FatWire SOFTWARE

GST Site Foundation v1.0.1

Prepared by

FatWire Global Solutions Team

Table of Contents

Document Revision History

Date	Revision	Author	Notes
June 11, 2010	1.0	Tony Field Michael Sullivan Dolf Dijkstra	Initial version
June 15,2010	1.0.1	Dolf Dijkstra	Fixing typo, removing dash from meta-.

Executive Summary

This document exists to define the foundation of a simple URL-management and request handling/request dispatching model that most sites can easily extend and build upon. The initial emphasis is on a non-rendering foundation that at a high-level provides the following primary benefits:

- Introduction of the notion of a web-referenceable asset, access to which is via a common controller
- Support for vanity URLs for <u>any</u> web-referenceable asset
- Built-in 404 response for deleted assets
- Provides for managed alias assets that can represent any other URL
- Can be used "out of the box" for a large number of sites
- Can be cleanly extended to address custom client requirements
- Fast and efficient handling/dispatching of requests
- Fast and efficient construction of vanity URLs
- Standardized set of meta attributes
- Standardized usage of the Page asset for navigation

Additionally, this foundation addresses problems that many sites face: addressability of assets by editors, and acknowledgement that the URL is more important than the content. By giving each asset its own URL at the discretion of the editor, both problems are solved.

This is done in a way that accomplishes the following:

1. The process for business users is greatly simplified
2. Coding is greatly simplified, to the point where much can be provided in common libraries
3. Extension points are clearly identified

The following patterns and guidelines, if strictly followed, will enable architects to design basic sites more quickly, and will enable developers to spend more time implementing project-specific requirements rather than standard functionality.

A checklist is included that helps an architect to determine if this rendering model is right for the project at hand.

For More Information

For more information about the components described herein, for detailed technical documentation, configuration information, source and binary downloads, please visit the GST Site Foundation website at:

Configuration information, detail technical documentation, binaries, and source code can be found at the following URL:

http://www.nl.fatwire.com/dta/contrib/gst-foundation/

Glossary

This rendering model introduces some new terms and re-works the way we think about some terms that are already in use. This is done to enhance understanding, visualizing, and working with the components of the model.

These terms may not be perfect, but henceforth they are to be adopted and utilized regularly in order to ensure that this rendering model can be discussed efficiently. Furthermore, their use in design documents as standard terms is encouraged.

Term	Definition
Alias	An alias (“GSTAlias”) is an asset that acts as a proxy to another asset.
Controller	A component of a model-view-controller architecture that performs business logic.
c	Variable referring to the current asset type being requested.
cid	Variable referring to the id of the current asset type being requested.
GST*	“GST” is a prefix that is reserved for use by FatWire’s Global Solutions Team.
Nav Link	A NavLink (“GSTNavLink”) is a subtype of a Page asset that contains only a pointer to a web-referenceable asset.
Navbar Name	A navbar name (“GSTNavName”) is a page subtype designating a page asset as being a placeholder for a nav bar.
p	Variable referring to the ID of the Page asset that the input asset (c/cid) is associated with.
Page	The name of an asset type that ships with Content Server that is used in the site plan tree.
Site plan tree	A tree structure represented within Content Server as a hierarchy of “Page” assets. Pages are organized into this tree structure using the SitePlanTree table and can be manipulated with siteplan tags.
Tagging	To tag an asset is to associate the asset to a keyword. The keyword is used to locate common assets quickly. For instance all the News assets.
URL Path	A URL path is a part of a URL that does not include the protocol, host, or port. It does not need to specify the entire path component of a URL, and can therefore be appended to a URL ending with a path.
Virtual Webroot	A virtual webroot is a URL prefix including protocol, host, port, and possibly path elements. A URL path can be appended to a virtual webroot to create a complete URL.
Web-Referenceable Asset (a.k.a. WRA)	A web-referenceable asset is an asset that can be specified on a URL by representing c/cid either explicitly or implicitly.
Wireframe	A typed template, specifying 1… n web page components, like header, breadcrumb, navigation, body, footer.

Summary of Design Principles

Overview

There are several fundamental principles that underlie this rendering model:

- Simplicity
- Extensibility
- Separation of concerns
- Use of naming conventions
- Minimalism

Simplicity

This rendering model is designed for the 70-80% of sites that do not need sophisticated, programmatically generated web content.

The model is designed to consider SEO from the ground up. Some SEO aspects will be automatic (like prevention of unnecessary 404s, guessable links), while others are forced (required attributes that provide keywords etc.)

Some sites will not be able to utilize all of the patterns and conventions in this model.

Extensibility

This rendering model only provides the very beginning of what is required to build a site. It allows the asset types that are involved to be extended by an architect as needed.

Inner page rendering is not a central component of this design. There are several ways of doing this and they vary from site-to-site. Therefore at this point, there is no plan to address page composition with this rendering model, and this task is left to the architect.

Separation of concerns

URL and navigation are independent concepts. In this model, the Page asset is responsible for defining primary navigation only.

A Page subtype is used to classify page assets representing navigation hierarchies. A second Page subtype is used to classify each element of the navigation structure (nav bar).

URL-related information is removed from the Page asset entirely and is stored in each individual web-referenceable asset.

Page composition is the responsibility of the web-referenceable asset.

Use of naming conventions

Some new naming conventions are defined that allow these components to cooperate freely.

The following table summarizes reserved words

Reserved Word Type	Reserved Words
Page Subtypes	GSTNavLink GSTNavName
Attribute Names	metatitle metadescription metakeyword h1-title link-title name (OOB field) path (OOB field) template (OOB field) target target_url popup linktext linkimage master_vwebroot env_vwebroot env_name
Named Associations	target parent_web_page
Table Names	GSTUrlRegistry GSTTagRegistry
Variable Names	virtual-webroot url-path
Asset Type Names	The prefix `GST` is reserved for asset type names
System Property	com.fatwire.gst namespace is reserved.
SiteEntry Names	`ErrorHandling/IndexPage`

Using a reserved word is allowed as long as it is semantically consistent with the usage defined in this document.

Minimalism

One of the core principles of this rendering model is to not attempt to do too much; many pieces are left out. Tools could be built to simplify the user experience, to perform routine functions, etc.

The principles of the model are the priority. Tools to support the model will be developed, but are ancillary to the model.

Pre-requisites Checklist

The following checklist should be completed and all answers must be positive before this Site Foundation can be used on a given site.

Yes	Requirement
☐	Do editors want to control the URLs?
☐	Do you intend to use the Advanced User Interface?
☐	Is the website largely not a form-based application?
☐	Do you have access to Java developers?
☐	Are you creating a new site in Content Server for this project?
☐	Does your infrastructure support URL rewriting?
☐	Are you planning on a standard Content Server deployment?

Design Details

Controller use

The controller is the component of a model-view-controller architecture that performs business logic. A controller must be utilized on sites conforming to this model.

The controller **must** perform the following functions:

- Process bad requests properly (e.g.401s, 404, etc.)
- Resolve “c”, “cid”, from the incoming request
- Sets the “site” variable based on the input data
- Read the template associated with c/cid
- Call the template

The controller **may** perform the following functions:

- Security checks
- Form processing
- Other custom business logic

Notes on implementation

The underlying technology utilized by the controller can be specified by the architect. The FirstSite II Rendering Model implements its controller using a CSElement-SiteEntry pair called *Wrapper*.

A compliant controller can be downloaded from the website. Other implementations or extensions can be created as required.

Site Plan Structure

In its current form, the Page asset is slightly awkward. It either lacks fields or has too many. Multiple attempts have been made in the past at adding fields to a Page asset by associating rich content to the Page.

In this model, the opposite approach is taken. Rather than treating a Page asset as the core of a web page, the asset becomes extremely lightweight and is used only to organize navigation structures ("nav bars").

The Page asset currently serves three roles:

1. It is used to define navigational structure (it defines nav bars)
2. It is used to provide organizational input into URLs (it defines URL path information)
3. It is used for page composition (it contains associations to content that is rendered on a page)

Having one structure in Content Server serve three roles causes complexity and limits how problems can be solved.

The GST Site Foundation addresses this problem by clearly delineating the responsibilities, and designating them to specific components of the model. No component is responsible for more than one job.

Creating Nav Bars & Breadcrumbs

The Site Plan Tree structure can be used effectively to create dropdown nav bars. However it's rarely the case that a site's navigation exactly mimics the URL structure. To address this, the responsibility of specifying URLs is removed from the Page asset, leaving creating Nav Bars & Breadcrumbs as the primary responsibility.

Naming Nav Bars

We define a new subtype of the Page asset "`GSTNavName`", which is a placeholder asset. It must be placed at the root of the SitePlanTree under the publication name.

The only field of any consequence in the `GSTNavName` pages is the `name`. This field is used as a handle so that templates can access a nav bar by name (although `render:lookup` or another method entirely can also be used). Additionally, content editors can open a `GSTNavName` page to see the links in the nav bar that a site browser would see. This will assist them in locating and placing content.

A site can contain multiple `GSTNavName` pages.

Organizing Pages into Nav Bars

A new subtype of page representing links in a nav bar is also defined. `GSTNavLink`-subtyped pages are then placed under the Nav Name pages in the order in which they are to appear in that nav bar. Multiple levels of hierarchy are supported in order to allow for multi-level navigation.

It should be noted that on a rendered web page, the first tier and second tier of the navigation don't have to appear in the same block of HTML for them to both read data from the same data structure (i.e. both HTML blocks read the hierarchy under the same `GSTNavName`).

`GSTNavLinks` simply point directly to a web-referenceable asset using the unnamed association.

Breadcrumbs

Breadcrumbs correspond to the clue at the top of a page that usually indicates how a user clicked to access the specific point in the site that they are currently viewing. Consequently, they are usually built using the `GSTNavName`-`GSTNavAlias` data structure.

Not all pages can be easily attributed to the navigation bar, however. For this reason, several different ways of generating breadcrumbs are required.

Notes on implementation

An API and a tag are defined to assist with the construction of the breadcrumb. Because the algorithm utilized to identify the pages that constitute the breadcrumb may vary from site to site, the algorithm is pluggable.

See the appropriate appendix for information about the utilities and algorithms.

URL Specification

URLs are specified in this rendering model using the concept of a "Virtual Webroot" and a "URL Path".

Each web-referenceable asset must specify its own complete URL (hence the term "web-referenceable"). It is comprised of `<virtual webroot><url path>`. There is no limit on the number of virtual webroots that can be deployed on a server or in a site, but practical considerations may limit the number created.

The **complete URL** for the web-referenceable asset is stored in the `path` field. It is stored as a single string. The `<virtual webroot>` part of the URL will vary from deployment environment to deployment environment, so a URL assembler is required to substitute the value of the `<virtual webroot>` in the URL to ensure that the proper URL is served from the right environment. See the URL Assembler section below for details.

Business users can browse the content on their site using URLs. A custom tree view is provided to facilitate this. The interface is similar to Windows Explorer, and provides access to commands using contextual menus.

The `path` field of web-referenceable assets should be presented in a way that is very clear to end-users. A single textbox is probably not sufficiently usable. Rather, plain text with DHTML widgets could help. Validation and auto-completion could also be added to improve usability. Some examples of path field editors are described in the appendix. The exact set of features is undefined and the field editor should be tailored to the business requirements.

Asset Event on Save to Populate GSTUrlRegistry Table

An asset event is defined upon the SAVE event. Upon save, any asset with a `path` attribute (field) specified will have that field copied into the `GSTUrlRegistry` table, along with the asset's type, and ID.

The schema of the GSTUrlRegistry as defined as follows:

name	Type	Notes
path	varchar(<maxvarcharsize>) not null	note that longer URLs are officially supported, however since this component exists for SEO-optimized URLs, longer URLs will be an exceptionally rare occurrence
opt_vwebroot	varchar(255)	Virtual webroot part of

		path. Optimization.
opt_url_path	varchar(<maxvarcharsize>)	Part of path excluding virtual webroot. Optimization.
opt_depth	INT(10)	The depth of this url, calculated as the number of slashes in the url.
assettype	varchar(255) not null	
assetid	long not null	
startdate	Datestamp	the asset's start date
enddate	Datestamp	the asset's end date

The table has a unique constraint on path + startdate + enddate.

The table has an index on path, another on assettype, another on assetid, another on opt_vwebroot, and another on opt_url_path.

Asset Event on Delete to Update GSTUrlRegistry Table

A second asset event is defined upon DELETE. When an asset is deleted, its path field is cleared, and the GSTUrlRegistry is updated accordingly (the row is removed). This ensures that old URLs are made available to new assets.

Virtual Webroot substitution for different environments

Moving from one environment to another is problematic when URLs include hostname and port. Therefore, a substitution method is required to ensure that the virtual webroot can be substituted when rendering assets on different environments. URL Assemblers address this cleanly. For more details, see GSTVirtualWebroot Assets and URL Assembly below.

URL Assembly

This rendering model requires a URL assembler similar to the ItemContext assembler used with the FSII rendering model. The assembler, WRAPathAssembler (i.e. Web Referenceable Asset Path Assembler, short name "wrapath"), performs the following general functions

- It identifies URLs that it can process by locating the request parameter "assembler-name" with value set to "wrapath". This assembler does not decode URLs that do not include this parameter.
- It concatenates virtual-webroot and uri-path to form the complete URL.
- It suppresses the following parameters that must be in the Definition according to Content Server, but which are not required in the URL:
    - c
    - cid
    - pagename
    - childpagename
- Configuration is trivial.

The use of mod-rewrite or equivalent technology is ***required*** in order to populate the virtual-webroot property upon disassembly. See below for details.

This URL assembler is a required component for use with this rendering model.

Simplicity, usability and transparency are essential.

Mod-rewrite functionality and virtual webroots

Mod-rewrite functionality must be set up to support mapping URLs matching a local virtual webroot to Content Server.

Whenever a specific virtual webroot is matched in a URL by the web server, the request must be forwarded to ContentServer. In particular, the incoming URL must be re-written as follows:

- the assembler-name request parameter is added and set to "wrapath"
- the path component of the URL is replaced with the value of `path.SatelliteServer` property in `ServletRequest.properties`.
- the virtual webroot is appended to the URL as a parameter called `virtual-webroot`
- the url path is appended to the URL as a parameter called `url-path`

Mod-rewrite rules (or equivalents) can be created to accomplish this. For information about automated tools to help with this, see the appendix. Note that automation of mod-rewrite rules is not required but is convenient.

Controller integration

The WRAPathAssembler must work with a controller in order to support argument decoding and in order to allow `pagename` to be suppressed from the URL.

Once the assembler has confirmed that the request can be serviced (by locating `assembler-name` request parameter), the assembler sets "`pagename`" to the value of the "`com.fatwire.gst.foundation.url.wrapathassembler.dispatcher`" property. If the property is not specified, `pagename` is set to "`GSF/Dispatcher`". A valid SiteEntry must be found at one of these locations.

It should be noted that this design has the implication that the `pagename` set by the assembler is globally utilized across the whole ContentServer instance. Sites may share the SiteEntry corresponding to the dispatcher in order to ensure that preview functions properly. This will create extra SiteCatalog entries for each shared version of the dispatcher that should be used for Preview only.

If site-specific logic is required, this must be handled within the controller, not the global dispatcher/wrapper.

Decoder Helper in Controller

The controller component must re-constitute c/cid from virtual-webroot/url-path.

Because this rendering model explicitly specifies everything that goes into a URL, we can provide all of the required helper code out-of-the-box.

The controller may simply query the `GSTUrlRegistry` table for `assettype` and `assetid` (within the appropriate start date-enddate range) given the result of concatenating `virtual-webroot` and `url-path`. If a valid result is found, the controller sets `c` and `cid` to the values retrieved from the `GSTUrlRegistry` table. `Pagename` is set to the pagename of the Template asset set in the `template` field of the asset referred to by the `GSTUrlRegistry` row.

Finally, the pagename should be dispatched to using `ics.ReadPage()` `render:calltemplate`, or `render:contentserver` (NOT `render:satellitepage`), passing `c, cid`, and any other parameters that were present on the URL (besides `virtual-webroot` and `url-path`). This will invoke the template.

If no valid row is found in `GSTUrlRegistry`, a 404 response code should be streamed to the client.

Helper APIs and tags support this functionality.

render:get*url tag customization

In order that the links are generated properly and that the required information is passed to the URL assembler, the parameters `virtual-webroot` and `url-path` need to be calculated using the input `c` and `cid` for the web-referenceable asset. There is currently no supported plugin support for this in the Render framework (though 2 places allow it in unsupported ways).

The traditional workaround to this is to add extensive code to Link templates to look up these components (often with the use of a helper), then pass those parameters directly into the tag as arguments.

This is awkward, and it does not work at all with embedded links.

Recently, one of the two unsupported ways of achieving this customization has become a de-facto public API due to extensive use in the field. Consequently, this framework adopts it as well.

When a link is generated using any of the render:get*url or related tags, a set of APIs is invoked that allows the parameters of the link to be altered prior to the dispatch of the URL generation to the URL Assembler module. This is precisely the hook that we must extend.

In the `futuretense_xcel.ini` file is a property called `xcelerate.pageref`. This property is set to a class that implements the `com.openmarket.xcelerate.interfaces.IPageRef` interface. (Corresponding properties also control the implementation of `IBlobRef` in the same package, but that is out of scope of this component).

A new implementation will extend the default implementation (`com.openmarket.xcelerate.publish.PageRef`) and override the `setParameters(Map<String,String> args, ICS ics)` method. It will alter the input `args` map by adding the derived parameters

- `virtual-webroot`
- `url-path`
- `pagename`

to the map before calling `super.setParameters(args, ics)`. This trivial customization will allow proper links to be built by only specifying `c` and `cid` as input parameters.

The URL assembler will then take over and process these parameters, concatenating `virtual-webroot` and `url-path`, and adding any additional parameters as query string arguments (unless otherwise suppressed).

This approach enables the render:get*url tag family to always be able to generate "pretty URLs" for any web-referenceable asset without having to do any additional coding, customization of FCKEditor, or any other work.

Support for multiple URL assemblers

This rendering model supports the usage of the WRAPathAssembler alongside other URL Assemblers.

There are two key aspects that determine how well an assembler co-exists with others:

- it can be called explicitly
- it can recognize and decode its URLs and ignore foreign URLs

The first topic is simple. Links can specify their assembler using a short form. By specifying "`wrapath`", this assembler will always be called.

The second topic is more involved. If this URL assembler is registered in an order that allows it to handle requests for URLs that are not WRAPathAssembler URLs, it can make this determination by examining the parameters set in it by the mod_rewrite module. If the required parameters are absent, this assembler will abort disassembly and give the next assembler an opportunity to decode the URL.

A Note about Multilingual Sites

The site plan and url definition and assembly components of this rendering model are language-agnostic. The language of an asset is not important until the web-referenceable asset is rendered.

Two different URL paths are normally required for the "same article" rendered in 2 different languages. In this case, each translation of the web-referenceable content should literally just specify their URLs in their `path` fields.

GST Flex Family

A new flex family will be created to support core infrastructure for this rendering model**. It should not be used as the main flex family for implementing sites**. A site should have its own flex family. This effectively reserves the flex family for future definition and future conventions to be defined by FatWire's Global Solutions Team.

Flex family asset types

Role	Name	Description
Attribute	GSTAttribute	GST Attribute
Flex Child	GSTAlias	Alias
Flex Child	GSTVirtualWebroot	Virtual Webroot
Flex Definition	GSTDefinition	GST Definition
Flex Parent (future use)	GSTParent	GST Parent
Flex Parent Definition (future use)	GSTPDefinition	GST Parent Definition
Flex Filter (future use)	GSTFilter	GST Flex Filter

GSTVirtualWebroot Assets

It is necessary to define an asset type designating a virtual webroot configuration asset to allow the URL assembler to specify the virtual webroot for a given environment.

Because all URLs are prefixed with the webroot information, the URLs are fully qualified. Fully qualified URLs are host-specific. Content Server sites must be able to be rendered on different environments, such as staging, qa, delivery, and dev. For this to occur, fully qualified URLs must be altered to make them refer to the proper hostname.

GSTVirtualWebroot assets are configuration assets that enable this.

The asset includes three key attributes: `master_vwebroot`, `env_vwebroot`, and `env_name`. The `gst_master_vwebroot` attribute corresponds to the virtual webroot present in the URLs on the web-referenceable assets. In fact, the path attribute editor can validate against instances of the GSTVirtualWebroot assets.

The `env_name` attribute specifies the name of the current Content Server environment. The environment is configured as a system property.

The `env_vwebroot` attribute specifies the webroot value to use as a prefix on the local environment.

Examples of `env_vwebroot` values include

- http://www.fatwire.com
- http://es.fatwire.com
- http://uk.fatwire.com
- http://www.fidelity.ca/en
- http://www.fidelity.ca/fr
- https://www.fatwire.com/my-flashy-new-microsite

Code will look up the system property name, and then look up the appropriate virtual webroots. It can then substitute the master webroot with the local webroot to produce an absolute URL to the current host.

System Property Name

Property Name	Purpose	Default
com.fatwire.gst.foundation.env-name	To define the current environment name to use	None. Absence disables the functionality.

Definition

Attribute Name	Attribute Description	Purpose
master_vwebroot	Master Virtual Webroot	Holds the virtual webroot value utilized in path elements of the web-referenceable asset
env_vwebroot	Environment-specific Virtual Webroot	Holds the value to use for the virtual webroot on the specified environment
env_name	Environment Name	Name of the environment (e.g. staging, production, dev-main, qa, etc.)

GSTAlias Assets

A GSTAlias asset is a shortcut to another asset. It allows an asset to be referenced in a place on the site other than its primary "home".

Aliases should not be used as a primary means of creating internal links throughout a site (the assets themselves can do that most of the time).

Aliases can override some values of the asset that they point to. For example, an alias can have its own URL, even though it references content that may or may not already have a URL defined for it.

Having multiple URLs for the same asset is possible; however it may not be desirable when considered in the context of SEO. The architect is advised to consider this.

If not assigned a URL, however, an alias acts like a proxy, or pointer asset, to any other asset on the site. This is typically done to create "link"-like behavior.

Thus, an alias is defined with a flexible definition that allows the architect to tailor it to the site's needs. For example, a `link-image` attribute can be defined so that the alias is rendered as an image link. A `target_url` attribute can be defined to point to external sites. In this way, the alias asset can be used to represent an external link.

Ordinarily, when an alias asset renders its target, it will render it using the information provided in the target. Namely, the alias will look up the target's wireframe template and render the target in that wireframe. However, this behavior can be overridden by specifying an overriding wireframe template for the alias.

The following attribute names are **reserved but optional** in the alias template. If a use case matches the purpose, then architects are encouraged to utilize the reserved attribute name:

Attribute Name	Attribute Description	Purpose
target	Target	Attribute of type asset corresponding to the target of the alias.
target_url	Target URL	Attribute of type String defining an externally-linkable URL that the alias refers to
popup	Display Popup	Attribute of type int (values of 0 or 1) indicating whether or not the alias will be opened in a popup

template	Template	Out-of-the-box field (not attribute) specifying the wireframe template to be used to render the target asset. Can be left blank to use the target's wireframe.
linktext	Link Text	Attribute of type string designating the text to be used to represent the link
linkimage	Link Image	Image to be specified for the link
path	Path	The URL used to override the URL of the target asset (if required).

This asset type will replace the oft created "Link", "CustomLink", "ExternalLink", and similar asset types. In addition, it will replace the "Proxy", "Shortcut", "Bookmark", "Symbolic Link" types often created for other reasons.

Web-Referenceable Assets & Pages

Deficiencies in the page asset prevent it from being used to directly represent the raw content that comprises regular web pages. (Attributes are missing, multiple-value support is not present, it cannot be assigned definitions, etc.).

For this reason, **the Page asset is no longer to be used to define the main content that appears on a web page**. Instead, assets that represent web pages, and which can in turn be directly referenced by a URL are defined as "Web-Referenceable Assets", and are responsible for this. Web-referenceable assets can be either basic assets or flex assets.

These assets must have certain pre-defined basic asset fields **or** flex attributes in order to be referenceable. They include:

Attribute	Description
metatitle	The <title> and meta name="title" fields. String. Required.
metadescription	Corresponds to the meta name="description" field. String. Required
metakeyword	Corresponds to the meta name="keyword" field. Comma-separated string. Optional.
h1title	Corresponds to the field on the page that will be rendered with a h1 tag. Required.
linktitle	Optional field designating the text to be displayed when rendering a link to the asset. When not specified, h1-title is used.
path	URL Path for the asset. Not specifying it mean the asset is not web-referenceable.
template	Standard field designating the wireframe to use to render this asset when it is being displayed by itself.

As noted above, the path attribute must be set for the asset to be considered web-referenceable. Without it, not pretty URL is generated for the asset.

It should also be noted that required fields must remain required for the provided code to function properly. However, flex filters may be configured to automatically populate some fields if desired.

Rendering the content of a Web-Referenceable Asset

Because web-referenceable assets do not specify a type or definition (only a set of common attributes) they can be of any type. Therefore, any asset definition that needs to be rendered can have its attributes added alongside the required attributes to populate the main content for the page.

Therefore, it is very easy to build a template that renders content of one of these assets – the fields are all local. In fact, no special practices are defined. Architects are free to specify the definition of their web-referenceable asset as they see fit.

It should be noted that a web-referenceable asset might point directly to other assets for the purpose of composing contents of the page. (e.g. modules, lead images, collections of articles).

Asset Tagging

Background

Most web-referenceable assets contain some sort of data structure to build an explicit linking path to the assets that will be displayed on them. Because of this, there is no need to do perform a query that could trigger an unknown dependency.

For example, if a web-referenceable asset represents "Editor's Picks", it likely has a recommendation asset associated to it that in turn links to individual articles. The individual articles end up being identified through explicit relationships, and there is no need to perform an unconstrained query to identify them.

Portal-like web pages, however, operate differently. Content "chooses" where it is to appear and it is up to the template for the portal page to ensure that the content magically appears.

This presents a problem normally for Content Server architects. The reason for this is that for a portal-like page to look up all assets that point to it, an unknown dependency ends up being recorded in the portal page's template to perform the reverse-lookup to find the children.

This section describes how a portal-like page can efficiently identify what content has chosen it, without triggering an unknown dependency.

Solution

Wiring

In order to find and group assets together based on keywords we propose a simple tagging mechanism. An editor tags an asset with one or more values. This tag value drives the placement of the asset on the site. For instance an editor tags the asset with 'news' and this asset will show up in the 'news' section. By allowing for multiple tags the asset can show up in multiple places on the web site. These places are usually listing pages, listing all the assets for the news section, or all the assets that have a tag in the form of another asset: `asset-1234:MyAsset`.

The implementation of the lookup is done in a single (global) transient lookup table. The lookup table, similar to the GSTUrlRegistry table, is populated when an asset is saved or published, can be queried at runtime and has a link to cache invalidation code.

The keyword is provided by the editor in the GSTTag attribute, or field for basic assets. This is a single-valued string attribute that is in the form of a comma-separated list.

`Note: though this table is now implemented as a database table, there is no reason why this could not be implemented as a Lucene index. We propose a database table for maintenance convenience.`

Accessing

To retrieve this content, what is needed is the lookup over the tag name. In fact, a new JSP tag is defined called `<gsf:tagged-list />`. This JSP tag takes a `tagname` as the input and returns a list of children. It is defined in detail below.

The JSP tag does NOT record any unknown dependency; however it does record a dependency on the tagname and the children.

Cache Flushing

To ensure that the addition of newly tagged content results in the content appearing on the page, the lookup page must be flushed whenever a tagged asset with the same tagname is added.

Content Server does not have enough built-in intelligence to determine this consistently. However, a publishing hook can help. Whenever publishing occurs, we can query all of the newly published assets that are "tagged". We can then add the tags to the list of compositional dependencies that "have been published", and trigger a flush of those pagelets too. This has the effect of re-building the page only when it will render different output, and it is much more efficient than an unknown dependency.

This functionality needs to be hooked up carefully. For delivery environments, it must only be hooked up to the publishing event. However, for pages to accurately be updated on staging, dev, and qa systems, the behavior described above must ALSO be triggered on the ASSET.SAVE event for the child asset.

The implementation of each of these is defined below. (see RealTime CacheUpdater Plugin and Asset Event on Save to Flush Parents)

<gsf:tagged-list/> Tag

This tag uses ICS.SQL(PreparedStmt, boolean) to query the GSTTagRegistry and retrieve the assets that point to the specified tag.

Input

- tagname – the name of the tag
- outlist – name of output list

Output

- The name of an IList object to be placed in the list pool. It contains two columns: ASSETTYPE, ASSETID.

Null is never returned, but the returned list can be empty.
A java method is provided in order for the same logic to be called from java.

Asset Event on Save to Populate GSTTagRegistry Table

An asset event is defined upon the SAVE event. Upon save, any asset with a `GSTTag` attribute (field) specified will have that field copied into the GSTTagRegistry table, along with the asset's type, and ID. As the GSTTag field/attribute supports multiple values, multiple rows can be inserted in the this table for one asset.

The schema of the GSTUrlRegistry as defined as follows:

Name	Type	Notes
Tag	varchar(255) not null	Name of the tag,
Assettype	varchar(255) not null	
Assetid	long not null	
Startdate	Datestamp	the asset's start date
Enddate	Datestamp	the asset's end date

RealTime CacheUpdater Plugin

RealTime publishing includes an API entitled RealTime CacheUpdater. We will override the default `com.fatwire.realtime.PageCacheUpdaterImpl` to override the `beforeSelect()` method. The flush and regen keys will be extended, to automatically include all pagelets containing the GSTTag attribute value. This ensures that even though parent has not changed, pagelets that reference it are automatically flushed. This ensures that by simply tagging an asset, it automatically and instantly appears in pages that render it.

The GSTTagRegistry table is read for the specific assets before the new values are inserted. This is to make sure that pagelets are also flushed with the 'old' tag values for the cases where the tag is deleted or the values have changed.

Implementation examples can be found in the guide "Customizing RealTime Publishing Cache Management".

Asset Event on save to tag referenced pagelets

When an asset is saved, this event checks to see if the `GSTTag attribute` is defined. If it is, it looks up the old value in the GSTTagRegistry table for the current asset, updates the GSTTagRegistry table for the current values and flushes the page cache for all the old and new tag values, using the `CacheManager` API.

This is only done if an environment is not a delivery environment. On delivery the asset listener should not be configured, only the publish event listener is. Both listeners share, to some extent, the some algorithms. By configuring different listeners in different environments we can fine tune the execution order.

Index Pages

Index pages are not actual assets, and they are not content assets either. Index pages are the web pages that appear when a visitor accesses an *invalid* URL that has child URLs below it that *are* valid. Typically, this looks like a directory listing.

In a sense, index pages are “pretty 404s” that have been elevated above the status of an error (and thus have a response code of 200).

If index pages are required, the following describes how they should be handled according to this rendering model.

If a 404 should be returned:

When the controller determines that no match was found for a given URL, the controller sets an error code header corresponding to the 404.

A servlet filter then reads the header and returns a 404 status code.

The servlet container is then responsible for calling the appropriate error handler page as defined in the web.xml file for the servlet. It should be noted that Content Server can be used to handle 404 requests by specifying a Content Server page in the 404 error handle block in web.xml.

If a 200 should be returned:

If the controller determines that a 200 status code should be streamed, then it should dispatch the request to a CSElement/SiteEntry pair called ErrorHandling/IndexPage that renders the index page content. Its input is the virtual-webroot and url-path variables processed by the controller.

Consult the appendix for a detailed algorithm and tools to assist with the automatic generation of index pages.

Landing Pages

Landing Pages are pages that typically are entry points into the website. They may be advertized on television or in a newspaper (by URL of course), or they may be sent out as links in email. They may also simply be the primary links off a nav bar.

In any event, landing pages practically always are designed to be attractive and provide links to other assets.

There is fundamentally no difference between a landing page and any other web-referenceable asset in this rendering model.

A Landing Page is not an Index Page, nor is it related to an index page.

A proxy or alias asset might act as a landing page for the target content.

Landing pages typically render content from various different sources though, which is probably not the case for all web-referenceable assets. This has no impact on the model, however.

Wireframes

Wireframes are typed or typeless templates. They combine the role of Layout and Detail as defined in FirstSiteII. Therefore, the entire markup that drives a page is defined in the wireframe template.

While it is true that calls to some common components like TopNav, Footer, etc. will be duplicated in each wireframe; the specification of these components has become so simple that the duplication is more than justified by the design simplicity. Preview is also simplified for business users as a result of this approach.

For example, skinning sites with most of the wireframe specified in a single template is very simple.

The wireframe may of course call out to nested site entries or templates to render common components, as well as of course other related components.

It should be noted that wireframes are compatible with modular-based designation of common page elements.

Ideas for the future

The following section describes some ideas for the future of this site foundation. They should not be considered well-developed and are not intended to define a specification, but are rather to indicate where we see the site foundation going in the future, as at the time of writing.

Page Asset UI Customization

At some point it probably makes sense to create custom forms for the Page asset's UI so that each subtype of Page asset can have its own custom form that only renders the required fields.

Since Pages are basic assets, and have their own ContentForm and ContentDetails forms, this customization should be simple.

The benefit of this customization is to emphasize that the page asset is to remain extremely lightweight as opposed to becoming something bloated with lots of extra content in it.

Tools

Any tools that will be created must be simple, compatible, and robust. It should not be expected that required tools for this rendering model will be updated on a regular basis. The tools should ship with a version number and the specification should correspond to that version number.

This document will describe the exact details of the version numbers of the jars required to support the ideas set out herein.

Postupdate action on asset delete to create an alias

When a web-referenceable asset is deleted, it could be very convenient to convert it to an alias in some cases.

Make use of IBlobRef for a pretty BlobServer URL

Pretty BlobServer URLs can easily be generated using the IBlobRef interface in the same way that the IPageRef interface is being leveraged.   This should be investigated.

301 support for GSTAlias assets

It is conceivable that GSTAlias assets will be required to represent “permanent redirects” or 301 status codes. This may require a new field and some intelligent logic to support it.

AddLink customization preventing linking to non-WRAssets

A customization to the addLink UIs to prevent people from linking to assets that don’t have path attributes set would help prevent broken or ugly links.

The rendering model specifies the requirement. A tool like this improves the user experience in attempting to achieve what is set out by the business requirements.