FatWire | Content Server 7

Customizing Content Server's Dash Interface

Document Revision Date: Jan. 31, 2012

FatWire®
SOFTWARE

Customizing Content Server's Dash Interface
Document Revision Date: Jan. 31, 2012
Product Version: FatWire Content Server 7.6 Patch 2

FatWire Headquarters
FatWire Corporation
330 Old Country Road
Suite 303
Mineola, NY 11501

Table of

Contents

Appendices

About This Guide

This guide describes the process of customizing the FatWire Content Server Dash interface.

Who Should Use This Guide

This guide is written primarily for developers who are familiar with FatWire Content Server and have a working knowledge of HTML, JavaScript, and CSS. For more advanced changes, developers must have a working knowledge of Java, JSF, Content Server tags, the Content Server API, and Content Server Explorer (a utility for directly managing Content Server's database).

Related Documents

See the following documents in the FatWire documentation set:

- *FatWire Content Server Developer's Guide*
- *FatWire Content Server Property Files Reference*

Conventions

The following text conventions are used in this guide:

- **Boldface** type indicates graphical user interface elements that you select.
- *Italic* type indicates book titles, emphasis, or variables for which you supply particular values.
- `Monospace` type indicates file names, URLs, sample code, or text that appears on the screen.
- **`Monospace bold`** type indicates a command.

Third-Party Libraries

FatWire Content Server 7.6 patch 2 and its applications include third-party libraries. For additional information, see *FatWire Content Server 7.6 Patch 2: Third-Party Licenses*.

Chapter 1

Introduction

This chapter outlines the skill set that developers must have in order to customize the FatWire Content Server Dash interface as outlined in this guide.

This chapter contains the following sections:

- Customization Options
- Prerequisites

Customization Options

The plug 'n play design of the FatWire Content Server Dash interface enables developers to easily customize many of its components. Extension and integration points support options such as:

- Changing the general "look and feel" of the Dash interface:
    - Customizing the site selection page
    - Customizing the home page, specifically the following sections in the right navigation pane: How do I?, Learn about FatWire, and online help, itself
    - Customizing pagination settings for search results lists
- Regulating access to tabs on Create, Edit, and Inspect screens and to the individual attributes
- Customizing the rendering of attributes on Create, Edit, and Inspect screens
- Creating filters to refine searches and regulate access to FatWire Content Server's tree
- Validating assets
- Redefining users' privileges to functions such as inspect, edit, and delete

Prerequisites

For making simple look and feel changes to Content Server's interfaces, you need to have a working knowledge of HTML, JavaScript, and CSS. For more advanced changes, you must have a working knowledge of Java, JSF, Content Server tags, the Content Server API, and Content Server Explorer (a utility for directly managing Content Server's database).

Knowledge of the Dash interface is also required, the following features in particular:

Figure 1: Site selection page

Select	Name ▲	Description	Roles
○	BurlingtonFinancial	Burlington Financial	Approver, GeneralAdmin, DashUser, Marketer, Author, SiteAdmin, Editor, Expert, Analyst, Designer, AdvancedUser, WorkflowAdmin, Checker
○	FirstSiteII	FirstSite II	ContentAuthor, Approver, MarketingEditor, GeneralAdmin, DashUser, ArtworkAuthor, DocumentAuthor, ContentEditor, SiteAdmin, ArtworkEditor, ProductAuthor, DocumentEditor, Designer, ProductEditor, AdvancedUser, WorkflowAdmin, MarketingAuthor
○	GE Lighting	GE Lighting	Approver, GeneralAdmin, DashUser, Marketer, Author, SiteAdmin, Editor, Designer, Pricer, AdvancedUser, Checker, WorkflowAdmin
○	HelloAssetWorld	Hello Asset World	HelloDesigner, AdvancedUser, SiteAdmin, WorkflowAdmin, HelloEditor, HelloAuthor, DashUser, GeneralAdmin
○	Spark	FatWire Spark pCM	SparkAdmin, SparkDocumentUser, AdvancedUser, SiteAdmin, WorkflowAdmin, GeneralAdmin, SparkContentUser, DashUser

Select

Figure 2: Home page – the tree and right navigation pane:

Figure 3: Search screen:

Figure 4: The Edit screen, its Create/Inspect counterpart screens, and all the associated tabs (such as Content, Metadata, and Relations):
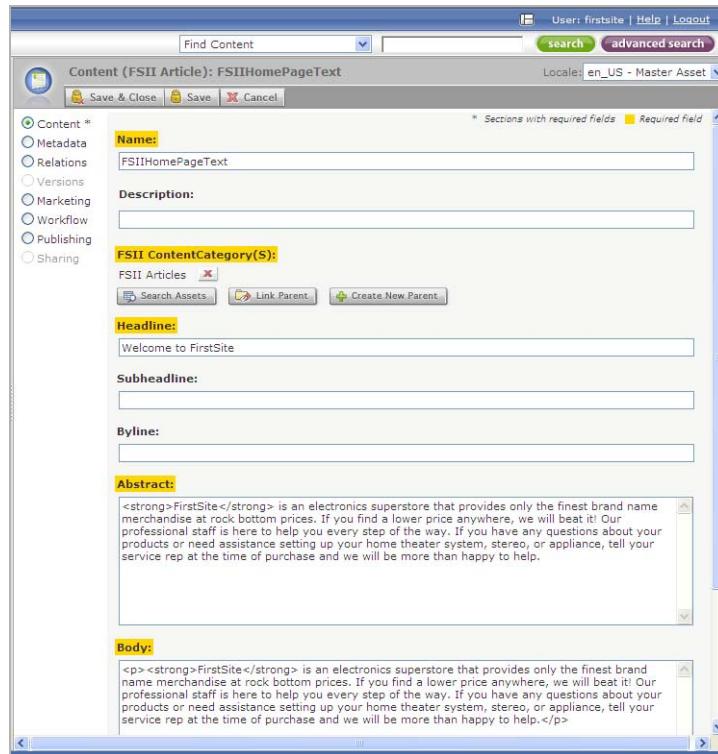
Chapter 2

Configuring Styles and Interface Features

This chapter outlines Dash interface styles and features that can be customized.

This chapter contains the following sections:

- 'Look and Feel' Changes
- Features Set by uiadmin.properties
- Features Set by ui.properties

'Look and Feel' Changes

Oracle ADF components, MyFaces, and various style sheets regulate the overall appearance of the Dash interface.

General Changes

Dash is built on Oracle ADF components. The style of these components is maintained in `cs.css`, located in the `webroot\skins\cs` folder. When a page is visited, a new `css` file is generated by JSF for the given operating system and browser. The path to the generated **css** file can be found by viewing the source of the rendered html page. For changes to be seen, the browser cache must be flushed.

The list of available JSF components can be found at the sites listed below:

- Oracle components: `www.oracle.com`

    At the time of this writing, the direct URL is:

    ```
    http://www.oracle.com/technology/products/jdev/htdocs/
    partners/addins/exchange/jsf/doc/tagdoc/core/imageIndex.html
    ```

- MyFaces: `www.irian.at`

    At the time of this writing, the direct URL is:

    ```
    http://www.irian.at/myfaces/home/jsf
    ```

The styles of ADF components can be changed by modifying `cs.css`. Details about the styles can be found on the page that explains skins. The list of available styles can be found at the following URL:

```
http://www.oracle.com/technology/products/jdev/htdocs/
  partners/addins/exchange/jsf/doc/skin-selectors.html
```

FCKEditor, Date Picker, Image Editor, Tables

Changes to special editors in the Dash interface can be made by modifying the style sheets associated with the editors. The style sheets can be found on the following sites:

- FCKEditor: `http://www.fckeditor.net/`
- Image Editor: `http://www.onlineimageeditor.info/`

The styles of JSF components can be changed by following the steps in "General Changes."

Features Set by uiadmin.properties

Changes to selected sections of the Dash interface can be made by modifying the `uiadmin.properties` file. Customizable sections of the Dash interface include the following:

- Login Page
- Site Selection Screen
- Search Results Screen

Login Page

You can customize the following components of the Dash login page:

- **Forgot your password?** link
- **Don't have an account?** link
- Logo

The components can be modified via the following properties:

- `forgotpassword` specifies the email address that will receive the user's request for a new password

- `noaccount` specifies the e-mail address that will receive requests for a CS user account

- `clientlogo` specifies the image that will be shown in this location

Site Selection Screen

The site selection screen is displayed to Content Server users who log in with permissions to multiple sites. The `sitestablerowcount` property is used to set the number of rows per page in the site selection table ().

Search Results Screen

To adjust the length and pagination of search results lists (), you would set the following properties in the `uiadmin.properties` file:

- `searchresultscount`: specifies the maximum number of search results that can be returned.

- `searchtablerowcount`: specifies the number of rows per page in the search results table.

- `popupsearchtablerowcount`: specifies the number of rows per page in the pop-up search results table.

- `searchtableattrcols`: specifies which columns will be displayed in the search results list (custom columns cannot be added).

Logo and Help Links

You can customize the following components of the Dash interface:

- The logo in the top left corner. The logo is an image file specified by the `clientlogodash` property (independently of the logo on the Dash login page).

- The **Help** link (top, right), specified by the `helplink` property.

- The text of each link in the **How do I ...** pane, specified by the `howdoilink.<nn>` property. (The help topics, themselves, are customizable.)

More information about `uiadmin.properties` is available in the *Property Files Reference*.
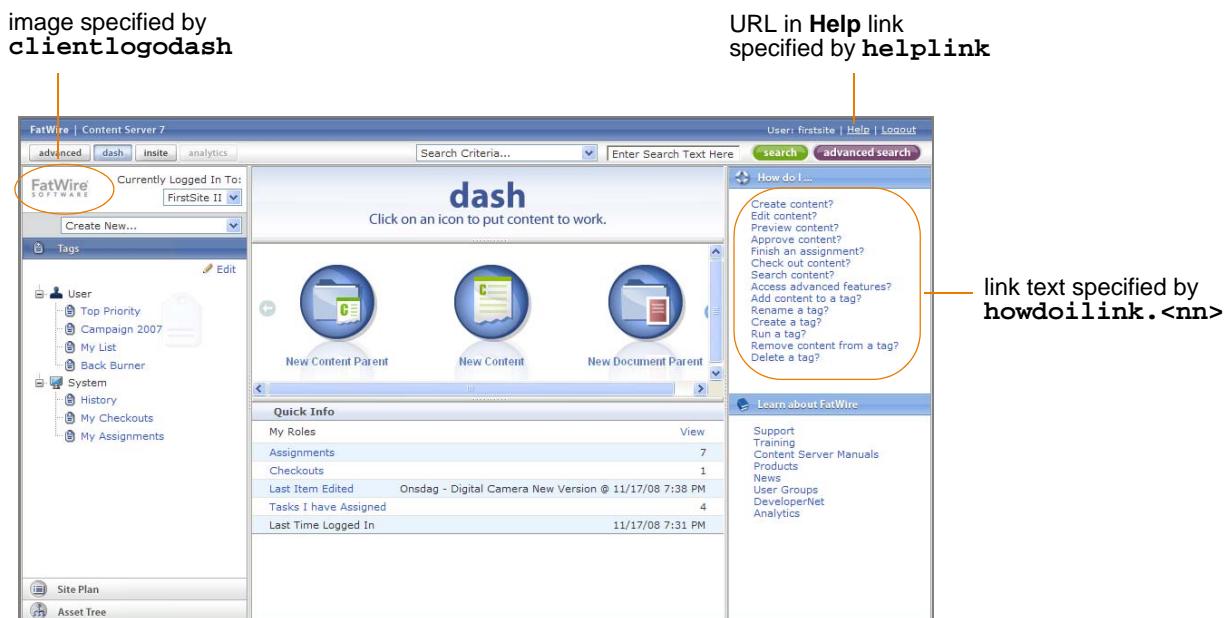
Features Set by ui.properties

The `ui.properties` file is used to configure the following portions of Content Server's Dash interface:

- Start menus in the quick access pane (start menus provide an easy way for content providers to create assets).

- Links to user-selected URLs, listed in the "Learn More about FatWire" pane.

For more information about `ui.properties`, see Figure 5, on page 17 and the *Property Files Reference*.

Figure 5: Customizing start menus and "learn more" links

`startmenu` properties are
used to customize start menus

`learnmorelink` properties
are used to modify and create
links to various URLs
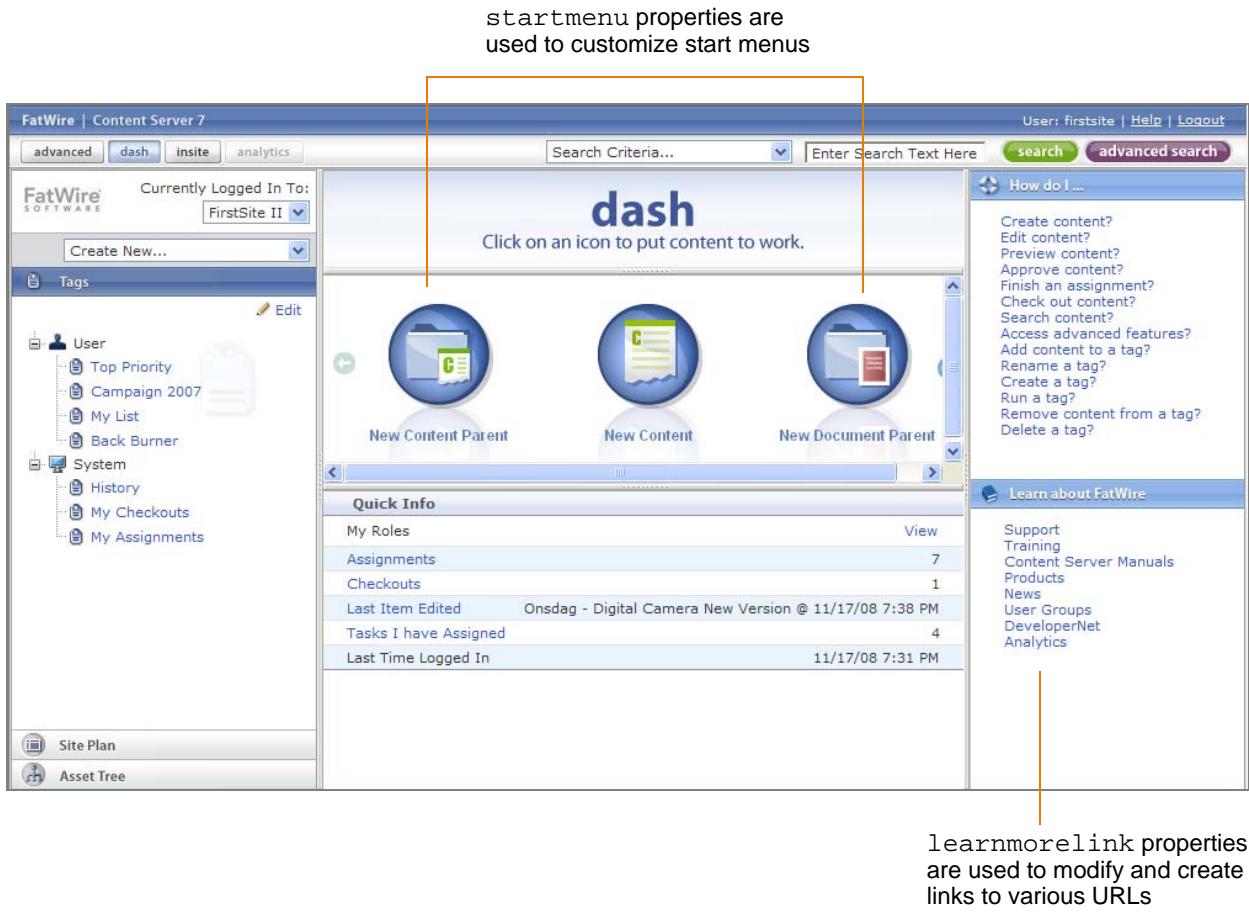
Chapter 3

Configuring Tabs and Attributes

Except for system-defined tabs, any tab on a Create, Edit, or Inspect screen can be displayed to or hidden from selected users.

This chapter contains the following sections:

- Configuration Options
- Configuration Method

Configuration Options

Tabs on Create, Edit, and Inspect screens can be selectively displayed to certain users and for certain asset types, depending on how you configure the tabs in the `AssetEditPane` table.

- Access to tabs can be configured on the basis of the user's role, asset type, and asset definition:

    - Tabs can be displayed to selected users, depending on the user's role.

    - Tabs can be displayed for selected asset types.

    - Tabs can be displayed for selected asset subtypes of a given asset type.

- Attributes can be selectively displayed to users of the tabs.

> **Note**
>
> This guide describes additional options for configuring attributes. For more information, see Chapter 4, "Customizing the Rendering of Attributes."

Configuration Method

Configuring a tab and controlling access to its attributes requires you to add an entry for the tab to the `AssetEditPane` table (using Content Server Explorer). For example, to allow only Dash users access to the **Content** tab in the example below, you would specify `DashUser` in the `role` field for this tab in the `AssetEditPane` table.
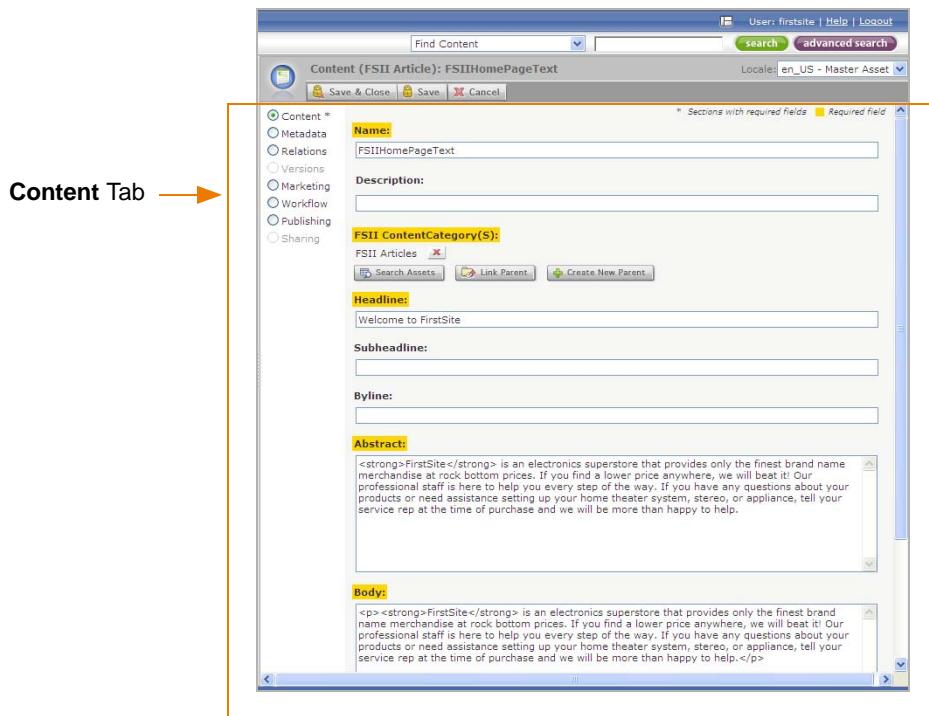
Columns in the `AssetEditPane` table are defined in Table 1.

Table 1: Columns in the `AssetEditPane` table

Column Name	Description
`id`	CS-generated ID for the tab that is being configured.
	Note: System-defined tabs cannot be customized. The tabs are as follows:
	• Workflow
	• Versioning
	• Publishing
	• Sharing
`role`	User role for which this tab will be displayed.
`assettype`	Type of asset for which this tab will be displayed.
`assetdefid`	For flex assets, `assetdefid` is the ID of the asset's flex definition. For basic assets, `assetdefid` is `-1`.
`tabname`	Internal name of the tab (used by Content Server's database).
`tabtitle`	Description of the tab (displayed in the interface). The description holds the key of the `SystemLocaleString` so that it can be localized.
`attributes`	Comma-separated list of attribute names. Including an attribute name displays the attribute in the tab.
	When entering attribute names, note the following:
	• If an attribute is user-defined, it must be prefixed with: `attribute_`
	• If you need to display attribute components (parent associations, for example), include the following system-defined attributes (exactly as shown below) in the list:
	`__parent_` To display the parent attribute in a tab.
	`__associations_` To display the parent associations in a tab.
	`__referencedby_` To display the "referenced by" attribute in a tab.
`tabordinal`	Sort order of the tabs.

Chapter 4

Customizing the Rendering of Attributes

This chapter describes attribute renderers and provides steps for creating them.

This chapter contains the following sections:

- Overview
- Methodology for Creating Attribute Renderers
- How AssetRenderer Works
- Working with Attribute Renderers
- How JSF Binding is Used in Attribute Renderers

Overview

Controlling the appearance of an attribute in the Dash interface requires you to create and configure an attribute renderer. Each renderer is then available to AssetRenderer, which paints Create, Edit, and Inspect screens.

When painting a screen, AssetRenderer first gets the attributes, then invokes their renderers, and finally draws the first tab ("Content"). AssetRenderer then invokes attribute renderers to paint the rest of the attributes on the remaining tabs. Which renderers are invoked depends on the attribute renderers' configurations, specified in the `FW_AttributeRendererConfig` table.

Attribute Renderers

An attribute renderer outputs a label for the attribute and, if required, one or more controls for displaying and managing the attribute's data. For example, in Figure 6, **A** and **B** represent the outputs of two attribute renderers.

In **A**, a renderer was created to paint the `Name` attribute on the "Content" tab (the same renderer painted the `Description` attribute). In **B**, a different renderer was created to paint the `FSIIManufacturer(S)` attribute, and the next attribute, `FSIISubcategory`.

Figure 6: Attributes rendered in an asset's "Edit" screen
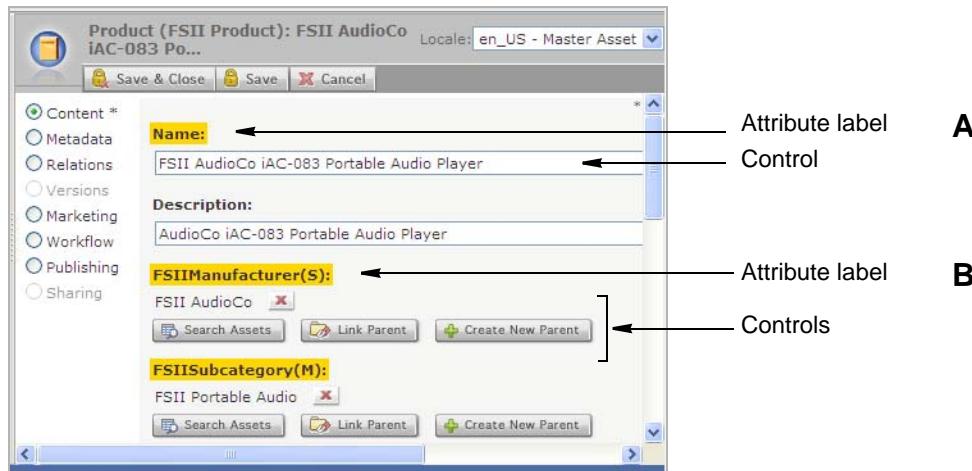
Terms and Definitions

The output of an attribute renderer can be an entire attribute, or any combination of the attribute's components: the attribute label, control, and/or inherited controls. The components are defined as follows:

Attribute label. The attribute's name or description.

Control. The component that holds the attribute's value (or values, for multi-valued attributes). A control can be a data entry field, a check box, a drop-down menu, and so on. The control determines the format in which the attribute's value is displayed (for example, font type, font size, weight, color, and alignment).

In Create/Edit screens, an attribute's control is typically used to manage data. In Inspect screens, it is used to display the attribute's value.

Inherited control.   The control that is created when you extend one of Content Server's default attribute renderers.

Types of Attribute Renderers

Dash supports two types of attribute renderers, Type I for Create/Edit screens, and Type II for Inspect screens. You define the type by configuring the attribute renderer in Content Server's database table FW_AttributeRendererConfig.

Type I

Type I attribute renderers are used to customize the appearance of attributes on Create and Edit screens. A Type I renderer has the following properties:

- A Type I renderer can output the attribute's label, the attribute's value (as editable information), the control (used to manage the attribute's value), and inherited controls. The output is determined by the developer.

- A Type I renderer's configuration is based on each attribute's attribute editor. The attribute editor determines which type of control(s) will be rendered: checkbox, text field, radio button, and so on. For the list of default attribute editors supported in Dash, see Table 3, on page 28.

Type II

Type II attribute renderers are used to customize the appearance of attributes on Inspect screens. For example, in the "Edit" screen in Figure 6, the attributes are painted by Type I renderers. A Type II renderer could be written to reformat the value of the "Name" attribute in the Inspect screen (for example, display the value in a different font type, size, color, and so on). A Type II renderer has the following properties:

- A Type II renderer can output the attribute's label and value (both non-editable), and the control (used to display the attribute's value). The output is determined by the developer. Note that a Type II renderer does not output inherited controls.

- A Type II renderer's configuration is based on the data type of the attribute's value(s): ASSET, BLOB, and FLOAT, to name a few. For the list of data types, see Table 3, on page 28.

Granularity

To provide for a range of granularity, Content Server enables you to configure generic attribute renderers or tailor them for certain attributes and asset types. For example configurations, see the appendices at the end of this guide.

Customization Options

Dash supports total and selective customization of attributes.

- Total customization gives you the following options:

    - Customize *all* components of an attribute for Create/Edit screens (using a Type I renderer).

    - Customize *all* components of an attribute for Inspect screens (using a Type II renderer).

    You can create an attribute renderer in the following ways:

- Write a CSElement-based attribute renderer; that is, create a CSElement asset with JSP logic for painting the attribute.

- Create a JSF-based attribute renderer; that is, implement the `AttributeDataRenderer.java` interface and overwrite the `render()` method.

- Selective customization gives you the option to customize any combination of the following components: the attribute's label, control, and/or inherited control (for Type I renderers). Selective customization requires you to write a JSF-based renderer by extending the relevant Java class and overwriting the applicable method.

Table 2 summarizes the customization options described above.

Table 2: Attribute Customization Options

Customization Option	Attribute Renderer		Customizable Component
Total	CSElement-based	Type I	All attribute components.
		Type II	All attribute components.
	JSF-based	Type I	All attribute components.
		Type II	All attribute components, except for inherited controls.
Selective	JSF-based	Type I	Attribute's label, control, inherited controls, any combination of the above.
		Type II	Attribute's label, control, any combination of the above.

For information about creating and configuring attribute renderers, see the following sections:

- For methodology, see "Methodology for Creating Attribute Renderers," on page 27.

- For procedures, see "Working with Attribute Renderers," on page 33.

System-Defined Tabs

System-defined tabs cannot be customized. The tabs are:

- Workflow
- Versioning
- Publishing
- Sharing

Methodology for Creating Attribute Renderers

Create, Edit, and Inspect screens are based on an HTML table that takes attributes, one per row. An attribute renderer is expected to return an HTML table row (`<tr>`).

How you go about creating an attribute renderer depends on whether you need a CSElement-based attribute renderer, or a JSF-based attribute renderer, and whether the renderer is of Type I or Type II.

Creating CSElement- and JSF-Based Attribute Renderers

To create a CSElement-based attribute renderer, you must create a CSElement asset and code its element logic in JSP. CSElement-based attribute renderers use a special, out-of-the-box attribute renderer named `CSElementAttributeRenderer`, which invokes the CSElement and creates a JSF component on the fly.

To create a JSF-based attribute renderer, you extend a Java class, or implement a Java interface, and then implement the applicable method.

When you create an attribute renderer, you must also configure the renderer in the `FW_AttributeRendererConfig` table of Content Server's database. The configuration procedure, while the same for all renderers, differs only in the information that you provide. For example:

- For CSElement-based attribute renderers, you must explicitly specify the special renderer (`CSElementAttributeRenderer`).

- Depending on the type of renderer you wish to create, you specify one of the following parameters:

  - All Type I configurations take *AttributeEditor* as a required parameter.

  - All Type II configurations take *DataType* as a required parameter. The *AttributeEditor* parameter is conditional, depending on your requirements.

  - For all renderers, *Attribute* and *AssetType* are conditional parameters, depending on the level of specificity you require.

Configuration syntax, terms, and parameters are defined in the rest of this section. Instructions for creating and configuring attribute renderers begin on page 33, "Working with Attribute Renderers."

Configuration Syntax for Attribute Renderers

When an attribute renderer is created, it must also be manually configured in Content Server's database, in the `FW_AttributeRendererConfig` table (one attribute renderer per row). An attribute renderer's configuration is represented by the following syntax:

Configuration Syntax

```
attributerendererkey:        (column name:
  value                       a value is required for all attribute renderers)
attributerendererclass:      (column name:
  value                       a value is required for all attribute renderers)
cselement:                   (column name:
  value                       a value is required for only CSElement-based attribute renderers)
```

Legal values for each column are defined in Table 3, on page 28. Configuration examples are available on page 29 and in the appendices at the end of this guide.

Table 3: `FW_AttributeRendererConfig` Table (Column Definitions and Values)

Column in Table: `FW_AttributeRendererConfig`	Value[a]	Data Type

`attributerendererkey`
 	
Key for identifying the attribute renderer and determining its type to be I or II.
	Attribute.AssetType.AttributeEditor.DataType.`attribute_renderer`
	
	Attribute:
	Name of the attribute to be rendered.
	
	AssetType:
	Asset type to which the attribute applies.
	
	AttributeEditor:[b]
	Attribute editor for the attribute renderer. Required for Type I attribute renderers. Conditional for Type II renderers (depending on your requirements). Supported attribute editors are as follows:
	
	`CHECKBOXES    DATEPICKER     EWEBEDITPRO`
	`FCKEDITOR     IMAGEEDITOR    IMAGEPICKER`
	`PICKASSET     PICKFROMTREE   PULLDOWN`
	`RADIOBUTTONS  RATING         REALOBJECT`
	`RENDERFLASH   TEXTAREA       TEXTFIELD`
	`TIMESTAMP     UPLOAD`
	
	DataType (for Type II renderers):
	Data type for which the attribute renderer is written. Supported data types are as follows:
	
	`ASSET         BLOB           DATE`
	`FLOAT         LARGE_TEXT     LONG`
	`MONEY         STRING         URL`
	
	`attribute_renderer`:
	Constant string that must not be changed.
		Varchar(255)

`attributerendererclass`
 	
(For CSElement-based and JSF-based renderers)
	`com.fatwire.cs.ui.view.renderer.attribute.`
	`CSElementAttributeRenderer`
	
	For CSElement-based attribute renderers, this parameter specifies the Java class that is responsible for invoking the CSElement that renders the attribute.
	
	-or-
	
	`com.fatwire.`*CustomAttributeRenderer*
	
	For JSF-based attribute renderers, this parameter specifies the Java class that is responsible for rendering the attribute.
		Varchar(255)

`cselement` (Required for CSElement-based attribute renderers)	`fatwire/dash/attributerenderers/`*AttributeRenderer* Applies only to CSElement-based attribute renderers. Specifies the rootelement logic (path to the CSElement asset that renders the attribute).	Varchar(255)

a. Sample configurations are available in the appendices at the end of this guide.

b. For information about attribute editors, see the following sources:
   - Attribute editors for flex assets: Chapter 17, "Designing Attribute Editors," in the *Content Server Developer's Guide*.
   - Attribute editors for basic assets: "Asset Descriptor Files," in the *Content Server Developer's Guide*. (Note that `AttributeEditor` is specified in the asset descriptor file. It is the value of the `TYPE` parameter in the `INPUTFORM` tag in the attribute's definition.)

CSElement-Based and JSF-Based Renderer Configurations

When an attribute renderer is created, it must also be manually configured in a row of the database table named `FW_AttributeRendererConfig` (located under the **Tables** node). Configuration syntax for CSElement-based and JSF-based attribute renderers is shown in the examples below to illustrate the main difference between the configurations.

Example 1. CSElement-based Attribute Renderer: The `Description` attribute for the `Content_C` asset type is rendered by a CSElement-based Type I attribute renderer named `htmltextdataRenderer`. The attribute renderer's configuration in the `FW_AttributeRendererConfig` table is the following:

```
attributerendererkey:
  description.Content_C.TEXTFIELD.attribute_renderer
attributerendererclass:
  com.fatwire.cs.ui.view.renderer.attribute.CSElementAttributeRenderer
cselement:
  fatwire/dash/attributerenderers/htmltextdataRenderer
```

`CSElementAttributerenderer` executes the JSP `htmltextdataRenderer.jsp`, which then renders the `Description` attribute. The skeleton of the CSElement is shown below. Notice that the CSElement will be invoked twice: the first time for rendering the attribute when the screen is drawn (paint mode), and once again for updating the attribute's value when the screen is saved (update mode):

```
String mode = (String ) request.getAttribute("mode");
if("paint".equals(mode))
{
    // This is where the code for rendering the attribute will be
  written
 } else if("update".equals(mode))
{
This mode is similar to ContentPost in AdvancedUI. This is where we
  read the values from the request and update the AttributeData.
}
```

(Sample code for this attribute renderer is shown in Appendix A, "CSElement-Based Type I Attribute Renderer Configurations.")

Example 2. JSF-Based Attribute Renderer: In this example, the `Description` attribute from example 1 is now rendered by a JSF-based Type I attribute renderer named `htmltextdataRenderer.java`. The attribute renderer's configuration in the `FW_AttributeRendererConfig` table is the following:

```
attributerendererkey:
  description.Content_C.TEXTFIELD.attribute_renderer
attributerendererclass:
  com.fatwire.cs.ui.view.renderer.attribute.htmltextdataRenderer
```

Information about extending basic renderers can be found in "CSElement-Based and JSF-Based Renderer Configurations," on page 29.

Working with JavaScript in CSElement-Based Attribute Renderers

How does Partial Page Rendering (PPR) work in Oracle ADF?

The partial page rendering model for Oracle ADF uses an iframe for submitting the form. The response is rendered into the iframe and ADF refreshes the parent window by replacing necessary html elements. This means the JavaScript variables or functions defined in the attribute renderer are defined in the scope of the iframe.

To resolve this problem, attribute renderer JavaScript must be specially treated, as follows:

- JavaScript functions and variables must be declared in the parent window scope:

    Example 1
    ```
    window.parent.myfunction = function() {//mycode};
    ```
    Example 2
    ```
    window.parent.myvar = "Hello World";
    ```
    Example 3
    ```
    window.parent.document.getElementById("myelement").innerHTML
        = "replace me";
    ```

- JavaScript functions must be invoked from the parent window scope:

    ```
    GenericUtil.addjavascriptfunctions ()
    ```

Adding JavaScript functions to the `SessionBean` object using the `addJavascriptFunctions()` method ensures that the function is invoked from the parent window.

How AssetRenderer Works

AssetRenderer is responsible for assembling Create, Edit, and Inspect screens. When a screen is requested by the user, AssetRenderer invokes attribute renderers to paint attributes on the proper tabs and in the proper order. For example, when painting an "Edit" screen, AssetRenderer takes the following steps, starting with the "Edit" pane:

1. The "Edit" pane is composed of many tabs (Figure 7). Referring to the `AssetEditPane` table, AssetRenderer determines which tab to draw first (basing its determination on the asset type, asset definition, and user role).

Figure 7: "Edit" Screen and its Components
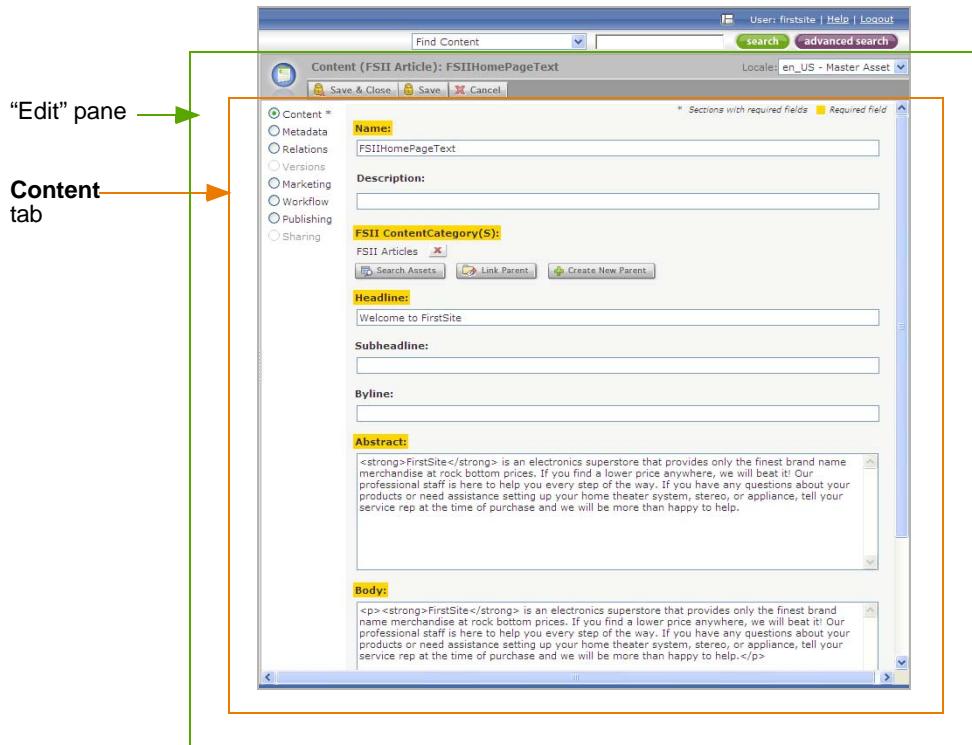
2. AssetRenderer paints the tab and continues as follows to complete the tab:

   a. AssetRenderer gets the list of attributes for the tab, determines the first attribute, determines the attribute's renderer, and paints the first attribute.

   b. AssetRenderer iterates through the list of attributes, and paints them one by one on the same tab.

3. As in step 2, AssetRenderer iterates through and paints the remaining tabs and attributes (painting only the tabs to which the user has permissions, based on role).

AssetRenderer's algorithm for determining Type I attribute renderers is shown in the flow chart of Figure 8. The same algorithm is used for determining Type II renderers (which take the attribute's data type as an additional term).

Figure 8: AssetRenderer's algorithm for determining an attribute's Type I renderer. *For Type II renderers, the same algorithm also tests for the attribute's data type.*
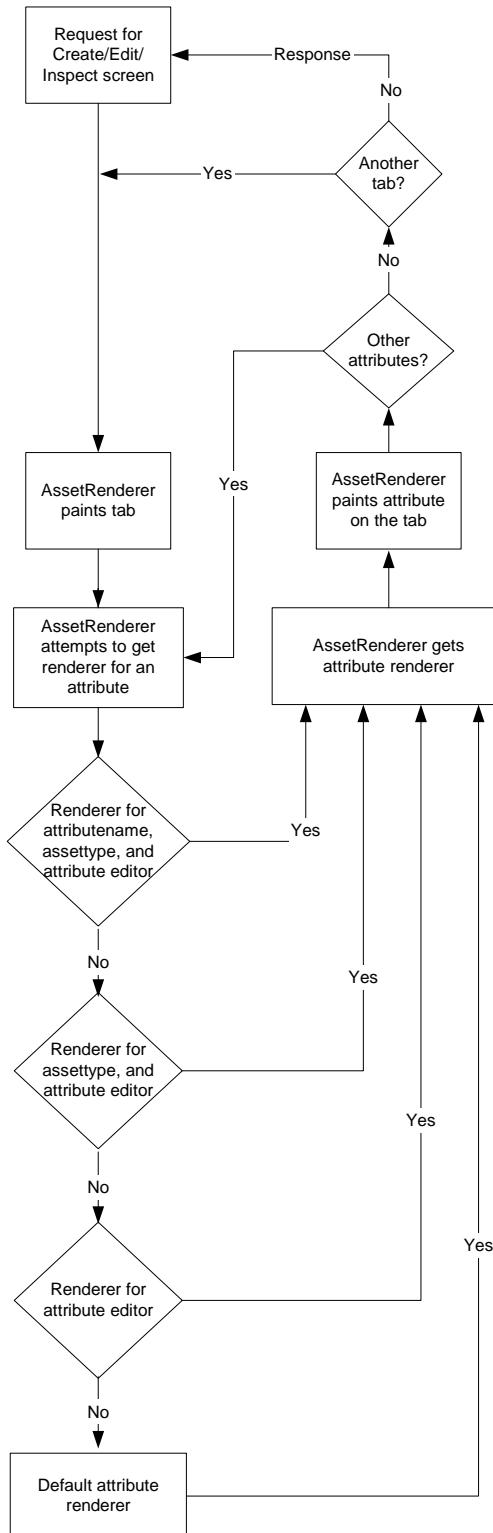
Working with Attribute Renderers

- Total Customization of Attributes
- Selective Customization of Attributes
- Customizing an Attribute Renderer

Total Customization of Attributes

To customize the appearance of an entire attribute, you can either create a CSElement-based attribute renderer or implement the `AttributeDataRenderer.java` interface.

> **Note**
>
> The steps in this section can be used to create Type I and Type II renderers.

To customize attributes via CSElement-based attribute renderers

1. Create a CSElement asset and code its JSP logic to render the attribute.

    Instructions for creating CSElement assets can be found in the *Content Server Developer's Guide*.

2. Configure the attribute renderer as a Type I or Type II renderer, as follows:

    Log in to Content Server Explorer and enter the attribute renderer's configuration information into a row of the `FW_AttributeRendererConfig` table (located under the **Tables** node). Use Table 3, on page 28 to determine which information to enter.

    If you need a generic renderer (which is not specific to a particular attribute, asset type, or attribute editor) exclude any or all of the following parameters:

    -   Type I configurations: *AttributeName*, *AssetType*
    -   Type II configurations: *AttributeName*, *AssetType*, *AttributeEditor*

    > **Reminder**
    >
    > For CSElement-based renderers, you must specify a value in the `cselement` column, as shown in Table 3, on page 28.

    For configuration examples, see Appendix A, "CSElement-Based Type I Attribute Renderer Configurations" and Appendix B, "CSElement-Based Type II Attribute Renderer Configurations."

To customize attributes via the `AttributeDataRenderer.java` **interface**

1. Implement the interface `AttributeDataRenderer.java` and overwrite the `render ()` method.

2. Configure the attribute renderer as a Type I or Type II renderer, as follows:

    Log in to Content Server Explorer and enter the attribute renderer's configuration information into a row of the `FW_AttributeRendererConfig` table (located under the **Tables** node). Use Table 3, on page 28 to determine which information to enter.

If you need a generic renderer (which is not specific to a particular attribute, asset type, or attribute editor) exclude any or all of the following parameters:

- Type I configurations: `AttributeName`, `AssetType`

- Type II configurations: `AttributeName`, `AssetType`, `AttributeEditor`

For configuration examples, see the appendices at the end of this guide.

Selective Customization of Attributes

Procedures in this section show you how to customize any combination of the following components: the attribute's label and its control (and/or inherited controls, if the renderer is a Type I attribute renderer).

Type I Renderer (Edit Screen)

Creating a Type I renderer to customize the appearance of an attribute's component(s) is the most common implementation.

To customize an attribute's component in the Edit screen

1. Create a JSF-based renderer by extending `BasicDataTemplate.java`, overwriting the `createComponents()` method, and returning a JSF component.

> **Note**
>
> **Customizing inherited controls.**
> Extending the `BasicDataTemplate.java` class allows the attribute renderer to inherit several features from its parent class. This includes the ability to display inherited controls, for example buttons such as **Link Asset**, **Show Asset**, **Add New**, **Up**, **Down**, **Delete** and its implementations. If these extra features must be customized, the custom attribute renderer can overwrite the `createComponents()` method, instead of the `renderData()` method, and return a JSF component.

2. Configure the attribute renderer as a Type I renderer:

Log in to Content Server Explorer and enter the attribute renderer's configuration information into a row of the `FW_AttributeRendererConfig` table (located under the **Tables** node). Use Table 3, on page 28 to determine which information to enter.

If you need a generic renderer (which is not specific to a particular attribute or asset type), exclude any or all of the following parameters: `AttributeName`, `AssetType`

For configuration examples, see Appendix C, "JSF-Based Type I Attribute Renderer Configurations."

Type II Renderer (Inspect Screen)

A Type II renderer is used to customize the appearance of an attribute's component(s) on "Inspect" screens.

To customize an attribute's component in the Inspect screen

1. Create a JSF-based renderer by extending `BasicDataRenderer.java` and overwriting the `renderData ()` method.

   2. Configure the attribute renderer as a Type II renderer:

   Log in to Content Server Explorer and enter the attribute renderer's configuration
   information into a row of the `FW_AttributeRendererConfig` table (located under
   the **Tables** node). Use Table 3, on page 28 to determine which information to enter.

   If you need a generic renderer, which is not specific to a particular attribute or asset
   type, exclude any or all of the following parameters: *AttributeName*, *AssetType*,
   AttributeEditor.

   For configuration examples, see Appendix D, "JSF-Based Type II Attribute Renderer
   Configurations."

Customizing an Attribute Renderer

All existing attribute renderers can be customized, except for those associated with
Content Server's sample sites. To customize an attribute renderer, you would reconfigure
its information in the `FW_AttributeRendererConfig` table (located under the **Tables**
node). Use Table 3, on page 28 to determine which information to enter or change. For
configuration examples, see the appendices at the end of this guide.

How JSF Binding is Used in Attribute Renderers

Value Binding and Method Binding

The two different types of binding expressions in JSF are 'value binding expressions' and
'method binding expressions.' Value binding expressions can be used inside JSF
components to:

- Automatically instantiate a JavaBean and place it in the request or session scope.
- Override the JavaBean's default values through its accessor methods.
- Traverse an object tree using an easy-to-learn DOM-style notation.
- Quickly retrieve `Map`, `List`, and array contents from a JavaBean.
- Synchronize form contents with value objects across a number of requests.

The syntax of binding expressions is based on the JavaServer Pages (JSP) 2.0 Expression
Language, which itself is based on the object accessing syntax of JavaScript. Page authors
can use value binding expressions in combination with the standard JSF user interface
component library. In doing so, page authors can easily create pre-populated forms in JSP
pages, and synchronize the contents of those forms with JavaBean values in the
conversational state of a servlet application.

How is It Used?

Value binding is used in an attribute renderer to bind the user interface (UI) components to
the attributes. The binding pattern is available to the attribute renderer in the form of
`AttributeDataBinding`, which contains the binding pattern for the attribute's data.
The pattern is similar to the string shown below:

```
#{session_bean.currentAsset.data.attributeData[index].dataAsList[@]}
```

where

   - `session_bean` is a backing bean available in the session scope

- `currentAsset` is an asset
  (`com.fatwire.ui.model.bo.asset.Asset.java`) object stored in session

- `data` is the attribute data encapsulated in the asset object

- `attributedata` is the list of attribute data

- `index` is the index of the attribute data

- `dataAsList` is the list of data for the attribute

- `@` is the placeholder string passed to the attribute renderer. The attribute renderer
  populates this string with a value of `0` for single-valued attributes, or an integer
  ranging from `0` to `n` for multivalued attributes (where `n` is the number of values).

> **Note**
>
> The `AttributeDataBinding` class also has the binding pattern for method
> bindings. These can be used for the following buttons: add new, up, and delete

PreSave Process and Auxiliary Map

The binding pattern from the `AttributeDataBinding` is used by the attribute renderers
to bind the UI components to the asset object.

For most of the JSF components the attribute data's data can be bound directly. But all JSF
components cannot be bound to the attribute data. For these attributes, the attribute data
must be converted to the JSP bindable data. Attribute renderer for the attribute converts
the attribute data This converted data is stored in an auxiliary data map
(`java,util.HashMap`) and the map is bound to the JSF components.When the form is
submitted, the data in the auxiliary data map is modified, because it is bound to JSF
components.

Presave process is the process that converts the attribute renderer data(`Auxiliary
Map`) to attribute data.This is used to convert the data displayed in the JSF components to
attribute data.

Attribute renderer registers the presave process in the asset object if the data is converted
and stored in auxiliary map.

When the **Save** button is clicked, the form is submitted and the asset service is invoked for
saving the asset. Asset service checks for the auxiliary map for each attribute.And if the
attribute has auxiliary map, it invokes the presave process. The presave process converts
the data from the auxiliary map to attribute data.

Before we save the asset, data in the auxiliary map should be converted to attribute
data.The process of converting the data is done using
`PreSaveProcess.Attributerenderer`, which uses these special JSF components to
handle converting data to attribute data to auxiliary data map and register the presave
process for converting back.

Example:

Let us see how Data Attribute Renderer uses this presave process and auxiliary map in
Dash.

Since there is no out-of-the-box JSF component for displaying date and time,
`CoreSelectInputDate` can be used for selecting the date. But there is none for

displaying time. So, we use the `CoreSelectOneChoice` for time (for choosing hour, minute, seconds, and periods am, pm).

There are five JSF components to display one attribute data. The attribute data cannot be used directly. So data is split and stored in `Auxiliary Map`. If the Attribute data has a single value which is "`2004-10-12 12:30:59 AM`" it will be converted to `Auxiliary Map` and it will look as follows:

```
{{Date, 2004-10-2},{Hour,12},{Min,30},{Sec,59},{Am_Pm,AM}}
```

When the form is submitted (on asset save), `Auxiliary Map` will be updated automatically by the JSF components. However, the attribute data does not yet have the new value. Before the asset is saved, the presave process for each attribute, if present, is invoked which reassembles the values in the map as attribute data.

Chapter 5

Filtering

Custom search filters can be written by implementing the `SearchFilter` interface.

This chapter contains the following sections:

Filtering Search Results

Search results are filtered using the filter infrastructure built into Dash. The filters are configured in `applicationContext.xml`. New search filters can be written by implementing the `SearchFilter` interface and its `filter ()` method. After the filter is written it should be injected into the search service.

Sample Code for a New Filter

```
/*
 * Copyright 2006 FatWire Corporation. Title, ownership rights,
and intellectual property rights in and to this software
 * remain with FatWire Corporation. This software is protected by
international copyright laws and treaties, and may be
 * protected by other law. Violation of copyright laws may result
in civil liability and criminal penalties.
 */
package com.fatwire.cs.ui.model.bo.search.filter;

import java.util.LinkedList;
import java.util.List;

import COM.FutureTense.Interfaces.ICS;

import com.fatwire.cs.ui.exception.UICSAccessFailureException;
import com.fatwire.cs.ui.model.bo.search.UISearchResult;
import com.fatwire.cs.ui.model.service.context.ServiceContext;
import com.fatwire.cs.ui.view.backing.GenericUtil;
import com.fatwire.cs.ui.view.constant.CSServiceConstants;

/**
 * Dash cannot support certain assettypes.Those assettypes should
be filtered.The AssetTypeFilter will filter using the
 * utility method available in GenericUtil.
 *
 * @author
 */
public class AssetTypeFilter implements SearchFilter
{

    /*
     * Filters the search results based on the assettype
     *
     * @see
com.fatwire.cs.ui.model.bo.search.filters.SearchFilter#filter(java
.util.List)
     */
    public List<UISearchResult> filter(ServiceContext context,
List<UISearchResult> searchresults) throws
UICSAccessFailureException
```

```
    {

        ICS ics = GenericUtil.getICS();
        List<UISearchResult> results = new
LinkedList<UISearchResult>();
        for (UISearchResult searchResult : searchresults)
        {
            String assetType = searchResult.getAssetType();
            boolean canSearch =
GenericUtil.canDashSearch(assetType, ics);
            if (canSearch)
            {

searchResult.setLink(!CSServiceConstants.RESTRICTED_ASSET_TYPES.co
ntains(assetType));
                results.add(searchResult);

            }
        }
        return results;
    }

}
```

Sample Filter Configuration

Below is the configuration for the sample filter shown on :

```
<bean id="assetTypeSearchFilter"
   class="com.fatwire.cs.ui.model.bo.search.filter.AssetTypeFilter">
</bean>
<bean id="searchService"
   class="com.fatwire.cs.ui.model.service.impl.SearchServiceImpl">
   <property name="filters">
      <list>
         <ref local="assetTypeSearchFilter"/>
      </list>
   </property>
</bean>
```

Chapter 6

Displaying Advanced Screens in the Dash Interface

The Dash interface is capable of displaying screens that are native to Content Server's Advanced interface.

This chapter contains the following sections:

- Configuring a Custom Tree Tab and its Nodes
- Sample CSElement: Creating Nodes in Custom Tree Tabs

Configuring a Custom Tree Tab and its Nodes

The asset tree in Dash can be configured with custom tabs to display custom pages. The first step is to create the custom tree tab; the next step is to populate the tree tab with nodes.

To create a custom tree tab

1. Log in to the Advanced interface.

2. Click the **Admin** tab and click on tree link.

3. Click **Add New Tree Tab**.

4. Populate the form. In the "Element name" field, specify the name of the CSElement that will populate the nodes of the custom tab.

To populate the custom tree tab with nodes

Create a CSElement asset that specifies as many as nodes as necessary for the custom tree tab. A sample CSElement is given in the next section.

Sample CSElement: Creating Nodes in Custom Tree Tabs

The following CSElement creates a node named "Workflow Email," which is displayed in any custom tree tab that specifies the CSElement and its path:

```
<ICS.ENCODE BASE="ContentServer" SESSION="true"
    OUTPUT="workflowEmail"> <ICS.ARGUMENT NAME="pagename"
VALUE="OpenMarket/Xcelerate/Admin/WorkflowSubjectFront"/>
    <ICS.ARGUMENT NAME="action" VALUE="list"/>
</ICS.ENCODE>
<CALLELEMENT NAME="OpenMarket/Gator/UIFramework/BuildTreeNode"
    SCOPED="STACKED">
<ARGUMENT NAME="Label" VALUE="WorkflowEmail"/>
<ARGUMENT NAME="Description" VALUE="WorkFlow Email"/>
<ARGUMENT NAME="ID" VALUE=""/>
<ARGUMENT NAME="ExecuteURL" VALUE="Variables.workflowEmail"/>
<ARGUMENT NAME="OpURL" VALUE=""/>
<ARGUMENT NAME="Image" VALUE="Variables.cs_imagedir/OMTree/
    TreeImages/QuestionMark.gif"/>
</CALLELEMENT>
```

`Label`
    Text to be displayed for this node in the tree. The label need not be unique.

`Description`
    The alternative to `Label`, if so chosen on the tree-wide pop-up menu.

`ID`
    String identifier which is unique within the tree. This ID is specified by the application. This will be used by the application to express selection paths, etc.

ExecuteURL
> URI value of the page to be displayed for "Execute" action, if the action is supported. If the node is not "executable", then this field must not be included in the node data. This value will be prepended with the value from the `ServerBaseURL` applet parameter value.

Image
> URI for the image to be prepended to the `Label`. If this field is not included in the node data, no image will be displayed for that node.

OpURL
> URI to execute a specified action on the server. This value will be prepended with the value from the `ServerBaseURL` applet parameter value.

Chapter 7

Validation

The Dash interface supports custom validation.

This chapter contains the following sections:

- Server Side Validation (pre/post insert/edit/delete)

- Usage

Server Side Validation (pre/post insert/edit/delete)

Custom validation is supported in the Dash interface in the same way it is supported in the Advanced user interface. Users can validate the asset in the `preupdate.xml` and `postupdate.xml` files. Validation errors can be added to the exception manager and will be handled by the Dash interface automatically.

In this framework, validation can be done only in a `.jsp` file. Hence, you will need to call a validation JSP from `preupdate.xml`.

Various parameters available in the `ics` scope are:

- `Asset` – Can be obtained using `ics.GetObj()`. The asset will have attribute details of the asset.

- `dashUpdate` – Can be obtained using `ics.GetVar`. This will say whether the call is from Dash.

- `dashUpdateType` – Can be obtained using `ics.GetVar`. This variable will have one of the following values: `edit`, `create`, or `delete`, depending on the user operation.

Usage

A sample implementation of the validation framework is shown below.

Preupdate.xml

`Preupdate.xml` is located in the following path:

`OpenMarket\Xcelerate\AssetType\XXXXXX\preupdate.xml`

where *XXXXX* represents the asset type (such as `Content_C`, or `Product_C`).

```
<if COND="Variables.updatetype=edit">
  <then>
  </then>

<else><if COND="Variables.updatetype=create">
  <then>
  </then>

<else><if COND="Variables.updatetype=delete">
  <then>
  </then>

<else><if COND="Variables.updatetype=remotepost">
  <then>
  </then>

<else><if COND="Variables.updatetype=updatefrom">
  <then>
  </then>
```

```
        <!-- If the asset is updated from dash -->
    <else><if COND="Variables.dashUpdate=dashUpdate">
    <then>
        <if COND="Variables.dashUpdateType=create">
    <then>
            <!--if the operation is create-->
    </then>
    </if>
    <if COND="Variables.dashUpdateType=edit">
    <then>
            <!--if the operation is edit-->
            <callelement NAME="OpenMarket/Xcelerate/AssetType/
                ${BasicAssetType}/Validate"/>
    </then>
    </if>
    <if COND="Variables.dashUpdateType=delete">
    <then>
            <!--if the operation is delete-->
    </then>
    </if>
      <!--
        Uncomment this code for refreshing dash tree for any
            asset create/update/delete operation
        <callelement NAME="OpenMarket/Xcelerate/Actions/
            RefreshDashTree"/>
      -->
    </then>
```

Validation

> **Note**
>
> Due to JSF binding, data must be treated as list, by using
>
> ```
> assetData.getAttributeData(AttributeName).getDataAsList()
> ```
> instead of:
> ```
> assetData.getAttributeData(AttributeName).getData()
> ```

The validation.jsp file is available in the DashCustomization_SampleCode folder, provided with this guide.

Chapter 8

Function Privileges

Changing permissions to functions in the Dash interface is accomplished by implementing the `Access` interface and changing the `applicationContext.xml` file.

This chapter contains the following section:

Changing Permissions to Functions

Functions implemented in Dash are the following:

Approve	Edit	Share
Check In	Inspect	Show Version
Check Out	Rollback	Undo Checkout

Utility methods are available in `AccessBean.java`. The implementations can be used by invoking the methods in `AccessBean.java`.

Permissions to the functions listed above are implemented in Dash using the `Access` interface. Changing permissions to functions is accomplished by implementing the `Access` interface and changing the `applicationContext.xml` file. For example:

```
<bean id="safe" class="com.fatwire.cs.ui.authorization.SAFE">
  </bean>
id=function privilege name
class=com.abc.customClass
```

Appendices

This part supplements the information in Chapter 4, "Customizing the Rendering of Attributes." It provides sample code and configurations for various attribute renderers.

This part contains the following appendices:

- Appendix A, "CSElement-Based Type I Attribute Renderer Configurations"
- Appendix B, "CSElement-Based Type II Attribute Renderer Configurations"
- Appendix C, "JSF-Based Type I Attribute Renderer Configurations"
- Appendix D, "JSF-Based Type II Attribute Renderer Configurations"

Appendix A

CSElement-Based Type I Attribute Renderer Configurations

This appendix supplements the information in Chapter 4, "Customizing the Rendering of Attributes" by providing sample code for attribute renderers, and configuration examples.

This appendix contains the following sections:

- Sample Code
- Configuration Models and Examples

Sample Code

For sample code, refer to the following text files (located in the folder `DashCustomization_SampleCode`, provided with this guide):

- `multiValuedTextRenderer.jsp`
- `htmltextdataRenderer.jsp`

> **Note**
>
> For information regarding the creation and configuration of attribute renderers, see Chapter 4, "Customizing the Rendering of Attributes."

Configuration Models and Examples

This section provides configuration examples for CSElement-based Type I attribute renderers. For definitions of terms and parameters, see Table 3, on page 28.

Model 1.   Includes All Parameters

Model 2.   Excludes AttributeName

Model 3.   Excludes AttributeName and AssetType (AttributeEditor is Required)

Model 1. Includes All Parameters

Description:

This model is the most specific. It is used to configure an attribute renderer for a specific attribute of a specific asset type, using a specific attribute editor

Syntax:

```
attributerendererkey:
  AttributeName.AssetType.AttributeEditor.attribute_renderer
attributerendererclass:
  com.fatwire.cs.ui.view.renderer.attribute.CSElementAttributeRenderer
cselement:
  fatwire/dash/attributerenderers/AttributeRenderer
```

Example:

```
attributerendererkey:
  unnamed.Page.PICKFROMTREE.attribute_renderer
attributerendererclass:
  com.fatwire.cs.ui.view.renderer.attribute.CSElementAttributeRenderer
cselement:
  fatwire/dash/attributerenderers/GroupedAssetInspectRenderer
```

Model 2. Excludes *AttributeName*

Description:

This model is used to configure an attribute renderer for all attributes of a specific asset type, using a specific attribute editor:

Syntax:

```
attributerendererkey:
  AssetType.AttributeEditor.attribute_renderer
attributerendererclass:
  com.fatwire.cs.ui.view.renderer.attribute.CSElementAttributeRenderer
cselement:
  fatwire/dash/attributerenderers/AttributeRenderer
```

Model 3. Excludes *AttributeName* and *AssetType* (*AttributeEditor* is Required)

Description:

This model is generic. It is used to configure an attribute renderer for all attributes of all asset types, using a specific attribute editor:

Syntax:

```
attributerendererkey:
  AttributeEditor.attribute_renderer
attributerendererclass:
  com.fatwire.cs.ui.view.renderer.attribute.CSElementAttributeRenderer
cselement:
  fatwire/dash/attributerenderers/AttributeRenderer
```

Examples:

```
attributerendererkey:
  CHECKBOXES.attribute_renderer
attributerendererclass:
  com.fatwire.cs.ui.view.renderer.attribute.CSElementAttributeRenderer
cselement:
  fatwire/dash/attributerenderers/CheckBoxDataRenderer


attributerendererkey:
  DATEPICKER.attribute_renderer
attributerendererclass:
  com.fatwire.cs.ui.view.renderer.attribute.CSElementAttributeRenderer
cselement:
  fatwire/dash/attributerenderers/DateDataRenderer


attributerendererkey:
  EWEBEDITPRO.attribute_renderer
attributerendererclass:
  com.fatwire.cs.ui.view.renderer.attribute.CSElementAttributeRenderer
```

```
cselement:
  fatwire/dash/attributerenderers/FCKEditorRenderer


attributerendererkey:
  FCKEDITOR.attribute_renderer
attributerendererclass:
  com.fatwire.cs.ui.view.renderer.attribute.CSElementAttributeRenderer
cselement:
  fatwire/dash/attributerenderers/FCKEditorRenderer


attributerendererkey:
  IMAGEEDITOR.attribute_renderer
attributerendererclass:
  com.fatwire.cs.ui.view.renderer.attribute.CSElementAttributeRenderer
cselement:
  fatwire/dash/attributerenderers/ImageEditorRenderer


attributerendererkey:
  IMAGEPICKER.attribute_renderer
attributerendererclass:
  com.fatwire.cs.ui.view.renderer.attribute.CSElementAttributeRenderer
cselement:
  fatwire/dash/attributerenderers/ImagePickerRenderer


attributerendererkey:
  PICKASSET.attribute_renderer
attributerendererclass:
  com.fatwire.cs.ui.view.renderer.attribute.CSElementAttributeRenderer
cselement:
  fatwire/dash/attributerenderers/PickFromTreeDataRenderer


attributerendererkey:
  PICKFROMTREE.attribute_renderer
attributerendererclass:
  com.fatwire.cs.ui.view.renderer.attribute.CSElementAttributeRenderer
cselement:
  fatwire/dash/attributerenderers/PickFromTreeDataRenderer


attributerendererkey:
  PULLDOWN.attribute_renderer
attributerendererclass:
  com.fatwire.cs.ui.view.renderer.attribute.CSElementAttributeRenderer
cselement:
  fatwire/dash/attributerenderers/PullDownDataRenderer


attributerendererkey:
  RADIOBUTTONS.attribute_renderer
attributerendererclass:
  com.fatwire.cs.ui.view.renderer.attribute.CSElementAttributeRenderer
```

```
cselement:
  fatwire/dash/attributerenderers/RadioButtonDataRenderer


attributerendererkey:
  RATING.attribute_renderer
attributerendererclass:
  com.fatwire.cs.ui.view.renderer.attribute.CSElementAttributeRenderer
cselement:
  fatwire/dash/attributerenderers/RatingDataRenderer


attributerendererkey:
  REALOBJECT.attribute_renderer
attributerendererclass:
  com.fatwire.cs.ui.view.renderer.attribute.CSElementAttributeRenderer
cselement:
  fatwire/dash/attributerenderers/FCKEditorRenderer


attributerendererkey:
  RENDERFLASH.attribute_renderer
attributerendererclass:
  com.fatwire.cs.ui.view.renderer.attribute.CSElementAttributeRenderer
cselement:
  fatwire/dash/attributerenderers/FlashDataRenderer


attributerendererkey:
  TEXTAREA.attribute_renderer
attributerendererclass:
  com.fatwire.cs.ui.view.renderer.attribute.CSElementAttributeRenderer
cselement:
  fatwire/dash/attributerenderers/TextAreaDataRenderer


attributerendererkey:
  TEXTFIELD.attribute_renderer
attributerendererclass:
  com.fatwire.cs.ui.view.renderer.attribute.CSElementAttributeRenderer
cselement:
  fatwire/dash/attributerenderers/TextDataRenderer


attributerendererkey:
  TIMESTAMP.attribute_renderer
attributerendererclass:
  com.fatwire.cs.ui.view.renderer.attribute.CSElementAttributeRenderer
cselement:
  fatwire/dash/attributerenderers/DateDataRenderer


attributerendererkey:
  UPLOAD.attribute_renderer
attributerendererclass:
  com.fatwire.cs.ui.view.renderer.attribute.CSElementAttributeRenderer
cselement:
  fatwire/dash/attributerenderers/FileDataRenderer
```

Appendix B

CSElement-Based Type II Attribute Renderer Configurations

This appendix supplements the information in Chapter 4, "Customizing the Rendering of Attributes" by providing sample code for attribute renderers, and configuration examples.

This appendix contains the following sections:

- Sample Code
- Configuration Models and Examples

Sample Code

For sample code, refer to the file `htmltextdatadisplayRenderer.jsp` (located in the folder `DashCustomization_SampleCode`, provided with this guide).

Note

For information regarding the creation and configuration of attribute renderers, see Chapter 4, "Customizing the Rendering of Attributes."

Configuration Models and Examples

This section provides configuration examples for CSElement-based Type II attribute renderers. For definitions of terms and parameters, see Table 3, on page 28.

Model 1.   Includes All Parameters

Model 2.   Excludes AttributeName

Model 3.   Excludes AttributeName, AssetType

Model 4.   Excludes AttributeName, AssetType, AttributeEditor (Datatype is Required)

Model 1. Includes All Parameters

Description:
This model is the most specific. It is used to configure an attribute renderer for a specific attribute of a specific asset type, using a specific attribute editor:

Syntax:
```
attributerendererkey:
  AttributeName.AssetType.AttributeEditor.Datatype.attribute_
  renderer
attributerendererclass:
  com.fatwire.cs.ui.view.renderer.attribute.CSElementAttributeRenderer
cselement:
  fatwire/dash/attributerenderers/AttributeRenderer
```

Example:
```
attributerendererkey:
  unnamed.Page.PICKFROMTREE.assetreference.attribute_renderer
attributerendererclass:
  com.fatwire.cs.ui.view.renderer.attribute.CSElementAttributeRenderer
cselement:
  fatwire/dash/attributerenderers/GroupedAssetInspectRenderer
```

Model 2. Excludes *AttributeName*

Description:

This model is used to configure an attribute renderer for all attributes of a specific asset type, using a specific attribute editor.

Syntax:
```
attributerendererkey:
  AssetType.AttributeEditor.Datatype.attribute_renderer
attributerendererclass:
  com.fatwire.cs.ui.view.renderer.attribute.CSElementAttributeRenderer
cselement:
  fatwire/dash/attributerenderers/AttributeRenderer
```

Examples:

<div align="center">

Note

</div>

The examples below are provided only to illustrate the model. They are not implemented in Dash.

```
attributerendererkey:
  Product_C.blob.attribute_renderer
attributerendererclass:
  com.fatwire.cs.ui.view.renderer.attribute.CSElementAttributeRenderer
cselement:
  fatwire/dash/attributerenderers/BlobInspectRendererProduct
```

```
attributerendererkey:
  Content_C.url.attribute_renderer
attributerendererclass:
  com.fatwire.cs.ui.view.renderer.attribute.CSElementAttributeRenderer
cselement:
  fatwire/dash/attributerenderers/BlobInspectRendererContent
```

Model 3. Excludes *AttributeName, AssetType*

Description:

This model is used to configure an attribute renderer for all attributes of all asset types, using a specific attribute editor.

Syntax:
```
attributerendererkey:
  AttributeEditor.Datatype.attribute_renderer
attributerendererclass:
  com.fatwire.cs.ui.view.renderer.attribute.CSElementAttributeRenderer
cselement:
  fatwire/dash/attributerenderers/AttributeRenderer
```

Examples:
```
attributerendererkey:
  FCKEDITOR.url.attribute_renderer
attributerendererclass:
  com.fatwire.cs.ui.view.renderer.attribute.CSElementAttributeRenderer
cselement:
  fatwire/dash/attributerenderers/FCKEditorInspectRenderer


attributerendererkey:
  FCKEDITOR.blob.attribute_renderer
attributerendererclass:
  com.fatwire.cs.ui.view.renderer.attribute.CSElementAttributeRenderer
cselement:
  fatwire/dash/attributerenderers/FCKEditorInspectRenderer


attributerendererkey:
  IMAGEPICKER.asset.attribute_renderer
attributerendererclass:
  com.fatwire.cs.ui.view.renderer.attribute.CSElementAttributeRenderer
cselement:
  fatwire/dash/attributerenderers/ImageInspectRenderer
```

Model 4. Excludes *AttributeName, AssetType, AttributeEditor (Datatype is Required)*

Description:

This model is generic. It is used to configure an attribute renderer for all attributes of all asset types, using all attribute editors.

Syntax:
```
attributerendererkey:
  Datatype.attribute_renderer
attributerendererclass:
  com.fatwire.cs.ui.view.renderer.attribute.CSElementAttributeRenderer
cselement:
  fatwire/dash/attributerenderers/AttributeRenderer
```

Examples:
```
attributerendererkey:
  asset.attribute_renderer
attributerendererclass:
  com.fatwire.cs.ui.view.renderer.attribute.CSElementAttributeRenderer
cselement:
  fatwire/dash/attributerenderers/AssetInspectRenderer


attributerendererkey:
  assetreference.attribute_renderer
attributerendererclass:
  com.fatwire.cs.ui.view.renderer.attribute.CSElementAttributeRenderer
```

```
cselement:
  fatwire/dash/attributerenderers/AssetInspectRenderer


attributerendererkey:
  BINARY.attribute_renderer
attributerendererclass:
  com.fatwire.cs.ui.view.renderer.attribute.CSElementAttributeRenderer
cselement:
  fatwire/dash/attributerenderers/BlobInspectRenderer


attributerendererkey:
  blob.attribute_renderer
attributerendererclass:
  com.fatwire.cs.ui.view.renderer.attribute.CSElementAttributeRenderer
cselement:
  fatwire/dash/attributerenderers/BlobInspectRenderer


attributerendererkey:
  date.attribute_renderer
attributerendererclass:
  com.fatwire.cs.ui.view.renderer.attribute.CSElementAttributeRenderer
cselement:
  fatwire/dash/attributerenderers/DateInspectRenderer


attributerendererkey:
  url.attribute_renderer
attributerendererclass:
  com.fatwire.cs.ui.view.renderer.attribute.CSElementAttributeRenderer
cselement:
  fatwire/dash/attributerenderers/BlobInspectRenderer


attributerendererkey:
    default_inspect_renderer
attributerendererclass:
  com.fatwire.cs.ui.view.renderer.attribute.CSElementAttributeRenderer
cselement:
  fatwire/dash/attributerenderers/StaticTextInspectRenderer
```

Appendix C

JSF-Based Type I Attribute Renderer Configurations

This appendix supplements the information in Chapter 4, "Customizing the Rendering of Attributes" by providing sample code for attribute renderers, and configuration examples.

This appendix contains the following sections:

- Sample Code
- Configuration Models and Examples

Sample Code

For sample code, refer to the file `TextDataRenderer.java` (located in the folder `DashCustomization_SampleCode`, provided with this guide).

Note

For information regarding the creation and configuration of attribute renderers, see Chapter 4, "Customizing the Rendering of Attributes."

Configuration Models and Examples

This section provides configuration examples for JSF-based Type II attribute renderers. For definitions of terms and parameters, see Table 3, on page 28.

Model 1.   Includes All Parameters

Model 2.   Excludes AttributeName

Model 3.   Excludes AttributeName and AssetType (AttributeEditor is Required)

Model 1. Includes All Parameters

Description:

This model is the most specific. It is used to configure an attribute renderer for a specific attribute belonging to a specific asset type, using a specific attribute editor

Syntax:

```
attributerendererkey:
 AttributeName.AssetType.AttributeEditor.attribute_renderer
attributerendererclass:
 com.fatwire.CustomAttributeRenderer
```

Example:

```
attributerendererkey:
 unnamed.Page.PICKFROMTREE.attribute_renderer
attributerendererclass:
 com.fatwire.cs.ui.view.renderer.attribute.GroupedAssetInspectRenderer
```

Model 2. Excludes *AttributeName*

Description:

This model is used to configure an attribute renderer for all attributes of a specific asset type, using a specific attribute editor:

Syntax:
```
attributerendererkey:
  AssetType.AttributeEditor.attribute_renderer
attributerendererclass:
  com.fatwire.CustomAttributeRenderer
```

Example:
```
attributerendererkey:
  Page.PICKFROMTREE.attribute_renderer
attributerendererclass:
  com.fatwire.cs.ui.view.renderer.attribute.GroupedAssetInspectRenderer
```

Model 3. Excludes AttributeName and AssetType (*AttributeEditor* is Required)

Description:

This model is used to configures an attribute renderer for all attributes of all asset types, using a specific attribute editor:

Syntax:
```
attributerendererkey:
  AttributeEditor.attribute_renderer
attributerendererclass:
  com.fatwire.CustomAttributeRenderer
```

Examples:
```
attributerendererkey:
  CHECKBOXES.attribute_renderer
attributerendererclass:
  com.fatwire.cs.ui.view.renderer.attribute.CheckBoxDataRenderer


attributerendererkey:
  DATEPICKER.attribute_renderer
attributerendererclass:
  com.fatwire.cs.ui.view.renderer.attribute.DateDataRenderer


attributerendererkey:
  EWEBEDITPRO.attribute_renderer
attributerendererclass:
  com.fatwire.cs.ui.view.renderer.attribute.FCKEditorRenderer


attributerendererkey:
  FCKEDITOR.attribute_renderer
attributerendererclass:
  com.fatwire.cs.ui.view.renderer.attribute.FCKEditorRenderer


attributerendererkey:
  IMAGEEDITOR.attribute_renderer
```

```
attributerendererclass:
  com.fatwire.cs.ui.view.renderer.attribute.ImageEditorRenderer


attributerendererkey:
  IMAGEPICKER.attribute_renderer
attributerendererclass:
  com.fatwire.cs.ui.view.renderer.attribute.ImagePickerRenderer


attributerendererkey:
  PICKASSET.attribute_renderer
attributerendererclass:
  com.fatwire.cs.ui.view.renderer.attribute.PickFromTreeDataRenderer


attributerendererkey:
  PICKFROMTREE.attribute_renderer
attributerendererclass:
  com.fatwire.cs.ui.view.renderer.attribute.PickFromTreeDataRenderer


attributerendererkey:
  PULLDOWN.attribute_renderer
attributerendererclass:
  com.fatwire.cs.ui.view.renderer.attribute.PullDownDataRenderer


attributerendererkey:
  RADIOBUTTONS.attribute_renderer
attributerendererclass:
  com.fatwire.cs.ui.view.renderer.attribute.RadioButtonDataRenderer


attributerendererkey:
  RATING.attribute_renderer
attributerendererclass:
  com.fatwire.cs.ui.view.renderer.attribute.RatingDataRenderer


attributerendererkey:
  REALOBJECT.attribute_renderer
attributerendererclass:
  com.fatwire.cs.ui.view.renderer.attribute.FCKEditorRenderer


attributerendererkey:
  RENDERFLASH.attribute_renderer
attributerendererclass:
  com.fatwire.cs.ui.view.renderer.attribute.FlashDataRenderer


attributerendererkey:
  TEXTAREA.attribute_renderer
attributerendererclass:
  com.fatwire.cs.ui.view.renderer.attribute.TextAreaDataRenderer
```

```
attributerendererkey:
  TEXTFIELD.attribute_renderer
attributerendererclass:
  com.fatwire.cs.ui.view.renderer.attribute.TextDataRenderer


attributerendererkey:
  TIMESTAMP.attribute_renderer
attributerendererclass:
  com.fatwire.cs.ui.view.renderer.attribute.DateDataRenderer


attributerendererkey:
  UPLOAD.attribute_renderer
attributerendererclass:
  com.fatwire.cs.ui.view.renderer.attribute.FileDataRenderer
```

Appendix D

JSF-Based Type II Attribute Renderer Configurations

This appendix supplements the information in Chapter 4, "Customizing the Rendering of Attributes" by providing sample code for attribute renderers and presenting configuration examples.

This appendix contains the following sections:

- Sample Code
- Configuration Models and Examples

Sample Code

For sample code, refer to the file `StaticTextInspectRenderer.java` (located in the folder `DashCustomization_SampleCode`, provided with this guide).

Note

For information regarding the creation and configuration of attribute renderers, see Chapter 4, "Customizing the Rendering of Attributes."

Configuration Models and Examples

This section provides configuration examples for JSF-based Type II attribute renderers. For definitions of terms and parameters, see Table 3, on page 28.

Model 1.   Includes All Parameters

Model 2.   Excludes AttributeName

Model 3.   Excludes AttributeName, AssetType

Model 4.   Excludes AttributeName, AssetType, AttributeEditor (Datatype is Required)

Model 1. Includes All Parameters

Description:

This model is the most specific. It is used to configure an attribute renderer for a specific attribute belonging to a specific asset type, using a specific attribute editor:

Syntax:
```
attributerendererkey:
  AttributeName.AssetType.AttributeEditor.Datatype.attribute_
  renderer
attributerendererclass:
  com.fatwire.CustomAttributeRenderer
```

Example:
```
attributerendererkey:
  unnamed.Page.PICKFROMTREE.assetreference.attribute_renderer
attributerendererclass:
  com.fatwire.cs.ui.view.renderer.attribute.GroupedAssetInspectRenderer
```

Model 2. Excludes *AttributeName*

Description:

This model is used to configure an attribute renderer for all attributes of a specific asset type, using a specific attribute editor.

Syntax:
```
attributerendererkey:
 AssetType.AttributeEditor.Datatype.attribute_renderer
attributerendererclass:
 com.fatwire.CustomAttributeRenderer
```

Examples:

> **Note**
>
> The following examples are provided only to illustrate the model. They are not
> implemented in Dash.

```
attributerendererkey:
 Product_C.blob.attribute_renderer
attributerendererclass:
 com.fatwire.cs.ui.view.renderer.attribute.BlobInspectRendererProduct
```

```
attributerendererkey:
 Content_C.url.attribute_renderer
attributerendererclass:
 com.fatwire.cs.ui.view.renderer.attribute.BlobInspectRendererContent
```

Model 3. Excludes *AttributeName, AssetType*

Description:

This model is used to configure an attribute renderer for all attributes of all asset types,
using a specific attribute editor.

Syntax:
```
attributerendererkey:
 AttributeEditor.Datatype.attribute_renderer
attributerendererclass:
 com.fatwire.CustomAttributeRenderer
```

Examples:
```
attributerendererkey:
 FCKEDITOR.url.attribute_renderer
attributerendererclass:
 com.fatwire.cs.ui.view.renderer.attribute.FCKEditorInspectRenderer
```

```
attributerendererkey:
 FCKEDITOR.blob.attribute_renderer
attributerendererclass:
 com.fatwire.cs.ui.view.renderer.attribute.FCKEditorInspectRenderer
```

```
attributerendererkey:
 IMAGEPICKER.asset.attribute_renderer
```

```
attributerendererclass:
  com.fatwire.cs.ui.view.renderer.attribute.ImageInspectRenderer
```

Model 4. Excludes AttributeName, AssetType, AttributeEditor (*Datatype* is Required)

Description:

This model is generic. It is used to configure an attribute renderer for all attributes of all asset types.

Syntax:
```
attributerendererkey:
  Datatype.attribute_renderer
attributerendererclass:
  com.fatwire.CustomAttributeRenderer
```

Examples:
```
attributerendererkey:
  asset.attribute_renderer
attributerendererclass:
  com.fatwire.cs.ui.view.renderer.attribute.AssetInspectRenderer
```

```
attributerendererkey:
  assetreference.attribute_renderer
attributerendererclass:com.fatwire.cs.ui.view.renderer.attribute.
  AssetInspectRenderer
```

```
attributerendererkey:
  BINARY.attribute_renderer
attributerendererclass:
  com.fatwire.cs.ui.view.renderer.attribute.BlobInspectRenderer
```

```
attributerendererkey:
  blob.attribute_renderer
attributerendererclass:
  com.fatwire.cs.ui.view.renderer.attribute.BlobInspectRenderer
```

```
attributerendererkey:
  date.attribute_renderer
attributerendererclass:
  com.fatwire.cs.ui.view.renderer.attribute.DateInspectRenderer
```

```
attributerendererkey:
  url.attribute_renderer
attributerendererclass:
  com.fatwire.cs.ui.view.renderer.attribute.BlobInspectRenderer
```

```
attributerendererkey:
    default_inspect_renderer
attributerendererclass:
 com.fatwire.cs.ui.view.renderer.attribute.StaticTextInspectRenderer
```