COMPUTECH

WTB Dynamic Website System Design Specifications

Version 1.0

January 19, 2006

Prepared by:
Computech
12th Floor
7735 Old Georgetown Road
Bethesda, MD 20814
(301) 656-4030
www.computechinc.com

Current Document Status

Version Number	1.0
File Name	WTB_Dynamic_Site_SDS.doc
Delivery Date	1/13/06
Owner	
Description	System design specifications for the dynamic version of the Federal Communications Commission (FCC or Commission) Wireless Telecommunications Bureau (WTB or Bureau) website

Document Revision History

Version Number	Date Of Change	Changed By	Revision Description
1.0	1/13/2006		Initial version.

Amendments

Amendments to this document are issued as needed. If you would like to suggest an amendment, please send an email to the document owner identified above.

Table of Contents

List of Tables

List of Figures

1. Introduction

1.1. Purpose

The Wireless Telecommunications Bureau (the "Bureau") operates a large internet website (http://wireless.fcc.gov), used primarily to disseminate information to the public and interface with the Bureau's online systems. The public reasonably expects the site to provide comprehensive, accurate, and timely information about Bureau activities of interest to them. To date, the website has satisfied these objectives; however, as the website matures, an opportunity presents itself to better serve the users with more reliable, scalable, and flexible technologies supporting the delivery of information.

The purpose of the WTB Dynamic Website System Design Specifications guide is to document the creation and continued maintenance of this Dynamic Wireless website as well as providing suggestions for possible future improvements to the website. It contains information regarding the creation and implementation of the system.

1.2. Scope

The Dynamic Wireless website will feature reusable, sharable content; user-focused navigation\automated content management processes; standards-based development; reduction in network load by drastically reducing page weight (by at least 50%); seamless support of any and all web browsers; and far greater accessibility-compliance.

1.3. Background

The Wireless website has grown exponentially as the business relevance of the website has become widespread across the organization. In fact, the site has almost doubled over the last two years. It is anticipated that the site will only continue to grow in size and scope

It is important to understand the characteristics of the website today as the Bureau positions itself for the future. The existing website almost exclusively embeds business information with display code; this is commonly referred to as "static HTML". This remains a common practice held over from the early days of web development. This method of delivering information was useful for posting simple information quickly on the internet. As the internet evolved, it became more relevant as a medium for conducting business. Unfortunately, many organizations that rushed to establish an online presence failed to revisit their strategies for managing the web as a business tool

The key to transforming the site from a series of static pages is the separation of content from display. The content, or information on the page, can be more concisely managed by isolating it from the display, or how that page is formatted and appears to the public. The separated content

can be easily reused and repurposed, and is portable across multiple platforms. This positions the Bureau to be able to react to new technologies, change, and deliver content on multiple platforms, without significant increase in existing resources.

1.4. References

• Avencom, Avencom Accessibility 508 Compliance site www.508help.org
• The World Wide Web Consortium (W3C) Accessibility Check list www.w3.org

1.5. Document Overview

This document includes the following sections:

- Section 1 describes the purpose and scope of the WTB Dynamic Website System Design specifications.
- Section 2 describes the framework of the WTB Dynamic Website.
- Section 3 describes technical environments of the Website.
- Section 4 describes how a request is processed by the system.
- Section 5 describes administration of the Dynamic Website, including production, indexing and caching issues.
- Section 6 describes the ISAS data updates
- Section 7 describes the configuration files for the Website.
- Section 8 describes the relevant ColdFusion components and usage.
- Section 9 offers suggestions for future improvements to the Website.
- Section 10 outlines the accessibility benefits of the Dynamic Website
- Section 11 describes the RSS Update process.

2. Website Framework

The Dynamic Website is conceived as a core framework, and subsites built into/onto the framework. The framework is an implementation of the MVC pattern.

The framework of the Dynamic Website is comprised of numerous CFC's and XML configuration files.

2.1. ColdFusion Components

The ColdFusion components (CFC's) of the framework are stored in the root of the dynamic site application's directory in the following directory structure:

[cfcomponents]

|----[exception]

|----[job] (MODEL)

|----[util]

|----[view] (VIEW)

|----DynamicWebsiteController.cfc (CONTROLLER)

When instantiating CFC's the CreateObject function is used, and paths to CFCs are expressed with dot notation, for example

```
<cfset utilities = CreateObject("component", "cfcomponents.util.Utilities">
```

2.2. Configuration Files

The configuration files are of two kinds. One kind (configuration.xml by convention) provides settings that control various aspects of the entire framework and of specific sub-sites. Another kind (<subsite>_navigation.xml by convention) defines the hierarchical navigational structure of a sub-site and is used to build navigational elements such as the left nav and breadcrumbs.

There is a global configuration.xml file and then each sub-site defines it's own.

[configuration]

|----configuration.xml (Settings for things like caching, error handling, locating sub-site configuration files etc…)

|----[auctions]

|---- configuration.xml

|----[services]

|---- configuration.xml

…

3. Environments

3.1. Environments

Table 1 Environments

Server	Root Directory	Access	Base URL
Chlorine (DEV)	/opt/bea8.1/WebSiteDomain/websitestage/websitecfmx/websitecfmxapp	Write	chlorine.fcc.gov:4020/ The webserver riga is configured to proxy *.htm requests to chlorine:4020 – for example: http://riga.fcc.gov/auctions/default.htm
Hegre (Daily Content Updates QA)	/opt/bea8.1/WebSiteDomain2/websitestage/websitecfmx/websitecfmxapp	Write	hegre.fcc.gov:4021/
Hegre & ramn (Major Code Updates QA)	/opt/bea8.1/WebSiteDomain/websitestage/websitecfmx/websitecfmxapp	No Access (Email A-Unix distribution list with detailed "copy from… to" instructions when updates need to be made)	hegre.fcc.gov:4020/ The webserver heidrun is configured to proxy *.htm requests to hegre:4020 and ramn:4020 – for example: http:// heidrun.fcc.gov/auctions/default.htm
Huldra & Ula (Production Code and Content)	/opt/bea8.1/WebSiteDomain/websitestage/websitecfmx/websitecfmxapp	No Access (Email A-Unix distribution list with detailed "copy from… to" instructions when updates need to be made. Major updates can only take place outside regular business hours and need to be coordinated ahead of time)	Huldra.fcc.gov:4020 and ula.fcc.gov:4020 The webserver grane (which serves wireless.fcc.gov) is configured to proxy *.htm requests to huldra:4020 and ula:4020, for example: http://wireless.fcc.gov/auctions/default.htm

4. How a Request is Processed

4.1. High Level Overview

Any request for a page includes a job parameter in the URL. This is key and is used to look up all required values.

Table 2 High Level Flow

Event	Details
1. Request is made. Url includes "job=" param	Example: http://wireless.fcc.gov/auctions/default.htm?**job=auction_summary**&id=60
2. Three SESSION scoped components are instantiated if not already so	Application.cfm contains code to instantiate the following components which are used frequently in the service of a request: **SESSION.xmlFactory (cfcomponents.XmlFactory.cfc)** Used to hide details of parsing an XML file into a ColdFusion Struct **SESSION.configuration (cfcomponents.Configuration.cfc)** Provides an interface for getting configuration settings which are stored in XML documents. For the sake of loose coupling, the framework makes much use of configuration documents. There is a central one, /configuration.configuration.xml which contains global settings including the locations of sub-site specific configuration files. These files are stored under a sub-site's root, for example /auctions/configuration/configuration.xml The CFC exposes specialized and frequently used methods such as getHost(), cachingIsEnabled(), getDeploymentEnvironment(), debugModeIsOn(). It also exposes a method used to search for a configuration setting in a specific configuration document. This method's signature is searchConfiguration(subsite, xPathExpression). The first argument is used to locate the appropriate configuration file to search in, and the second argument is an XPath expression which queries the XML for a specific node or nodes. Examples: `<cfset administratorsNode = SESSION.configuration.searchConfiguration("main", "//administrator")>` Gets a collection of all <administrator> nodes from the main configuration file (/configuration.configuration.xml)

	<cfset jobNode = SESSION.configuration.searchConfiguration("auctions", "//job[@name='#jobName#']")> Gets a collection of <job> nodes whose name attribute equals the value of jobName. **SESSION.cacheManager (cfcomponents.util.CacheManager.cfc)** Handles all the details of managing cache. The cache is currently implemented as an APPLICATION scoped Struct whose keys are query strings and whose values are the HTML for entire pages. Each time a page is requested the query string is used to check this Struct for the existence of a stored page. If found, the HTML is output from cache, otherwise processing continues to create the HTML and then store it in the Struct. (There is an admin tool which flushes specific pages or all pages from cache)

3. The sub-site's main file delegates processing to the central controller	The framework's central controller is a CFC: It is the **CONTROLLER in MVC pattern**.
	`cfcomponents.DynamicWebsiteController.cfc`
	It exposes a single public method, serviceRequest().
	It is created and invoked like this:
	`controller = CreateObject("component", "cfcomponents.DynamicWebsiteController");`
	`controller.serviceRequest("auctions", "default.htm");`
	The arguments passed to the serviceRequest() method are1) the sub-site name and 2) the name of the calling file.

4. DynamicWebsite-Controller creates a JobFactory instance and uses it to create a XXXJob CFC instance	The notion of a "Job" is central to the framework. It is modelled in XXXJob.cfc's stored in the cfcomponents.job package.
	These components are the **MODEL of the MVC pattern**, handling the details of making connection to data, and setting various parameters in the REQUEST scope which are used throughout the framework. You can think of them as setting up everything that will be needed as the request is page is built
	In DynamicWebsiteController:
	`jobFactory = CreateObject("component", "cfcomponents.job.JobFactory");`
	`job = jobFactory.createJob(URL.job);`
	In JobFactory
	`function createJob(jobName){`
	`jobNode = SESSION.configuration.searchConfiguration(REQUEST.subsite, "//job[@name='#jobName#']");`
	`if(ArrayLen(jobNode)){`
	`cfcName = jobNode[1].XmlAttributes["cfc"];// Example <job ...`

```
cfc="cfcomponents.job.ServiceJob"/>
      job = CreateObject("component", cfcName);
      job.init(jobNode);
      return job;
   }
   else {
      throwJobNotFoundException(jobName);
   }
}
```

JobFactory uses the job= parameter from the URL to look up a <job> XML element in the subsite's configuration XML.

For example /auctions/configuration/configuration.xml defines this among other jobs:

```
<job name="auction_summary"
   xml="/auctions/xml/auctions/auction_id_[ID].xml"
   xsl="/auctions/[XSL_DIR]/auction_index.xsl"
   page_layout="auctions"
   cfc="cfcomponents.job.Job"
   output_method="transform"
/>
```

Quite a lot is configured here. If you get the hang of all of this you'll be well on your way to getting how the whole framework functions.

All possible <job> attributes:

- **name:** matches the job= param that a calling url would include
- **xml:** tells the framework where to find the XML data it needs to fulfill the request
- **xsl:** tells the framework where to find the XML data it needs to fulfill the request
- **code:** if output_method="run_code", tells the framework where to get the ColdFusion code to run
- **page_layout**: tells the framework which page_layout it should use to build the response page (more on this below)
- **cfc**: tells the framework which XXXJob.cfc. it should instantiate to fulfill the request (the JobFactory.createJob() method uses it – see code snippet above).
- **navigation_xml:** tells the framework where to find the XML which defines the navigation for this subsite.
- **output_method**: tells the framework how the response HTML is to be generated – currently either transform (meaning XML/XSLT), run_code or static_file
- **cachable**: tells the framework if this page can be stored in cache or not. If not specified, default is true.

<page_layout>

As seen, a <job> has a page_layout attribute.

Page layouts are defined in the main configuration file (/configuration/configuration.xml). Here's an example:

```
<page_layout id="help">
   <page_element id="head"
cfc="cfcomponents.view.help.PageElementHead"/>
   <page_element id="masthead"
cfc="cfcomponents.view.PageElementMastHead"/>
   <page_element id="test"
cfc="cfcomponents.view.help.PageElementHelpNavAndContent"/>
</page_layout>
```

So, a Page Layout is comprised of a set of PageElementXXX's (CFCs) in a specific order. More on this later.

The "Base" Page Layout

A "base" page_element defined in the main configuration file. If a new page layout has to be created and it only differs minimally from the base one, then it can override the base page element. The code handling this is in the JobViewer CFC.

Here's the base <page_layout>

```
<page_layout id="base">
   <page_element id="head" cfc="cfcomponents.view.PageElementHead"/>
   <page_element id="masthead"
cfc="cfcomponents.view.PageElementMastHead"/>
   <page_element id="titlebar"
cfc="cfcomponents.view.PageElementTitleBar"/>
   <!-- The left column page element is made up of sub elements -->
   <page_element id="leftcolumn"
cfc="cfcomponents.view.PageElementLeftColumn">
<page_element id="search" cfc="cfcomponents.view.PageElementSearchBox"/>
<page_element id="mainnav" cfc="cfcomponents.view.PageElementMainNav"/>
<page_element id="relatedsites"
cfc="cfcomponents.view.PageElementRelatedSites"/>
   </page_element>
   <page_element id="breadcrumbs"
cfc="cfcomponents.view.PageElementBreadCrumbs"/>
   <page_element id="main_content"
cfc="cfcomponents.view.PageElementMainContent"/>
   <page_element id="footer" cfc="cfcomponents.view.PageElementFooter"/>
</page_layout>
```

And here's one that extends it:

```
      <page_layout id="services" extends_base="true">
```

	<page_element id="breadcrumbs" cfc="cfcomponents.view.PageElementBreadCrumbsService"/> </page_layout> So in the case of the services page layout, all the page elements of the base layout will be used, but the page element with an id of breadcrumbs will be overridden resulting in the use of a different CFC.
5. DynamicWebsite-Controller calls the job's processJob() method	This method grabs values from URL params, and from attributes of the <job> from configuration to populate the following REQUEST scope variables. Subclasses of Job.cfc override certain default values to set values specific to certain sites. REQUEST.pageIdentifier Uniquely identifies a page (it's set to the value of QUERY_STRING) REQUEST.titleBarText Displays in the page title bar REQUEST.pageTitlePrefix Used to prefix the content of the <title> tag REQUEST.job Unique identifier for the current request Passed as param to XSLT) REQUEST.id Can be the id of an auction, a release, a service etc...(Passed as param to XSLT) REQUEST.page Can be name or number of page Passed as param to XSLT) includes any qualifier REQUEST.page_qualifier_stripped Can be name or number of page Passed as param to XSLT) excludes any qualifier REQUEST.m Month (default is current month) Passed as param to XSLT) REQUEST.y Year (default is current year) Passed as param to XSLT) REQUEST.ry Another Year (used by some XSLT) Passed as param to XSLT) REQUEST.mode Currently used to change the xsl directory to N4 versions REQUEST.xmlData REQUEST.xslForTransform REQUEST.staticFileLocation REQUEST.codeLocation REQUEST.navigationXml REQUEST.outputMethod REQUEST.pageLayout REQUEST.jobIsCachable
6. DynamicWeb	JobViewer and the various PageElementXXX's are the **VIEW component in the MVC pattern**. They belong to the cfcomponents.view package.

site-Controller creates a JobViewer and calls its viewJob() method	JobViewer uses the <page_layout> element to loop though a set of <page_element> nodes, using the cfc attribute of each one to instantiate the appropriate CFC and call its viewPageElement() method.
	Here's where it happens:
	`subsitePageElements = SESSION.configuration.getPageElementsNodeSet(REQUEST.pageLayout);`
	`subsitePageElementsFound = ArrayLen(subsitePageElements);`
	`for (i = 1; i LTE subsitePageElementsFound; i=i+1){`
	`pageElementNode = subsitePageElements[i];`
	`pageElementComponentName = pageElementNode.XmlAttributes["cfc"];`
	`pageElement = CreateObject("component", pageElementComponentName);`
	`pageElement.viewPageElement(pageElementNode);`
	`}`
	Every PageElementXXX exposes a viewPageElement() method, the details of which vary widely. If this were true OO, then there'd be a PageElement interface, forcing implementing classes to implement this method. As it is the developer just has to be sure to do it right!
	JobViewer also handles the logic required when a <page_layout> extends the base layout.

7. PageElement MainNav	Each Dynamic Site sub-site (auctions is currently the only one in production) has an XML file that defines the navigation for that subsite. At a high level the XML is comprised thus:
	`...`
	`<page job="" label="">`
	`<page job="" label=""/>`
	`<page job="" label="">`
	`<page job="" label=""/>`
	`</page>`
	`</page>`
	`...`
	So the navigation XML is made up of `<page>` elements nested in such a way as to represent the hierarchical structure of the sub-site.
	The location of this navigation XML file is specified in the navigation_xml attribute of a <job>.

8. PageElement BreadCrumbs (and subclasses)	XXXPageElementBreadCrumbs components actually delegate the complexities of building the breadcrumbs to another component called BreadCrumbsBuilder. It happens like this:
	`<cfcomponent displayname="PageElementBreadCrumbs" extends="AbstractPageElement">`
	`<cffunction name="viewPageElement">`
	`<cfscript>`
	`breadCrumbsBuilder = CreateObject("component", "cfcomponents.view.BreadCrumbsBuilder");`

	<pre>breadCrumbsBuilder.buildBreadCrumbs(**THIS**, true);
	  </cfscript>
	  </cffunction>
	  <cffunction name="buildQueryString">
	  <!---
	  Stuff goes here that is specific to needs of a particular subsite.
	  --->
	  </cffunction>
	</cfcomponent></pre>
	In this method call: breadCrumbsBuilder.buildBreadCrumbs(**THIS**, true); the first argument passed in is THIS – a reference to the PageElementBreadCrumbs component itself. The buildQueryString() method of PageElementBreadCrumbs is a callback method – the buildBreadCrumbs()method of BreadCrumbsBuilder calls it to create links which follow rules specific to particular sub-sites. This approach was taken so that the central functionality which creates breadcrumbs (and which is common to all sub-sites) could be coded in a re-useable way.
	Breadcrumbs are generated from the same navigation XML as the main navigation.

9. PageElement MainContent	What happens here depends upon the output_method attribute of the <job> (in the sub-site's configuration.xml file).
	This CFC tries to get the required page from cache first and if not found generates it in the following code:
	<pre>if(REQUEST.outputMethod EQ "transform" or REQUEST.outputMethod EQ "transform_to_xml"){
	  transformer = CreateObject("component", "cfcomponents.util.Transformer");
	  REQUEST.htmlToOutput = transformer.transformXML();
	}
	else if(REQUEST.outputMethod EQ "run_code"){
	  includeTemplate(REQUEST.codeLocation);
	}
	else if(REQUEST.outputMethod EQ "static_file"){
	  fileToRead=REQUEST.staticFileLocation;
	  fileReader = CreateObject("component", "cfcomponents.util.FileReader");
	  REQUEST.htmlToOutput = fileReader.readFile(fileToRead);
	}
	else{
	  writeoutput("</pre>

	<pre>An invalid output_method value was supplied: Check the outputmethod='' of the <job > element for this request in the subsite's configuration file"
	);
	}</pre>
	If the outputMethod is “run_code” then the ColdFusion page is included, otherwise REQUEST.htmlToOutput is populated and output.

5. Administering the Dynamic Website

This section describes administration of the Dynamic website, including production, indexing and caching issues.

The Dynamic Web Site is administered at /admin/index.cfm.

Table 3 Website Administration

Process	URL
Development	http://**chlorine.fcc.gov:4020**/admin/index.cfm
Daily Updates QA	http://**hegre.fcc.gov:4021**/admin/index.cfm (Used every day)
Code QA	http://**hegre.fcc.gov:4020**/admin/index.cfm
Production	http://**huldra.fcc.gov:4020**/admin/index.cfm and http://**ula.fcc.gov:4020**/admin/index.cfm These two servers are load balanced, but they do not automatically replicate. Therefore changes have to be made to both machines.

This page contains links to pages that manage cache, indexes etc. It also makes available the "Production Prep" page which is designed to guide operations staff through the process of updating content. The Production Prep is mainly used in the Daily Updates QA environment (hegre on port 4021).

The admin piece of the dynamic site has not been worked on as a project in itself. For the most part it is built into the Dynamic Site Framework. Sometimes processes are triggered by chains of <cfhttp>'s which is really hard to follow. These techniques were used for speed – and also to allow processes to be run on the production machine without needing a separate login. (Eventually, when the dynamic site admin and the releases tool share a common login, we should make production admin a separate login for security sake.)

It should be noted that in production the webserver (grane) is configured to proxy *.htm requests to the application servers huldra and ula. For this reason admin ColdFusion files should NEVER be given .htm extensions, but ALWAYS .cfm so they cannot be run from the web server. It's an added security measure.

5.1. Production Preparation

This page takes a user through the steps required to update content on the dynamic site. In terms of programming, the most important steps are Step 3: Refresh Indexes (Should be Indices!) and Step 7: Refresh Production Cache and Indexes.

5.2. Refreshing Indexes

Indexes are used on the dynamic website for the same reason they are used in relational database management systems – to boost performance when performing lookups. There are, at time of writing, two such indexes located at `<host>/xml_indexes`. One is an index of auction details (auction_details_index.xml) and the other an index of releases details (index_of_releases.xml).

Generally, when updates are made to content, these indexes need to be refreshed. The operations team runs the refresh process for every update even though for some updates they may not actually be required.

5.3. How the Refreshing Process Works

1. On the production prep page the link “Refresh Indexes” has this url:
   `<host>/admin/production_prep.cfm?action=1`

2. production_prep.cfm contains the following code which handles this:

```
<cfif url.action EQ "1">
      <cfhttp method="get"
      url="#host#/admin/manage_indexes.cfm?action=build_all
      _indexes">
      <cfset indexesDone = "DONE">
      ...
      </cfif>
```

3. manage_indexes.cfm contains the following code which handles this:

```
...
<cfif url.action EQ "build_all_indexes">
<cfset host =
SESSION.configuration.getHost("application_server")>
<cfhttp method="get"
url="#host#/admin/index.cfm?job=build_auction_index"></cfhttp>
<cfhttp method="get"
url="#host#/admin/index.cfm?job=build_releases_index"></cfhttp
>
```

```
</cfif>
   ...
```

4. If you look at the /admin/configuration/configuration.xml file you'll see that the build_auction_index and the build_releases_index jobs result in XSLT transformations which get written to disk:

```
<job name="build_auction_index"
   xml="/configuration/configuration.xml"
   xsl="/admin/xsl/auctions_indexer.xsl"
   navigation_xml="/admin/configuration/navigation.xml"
   page_layout="main_only"
   cfc="cfcomponents.job.AdminJob"
   directory_to_index="/auctions/xml/auctions/"
   output_file="/xml_indexes/auction_details_index.xml"
   output_method="transform_to_disk"
/>
<job name="build_releases_index"
   xml="/configuration/configuration.xml"
   xsl="/admin/xsl/releases_indexer.xsl"
   navigation_xml="/admin/configuration/navigation.xml"
   page_layout="main_only"
   cfc="cfcomponents.job.AdminJob"
   directory_to_index="/auctions/xml/auctions/"
   output_file="/xml_indexes/index_of_releases.xml"
   output_method="transform_to_disk"
/>
```

5. Look at /admin/xsl/auctions_indexer.xsl. It runs from "indexStart" to "indexStop" which are set from parameters in /configuration/configuration.xml.

```
<index id="auctions" start="1" stop="150">
<directory>auctions.xml.auctions</directory>
</index>
```

   This should run on whatever files are in the /auctions/xml/auctions/ directory, but it currently runs looks for auction_id_1.xml to

auction_id_150.xml (based on the stop=”150” setting). What happens when we reach auction 151?

The XSL transforms the XML in numerous XML files into a single XML document – the index. This part, at least, is pretty nice and XSLT is *perfect* for the job. Notice the recursive call to the outputAuctionElement template. In XSLT that’s the only way you can iterate/loop (for-each can only be used to loop over an existing XML structure, you can’t use it like cfloop).

6. Now look at /admin/xsl/releases_indexer.xsl. Starting from “thisYear” it transforms the XML of /xml/releases/releases_<YEAR>.xml files. Again, outputIndex calls itself recursively, decrementing the year each time it does so. The recursive call terminates when the year is less than 1994 (there are no earlier releases XML Files)

7. The output_method of the two jobs is “transform_to_disk”. This is handled in the following lines in DynamicWebsiteController.cfc

```
if(REQUEST.outputMethod EQ "transform_to_disk"){
jobWriter = createObject("component",
"cfcomponents.util.JobWriter");
jobWriter.writeJobToXmlFile();
jobWriter.displayConfirmation();
}
```

5.4. Refreshing Production Cache and Indexes

As well as refreshing the indexes on the production app servers this link also causes the cache to be refreshed.

In the /configuration/configuration.xml the following element switches caching for the entire dynamic website on or off:

```
<caching enabled="true" max_pages="1000">
```

This can be accessed within the framework with the following:

```
<cfset cachingEnabled = SESSION.configuration.cachingIsEnabled()>
```

By default the output of PageElementMainContent will be cached*. If it is desired to override caching for a job then cachable=”false” should be added to the attribute of the <job>. For example `<job name=”dynamic_lookup” cachable=”false” ... />`

Caching is implemented by storing the HTML for Main Content in an APPLICATION scope struct. Each subsite has its own cache. The naming convention for a cache is `<subsite_name>Cache`, for example auctionsCache, servicesCache etc.

(This naming is handled automatically however – the developer does not need to do anything. The value of REQUEST.subsite is used to create a cache, and this is set from the following code in a subsite's index.htm: controller.serviceRequest(**"services"**, "index.htm");

Separating caches by subsite in this way makes it possible flush cache of one subsite while leaving the others intact.

Typical caching logic is employed whereby when a page is requested, the system tries to get it from the cache first. If not present the page is rendered (usually via XSLT) and the result added to the cache for future use. The key used to store HTML in a cache (which again is just a CF struct) is the QUERY_STRING of the page. QUERY_STRING is used to set the value of REQUEST.pageIdentifier. The only reason to do this was to make explicit the fact that this is value uniquely identifying a page.

5.5. Caching Specifics

Caching is handled in two places. 1) in the PageElementMainContent components at high level, and 2) in cfcomponents.util.CacheManager2.cfc at a low level.

Table 4 Caching process

If caching is enabled attempt to get the page from cache	In the viewPageElement() method of PageElementMainContent.cfc is the following code:
	`if (SESSION.configuration.cachingIsEnabled() and REQUEST.mode NEQ "n4_xsl"){`
	`REQUEST.htmlToOutput = attemptGetPageFromCache();`
	`}`
	This is the attemptGetPageFromCache() function:
	`<cffunction name="attemptGetPageFromCache">`
	`<cfif (SESSION.cacheManager.pageIsInCache())>`
	`<cfreturn "#SESSION.cacheManager.getPageFromCache()#">`
	`<cfelse>`
	`<cfreturn "">`
	`</cfif>`
	`</cffunction>`
	So, if the page is not in cache the function returns an empty string. (See getPageFormCache() below – I actually have redundant code above!)

pageIsInCache() method of CacheManager2.cfc	Here it is:
	`function getPageFromCache(){`
	`cache = getCache();`
	`page = "";`
	`if (REQUEST.jobIsCachable){ // default is true, unless <job ...cachable="false"... />`
	`if (pageIsInCache()){`
	`// get the page (a string of HTML) from the cache by it's key`
	`page = cache[REQUEST.pageIdentifier];`
	`}`
	`}`
	`return page;`
	`}`
	So if the page is not stored in cache the method returns an empty string.
	You should explore the getCache() and pageIsInCache() methods which this method calls as helper methods.

Caching a page	If caching is enabled the page is stored in cache for future requests.
	`if (SESSION.configuration.cachingIsEnabled() and REQUEST.mode NEQ "n4_xsl"){`
	`SESSION.cacheManager.setPageInCache();`
	`}`
	The details of this are in the setPageInCache() method of CacheManager2.cfc

setPageInCache() method of CacheManager2.cfc	Here it is:
	`<cffunction name="setPageInCache">`
	`<cfset verifyCache()> (Makes sure the cache has been defined)`
	`<cfif REQUEST.jobIsCachable>`
	`<cfif not(pageIsInCache()) and not(cacheIsFull()) and (REQUEST.job NEQ "not_found")>`
	`<cflock scope="application" timeout="2" type="exclusive">`
	`<cfset StructInsert(cache, REQUEST.pageIdentifier, REQUEST.htmlToOutput)>`
	`</cflock>`
	`</cfif>`

	</cfif> </cffunction> This method takes care of everything – the calling code doesn't have to do anything – no checks etc. (I need to do away with the attemptGetPageFromCache() in PageElementMainContent and simply call getPageFromCache() in the same way!) Notice it's the value of REQUEST.htmlToOutput that is stored. And this only happens if the following conditions are met: 1. The page is not already in cache (handled by pageIsInCache()) AND 2. The cache is not full AND 3. The page is not the "not_found" page. (I think this is old code and not relevant anymore)

cfcomponents.util.CacheManager2.cfc might be refactored in the future. For example as more and more dynamic subsites are releases we may find that the use of APPLICATION scoped structs is not a good mechanism. It might be better to write HTML to files on the app server. If this is the case the "interface" – i.e. the method names would stay the same and only their implementation would change. Probably the structs would remain but instead of storing HTML they'd store the paths to the files containing the HTML.

CacheManager2.cfc has other methods for managing content. Here are some notable ones:

Table 5 Cache alternate methods

flushPageFromCache()	Uses the values of REQUEST.subsite (via call to getCache()) and REQUEST.pageIdentifier to determine the cache and the page to flush <cffunction name="flushPageFromCache"> <cfset cache = getCache()> <cfif pageIsInCache()> <cflock scope="application" timeout="2" type="exclusive"> <cfset StructDelete(cache, REQUEST.pageIdentifier)> </cflock> </cfif> </cffunction>
flushPageFromCacheByParams()	Does the same thing as flushPageFromCache() but uses the

	two passed params to do it.
	(If this were true OO we'd have two overloaded methods both called flushPageFromCache() but alas!
	`<cffunction name="flushPageFromCacheByParams">`
	`<cfargument name="_cacheName">`
	`<cfargument name="_pageIdentifier">`
	`<cfif isdefined("APPLICATION.#_cacheName#Cache")>`
	`<cfset cache = APPLICATION[_cacheName & "Cache"]>`
	`<cflock scope="application" timeout="2" type="exclusive">`
	`<cfset StructDelete(cache, _pageIdentifier)>`
	`</cflock>`
	`</cfif>`
	`</cffunction>`

flushCache	`<cffunction name="flushCache">`
	`<cfargument name="cacheName">`
	`<cflock scope="application" timeout="2" type="exclusive">`
	`<cfset StructDelete(APPLICATION, cacheName)>`
	`<cfset APPLICATION[cacheName]=StructNew()>`
	`</cflock>`
	`</cffunction>`

flushAllCaches	`<cffunction name="flushAllCaches">`
	`<cfloop list="#StructKeyList(APPLICATION)#" index="key">`
	`<cfif find("cache", key)>`
	`<cfset StructDelete(APPLICATION, key)>`
	`</cfif>`
	`</cfloop>`
	`</cffunction>`

cacheIsFull()	`function cacheIsFull(){`
	`cache = getCache();`
	`return StructCount(cache) GT SESSION.configuration.getCacheMax();`
	`}`

The Dynamic Web Site Cache is managed using the "Manage Caches" page on the Dynamic Web Site Administration site. This page exists in development and in the QA environments – but caching is not enabled there so there's not a lot of pointing in checking it out! Look at them in production.

http://huldra.fcc.gov:4020/admin/manage_cache.cfm and
http://ula.fcc.gov:4020/admin/manage_cache.cfm.

These pages let you look at what is in each cache (currently just auctions) and also allow you to flush specific pages from cache and flush the entire cache. There is also a link "Reset Cache to Default" which is not used anymore – needs to be removed.

Notes:

1. This needs to be fixed – but currently you have to login before accessing this page (if you want to do anything to the cache beyond observe what's in there. So you'd go to the admin home page first to login.
2. The links to various caches should be dynamically generated (from the `<controllers>` structure in configuration/configuration.xml) but it's not done that way yet.
3. When you look at cfcomponents.util.CacheManager.cfc you'll see references to staticHtml. Caching also used to create static HTML versions of cached pages, but this is not done anymore. They were created so that Google etc could index them – but Google and other now index dynamic pages fine so we don't need this added piece.

You'll need to be logged in to do anything.

When clicking the "Refresh Production Cache and Indexes" on the production prep page the following happens:

1. The url is <host>/admin/index.cfm?job=production_prep&action=3&subsite=auctions
2. The <job> in configuration is:

```
<job name="production_prep"
      code="/admin/production_prep.cfm"
      navigation_xml="/admin/configuration/navigation.xml"
      page_layout="admin"
      cfc="cfcomponents.job.AdminJob"
      output_method="run_code"
      cachable="false"
/>
```

   So, the output of the job comes from the ColdFusion page /admin/production_prep.cfm

3. The file production_prep.cfm has the following code to handle action 3:

```
...
<cfelseif(url.action EQ "3")>
<cfif (isdefined("url.subsite"))>
<cfset appserver1 =
SESSION.configuration.getHost('production_application_server_1')>
<cfset appserver2 =
SESSION.configuration.getHost('production_application_server_2')>


            <cfhttp method="get"
            url="#appserver1#/admin/refresher_huldra.cfm?subsite=#url.s
            ubsite#&urlinvoked=#appserver1#">
<cfhttp method="get"
url="#appserver2#/admin/refresher_ula.cfm?subsite=#url.subsite#&urlinvo
ked=#appserver2#">
</cfif>
</cfif>
...
```

The values of appserver1 and appserver2 are set from configuration (to “huldra.fcc.gov” and “ula.fcc.gov” respectively). The only difference between the files refresher_huldra.cfm and refresher_ula.cfm is the message they send (via `<cfmail>`). The files are, then, run on the production application servers.

4. Take a look at these refresher_huldra.cfm and refresher_ula.cfm. Ultimately the key calls that get made are:

   Relating to Cache:

   ```
   SESSION.cacheManager.setDefaultPagesInCache(false /*writeHtmlToFile*/);
   which in turn calls the flushAllCaches() method of CacheManager2.
   ```

   Relating to Indexes:

   ```
   <cfhttp method="get"
   url="#host#/admin/index.cfm?job=build_auction_index"></cfhttp>
<cfhttp method="get"
url="#host#/admin/index.cfm?job=build_releases_index"></cfhttp>
```

* Unless the output method of the job Is “run_code.”

6. ISAS Data Update

Paul needs to provide more information for this section on the Integrated Spectrum Auction System.

6.1. Summary data

[admin]

|----[scheduled_processes]

|----[isas_data_update]

|----[cfc]

|----IsasAuctionPoller.cfc

|----IsasWebserviceInvoker.cfc

|----[webservice]

|----AuctionSummaryService.cfc

|----[xml]

|----force_isas_auction_poll.xml

|----[xsd]

|----force_isas_auction_poll.xsd

|----main.cfm

The summary data displayed at the top of an in progress and closed auction (bidding days, net bids etc) comes directly out of ISAS. While an auction is in progress this data needs to be updated at the close of each round.

6.2. ISAS Data Update Mechanism Functions

At a high level the mechanism to perform this update works like this:

1. The process is scheduled to run every 5 minutes

2. If an auction is in progress (either Live, Internal Mock or Customer Mock) the process invokes a web service to get the latest ISAS data as XML

3. If that data has been updated the process writes a copy of the XML to the server on which it is running.

6.3. Files

main.cfc

This is the file that is actually scheduled (in CF Administrator)

It determines what auctions, if any, are in progress and in which "mode" they are running (Live, Mock etc)

It then creates IsasAuctionPoller.cfc instances for each in progress auction and fires their poll() methods

IsasAuctionPoller.cfc

Using instances of IsasWebServiceInvoker.cfc this object invokes the webservice which returns a string of XML data.

It parses the XML and checks it to see if it has been updated since the process last ran.

IsasAuctionPoller's are stored in APPLICATION scope and maintain a “lastRoundWritten" variable which is used to determine if the XML data has been updated

If the data has been updated it is written to disk

IsasWebServiceInvoker.cfc

Handles the details of actually invoking the web service.

The details of which web service it should invoke is hidden from the object - it doesn't need to know, all it needs to do is invoke a service and return it's results. The actual webservice to invoke is configured in the `<isas_data_update>` section of /admin/configuration/configuration.xml

In the case of the DEV and STAGING environments the web service is generally set to AuctionSummaryService.cfc in this package - rather than the ISAS webservice. In the case of PRODUCTION the webservice is set to the ISAS service.

This is done so that only the production servers poll ISAS directly, while DEV and STAGING servers poll the copies of the XML files written to our production servers.

In summary:

- In PRODUCTION the process invokes the ISAS webservice and writes the results to PRODUCTION servers.
- In DEV and STAGING the process invokes *our* webservice, which gets the XML from our PRODUCTION servers, and writes the results to DEV and STAGING servers respectively.Bullet 3

7. Configuration Files

The Framework makes use of configuration files to control the way a particular request for a page is processed.

This is done for a number of reasons:

- Key values like source of xml, xslt, code, urls etc do not have to be hard coded into the logic but can be looked up at run time.
  Should any of these values change the change can be made in one place without having to touch any code.
- The code can be simpler as less decisions have to be made – a lot of the decisions have already been made in the choice of configuration settings (such as what XXXJob component and what page_layout to use
- Code can be loosely coupled meaning dependencies between different components are reduced.
  For example, the JobFactory component is able to create an appropriate XXXJob object without having to know anything about what different XXXJob's are available or what they do.
- In many cases it should be possible to add new functionality without modifying existing code.
  For example there will be cases where new behavior can be achieved by creating a new XXXJob.cfc and referencing it in the configuration

8. ColdFusion

8.1. ColdFusion Components

Table 6 ColdFusion Components

Package	Description	Components

cfcomponents	Contains top level components	`DynamicWebsiteController.cfc`
		`Configuration.cfc`
		`Data.cfc` (deprecated)
		`View.cfc` (deprecated)

cfcomponents.exception	Components for handling Exceptions	`ExceptionHandler2.cfc` `ExceptionHandler.cfc` (deprecated)

cfcomponents.job	Components for setting REQUEST scope variables needed for processing of a page	`AdminJob.cfc`
		`Job.cfc`
		`JobFactory.cfc`
		`ProtectedAdminJob.cfc`
		`ServiceJob.cfc`
		`WeeklyPNsJob.cfc`
		(XXXJob.cfc's will be added as needed)

cfcomponents.view	Components for building the actual page.	**`Utility classes for building the page:`**
		`BreadCrumbsBuilder.cfc`
		`JobViewer.cfc`
		
		`PageElements:`
		`AbstractPageElement.cfc`
		`PageElementAdminLogin.cfc`
		`PageElementBreadCrumbs.cfc`
		`PageElementBreadCrumbsOutreach.cfc`
		`PageElementBreadCrumbsService.cfc`
		`PageElementFooter.cfc`
		`PageElementHead.cfc`
		`PageElementLeftColumn.cfc`
		`PageElementMainContent.cfc`
		`PageElementMainNav.cfc`

		PageElementMastHead.cfc PageElementRelatedSites.cfc PageElementSearchBox.cfc PageElementTitleBar.cfc
cfcomponents.util	Various useful functionality bundled into components	Authenticator.cfc CacheManager2.cfc Configuration.cfc Directory.cfc FileReader.cfc HttpRequestSender.cfc JobWriter.cfc Logger.cfc Utilities.cfc VariableInitializer.cfc XmlFactory.cfc

8.2. ColdFusion Usage

Table 7 ColdFusion Usage

Component	Description	Usage
cfcomponents.**DynamicWebsiteController**.cfc	The central controller of the dynamic website	Every request for a dynamic page should pass through this component. ▪ At a very high level it controls the whole process of building the page, but does not get involved in the details – all details are delegated to other components. ▪ It is where all exceptions are caught and passed off to be handled
cfcomponents.**Configuration**.cfc	Acts as an interface for accessing any configuration value. Provides methods for searching configuration XML and for quickly getting frequently used values	An instance of this component is created at the start of a session and stored in the session. When created it loads the main configuration.xml file and other subsite configuration.xml files as needed. Whenever a process needs to check something in the configuration, it can either call this component's searchConfiguration() method or use a number of frequently used methods like getHost()

		Examples: `<cfset host = SESSION.configuration.getHost("dynamic_website_host")>` This saves the hassle and performance hit of parsing and searching the XML file every time configuration needs to be checked.
cfcomponents.util.**XmlFactory**	Provides a neat way to parse XML into a ColdFusion XML Struct	An instance of this component is created at the start of a session and stored in the session. Whenever there is need to parse an XML file into a ColdFusion XML Struct a single method call will do it: `<cfset orginalXml = SESSION.xmlFactory.parseXml("/mydata/data.xml")>` This hides some of the details involved.
cfcomponents.job.**Job**	Base component of all XXXJob.cfc's. Any new XXXJob'cfc's should extend this and add any necessary functionality	Populates the REQUEST with tons of variables. The REQUEST is populated with all values required for processing any job. This includes job name, id, page etc.

9. RSS

This is a description of the RSS framework built into the Dynamic Site. This is perhaps one of the most complicated aspects of the site to date.

As of writing (01/11/2006) there are two feeds for each auction:

Auction Round By Round Summary feeds are for those who want to track the progress of an auction as each round closes

Auction Information feeds include the same data as above but also includes releases for the auction and upcoming key dates

Various steps are involved in the creation of these feeds, all of which happen automatically.

9.1. High Level Overview of the Process

1. On a scheduled basis data for feeds is gathered from 1) ISAS and 2) Wireless.fcc.gov
2. This data is written to an XML file for each feed (the XML is RSS plus some additional non-standard attributes used for processing)
3. When a feed is requested it is generated via the Dynamic Website Framework and is transformed from our custom RSS to pure RSS

9.2. Scheduled Processes

Certain scheduled processes (ColdFusion programs scheduled in ColdFusion Administrator) are key to the creation and update of RSS feeds.

The programs are in the following directory:

/admin/scheduled_processes/

and are specifically:

/admin/scheduled_processes/isas_data_update/main.cfm

/admin/scheduled_processes/rss_update/main.cfm

9.3. ISAS Update Process

The file /admin/scheduled_processes/isas_data_update/main.cfm is invoked by the ColdFusion Administrator every 5 minutes, however it terminates without doing anything 95% of the time. The fig below shows its flow of control:

So it can be seen that process only concerns round-by-round auction summary data (this may be expanded to include other data such as auction announcements)

It will only result in anything being done if 1) at least one auction is in progress OR 2) an auction is referenced in an XML file which forces the process to check ISAS. This is useful for mock auctions.

The force update XML is located at

/admin/scheduled_processes/isas_data_update/xml/ force_isas_auction_poll.xml

And looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<force_isas_auction_polling>
 <poll_auction
  auction_mode="L"
auction_id="81"
enabled="true"/>
</force_isas_auction_polling>
```

Auction mode can be L (Live), or I (Internal)

This process can be tested by appending the following to the relevant host:

`/admin/scheduled_processes/isas_data_update/main.cfm`

`Files in /admin/scheduled_processes/isas_data_update/ package`

`[cfc]`

`|---- IsasAuctionPoller.cfc` (contains logic to trigger Web Service invocation, perform updates if neccessary)

`|---- IsasWebserviceInvoker.cfc` (handles the actual web service invocation)

[webservice]

`|---- AuctionSummaryService.cfc` (mimics ISAS's web service. Used for testing purposes)

[xml]

`|---- force_isas_auction_poll.xml`

`main.cfm` (The file this actually scheduled. It determines if there are any auctions to run against.)

READ_ME.txt (Read it!)

These files should be studied to understand their working; it is not documented in detail here.

A.1.1. ISAS Update Process Figure

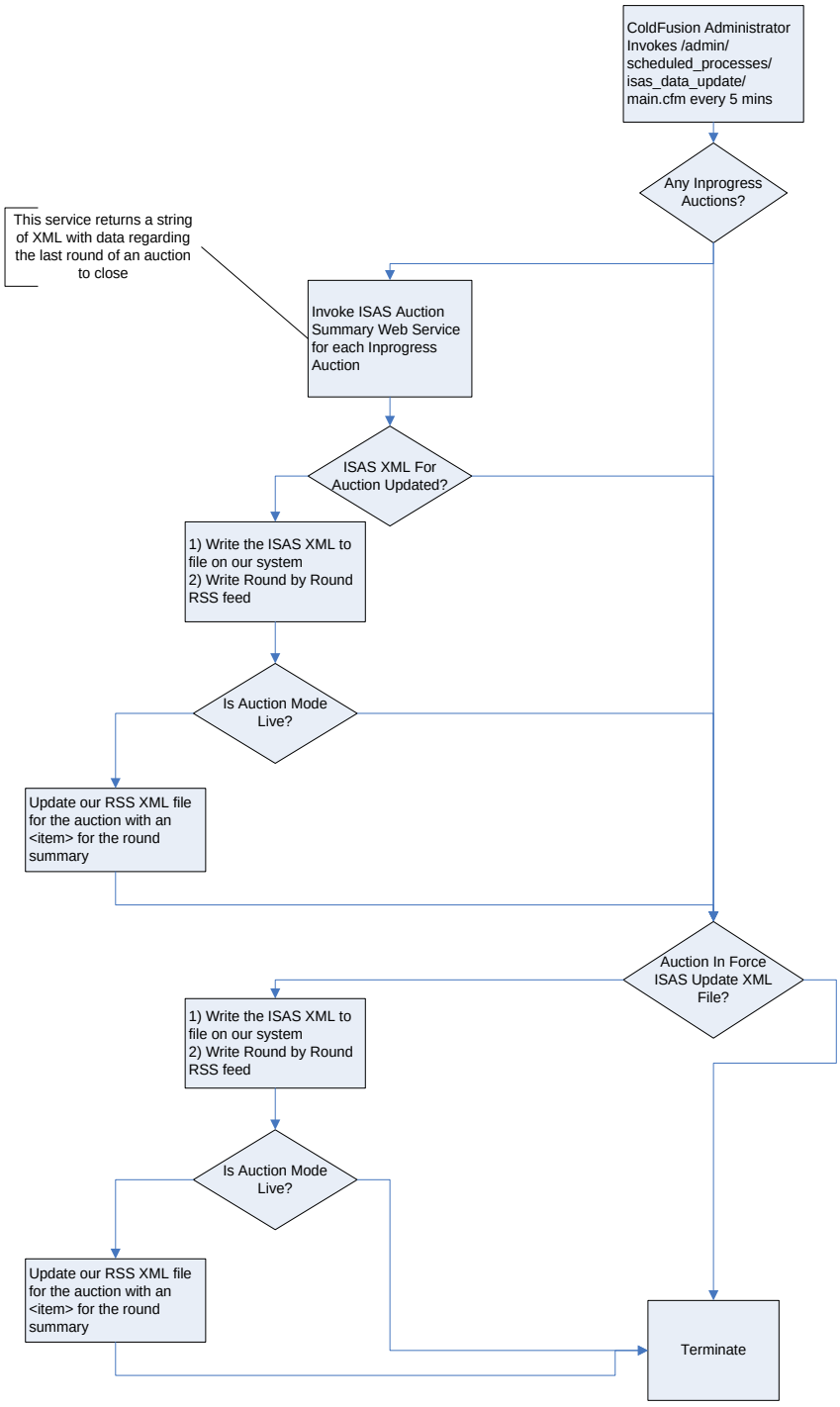


Figure 1 - ISAS Update Process

The following XML files are created by this process:

Say the auction is auction 64, the process will create:

1. /auctions/xml/auctions/isas_data/auction_id_64_isas.xml

   It is used to populate data on the auction summary page for auction 64 (but only if the auction mode is “L”)

2. /rss/xml/auction_rounds_64.xml

   It is used for testing RSS during an mock auction

3. /rss/xml/auction_info_64.xml

   It is only updated if the auction is live.

9.4. RSS Update Process

The file /admin/scheduled_processes/rss_update/main.cfm is invoked by the ColdFusion Administrator according to the following schedule:

08:00 (At which time *all* rss feeds are updated (if necessary) for *all* auctions. This is much more intensive which is why we only do it once)

12:00 Unscheduled, Scheduled, Inprogress and Closed Auctions with ISAS summary data only

16:00 Same as 12:00

18:00 Same as 12:00

It is the code which determines whether it is run against all auctions or only the limited set.

The scheduled file determines, for various “types” of RSS `<item>` whether updates are required. This notion of item type is for our own purposes and is not part of pure RSS. The types currently handled are:

- rounds_results
- release
- key_date

The file uses an instance of cfcomponents.rss.AuctionInfoRssGenerator.cfc to perform the actual updates to the XML file.

These files should be studied for details of their functioning. They are pretty well commented.

==== TO BE COMPLETED ====

10. Setting Up Help Files

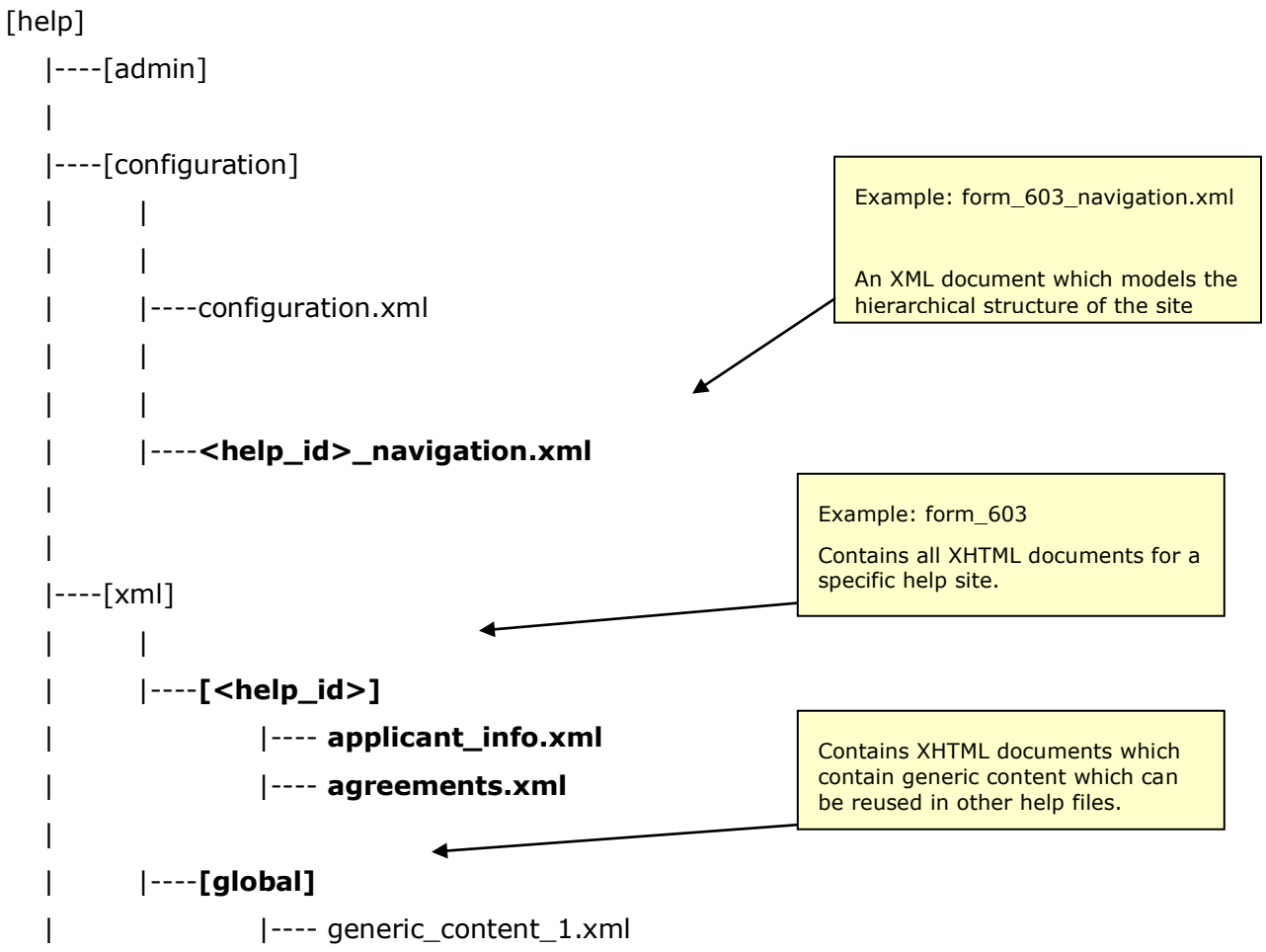
10.1. Purpose

This section of the Dynamic Website System Design Specifications document is intended to help content creators for the Dynamic help file sites.

10.2. How a Help Application is Organized

At the application server root directory (websitecfmxapp in Development, QA and Production environments) the help directory contains all files for all help applications.

The following figure shows the pieces you'll be working with and how these pieces fit into the whole.



|

|----[xsl]

10.3. Help ID

When creating a new help site you'll need to decide upon an id for it. This can be descriptive, such as "form_603". This id is then used in a number of places and needs to be used consistently in all of them.

10.3.1. URLs

A help file is called up by a url whose query string is formed thus:

`job=<job_name>&id=<help_id>&page=<page_name>`

For example:

`job=help_topic&id=form_603&page=applicant_info`

10.3.2. Navigation XML File

The file needs to be named according to the following convention:

<help_id>_navigation.xml

For example:

form_603_navigation.xml

10.3.3. XML Directory

Under /help/xml/ you'll need to create a directory to hold all XHTML files for the help site you're creating and this directory must be named with your chosen help id.

For example:

[help]

|----[xml]

|----**[form_603]**

|---- applicant_info.xml

|---- agreements.xml

ColdFusion will need to be able to write to this directory, and files written to it will need to be read/write accessible to the wtbweb group. Request this of A-Unix as needed.

10.4. Navigation

Navigation for a help site is created in a single XML file. Each help site has its own navigation file. The naming of the navigation file must follow the convention shown above where the id of the help site is used as prefix to the file name. For example, **form_603**_navigation.xml, **form_601**_navigation.xml and so on.

The XML used by the Dynamic Website framework to display main navigation is comprised of nested `<page>` elements. In the case of the help sites only level-one pages will initially display as links in the left nav. If a level one `<page>` in the XML has children `<page>` elements they will be displayed when the user clicks that level one link.

It's easy to change the navigation by cutting and pasting `<page>`'s from one place to another:

Any number of `<page>`'s can be nested under each other and to any depth (although in practice we'll only go four deep.)

10.4.1. Attributes of <page>

A typical `<page>` element looks like this:

```
<page job="help_topic" page="applicant_info" subsite="help"
label="Applicant Info" locations="leftnav;pagetitle" />
```

Table 8 Attributes of <page>

Attribute	**Description**
job	Typically the value will be "help_topic".
page	The name of the actual help topic.
subsite	This will always be "help"
label	The value of this attribute is used to populate: - The left nav link to the page - The secondary titlebar - The <title> in the HTML
locations	Just leave this set to "leftnav;pagetitle".

10.4.2. Nesting of Pages

To model the hierarchical nature of the help site you're working on, you nest `<page>`'s. Here are some examples (with attributes removed for clarity):

Example 1: Two page, one level website:

```
<page label="Animals"/>
```

```
<page label="Space"/>
```

Example 2: Five page, two level website:

```
<page label="Animals">
    <page label="Mammals"/>
    <page label="Reptiles"/>
</page>
<page label="Space"/>
```

Example 3: Seven page, three level website:

```
<page label="Animals">
    <page label="Mammals">
        <page label="Big Mammals"/>
        <page label="Small Mammals"/>
    </page>
    <page label="Reptiles"/>
</page>
<page label="Space"/>
```

Example 4: Nine page, four level website:

```
<page label="Animals">
    <page label="Mammals">
        <page label="Big Mammals">
            <page label="Elephants"/>
            <page label="Rhinos"/>
        </page>
        <page label="Small Mammals"/>
    </page>
    <page label="Reptiles"/>
</page>
<page label="Space"/>
```

(Note: be sure to close `<page>` elements which have no children with the forward slash `<page label="Space"/>`)

These are simple examples – in reality there will be tons of pages. An XML editor like XmlSpy 2005 lets you expand and collapse the hierarchy which is very helpful.

For more information see Appendix A: Representing the hierarchical structure of web pages in XML

10.5. Creating Content

The XHTML files containing the content for help sites contain pure XHTML within a root element of `<help_topic>`. For example:

```
<help_content>
    <h1>Applicant Information</h1>
    <p>
          Applicant Information is a page that
lets you
          identify the applicant…
    </p>
</help_content>
```

Eventually, XStandard will be used to create a small update tool for updating these XHTML documents. The tool will provide functionality to load XStandard with XHTML from the server and then save it back to server once edited.

In the meantime, XStandard can be used standalone to create XHTML which can then be copied and pasted into XML documents.

When creating a new document you would do the following:

1. In DreamWeaver (or other FTP client) navigate to the directory containing the XHTML for the help site you're working on
2. Create a new file there and name it <whatever>.xml
3. Create the following markup in the document
   `<help_content>`

   `</help_content>`
4. Use XStandard to create the XHTML content you need.
5. In XStandard switch to XML view and copy all the XHTML it generated
6. Switch to your XML editor and paste the XHTML *between* the opening and closing `<help_content>` tags.

7. FTP the XHTML to the server and test in browser (the query string will be `?job=help_topic&id=<your_help_id>&page=<whatever>`.
   For example if the help id is form_603 and the page you created is called appinfo_unincorporated.xml then the query string would be:
   `?job=help_topic&id=form_603&page=appinfo_unincorporated`

10.6. Reusing Content

There is a simple and limited facility for reusing generic content. There are two parts to this – one is to create the reusable content and the other is to reuse it. Follow these steps to do this:

10.6.1. Creating Reusable Content

1. Navigate to the /help/xml/global directory (if you're sure the content will only be used within one help site you can create it in the XML directory for that site.)
2. Follow steps 2 -7 above

10.6.2. Using Reusable Content

1. Navigate to the XHTML file you want to add reusable content to
2. Copy and paste the XHTML from your XML editor to XStandard
3. In the location where you'd like to include the content enter the following:
   `<help_topic_reference page="<whatever>" id="global" />`

   For Example, if you named the XHTML file generic_content.xml then the code, in context, would like this:

```
<help_content>
    <h1>Help Topic</h1>
    <p>blah blah</p>
    <help_topic_reference page="generic_content" id="global"/>
</help_content>
```

4. Follow steps 5-7 above. The content in generic_content.xml should now be included in the main page.

11. Setting up Reminders

Reminders are email messages sent out on a daily, weekly, monthly, one off or custom basis. They are configured and run on hegre:4021, the Daily Update/QA environment. They are configured in a simple XML file.

11.1. Reminder Configuration

The following package contains ColdFusion Code and an XML file which controls reminders.

[admin]

|----[scheduled_processes]

|----[reminder]

|----main.cfm

|----[cfc]

contains CFC's which perform the logic required to determine if a particular reminder should be run.

|----[xml]

|----reminder.xml (configures the reminders)

A reminder as defined in the XML looks like this:

```
<reminder>
<type>Daily | Weekly | Monthly | OneOff | Custom</type>
<day_of>1</day_of><!-- Use for Weekly and Monthly reminders. Represents The "day of" the Week or Month -->
<date>2005-09-27</date><!-- Use for OneOff reminders -->
<subject>Monthly Reminder</subject>
<message>Reminder to do something</message>
<to>pharvey@fcc.gov; rwilliams@fcc.gov</to><!-- List of email addresses separated by semi-colons -->
</reminder>
```

11.2. Technical explanation

The main.cfm file is scheduled to run on hegre:4021 at 2:00 AM every morning. Whether or not reminders are actually sent depends on the contents of the reminders.xml file. Main.cfm loops over this xml and for every <reminder> encountered instantiates appropriate ColdFusion

Components in the cfc directory. There is a base Reminder.cfc which are subclassed by DailyReminder, WeeklyReminder etc. ReminderFactory is used to actually create these instances. The XXXReminder objects contain logic to determine if a particular type of Reminder (Daily, Weekly) etc should actually be sent on a given day. XXXReminders supply the following methods:

```
init(reminderXmlNode)
reminderIsDue() which returns true of false
getSubject()
getMessage()
sendReminder()
```

The implementations of all methods apart from reminderIsDue() are defined in the base class, Reminder.cfc. The reminderIsDue() method is abstract in the base class, implementations being provided in sub-classes. The subclasses are, thus, very light-weight and simple.

12. Setting up the Daily Update QA Environment

The Daily Update QA Environment is on hegre. It is located at:

`/opt/bea8.1/WebSiteDomain2/websitestage/websitecfmx/websitecfmxapp` on that machine and is accessed as http://hegre.fcc.gov:4021.

In setting up your machine to update files on hegre you need create a mapped drive assigning is the letter X. It should be mapped to a directory we all share for this purpose.

1. Map the X Drive:
    1. In Windows Explorer Right Click your mchine name (in my case it's PHARVEY1).
    2. Choose "Map Network Drive…"
    3. Choose X: as the drive letter
    4. In the Folder field enter \\chlorine\websitecfmx
    5. Make sure "Reconnect at Login" is selected
    6. Click OK
2. Create the DreamWeaver Site

    This DreamWeaver site uses X:\DYNAMIC_SITE_XML as the local site and hegre as the remote site
    1. In DreamWeaver choose Site ➔ New Site
    2. Under local information, enter a "Site Name" (something like "HegreQADailyUpdates")
    3. For "Local Root Folder" enter X:\DYNAMIC_SITE_XML
    4. Under Remote Info Choose FTP Access
    5. For "FTP Host" enter hegre
    6. For host directory enter /opt/bea8.1/WebSiteDomain2/websitestage/websitecfmx/websitecfmxapp
3. Set up XMLSpy Project
    1. Choose Project ➔ Open Project
    2. Navigate to and open X:\DYNAMIC_SITE_XML\staging_xml.spp
4. Set up PVCS
    1. In PVCS Version Manager right click the Web-DynamicSite-DATA project in the left pane
    2. Choose "Set Workfile Location"
    3. Enter X:\DYNAMIC_SITE_XML
    4. To save this setting and assign it to a "Work Space" choose File ➔ Set WorkSpace.
    5. Click "New…"
    6. Enter a name for the workspace – something like "HegreQADailyUpdates"
    7. Make sure "Make this your default workspace" is selected
    8. Click OK

What you just did is configure a workspace setting (the root work file location of all files under the Web-DynamicSite-DATA pointed to X:\DYNAMIC_SITE_XML as the checkout to and check in from location. Then you set a workspace which affectively saves the configuration – otherwise it'd be lost when you closed PVCS.

5. Have Permissions Set

You'll need to get Maria to request the following permissions

Hegre: allow FTP write access to /opt/bea8.1/WebSiteDomain2/websitestage/websitecfmx/websitecfmxapp and all sub dirs

Chlorine: allow network write access to the mapped drive which is /opt/bea8.1/WebSiteDomain/websitestage/websitecfmx

13. Adding a Subsite to the Dynamic Website Framework

The following steps will take you through the setting up of a small demo subsite.

The subsite will be called demo and its main file will called be index.htm.

It will have three pages corresponding to the following jobs: “all_service_intros”, “all_service_links”, and “single_service_intro”. Content will be in XML documents, XSLT will be used to transform the XML to HTML.

This will be the site’s directory structure:

```
[root]
   |----[demo]
            |----[configuration]
                |----configuration.xml
                |----navigation.xml
            |----[xml]
                |----services.xml
                |----auctions.xml
                |----help_topics.xml
            |----[xsl]
                |----services.xsl
                |----auctions.xsl
                |----help_topics.xsl
            |----index.htm
```

13.1. Subsite Setup Demo

1) **You need to create a subdirectory to house the subsite.** Create a sub directory off the root to store the configuration, navigation, xml content, xslt/cf code etc for the subsite. Give it the name “demo”.

2) **The subsite needs a central “controller” file to start processing.** In this directory create a file and name it index.htm basing it on an existing one. (See /services/index.htm as a starting point.) In reality you may find that you need to modify the functionality of the page. Index.htm should include the following:

```
controller = CreateObject("component",
"cfcomponents.DynamicWebsiteController");
```

```
controller.serviceRequest("demo", "index.htm");
```

3) **You need a directory to store configuration files for the subsite.** Under the `[demo]` directory create a directory called configuration

4) **The subsite needs a configuration file to define settings for each distinct "job".** In the `[configuration]` directory create a file called `configuration.xml`. (Look at the configuration file for services as an example). The file will contain - <job>'s which specify many of the details the framework needs to respond to a request – such as which page layout to use, what XXXJob.cfc to use, where to find the XML and code for the request. Etc.

5) **You need to define at least one <job>.** This will the tell the framework what to do, what xml, xslt, cfc's to use etc.

   In configuration.xml enter the following:

```
<config>

      <job name="all_service_intros"

           xml="/demo/xml/all_service_intros.xml" (You'll create this
      file in a bit)

           xsl="/demo/xsl/all_service_intros.xsl" (You'll create this
      file in a bit)

           navigation_xml="/demo/configuration/navigation.xml" (You'll
      create this file in a bit)

           cfc="cfcomponents.job.Job"      (Already exists in the
      framework)

           page_layout="base"              (Already exists in the
      framework)

           output_method="transform"       (Tells the framework it
      needs to perform an XSL Transformtation)

      />

</config>
```

6) **The Framework expects an XML file defining navigation for the subsite.** In the `[configuration]` directory create a file and name it `navigation.xml`.

   Define the navigation thus:

```
<?xml version="1.0" encoding="UTF-16"?>
<pages>

      <page job=”all_service_intros” subsite=”demo”
      locations=”leftnav;pagetitle;breadcrumbs” label=”Service
      Introductions”/>

</pages>
```

This defines a one page subsite. The Framework is built to create left navigation using nested <page> nodes. The attribute locations tells the Framework to use the <page> element to create the <title> of the HTML page (“pagetitle”), a link in the left nav (“leftnav”), and the breadcrumb trail (“breadcrumbs”).

7) **The Framework needs to know where to find a subsite’s configuration file and what its main controller file name is.** This is defined in the main configuration file (<root>/configuration/configuration.xml). In the main configuration file add the following:

   a. Under <controllers> add

   ```
   <controller id="<any unassigned id>" name="<subsite name>"
   file_name="name of main file"/>
   ```

   In this case:

   ```
   <controller id="10" name="demo" file_name="index.htm"/>
   ```

   b. Under <configuration_files> add

   ```
   <configuration_file
      site="<subsite name>"
      path=”<path to configuration.xml file for subsite starting
   from document root>"
    />
   ```

   In this case:

   ```
   <configuration_file
      site="demo"
      path=”/demo/configuration/configuration.xml"
    />
   ```

8) **The Framework needs to know what page layout to use.** Ordinarily you’d add a <page_layout> to the main configuration file specifying the CFC’s that will output the various page elements (usually these CFCs will be the same as most of them are very generic. Here and there a new one may need to be created.

   In this case in the <job> you specified a page_layout attribute of “base” which is already created so you don’t need to do this.

9) **If it does not already exist the XXXJob.cfc specified in the cfc attribute of <job> must be created.**

In this case `cfcomponents.job.Job` was specified which is the base class and already exists.

10) **If they do not already exist the CFCs specified in the <page_layout> must be created.**
In this case you're using the "base" page layout and everything is already created.

11) **The Framework needs the XML file specified in `<job xml="" >`.** In this case it is `/demo/xml/all_service_intros.xml`
Create the file and enter the following:

```
<?xml version="1.0" encoding="UTF-16"?>
<service_introductions>
<service_introduction id="218_219" label="218-219 MHz Radio Service">
<page_section type="introduction">
                        <paragraph>
                                The 218-219 MHz Service <foot_note_ref id="1"/> is a short-
                                distance communication service designed for licensees to
                                transmit information, product, and service offerings to
                                subscribers and receive interactive responses within a
                                specified service area. Mobile operation is permitted.
                                Rules permit both common carrier and private operations, as
                                well as one- and two-way communications. Potential
                                applications include ordering goods or services offered by
                                television services, viewer polling, remote meter reading,
                                vending inventory control, and cable television theft
                                deterrence.
                        </paragraph>
<paragraph>
                                Although new rules are designed to allow licensees the
                                maximum flexibility to structure services to meet market
                                demand, The 218-219 MHz band is insufficient for the
                                transmission of conventional full-motion video. 218-219 MHz
                                Service channels may also be unable to support proposed
                                operations that require large amounts of spectrum,
                                including certain video, voice, and advanced data
                                applications.
                        </paragraph>

</page_section>
                <page_section type="introduction">
<paragraph>
                                The 700 MHz guard bands consist of a total of six megahertz
                                of paired spectrum that was allocated to protect public
                                safety operations in immediately adjacent bands from
                                harmful interference while at the same time promoting the
                                efficient use of this spectrum. These guard bands are
                                licensed to a new class of licensee, Guard Band Manager,
                                engaged in the business of leasing spectrum to third
                                parties on a for-profit basis. Guard Band Managers must
                                adhere to specific technical and operational measures
```

```
                                    designed to minimize interference to public safety
                                    licensees. Guard Band Managers can lease their spectrum to
                                    system operators or directly to end users. You can read
                                    more <url job="about" id="700_guard" subsite="services"
                                    label="about 700 MHz guard bands"/>, the <url
                                    job="licensing" id="700_guard" subsite="services"
                                    label="licensing process"/>, and current ways of <url
                                    job="licensing 1" id="700 guard" subsite="services"
                                    label="obtaining spectrum"/>.
</paragraph>
        </page_section>
        </service_introduction>
        </service_introductions>
```

The <page_section>'s are taken from the following two files:

`/services/xml/218-219/topic_home.xml`

`/services/xml/700_guard/topic_home.xml`

Open the topic_home.xml files of another 3 services (every service has one) then copy and paste the first <page_section> from each into your new file. (In each case the type attribute of the page_section will be "introduction").

12) **The Framework expects the XSLT file specified in the `<job xsl="">`.** In this case it is `/demo/xml/all_service_intros.xsl`
Create the file and enter the following:

```
        <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

        <xsl:output method="html"/>

        <xsl:template match="/">
                        <div class="singleColumn">
<h1>All Service Introductions</h1>

<!--
TO DO
-->
        </div>
        </xsl:template>

        </xsl:stylesheet>
```

13.2. Assignment 1: The Flow of Control Approach

In the "TO DO" section above, write XSLT code that will output the following HTML for each <page_section>.

```
...
<div class="label"> <!-- Value of label attribute of the page_section --> </div>
<div class="text"> <!-- Contents of all <paragraph>'s in the page_section --> </div>
<div class="row"> </div>
...
```

Don't worry about elements within <paragraph> (such as <url>, <foot_note_ref> or formatting tags etc. Just output the text

1. This will loop over each <page_section> in your XML source document and output the value of the label attribute of each one. (Add this code inside)

```
<xsl:for-each select="//page_section"> ← The // in the select means "all nodes wherever they occur within the current context."
<xsl:value-of select="@label"/><br/>
</xsl:for-each>
```

2. This will loop over each <page_section> and output the value of the label attribute and the values of the <paragraph>s of each one.

```
<xsl:for-each select="//page_section">
<strong><xsl:value-of select="@label"/></strong><br/>
<xsl:for-each select="//paragraph">
<xsl:value-of select="."/> ← select="." Refers the to text content of the current node
</xsl:for-each>
</xsl:for-each>
```

13.3. Assignment 2: The Event Driven Approach

In above assignment you directly controlled the flow of events by saying, in affect, "for-each page_section show me the value-of it's label attribute". With the event driven approach you create templates which sort of "catch" and handle nodes and attributes as the XSLT processor churns through them. It's akin to SAX in this respect.

1. Inside `<div class="singleColumn">` do this:

```
<xsl:apply-templates/>
```

That's it, upload and reload the webpage. This tells the processor to churn through the entire XML document and hand over each and every node it encounters to templates assigned to handle them. However, you did not have not defined any templates of your own yet.

2. After the closing tag of the `<xsl:template match="/">` template define this template:

```
<xsl:template match="page_section">
<xsl:value-of select="@label"/><br/>
</xsl:template>
```

Now when you reload the page, you have an instruction or "rule" for the XSLT processor to follow whenever it encounters a `<page_section>`

13.4. Extending Job.cfc

If you need to create a new XXXJob.cfc it should start like this:

```
<cfcomponent displayname="XXXJob" extends="Job">
  <cfscript>


    function processJob(){
      /* First, call the parent, or super, class's processJob() method which
does stuff that every job needs */
SUPER.processJob();


/* Stuff specific to the Sub Site Goes Here */
...
    }
  </cfscript>
</cfcomponent>
```

14. 404 Handling and Aliases

1) A request is made to grane such as http://wireless.fcc.gov/services/personal/radiocontrol/data/bandplan.html
2) Grane can't find the file so it pulls up a custom 404 page - 404_PROD.html
3) 404_PROD.html simply contains the following:

```
<script>
      document.location.href="http://wireless.fcc.gov/auctions/custom404.htm?page=" + document.location + "&ref=" + document.referrer;
</script>
```

4) So now control has passed to custom404.htm on the application server with page and ref parameters passed to it
5) Custom404.htm uses the values of `url.page` and `url.ref` to create a search string, which in this case would be "`/services/personal/radiocontrol/data/bandplan.html`"
6) It then searches `/configuration/services_404_mappings.xml` for this string using XPath (if it were an auctions page it would search `/configuration/auctions_404_mappings.xml`
7) If it finds it uses data from the XML to construct the new url and redirects to it with `<cflocation>`
8) If it does not find it forwards to the "not found" page

For development and testing of new redirects use http://riga.fcc.gov/<file_path_to_test> (Note there is no Port# when doing this)

This mechanism, while created to redirect users who have book marked old static pages now replaced with dynamic counterparts, is also useful when the FCC wants to give out a simple url rather then a full blown dynamic url. For example, when a new auction is announced they want to publish a url such as http://wireless.fcc.gov/auctions/62 rather than http://wireless.fcc.gov/auctions/default.htm?job=auction_summary&id=62. In this case an entry needs to be added to the mappings XML file.

15. Accessibility and the Dynamic Website

One of the benefits of a dynamic website which uses "templates" to build pages is that greater consistency can be achieved. This certainly applies to accessibility features. Many accessibility features are built right into the framework and will be employed on pages automatically. For example accessibility requirements for tables, while quite laborious to add by hand, are programmed in to the dynamic website and are rendered consistently and accurately. The following are some of the accessibility enhancements built into the dynamic website.

15.1. Page layout

"Use CSS for layout where possible but do not require it…A good rule for design is to separate content from presentation using CSS. Style sheets serve an additional purpose for a user with a disability. A user can program a custom style sheet to override the styles set by the original page developer. This custom style sheet can enlarge the text or create a particularly high- or low-contrast color scheme. By eliminating text and paragraph attributes in the HTML file and defining them in a separate CSS file, the page content is more easily manipulated to meet the individual needs of the user."

- The dynamic website uses CSS for page layout. This eliminates the use of deeply nested tables which have hitherto been used to control layout, an approach which presents difficulties for users of assistive technology.

15.2. Tables

"Identify table row and column headers."

- Cells which are headers (column or row headings) always use the `<th>` tag.

"Associate every cell with a row and column when using data tables."

- The `scope="col"` or `scope="row"` attribute values are used inside `<th>` tags to associate them with the rows and columns they label.
- Additionally, the "id" and "headers" attributes are used to associate specific cells with the header associated with them.

In addition, the following measures have been taken to improve accessibility:

- The summary attribute of tables is used to provide a synopsis of the content of the table. (This is not a requirement of 508 but is a good thing to do).
- Tables are kept simple in structure. The dynamic site does not facilitate the creation of overly complex tables which are difficult for users of assistive technology to interpret.

15.3. Scaling of Text

*"Use relative rather than absolute units in markup language attribute values and style sheet property values." (*W3C Accessibility Check List)

- While this is not actually a 508 requirement, we have implemented this recommendation. This supports the Text Size feature of IE allowing users to adjust text size for easier reading.

15.4. Alternative Text

"Provide a text equivalent for every non-text element"

- All images have alt tags.
- In addition, when hyperlink text is simply "pdf" or "Word" the actual title of the document is provided in the title attribute of the alt tag to be read by assistive technology.

15.5. Forms

"Explicitly associate form controls and their labels with the LABEL element"

Labels for input fields are consistently associated with the input fields they refer to using the label tag and id and for attributes.

16. Proposed Website System Updates

This section contains a list of desired changes to the Dynamic Site Code.

16.1. CacheManager2.cfm

Remove old code for handling “default” pages. (NOTE: be careful – DO NOT DELETE any methods! DO NOT BREAK ANY CALLING CODE!!!)

Ultimately what gets cached, rather than a string of HTML, could be a Component something like:

```
<cfcomponent name=”cachedPageElements
lastUpdated = ””;
mainNavHTML = “”;
mainContentHTML = ””;

setMainNav(stringOfHTML){
mainNavHTML = stringOfHTML
}
setMainContent(stringOfHTML){
mainContentHTML = stringOfHTML
}

getMainNav(){
return mainNavHTML;
}
       getMainNav(){
return mainContentHTML;
}

</fcomponent>
```

This way we could cache more than just the main content. Nav could also be cached which is good – rendering the nav for complex sites (like help) is expensive.

16.2. XHTML for Service Narrative

I would propose making all of our existing narrative type content XHTML – wrapped in custom tags. This would affect auctions articles, auction factsheets, services etc. It would pave the way for easier update of such content using XStandard or something similar.

What's Involved?

p, br, em, strong, a, div, table, tr, td, th,

In our current custom XML we have things like `<cell type="data">` or `<row type="column_labels">`. How should these be handled in XHTML?

Table 9 XML to XHTML Conversion

Our Custom XML	XHTML Version
Paragraph	
`<paragraph>Text</paragraph>`	`<p>Text</p>`

`<paragraph label="Topic">Text</paragraph>`	Modified XHTML:
	`<p label="Topic">`
	- or -
	Pure XHTML
	`<h5>Topic</h5>`
	`<p>Text</p>`
	- or -
	`<div class="label">Topic</div>`
	`<p>Text</p>`

url	
`<url host="wireless.fcc.gov" path="/auctions/01/" file="release1.pdf" label="Click Here">`	`<a href="http://wireless.fcc.gov/auctions/01/release1.pdf">Click Here</a>` TO THINK ABOUT: How to handle auto generation of "PDF", "Word" etc...
`<url job="auction_summary" subsite="auctions" id="1" label="Auction 1"/>`	`<a href="/auctions/default.htm?job=auction_summary&id=1">Auction 1</a>`
`<url ... with_arrow="true"/>`	`<a href="" class="arrow-link">Click Here</a>`
foot_note_ref	
`<foot_note_ref id="1"/>`	`<a href="#1">1</a>`
list	
`<list type="unordered">`	`<ul>`

<list_item>Report Filing</list_item> <list_item>Report Search</list_item> </list> <list type="ordered"> <list_item>Report Filing</list_item> <list_item>Report Search</list_item> </list> <list type="ordered" **space_between_items="true"**> <list_item>Report Filing</list_item> <list_item>Report Search</list_item> </list>	- Report Filing - Report Search - Report Filing - Report Search <ol class=”spacing”> - Report Filing - Report Search
tabular_data	
<tabular_data summary="">	<table> <summary>Blah</summary> <caption>Blah</caption> ... </table> No border attributes, but possibly different classes
row	
<row type="column_labels"> … </row>	<th> … </th>
<row>	<tr>
<row type="separator"/>	<tr class=”separator”><td/td></tr>?
cell	
<cell>Value</cell>	<td>
<cell type=”label”>Value</cell>	<td class=”label”>Value<td>
<cell type=”data”>Value</cell>	<td class=”data”>Value</td>

value (value of a cell – used when a cell has more than one value)	
<cell> <value>Text</value> </cell>	<cell> Text </cell> </cell>
<cell> <value type="label">Text</value> <value>Text</value> </cell>	<cell> Text Text </cell>
<cell> <value type="note">Text</value> </cell>	<cell> Text </cell>

Questions.

Should the XHTML be pure XHTML or should it be XHTML "plus" – adding in our own elements and attributes? I favor strict XHTML.

Appendix A. Representing the Hierarchical Structure of Web Pages in XML

In XML, hierarchical structure is represented by nesting one element inside another. So if one `<page>` is the parent of another `<page>` it would be coded like this:

```
<page label="parent">
    <page label="child"/>
</page>
```

If there are three levels in the hierarchy it would look like this:

```
<page label="grand parent">
    <page label="parent">
        <page label="child"/>
    </page>
</page>
```

In the Dynamic Website, the navigation of a subsite is described in XML by the use of such `<page>` elements. Just as the pages in a subsite are organized hierarchically, so the XML models this same hierarchy.

Appendix B. Dynamic Website Acronyms

CFC – ColdFusion Component

CSS – Cascading Style Sheets

FCC – Federal Communications Commission

HTML – Hypertext Markup Language

ISAS – Integrated Spectrum Auction System

PVCS -

QA – Quality Assurance

RSS – Rich Site Summary (or Really Simple Syndication)

URL – Uniform Resource Locator

WTB - Wireless Telecommunications Bureau

XHTML - Extensible Hypertext Markup Language

XML – Extensible Markup Language