Oracle® WebCenter Sites

Developer’s Guide for Customizing the Contributor Interface

11*g* Release 1 (11.1.1) Bundled Patch 1

October 2012

ORACLE®

Oracle WebCenter Sites Developer's Guide for Customizing the Contributor Interface, 11*g* Release 1 (11.1.1) Bundled Patch 1

Primary Author: Promila Chitkara, Tatiana Kolubayev

Contributor: Vijayalakshmi Rajan, Patrice Palau, Ravi Khanuja, Kannan Appachi

Contents

List of Tables

Preface

This guide begins with an overview of the development environment for customizing the WebCenter Sites Contributor interface. Later chapters provide information about customizable interface components, customization methods, and supporting code.

The Oracle WebCenter Sites application, discussed in this guide, is a former FatWire product. *Oracle WebCenter Sites* is the current name of the application previously known as *FatWire Content Server*. In this guide, *Oracle WebCenter Sites* is also called *WebCenter Sites*.

Audience

This document is intended for developers with a working knowledge of the Oracle WebCenter Sites Contributor interface and its development environment.

Related Documents

For more information, see the following documents in the Oracle WebCenter Sites 11*g*R1 Bundled Patch 1 documentation set:

- *Oracle WebCenter Sites User's Guide*
- *Oracle WebCenter Sites Developer's Guide*
- *Oracle WebCenter Sites Developer Tools*
- *Oracle WebCenter Sites Tag Reference*
- *Oracle WebCenter Sites Javadoc*
- *Oracle WebCenter Sites Administrator's Guide*

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1 About Customizing the WebCenter Sites Contributor Interface

This guide describes procedures for customizing components of the WebCenter Sites Contributor interface. This chapter summarizes the customizable components and guides you to the location of sample code provided with WebCenter Sites.

1.1 Before You Begin

Developers using this guide are required to have a working knowledge of the Contributor interface; experience with Java, JavaScript, and HTML; and solid familiarity with WebCenter Sites development tools.

Information about the Contributor interface is available in the *WebCenter Sites User's Guide*. Information about the development environment and functionality supporting content management operations is available in the *WebCenter Sites Developer's Guide* and *WebCenter Sites Developer Tools*.

1.2 What Can You Customize in the Contributor Interface?

The following list summarizes the components you can customize in the Contributor interface:

- Dashboard. See Chapter 3, "Customizing the Dashboard."
- Search views. See Chapter 4, "Customizing Search Views."
- Global and site-specific configuration properties, toolbar, and menu bar. See Chapter 5, "Customizing Global Properties, Toolbar, and Menu Bar."
- Asset forms. See Chapter 6, "Customizing Asset Forms."

1.3 Where to Find Sample Code?

Some of the sample code for illustrating interface customization is provided in this guide. Other code is either packaged in WebCenter Sites, or available independently, in the zip file containing this guide. Paths to such code are listed in the individual chapters of this guide.

1.4 Where to Begin?

The Contributor interface framework contains a component called the *UI Controller*, which handles most of the interface-related requests, except for those pertaining to asset forms. The UI Controller is described in Chapter 2, "Contributor Interface Framework."

- If you are customizing any of the following components: dashboard, search views, configuration properties, toolbars, or menu bars, we recommend starting with Chapter 2 to obtain basic information about the concepts and code you will be using in most of this guide and your customization process.
- If you are customizing asset forms, you can skip to Chapter 6, "Customizing Asset Forms" for information about modifying asset form headers and building an attribute editor.

2 Contributor Interface Framework

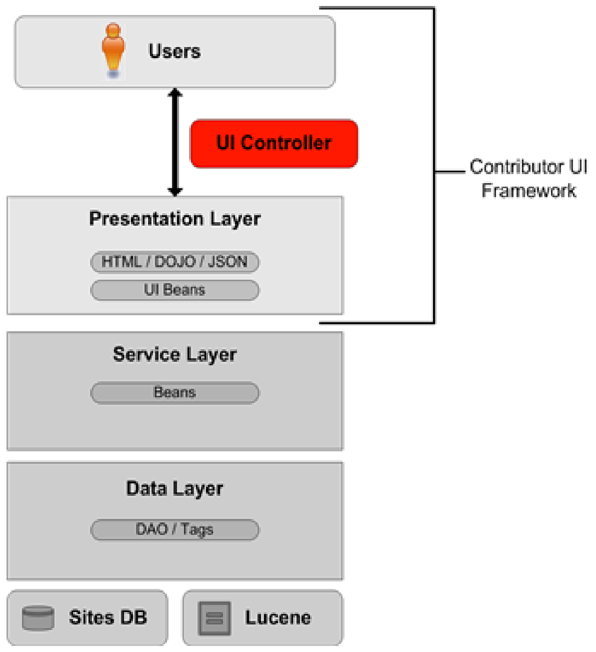
This chapter describes the framework of the WebCenter Sites Contributor interface, particularly the UI Controller, which handles all requests pertaining to the interface. This chapter also discusses elements that are executed by the UI Controller. The concepts introduced here are basic to customizing the Contributor interface. They are used throughout this guide.

This chapter contains the following topics:

2.1 Overview of the Contributor Framework

The framework of the WebCenter Sites Contributor interface sits on top of the Services Layer and handles client requests. As shown in Figure 2–1, the framework consists of the Presentation Layer and UI Controller.

Figure 2–1 Contributor Interface Framework



The Presentation Layer consists of elements that render views and elements that generate a response. The UI Controller is used to process the requests it receives from the Contributor interface, as explained in Section 2.2, "UI Controller."

Note: The UI Controller is not used to process requests pertaining to asset forms, given that asset forms exist outside the Contributor framework. Asset forms are discussed in Chapter 6, "Customizing Asset Forms."

2.2 UI Controller

The UI Controller, shown in Figure 2–1, is the entity that processes the requests it receives from the Contributor interface. This section describes the UI Controller's request processing phases, conventions for naming the elements in each phase, and the process by which the UI Controller checks for custom elements.

This section contains the following topics:

- Section 2.2.1, "How the UI Controller Processes Requests"
- Section 2.2.2, "Example: UI Controller Processing an Element Request"

2.2.1 How the UI Controller Processes Requests

The UI Controller can be reached by invoking the `fatwire/ui/controller` SiteCatalog entry. The UI Controller requires the incoming request to provide at least one parameter, `elementName`, which determines the controller element to be executed. For example, the following URL invokes the controller element `Foo/Bar`:

```
http://localhost:7001/sites/ContentServer?pagename=fatwire/ui/
 controller/controller&elementName=Foo/Bar
```

A controller element is processed in the following three phases:

1. Configuration phase
2. Action phase
3. Presentation phase

where each phase consists of running a distinct element. For each phase, the corresponding element name is determined by a naming convention, described below.

Note: A controller element is any element that can be invoked through the UI Controller.

In each of the phases described above, the UI Controller first tests for the custom element specific to that phase. The process flow is illustrated in the steps of Section 2.2.2, "Example: UI Controller Processing an Element Request."

---

1. Configuration Phase

This phase consists of evaluating the configuration element. The configuration element is meant to contain configuration settings used by the controller element being invoked. The expected element name is `<controllerElementName>Config`, where `<controllerElementName>` is the value of the `elementName` parameter. For instance, in our example, where the controller element name is assumed to be `Foo/Bar`, the expected name of the configuration element is `Foo/BarConfig`.

The Configuration phase is based on Apache Commons Configuration and requires configuration data to be formatted as a valid XML document. For example:

```
<myconfig>
<foo>123</foo>
<bar>foobar</bar>
</myconfig>
```

The XML configuration data is evaluated into a configuration object, that is, an instance of `org.apache.commons.configuration.beanutils.ConfigurationDynaBean`, which is kept in the request scope, where it is identified by the name of the XML root element. In our example, the configuration object can be accessed in the Action phase or Presentation phase as follows:

```
ConfigurationDynaBean configBean =
(ConfigurationDynaBean)request.getAttribute("myconfig");
```

where `myconfig` matches the name of the top-level XML element in the configuration element. More information about Apache commons configuration can be obtained at the following URL: `http://commons.apache.org/configuration`.

Note: About Configuration Elements in the Configuration Phase:

1. The Configuration phase is conditional. If the element `<controllerElementName>Config` does not exist, the UI Controller skips this phase and moves on to the next phase without creating a configuration object.
2. Unlike in the other two phases, the configuration element is not evaluated directly (using, for example, `ics.callElement`). Instead, it is invoked through the `fatwire/ui/controller/readConfiguration` SiteCatalog entry, using `ics.ReadPage()`, allowing to capture its output.

2. Action Phase

In this phase, the UI Controller evaluates the action element. The expected name of the action element is `<controllerElementName>Action`. In our example, the action element name is `Foo/BarAction`.

The action element is meant to contain arbitrary business logic. It typically builds java objects in the request scope, in order to be consumed by the next phase.

Note: The Action phase is conditional. If the element `<controllerElementName>Action` does not exist, the UI Controller skips this phase and moves on to the Presentation phase.

3. Presentation Phase

In this last phase, the UI Controller evaluates the presentation element, whose name depends on the content type of the generated output. The UI Controller can serve either HTML (the default behavior) or JSON. The element name would then be `<controllerElementName>Html` or `<controllerElementName>Json`.

In our example, the UI Controller would attempt to evaluate `Foo/BarHtml`, because HTML is the default content type. If you wish to generate JSON data instead, you must explicitly specify a response type as follows:

```
http://localhost:7001/sites/ContentServer?pagename=fatwire/ui/
 controller/controller&elementName=Foo/Bar&responseType=json
```

In this case, the UI Controller will attempt to evaluate the presentation element called `Foo/BarJson`.

2.2.2 Example: UI Controller Processing an Element Request

When the UI Controller processes any element request, it tests for the custom element as follows: In each phase (Configuration, Action, and Presentation), the UI Controller first looks for the custom element specific to that phase (for more information, see Section 2.3.2, "How the UI Controller Locates Elements"). If the custom element is not found, the UI Controller looks for the default element. If the default element is not found, the UI Controller skips the phase and moves on to the next phase.

The steps below explain, by example, how the UI Controller processes an element request. In this example, the request is for an existing element named `UI/Layout/LeftNavigation`, and the response type is `Html`:

1. **Configuration Phase.** The UI Controller looks for the `LeftNavigation` element's configuration. That is, the UI Controller looks for the element named `LeftNavigationConfig.jsp` under `CustomElements` (in the `ElementCatalog`). If the element exists, the UI Controller reads this element. Otherwise, it reads the default element `LeftNavigationConfig.jsp` (in `UI/Layout/`). The UI Controller then generates the configuration object and keeps this object in the request scope.

   An alternative is to pass the configuration file name as an argument to the UI Controller call. The passed parameter is named `configName`. If `configName` is passed, the UI Controller looks for the element specified in that parameter.

2. **Action Phase.** The UI Controller now looks for the element `LeftNavigationAction.jsp`. If it finds the element under `CustomElements`, the UI Controller executes this element. Otherwise, it executes the default `LeftNavigationAction.jsp` element (in `UI/Layout/`).

3. **Presentation Phase.** In the current example, the response type is `Html`. Therefore, the UI Controller looks for the element `LeftNavigationHtml.jsp`. If it finds the element under `CustomElements`, the UI Controller executes this element to generate an `Html` response. Otherwise, it executes the default `LeftNavigationHtml.jsp` element (in `UI/Layout/`).

2.3 Custom Elements

When customizing the Contributor interface, store your custom elements in the recommended location as discussed in this section, and ensure you have a clear understanding of how they are located by the UI Controller.

This section contains the following topics:

- Section 2.3.1, "Element Storage"
- Section 2.3.2, "How the UI Controller Locates Elements"
- Section 2.3.3, "Element Naming Conventions in This Guide"

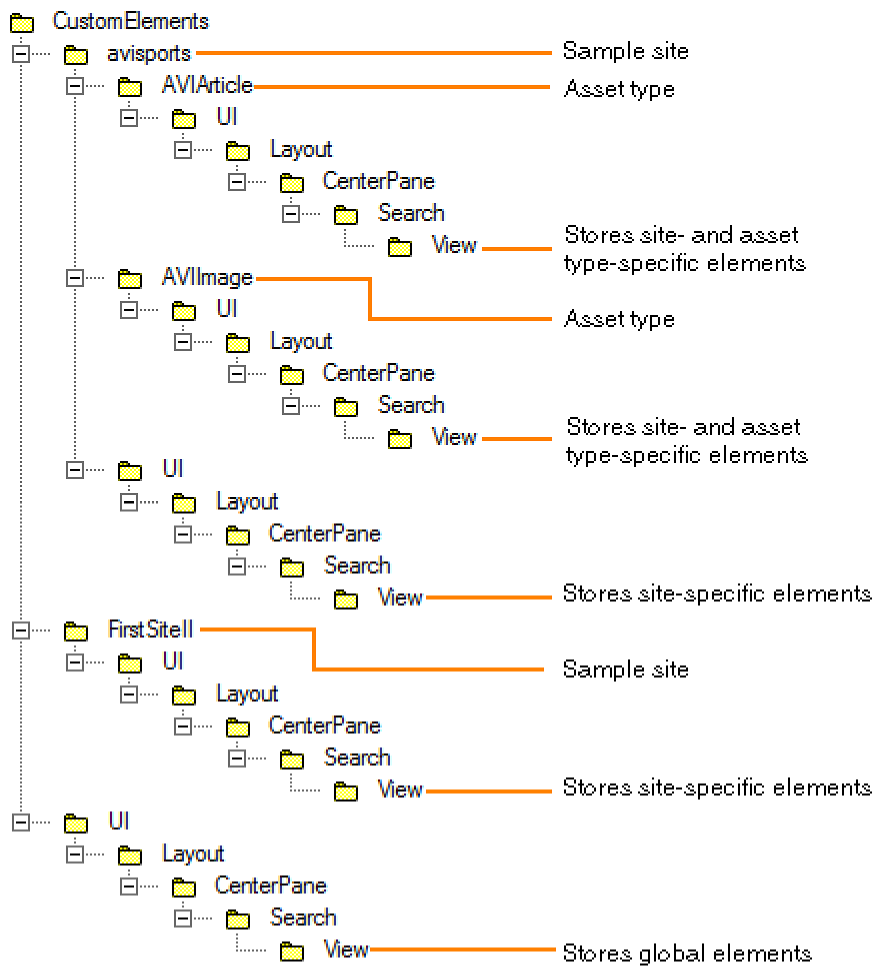
2.3.1 Element Storage

The framework of the Contributor interface allows developers to keep their custom elements separate from the system default elements, in accordance with best practices.

Note: We recommend not modifying the system's default configurations. Instead, create your own custom elements and store them under `CustomElements` of the `ElementCatalog` to ensure their preservation during upgrades.

The path to a custom element depends on whether the element is global, site-specific, site- and asset type- specific, or just asset type-specific. For an example, see Figure 2–2.

Figure 2–2 Paths to Custom Elements



2.3.2 How the UI Controller Locates Elements

When the UI Controller looks for an element:

1. The UI Controller first looks for the customized version of the element by traversing all paths under `CustomElements` in the following order:
   a. Site-specific and asset type-specific paths
   b. Asset type-specific paths
   c. Site-specific paths
   d. Global paths

(For an example of paths, see Figure 2–2.)

2. If the custom element is not found, the UI Controller uses the system-defined element.

> **Note:** For the UI Controller to use the asset type-specific element, the `assetTypeParam` parameter must be passed with a valid asset type as its value.

2.3.3 Element Naming Conventions in This Guide

When referring to a system-defined element or sample element packaged with WebCenter Sites, this guide provides the full path to the element. The full path always begins with `UI/Layout/`. For example, the system-defined element `DashBoardContentsConfig.jsp` is presented as follows in this guide:

```
UI/Layout/CenterPane/DashBoardContentsConfig
```

When referring to a custom-defined element that you create, this guide provides only the name of the element (JSP), given that its path is unknown. For example:

```
DashBoardContentsConfig.jsp
```

It is assumed that the custom element is stored under `CustomElements`.

3
Customizing the Dashboard

This chapter describes how to customize the dashboard of the WebCenter Sites Contributor interface. It familiarizes you with the dashboard configuration and provides sample code, which you can use while you perform the procedures described in this chapter.

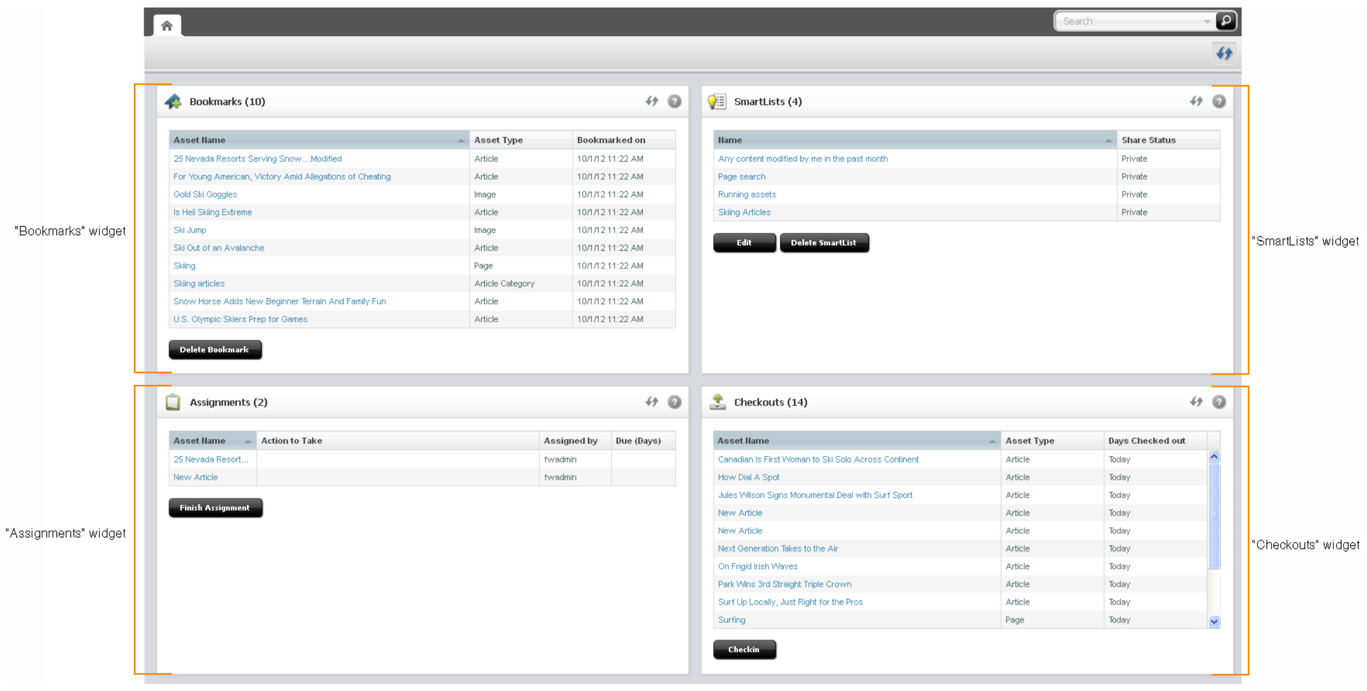
This chapter contains the following topics:

- Section 3.1, "Overview of Dashboard Customization"
- Section 3.2, "Customizing the Dashboard"
- Section 3.3, "Examples of Customizing the Dashboard"

3.1 Overview of Dashboard Customization

When you log in to the Contributor interface, the dashboard is displayed. By default, the dashboard displays the following out-of-the-box widgets: “Bookmarks”, “SmartLists”, “Checkouts” and “Assignments”, as shown in Figure 3–1.

Figure 3–1 Dashboard with Default Widgets



Bookmarks (10)

Asset Name	Asset Type	Bookmarked on
25 Nevada Resorts Serving Snow....Modified	Article	10/1/12 11:22 AM
For Young American, Victory Amid Allegations of Cheating	Article	10/1/12 11:22 AM
Gold Ski Goggles	Image	10/1/12 11:22 AM
Is Heli Skiing Extreme	Article	10/1/12 11:22 AM
Ski Jump	Image	10/1/12 11:22 AM
Ski Out of an Avalanche	Article	10/1/12 11:22 AM
Skiing	Page	10/1/12 11:22 AM
Skiing articles	Article Category	10/1/12 11:22 AM
Snow Horse Adds New Beginner Terrain And Family Fun	Article	10/1/12 11:22 AM
U.S. Olympic Skiers Prep for Games	Article	10/1/12 11:22 AM

SmartLists (4)

Name	Share Status
Any content modified by me in the past month	Private
Page search	Private
Running assets	Private
Skiing Articles	Private

Assignments (2)

Asset Name	Action to Take	Assigned by	Due (Days)
25 Nevada Resort...		fwadmin	
New Article		fwadmin	

Checkouts (14)

Asset Name	Asset Type	Days Checked out
Canadian Is First Woman to Ski Solo Across Continent	Article	Today
How Dial A Spot	Article	Today
Jules Wilson Signs Monumental Deal with Surf Sport	Article	Today
New Article	Article	Today
New Article	Article	Today
Next Generation Takes to the Air	Article	Today
On Frigid Irish Waves	Article	Today
Park Wins 3rd Straight Triple Crown	Article	Today
Surf Up Locally, Just Right for the Pros	Article	Today
Surfing	Page	Today

You can customize the following portions of the dashboard and its widgets:

- Number of columns
- Column width
- Widget's display name, height, and position on the dashboard
- You can also add new widgets.

3.2 Customizing the Dashboard

The system-defined dashboard is generated by the controller element `UI/Layout/CenterPane/DashboardContentsConfig`. You can override this element.

Your dashboard configuration can be global or site-specific. You can customize the default widgets, add new widgets, and delete the ones not required.

To customize the dashboard

Override the element `UI/Layout/CenterPane/DashBoardContentsConfig` by creating your own `DashBoardContentsConfig.jsp` under `CustomElements` and customizing its properties.

Note: You must flush the browser cookies for the changes to take place.

The `UI/Layout/CenterPane/DashBoardContentsConfig` element is shown next, followed by property descriptions in Table 3–1.

Element `UI/Layout/CenterPane/DashboardContentsConfig`:

```
<dashboardconfig>
   <dashboardlayout>
      <numberofcolumns></numberofcolumns>
      <columnwidths></columnwidths>
   </dashboardlayout>
   <components>
      <component id="widgetId">
         <name>widgetName</name>
         <url>widgetURL</url>
         <height>height_in_px</height>
         <dragRestriction>true | false </dragRestriction>
         <column>number_of_column_in_which_to_display_widget</column>
     </component>
         …
         …
         …
   </components>
</dashboardconfig>
```

Table 3–1 Properties in `UI/Layout/CenterPane/DashBoardContentsConfig.jsp`

Property	Description	Value
`<numberofcolumns>`	Number of columns in the dashboard display.	Integer greater than 0. The system default is `2`.
`<columnwidths>`	Comma-separated widths of columns.	For example, if there are `3` columns in `<numberofcolumns>` then the `<columnwidths>` can be `30,30,40`.
`<components>`	This section is used to define dashboard widgets.	N/A
`<component>`	Used to define a single widget.	N/A
`<id>`	ID of the widget.	Alpha-numeric value unique across widgets. Special characters are not allowed.
`<name>`	Displayed name of the widget.	Arbitrary string.
`<url>`	Controller URL.	The file location of the widget in the `UI/Layout/CenterPane/DashBoard/<Your_Element>/` directory.
`<height>`	Height of the widget.	Height in pixels. For example, `300px`.
`<dragRestriction>`	Restricts dragging of the widget.	`true | false`
`<column>`	The column in which the widget is displayed.	`1` to *n*, where *n* is the value specified in `<numberofcolumns>`.

3.3 Examples of Customizing the Dashboard

You can add new widgets to the WebCenter Sites Contributor dashboard. Adding a new widget involves two basic steps:

1. Creating the widget element.
2. Registering the new widget in your custom `DashBoardContentsConfig.jsp` element.

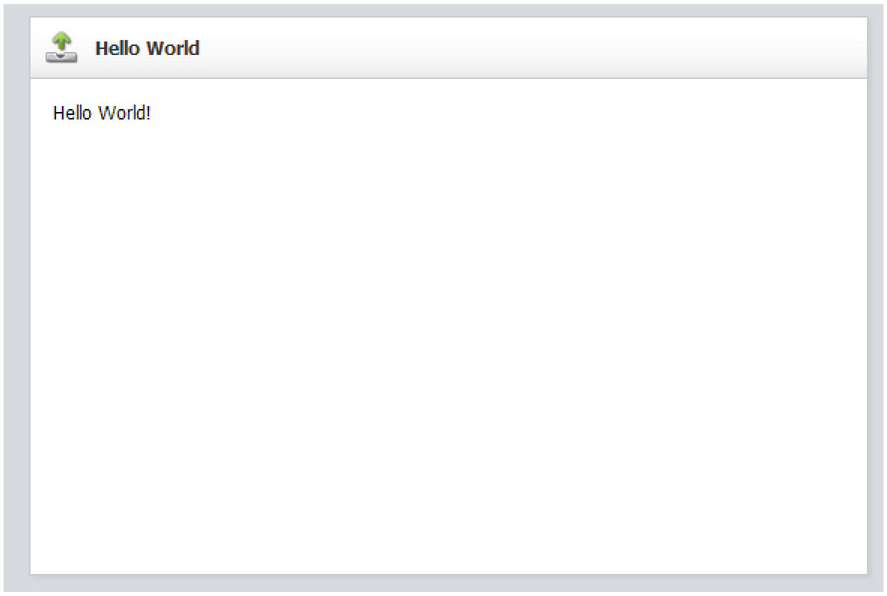
This section illustrates the process of adding a widget to the dashboard. This section contains the following examples:

- Section 3.3.1, "Adding a 'Hello World' Widget"
- Section 3.3.2, "Adding a Widget that Shows Recently Modified Assets"

3.3.1 Adding a ‘Hello World’ Widget

In this section, you will create and register a simple widget, shown in Figure 3–2.

Figure 3–2 ’Hello World’ Widget



To add your widget to the dashboard

1. Create your widget:

   a. Create a JSP element under `CustomElements`. In this example, we name the element `HelloWorldHtml`.

   b. For widget code, you can navigate to the sample file provided with this guide and copy its content.

2. Register your widget (add it to the dashboard):

   a. Open your custom `DashBoardContentsConfig.jsp`, locate the `<components>` section, and add the newly created widget’s specifications. For example:

```
<component id="helloworld">
  <name>Hello World</name>
  <url>Path_to_your_widget_under_CustomElements</url>
  <height>300px</height>
  <closable>false</closable>
  <open>true</open>
  <dragRestriction>true</dragRestriction>
  <style>checkoutPortlet</style>
  <column>2</column>
</component>
```

   b. Go to the `<applicationServer_install_directory>/webapps/<cs_context>/WEB-INF/classes/ReqAuthConfig.xml` file and add the path to the sample element, under the `excludedControllerElements` list. In our example, the path is:

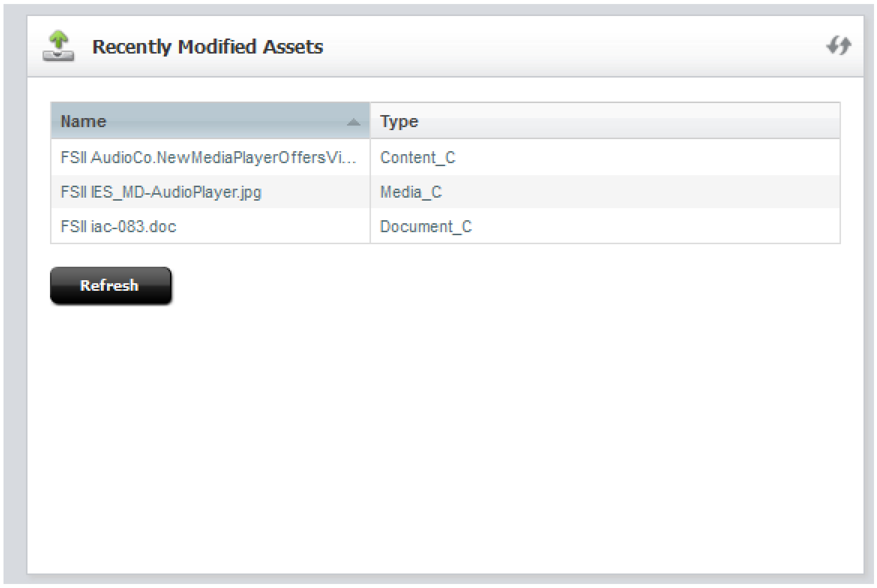
```
<property name="excludedControllerElements">
   <list>Hello World</name>
       <value>/UI/Layout/CenterPane/DashBoard/HelloWorld</value>
   </list>
</property>
```

c. Refresh the home page of your Contributor interface. The new widget is displayed on your dashboard (Figure 3–2).

3.3.2 Adding a Widget that Shows Recently Modified Assets

In this section, you will create a widget that shows which assets were modified in the past week. After completing the steps in this section, your dashboard will display a widget similar to the one in Figure 3–3.

Figure 3–3 'Recently Modified Assets' Widget



Name	Type
FSII AudioCo.NewMediaPlayerOffersVi...	Content_C
FSII IES_MD-AudioPlayer.jpg	Media_C
FSII iac-083.doc	Document_C

To add your widget to the dashboard

1. Create your widget:

   a. Create an `Action` JSP element under `CustomElements`. In this example, we name the element `RecentlyModifiedAssetsAction.jsp`. For the widget code, you can navigate to the sample file provided with this guide and copy its content.

   b. Create a `Json` JSP element for the `Action` element created in the previous step. In this example, we name the element `RecentlyModifiedAssetsJson.jsp`. For the code, you can navigate to the sample file provided with this guide and copy its content. Place the element in the same location as the `RecentlyModifiedAssetsAction.jsp` element.

   c. Create a presentation element under `CustomElements` for your widget. Name the element after the widget element. In this example, we name the display element `RecentlyModifiedAssetsHtml.jsp`. For the code, you can navigate to the sample file provided with this guide and copy its content.

   Note: The presentation element will call the `RecentlyModifiedAssetsAction.jsp` element. Enter the path to that element.

   ---

2. Register your widget (add it to the dashboard):

   a. Open your custom `DashBoardContentsConfig.jsp`, locate the `<components>` section, and add the newly created widget's specifications. For example:

   ```
   <component id="myrecent">
   <!-- a unique identifier for the component. This must be unique among all
   the components. It can be alpha numeric but no special characters allowed
   -->
         <name>Recently Modified Assets</name>
         <url>Path_to_your_custom_widget's presentation_element</url>
         <height>300px</height>
         <closable>false</closable>
         <open>true</open>
         <dragRestriction>false</dragRestriction>
         <style>checkoutPortlet</style>
         <column>2</column>
   </component>
   ```

   b. Go to the `<applicationServer_install_directory>/webapps/<cs_context>/WEB-INF/classes/ReqAuthConfig.xml` file and add the path to the sample element, under the `excludedControllerElements` list. In our example, the path is:

   ```
   <property name="excludedControllerElements">
      <list>Hello World</name>
         <value>/UI/Layout/CenterPane/DashBoard/RecentlyModifiedAssets</value>
      </list>
   </property>
   ```

   c. Refresh the dashboard to see the newly configured widget. For example, see Figure 3–3.

4 Customizing Search Views

This chapter describes how to customize the List and Thumbnail search views of the WebCenter Sites Contributor interface.

This chapter contains the following topics:

- Section 4.1, "Overview of Search View Customization"
- Section 4.2, "Customization Processes"
- Section 4.3, "Customizing Undocked Views"
- Section 4.5, "Customizing Sort Menus, Context Menus, and Tooltips"

4.1 Overview of Search View Customization

When users log in to the WebCenter Sites Contributor interface and access their sites, they can perform a simple or advanced search to locate the required assets. Search results are then presented in either List view or Thumbnail view. This section describes the different search views, which of their features can be customized, and which elements control the configuration of search views. If you need information about search functionality and views, see the *Oracle WebCenter Sites User's Guide*.

This section contains the following topics:

- Section 4.1.1, "Types of Search Views"
- Section 4.1.2, "What You Can Customize in Search Views"
- Section 4.1.3, "View-Rendering Process"
- Section 4.1.4, "Configuration Elements for Search Views"

4.1.1 Types of Search Views

The search results panel can be either undocked or docked and displayed as a List view or Thumbnail view. Thus, the Contributor interface displays the following views:

- List Undocked
- List Docked
- Thumbnail Undocked
- Thumbnail Docked

An undocked view opens only when no assets are open for editing. A docked view is attached to assets in edit mode and therefore opens only when an asset is open in edit mode.

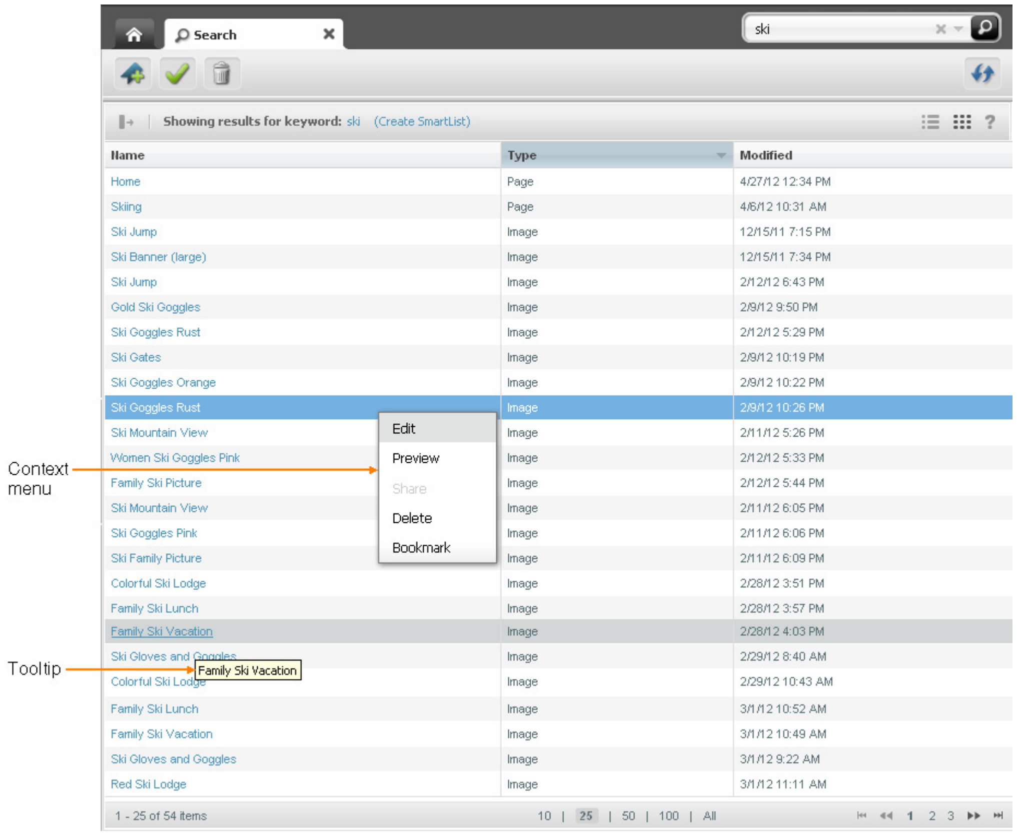
4.1.2 What You Can Customize in Search Views

Figure 4–1 summarizes the features you can customize in List view. Figure 4–2 summarizes the features you can customize in Thumbnail view. For more information, see Section 4.3, "Customizing Undocked Views," which also applies to docked views.

Sort menus, context menus, and tooltips, are customized separately. For more information, see Section 4.5.

Which view opens by default for a given mode depends on your configuration settings and the user's search habits. For example, if you set Thumbnail view as the default view for undocked mode, Thumbnail view will open when the user first runs search in undocked mode and will continue to open until the user switches to List view (search remembers the user's choice until browser cookies are cleared).

Figure 4–1 Customizable Features in List View



Name	Type	Modified
Home	Page	4/27/12 12:34 PM
Skiing	Page	4/6/12 10:31 AM
Ski Jump	Image	12/15/11 7:15 PM
Ski Banner (large)	Image	12/15/11 7:34 PM
Ski Jump	Image	2/12/12 6:43 PM
Gold Ski Goggles	Image	2/9/12 9:50 PM
Ski Goggles Rust	Image	2/12/12 5:29 PM
Ski Gates	Image	2/9/12 10:19 PM
Ski Goggles Orange	Image	2/9/12 10:22 PM
Ski Goggles Rust	Image	2/9/12 10:26 PM
Ski Mountain View	Image	2/11/12 5:26 PM
Women Ski Goggles Pink	Image	2/12/12 5:33 PM
Family Ski Picture	Image	2/12/12 5:44 PM
Ski Mountain View	Image	2/11/12 6:05 PM
Ski Goggles Pink	Image	2/11/12 6:06 PM
Ski Family Picture	Image	2/11/12 6:09 PM
Colorful Ski Lodge	Image	2/28/12 3:51 PM
Family Ski Lunch	Image	2/28/12 3:57 PM
Family Ski Vacation	Image	2/28/12 4:03 PM
Ski Gloves and Goggles	Image	2/29/12 8:40 AM
Colorful Ski Lodge	Image	2/29/12 10:43 AM
Family Ski Lunch	Image	3/1/12 10:52 AM
Family Ski Vacation	Image	3/1/12 10:49 AM
Ski Gloves and Goggles	Image	3/1/12 9:22 AM
Red Ski Lodge	Image	3/1/12 11:11 AM

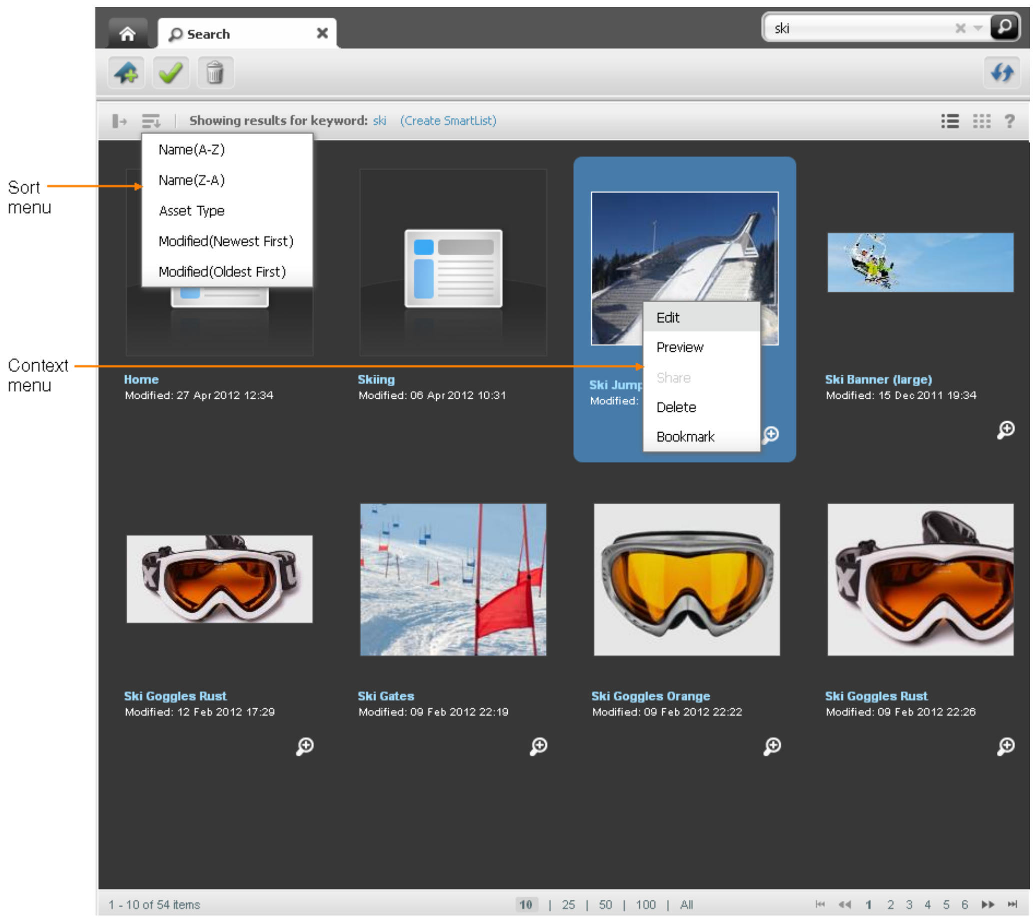
1 - 25 of 54 items 10 | 25 | 50 | 100 | All 1 2 3

Customizable features for List view include:

- Maximum number of items to return
- Number of rows per page
- Fields (columns) to display
- Column display name
- Column width
- Format of date and other fields
- Default sort field and sort order
- Sort menu (docked mode)

- Context (right-click) menu
- Tooltip (docked mode)

Figure 4–2 Customizable Features in Thumbnail View



Customizable features for Thumbnail view include:

- Maximum number of items to return
- Number of rows per page
- Asset types for which special thumbnails will be shown
- Fields to display
- Format of date and other fields
- Default sort field and sort order
- Sort menu
- Context (right-click) menu
- Tooltip (docked mode)

4.1.3 View-Rendering Process

System-defined and custom-defined views are rendered by similar processes. To illustrate, we begin with system-defined views.

System-defined views are rendered by the following elements (JSPs), whose names for undocked and docked views differ only by the `Docked` prefix.

When undocked:

- List view is rendered by `the element:`
  `UI/Layout/CenterPane/Search/View/ListViewHtml`
- Thumbnail view is rendered by the element:
  `UI/Layout/CenterPane/Search/View/ThumbnailViewHtml`

When docked:

- List view is rendered by the element:
  `UI/Layout/CenterPane/Search/View/DockedListViewHtml`
- Thumbnail view is rendered by the element:
  `UI/Layout/CenterPane/Search/View/DockedThumbnailViewHtml`

Rendering of undocked and docked views is similar (except that the names of elements for docked views start with `Docked`). The steps below illustrate the rendering of undocked views.

1. When a user runs a search routine, the search functionality determines the user's current view, which is either the default view or a subsequently chosen view.

   Note: "Default view" is the view that the system renders the first time search is run. (List view is the system-defined default view for both undocked and docked modes.) If the user switches to a different view, search remembers and continues to display the user's choice until browser cookies are cleared.

2. Search functionality reads `UI/Layout/CenterPane/Search/SearchResultsConfig` to obtain the path to the element that will initiate the rendering of the view:
   - If the user is running search for the first time, or continues using the default view, search reads the value of the `<defaultview>` property.
   - If the user's view is other than the default view, search reads the value of either the `<listview>` or `<thumbnailview>` property (depending on which view was determined in step 1).
3. If search determines that List view must be rendered, it reads the element `UI/Layout/CenterPane/Search/View/ListViewConfig` and invokes `UI/Layout/CenterPane/Search/View/ListViewHtml`, which then renders the list view. If search determines that the Thumbnail view must be rendered, it reads the element `UI/Layout/CenterPane/Search/View/ThumbnailViewConfig` and invokes `UI/Layout/CenterPane/Search/View/ThumbnailViewHtml`, which then renders the Thumbnail view.

You can override all of the above system-defined elements by customizing your own identically named elements and placing them under `CustomElements` to actualize the changes shown in Figure 4–1 and Figure 4–2. You can also customize individual features, such as context (right-click) menus, sort menus, and tooltips by using the elements `UI/Layout/CenterPane/Search/View/SearchTopBarConfig`, `UI/Layout/CenterPane/Search/View/ContextMenuConfig`, and `UI/Layout/CenterPane/Search/View/SearchToolTipHtml`.

For a comprehensive list of elements, see Section 4.1.4, "Configuration Elements for Search Views."

4.1.4 Configuration Elements for Search Views

This section summarizes the JSP elements you will use to customize search views.

- **System-defined configuration elements**: You will be configuring identically named elements to customize search views and searches that are global or specific to a site, asset type(s), or site and asset type(s). All of your customized elements should be stored under `CustomElements` (for an example, see Figure 4–3). For a summary of the elements, see the following tables:
  - Table 4–1 lists system-defined configuration elements that define out-of-the-box undocked views (all of the element names end with `Config`).
  - Table 4–2 lists system-defined configuration elements that define out-of-the-box docked views (all of the element names end with `Config`).
  - Table 4–3 lists system-defined elements for customizing a search view's individual features, such as sort menus, context menus, and tooltips (element names end with either `Config` or `Html`).
- **Custom elements**: Table 4–4 lists sample custom elements that are packaged with WebCenter Sites to help illustrate customization code.

Table 4–1 Configuration Elements for Undocked Search Views

Path to Configuration Element (JSP)	Description	See ...
`UI/Layout/CenterPane/Search/SearchResultsConfig`	Element for setting the default search view (List view or Thumbnail view) in undocked mode.	Section 4.3.2, "Setting the Default Undocked View to List or Thumbnail"
`UI/Layout/CenterPane/Search/View/ListViewConfig`	Element for configuring the undocked List view.	Section 4.3.3, "Customizing the Undocked List View"
`UI/Layout/CenterPane/Search/View/ThumbnailViewConfig`	Element for configuring the undocked Thumbnail view.	Section 4.3.4, "Customizing the Undocked Thumbnail View"

Table 4–2 Configuration Elements for Docked Search Views

Path to Configuration Element (JSP)	Description	See ...
`UI/Layout/CenterPane/Search/DockedSearchResultsConfig`	Element for setting the default search view (List view or Thumbnail view) in docked mode.	Section 4.2, "Customization Processes"
`UI/Layout/CenterPane/Search/View/DockedListViewConfig`	Element for configuring the docked List view.	Section 4.2, "Customization Processes"
`UI/Layout/CenterPane/Search/View/DockedThumbnailViewConfig`	Element for configuring the docked Thumbnail view.	Section 4.2, "Customization Processes"

Table 4–3 Configuration and Presentation Elements for Other Features in Search Views

Path to Configuration Element (JSP)	Description	See ...
`UI/Layout/CenterPane/Search/View/SearchTopBarConfig`	Element for configuring fields as sort options in the sort drop-down menus for docked List, undocked Thumbnail, and docked Thumbnail views.	Section 4.5.1, "Customizing Sort Menus"
`UI/Layout/CenterPane/Search/View/ContextMenuConfig`	Element for configuring context (right-click) menus. This element is valid for all search views.	Section 4.5.2, "Customizing Context Menus"
`UI/Layout/CenterPane/Search/View/SearchToolTipHtml`	Element for configuring tooltips for docked views (List and Thumbnail). This element enables you to configure tooltip appearance and custom messages.	Section 4.5.3, "Customizing Tooltips for Search Results"

Table 4–4 Custom Sample Elements for Search Views

Path to Sample Element	Description
`CustomElements/avisports/AVIArticle/UI/Layout/CenterPane/Search/View/ThumbnailViewConfig`	Configuration element for undocked Thumbnail view for the AVIArticle asset type in the avisports sample site.
`CustomElements/avisports/AVIArticle/UI/Layout/CenterPane/Search/View/DockedThumbnailViewConfig`	Configuration element for docked Thumbnail view for AVIArticle asset type in avisports site.
`CustomElements/avisports/AVIImage/UI/Layout/CenterPane/Search/View/ThumbnailViewConfig`	Configuration element for undocked Thumbnail view for the AVIImage asset type in the avisports sample site.
`CustomElements/avisports/AVIImage/UI/Layout/CenterPane/Search/View/DockedThumbnailViewConfig`	Configuration element for docked Thumbnail view for the AVIImage asset type in the avisports sample site.
`CustomElements/avisports/UI/Layout/CenterPane/Search/View/ThumbnailViewConfig`	Configuration element for undocked Thumbnail view for the avisports sample site.
`CustomElements/avisports/UI/Layout/CenterPane/Search/View/DockedThumbnailViewConfig`	Configuration element for docked Thumbnail view for the avisports sample site.

4.2 Customization Processes

When customizing views in undocked and docked mode, you will follow similar procedures. The main differences are the following:

- **Customizing undocked and docked views:**

  When customizing undocked views, you will follow instructions in Section 4.3, "Customizing Undocked Views" and name your configuration elements (JSPs) as shown in that section (also in Table 4–1). When customizing docked views, you will also follow instructions in Section 4.3, "Customizing Undocked Views," but name your configuration elements as shown in Table 4–2 (that is, include the `Docked` prefix).

- **Customizing sort menus, context menus, and tooltips for search views:**

  Elements for creating sort menus, context menus, and tooltips apply to both undocked and docked mode. You will name the elements exactly as shown in Table 4–3 (and Section 4.5, "Customizing Sort Menus, Context Menus, and Tooltips," regardless of mode. For example, context menus for all views are created via UI/Layout/CenterPane/Search/View/ContextMenuConfig (ContextMenuConfig.jsp does not have a counterpart DockedContextMenuConfig.jsp).

- **If you wish to display a field in docked List view or docked Thumbnail view:**

  By default, the UI/Layout/CenterPane/Search/View/DockedListViewConfig element points to the UI/Layout/CenterPane/Search/View/ListViewConfig element to get only the first listed field and display its name in docked List view. The field is defined in the first <field> property, as follows:

  ```
  <field>
      <fieldname>fieldname</fieldname>
      <displayname>DisplayName</displayname>
  ```

  If you want to display any other field name in the docked List view, you will have to specify that name in your custom DockedListViewConfig.jsp element. The same logic applies to displaying a field name in docked Thumbnail view (except that your configuration elements are named ThumbnailViewConfig and DockedThumbnailViewConfig).

4.3 Customizing Undocked Views

Customizing the undocked List and Thumbnail views involves overriding the system-defined elements shown in Table 4–1 by configuring your own identically named elements and placing them under CustomElements.

This section contains the following topics:

- Section 4.3.1, "Basic Steps for Customizing Undocked Views"
- Section 4.3.2, "Setting the Default Undocked View to List or Thumbnail"
- Section 4.3.3, "Customizing the Undocked List View"
- Section 4.3.4, "Customizing the Undocked Thumbnail View"

4.3.1 Basic Steps for Customizing Undocked Views

To customize an undocked view, you can take any combination of the following steps:

- Set the default undocked view to be List or Thumbnail for all asset types or your choice of asset types. To set the view(s), you will override the element UI/Layout/CenterPane/Search/SearchResultsConfig, as shown in Section 4.3.2, "Setting the Default Undocked View to List or Thumbnail."
- Configure the undocked List and/or Thumbnails views. You can specify the number of columns to be displayed in the view(s), configure column names and column widths, specify the sort order of returned items, and more (see Figure 4–1 and Figure 4–2).
  - To configure the List view, you will override the element UI/Layout/CenterPane/Search/View/ListViewConfig, described in Section 4.3.3, "Customizing the Undocked List View."
  - To configure the Thumbnail view, you will override the element UI/Layout/CenterPane/Search/View/ThumbnailViewConfig, described in Section 4.3.4, "Customizing the Undocked Thumbnail View."
- Configure additional features, such as context menus for the views. In this step, you will be configuring JSP elements that are specific to the features of the view

(such as a context menu), rather than the view itself. For more information, see Section 4.5, "Customizing Sort Menus, Context Menus, and Tooltips."

4.3.2 Setting the Default Undocked View to List or Thumbnail

When setting the default search view (List or Thumbnail), you can set it globally for all asset types. You can also specify a default search view for selected asset types of your choice.

To set the default search view(s)

Override the element `UI/Layout/CenterPane/Search/SearchResultsConfig` by creating your own `SearchResultsConfig.jsp` under `CustomElements` and customizing its properties.

The `UI/Layout/CenterPane/Search/SearchResultsConfig` element is shown next, followed by property descriptions in Table 4–5.

Element `UI/Layout/CenterPane/Search/SearchResultsConfig`:

```
<searchconfig>
    <listview>UI/Layout/CenterPane/Search/View/ListView</listview>
    <thumbnailview>UI/Layout/CenterPane/Search/View/ThumbnailView</thumbnailview>
    <defaultview>listview</defaultview>
    <assettypeviews>
        <assettype id="Page" name="Page">listview</assettype>
          …
          …
          …
    </assettypeviews>
</searchconfig>
```

Table 4–5 Properties in `UI/Layout/CenterPane/Search/SearchResultsConfig`

Property	Description	Value
`<listview>`	Path to the ListView controller element.	`UI/Layout/CenterPane/Search/View/ListView` **Note:** Do not change the value of this property.
`<thumbnailview>`	Path to the ThumbnailView controller element.	`UI/Layout/CenterPane/Search/View/ThumbnailView` **Note:** Do not change the value of this property.
`<defaultview>`	Specifies whether List or Thumbnail is the default view. **Note:** The default view is the view that opens the first time search is run. If the user switches the view, search remembers the user's choice until browser cookies are cleared.	`listview | thumbnailview` **Note:** The value of this property is case sensitive.
`<assettypeviews>`	Used to selectively configure a default view for one or more asset types.	N/A
`<assettype id= name= >`	Used to specify the asset type and its default view (which remains until the user either switches to a different view or clears browser cookies). You can specify as many asset types as necessary (one per `<assettype>`).	`<assettype id="`*`unique_identifier`*`" name="`*`AssetTypeName`*`"> listview | thumbnailview </assettype>`

4.3.3 Customizing the Undocked List View

When customizing the List view, you can set the type of content to be returned and its presentation.

To customize the undocked List view

Override the `UI/Layout/CenterPane/Search/View/ListViewConfig` element by creating your own `ListViewConfig.jsp` under `CustomElements` and customizing its properties.

The `UI/Layout/CenterPane/Search/View/ListViewConfig` element is shown next, followed by property descriptions in Table 4–6.

Element `UI/Layout/CenterPane/Search/View/ListViewConfig`:

```
<listviewconfig>
    <numberofitems>1000</numberofitems>
    <numberofitemsperpage>100</numberofitemsperpage>
    <defaultsortfield>  </defaultsortfield>
    <defaultsortorder>  </defaultsortorder>
    <fields>
      <field id="name">
        <fieldname>name</fieldname>
        <displayname>Name</displayname>
        <width>350px</width>
        <formatter>fw.ui.GridFormatter.nameFormatter</formatter>
        <displayintooltip>true</displayintooltip>
    </field>
    <field id="updateDate">
        <fieldname>updateddate</fieldname>
        <displayname>Modified</displayname>
        <!-- <dateformat>MM/dd/yyyy hh:mm a z </dateformat> -->
        <javadateformat>SHORT</javadateformat>
        <width>auto</width>
        <formatter></formatter>
        <displayintooltip>true</displayintooltip>
      </field>
        …
        …
        …
      </fields>
</listviewconfig>
```

Table 4–6 Properties in `UI/Layout/CenterPane/Search/View/ListViewConfig`

Property	Description	Value
`<numberofitems>`	Maximum number of items returned by search.	Integer greater than `0`. **Note:** If `-1` is entered for instance, then all results matching the search criteria are returned.
`<numberofitemsperpage>`	Number of rows per page needed in the search results.	`100` is the default.
`<defaultsortfield>`	Default field that search should sort when fetching search results.	The default is empty. Therefore, search results are displayed by relevance. Configure this element if any other field should be set as the default for sorting.

Table 4–6 (Cont.) Properties in `UI/Layout/CenterPane/Search/View/ListViewConfig`

Property	Description	Value
`<defaultsortorder>`	Sort order used by search.	`ascending | descending` Required only when `<defaultsortfield>` is specified.
`<fields>`	Columns that will be shown in List view. These columns will be shown in the same order as listed under `<fields>`. **Note:** If you are creating an asset type-specific configuration and you wish to display asset type-specific attributes in the search results, you will have to enable the asset type index and attribute search. For more information, see the following sections of the *WebCenter Sites Administrator's Guide*: ■ "Adding Asset Types to the Search Index" ■ "Configuring Attributes for Asset Types" If you skip this procedure, search will use the global index.	N/A
`<field id= >`	Defines a column to be shown in List view.	`<field id="`*`unique_identifier`*`">`
`<fieldname>`	Asset's field name to render in the column.	This name must match the column name in the Lucene index. **Note:** If `locale` is added as the field name, it will be displayed only if the site dimension is enabled.
`<displayname>`	Display name shown in the column header.	Alphanumeric string
`<width>`	Width of the column in pixels.	Width in units of `px` (e.g., `350px`). **Note:** We recommend setting the width to `auto` for the last field.
`<formatter>`	Dojo formatter function to display column values in your preferred format.	The formatter must be made available in a dojo module. See the `modules` property in `UI/Config/GlobalHtml`.

Table 4–6 (Cont.) Properties in `UI/Layout/CenterPane/Search/View/ListViewConfig`

Property	Description	Value
`<displayintooltip>`	Indicates whether the associated field must be listed in the tooltip for docked List view. **Note:** The element `UI/Layout/CenterPane/Search/View/SearchToolTipHtml` renders tooltips and uses the value of this property to determine whether to list the associated field name in the tooltip (the field value will also be listed). Tooltips can be customized only for docked views. For instructions, see Section 4.5.3, "Customizing Tooltips for Search Results."	`true | false`
`<dateformat>`	Applies to date fields only. This is an option to specify a custom date format if the date needs to be displayed in a format other than `javadateformat`.	A valid date format string. **Note:** If `<dateformat>` is used, it takes precedence over `<javadateformat>`.
`<javadateformat>`	Applies to date fields only.	Valid values are `SHORT`, `MEDIUM`, `LONG`, and `FULL`. **Note:** If `<javadateformat>` is omitted or left blank, the system uses `SHORT` by default. If `<dateformat>` is used, it takes precedence over `<javadateformat>`.

4.3.4 Customizing the Undocked Thumbnail View

When customizing the Thumbnail view, you can set the type of content to be returned and its presentation.

To customize the undocked Thumbnail view

Override the element `UI/Layout/CenterPane/Search/View/ThumbnailViewConfig` by creating your own `ThumbnailViewConfig.jsp` under `CustomElements` and customizing its properties.

The `UI/Layout/CenterPane/Search/View/ThumbnailViewConfig` element is shown below, followed by property descriptions in Table 4–7.

Note: Pay particular attention to the following properties: `<formatter>` and `<assettypes>`. While the element `UI/Layout/CenterPane/Search/View/ThumbnailViewConfig` is mostly the same as `UI/Layout/CenterPane/Search/View/ListViewConfig`, the `<formatter>` property is defined differently. Also, the `<assettypes>` property is exclusive to `ThumbnailViewConfig`, where it is used to render thumbnails.

The `<assettypes>` property is described in detail in Section 4.3.4.1, where its usage is illustrated with examples. One of the examples shows you how to supplement video assets with a custom element that displays a video player.

Element `UI/Layout/CenterPane/Search/View/ThumbnailViewConfig`:

```
<thumbnailviewconfig>
```

```
          <numberofitems>1000</numberofitems>
          <defaultsortfield></defaultsortfield>
          <defaultsortorder></defaultsortorder>
          <numberofitemsperpage>12</numberofitemsperpage>
          <formatter>fw.ui.GridFormatter.thumbnailFormatter</formatter>
          <fields>
            <field id="name">
              <fieldname>name</fieldname>
              <displayname>Name</displayname>
              <displayintooltip>true</displayintooltip>
            </field>
            <field id="updateDate">
              <fieldname>updateddate</fieldname>
              <displayname>Modified</displayname>
              <!-- <dateformat>MM/dd/yyyy hh:mm a z </dateformat> -->
              <javadateformat>SHORT</javadateformat>
              <displayintooltip>true</displayintooltip>
            </field>
             …
             …
             …
          </fields>
        <assettypes>
           <assettype id="unique_identifier">
              <type>AVIImage</type>
              <subtype>Image</subtype>
              <element>UI/Layout/CenterPane/Search/View/ImageThumbnail</element>
              <attribute>imageFile</attribute>
            </assettype>
               …
               …
               …
          </assettypes>
      </thumbnailviewconfig>
```

Table 4–7 Properties in UI/Layout/CenterPane/Search/View/ThumbnailViewConfig

Property	Description	Value
`<numberofitems>`	Maximum number of items to be returned by search.	Integer greater than `0`. **Note:** If `-1` is entered for instance, then all results matching the search criteria are returned.
`<numberofitemsperpage>`	Number of rows per page needed in the search results.	`100` is the default value.
`<formatter>`	Dojo formatter function to display values in your preferred format.	The formatter must be made available in a dojo module. See the `modules` property in `UI/Config/GlobalHtml`.
`<defaultsortfield>`	Default sort field that search should sort when fetching search results.	The default is empty. Therefore, search results are displayed by relevance. Configure this element if any other field should be set as a default for sorting.
`<defaultsortorder>`	Sort order used by search.	`ascending | descending` This is required only when `<defaultsortfield>` is specified.

Table 4–7 Properties in `UI/Layout/CenterPane/Search/View/ThumbnailViewConfig`

Property	Description	Value
`<fields>`	Fields that will be shown below the thumbnails in Thumbnail view. These fields will be shown in the same order as listed under `<fields>`. **Note:** If you are creating an asset type-specific configuration and you wish to display asset type-specific attributes in the search results, you will have to enable the asset type index and attribute search. For more information, see the following sections of the *WebCenter Sites Administrator's Guide*: ■ "Adding Asset Types to the Search Index" ■ "Configuring Attributes for Asset Type Index" If you skip this procedure, search will use the global index.	N/A
`<field id=>`	Describes a field under the thumbnail.	`<field id="`*`unique_identifier`*`">`
`<fieldname>`	Asset's field name to render below the thumbnail.	This name must match the column name in the Lucene index. **Note:** If `locale` is added as the field name, it will be displayed only if the site dimension is enabled.
`<displayname>`	Display name to render below the thumbnail.	Alphanumeric string
`<dateformat>`	Applies to date fields only. This is an option to specify a custom date format if the date needs to be displayed in a format other than `javadateformat`.	A valid date format string. **Note:** If `<dateformat>` is used, it takes precedence over `<javadateformat>`.
`<javadateformat>`	Applies to date fields only.	Valid values are `SHORT`, `MEDIUM`, `LONG`, and `FULL`. **Note:** If `<javadateformat>` is omitted or left blank, the system uses `SHORT` by default. If `<dateformat>` is used, it takes precedence over `<javadateformat>`.
`<displayintooltip>`	Indicates whether the associated field must be listed in the tooltip for docked Thumbnail view. **Note:** The element `UI/Layout/CenterPane/Search/View/SearchToolTipHtml` renders tooltips. It uses the value of the `<displayintooltip>` property to determine whether to list the associated field in the tooltip (the field value will also be listed). Tooltips can be customized only for docked views. For instructions, see Section 4.5.3, "Customizing Tooltips for Search Results."	`true | false`

Table 4–7 Properties in `UI/Layout/CenterPane/Search/View/ThumbnailViewConfig`

Property	Description	Value
`<assettypes>`	This section specifies the asset types for which special thumbnails will be shown. Each asset type must have an attribute whose content will be rendered as a thumbnail. For more information as to when this section must be customized, see the following sections: ■ Section 4.3.4.1.1, "If You Wish to Use Static Icons" ■ Section 4.3.4.1.2, "If You Wish to Re-use the System-Defined Image Thumbnail Element" ■ Section 4.3.4.1.3, "If You Wish to Use a Custom Thumbnail-Rendering Element"	N/A
`<assettype id= >`	Describes the asset type for which a special thumbnail will be shown.	`<assettype id="`*`unique_identifier`*`">`
`<type>`	Name of the asset type for which a thumbnail will be rendered.	For more information, see Section 4.3.4.1.2 and Section 4.3.4.1.3.
`<subtype>`	Subtype of the asset type.	For more information, see Section 4.3.4.1.2 and Section 4.3.4.1.3.
`<element>`	Path to the controller element that renders the content specified in `<attribute>` as a thumbnail.	For more information, see Section 4.3.4.1.2 and Section 4.3.4.1.3. **Note:** If you do not specify an element, the system-defined element `UI/Layout/CenterPane/Search/View/GlobalThumbnail` will be used to render static icons, stored in the `images/search` directory. For more information, see Section 4.3.4.1.1.
`<attribute>`	Attribute whose content will be shown as a thumbnail.	For more information, see Section 4.3.4.1.2 and Section 4.3.4.1.3.

4.3.4.1 More About the `<assettypes>` Section in the ThumbnailViewConfig Element

Table 4–7 contains the `<assettypes>` section, which may need to be configured, depending on which features you choose to customize. Various `<assettypes>` configuration scenarios are discussed below in the context of the most commonly performed customizations.

This section contains the following topics:

- Section 4.3.4.1.1, "If You Wish to Use Static Icons"
- Section 4.3.4.1.2, "If You Wish to Re-use the System-Defined Image Thumbnail Element"
- Section 4.3.4.1.3, "If You Wish to Use a Custom Thumbnail-Rendering Element"

4.3.4.1.1 If You Wish to Use Static Icons

If you plan to use your own static thumbnails (stored in the file system), there is no need to customize the `<assettypes>` section of the

UI/Layout/CenterPane/Search/View/ThumbnailViewConfig element, as long as you observe the following conventions:

- The name of the thumbnail icon should not contain spaces (they will be replaced with underscores). The name must be in one of the following formats, depending on the size of the thumbnail:
  - `<assettypename>.png` or `<assettypename>-<subtype>.png` (small thumbnail, 96x96, docked view)
  - `<assettypename>_large.png` or `<assettypename>-<subtype>_large.png` (large thumbnail, 170x170, undocked view)
- The storage location of the icon is the `/images/search` directory of the file system.

If the above conventions are followed, the icon will be automatically rendered as a thumbnail by the `UI/Layout/CenterPane/Search/View/ThumbnailViewConfig` element, which is coded to look for icons in the `/images/search` directory. Naming the icon after the asset type and subtype automatically associates the icon with assets of that type and subtype.

4.3.4.1.2 If You Wish to Re-use the System-Defined Image Thumbnail Element

Customizing the `<assettypes>` section of the `ThumbnailViewConfig.jsp` element is a requirement if you wish to dynamically render *custom images* as thumbnails by re-using the system-defined element `ImageThumbnailHtml.jsp`. This element processes images that are associated with image attributes belonging to specific asset types and/or subtypes.

To re-use the System-Defined ImageThumbnailHtml.jsp

In your custom `ThumbnailViewConfig.jsp`, do the following:

1. Specify the asset types that require a custom image thumbnail. **Each asset type must have an image attribute.**

```
<assettypes>
    <assettype>
        <type>Name_of_AssetType_containing_the_image_attribute</type>
        <subtype>Name_of_subtype_containing_the_image_attribute</subtype>
        <element>UI/Layout/CenterPane/Search/View/ImageThumbnail</element>
        <attribute>Name_of_imageAttribute_containing_the_image</attribute>
    </assettype>
     …
     …
    </assettypes>
```

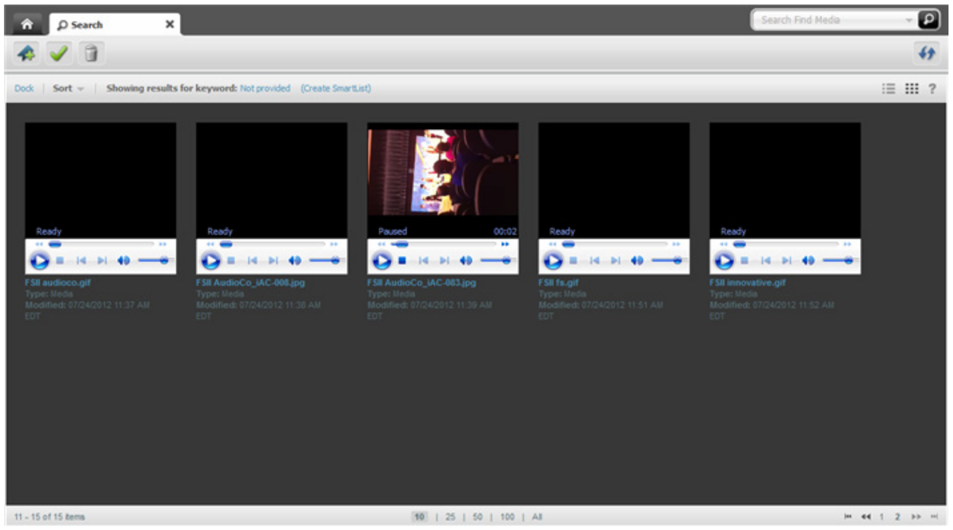
2. For `<element>`, specify the path to the system-defined element `ImageThumbnailHtml.jsp`, exactly as shown in the sample code above.

4.3.4.1.3 If You Wish to Use a Custom Thumbnail-Rendering Element

Customizing the `<assettypes>` section of the `ThumbnailViewConfig.jsp` is a requirement if you plan to use a custom element that dynamically renders the content of an asset type's (or subtype's) `blob` attribute as a thumbnail.

In the example below, you will create elements that work together to render video thumbnails. Figure 4–3 displays a sample video thumbnail view, which you can reproduce by following the steps in this section.

Figure 4–3 Sample Video Thumbnail View



The steps below provide guidelines for (1) creating elements that work together to dynamically render video thumbnails, and (2) customizing the `<assettypes>` section of the `ThumbnailViewConfig` element.

Note: To make this sample work, ensure that you have assets with a `blob` attribute and video files for that `blob` attribute are uploaded to your site's directory.

To create elements that render video thumbnails

1. Write a video thumbnail `Action` element that uses the AssetAPI and gets the URL of the `blob` using `BlobUtil` for the video attribute specified in the element. (The element can be named as you wish, but it must end in `Action`. The element should be stored in a directory under `CustomElements`.)

   A sample element named `VideoThumbnailAction.jsp` is available in the zip file containing this guide.

2. Write a video thumbnail `Html` element, which takes the URL built in the previous step and renders the video and other asset details below the thumbnail. (The element can be named as you wish, but it must end in `Html`. The element should be stored in a directory under `CustomElements`.) This `Html` element calls the `Action` element in step 1.

   A sample element named `VideoThumbnailHtml.jsp` is available in the zip file containing this guide.

3. To use the video thumbnail `Html` element, configure the `<assettype>` property in your custom `ThumbnailViewConfig.jsp` element as shown below:

   ```
   <assettype>
       <type>Name_of_AssetType_containing_blob_attribute</type>
       <subtype>Name_of_asset_subtype</subtype>
       <element>CustomElements/path_to_your_element/Element</element>
       <attribute>Name_of_attribute_containing_video</attribute>
   </assettype>
   ```

   A sample element named `ThumbnailViewConfig.jsp` is available in the zip file containing this guide.

4.4 Customizing Docked Views

Methods for customizing docked views are similar to those for undocked views. The main differences are outlined in Section 4.2, "Customization Processes."

4.5 Customizing Sort Menus, Context Menus, and Tooltips

Features discussed in this section can be customized for undocked views, docked views, or both, as shown in Table 4–8.

This section contains the following topics:

- Section 4.5.1, "Customizing Sort Menus"
- Section 4.5.2, "Customizing Context Menus"
- Section 4.5.3, "Customizing Tooltips for Search Results"

Table 4–8 Customizing Other Features for Search Views

Customization Option	Undocked List	Undocked Thumbnail	Docked List View	Docked Thumbnail	See ...
Sort Menus	No	Yes	Yes	Yes	Section 4.5.1
Context Menus	Yes	Yes	Yes	Yes	Section 4.5.2
Tooltips for Search Results	No	No	Yes	Yes	Section 4.5.3

4.5.1 Customizing Sort Menus

Sort menus can be customized only for the views listed in Table 4–8. You can specify which sort fields to display in a sort menu. You can also specify sort order for each field.

To customize a sort menu

Override the element `UI/Layout/CenterPane/Search/View/SearchTopBarConfig` by creating your own `SearchTopBarConfig.jsp` under `CustomElements` and customizing its properties.

The `UI/Layout/CenterPane/Search/View/SearchTopBarConfig` element is shown next, followed by property descriptions in Table 4–9.

Element `UI/Layout/CenterPane/Search/View/SearchTopBarConfig`:

```
<sortconfig>
     <sortfields>
        <sortfield id="unique_identifier">
           <fieldname>name</fieldname>
           <displayname>Name(A-Z)</displayname>
           <sortorder>ascending</sortorder>
        </sortfield>
        <sortfield id="unique_identifier">
           <fieldname>name</fieldname>
           <displayname>Name(Z-A)</displayname>
           <sortorder>descending</sortorder>
        </sortfield>
        <sortfield id="unique_identifier">
           <fieldname>AssetType_Description</fieldname>
           <displayname>Asset Type</displayname>
           <sortorder>ascending</sortorder>
        </sortfield>
```

```
                …
                …
                …
     </sortfields>
</sortconfig>
```

Table 4–9 Properties in `UI/Layout/CenterPane/Search/View/SearchTopBarConfig`

Property	Description	Value
`<sortfield id= >`	Describes the search index field by which to sort search results.	`id="`*`unique_identifier`*`"`
`<fieldname>`	Name of the search index field. **Note:** The same field can be repeated multiple times to provide multiple sort orders.	For example, `name` in the code above.
`<displayname>`	Display name of the user-readable field.	For example, `Name` in the code above.
`<sortorder>`	Sort order.	`ascending	descending`

4.5.2 Customizing Context Menus

Context menus can be customized for all views. A context menu is a right-click menu of actions (such as Edit, Preview, and Bookmark) to be performed on items that are returned as search results

To customize a context menu

Override `UI/Layout/CenterPane/Search/View/ContextMenuConfig` by creating your own `ContextMenuConfig.jsp` under `CustomElements` and customizing its properties.

The `UI/Layout/CenterPane/Search/View/ContextMenuConfig` element is shown next, followed by property descriptions in Table 4–10.

Element `UI/Layout/CenterPane/Search/View/ContextMenuConfig`:

```
<contextmenuconfig>
    <contextmenus>
      <menu id="unique_identifier">
        <label>Edit</label>
        <functionid>edit</functionid>
      </menu>
      <menu id="unique_identifier">
        <label>Preview</label>
        <functionid>preview</functionid>
      </menu>
      <menu id="unique_identifier">
        <label>Bookmark</label>
        <functionid>bookmark</functionid>
        <bulkoperation>yes</bulkoperation>
      </menu>
       …
       …
       …
    </contextmenus>
</contextmenuconfig>
```

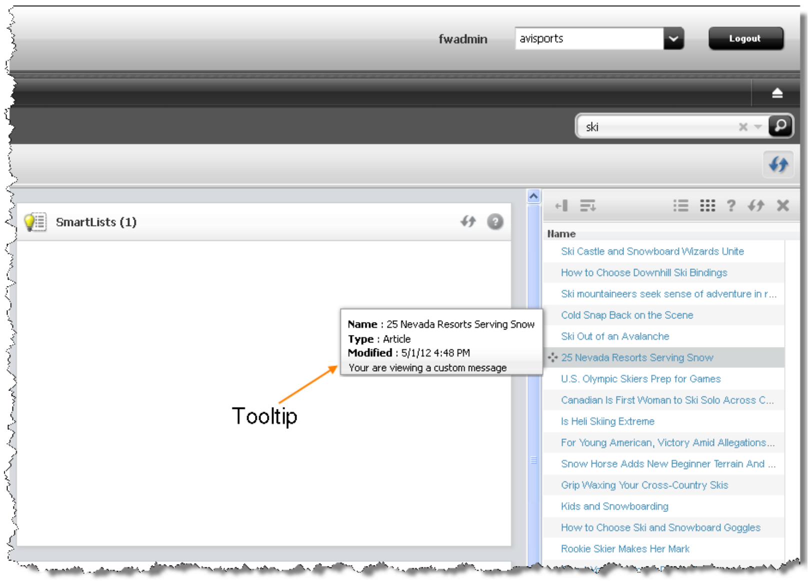
Table 4–10 Properties in `UI/Layout/CenterPane/Search/View/ContextMenuConfig`

Property	Description	Value
`<menu id= >`	Describes the context menu item.	`id="`*`unique_identifier`*`"`
`<label>`	Display name of the menu item.	A name that suggests the action to perform on the asset. For example: `Edit`
`<functionid>`	Action name as defined in the `fw.ui.document.AssetDocument` section of the `UI/Config/GlobalHtml` element.	A value (action) that applies to the asset.

4.5.3 Customizing Tooltips for Search Results

Tooltips can be customized only for docked views. Docked views are displayed in a limited space and therefore provide a limited amount of information about the assets that are returned as search results. Tooltips are a way of displaying more information about the returned assets. For example, you can customize tooltips to display field names and values in addition to those already displayed in docked mode, as shown in Figure 4–4. You can also customize tooltips to display custom messages, and you can modify the appearance of tooltips.

Figure 4–4 Tooltip in Undocked List View



Name : 25 Nevada Resorts Serving Snow
Type : Article
Modified : 5/1/12 4:48 PM
Your are viewing a custom message

Ski Castle and Snowboard Wizards Unite
How to Choose Downhill Ski Bindings
Ski mountaineers seek sense of adventure in r...
Cold Snap Back on the Scene
Ski Out of an Avalanche
25 Nevada Resorts Serving Snow
U.S. Olympic Skiers Prep for Games
Canadian Is First Woman to Ski Solo Across C...
Is Heli Skiing Extreme
For Young American, Victory Amid Allegations...
Snow Horse Adds New Beginner Terrain And ...
Grip Waxing Your Cross-Country Skis
Kids and Snowboarding
How to Choose Ski and Snowboard Goggles
Rookie Skier Makes Her Mark

The default tooltip for docked search results is rendered by the element `UI/Layout/CenterPane/Search/View/SearchToolTipHtml`. This element renders the tooltip as a box (as shown in Figure 4–4). Within the box, it renders the name of each field in the `<fields>` section of `UI/Layout/CenterPane/Search/View/ListViewConfig` (or `UI/Layout/CenterPane/Search/View/ThumbnailViewConfig`), **but only if the field's `<displayintooltip>` property is set to `true`**. For example, the `Name`, `Type`, and `Modified` fields in the `ListViewConfig.jsp` below are displayed as part of the tooltip in Figure 4–4, given that `<displayintooltip>` is set to `true`:

```
<fields>
        <field>
            <fieldname>name</fieldname>
            <displayname>Name</displayname>
            <width>350px</width>
```

```
    <formatter>fw.ui.GridFormatter.nameFormatter</formatter>
    <displayintooltip>true</displayintooltip>
</field>
<field>
    <fieldname>type</fieldname>
          <displayname>Type</displayname>
          <width>auto</width>
          <formatter></formatter>
          <displayintooltip>true</displayintooltip>
</field>
<field>
          <fieldname>updateddate</fieldname>
          <displayname>Modified</displayname>
          <javadateformat>SHORT</javadateformat>
          <width>auto</width>
          <formatter></formatter>
          <displayintooltip>true</displayintooltip>
</field>
```

Note: The `UI/Layout/CenterPane/Search/View/SearchToolTipHtml` element also renders field values. However, customized messages and changes to tooltip appearance must be coded in the custom `SearchToolTipHtml.jsp` element.

To create a tooltip or add fields to the tooltip

1. To create a tooltip, override the element `UI/Layout/CenterPane/Search/View/SearchToolTipHtml` by creating your own `SearchToolTipHtml.jsp` under `CustomElements`.
2. To add fields to the tooltip, add the fields to your custom `ListViewConfig.jsp` or `ThumbnailViewConfig.jsp` and set each field's `<displayintooltip>` property to `true`.
3. To display a custom message in the tooltip (custom or system-defined) or to change the appearance of the tooltip, code your custom `SearchToolTipHtml.jsp` element. For example:

```
<%@ taglib prefix="cs" uri="futuretense_cs/ftcs1_0.tld"
%><%@ taglib prefix="ics" uri="futuretense_cs/ics.tld"
%><cs:ftcs>
<style>
.customSearchTooltip {
     font-weight: bold;
     color: #333;
     font-style: italic;
}
</style>

<div class='customSearchTooltip'>
     You are Viewing a Custom Tooltip
</div>
</cs:ftcs>
```

5 Customizing Global Properties, Toolbar, and Menu Bar

This chapter describes `UI/Config/GlobalHtml`, the global configuration element. This chapter also shows you how to customize the features the global element defines for the WebCenter Sites Contributor interface.

This chapter contains the following topics:

- Section 5.1, "Customizing Global Configuration Properties"
- Section 5.2, "Customizing the Toolbar"
- Section 5.3, "Customizing the Menu Bar"

5.1 Customizing Global Configuration Properties

Global configuration properties are used to set display conditions for the Contributor interface across all content management sites.

This section contains the following topics:

- Section 5.1.1, "Overview of the Configuration Properties"
- Section 5.1.2, "Modifying Default Configuration Properties"
- Section 5.1.3, "Adding Custom Configuration Properties"

5.1.1 Overview of the Configuration Properties

The client-side framework retrieves its main configuration settings from the server-side controller element `UI/Config/GlobalHtml`. This presentation element serves JavaScript code, which is executed by the client-side application at startup. The JavaScript code defines a JavaScript function, whose name is given as a request parameter by the client-side application:

```
<%@ taglib prefix="cs" uri="futuretense_cs/ftcs1_0.tld"%>
<cs:ftcs>
webcenter.sites['${param.namespace}'] = function (config) {
    config.maxTabCount = 50;
    config.defaultView =  …;
     … merge
}
</cs:ftcs>
```

The `config` object is then manipulated as needed in the function body, by setting the properties expected by the client-side application.

In addition, as explained below, the client-side application is capable of retrieving additional configuration properties from the server-side, which allows to merge settings from multiple sources, without having to duplicate the global properties in multiple locations.

5.1.2 Modifying Default Configuration Properties

Table 5–1 describes the system-defined configuration properties and indicates which properties can be modified.

Table 5–1 Configuration Properties in `UI/Config/GlobalHtml`

Property Name	Description	Values and Examples
`maxTabCount`	Maximum number of tabs that can remain open simultaneously. A tab is the tab of an open asset.	Any integer greater than `0`. Ex: `config.maxTabCount = 30;`
`enableContextMenu`	Indicates whether the default browser context (right-click) menu should be enabled when users work in web mode.	`true | false` Ex: `config.enableContextMenu = true;`
`enableWebMode`	Indicates whether web mode should be enabled. When this property is set to `false`, users are able to work only with assets in form mode and use the preview functionality. By default, this property takes the value of the `xcelerate.enableinsite` property, found in `futuretense_xcel.ini`.	`true | false` **Ex:** `config.enableWebMode = true;`
`enableDatePreview`	Indicates whether date-based preview should be enabled. By default, this property takes the value of the `cs.sitepreview` property, found in `futuretense_xcel.ini`.	`true | false` **Ex:** `config.enableDatePreview = false;`
`enablePreview`	Indicates whether preview is allowed. By default, this property takes the value of the "Preview method" attribute in the "Edit Site" screen (accessible from the Administrator interface: Select **Admin** tab, expand **Sites**, double-click ***SampleSite***, and select **Edit**).	`true | false` **Ex:** `config.enablePreview = true;`

`defaultView`	Defines the preferred view for working with assets (i.e., whether assets will be viewed, by default, in form mode or web mode).
	Note: An asset will be opened in web mode only if the asset is associated with a default template.
		The expected value is one of the following:
		■ `"default": "form" | "web"`
		■ `"`*`assetType`*`" : "form" | "web"`
		■ `"`*`assetType/subtype`*`": "form" | "web"`
		where *`assetType`* is a valid asset type name, and *`subtype`* is a valid subtype or definition name.
		Ex:
		`config.defaultView = {`
		` "default": "form",`
		` "AVIArticle": "web",`
		` "Page/AVISection": "web"`
		`}`

Table 5–1 Configuration Properties in `UI/Config/GlobalHtml`

Property Name	Description	Values and Examples
`toolbars`	For each type of view, this property defines the list of available toolbar actions.	See Section 5.2, "Customizing the Toolbar."
`toolbarButtons`	Used to define the behavior of specific toolbar buttons.	See Section 5.2.2.3, "Customizing the Toolbar with Custom Actions."
`menubar`	Defines the list of available actions in the menu bar.	See Section 5.3, "Customizing the Menu Bar."
`documents`	Registers available implementations of documents.	Do not modify the value of this property. The only supported value is `asset`.
`views`	Registers view implementations.	Do not modify the value of this property.
`controllers`	Registers controller implementations and the set of actions supported by each controller.	Do not modify the value of this property.
`roles`	Contains the list of roles for the currently logged in user.	Do not modify the value of this property.
`supportedTypes`	Contains the list of asset types that can be edited from the Contributor interface.	Do not modify the value of this property.
`searchableTypes`	Contains the list of asset types that can be searched from the Contributor interface.	Do not modify the value of this property.
`token`	Used for security when uploading binary file.	Do not modify the value of this property.
`sessionid`	Used for security when uploading binary file.	Do not modify the value of this property.

5.1.3 Adding Custom Configuration Properties

In addition to retrieving the global properties, stored in `UI/Config/GlobalHtml`, the Contributor application will attempt to retrieve additional settings in `UI/Config/SiteConfig` *and any element present in* `UI/Config`. Depending on the requirement, this allows you to set global properties, or site-specific properties, without having to replicate all the properties defined in `UI/Config/GlobalHtml`, but only the properties that actually change.

This section contains the following topics:

- Section 5.1.3.1, "Adding Custom Global Properties"
- Section 5.1.3.2, "Adding Site-Specific Properties"

5.1.3.1 Adding Custom Global Properties

Custom global properties are meant to be shared across all sites on a given content management system. The recommended approach consists of creating a custom configuration element defined as follows:

- The presentation element name must start with `UI/Config/`.
- The element code must follow the pattern shown in Section 5.1.1, "Overview of the Configuration Properties."

For example, you may want to:

- Override the value of `maxTabCount` for all sites.

- Override the default view for Page assets.
- And, define an additional custom property called `foo`.

To do this, you could create an element called `UI/Config/MyConfigHtml`, containing the following code:

```
<%@ taglib prefix="cs" uri="futuretense_cs/ftcs1_0.tld" %>
<cs:ftcs>
webcenter.sites['${param.namespace}'] = function (config) {
    // override existing properties
    config.maxTabCount = 60;
    config.defaultView.Page = "form";

    // add custom properties
    config.foo = "bar";

}
</cs:ftcs>
```

5.1.3.2 Adding Site-Specific Properties

In some cases, the Contributor interface must be configured differently for each content management site. The recommended approach consists of overriding the core controller element called `UI/Config/SiteConfig` by creating an element as follows:

`CustomElements/`*`siteName`*`/UI/Config/SiteConfigHtml`

where *`siteName`* is the name of the content management site (for instance, `avisports`).

For example, the avisports demo site enforces web mode as the default mode for assets of type Page and AVIArticle. This is done by defining the JSP element `CustomElements/avisports/UI/Config/SiteConfigHtml`, and providing the following settings:

```
<%@ taglib prefix="cs" uri="futuretense_cs/ftcs1_0.tld" %>
<%@ taglib prefix="ics" uri="futuretense_cs/ics.tld" %>
<cs:ftcs>
webcenter.sites['${param.namespace}'] = function (config) {
    // default view modes for avisports
    config.defaultView.Page = "web";
    config.defaultView.AVIArticle = "web";
}
</cs:ftcs>
```

Loading of Configuration Elements

The global configuration element is always loaded first. Additional configuration elements are loaded in alphabetical order. For instance, using the examples above, configuration properties would be loaded in the following order:

1. `UI/Config/GlobalHtml`
2. `UI/Config/MyConfigHtml`
3. `UI/Config/SiteConfigHtml`

Property Values

The value of some properties is, in some cases, an object. That is:

```
config.someProperty = {
  foo: "bar",
  x: 123
};
```

When partially overriding this property, it is important to distinguish between the following types of code:

```
config.someProperty = {
  x: 3456
};
```

vs.

```
config.someProperty.x = 3456;
```

In the first case, the property `foo` is overridden as "undefined", whereas in the second case, the original value of `foo` is preserved.

5.2 Customizing the Toolbar

The toolbar can be customized to list actions for operating on assets in web mode or form mode. The toolbar can be further customized per asset type and subtype.

This section contains the following topics:

- Section 5.2.1, "Overview of Toolbar Customization"
- Section 5.2.2, "Examples of Toolbar Customization"

5.2.1 Overview of Toolbar Customization

The global configuration element (`UI/Config/GlobalHtml`) describes for each type of view (such as web mode inspect, web mode edit, form mode edit, and form mode inspect), the list of actions to display in the toolbar to the user. This is done through the `toolbars` property. Its value is an object with the following syntax:

```
config.toolbars = {
    "viewAlias": [action_1, action_2,  …],
    or:
    "viewAlias": {
        "view_mode_1": [action_1, action_2,  …],
        "view_mode_2": [action_1, action_2,  …]
    }
 …
}
```

where:

`viewAlias` indicates for which type of view this toolbar must be used. The alias must match one of the view aliases defined in the `config.views` section.

`action_i` is an action name. For standard actions, such as `save` and `approve`, the action name is automatically mapped to a given icon, title, alternate text, and so on. For more information about standard actions, custom actions, or customizing the appearance of a custom button, see Section 5.2.2, "Examples of Toolbar Customization."

`view_mode_i` is one of the modes supported by the view (typically, `edit` or `view`).

5.2.2 Examples of Toolbar Customization

This section contains the following topics:

- Section 5.2.2.1, "Customizing the Toolbar with Standard Actions for Web Mode"
- Section 5.2.2.2, "Customizing the Toolbar with Standard Actions for Asset Type and Subtype"
- Section 5.2.2.3, "Customizing the Toolbar with Custom Actions"

5.2.2.1 Customizing the Toolbar with Standard Actions for Web Mode

The following configuration determines which toolbar actions are available in web mode for all asset types:

```
config.toolbars = {
    ( …)

    "web": {
      "edit": ["form-mode", "inspect", "separator", "save", "preview",
               "approve", "delete", "separator", "changelayout",
               "separator", "checkincheckout", "refresh"],
      "view": ["form-mode", "edit", "separator", "preview", "approve",
               "delete", "separator", "checkincheckout", "refresh"]

    ( …)

}
```

The above configuration defines two lists of actions (`edit` and `view`), corresponding to the asset's views: Edit and Inspect.

Note: To find the set of standard actions, refer to the list of actions specified in the following properties under the `controllers` property:

- `fw.ui.document.AssetDocument` (all actions supported by assets)
- `fw.ui.controller.InsiteController` (all actions supported by the view controller).

5.2.2.2 Customizing the Toolbar with Standard Actions for Asset Type and Subtype

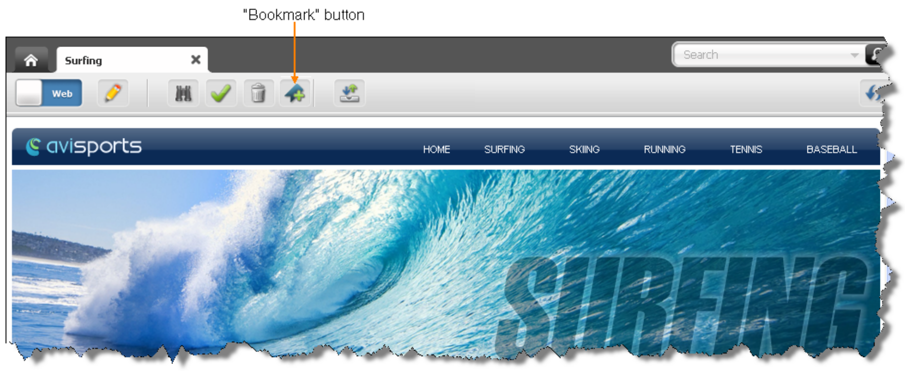
Each toolbar configuration can be customized by asset type and subtype by adding a property named:

- *`viewAlias/assetType`*
- *`viewAlias/assetType/assetSubtype`*

For example, we can add the bookmark/unbookmark buttons for Page assets in web mode. In a custom configuration element (such as `CustomElements/avisports/UI/Config/SiteConfigHtml`), we can add the following property:

```
config.toolbars["web/Page/AVISection"] = {
      "edit": config.toolbars.web.edit, // reuse default for edit mode
       "view": ["form-mode", "edit", "separator", "preview", "approve", "delete",
                       "bookmark", "unbookmark", "separator",
                        "checkincheckout",  "refresh"]
}
```

Inspecting the "Surfing" Page asset now shows the following toolbar:



Note: Keep in mind the following:

- We are customizing only the `view` mode. When a Page/AVISection asset is being edited in web mode, the standard toolbar will be shown.
- The "Bookmark" and "Unbookmark" buttons are not shown simultaneously, since they both depend on the asset's current state (whether it is already bookmarked or not).

5.2.2.3 Customizing the Toolbar with Custom Actions

Custom actions can be defined by adding new entries to the `config.toolbarButtons` property, as follows:

```
config.toolbarButtons.<customActionName> = {
  src: <path_to_icon>,
  onClick: <click_handler>
}
```

For example, let's define the following `helloWorld` custom action:

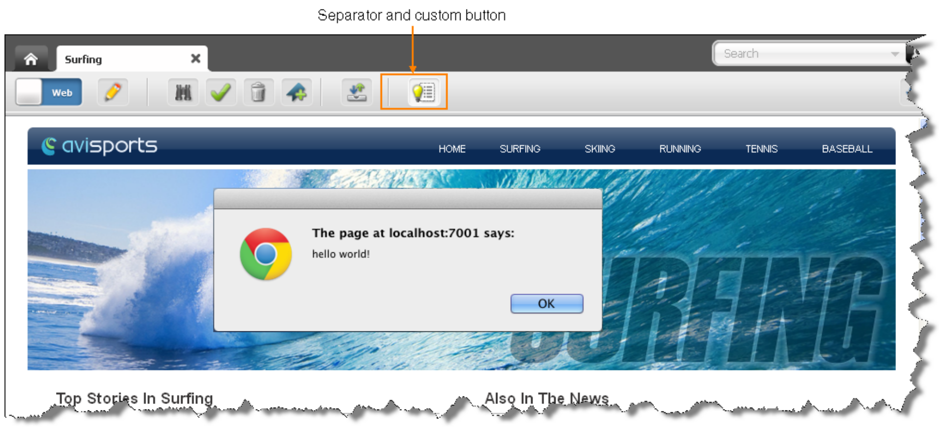
```
config.toolbarButtons.helloWorld = {
      src: 'js/fw/images/ui/ui/toolbarButton/smartlist.png",
      onClick: function () {
          alert('Hello World!!');
      }
}
```

The `helloWorld` action can now be referenced from a toolbar configuration as follows (we will reuse our example from the previous section):

```
config.toolbars["web/Page/AVISection"] = {
    "view":
          ["form-mode", "edit", "separator", "preview", "approve",
           "bookmark", "unbookmark", "separator",
           "checkincheckout", "separator", "helloWorld", "refresh"],

    "edit": config.toolbar.web.edit // reuse default web mode toolbar
}
```

Note: In this example, we have added a separator (a vertical line) and the custom button at the end of the toolbar:



The page at localhost:7001 says:

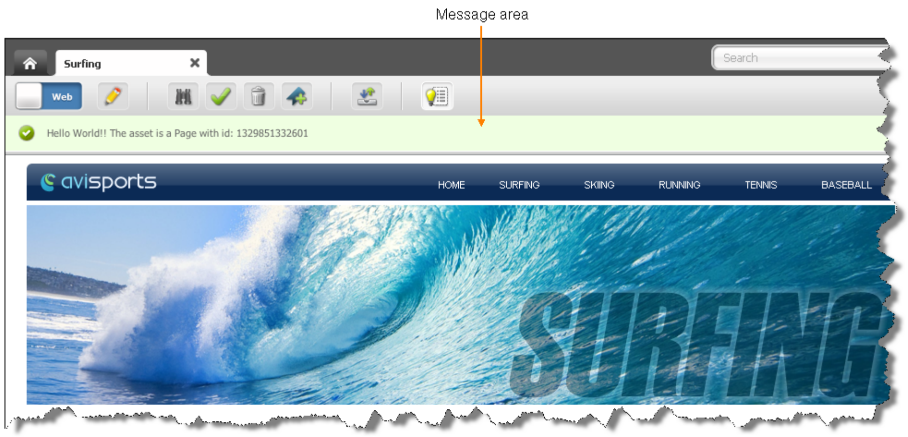
hello world!

A more elaborate example would involve, for instance, retrieving the id and type of the current asset, and giving some feedback in the message area:

```
config.toolbarButtons.helloWorld = {
  src: 'js/fw/images/ui/ui/toolbarButton/smartlist.png",
  onClick: function () {
var doc = SitesApp.getActiveDocument(), // the document in the active tab
    asset = doc.get('asset'),      // the asset object
    view = SitesApp.getActiveView();     // the active view

view.info('Hello World!! The asset is a ' + asset.type + ' with id: '
+ asset.id);

  }
}
```



Hello World!! The asset is a Page with id: 1329851332601

5.3 Customizing the Menu Bar

The menu bar can be customized to support menus for operating on assets of a certain type. or type/subtype. Submenus can be actionable items, additional menus, or menu separators.

This section contains the following topics:

- Section 5.3.1, "Overview of Menu Bar Customization"
- Section 5.3.2, "Adding a Custom Action to the Menu Bar"

5.3.1 Overview of Menu Bar Customization

The menu bar configuration is defined by the `config.menubar` property:

```
config.menubar = {
    "key_i": [
          //menu_i
          {
              "id": "menu_id",
              "label": "menu_label",
              "children": [
                  //submenus
                  //- actionable menu item
                  {
                     label: 'menu_item_label',
                     action: 'action_name' | click_handler
                  },
                  //- deferred pop-up menu
                  {
                     label: 'menu_item_label',
                     deferred: 'controller_element',
                     cache: true|false
                  },
                  //- pop-up menu
                  {
                     label: 'menu_item_label',
                     children: [
                         // submenu_1
                         {
                            label: 'menu_item_label',
                            action: 'action_name' | click_handler
                         }'
                         // submenu_2
                         {
                            label: 'menu_item_label',
                            action: 'action_name' | click_handler
                         },
                          …
                          …
                          …
                     ]
                  },
                  //- menu item separator
                  {separator: true}

              //additional menu_i
               …
               …
               …
              ]
          }
```

where:

key_i is one of the following:

- *`default`* – Defines the default menu bar.
- *`assetType`* – Defines the customized menu bar for all assets of type *`assetType`*.
- *`assetType/subtype`* – Defines the customized menu bar for all assets of type *`assetType`* and subtype *`subtype`*.

`//menu_i` starts a section that describes each top menu, where:

- *`menu_id`* is the identifier of the menu.
- *`menu_label`* is the display name of the menu.
- ***`submenus`*** can be any of the following:
  - An **actionable menu item** (clicking the menu item produces an action), where:
    - `label` specifies the display name of the menu item.
    - `action` can be any action supported by a controller, such as `edit` and `inspect`, or a custom click handler (see the customization example in Section 5.3.2, "Adding a Custom Action to the Menu Bar").

    Note: When a given action is not supported by the current document/view, it appears disabled (greyed out) in the menu.

    For instance, a menu item triggering a `save` action is defined as follows:

    ```
    {
        label: "Save",
        action: "save"
    }
    ```

  - A **deferred pop-up menu** (the pop-up menu is determined dynamically by running a controller element on the server-side), where:
    - `label` specifies the display name of the menu item.
    - `deferred` specifies a controller element name, such as `UI/Data/StartMenu/New`.
    - `cache` is a boolean indicating whether the output of the controller element should be cached or not.

    For instance, the "New" pop-up menu, which reads all available start menu items for the current site/user is defined as follows:

    ```
    {
        label: "New",
        deferred: "UI/Data/StartMenu/New",
        cache: true
    }
    ```

  - A **pop-up menu** (the child menu items are hard wired in the configuration itself).
  - A **menu item separator** (a horizontal line), which is used to group menu entries together.

5.3.2 Adding a Custom Action to the Menu Bar

In this example, we want to add the `helloWorld` custom action defined in Section 5.2.2.3, "Customizing the Toolbar with Custom Actions" to the menu bar (to run the custom onClick handler). We can add this action by adding a new entry to the menu bar called "Custom Menu", with a single menu item called "Hello World", which will trigger the custom action. Our steps are the following:

1. First, we reuse the default menu bar, and add to it. The simplest way to do this is to make a copy of the original array:

```
config.menubar["Page/AVISection"] = config.menubar["default"].slice(0);
```

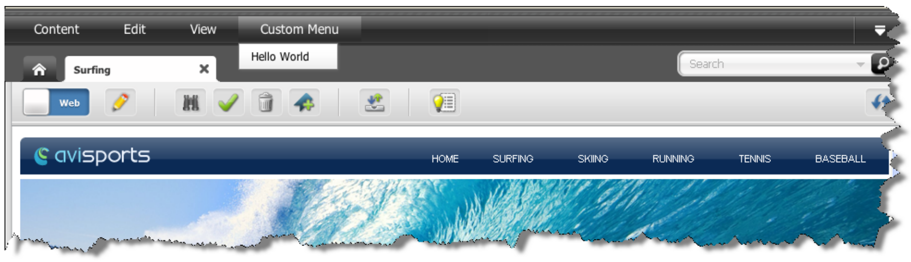
2. We can then add our menu as follows:

```
config.menubar["Page/AVISection"].push(
    "id": "myCustomMenu",
    "label": "Custom Menu",
    "children": [
        // Children go here
    ]
);
```

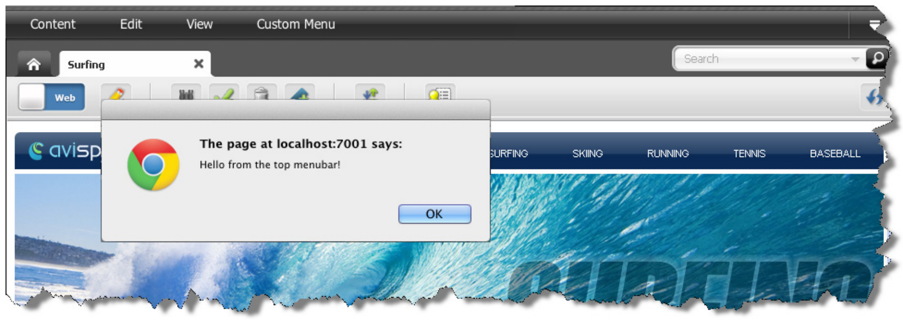
3. Finally, we define the child menu items:

```
config.menubar["Page/AVISection"].push({
    "id": "myCustomMenu",
    "label": "Custom Menu",
    "children": [{
      "label": "Hello World",
      "action": function () {
         alert("Hello from the top menubar!");
      }
    }]
});
```

The **Custom Menu** can now be seen whenever an "AVISection" Page section is viewed:



Selecting the **Hello World** menu item should run the custom onClick handler:



The page at localhost:7001 says:
Hello from the top menubar!

4. To run the exact same code, whether clicked from the menu bar or toolbar, we could write the following:

```
// define the helloWorld code once
config.myActions = {
    hello: function (args) {
      var doc = SitesApp.getActiveDocument(),
        asset = doc.get('asset'),
        view = SitesApp.getActiveView();

      view.info('Hello World!! The asset is a ' + asset.type + ' with id:
            + asset.id);
    }
};

// attach it to the helloWorld button
config.toolbarButtons['helloworld'] = {
    src: 'js/fw/images/ui/ui/toolbarButton/smartlist.png',
    onClick: config.myActions.hello
};

config.toolbars["web/Page/AVISection"] = {
    "edit": config.toolbars.web.edit, // reuse default for edit mode
    "view": [  "form-mode", "edit", "separator", "preview", "approve",
    "delete", "bookmark", "unbookmark", "separator",
    "checkincheckout","separator","helloworld", "refresh"]
}

// attach it to the menubar, under "Custom Menu">"Hello World"
config.menubar['Page/AVISection'] = config.menubar['default'].slice(0);

config.menubar["Page/AVISection"].push({
    "id": "myCustomMenu",
    "label": "Custom Menu",
    "children": [{
        "label": "Hello World",
        "action": config.myActions.hello
    }]
});
```

6 Customizing Asset Forms

This chapter discusses customizations that you can apply to asset forms in the WebCenter Sites Contributor interface. It also provides a step-by-step example for building a single-valued and multi-valued attribute editor for some of the supported data types.

This chapter contains the following topics:

- Section 6.1, "Overview of Asset Forms Customization"
- Section 6.2, "Modifying the Header of Asset Forms"
- Section 6.3, "Building an Attribute Editor"

6.1 Overview of Asset Forms Customization

You can perform two types of customizations on asset forms in the Contributor interface. One, you can modify the header of an asset form. The other is, you can either customize an existing attribute editor, as explained in the *Oracle WebCenter Sites Developer's Guide* (in the chapter "Designing Attribute Editors," section "Customizing Attribute Editors"), or you can build a custom attribute editor for the data types supported by WebCenter Sites, as described in Section 6.3, "Building an Attribute Editor."

Note: Unlike other components of the Contributor interface, asset forms are not in the Contributor framework. Therefore, requests for asset forms are not processed by the UI Controller.

6.2 Modifying the Header of Asset Forms

You can modify the header of any asset form by creating a custom assettype-specific element in the `OpenMarket/Xcelerate/AssetType/<AssetTypeName>/` directory. You can include additional stylesheets or JavaScript code instead of modifying the body of the HTML pages. The name of the element must be `Header`.

6.3 Building an Attribute Editor

You can customize the look and feel of some of the existing out-of-the-box attribute editors, as described in the *Oracle WebCenter Sites Developer's Guide* (see "Customizing Attribute Editors" in the chapter "Designing Attribute Editors"). You can also create a custom attribute editor for the data types supported in WebCenter Sites.

This section describes how to build a custom attribute editor that supports a single value of data type `text`, `string`, `integer`, or `money`. This section also provides pointers and sample code for implementing a multi-valued attribute editor for the same data types.

Note: If you want to create a custom attribute editor for the `blob` or `asset` data type, you can base your implementation on the `UPLOADER` attribute editor for the `blob` type and the `PICKASSET` attribute editor for the `asset` data type.

Steps for building an attribute editor are the following:

- Section 6.3.1, "Creating a Dojo Widget and its Template"
- Section 6.3.2, "Defining the Attribute Editor as a Presentation Object"
- Section 6.3.3, "Creating the Attribute Editor Element"
- Section 6.3.4, "Creating the Attribute Editor"
- Section 6.3.5, "Implementing a Multi-Valued Attribute Editor"

6.3.1 Creating a Dojo Widget and its Template

This section describes how to create a dojo widget to handle a single value of data type `text`, `string`, `integer`, or money.

This section contains the following topics:

- Section 6.3.1.1, "Create a Template for the Dojo Widget"
- Section 6.3.1.2, "Creating a Dojo Widget"

6.3.1.1 Create a Template for the Dojo Widget

To create an HTML template for the Dojo widget:

1. In your WebCenter Sites installation directory, navigate to the `<context_root>/js/` directory.
2. Create a new directory structure under the `js` directory as follows: `extensions/dijit/templates`.
3. In the `<context_root>/js/extensions/dijit/templates` directory, create an HTML template file and give it a meaningful name. For example: `MyWidget.html`
4. In the HTML template file, define the look and feel of the new dojo widget. The content of this HTML template would look similar to this:

```
<div>
     <div>
         <input type="text" dojoAttachPoint='inputNode' name='${name}'
size='60' class='valueInputNode'></input>
     </div>
</div>
```

   If you use the above code for the template, then the input node will take the input from the end user and the value of the input node will be maintained in the dojo widget which you will create in Section 6.3.1.2, "Creating a Dojo Widget."

5. Save your template file.
6. Continue to Section 6.3.1.2, "Creating a Dojo Widget."

6.3.1.2 Creating a Dojo Widget

To create a Dojo widget:

1. Navigate to the `js/extensions/dijit` directory of the WebCenter Sites installation.
2. Create a `dojo` widget, for example, `MyWidget.js` (see Example 6–1) by implementing the following mandatory functions:
   - `_setValueAttr` – This `setter` method sets the value of the attribute.
   - `_getValueAttr` – This `getter` method gets the attribute value.
   - `isValid` – This method runs validations to see if the given value is valid or not.
   - `focus` – This sets the focus on the attribute editor.
   - `onChange` – This method is called whenever the user updates the value of the attribute.
   - `onBlur` – This method updates the widget when the attribute value is entered by the user. An update will be triggered when the user selects another field.

Example 6–1 Sample Code for a Dojo Widget

```
dojo.provide('extensions.dijit.MyWidget');
dojo.require('dijit._Widget');
dojo.require('dijit._Templated');
dojo.declare('extensions.dijit.MyWidget', [dijit._Widget, dijit._Templated], {
    //string.
    //The value of the attribute.
    value: '',
    //int
    //The Attribute editor's MAXALLOWEDCHARS should be assigned to this variable.
    maxAllowedLength: 15,
    //string
    //   This variable is required only for single valued instance.
    //   The server should recieve information from input element with this name.
    name: '',
    //HTMLElement
    //   This stores the cached template of the widget's representation.
    templateString: dojo.cache('extensions.dijit', 'templates/MyWidget.html'),
    //string
    //   This class will be applied to the top div of widget.
    //   It will help in managing css well.
    baseClass: 'MyWidget',
    postCreate: function() {
        var self = this;
        // Do not allow typing characters more than allowed length.
           dojo.connect(this.inputNode, 'onkeypress', function(e) {
           if (this.value.length >= self.maxAllowedLength && e.keyCode !=
dojo.keys.BACKSPACE)
         e.preventDefault();
      });
},
// Start -  Mandatory functions
_setValueAttr: function(value) {
    //  summary:
    //     Set the value to 'value' attribute and input node
    if (value === undefined || !this._isValid(value)) return;
    this.value = value;
    this._setInputNode(value);
},
_getValueAttr: function() {
   // summary:
   //Get the latest value and return it.
   return this.value;
},
_isValid: function(newVal) {
   //summary:
   //Verify if the given value is as per the expectation or not.
   if (newVal.length > this.maxAllowedLength) {
   return false;
}
   return true;
},
   focus: function() {
   //summary:
   //Set the focus to the representation node i.e. input node here.
   if (typeof this.inputNode.focus === 'function')
   this.inputNode.focus();
},
onBlur: function() {
   //summary:
   //Custom selected browser event when the value should be updated
   //Any activity which leads to value change should update the widget value as
well.
   this.updateValue();
},
_onChange: function(newValue) {
   //summary:
   //Internal onChange method
   this.onChange(newValue);
},
onChange: function(newValue) {
   //summary:
   //A public hook for onChange.
},
   // End -  Mandatory functions
   // Extra functions used in Mandatory functions
_setInputNode: function(value) {
  //summary:
  //Sets the value to input node.
  this.inputNode.value = value;
},
updateValue: function() {
  //summary:
  //Validate the newly entered value and if it is successful then update widget's
value.
  var newVal = this.inputNode.value;
  if (!this._isValid(newVal)) return;
  if (this.value != newVal)
  this._onChange(newVal);
  this.set('value', newVal);
}
});
```

Note: For information about creating dojo widgets, see the Dojo documentation at `http://dojotoolkit.org/`.

3. In the `js/extensions/themes/`directory, create a CSS (for example, `MyWidget.css`) for this widget. You can use the following code in the CSS file, or write your own code:

   ```
   .fw .MyWidget .valueInputNode {
   color: blue;
   }
   ```

4. In the `js/extensions/themes/`directory, update the `UI.css` with an import statement for the dojo widget's CSS. For example, `@import url("MyWidget.css");`
5. Save your work.
6. Continue to Section 6.3.2, "Defining the Attribute Editor as a Presentation Object."

6.3.2 Defining the Attribute Editor as a Presentation Object

This section describes how to define input tags (presentation objects) for flex attributes. It also describes how to assign arguments that the input tags can pass from the attribute editor to the display elements.

To define the attribute editor:

1. In your WebCenter Sites installation, navigate to the `Sites\11gR1\Sites\11.1.1.6.1\presentationobject.dtd` file.
2. In the `presentationobject.dtd` file, do the following;
   a. Add a new tag (presentation object) to the list in the `<!ELEMENT PRESENTATIONOBJECT …>` statement. In this example, the new tag is named `MYATTREDITOR`.

      In the following line, `MYATTREDITOR` is your custom attribute editor whose name matches the name of the element you will create in Section 6.3.3, "Creating the Attribute Editor Element." All other tags are out-of-the-box attribute editors.

      ```
      <!ELEMENT PRESENTATIONOBJECT (TEXTFIELD | TEXTAREA | PULLDOWN |
      RADIOBUTTONS | CHECKBOXES | PICKFROMTREE | EWEBEDITPRO | REMEMBER |
      PICKASSET | FIELDCOPIER | DATEPICKER | IMAGEPICKER | REALOBJECT |
      CKEDITOR | DATEPICKER | IMAGEPICKER | REALOBJECT | CKEDITOR | FCKEDITOR |
      UPLOAD | MAGEEDITOR | RENDERFLASH | PICKORDERASSET | TYPEAHEAD | UPLOADER |
      MYATTREDITOR)>
      ```

   b. Add an `<!ELEMENT …>` section that defines the new tag (presentation object) and the arguments it takes. This new tag includes elements that supply the logic behind the format and behavior of the attribute when it is displayed on a form. Ensure that `MAXALLOWEDCHARS` is marked as a required attribute.

      ```
      <!ELEMENT MYATTREDITOR ANY>
      <!ATTLIST MYATTREDITOR MAXALLOWEDCHARS CDATA #REQUIRED>
      <!ATTLIST MYATTREDITOR MAXVALUES CDATA #IMPLIED>
      ```

   c. Save and close the `presentationobject.dtd` file.
3. Continue to Section 6.3.3, "Creating the Attribute Editor Element."

6.3.3 Creating the Attribute Editor Element

This section describes how to create an element that displays an "edit" view of an attribute (single-valued) when it appears in a “New” or “Edit” form. This element must be located in the `OpenMarket/Gator/AttributeTypes` directory in the `ElementCatalog` table. The element name must exactly match the name of the tag you defined in Section 6.3.2, "Defining the Attribute Editor as a Presentation Object," so that it can be invoked by the tag (in this example, `MYATTREDITOR`).

1. Navigate to the `OpenMarket/Gator/AttributeTypes` directory in your ElementCatalog.
2. Create an attribute element for your new editor (in this example, `MYATTREDITOR.jsp`. See Example 6–2.) Ensure that the name of this element matches the tag name you defined in the `presentationobject.dtd` file (Section 6.3.2, "Defining the Attribute Editor as a Presentation Object").
3. To prevent the default rendering of the attribute editor, set the `doDefaultDisplay` variable to `no`.
4. To display the attribute name, call the element `OpenMarket/Gator/FlexibleAssets/Common/DisplayAttributeName`. The code is:

   ```
   <ics:callelement
     element="OpenMarket/Gator/FlexibleAssets/Common/DisplayAttributeName"/>
   ```

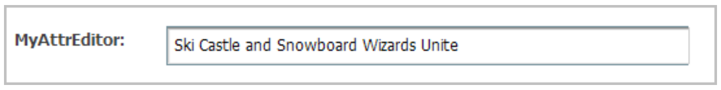
5. To render the widget, call the element `OpenMarket/Gator/AttributeTypes/CustomTextAttributeEditor` by using the following parameters:

   - `editorName` – Name of the widget created in Section 6.3.1.2, "Creating a Dojo Widget." In this example it is `extensions.dijit.MyWidget`.
   - `editorParams` – This argument passes the JSON string of parameters to the widget. In this example, it passes the `maxAllowedLength` value. For example, the value can look like this: `{ maxAllowedLength: "10" }`
   - `maximumValues` – Required only for a multi-valued widget. This is the maximum number of values allowed to be rendered in a multi-valued widget.

     For a single-valued widget, the complete code with the initialization parameters and formatting styles should look like the code in Chapter 6–2, "Sample Single-Valued Attribute Editor (MYATTREDITOR)." To implement a multi-valued widget, see Section 6.3.5, "Implementing a Multi-Valued Attribute Editor."

   If you use the code given in Example 6–2, then the single-valued attribute editor would look like the editor in Figure 6–1.

Figure 6–1 Single-Valued Attribute Editor



MyAttrEditor: Ski Castle and Snowboard Wizards Unite

Note: In Example 6–2, the following core logic is implemented to render the single-valued attribute using the new attribute editor:

```
<ics:if condition='<%= "no".equals(ics.GetVar("MultiValueEntry"))
%>'>
<ics:then>
     <div dojoType='<%= ics.GetVar("editorName") %>'
     name='<%= ics.GetVar("cs_SingleInputName") %>'
     value='<%= attributeValue %>'
     >
     </div>
</ics:then>
```

The name that is coded in the element must be `ics.GetVar("cs_SingleInputName")` to ensure that the input node in the dojo template will have the same name. The input node value will be sent to the server for saving the attribute.

Example 6–2 Sample Single-Valued Attribute Editor (MYATTREDITOR)

```
<%@ taglib prefix="cs" uri="futuretense_cs/ftcs1_0.tld" %>
<%@ taglib prefix="ics" uri="futuretense_cs/ics.tld" %>
<%@ taglib prefix="satellite" uri="futuretense_cs/satellite.tld" %>
<%//
// OpenMarket/Gator/AttributeTypes/MYATTREDITOR
//
// INPUT
//
// OUTPUT
//%>
<%@ page import="COM.FutureTense.Interfaces.FTValList" %>
<%@ page import="COM.FutureTense.Interfaces.ICS" %>
<%@ page import="COM.FutureTense.Interfaces.IList" %>
<%@ page import="COM.FutureTense.Interfaces.Utilities" %>
<%@ page import="COM.FutureTense.Util.ftErrors" %>
<%@ page import="COM.FutureTense.Util.ftMessage"%>
<cs:ftcs>
<ics:setvar name="doDefaultDisplay" value="no" />
<script>
     dojo.require('extensions.dijit.MyWidget');
</script>
<link href="<%=ics.GetVar("cs_imagedir")%>/../js/extensions/themes/MyWidget.css"
     rel="stylesheet" type="text/css"/>
<%
FTValList args = new FTValList();
args.setValString("NAME", ics.GetVar("PresInst"));
args.setValString("ATTRIBUTE", "MAXALLOWEDCHARS");
args.setValString("VARNAME", "MAXALLOWEDCHARS");
ics.runTag("presentation.getprimaryattributevalue", args);
args.setValString("NAME", ics.GetVar("PresInst"));
args.setValString("ATTRIBUTE", "MAXVALUES");
args.setValString("VARNAME", "MAXVALUES");
ics.runTag("presentation.getprimaryattributevalue", args);
String maximumValues = ics.GetVar("MAXVALUES");
maximumValues = null == maximumValues ? "-1" : maximumValues;
String editorParams = "{ maxAllowedLength: " + ics.GetVar("MAXALLOWEDCHARS") + "
}";
%>
```

```
<tr>
<ics:callelement
element="OpenMarket/Gator/FlexibleAssets/Common/DisplayAttributeName"/>
    <td></td>
    <td>
    <ics:callelement
element="OpenMarket/Gator/AttributeTypes/CustomTextAttributeEditor">
         <ics:argument name="editorName" value="extensions.dijit.MyWidget" />
         <ics:argument name="editorParams" value='<%= editorParams %>' />
         <ics:argument name="maximumValues" value="<%= maximumValues %>" />
</ics:callelement>
    </td>
 </tr>
</cs:ftcs>
```

6. Continue to Section 6.3.4, "Creating the Attribute Editor."

6.3.4 Creating the Attribute Editor

This section describes how to create an attribute editor asset to make it available to content contributors on their content management sites. This asset will support the input types you defined in Section 6.3.3, "Creating the Attribute Editor Element," for example, check boxes, radio options, and drop-down lists. The developer selects this editor when creating or modifying the attribute.

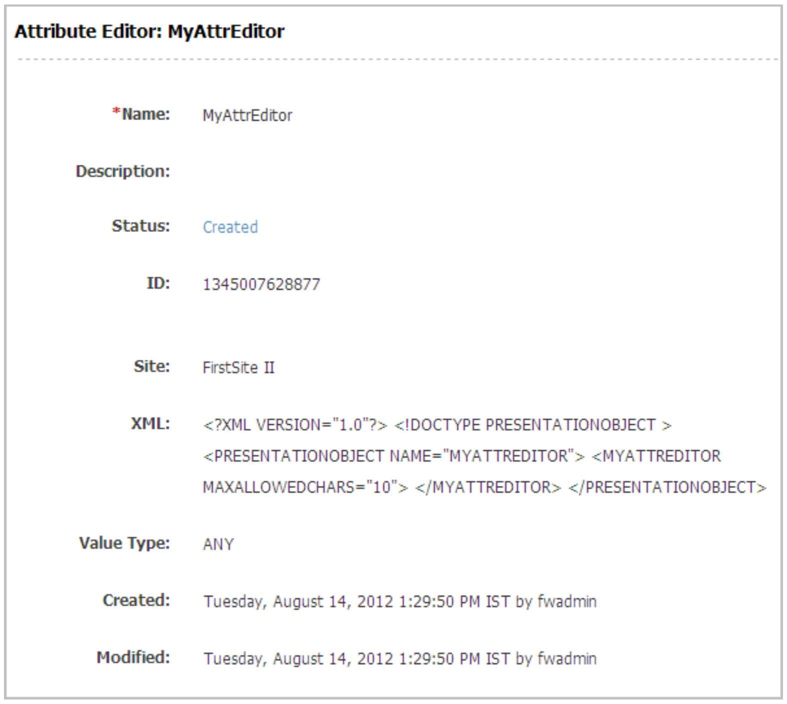
1. Log in to the Admin interface of your site.
2. On the **New** page, under the **Name** column, click **New Attribute Editor**.
3. In the **Name** field, enter a meaningful name for your editor. For example, `MyAttrEditor`.
4. In the **XML** box, enter the XML code for your attribute editor. Ensure that the name of the attribute editor in this code is exactly the same as the element name. For example:

```
<?XML VERSION="1.0"?>
<!DOCTYPE PRESENTATIONOBJECT >
<PRESENTATIONOBJECT NAME="MYATTREDITOR">
<MYATTREDITOR MAXALLOWEDCHARS="10"> </MYATTREDITOR> </PRESENTATIONOBJECT>
```

5. In the **Attribute Type** box, accept the appropriate value(s).
6. Click the **Save** icon.

   The attribute editor similar to the editor in Figure 6–2 is created for your site.

Figure 6–2 Sample Attribute Editor for a Site



Attribute Editor: MyAttrEditor

*Name:	MyAttrEditor
Description:	
Status:	Created
ID:	1345007628877
Site:	FirstSite II
XML:	<?XML VERSION="1.0"?> <!DOCTYPE PRESENTATIONOBJECT > <PRESENTATIONOBJECT NAME="MYATTREDITOR"> <MYATTREDITOR MAXALLOWEDCHARS="10"> </MYATTREDITOR> </PRESENTATIONOBJECT>
Value Type:	ANY
Created:	Tuesday, August 14, 2012 1:29:50 PM IST by fwadmin
Modified:	Tuesday, August 14, 2012 1:29:50 PM IST by fwadmin

7. Continue to Section 6.3.5, "Implementing a Multi-Valued Attribute Editor."

6.3.5 Implementing a Multi-Valued Attribute Editor

In Section 6.3.3, "Creating the Attribute Editor Element," Example 6–2 shows the implementation for a single-valued attribute editor. To implement a multi-valued attribute editor for `text`, `integer`, `string`, or `money` data types, you can write code similar to the code in Example 6–3.

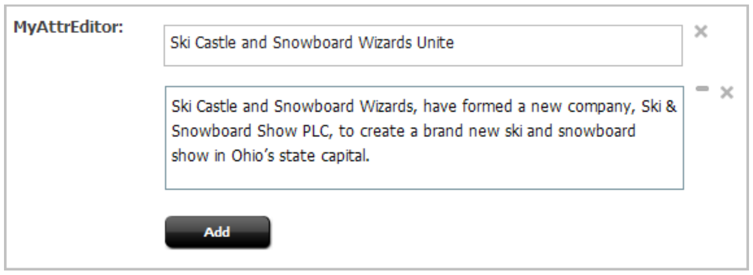
Example 6–3 Sample Multi-Valued Attribute Editor (CustomTextAttributeEditor)

```
<%@ taglib prefix="cs" uri="futuretense_cs/ftcs1_0.tld" %>
<%@ taglib prefix="ics" uri="futuretense_cs/ics.tld" %>
<%@ taglib prefix="satellite" uri="futuretense_cs/satellite.tld" %>
<%//
// OpenMarket/Gator/AttributeTypes/CustomTextAttributeEditor
//
// INPUT
//
// OUTPUT
//%>
<%@ page import="COM.FutureTense.Interfaces.FTValList" %>
<%@ page import="COM.FutureTense.Interfaces.ICS" %>
<%@ page import="COM.FutureTense.Interfaces.IList" %>
<%@ page import="COM.FutureTense.Interfaces.Utilities" %>
<%@ page import="COM.FutureTense.Util.ftErrors" %>
<%@ page import="COM.FutureTense.Util.ftMessage"%>
<cs:ftcs>
<%
IList attributeValueList = ics.GetList("AttrValueList", false);
boolean hasValues = null != attributeValueList && attributeValueList.hasData();
String attributeValue = hasValues ? attributeValueList.getValue("value") : "";
%>
```

```
<ics:if condition='<%= "no".equals(ics.GetVar("MultiValueEntry")) %>'>
<ics:then>
       <div dojoType='<%= ics.GetVar("editorName") %>'
            name='<%= ics.GetVar("cs_SingleInputName") %>'
            value='<%= attributeValue %>'
       >
       </div>
</ics:then>
<ics:else>
<ics:callelement
element="OpenMarket/Gator/AttributeTypes/RenderMultiValuedTextEditor">
       <ics:argument name="editorName" value='<%= ics.GetVar("editorName") %>' />
       <ics:argument name="editorParams" value='<%= ics.GetVar("editorParams") %>'
/>
       <ics:argument name="multiple" value="true" />
       <ics:argument name="maximumValues" value='<%= ics.GetVar("maximumValues")
%>' />
</ics:callelement>
</ics:else>
</ics:if>
</cs:ftcs>
```

If the code in Example 6–3 is used, then the multi-valued attribute editor will look similar to the editor in Figure 6–3.

Figure 6–3 Multi-Valued Attribute Editor



Ski Castle and Snowboard Wizards Unite

Ski Castle and Snowboard Wizards, have formed a new company, Ski & Snowboard Show PLC, to create a brand new ski and snowboard show in Ohio's state capital.

Note the following points about Example 6–3:

- You can instantiate a multi-valued widget, which uses a single-valued widget to render multi-valued representations.
- The `MultiValueEntry` variable with the `no` value indicates that the attribute editor renders a single value. Changing the variable value to `yes` will enable the attribute editor to render multiple values.
- You can implement a multi-valued widget that accepts values in the JSON object or in any other format.
- For a multi-valued attribute editor, the `RenderMultiValuedTextEditor` element creates hidden input nodes required for `Save` logic. The value of each node is sent to the server.
- The multi-valued widget is rendered by calling the `OpenMarket/Gator/AttributeTypes/RenderMultiValuedTextEditor` element using the following code in Example 6–3:

  ```
  <ics:else>
       <ics:callelement
  element="OpenMarket/Gator/AttributeTypes/RenderMultiValuedTextEditor">
            <ics:argument name="editorName" value='<%= ics.GetVar("editorName")
  ```

```
%>' />
          <ics:argument name="editorParams" value='<%=
ics.GetVar("editorParams") %>' />
          <ics:argument name="multiple" value="true" />
          <ics:argument name="maximumValues" value='<%=
ics.GetVar("maximumValues") %>' />
     </ics:callelement>
</ics:else>
</ics:if>
```

If you want to create a custom attribute editor for the `blob` or `asset` data type, you can base your implementation on the `UPLOADER` attribute editor for the `blob` type and the `PICKASSET` attribute editor for the `asset` data type.