

**Content Server URL Assemblers  
including  
Lightweight Abstract Assembler  
Item Context Assembler  
Site Plan Tree Support**

Author: Tony Field

Created: July 23, 2009

# Table of Contents

Table of Contents 2

Overview 3

Executive Summary 3

Quick Start for the Impatient 3

Configuration 3

Deployment 4

Template Changes 4

FW/Wrapper.jsp 4

Page/Link and Article/Link 4

Asset Configuration 4

Component Overview 5

LightweightAbstractAssembler 5

ItemContextAssembler 5

Overview 5

Restrictions 6

Usage 6

Multivariate Testing Support 6

Configuration 6

For more information 7

Site Plan Tree Helper 7

Generating URLs 7

Resolving URLs 8

How Helper constructs item-context 10

Aliasing Strategies 10

Configuration 10

Asset Aliasing Strategies 11

Overview 11

IdAliasingStrategy 12

NameAliasingStrategy 12

PathAliasingStrategy 12

PageMetadataArticleAliasingStrategy 13

MultilingualPageMetadataArticleAliasingStrategy 13

Deployment Dependencies 13

# Overview

Writing URL assemblers for Content Server can be a tedious task. This library exists for two reasons: to simplify the work required to develop an assembler, and to create a set of multi-purpose, easy-to-use assemblers that can be used in the vast majority of sites without requiring that any code be written.

# Executive Summary

A new abstract assembler (base class) has been written that is much easier to use than the abstract assembler that ships with Content Server. This facilitates development of new assemblers.

A generic assembler, called ItemContextAssembler, is applicable for any site needs to represent a single object placed in a specific context in its URL. It turns out that this applies to most sites. (E.g. “products in a catalogue”, “articles on a page”, and “files in a folder” all fit this model)

A helper class that supports ItemContextAssembler when used with sites based on the Site Plan Tree is provided. It includes an infrastructure for converting an asset into a pretty alias using a configured strategy (e.g. conversion of the asset ID Page:123123123123 into the alias “about-us”). The context is automatically generated using “p” and the aliasing strategy. This library includes 5 such strategies.

Developing a new URL style often is as simple as simply building a new strategy, which is many times shorter than developing a whole assembler from scratch.

# Quick Start for the Impatient

This section describes how to configure the ItemContextAssembler to be used with the SitePlanTree, for sites using the path of the Page asset as the alias. Several optional features are left out. For information about how to do things a little differently, please read this whole document.

## Configuration

In cs.war/WEB-INF/classes/servlet-request.properties, set the following properties:

uri.assembler.1.shortform = ic

uri.assembler.1.classname = com.fatwire.developernet.uri.itemcontext.ItemContextAssembler

uri.assembler.2.shortform = query

uri.assembler.2.classname = com.fatwire.cs.core.uri.QueryAssembler

com.fatwire.developernet.uri.itemcontext.global-wrapper-pagename = FW/Wrapper

com.fatwire.developernet.uri.itemcontext.global-template-pagename = FW/Layout

## Deployment

Deploy cs-url-assemblers-?.jar and cs-facades-?.jar files into cs.war/WEB-INF/lib where the ? denotes the version number of the library.

## Template Changes

### FW/Wrapper.jsp

Add the following code to the top of your wrapper:

<%

Helper.resolveItemContextAliasesAndPopulateIcs(ics);

%>

### Page/Link and Article/Link

Integrate the following code into your Link templates to ensure that links are properly generated:

<%

String sLocaleName = “en\_US”; // or however you get it

String linktext = “click here”; // or however you get it

Helper helper = new Helper(ics);

String ictx = helper.computeItemContext(id.getId(), sLocaleName);

String ialias = helper.computeAlias(id, sLocaleName);

%>

<render:gettemplateurl outstr="aUrl"

tid='<%=ics.GetVar("tid")%>'

slotname="pagelink"

site='<%=ics.GetVar("site")%>'

c='Page’ cid='<%=ics.GetVar("cid")%>'

tname='/Layout'
wrapperpage= RenderPage'>

<render:argument name="p" value='<%=ics.GetVar("p")%>' />

<render:argument name="item-alias" value='<%=ialias %>' />

<render:argument name="item-context" value='<%=ictx %>' />

</render:gettemplateurl>

<a href='<string:stream variable="aUrl"/>'<%=linktext%></a>

## Asset Configuration

Make sure your assets are populated into the SitePlanTree. Add a meta page (just a regular page asset) to the SitePlanTree immediately below the site name called “Main”, and add your main navigation area just below that. (see “lowest level” discussion below for more info)

Make sure that your page asset and all articles linkable directly from your page assets specify a “path” attribute that contains no spaces.

Make sure your wrapper SiteEntry is set to FW/Wrapper (or update the property to point to your own wrapper). Make sure your layout template is set to FW/Layout (or update the property to point to your own layout template).

# Component Overview

## LightweightAbstractAssembler

The LightweightAbstractAssembler class greatly simplifies the work required to write a URL assembler. Handling property configuration, default value support, URL encoding, decoding, and query string processing, it handles all of the tedious tasks dealing with URL assembly, leaving only the core business logic to be developed by the developers.

Unlike its predecessor, com.fatwire.cs.core.uri.AbstractAssembler, the complexity of utilizing this helper class is gone. There are no callbacks, no complex sub-interfaces. The assemble and disassemble methods are not implemented in this class – it only provides clear helper methods.

## ItemContextAssembler

### Overview

The ItemContextAssebler is designed to be a general purpose URL assembler that is applicable whenever a URL needs to be constructed by putting an item in a context. It does not support many other features, but the applicability of this approach is very broad.

Examples of single items in contexts are:

* A file in a folder
* An article on a site plan tree Page
* A product in a catalogue
* A press release on a given date

Three key parameters drive this assembler: “item-context”, “item-type”, and “item-alias”. They are concatenated together in a structure that looks like this:

<item-context>/<item-type>/<item-alias>

If this structure is generally suitable, this assembler is applicable.

### Restrictions

Besides requiring the three parameters described above, the assembler requires that the URL be a Satellite Server URL, designed to serve pages, not images or other blobs. Additionally, this assembler requires that pagename and childpagename match a configured value. If all of these conditions are met, the assembler will attempt to assemble the URL. If all of these conditions are not met, it will delegate URL assembly to the backup assembler, which as of July 24, 2009 is defined as the QueryAssembler. Perhaps in the future the backup assembler can be configurable.

### Usage

While this assembler can be used unmodified in all of the above situations, a nagging problem exists in that “item-context”, “item-type” and “item-alias” are not standard variables used by Content Server using any standard rendering models. However, they do naturally map to some. All that is needed is a conversion mechanism to map known variables into these three item-\* parameters. This mapping is done through a series of helper classes.

As of July 24, 2009, only the Site Plan Tree Helper exists. However, it is conceivable that others could be added and ship with this package at a later date. These helpers take input parameters and convert them into the item-\* parameters, and then back again. Because this conversion often requires database access, it is not possible for the conversion to occur in the assembler itself. Instead, conversion of site params into ItemContextAssembler params must be done at the time the link is created – usually just before invoking the render:calltemplate tag. In the FSII rendering model, this occurs in Link templates. The reverse lookup must occur before the site params are needed. Often the params are needed right away, which means the conversion back has to happen in a wrapper page or in a layout template (depending on caching and security requirements).

### Multivariate Testing Support

This assembler has support for integer-based variant parameter, which is appended to the end of the URL using a suffix “/v<int>” (e.g. /v1). This feature transparently allows multivariate testing by allowing this parameter to be passed on a URL (without requiring a query string, which is helpful for working with some popular web analytics packages) and converted into a variable that can be used in templates to differentiate page layouts, views, or content, etc. The variant parameter must be an integer, and it is optional.

### Configuration

The ItemContextAssembler contains several configuration parameters that address the definition of the required pagename and childpagename, a mapping system for item-type parameters (optional), the item type used by the item-context, and finally, a list of arguments that should always be unpacked from the ugly packedargs parameter.

The configuration file is the ServletRequest.properties file in cs.war/WEB-INF/classes. Here is the simplest possible configuration:

uri.assembler.1.shortform = ic

uri.assembler.1.classname = com.fatwire.developernet.uri.itemcontext.ItemContextAssembler

uri.assembler.2.shortform = query

uri.assembler.2.classname = com.fatwire.cs.core.uri.QueryAssembler

com.fatwire.developernet.uri.itemcontext.global-wrapper-pagename = FW/Wrapper

com.fatwire.developernet.uri.itemcontext.global-template-pagename = FW/Layout

### For more information

For more information about exactly how to use this assembler, consult the Java API Reference Guide for com.fatwire.developernet.uri.itemcontext.ItemContextAssembler.

## Site Plan Tree Helper

Most simple sites based on Content Server can be modeled using the Site Plan Tree. FatWire’s *BurlingtonFinancial* and *FirstSiteII* rendering models both utilize the site plan tree heavily for modeling their designs. In these cases, if it is true that any given URL will only need to specify either the Page (asset) being rendered, or a single asset placed on a page, then this assembler is very suitable.

Ultimately, URLs in this case require only c, cid, p, in addition to pagename (and possibly childpagename) to be able to render properly.

In order to take advantage of the ItemContext assembler, c/cid must be converted into item-type and item-alias, and p must be converted into item-context.

Finally, as long as pagename and childpagename are consistent for all URLs for the site, there will be no problem utilizing the ItemContext assembler.

The Site Plan Tree Helper class provides the conversion utilities described above.

### Generating URLs

To generate URLs that contain information about item-context, item-type and item-alias, it is important to pass c/cid and p into the com.fatwire.developernet.uri.siteplan.Helper class’s utility functions, and to then pass the result into the render:gettemplateurl tag. The code is very simple and looks like this:

<%

String sLocaleName = “en\_US”; // or however you get it

String linktext = “click here”; // or however you get it

Helper helper = new Helper(ics);

String ictx = helper.computeItemContext(id.getId(), sLocaleName);

String ialias = helper.computeAlias(id, sLocaleName);

%>

<%-- Note that we are still pasing c, cid, p, though the assembler

drops these. These are still in place in case for some

reason the assembler is unable to use the item-\*

information. --%>

<render:gettemplateurl outstr="aUrl"

tid='<%=ics.GetVar("tid")%>'

slotname="pagelink"

site='<%=ics.GetVar("site")%>'

c='Page’ cid='<%=ics.GetVar("cid")%>'

tname='/Layout'
wrapperpage= RenderPage'>

<render:argument name="p" value='<%=ics.GetVar("p")%>' />

<render:argument name="item-alias" value='<%=ialias %>' />

<render:argument name="item-context" value='<%=ictx %>' />

</render:gettemplateurl>

<a href='<string:stream variable="aUrl"/>'<%=linktext%></a>

From here, the ItemContext assembler will create a nice looking URL based on how the helper constructs item-alias and item-context.

### Resolving URLs

When the URL assembler processes the incoming request, it extracts item-context, item-type and item-alias from the URL. These parameters need to be converted back into c, cid, and p.

For this to occur, the Helper class must be utilized again. Ideally, resolution of c/cid/p would happen as soon as possible, but since there is often a performance cost to resolving these variables (due to database lookups), it is sometimes beneficial instead to defer the lookups at least far enough into the page’s call structure to allow the database lookup to be protected by a page cache.

This usually boils down to a decision between putting the lookup code in the Wrapper CSElement or into the Layout template.

For the sake of simplicity, we will assume that the lookup will occur in either the Wrapper or in java code called by the wrapper.

Resolving variables is simple. Here is the relevant code snippet:

#### Explicit

<%

String c = ics.GetVar("item-type");

String ialias = ics.GetVar("item-alias");

long longCid = -1;

String ictx = ics.GetVar("item-context");

long longP = -1;

Helper helper = new Helper(ics);

// Must get ‘p’ first because it is needed to resolve cid

if (ictx!= null) {

if (ics.GetVar("p") != null) {

// unexpected. Use p, instead of ictx, but maybe log warning

}

else {

longP = helper.resolvePForItemContext(ictx);

ics.SetVar("p", Long.toString(longP));

}

}

if (ialias != null) {

if (ics.GetVar("cid") != null) {

// unexpected. Use cid instead of ialias but maybe log warning

}

else {

longCid = helper.resolveCidFromAlias(c, ialias, longP);

ics.SetVar("cid", Long.toString(longCid));

}

}

%>

There is, however, a shortcut method that encapsulates all of the above code, making the required wrapper code much simpler. Instead of including the snippet above, it is acceptable to include the following snippet:

#### Automatic

<%

Helper.resolveItemContextAliasesAndPopulateIcs(ics);

%>

This shortcut function does all of the above and provides logging as well. However, it requires that the input variables be called item-type, item-alias, and item-context, and the output variables must be called c, cid, and p. In addition, this helper method returns the Dimension corresponding to the dimension used in mapping p to item-context. The longer example above does not describe how to look up dimensions, but a trivial function in the helper, resolveDimensionForItemContext, will work if required.

### Lowest-level-to-include and how Helper constructs item-context

The Site Plan Tree Helper class constructs the item-context variable by looking up the p page in the Site Plan Tree, and then walking the tree to the root page, stopping at a configured lowest level. This provides a list of page ids that need to be converted into pretty strings.

The lowest level configuration option exists so that it’s not necessary to display every single page in the site plan hierarchy in the URL. For instance, if a site plan tree has 4 pages at the root node that correspond to major functional areas on the site, but that do not need to be reflected in the URL, but all of the pages below them need to be reflected in the URL, the lowest-level-to-include should be set to 1. If there are 2 levels of metadata pages, the lowest-level-to-include should be set to 2. If there are none of these metadata pages, and every node in the site plan tree should appear in the URL, set the property com.fatwire.developernet.uri.siteplan.helper.lowest-level-to-include to 0.

Once the list of page ids is constructed, it is converted to a pretty string by using the exact same mechanism that is used to convert a c/cid pair into an item-alias. Thus, it is critical that the aliasing strategy be well defined and well understood.

If any path element in the URL does not have a valid alias, then item-context is not constructed.

### Aliasing Strategies

This helper class uses the AssetAliasingStrategy interface to map between an asset and its pretty alias. In addition, as described above, the Page asset’s p-based context is also derived using the AssetAliasingStrategy. This helper allows the strategy to be plugged in, supporting any arbitrary mechanism for converting an asset ID into a pretty alias.

Simply specify the class name in the property com.fatwire.developernet.uri.siteplan.helper.aliasing-strategy-class to achieve the desired behavior.

### Configuration

There are two required steps, and one conditional step in configuring this system:

1. Configure the ItemContextAssembler:

In servlet-request.properties, set the following properties:

uri.assembler.1.shortform = ic

uri.assembler.1.classname = com.fatwire.developernet.uri.itemcontext.ItemContextAssembler

uri.assembler.2.shortform = query

uri.assembler.2.classname = com.fatwire.cs.core.uri.QueryAssembler

com.fatwire.developernet.uri.itemcontext.item-type.parameter.FW\_Content\_C=article com.fatwire.developernet.uri.itemcontext.item-type.alias.article = FW\_Content\_C

com.fatwire.developernet.uri.itemcontext.global-wrapper-pagename = FW/Wrapper

com.fatwire.developernet.uri.itemcontext.global-template-pagename = FW/Layout

com.fatwire.developernet.uri.itemcontext.nopack-args = rendermode

1. Configure the SitePlanTree Helper:

In servlet-request.properties, set the following properties:

com.fatwire.developernet.uri.siteplan.helper.lowest-level-to-include = 1 // 1 is default)

com.fatwire.developernet.uri.siteplan.helper.aliasing-strategy-class = com.fatwire.developernet.uri.itemcontext.aliasing.PathAliasingStrategy // (or whatever strategy you want)

1. Configure your aliasing strategy

If your aliasing strategy requires additional configuration, this step needs to occur. Refer to the Java API Reference guide for the strategy in question to learn how to configure it.

## Asset Aliasing Strategies

### Overview

Asset Aliasing Strategies are pluggable components that convert a single asset into a pretty alias string, and back. This conversion is a core aspect of automatic pretty URL generation, but it can be re-used in other capacities as required. This document describes only how it is can be used in the context of URL assembly.

The primary interface contains only two worker methods:

/\*\*

\* Given an asset ID, compute the alias for that asset. It is not an

\* error if it is not possible to return an alias. In that case,

\* null is returned.

\*

\* @param id Asset ID

\* @param localeName name of locale to find the path for. Null is allowed,

\* in which case to translation is sought.

\* @return alias string fragment. If null is returned, it means that no

\* alias could be computed.

\*/

String computeAlias(AssetId id, String localeName);

/\*\*

\* This method attempts to resolve an asset ID from

\* an input item type name and alias.

\*

\* @param type asset type

\* @param alias alias for the asset

\* @return list of matching asset IDs, never null. List may return

\* now rows.

\*/

List<CandidateInfo> findCandidatesForAlias(String type, String alias);

Their behavior is quite simple. Given an asset, convert it into an alias, optionally using a locale. Secondly, given an asset type and an alias, get a list of all possible candidate asset ids that could have created that alias.

The reverse-mapping aspect of the aliasing system reflects the reality that many assets might have the same alias. The idea that each asset needs to have its own alias to be uniquely identifiable is not realistic. However, it is entirely reasonable to expect that the context in which an alias is used would be enough, when combined with an alias, to determine the exact asset ID corresponding to the alias.

The Site Plan Tree Helper uses this approach with the ItemAliasingStrategy to resolve asset Ids. That convention is not part of this interface.

The aliasing strategy employed is pluggable. This assembler ships with 5 aliasing strategies. It is believed that these 5 strategies will handle the vast majority of applications of the ItemContextAssembler. However, it is always possible to create custom aliasing strategies, and not very difficult to do so if required.

Some aliasing strategies require configuration while others do not.

The following section summarizes several shipping aliasing strategies.

### IdAliasingStrategy

This strategy simply converts the asset ID into an alias by calling toString() on the asset ID. Not practical for day-to-day use, but guaranteed to always work and helpful for testing.

### NameAliasingStrategy

This strategy simply uses the name of the asset as the alias. Guaranteed to create an alias, but the name field usually does not usually look good enough to use in a URL. In particular, spaces and capitals often cause problems.

### PathAliasingStrategy

This strategy uses the path field of an asset as its alias. Users must specify the value of the path field, so it is not guaranteed to work, but because it is handcrafted, it can result in very nice-looking URLs when used consistently. This is the simplest aliasing strategy that is acceptable for practical use. Complex multilingual sites may not be able to get away with something this simple and may have to use one of the following instead. However, sites with parallel site plan trees based on locale should be able to utilize this strategy.

### PageMetadataArticleAliasingStrategy

This strategy looks up a metadata article corresponding to a specific named association on the Page asset and uses its path field as the alias. This one is useful when site data is not being stored in the SitePlanTree but rather an attached asset instead. The name and type of the associated asset are configurable.

### MultilingualPageMetadataArticleAliasingStrategy

This strategy looks up a metadata article corresponding to a specific named association of the page asset, translates it into the specified locale, and uses the translated article’s path field as the alias. This strategy is useful for complex multilingual sites that have identical content but for which the content associated with the Page is translated. The name and type of the associated asset are configurable.

For information on configuring each aliasing strategy, consult each one’s Java API Reference documentation.

## Deployment Dependencies

This library depends on the cs-facades library being deployed alongside it. The cs-facades library can be found on DeveloperNet.