: Search Request Data Objects from the Firewall MonitorLog Screen

The data in the `LogMonitorCriteria` object is populated by the form fields on criterion section of page. The data in the `LogMonitorRequest` object is populated by fields in the result set table section of the page that will be discussed below.

Most of the criteria form is composed of UI widgets that we have already discussed in the Configuration Pages section above. Here is the JSP code for the form widgets on the left side of the criteria section:

```
<c:division id='searchColumnOneCriteria'>
  <c:textField id='startDate' required='false'
       value='#{monitorLog.logRequest.criteria.startDate}'>
    <f:converter converterId="converter.Date" />
  </c:textField>
  <c:dropDownList id='protocol'
       value='#{monitorLog.logRequest.criteria.protocol}'>
     <ctags:enumConverter
         type='org.stillsecure.cobia.base.domain.values.Protocol' />
     <c:enumSelectItems
         type='org.stillsecure.cobia.base.domain.values.Protocol' />
  </c:dropDownList>
  <c:dropDownList id='incomingInterface' required='false'
       value='#{monitorLog.logRequest.criteria.incomingInterface}'>
     <f:selectItems value="#{monitorLog.selectListForInterfaces}" />
   </c:dropDownList>
   <c:dropDownList id='outgoingInterface' required='false'
       value='#{monitorLog.logRequest.criteria.outgoingInterface}'>
     <f:selectItems value="#{monitorLog.selectListForInterfaces}" />
   </c:dropDownList>
   <c:dropDownList id='ruleOption' required='false'
       value='#{monitorLog.logRequest.criteria.ruleDbIndex}'>
     <f:selectItems value="#{monitorLog.selectListForLoggingRules}" />
   </c:dropDownList>
</c:division>
```

However, there is one new widget that is unique to search screens. It is the combination of a drop-down list with a text field. This widget is called StringSearchCriterion.

The drop-down list is specifically populated with an enum for selecting the type of string comparison:

- contains – matches if the DB field contains the string in the text field
- does not contain – matches if the DB field does not contain the string in the text field
- starts with – matches if the DB field starts with the string in the text field
- ends with – matches if the DB field ends with the string in the text field
- is – matches if the DB field is exactly the string in the text field
- is not – matches if the DB field is not the string in the text field

Here is the JSP code for the form widgets on the right side of the criteria section:

```
<c:division id='searchColumnTwoCriteria'>
  <c:textField id='endDate' required='false'
       value='#{monitorLog.logRequest.criteria.endDate}'>
    <f:converter converterId="converter.Date" />
  </c:textField>
  <c:stringSearchCriterion id='sourceIP' required='false'
       value='#{monitorLog.logRequest.criteria.sourceIP}' />
  <c:stringSearchCriterion id='destinationIP' required='false'
       value='#{monitorLog.logRequest.criteria.destinationIP}' />
  <c:stringSearchCriterion id='sourcePort' required='false'
       value='#{monitorLog.logRequest.criteria.sourcePort}' />
  <c:stringSearchCriterion id='destinationPort' required='false'
       value='#{monitorLog.logRequest.criteria.destinationPort}' />
</c:division>
```

The StringSearchCriterion widget takes the data from the enum and the text field and creates a `StringSearchCriterion` object from the Cobia `org.stillsecure.cobia.util` package. This widget has a built-in converter to create these objects.

Lastly, here is the JSP code for the search buttons:

```
<c:division id='searchButtons' styleClass='searchNavigation'>
  <ctags:loadStyle src='/components/styles/contentButton.css' />
  <f:verbatim><![CDATA[
    <ul class="searchNavigation">
      <li>
  ]]></f:verbatim>
         <h:commandLink id="search" action="#{monitorLog.search}" styleClass="contentButton contentSearchButton" value="search"/>
  <f:verbatim><![CDATA[
      </li>
      <li>
  ]]></f:verbatim>
  <jstl:choose>
   <jstl:when test="${monitorLog.logRequest.criteria.default}">
          <h:commandLink id="reset" action="#{monitorLog.reset}" styleClass="contentButton contentResetButtonInactive" value="reset"/>
   </jstl:when>
   <jstl:otherwise>
          <h:commandLink id="reset" action="#{monitorLog.reset}" styleClass="contentButton contentResetButton" value="reset"/>
   </jstl:otherwise>
  </jstl:choose>
  <f:verbatim><![CDATA[
      </li>
    </ul>
  ]]></f:verbatim>
</c:division>
```

As you can see this JSP code does not use Cobia UI widgets. It uses a combination of hard-coded HTML, plus standard JSP tag libraries (JSTL) and standard JSF component such as `commandLink`. Future version of the Cobia UI Framework might include widgets for this set of buttons.

The Result Set Section

The bottom half of Figure 54 on page 95 shows the result set in a paged table. There are no built-in widgets for the elements in this section of the screen. Review the JSP code in the Firewall module at `modules/firewall/web/MonitorLog.jspx` to see how to recreate this code for you search screens. Also you will need to review the backing bean code at `modules/firewall/src/org/stillsecure/cobia/module/firewall/web/MonitorLog.java` to see how the screen object supports the functionality of this screen.

Reporting Pages

This section describes how to create reporting screens.

Not designed in Cobia v0.3.

Help Pages

This section describes how to create help screens.

Not designed in Cobia v0.3.

Dashboard Screens

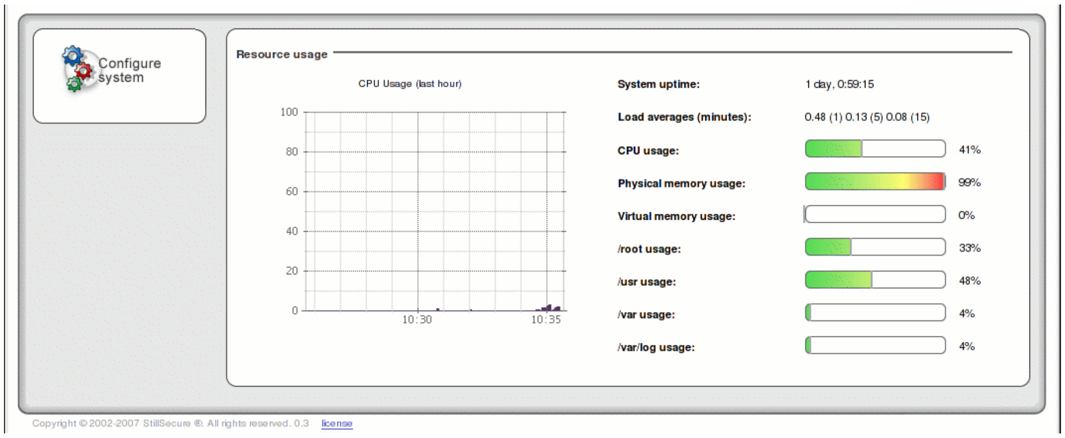
This section describes how to create dashboard screens.

The Purpose of Dashboard Screens

The primary purpose of a Dashboard screen is to present the status of the module's service. This is usually done with summary data and graphs. Dashboards are the hardest screens to create because there is no standard way of constructing them. There are some Cobia widgets that help, but creating these screens is still an art. In the next couple of pages we will show you how the current Cobia module dashboards were built.

The Admin Dashboard

Figure 56 shows Admin dashboard.



System uptime:	1 day, 0:59:15	
Load averages (minutes):	0.48 (1) 0.13 (5) 0.08 (15)	
CPU usage:		41%
Physical memory usage:		99%
Virtual memory usage:		0%
/root usage:		33%
/usr usage:		48%
/var usage:		4%
/var/log usage:		4%

Copyright © 2002-2007 StillSecure ®. All rights reserved. 0.3 license

Figure 56: The Admin Dashboard

There are two major elements to this dashboard: the CPU usage graph on the left and the server statistics shows with text and thermometer-style charts on the right. Here is the JSP code for this content:

```
<c:dataHeading id="resourceUsage" />
<c:division id="monitoring">
  <a4j:region id="ajax_monitoring">
    <a4j:poll id="poller" interval="5000"
        onsubmit="return navigation.a4jSubmit();"
        oncomplete="refreshGraphs();"
        reRender="cpuGraph,sysUptime,sysLoadAverage,cpuUsage,memoryUsage,swapUsage,dfRoot,dfUsr,dfVar,dfVarLog"/>

    <c:division id="graphing">
      <c:division id="GR">
        <c:graph id="cpuGraph" value="#{grapher.cpuGraph}"/>
      </c:division>
    </c:division>

    <c:division id="sysStats">
      <c:dataList id='times' type='definition' styleClass='dashDataList'>
       <c:dataListItem id='sysUptime' itemValue='#{systemMetrics.uptime}' />
       <c:dataListItem id='sysLoadAverage' itemValue='#{systemMetrics.loadAverage}' />
```

```
19.       </c:dataList>
20.       <f:verbatim><br /></f:verbatim>
21.       <c:division id="usage">
22.         <c:percentageScale id="cpuUsage" value="#{systemMetrics.cpuUsage}"/>
23.         <c:percentageScale id="memoryUsage"
value="#{systemMetrics.memoryUsage}"/>
24.         <c:percentageScale id="swapUsage"
value="#{systemMetrics.swapUsage}"/>
25.         <c:percentageScale id="dfRoot" value="#{systemMetrics.dfRoot}"/>
26.         <c:percentageScale id="dfUsr" value="#{systemMetrics.dfUsr}"/>
27.         <c:percentageScale id="dfVar" value="#{systemMetrics.dfVar}"/>
28.         <c:percentageScale id="dfVarLog" value="#{systemMetrics.dfVarLog}"/>
29.       </c:division>
30.     </c:division>
31.     <f:verbatim><br clear="left"/></f:verbatim>
32.
33.   </a4j:region>
34. </c:division>
```

There is a lot going on here; we will break it down into these elements:

- Headings
- Graphs
- Data lists
- Percentage scales
- AJAX functionality

The Cobia UI widget DataHeading (line 1) creates a section heading for Dashboards. The text of the heading is provided in the `text.properties` file:

```
admin.Dashboard.resourceUsage.Label=Resource usage
```

The Cobia UI widget Graph (line 11) creates an HTML `<img>` tag and JavaScript code that includes the `refreshGraphs` function. This function iterates over every `<img>` element within the `<div>` tag whose ID is `graphing`; lines 8-13. The text of the graph title is provided in the `text.properties` file:

```
admin.Dashboard.cpuGraph.Label=CPU Usage (last hour)
```

The Cobia UI widget DataList (lines 16-19) creates an HTML `<dl>` structure with text labels and text data. Each row is created by a single DataListItem subcomponent (lines 17 and 18). The labels of these elements is provided in the `text.properties` file:

```
admin.Dashboard.sysUptime.Label=System uptime:
admin.Dashboard.sysLoadAverage.Label=Load averages (minutes):
```

The Cobia UI widget PercentageScale (lines 22-28) creates individual HTML `<dl>` structures with text labels and a thermometer-styled graphic that reflects an integer value from 0 to 100 percent. The labels for these components are handled in with the Label text attribute.

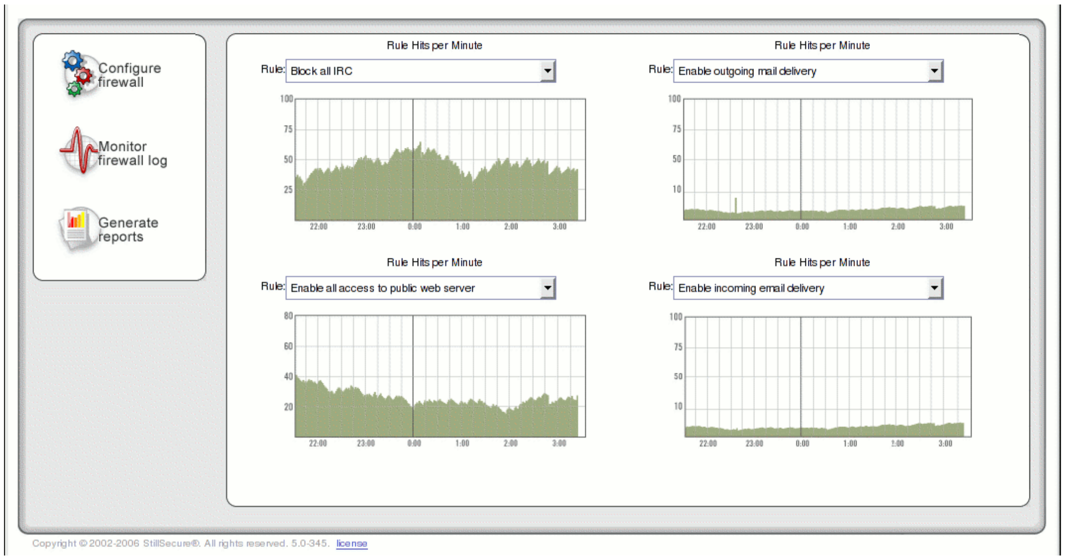
All of these components are made dynamic by using AJAX (Asynchronous JavaScript and XML). The Cobia project uses the A4J (AJAX for JSF) library which is built into the JBOSS RichFaces component library which is also include. The `<a4j:region>` tag (lines 3-33) surround all of the components that need to be AJAX-enabled. The `<a4j:poll>` tag (lines 4-7) creates a periodic timer which sends a JSF request to rerender a chunk of the current page. By default, the whole screen will be rerendered ; however, the `<a4j:region>` tag will isolate this process to only tags within this region.

The poller uses the `onsubmit` to modify how the form submission occurs; you need this. After the request comes back the poller executes the JavaScript in the `oncomplete` attribute; the `rehreshGraphs` function is called to have the graphs pull a fresh image from the server.

Review the `modules/admin/web/styles/admin.css` file to understand how these elements are styled. Because most of these elements are *floated* in several locations in the JSP code we embedded HTML break tags (`<br>`); see lines 20 and 31.

The Firewall Dashboard

Figure 57 shows the Firewall dashboard.



Rule: Block all IRC
Rule: Enable outgoing mail delivery
Rule: Enable all access to public web server
Rule: Enable incoming email delivery

Copyright © 2002-2006 StillSecure®. All rights reserved. 5.0-345. license

Figure 57: The Firewall Dashboard

This dashboard contains for graphs that display the number of hits per minute for a specific rule. The rule is selected from the drop-down list above the graph. Here is the JSP code for this dashboard:

```
1.  <c:dataHeading id='ruleHitCounts' />
2.  <c:division id='ruleHitsGraphs'>
3.     <a4j:region id='ajax_monitoring'>
4.        <a4j:poll id='poller' interval='5000'
5.                  onsubmit='return navigation.a4jSubmit();'
6.                  oncomplete='refreshGraphs();'
7.                  reRender='ruleHits1Graph, ruleHits2Graph,
8.                            ruleHits3Graph, ruleHits4Graph'
9.                  action="#{firewallHome.refreshGraphs}" />
10.
11.       <c:division id='graphing'>
12.          <c:division id='rule1'>
13.             <c:htmlTag value="h4">
```

```
14.                <h:outputText value="Rule Hits per Minute"/>
15.             </c:htmlTag>
16.             <c:dropDownList id='ruleHits1GraphSelect' divClass='graphSelect'
17.                             value='#{firewallHome.rule1ToGraph}'>
18.                <f:selectItems
value='#{firewallHome.selectListForCountingRules}'/>
19.                <a4j:support event='onchange'
20.                             onsubmit='navigation.a4jSubmit(false)'
21.                             reRender='ruleHits1Graph'
22.                             ajaxSingle='true'/>
23.             </c:dropDownList>
24.             <c:division id='rule1Scale' styleClass='scale'>
25.                <c:graph id='ruleHits1Graph' inhibitLabel='true'
26.                         value='#{firewallHome.rule1GraphImgUrl}'/>
27.             </c:division>
28.          </c:division>
29.
30.          <c:division id='rule2'>
31.             <!-- CONTENT FOR RULE #2 -->
32.          </c:division>
33.
34.          <c:division id='rule3'>
35.             <!-- CONTENT FOR RULE #3 -->
36.          </c:division>
37.
38.          <c:division id='rule4'>
39.             <!-- CONTENT FOR RULE #4 -->
40.          </c:division>
41.          <f:verbatim><br clear='left'/></f:verbatim>
42.
43.       </c:division>
44.    </a4j:region>
45. </c:division>
46. <f:verbatim><br clear='left'/></f:verbatim>
```

The DropDownList component (line 16-23) uses the `<a4j:support>` tag to issue an AJAX request when the user selects a item in the drop-down list. To get the graph title (a <h4> tag) to render above the drop-down list, we use an `<a4j:htmlTag>` tag to render the <h4> tag and use the `inhibitLabel` attribute on the Graph component. In the future, we might create a single component for this combination of elements if we find further uses for it.

The Router Dashboard

Figure 58 shows the Router dashboard.

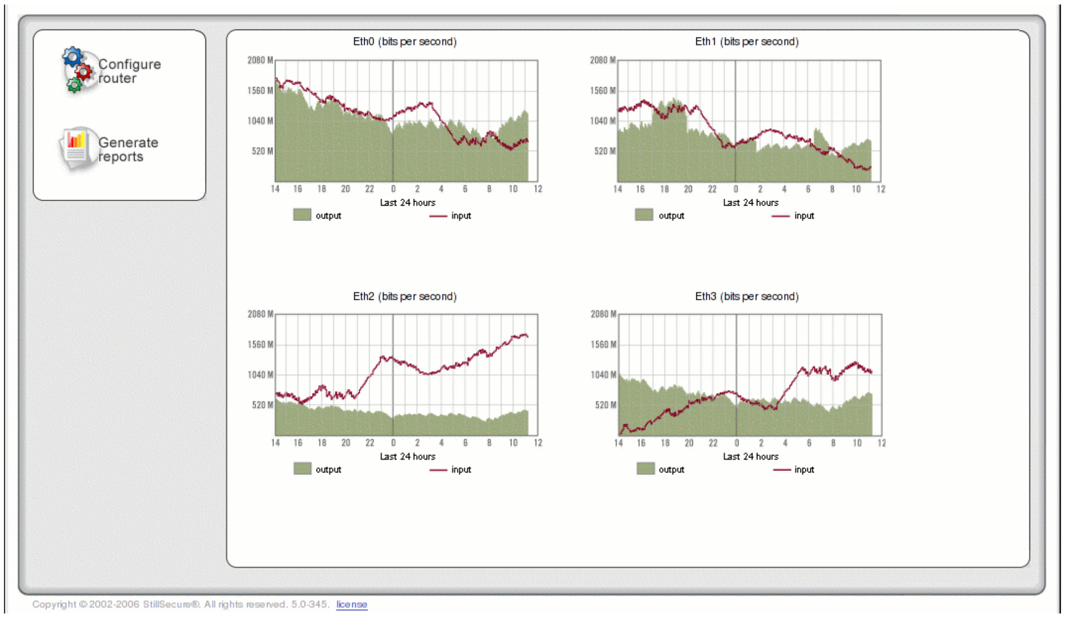


Figure 58: The Router Dashboard

This dashboard contains a graph for every active Ethernet interface on the appliance.

```
1.  <c:dataHeading id="ethernetThroughput" />
2.  <a4j:region id="ajax_monitoring"
3.              ajaxListener="#{grapher.createInterfaceGraphs}">
4.     <a4j:poll id="poller" interval="5000"
5.               onsubmit="return navigation.a4jSubmit();"
6.               oncomplete="refreshGraphs();"
7.               reRender="interfaceGraphs"/>
8.     <c:division id="graphing">
9.        <t:dataList id="interfaceGraphs" layout="simple"
10.                   var="intf" value="#{routerHome.interfaces}">
11.          <c:graph id='ethThroughputGraph'
12.                   value="/graphs/#{intf.interface.device}.png"/>
13.       </t:dataList>
14.    </c:division>
15.    <f:verbatim><br clear='left' /></f:verbatim>
16. </a4j:region>
```

There are two new elements in the page. First, we need to iterate over each interface and create a Graph element for each. This is accomplished using the `<t:dataList>` from the Tomahawk tag library which is part of MyFaces. This component acts like the JSTL `<core:forEach>` tag in that it iterates over a collection in the `value` attribute (line 10) and assigns each item in the collection to the EL variable declared in the `var` attribute. This EL variable is then available for use in the `value` attribute of the `<c:graph>` tag (lines 11 and 12).

Second, the title for each graph requires a unique string. This is accomplished using a trick in the `text.properties` file:

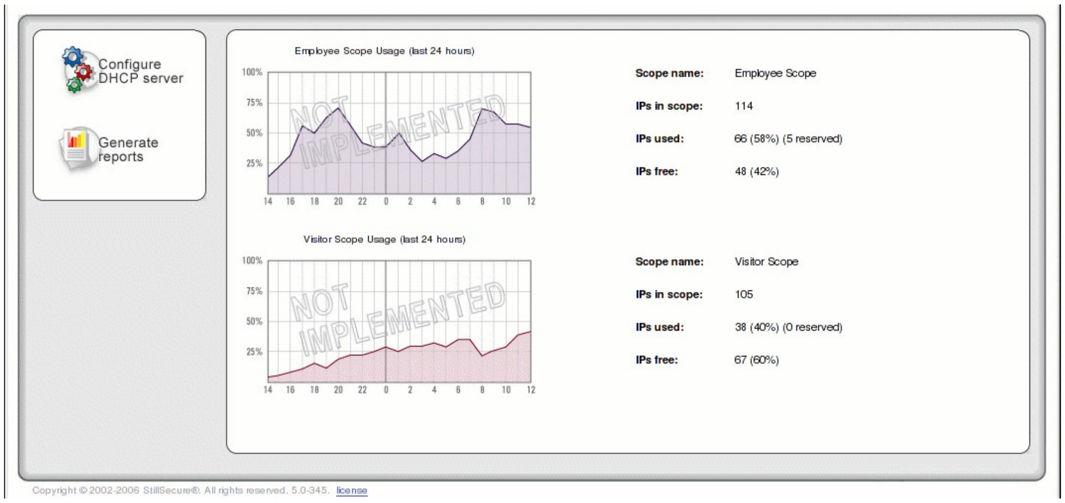
```
router.Dashboard.ethThroughputGraph.Label={0} (bits per second)
router.Dashboard.ethThroughputGraph.labelArgs=#{intf.interface.device}
```

The `Label` attribute uses a message argument `{0}` in the title. This argument is populated by the second text attribute, `labelArgs`. The trick is that this attribute is a

JSF value binding which is evaluated at run-time to determine the text that is inserted into the `Label` attribute.

The DHCP Dashboard

Figure 59 shows the DHCP dashboard.



Scope name:	Employee Scope
IPs in scope:	114
IPs used:	66 (58%) (5 reserved)
IPs free:	48 (42%)

Scope name:	Visitor Scope
IPs in scope:	105
IPs used:	38 (40%) (0 reserved)
IPs free:	67 (60%)

Copyright © 2002-2006 StillSecure®. All rights reserved. 5.0-345. license

Figure 59: The DHCP Dashboard

The DHCP dashboard is similar to the Admin dashboard in that includes a graph next to a block of dynamic, textual information. The main difference is that there is a unique graph/text combination for every DHCP scope defined by the user. Therefore, this screen must also use the Tomahawk DataList component. Here is the JSP code for this screen:

```
<c:dataHeading id="dhcpScopes"/>
<c:division id="dashboardScopes">
<a4j:region id="ajax_monitoring"
            ajaxListener="#{dhcpHome.createAllDashboardGraphs}">
   <a4j:poll id="poller" interval="5000"
             onsubmit="return navigation.a4jSubmit();"
             oncomplete="refreshGraphs();"
             reRender="scopeGraphs"/>

   <c:division id='graphing'>
      <t:dataList id="scopeGraphs" layout="simple"
                  var="scope" value="#{dhcpHome.scopes}">
         <c:division styleClass="scope">
            <c:division styleClass="scopeGraph">
               <c:division id="GR">
                  <c:graph id='scopeUsage'
value="/dhcp/images/dhcp_usage_xybar_#{scope.ethInterface.device}.jpg"/>
               </c:division>
            </c:division>
            <c:dataList id="dhcpScopeStats" type="definition"
                 styleClass='dashDataList' divStyleClass='scopeStatistics'>
               <c:dataListItem id='scopeName' itemValue="#{scope.name}" />
               <c:dataListItem id='ipsInScope' itemValue="#{scope.totalIPs}"
/>
               <c:dataListItem id='ipsInUse'
itemValue="#{messages['dhcp.Dashboard.ipsInUse.Value'][scope.usedIPs][scope.
precentUsedIPs]}" />
               <c:dataListItem id='ipsFree'
itemValue="#{messages['dhcp.Dashboard.ipsFree.Value'][scope.freeIPs][scope.p
recentFreeIPs]}" />
```

```
25.             </c:dataList>
26.          </c:division>
27.          <f:verbatim><br clear='left' /></f:verbatim>
28.       </t:dataList>
29.    </c:division>
30. </a4j:region>
31. <core:if test="${empty dhcpHome.scopes}">
32.    <f:verbatim><h4 class="emptyDashboard">
33.                   ${messages['dhcp.Dashboard.noScopesMessage']}
34.                </h4>
35.     </f:verbatim>
36. </core:if>
37.
38. </c:division><!-- END: dashboardScopes -->
```

The Tomahawk DataList component (lines 11-29) iterates over every DHCP scope and generates a Cobia Graph and Cobia DataList for each scope. Two of the DataListItem components use an odd value binding on lines 23 and 24. The `message` JSF attribute acts like a Java `MessageFormat` object by first looking up the resource key in the first bracket in the text.properties file. Each bracket following that is an argument to the message. Here is the text for the `dhcp.Dashboard.ipsInUse.Value` key:

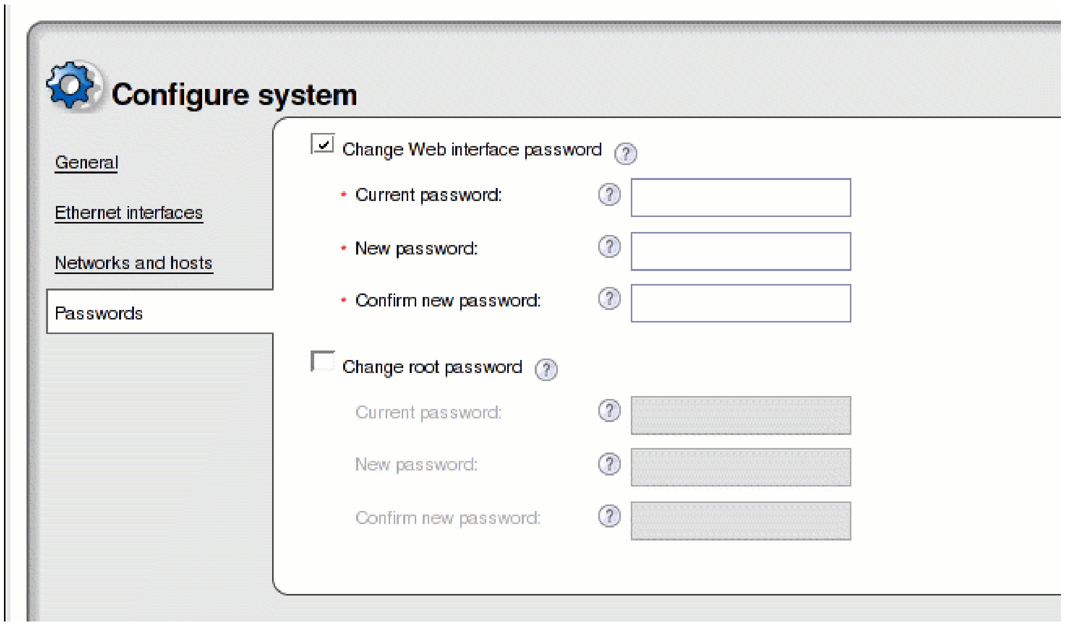
```
dhcp.Dashboard.ipsInUse.Value={0} ({1}%)
```

The value of `scope.usedIPs` is placed in the first argument `{0}` and the value of `scope.precentUsedIPs` is placed in the second argument `{1}`.

The JSP code lines 31 through 36 are used to put something in the dashboard if there are no scopes defined. This prevents the content portion of the screen from being completely empty.

Client-side Screen Behavior

Some screens may require special client-side behavior. The Cobia UI Framework supports this by automatically linking to a JS file for each page. For example, Figure 60 shows the EditPassword page in the Admin module.



[✓] Change Web interface password
* Current password:
* New password:
* Confirm new password:
[ ] Change root password
Current password:
New password:
Confirm new password:

Figure 60: The EditPassword Page in the Admin Module

The two checkboxes enable the three text fields below each checkbox. This is a behavior that goes beyond the standard behavior of the BooleanCheckbox component so it must be added using JavaScript. Here is the JSP code for the main content of this page:

```
<c:formWrapper id='changePasswords'>

   <c:booleanCheckbox id='uiPassword' value='#{editPassword.uiPassword}' />
   <c:subSection id='changeUIPassword'>
      <c:secretField id='currentUIPwd' required='true'
                   value='#{editPassword.currentUIPwd}'
                   validator="#{editPassword.validateCurrentUIPassword}" />
      <c:secretField id='newUIPwd' required='true'
                   value='#{editPassword.newUIPwd}'
                   validator="#{editPassword.validateUIPassword}" />
      <c:secretField id='confirmUIPwd' required='true'
                   value='#{editPassword.confirmUIPwd}'
                   validator="#{editPassword.validateUIPasswordsMatch}" />
   </c:subSection>

   <c:booleanCheckbox id='rootPassword' value='#{editPassword.rootPassword}'
/>
   <c:subSection id='changeRootPassword'>
      <c:secretField id='currentRootPwd' required='true'
                 value='#{editPassword.currentRootPwd}'
                 validator="#{editPassword.validateCurrentRootPassword}" />
      <c:secretField id='newRootPwd' required='true'
                 value='#{editPassword.newRootPwd}'
                 validator="#{editPassword.validateRootPassword}" />
      <c:secretField id='confirmRootPwd' required='true'
                     value='#{editPassword.confirmRootPwd}'
                     validator="#{editPassword.validateRootPasswordsMatch}" />
   </c:subSection>

</c:formWrapper>
```

When a screen or page is rendered (see the Screen Rendering Process section on page 28), the Cobia UI Framework automatically includes a `<script>` tag to link to the page's JS file defined by the path:

`/<moduleID>/scripts/<screenID or pageID>.js`

So, for the EditPassword page this JS file would be:

`/admin/scripts/EditPassword.js`

Here is the code in this JS file:

```
/**
* Enables or disables the form subsection controlled by the uiPassword
checkbox.
*
* @param chkBox -- The x:uiPassword checkbox HTMLElement
*/
function changedUIPassword(chkBox)
{
   var pwdSubSection = $('x:changeUIPassword');

   if ( chkBox.checked )
   {
      enableObject(pwdSubSection);
   }
   else
   {
      disableObject(pwdSubSection);
   }
}

/**
* Enables or disables the form subsection controlled by the rootPassword
checkbox.
*
* @param chkBox -- The x:rootPassword checkbox HTMLElement
*/
function changedRootPassword(chkBox)
{
   var pwdSubSection = $('x:changeRootPassword');

   if ( chkBox.checked )
   {
      enableObject(pwdSubSection);
   }
   else
   {
      disableObject(pwdSubSection);
   }
}

var EditPassword = {

   /**
   * Initialize the screen.
   * This will be the last thing added to the "on-DOM-load" event handler.
   */
   initialize : function ()
46.     {
47.        // Initialize the state of the screen
48.        changedUIPassword($('uiPasswordCHECKBOX'));
49.        changedRootPassword($('rootPasswordCHECKBOX'));
50.        // Configure the radio buttons
51.        Event.observe('uiPasswordCHECKBOX', 'click',
52.             function () { changedUIPassword($('uiPasswordCHECKBOX')) });
53.        Event.observe('rootPasswordCHECKBOX', 'click',
54.             function () { changedRootPassword($('rootPasswordCHECKBOX')) });
55.     }
56.
57. }
```

The first two functions handle each checkbox; one for the UI password subsection and one for the root password subsection. The last chunk of JS code (lines 39-57) define a JavaScript namespace for this page and in it defines an `initialize` function (lines 45-55). This function initializes the state of the screen by calling the previously defined JS functions to enable or disable the subsections based upon the state of the corresponding checkbox (lines 48 and 49). Next the `initialize` function uses the Prototype `observe` function to add these `changed`*`Xyz`*`Password` functions to the appropriate checkbox element.

Any screen or page that requires this type of initialization must create a namespace for that screen/page ID *and* an `initialize` function must also be created.

This `initialize` function must be called after the page's DOM (Document Object Model) is completely loaded; therefore, the page's `initialize` function is added to a DOM-load event handler. This is also part of the screen rendering process. Here is the JS code rendered at the bottom of the EditPassword page:

```
   <script language="javascript" type="text/javascript">
      if ( typeof EditPassword != 'undefined' )
      {
        if ( typeof EditPassword.initialize == 'function' )
        { addDOMLoadEvent(EditPassword.initialize); }
      }
   </script>
```

There is a lot you can do with client-side JavaScript. However, using JavaScript with the Cobia UI Framework requires an understanding of how the component are rendered in order to determine the IDs of various rendered HTML elements for complex components. The best way to do this is either by reading the source code of the renderer classes of the JSF components or by looking at the actual rendered HTML content.

Dynamic JavaScript

One last comment on JavaScript files. You can use JSPX code to create dynamic JavaScript. For example, there are components in the DHCP module which require a JS array of DHCP options which is generated by the web server.

Here is the code for the file `modules/dhcp/web/scripts/dhcp_options.jspx`:

```
<jsp:root
   xmlns:jsp='http://java.sun.com/JSP/Page' version='2.0'
   xmlns:f='http://java.sun.com/jsf/core'
   xmlns:c='http://java.sun.com/jsp/jstl/core'>
<jsp:directive.page contentType='text/javascript' />
<f:view>
<f:verbatim><![CDATA[
var DHCP = { // START: DHCP namespace
9.
10.     //
11.     // Option type constants
12.     //
13.     OPTION_TYPE_BOOLEAN : 0,
14.     OPTION_TYPE_BOOLEANS : 1,
15.     OPTION_TYPE_UNSIGNED_8_BIT_INTEGER : 2,
16.     OPTION_TYPE_UNSIGNED_16_BIT_INTEGER : 3,
17.     OPTION_TYPE_UNSIGNED_32_BIT_INTEGER : 4,
18.     OPTION_TYPE_SIGNED_8_BIT_INTEGER : 5,
19.     OPTION_TYPE_SIGNED_16_BIT_INTEGER : 6,
20.     OPTION_TYPE_SIGNED_32_BIT_INTEGER : 7,
21.     OPTION_TYPE_STRING : 8,
22.     OPTION_TYPE_TEXT : 9,
23.     OPTION_TYPE_IP_ADDRESS : 10,
24.     OPTION_TYPE_IP_ADDRESSES : 11,
25.     OPTION_TYPE_INTEGERS : 12,
26.     OPTION_TYPE_2D_ARRAYS : 13, /* not supported */
27.
28.     //
29.     // Option object constructor
30.     //
31.     Option : function(code, name, description, type)
32.     {
33.         this.code = code;
34.         this.name = name;
35.         this.description = description;
36.         this.type = type;
37.
38.         return this;
39.     },
40.
41.     // Global variable of all DHCP options (used by the Edit/Add DHCP Option
    screen)
42.     OPTIONS_ARRAY : new Array(),
43.
44.     initialize : function()
45.     {
46. ]]></f:verbatim>
47. <c:forEach var="optionKey" items="${editOption.allOtherOptionKeys}"
    varStatus="vs" >
48. <f:verbatim><![CDATA[       DHCP.OPTIONS_ARRAY[${vs.index}] = new
    DHCP.Option(${optionKey.code}, "${optionKey.userName}",
    "${optionKey.userDescription}", DHCP.${optionKey.type});]]></f:verbatim>
49. </c:forEach>
50. <f:verbatim><![CDATA[
51.     }
52.
53. } // END: DHCP namespace
54.
55. addDOMLoadEvent(DHCP.initialize);
56. ]]></f:verbatim>
57. </f:view>
58. </jsp:root>
```

Most of this code is static content (lines 8-45 and 51-55). This sets up a JS namespace called `DHCP`. The code from lines 47 through 49 uses the JSTL `forEach` tag to iterate over every DHCP option from the domain model and create `OPTIONS_ARRAY` elements, one for each option.

This file is included in the JSP files that need this array, such as the AddOption screen. Here is the JSP code that performs the include:

```
1. <c:screen value='#{editOption}'>
2. <ctags:loadScript src='scripts/dhcp_options.jsf' />
```

```
  <!-- the rest of the screen -->
</c:screen>
```

Including Raw HTML Content

We want you to use the Cobia UI component library; however, we recognize that there will be cases in which you need either raw HTML or a combination of raw HTML with JSF standard (or third-party) components.

If the HTML you need to include is complete (meaning the complete XML structure is in place), then you can use the JSF `verbatim` tag. Here is an example:

```
<f:verbatim>
   <h4 class="emptyDashboard">
      ${messages['dhcp.Dashboard.noScopesMessage']}
   </h4>
</f:verbatim>
```

The `<h4>` tag is complete; it is started and ended within the `verbatim` tag.

In some cases it is not possible to have complete HTML code within the `verbatim` tag. Here is an example:

```
<f:verbatim><![CDATA[
    <ul class="searchNavigation">
      <li>
]]></f:verbatim>
         <h:commandLink id="search" action="#{monitorLog.search}"
styleClass="contentButton contentSearchButton" value="search"/>
<f:verbatim><![CDATA[
      </li>
      <li>
]]></f:verbatim>
```

In this situation you must use the XML CDATA notation (`<![CDATA[ any text ]]>`) within the `verbatim` tag.

WARNING: In some cases, the JSF verbatim tag does not work well with AJAX; uses these techniques at your own risk.