

Web Experience Management Framework

Version 1.1

Developer's Guide



FATWIRE CORPORATION PROVIDES THIS SOFTWARE AND DOCUMENTATION “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. In no event shall FatWire be liable for any direct, indirect, incidental, special, exemplary, or consequential damages of any kind including loss of profits, loss of business, loss of use of data, interruption of business, however caused and on any theory of liability, whether in contract, strict liability or tort (including negligence or otherwise) arising in any way out of the use of this software or the documentation even if FatWire has been advised of the possibility of such damages arising from this publication. FatWire may revise this publication from time to time without notice. Some states or jurisdictions do not allow disclaimer of express or implied warranties in certain transactions; therefore, this statement may not apply to you.

Copyright © 2011 FatWire Corporation. All rights reserved.

The release described in this document may be protected by one or more U.S. patents, foreign patents or pending applications.

FatWire, FatWire Content Server, FatWire Engage, FatWire Satellite Server, CS-Desktop, CS-DocLink, Content Server Explorer, Content Server Direct, Content Server Direct Advantage, FatWire InSite, FatWire Analytics, FatWire TeamUp, FatWire Content Integration Platform, FatWire Community Server and FatWire Gadget Server are trademarks or registered trademarks of FatWire, Inc. in the United States and other countries.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. AIX, AIX 5L, WebSphere, IBM, DB2, Tivoli and other IBM products referenced herein are trademarks or registered trademarks of IBM Corporation. Microsoft, Windows, Windows Server, Active Directory, Internet Explorer, SQL Server and other Microsoft products referenced herein are trademarks or registered trademarks of Microsoft Corporation. Red Hat, Red Hat Enterprise Linux, and JBoss are registered trademarks of Red Hat, Inc. in the U.S. and other countries. Linux is a registered trademark of Linus Torvalds. SUSE and openSUSE are registered trademarks of Novell, Inc., in the United States and other countries. XenServer and Xen are trademarks or registered trademarks of Citrix in the United States and/or other countries. VMware is a registered trademark of VMware, Inc. in the United States and/or various jurisdictions. Firefox is a registered trademark of the Mozilla Foundation. UNIX is a registered trademark of The Open Group in the United States and other countries. Any other trademarks and product names used herein may be the trademarks of their respective owners.

This product includes software developed by the Indiana University Extreme! Lab. For further information please visit <http://www.extreme.indiana.edu/>.

Copyright (c) 2002 Extreme! Lab, Indiana University. All rights reserved.

This product includes software developed by the OpenSymphony Group (<http://www.opensymphony.com/>).

The OpenSymphony Group license is derived and fully compatible with the Apache Software License; see <http://www.apache.org/LICENSE.txt>.

Copyright (c) 2001-2004 The OpenSymphony Group. All rights reserved.

You may not download or otherwise export or reexport this Program, its Documentation, or any underlying information or technology except in full compliance with all United States and other applicable laws and regulations, including without limitations the United States Export Administration Act, the Trading with the Enemy Act, the International Emergency Economic Powers Act and any regulations thereunder. Any transfer of technical data outside the United States by any means, including the Internet, is an export control requirement under U.S. law. In particular, but without limitation, none of the Program, its Documentation, or underlying information of technology may be downloaded or otherwise exported or reexported (i) into (or to a national or resident, wherever located, of) any other country to which the U.S. prohibits exports of goods or technical data; or (ii) to anyone on the U.S. Treasury Department's Specially Designated Nationals List or the Table of Denial Orders issued by the Department of Commerce. By downloading or using the Program or its Documentation, you are agreeing to the foregoing and you are representing and warranting that you are not located in, under the control of, or a national or resident of any such country or on any such list or table. In addition, if the Program or Documentation is identified as Domestic Only or Not-for-Export (for example, on the box, media, in the installation process, during the download process, or in the Documentation), then except for export to Canada for use in Canada by Canadian citizens, the Program, Documentation, and any underlying information or technology may not be exported outside the United States or to any foreign entity or “foreign person” as defined by U.S. Government regulations, including without limitation, anyone who is not a citizen, national, or lawful permanent resident of the United States. By using this Program and Documentation, you are agreeing to the foregoing and you are representing and warranting that you are not a “foreign person” or under the control of a “foreign person.”

FatWire Web Experience Management Framework

Document Revision Date: Mar. 28, 2011

Product Version: Version 1.1

FatWire Technical Support

www.fatwire.com/Support

FatWire Headquarters

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

www.fatwire.com

Table of Contents

1	Welcome to FatWire WEM Framework!	7
	Introduction	8
	Prerequisites for Application Development	10
	Getting Started.....	13
2	Overview	15
	WEM Framework	16
	REST Services.....	16
	UI Container	18
	Registration.....	18
	WEM Context Object	19
	Single Sign-On	20
	Authorization Model	21
	Custom Applications	23
3	‘Articles’ Sample Application	25
	Overview	26
	Launching the ‘Articles’ Sample Application.....	27
	Building and Deploying the ‘Articles’ Application	27
	Registering the ‘Articles’ Sample Application	28
	Testing the ‘Articles’ Application.....	30
4	Developing Applications	31
	Overview	32
	Application Structure.....	32
	Making REST Calls.....	36
	Making REST Calls from JavaScript	36
	Making REST Calls from Java	38
	Constructing URLs to Serve Binary Data	38

Context Object: Accessing Parameters from the WEM Framework	39
Same Domain Implementations	39
Cross-Domain Implementations	40
Methods Available in Context Object	42
Registration Code	43
Registering Applications with an iframe View	43
Registering Applications with JavaScript and HTML Views	44
5 Developing Custom REST Resources	47
‘Recommendations’ Sample Application	48
Overview	48
Building and Deploying the Application	48
Testing the Application	48
Creating REST Resources	49
Application Structure	49
Steps for Implementing Custom REST Resources	50
6 Single Sign-On for Production Sites	51
SSO Sample Application	52
Deploying the SSO Sample Application	52
Application Structure	53
Implementing Single Sign-On	54
Implementing Single Sign-Out	55
7 Using REST Resources	57
Authentication for REST Resources	58
Acquiring Tickets from Java Code	58
Acquiring Tickets from Other Programming Languages (Over HTTP)	59
SSO Configuration for Standalone Applications	61
Configuring CAS	65
REST Authorization	66
Security Model	66
Using the Security Model to Access REST Resources	67
Configuring REST Security	67
Privilege Resolution Algorithm	67
Managing Assets Over REST	68
8 Customizable Single Sign-On Facility	69
Customizing Login Behavior for the WEM Framework	70
Components of the Default CSSO Implementation	71
Configuring and Deploying Custom SSO Behavior	72
Extending the Default CSSO Classes	73
Identifying Your Java Classes to Spring for Instantiation	75

Mapping External User Identifiers to Content Server Credentials	78
Restarting the CAS Web Application	80
Running the CSSO Sample Implementation	81
Sample CSSO Classes	82
Sample Spring Configuration File	83
Sample CSSO Components	85
9 Buffering	87
Introduction	88
Architecture	88
Using Buffering	89
Appendix A. Registering Applications Manually	91
Registration Steps	92
Reference: Registration Asset Types	96
FW_View Asset Type	96
FW_Application Asset Type	97

Chapter 1

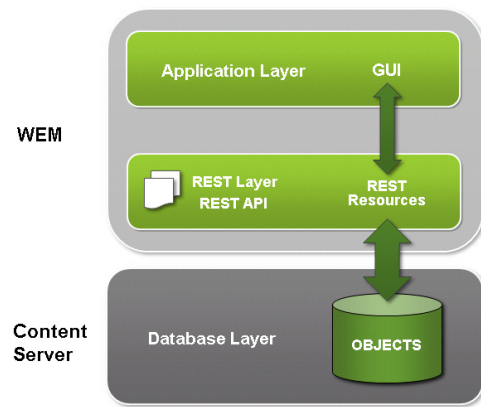
Welcome to FatWire WEM Framework!

- [Introduction](#)
- [Prerequisites for Application Development](#)

Introduction

FatWire Web Experience Management (WEM) Framework provides the technology for developing applications to run on the FatWire product suite. A single default administrative interface, WEM Admin, supports centralized application management and user authorization. Single sign-on enables users to log in once and gain access to all applications allowed to them during the session.

The WEM Framework requires a content management platform. In this release, the WEM Framework runs on FatWire Content Server and ships with the CS Representational State Transfer (REST) API. Objects in Content Server's database, such as sites, users, and data model map to REST resources in WEM.



When implemented on the WEM Framework, applications communicate with Content Server's database through REST services. The applications appear in WEM Admin as list items on the **Apps** page (Figure 1). Administrators authorize users, which involves configuring access to the applications and their resources. To this end, the WEM Admin interface exposes authorization items (along with applications) through links on the **menu bar**.

Figure 1: Apps Page, WEM Admin

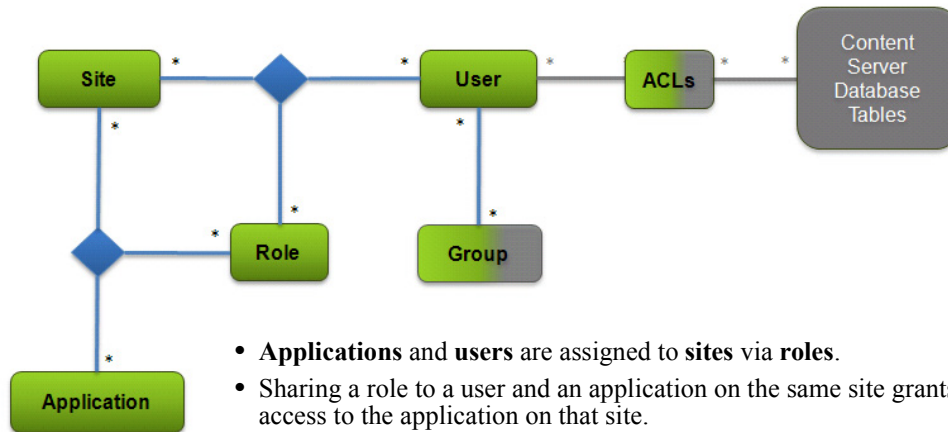
The screenshot shows the 'Apps' page in the WEM Admin interface. The page has a navigation menu at the top with links for 'Admin', 'Sites', 'Apps', 'Users', and 'Roles'. The main content area is titled 'Apps' and contains a table of applications. The table has two columns: 'APP NAME' and 'DESCRIPTION'. The table lists four applications: 'Admin' (WEM Admin), 'Advanced' (Advanced), 'Dash' (Dash), and 'Insite' (Insite). Above the table is a search bar and a 'Sort by: App Name' dropdown. Below the table is a 'Show rows: 5' dropdown and a '1-4 of 4' indicator. On the right side of the page, there is an 'About Apps' section with text explaining the application interface and a gear icon.

APP NAME	DESCRIPTION
Admin	WEM Admin
Advanced	Advanced
Dash	Dash
Insite	Insite

Applications
List

Coupling the items as shown in [Figure 2](#) enables applications for users.

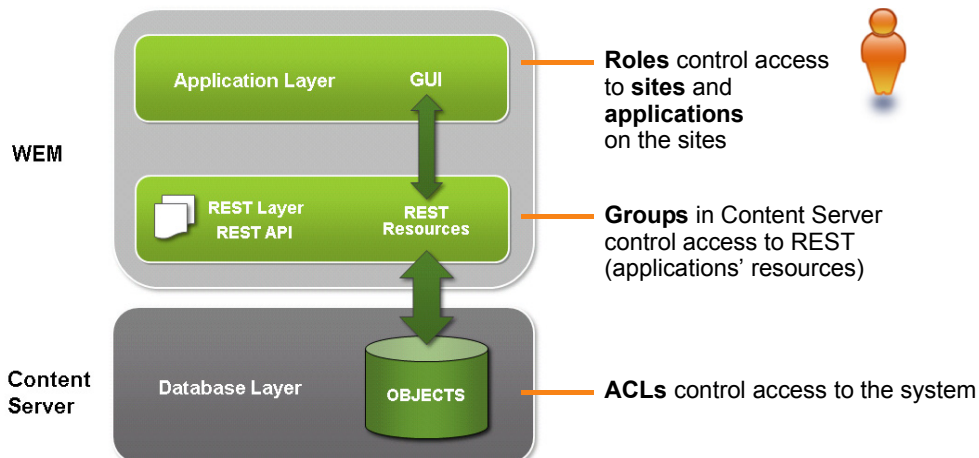
Figure 2: Authorization Model



- **Applications** and **users** are assigned to **sites** via **roles**.
- Sharing a role to a user and an application on the same site grants the user access to the application on that site.
- Users are assigned to **groups**, which control access to applications' resources (REST resources).
- **ACLs** are assigned to users, providing them with access to the system.

Using WEM Admin, general administrators can create and otherwise manage sites, applications, users, and roles. Groups and ACLs must be configured in Content Server Advanced. They are exposed in WEM Admin, in user accounts.

Once the coupling is complete, users are authorized at the database, REST, and application levels.



Experienced Content Server developers will recognize that the WEM Framework extends the use of sites and roles to control access to applications. However, unlike Content Server, the WEM Admin interface does not expose the data model. The REST API does. In this respect, WEM Admin can be thought of as strictly an authorization interface, supported by Content Server Advanced (for configuring ACLs and groups).

Although WEM Admin is seldom used by developers, the concepts behind user authorization can come into play in application development. The rest of this guide describes the WEM Framework as it relates to application development and provides examples of application code.

Prerequisites for Application Development

Developing an application involves coding the application's logic, deploying the application, and registering the application to expose it in WEM Admin for administrators to manage and make available to other users. This guide is not intended to be a tutorial on application development, but a reference to orient experienced application developers to the WEM Framework. Users of this guide must be expert Content Server developers with a working knowledge of the technologies listed in this section. Required resources are also listed below.

Technologies

- Representational State Transfer (REST), used to communicate with the Content Server platform
- Central Authentication Service (CAS), which is deployed during Content Server installation to support single sign-on for WEM
- Java Server Pages Standard Tag Library (JSTL), Java, JavaScript, Jersey, and the Spring MVC framework, in order to follow the code of the “Articles” sample application provided with WEM

Content Server Interfaces, Objects, and APIs

Developers must have a working knowledge of:

- CS Advanced (Content Server's administrative interface)
- Content Server's basic and flex asset models
- Asset API
- ACLs, which protect database tables and define the types of operations that can be performed on the tables
- Concept of sites and roles

Documentation

To follow this guide you will need the following documentation:

- *REST API Resource Reference*
- *REST API Bean Reference*

Information about Content Server's data model and Asset API is available in the *Content Server Developer's Guide*. Information about ACLs, sites, and roles, and their usage in Content Server is available in the *Content Server Administrator's Guide*.

Note

Product documentation and specifications are available on the FatWire e-docs site at:

<http://support.fatwire.com>

Accounts can be opened from the home page.

Sample Applications and Files

- The following sample applications are used in this guide:
 - *Articles*, a lightweight content management application
 - SSO sample application, a small authentication application for production sites. The application is packaged as `wem-ss0-api-cas-sample.war`.
 - *Recommendations*, which demonstrates the process of creating REST resources
- The Customizable Single Sign-On facility is used in this guide to illustrate customization of login behavior for the WEM Framework.
- WEM Framework ships with sample files to illustrate cross-domain implementations and management of assets over REST using our API.

All sample applications and files are located in the `/Samples/WEM Samples` folder in your Content Server installation directory.

Application Access

When using this guide, or developing and testing, you will access some or all of the applications listed below:

- **CAS web application.** You will specify its URL in the “Articles” sample application to enable single sign-on:

`http://<server>:<port>/<cas_application_context>/login`

where `<server>` is the host name or IP address of the machine running CAS and `<cas_application_context>` is the context path of the CAS web application.

- **Content Server Advanced** interface, if you decide to register applications manually:

`http://<server>:<port>/<cs_application_context>/Xcelerate/LoginPage.html`



Log in with the credentials of the general administrator that was used during the Content Server installation process (or an equivalent general admin). The default login credentials are `fwadmin/xceladmin` (**same for logging in to WEM Admin**).

Note

General administrators on Content Server systems running the WEM Framework are specially configured. During the installation process, `fwadmin` was automatically added to the `RestAdmin` group for unrestricted access to REST services, and enabled on `AdminSite` where the WEM Admin interface runs. More information about WEM-related changes to Content Server is available in the *Rollup Installation Guide*.

- **WEM Admin**, to test the results of your application registration process:
`http://<server>:<port>/<cs_application_context>/login`
 Log in as fwadmin (or an equivalent user). The sequence of screens is the following:

1. Login Screen:

2. Transition screen (if you are logging in for the first time or in to a site that you have never accessed before). Select **AdminSite** and the first icon, **Admin**:

3. WEM Admin **Sites** page. Registered applications are listed on the **Apps** page.

SITE NAME	DESCRIPTION
A New Site	This is another new site.
AdminSite	AdminSite
BurlingtonFinancial	Burlington Financial
FirstSiteII	FirstSite II
GE Lighting	GE Lighting

Getting Started

The chapters of this guide can be read in any order:

- For information about the WEM Framework, see [Chapter 2, “Overview.”](#)
- For a demonstration of the “Articles” application, see [Chapter 3, “Articles’ Sample Application.”](#)
- For information about the “Articles” application code, programmatic application registration, and cross-domain implementations, see [Chapter 4, “Developing Applications.”](#) (An example of manual application registration is available in [Appendix A.](#))
- For information about creating REST resources, see [Chapter 5, “Developing Custom REST Resources.”](#)
- For a demonstration of the SSO sample application, see [Chapter 6, “Single Sign-On for Production Sites.”](#)
- For information about system security, see [Chapter 7, “Using REST Resources.”](#)
- For information about customizing the login behavior for the WEM Framework, see [Chapter 8, “Customizable Single Sign-On Facility.”](#)
- For information about buffering, see [Chapter 9, “Buffering.”](#)

Chapter 2

Overview

- [WEM Framework](#)
- [REST Services](#)
- [UI Container](#)
- [Single Sign-On](#)
- [Authorization Model](#)
- [Custom Applications](#)

WEM Framework

The application developer's environment consists of the WEM Framework running on Content Server via REST services. Applications can be written in any language to make REST calls to Content Server. Custom-built applications can be deployed to an application server other than the platform's, and therefore written independently of the platform's deployment infrastructure.

Support for application development is in the following components (which are also described in their own sections in this chapter):

- **REST services**, a set of programmatic interfaces that provide access to Content Server's objects.
- **UI container**, which exposes registered applications. Registration enables rendering of the applications' interfaces. The UI container also supports the WEM Context object, used by applications to get details from the WEM Framework about the logged-in user and current site.
- **Single sign-on (SSO)**, which enables authenticated WEM users to log in only once to access all applications allowed to them during the session. (The Content Server installation process installs the Central Authentication Service web application to support single sign-on in WEM.)
- **REST authorization model**, which provides fine-grained access control over REST resources, based on group membership. Application development does not directly involve authorization (which is configured graphically in WEM Admin and Content Server Advanced), except when a predefined user is specified in the code.

WEM Admin is also part of the WEM Framework, but seldom used in application development, mainly to test the results of the application registration process, or to obtain administrative information about sites, users, groups, and roles. Information about WEM Admin is available in the *WEM Framework Administrator's Guide*

REST Services

The REST API exposes Content Server's data model:

- Basic asset types and basic assets (read-write)
- Flex asset types and definitions (read only)
- Flex children and parents (read-write)
- Indexing to support asset searches

The following objects are also exposed by the REST API. They are used mainly by administrators in the authorization process (the objects are displayed in the WEM Admin interface):

- Sites (read-write)
- Users (read-write)
- Roles (read-write)
- ACLs (read only)
- Groups (read only), introduced in this release to control access to the REST layer.
- Auxiliary services: user locale and server time zone

(Sites, roles, and users can be configured in WEM Admin. ACLs and groups are exposed in WEM Admin (under Users) as read-only items; they must be configured in Content Server Advanced.)

Objects in Content Server map to REST resources in WEM. All other features, such as publishing, workflow, database management tools, and page caching must be accessed from Content Server's Advanced interface or via JSP and XML tags.

Among the authorization objects that general administrators manage, sites and roles are the most likely candidates for application development, depending on your requirements. You can also specify "predefined" users to simplify administrators' authorization tasks.

- **Sites:** Using sites in application code is a requirement when the application's asset types and assets must be programmatically installed. The code must specify at least one site on which to enable the asset types (site-specific access to assets requires their asset types to be enabled on at least one site). Otherwise, you can install just the asset types (without naming any sites). Administrators will follow up by using Content Server's Advanced interface to enable the asset types and assets on sites of their own choice.
- **Roles** in WEM are used to manage access to applications. Sharing a role to a user and an application on the same site grants the user access to the application on that site. Roles can be used in application code to protect interface functions, such as "Edit." Content Server Advanced exemplifies an application with role-protected interface functions.
- **Users:** The only user you are likely to specify in your application code is the "predefined" user, to simplify administrators' authorization processes. Specifying the user involves coding a user name and password. Instead of authorizing all application users individually at the REST level, an administrator will authorize your predefined user. Permissions granted to the predefined user will be passed to the logged-in users when they access the application. More information about predefined users and the authorization model can be found on [page 21](#).

Keeping track of how sites and roles are used across the system is an administrators task that requires support from application developers. Tracking becomes especially important when the Content Server platform also functions as a staging system, only because the WEM Framework uses Content Server's database. For example, sites created in WEM Admin are stored in the database. They might not be used in Content Server for staging, but they are exposed in the Content Server Advanced interface, along with its dedicated CM sites. Conversely, sites that are created in Content Server Advanced for CM purposes are exposed in WEM Admin, where other applications can be assigned to those sites. For users to be properly authorized, developers must communicate to administrators the nature of the custom-built applications: the resources they use, role-protected interface functions, and predefined users, if any.

UI Container

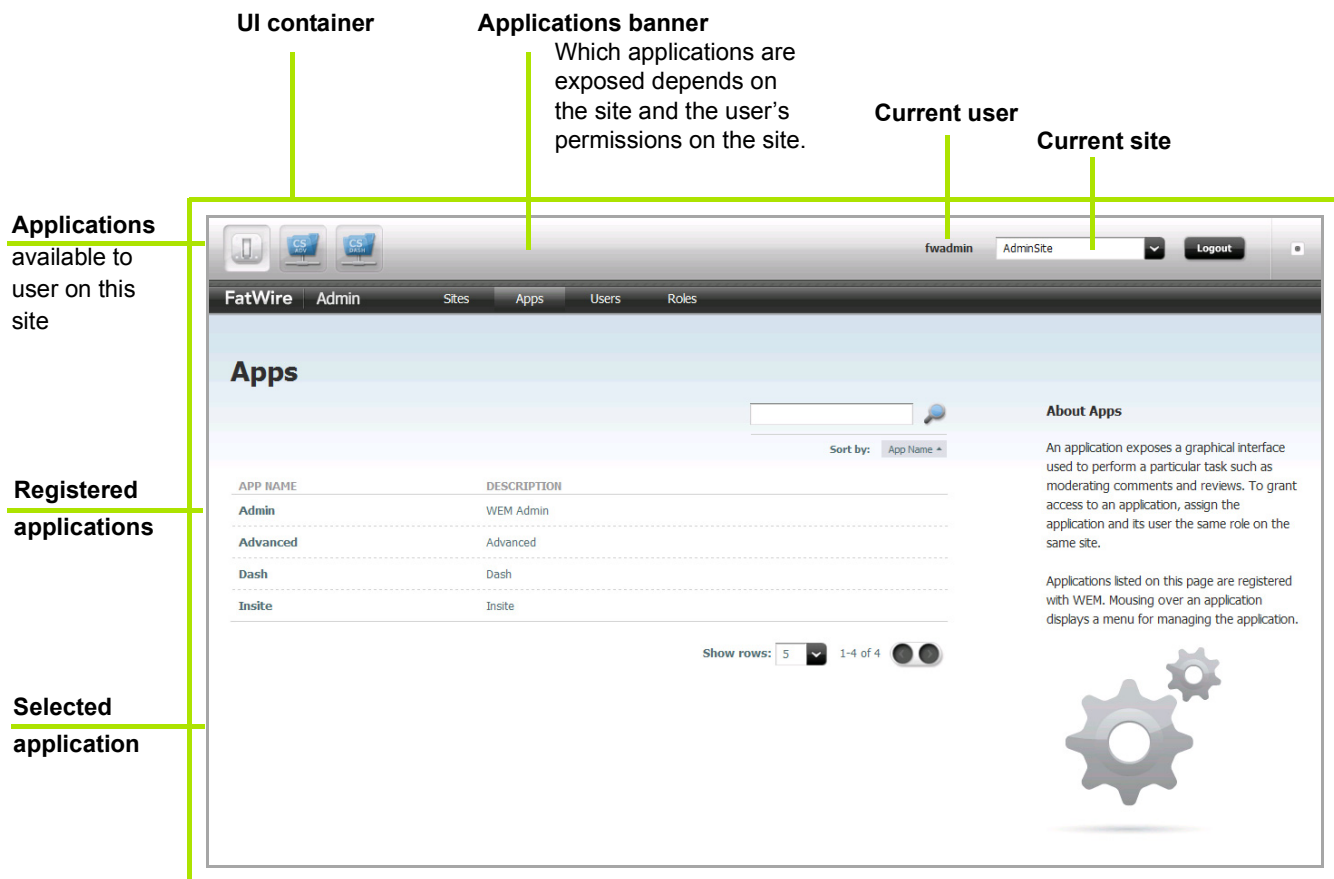
The UI container exposes registered applications and supports the Context object, used by applications to get information from the WEM Framework.

Registration

The purpose of registering an application is to expose the application in WEM Admin for administrators to manage and make available to other users. Registration allows the system to recognize the application as an asset, which in turn allows the system to

- list the application on the Apps page in WEM Admin,
- locate the icon you have chosen to represent the application,
- display the icon on the WEM login page, and in the applications banner on each site to which the application is assigned ([Figure 3](#)), and
- render the application's interface when the application's icon is selected.

Figure 3: Registered Applications in UI Container



Registering an application includes registering its views. While multiple and shared views are supported, applications with a single, unshared view are typical (and used in this guide). Views can be of type `iframe`, `HTML`, and `JavaScript`.

To support registration, WEM ships with the basic asset types `FW_Application` and `FW_View`. Both are created when the WEM option is selected during the Content Server installation process. They are enabled by default on `AdminSite` (also created during the Content Server installation process).

Registering an application (once it is deployed) requires creating an instance of `FW_Application`, creating an instance of `FW_View` for each view, and associating the `FW_View` instances with the `FW_Application` instance. Applications must be registered on `AdminSite`, even if they will be used on other sites. Registration allows applications to be assigned to other sites.

Applications can be registered either programmatically via the REST API's `applications` service, or manually from Content Server's Advanced interface. Programmatic registration is preferred. For an example, see "[Registering the 'Articles' Sample Application](#)," on page 28. For general instructions, see "[Registration Code](#)," on page 43. (An example of manual registration is available in [Appendix A](#).)

WEM Context Object

The UI container provides a JavaScript Context object (`WemContext`) to all applications inside the container. The Context object is used by the applications to get details from the WEM Framework about the logged-in user and site (for example, the current site's name from the UI container). The Context object also provides various utility methods that applications will use to share data. The Context Object can be used by applications running in the same domain as Content Server or in different domains. For more information, see "[Context Object: Accessing Parameters from the WEM Framework](#)," on page 39.

Single Sign-On

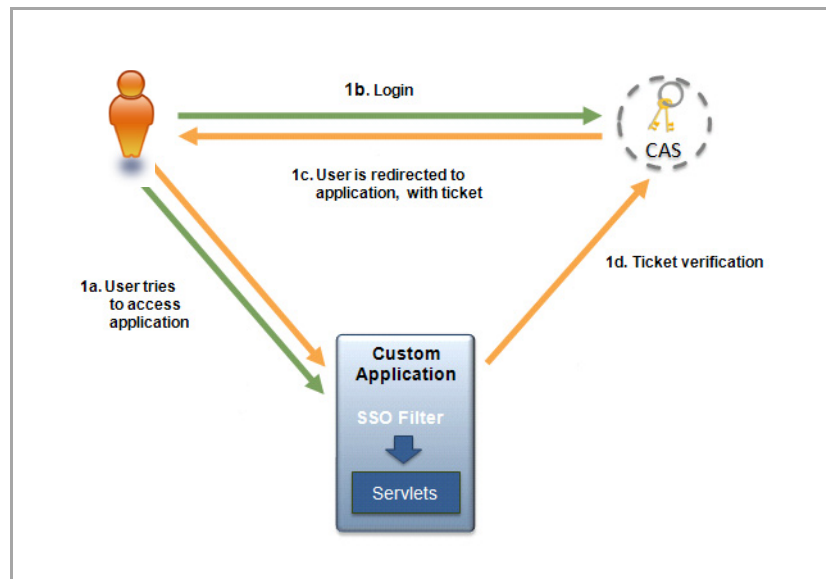
Single sign-on in the WEM Framework is implemented using Central Authentication Service (<http://www.jasig.org/cas>). As shown in the sample “Articles” example, the servlet filter that ships with WEM can be used out-of-the-box for any application that is deployed as a Java web application. If your application is developed using a different technology, refer to CAS clients specific to your choice of technology, at the following URL:

<http://www.ja-sig.org/wiki/display/CASC/Official+Clients>

When a user tries to access an application protected by CAS, the authentication system responds with the steps below.

1. Initial Access

- a. When the user first attempts to access an application protected by CAS,
- b. the user is redirected to the CAS login page. Upon successful login,
- c. the user is redirected back to the application with a ticket. The cookie for the CAS login page is saved.
- d. The application verifies the user’s identity by verifying the ticket against CAS. (On content management systems, CAS authenticates by default against Content Server’s database.)



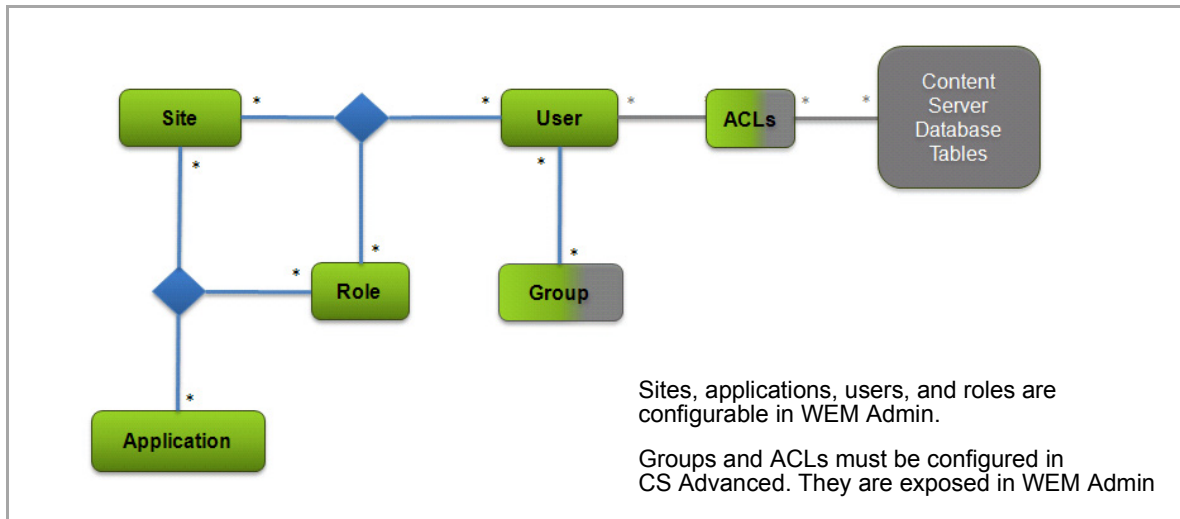
2. Subsequent Access

- a. When the user attempts to access another application protected by CAS, the user is redirected to the CAS login page.
- b. The cookie is retrieved from the request, implicit login is performed, and the login page is bypassed.
- c. The user is redirected back to the application with a ticket.
- d. The application verifies the user’s identity by verifying the ticket against CAS.

Authorization Model

Authorization is the process of granting users access to applications. General administrators are responsible for authorization by using WEM Admin to couple objects as shown in Figure 4. Developers can simplify the administrator's task by coding a predefined user in their applications. How the user fits into the authorization model is explained below.

Figure 4: Authorization Model



In Figure 4, Site, Application, User, and Role each have a counterpart menu option in WEM Admin. ACLs and groups are exposed on each user's page.

WEM Admin Menu bar

The screenshot shows the WEM Admin interface. The top navigation bar includes the following items:

- FatWire
- Admin
- Sites
- Apps
- Users
- Roles

The main content area displays the **Sites** page, which includes a table of sites and an **About Sites** section.

SITE NAME	DESCRIPTION
AdminSite	AdminSite
BurlingtonFinancial	Burlington Financial
FirstSiteII	FirstSite II
GE Lighting	GE Lighting
HelloAssetWorld	Hello Asset World

The **About Sites** section provides information about sites and their management.

Authorization is managed at three levels: application, REST, and database.

- Application-level authorization requires sharing a role to a user and an application on the same site, which grants the user access to the application on that site. If interface functions are role-protected, their roles as well must be shared to the application users.
- REST-level authorization regulates the user's permission to operate on the application's resources – *assuming ACLs are correctly assigned*. REST-level authorization requires configuring groups with privileges to operate on objects that map to REST resources. Users who are assigned to a group gain the group's privileges.

Developers can define a user in their applications (by user name and password) to act as a proxy for logged-in users, which eliminates the need for administrators to configure REST security for each logged-in user. Once an application is deployed and registered, a general administrator authorizes its predefined user by: 1) configuring the predefined user in WEM Admin for application access, 2) configuring a group (in CS Advanced) with privileges to operate on the applications' resources, and 3) assigning the predefined user to the group (by using either the WEM Admin or CS Advanced interface). The group's privileges are passed to the predefined user and then to logged-in users when they access the application. Supported security configurations are described and listed in "[REST Authorization](#)," on page 66. (The "Articles" sample application provided with WEM specifies a predefined user.)

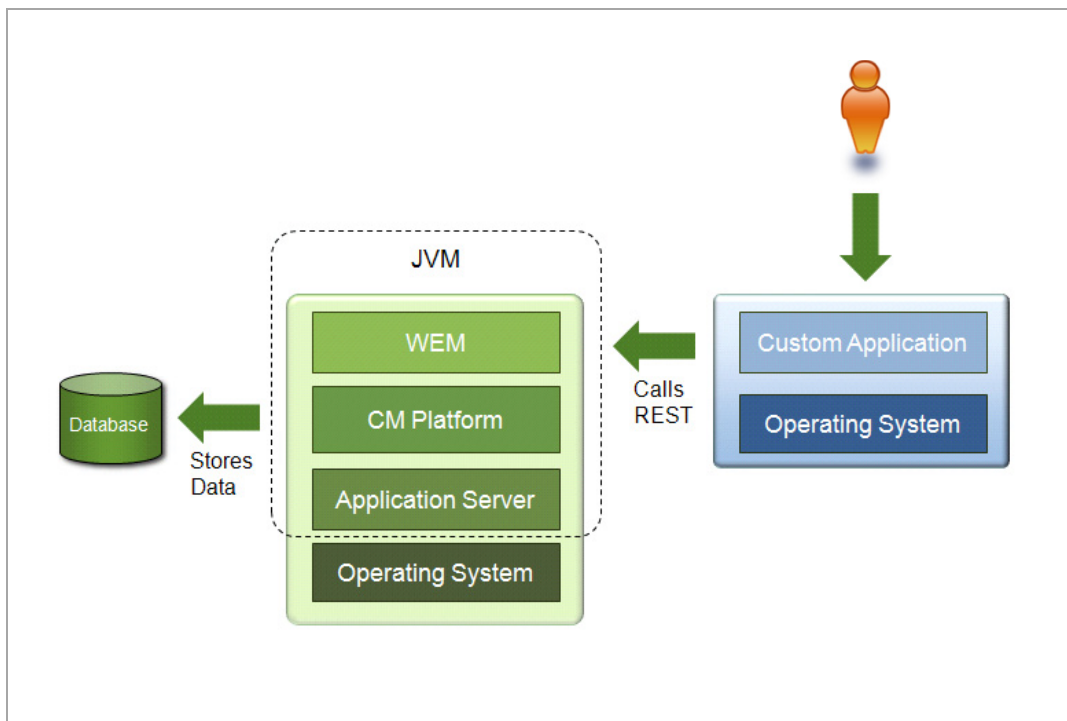
- At the database level, ACLs determine the individual user's access to the system, i.e., permission to log in and operate on the database, *regardless of the user's membership in any groups*. If a user lacks the appropriate ACLs and therefore permissions to the database tables, then membership in a group does not grant those permissions.

Default ACLs give users almost unrestricted permissions – but not the means – to operate on objects in many of the database tables. Those permissions are modulated at the REST level: Either directly by the user's membership in groups (in the absence of a predefined user), or indirectly by the application's predefined user and his membership in groups. Modifying a group's privileges to operate on objects modifies the group member's privileges to operate on resources. The same user on the Content Server side remains unaffected by group memberships. Permissions to content are still regulated by ACLs and actuated by sites and roles.

Custom Applications

Custom applications developed in WEM are often implemented in a loosely coupled manner to the content management platform. Because custom applications utilize the REST API Web services and SSO mechanism enabled by WEM, they are often deployed to an application server other than the platform's application server. Developers can therefore write custom applications completely independently of the platform's deployment infrastructure. Most custom applications are deployed remotely (Figure 5).

Figure 5: Remote Application Deployment



Custom applications can be implemented as content management or delivery applications. We recommend getting started with the content management side, as it typically does not require much performance tuning effort.

WEM ships with several lightweight sample applications, which you can launch and analyze as models for developing your own applications. “Articles” illustrates a content management application. [Chapter 3](#) contains instructions for launching “Articles.” Specifications can be found in [chapter 4](#), source code is provided in Content Server’s `Samples` folder, and other supporting information is provided in the REST API resource and Bean references. The SSO sample application is for authentication on live sites and the “Recommendations” application illustrates the creation of REST resources.


Chapter 3

'Articles' Sample Application

- [Overview](#)
- [Launching the 'Articles' Sample Application](#)
- [Testing the 'Articles' Application](#)


Overview

“Articles” is a simple content management application with richly documented source code and a self-installation process to help you quickly master information that is most important to developing applications. As the name implies, “Articles” enables the management of article assets. The application’s home page looks like this:



Name: Strategies
Description: Strategies for every playing surface
Category: Health
Source: <http://fatwire.com>

Do you triumph on clay but barely pass on grass? The key to versatility is rethinking how you play when conditions change. Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusantium doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo inventore veritatis et quasi architecto beatae vitae dicta sunt explicabo. Nemo enim ipsam voluptatem quia voluptas sit aspernatur aut odit aut fugit, sed quia consequuntur magni dolores eos qui ratione voluptatem sequi nesciunt. Neque porro quisquam est, qui dolorem ipsum quia dolor sit amet, consectetur, adipisci velit, sed quia non numquam eius modi tempora incidunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim ad minima veniam, quis nostrum exercitationem ullam corporis suscipit laboriosam, nisi ut aliquid ex ea commodi consequatur? Quis autem vel eum iure reprehenderit qui in ea voluptate velit esse quam nihil molestiae consequatur, vel illum qui dolorem eum fugiat quo voluptas nulla pariatur? Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusantium doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo inventore veritatis et quasi architecto beatae vitae dicta sunt explicabo. Nemo enim ipsam voluptatem quia voluptas sit aspernatur aut odit aut fugit, sed quia consequuntur magni dolores eos qui ratione voluptatem sequi nesciunt. Neque porro quisquam est, qui dolorem ipsum quia dolor sit amet, consectetur, adipisci velit, sed quia non numquam eius modi tempora incidunt ut labore et dolore magnam aliquam quaerat voluptatem.



Name: Tips
Description: What tennis tips can you learn from the pros?
Category: Sports
Source: <http://fatwire.com>

You may not play on the same level as Federer or Nadal, but that doesn't mean you can't learn something from their example. Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusantium doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo inventore veritatis et quasi architecto beatae vitae dicta sunt explicabo. Nemo enim ipsam voluptatem quia voluptas sit aspernatur aut odit aut fugit, sed quia consequuntur magni dolores eos qui ratione voluptatem sequi nesciunt. Neque porro quisquam est, qui dolorem ipsum quia dolor sit amet, consectetur, adipisci velit, sed quia non numquam eius modi tempora incidunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim ad minima veniam, quis nostrum exercitationem ullam corporis suscipit laboriosam, nisi ut aliquid ex ea commodi consequatur? Quis autem vel eum iure reprehenderit qui in ea voluptate velit esse quam nihil molestiae consequatur, vel illum qui dolorem eum fugiat quo voluptas nulla pariatur? Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusantium doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo inventore veritatis et quasi architecto beatae vitae dicta sunt explicabo. Nemo enim ipsam voluptatem quia voluptas sit aspernatur aut odit aut fugit, sed quia consequuntur magni dolores eos qui ratione voluptatem sequi nesciunt. Neque porro quisquam est, qui dolorem ipsum quia dolor sit amet, consectetur, adipisci velit, sed quia non numquam eius modi tempora incidunt ut labore et dolore magnam aliquam quaerat voluptatem.

The “Articles” home page displays two articles that can be edited directly in WEM, from the custom interface that you see in the figure above. The application demonstrates usage of Content Server’s REST API to perform a search query from Java code and an asset modification query from JavaScript code. The “Articles” application and REST services can be run on different application servers. Cross-domain restrictions in JavaScript prevent AJAX calls directly from the “Articles” application to the REST services. This is why a simple ProxyController is introduced. It redirects calls from JavaScript to WEM REST Web Services. Custom implementations may reuse this controller implementation.

The “Articles” application is based on the Spring MVC framework. “Articles” includes a predefined administrative user named `fwadmin` with password `xceladmin`, who is assigned to the REST group named `RestAdmin`. The application’s self-installer contains specifications for registering the “Articles” application and installing its asset model and sample articles. The application does not have internally configured sites or role-protected functions. It has a single, iframe view. Additional specifications are available in [Chapter 4](#), “Developing Applications.”

Launching the 'Articles' Sample Application

In this section, you will first build and deploy the “Articles” application, then run the installer.

Building and Deploying the 'Articles' Application

1. Determine or create the site to which you will assign the sample articles application. The default site is FirstSite II (a sample Content Server site). It is possible that FirstSite II is not installed on your system.

To select or create a site, log in to WEM Admin at the URL `http://<server>:<port>/<cs_application_context>/login` using the credentials of a general administrator (`fwadmin` / `xceladmin` are the default values).

Note

In [step 5](#), you will specify the site you have chosen here, which will allow the installer to enable the application's asset model and assets on that site.

2. Download and install SUN JDK (1.5 or later) from the following URL:
`http://java.sun.com/`
3. Download the latest Apache Ant from `http://ant.apache.org/` and place the Ant bin directory into the system PATH.
4. Copy `javax-servlet-api.jar` to the “Articles” application lib folder. The jar file can be taken from your application server's home directory (for example, Tomcat's `javax-servlet-api.jar` is located in the `home/lib` directory).
5. Set the following parameters in the `applicationContext.xml` file (in `src\articles\src\main\webapp\WEB-INF\`):
 - `casUrl`: Specify the URL of the CAS application:
`http://<server>:<port>/<context_path>`
 - `csSiteName`: Specify the name of the site that you selected in [step 1](#).
 - `csUrl`: Specify the URL where the Content Server platform is running:
`http://<server>:<port>/<context>`
 - `csUserName`: The default value is `fwadmin`. This is the application's predefined user, a general administrator with membership in the RestAdmin group which has unrestricted permissions to REST services. If you specify a different user, you must name a user equivalent to `fwadmin`. Instructions for creating a general administrator can be found in the *WEM Framework Administrator's Guide*.
 - `csPassword`: Specify the predefined user's password.
 - `articlesUrl`: Point to the URL where the sample application will be accessed.
6. Run the Ant build with the default target (enter **ant** on the command line).
7. Deploy the resulting `target/articles-1.0.war` to an application server.

On deployment, the following content is copied from source to target: The contents of the `lib` folder are copied to `/WEB-INF/lib`. The contents of the `resources` folder

are copied to `/WEB-INF/classes/`. For information about the structure of the source application, see [Chapter 4, "Developing Applications."](#)

Registering the 'Articles' Sample Application

The "Articles" application has a self-installer, which starts running when you log in to the `install.app` page. The installer registers the sample application (including the view) and creates its data model and assets in Content Server's database.

Note

Specifications for the registration asset types `FW_View` and `FW_Application` can be found in the *REST API Bean Reference* (and in [Appendix A](#)).

To run the 'Articles' installer

1. Navigate to the `install.app` page:

`http://<hostname>:<portnumber>/<context_path>/install.app`

For example:

`http://localhost:9080/articles-1.0/install.app`

2. Use any credentials to log in (the application's predefined user, specified by `csUserName` and `csPassword` on [page 27](#), provides you with permissions to the application. The sample application does not perform authorization checks as it does not use roles.)
3. The self-installation process invokes `InstallController.java`, which first registers the application (including the view, in an application Bean), then writes the sample asset type and assets to the database.
 - a. `InstallController.java` registers the "Articles" application with the WEM Framework:
 - `InstallController.java` creates an application asset named `Articles` (asset type `FW_Application`) in Content Server's database.

The `iconurl` attribute points to the URL where the icon representing the application is located.

The `layouturl` attribute specifies the URL of the `layout.app` page (implemented by `LayoutController.java`). The `layout.app` page defines the application layout.

The `layouttype` attribute takes the default (and only) value: `layoutrenderer`. Using the `layoutrenderer` value, the UI container is responsible for rendering the application's associated views by using the `layout.app` page, specified by `layouturl`.
 - `InstallController.java` creates a view asset named `ArticlesView` (asset type `FW_View`) in Content Server's database. The association between the view asset and the application asset is made through the `views` attribute in the `FW_Application` asset type.

Testing the 'Articles' Application

1. Navigate to the home . app page:


`http://<hostname>:<portnumber>/<context_path>/home.app`

For example:

`http://localhost:8080/articles-1.0/home.app`

2. Use any credentials to log in (the application's predefined user, specified by `csUserName` and `csPassword` on [page 27](#), provides you with permissions to the application. The sample application does not perform authorization checks as it does not use roles.)

WEM displays the application's home page:




Name: Strategies


Description: Strategies for every playing surface

Category: Health

Source: <http://fatwire.com>



Do you triumph on clay but barely pass on grass? The key to versatility is rethinking how you play when conditions change. Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusantium doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo inventore veritatis et quasi architecto beatae vitae dicta sunt explicabo. Nemo enim ipsam voluptatem quia voluptas sit aspernatur aut odit aut fugit, sed quia consequuntur magni dolores eos qui ratione voluptatem sequi nesciunt. Neque porro quisquam est, qui dolorem ipsum quia dolor sit amet, consectetur, adipisci velit, sed quia non numquam eius modi tempora incidunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim ad minima veniam, quis nostrum exercitationem ullam corporis suscipit laboriosam, nisi ut aliquid ex ea commodi consequatur? Quis autem vel eum iure reprehenderit qui in ea voluptate velit esse quam nihil molestiae consequatur, vel illum qui dolorem eum fugiat quo voluptas nulla pariatur? Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusantium doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo inventore veritatis et quasi architecto beatae vitae dicta sunt explicabo. Nemo enim ipsam voluptatem quia voluptas sit aspernatur aut odit aut fugit, sed quia consequuntur magni dolores eos qui ratione voluptatem sequi nesciunt. Neque porro quisquam est, qui dolorem ipsum quia dolor sit amet, consectetur, adipisci velit, sed quia non numquam eius modi tempora incidunt ut labore et dolore magnam aliquam quaerat voluptatem.




Name: Tips

Description: What tennis tips can you learn from the pros?

Category: Sports

Source: <http://fatwire.com>



You may not play on the same level as Federer or Nadal, but that doesn't mean you can't learn something from their example. Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusantium doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo inventore veritatis et quasi architecto beatae vitae dicta sunt explicabo. Nemo enim ipsam voluptatem quia voluptas sit aspernatur aut odit aut fugit, sed quia consequuntur magni dolores eos qui ratione voluptatem sequi nesciunt. Neque porro quisquam est, qui dolorem ipsum quia dolor sit amet, consectetur, adipisci velit, sed quia non numquam eius modi tempora incidunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim ad minima veniam, quis nostrum exercitationem ullam corporis suscipit laboriosam, nisi ut aliquid ex ea commodi consequatur? Quis autem vel eum iure reprehenderit qui in ea voluptate velit esse quam nihil molestiae consequatur, vel illum qui dolorem eum fugiat quo voluptas nulla pariatur? Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusantium doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo inventore veritatis et quasi architecto beatae vitae dicta sunt explicabo. Nemo enim ipsam voluptatem quia voluptas sit aspernatur aut odit aut fugit, sed quia consequuntur magni dolores eos qui ratione voluptatem sequi nesciunt. Neque porro quisquam est, qui dolorem ipsum quia dolor sit amet, consectetur, adipisci velit, sed quia non numquam eius modi tempora incidunt ut labore et dolore magnam aliquam quaerat voluptatem.

3. If you wish to experiment with this application (for example assign it to other sites and add users), use WEM Admin. For more information, refer to the *WEM Framework Administrator's Guide*.

Chapter 4

Developing Applications

- [Overview](#)
- [Application Structure](#)
- [Making REST Calls](#)
- [Constructing URLs to Serve Binary Data](#)
- [Context Object: Accessing Parameters from the WEM Framework](#)
- [Registration Code](#)

“Articles” is a Java Web application developed on Spring MVC. The following pages are available:

- `/install.app` is the “Articles” installation page, which also displays a confirmation message when the application is successfully installed
- `/home.app` is the home page of the “Articles” application ([page 26](#)).

Configuration Files

- `applicationContext.xml` (in `/WEB-INF/`) holds SSO and application-specific configurations (such as a predefined user and the site on which to enable the data model and assets).
- `spring-servlet.xml` (in `/WEB-INF/`) is the default Spring configuration file. This file stores the Spring configuration and references the following controllers (described in “[Source Files](#)”):
 - `HomeController`
 - `InstallController`
 - `LayoutController`
 - `ProxyController`
- `log4j.properties` (in `/resources/`) is the logging configuration file. On application deployment, it is copied from `/resources/` to `/WEB-INF/classes/`.

Source Files: `/sample app/articles/src/main/java/`

The `/sample/` folder contains the source files listed below:

- `Configuration.java` is populated (by the Spring framework) from the `applicationContext.xml` file (described in “[Configuration Files](#)”).
- `HomeController.java` is the home page controller, which renders a single home page. This controller reads the list of sample articles from the Content Server platform using the REST API and displays them on the home page.

The sample articles consist of images and text, stored in `/sample app/articles/src/main/resources/install`. The sample articles are installed in Content Server’s database by `InstallController.java`.

- `InstallController.java` registers the “Articles” application, and writes the application’s asset model and sample assets to the database
- `LayoutController.java` displays the application’s layout page (`layout.app`) used by the WEM UI framework. `LayoutController.java` is also used during the application registration procedure.
- `ProxyController.java` delegates AJAX requests to the Content Server REST servlet.
- `TldUtil.java` utility class contains TLD function implementations.

Installer Resources: `/sample app/articles/src/main/resources/install`

The `/install/` folder contains the following resources, used by the `InstallController` to construct the home page ([Figure 8, on page 35](#)):

- `strategies.png`
- `strategies.txt`

- `tips.png`
- `tips.txt`

Home Page Files: `/sample app/articles/src/main/webapp/images`

The `/images/` folder contains:

- `articles.png` icon (Figure 7), which represents the ‘Articles’ application in the banner of the WEM interface
- In Figure 8:
 - `edit.png` is the icon for the **Edit** function
 - `save.png` is the icon for the **Save** function
 - `cancel.png` is the icon for the **Cancel** function

Scripts: `/sample app/articles/src/main/webapp/scripts`

The `/scripts/` folder contains the `json2.js` utility script, used to convert strings to and from JSON objects.

Styles: `/sample app/articles/src/main/webapp/styles`

The `/styles/` folder contains `main.css`, which specifies CSS styles used by this Web application.

Views: `/sample app/articles/src/main/WEB-INF/jsp`

The `/jsp/` folder contains:

- `home.jsp`, which is used to render the home page view of the ‘Articles’ application (Figure 8)
- `layout.jsp`, which defines the application layout

WEB-INF: `/sample app/articles/src/main/WEB-INF`

The `/WEB-INF/` folder contains:

- `articles.tld`, the TLD declaration file
- `spring-servlet.xml`, the Spring configuration file
- `web.xml`, the Web application deployment descriptor

Figure 7: 'Articles' Icon (articles.png)

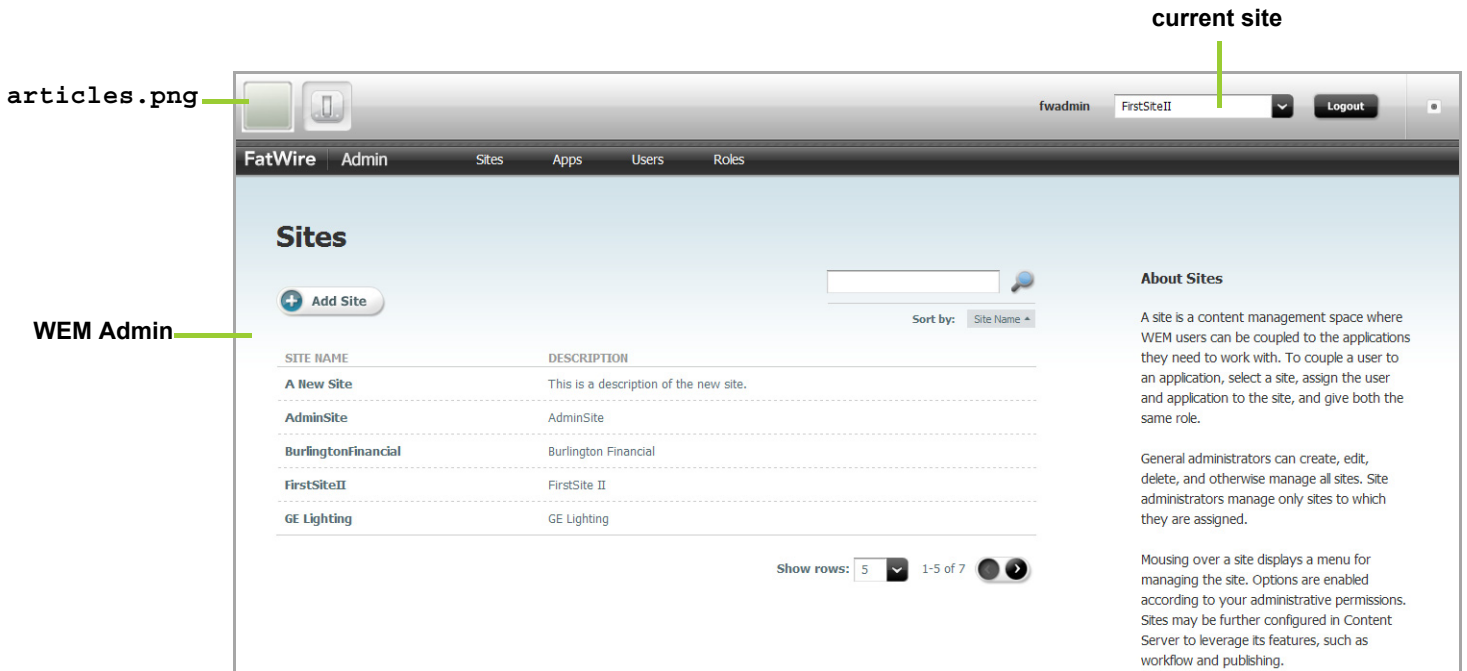
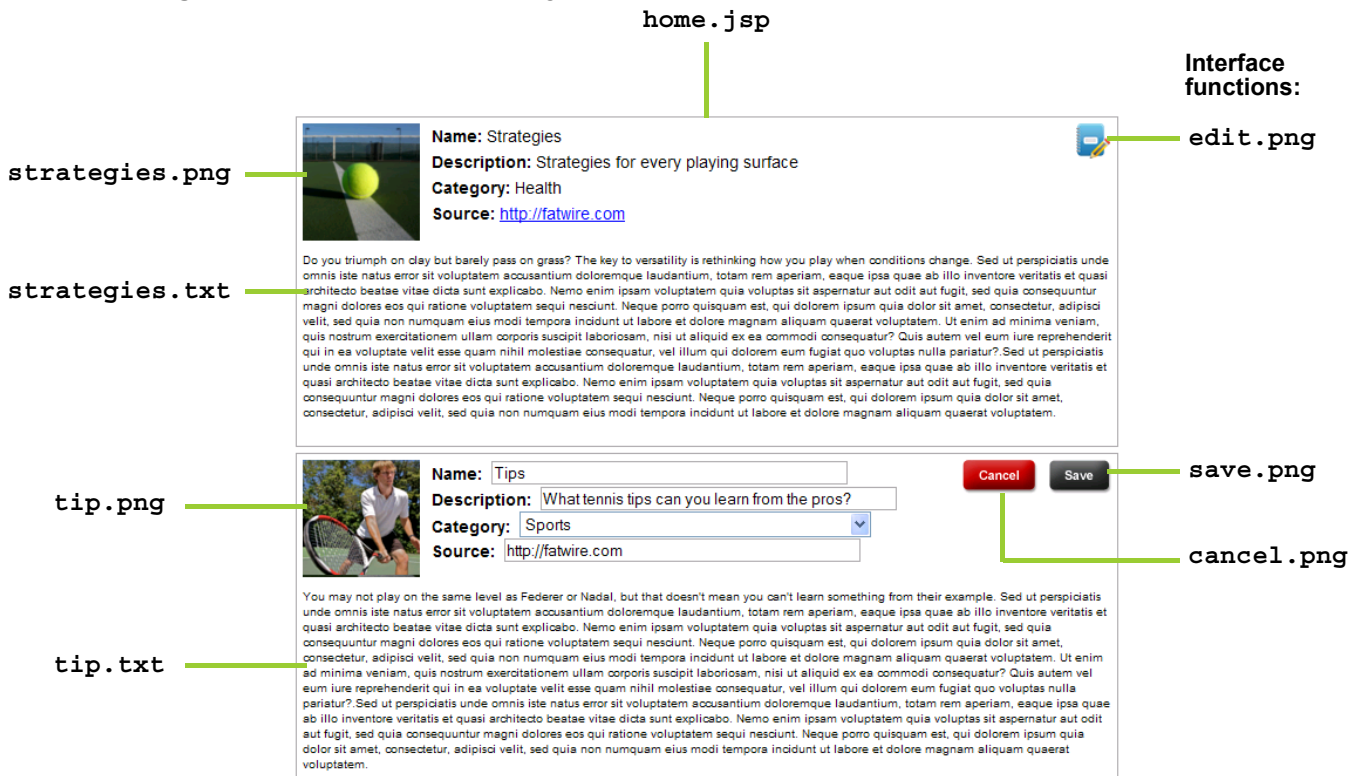


Figure 8: 'Articles' Home Page



Making REST Calls

Content Server REST resources support two types of input and output formats: XML and JSON. To get the desired return formats, you will need to set HTTP headers that specify the MIME type `application/xml` or `application/json`.

For example, when specifying input format to be XML, set `Content-Type` to `application/xml`. When specifying the output format, set `Accept` (the expected format) to `application/xml`. If other output formats are specified, they will be ignored. The default is XML, if not specified in `Content-Type` or `Accept` (for sample code, see [lines 64 and 66 on page 38](#)).

For more detailed information about REST calls, see the following topics in this section:

- [Making REST Calls from JavaScript](#)
- [Making REST Calls from Java](#)

Making REST Calls from JavaScript

The following code (in `home.jsp`) performs AJAX calls to the asset REST services to save asset data. Note that the request is actually performed to the proxy controller which redirects the request to the destination REST service.

Note

We use the JSON stringify library (<http://json.org/js.html>) to serialize a JavaScript object as a string. It is much more convenient to write JSON objects instead of strings.

```
1 // Form the URL pointing to the asset service
2 // to the proxy controller, which will redirect this request to
  the CS REST servlet.
3 var idarr = assetId.split(":");
4 var assetUrl = "${pageContext.request.contextPath}/REST/sites/
  ${config.csSiteName}/types/" + idarr[0] + "/assets/" +
  idarr[1];
5
6 // For the data object to be posted.
7 var data =
8 {
9   "attribute" :
10  [
11    {
12      "name" : "source",
13      "data" :
14        {
15          "stringValue" : document.getElementById("source_e_" +
16            assetId).value
17        }
18    },
19    {
```

```
19     "name" : "cat",
20     "data" :
21     {
22         "stringValue" : document.getElementById("cat_e_" +
                assetId).value
23     }
24 }
25 ],
26 "name" : document.getElementById("name_e_" + assetId).value,
27 "description" : document.getElementById("desc_e_" +
        assetId).value,
28 // TODO: this should be removed.
29 "publist" : "${config.csSiteName}"
30 };
31 // Convert JSON data to string.
32 var strdata = JSON.stringify(data);
33
34 // Perform AJAX request.
35 var req = getXmlHttpRequest();
36 req.onreadystatechange = function ()
37 {
38     if (req.readyState == 4)
39     {
40         if (req.status == 200)
41         {
42             // On successful result
43             // update the view controls with new values and switch the
                mode to 'view'.
44             for (c in controls)
45             {
46                 document.getElementById(controls[c] + "_v_" +
                    assetId).innerHTML =
47                 document.getElementById(controls[c] + "_e_" +
                    assetId).value;
48             }
49             switchMode(assetId, false);
50         }
51         else
52         {
53             // Error happened or the session timed out,
54             // reload the current page to re-acquire the session.
55             alert("Failed to call " + assetUrl + ", " + req.status + " "
                + req.statusText);
56             window.location.reload( false );
57         }
58     }
59 };
60 // We put Content-Type and Accept headers
61 // to tell CS REST API which format we are posting
62 // and which one we are expecting to get.
63 req.open("POST", assetUrl, true);
```

```
64     req.setRequestHeader("Content-Type", "application/
        json;charset=utf-8");
65     req.setRequestHeader("Content-Length", strdata.length);
66     req.setRequestHeader("Accept", "application/json");
67     req.send(strdata);
68 }
```

Making REST Calls from Java

The code below (in `HomeController.java`) calls the assets search service to list all assets of type `FW_Article`. The code uses the Jersey Client library passing objects from the `rest-api-xxx.jar` library provided by WEM. This way we leverage strong typing in Java.

It is important to note that a token must be acquired from Java code by calling the `SSOAssertion.get().createToken()` method. It is unnecessary to do so in JavaScript as that side is already authenticated against WEM SSO.

```
// Use Jersey client to query CS assets.
Client client = Client.create();
String url = config.getRestUrl() + "/types/FW_Article/search";
WebResource res = client.resource( url );

// Construct URL and add token (for authentication purposes)
// and fields (specify which fields to retrieve back) parameters.
res = res.queryParam("fields",
    URLEncoder.encode("name,description,content,cat,source", "UTF-8"));
res = res.queryParam("ticket",
    SSO.getSSOSession().getTicket(res.getURI().toString(),
    config.getCsUsername(), config.getCsPassword()));
// Put Pragma: auth-redirect=false to avoid redirects to the CAS
// login page.
Builder bld = res.header("Pragma", "auth-redirect=false");

// Make a network call.
AssetsBean assets = bld.get(AssetsBean.class);
```

Note

The custom `Pragma: auth-redirect=false` header instructs the CAS SSO filter not to redirect to the CAS sign-in page, but to return a 403 error instead, when no ticket is supplied or the supplied ticket is invalid.

Constructing URLs to Serve Binary Data

The “Articles” application leverages the Blob server in Content Server to serve BLOB data. The following utility function could be used to construct the URL pointing to the

binary data for a given attribute in a given asset, where `blobUrl` points to the Blob server (<http://localhost:8080/cs/BlobServer> by default).

```
public String getBlobUrl(String assetType, String assetId, String
    attrName, String contentType)
    throws Exception
{
    String contentTypeEnc = URLEncoder.encode(contentType,
        "UTF-8");

    return blobUrl + "?" +
        "blobkey=id" +
        "&blobnocache=true" +
        "&blobcol=thumbnail" +
        "&blobwhere=" + assetId +
        "&blobtable=" + assetType +
        "&blobheader=" + contentTypeEnc +
        "&blobheadername1=content-type" +
        "&blobheadervalue1=" + contentTypeEnc;
}
```

An alternative way to get binary data is to load an asset using the resource `/sites/{sitename}/types/{assettype}/assets/{id}`. When loaded, the asset will contain the URL pointing to the BLOB server.

Context Object: Accessing Parameters from the WEM Framework

The UI container provides a JavaScript Context object (`WemContext`) to all applications inside the container. The Context object is used by the applications to get details from the WEM Framework about the logged-in user and site (typically, to get the current site's name from the UI container). The Context object also provides various utility methods that the applications will use to share data. The Context Object can be used by applications running in the same domain as Content Server or in different domains.

Note

The `wemcontext.html` file lists the exposed methods, summarized on [page 42](#).

Same Domain Implementations

To initialize and use Context Object for applications in Content Server's domain:

1. Include `wemcontext.js` ([line 1](#) in the sample code below; `wemcontext.js` is located in `<cs webapp path>/wemresources/js/WemContext.js`).
2. Retrieve an instance of the `WemContext` object ([line 3](#)).
3. Use the methods of `WemContext` ([lines 4 and 5](#)).

Sample Code for Same-Domain Implementations

```

1 <script src='http://<csinstalldomain>/<contextpath>/
  wemresources/js/WemContext.js'></script>
2 <script type="text/javascript">
3   var wemContext = WemContext.getInstance(); // Instantiate
    Context Object
4   var siteName = wemContext.getSiteName(); // Get Site Name
5   var userName = wemContext.getUserName(); // Get UserName
6 </script>

```

Cross-Domain Implementations

To initialize and use Context Object for cross-domain applications:

1. Copy `wemxdm.js`, `json2.js`, and `hash.html` (from the `/Samples` folder) to your application.
2. Open the `sample.html` file and make the following changes to perform cross-domain calls:
 - a. Change the paths of `wemxdm.js` and `json.js` and `hash.html` to their paths in the application (see [lines 1 – 4](#) in the code below).
 - b. Change the path of `wemcontext.html` to its location in Content Server (`wemcontext.html` is located under `/wemresources/wemcontext.html`. Use the Content Server host name and context path. See [line 14](#).)
 - c. In the interface declaration, specify methods that will be used in the framework ([line 15](#)).
 - d. Implement those methods in the local scope and invoke the remote method ([line 30](#)).

sample.html for Cross-Domain Calls

```

1 <script type="text/javascript" src="../../js/wemxdm.js">
  </script>
2 <script type="text/javascript">
3   // Request the use of the JSON object
4   WemXDM.ImportJSON("../js/json2.js");
5   var remote;
6
7   window.onload = function() {
8     // When the window is finished loading start setting up
      the interface
9     remote = WemXDM.Interface(** The channel configuration */
10    {
11      // Register the url to hash.html.
12      local: "../hash.html",
13      // Register the url to the remote interface
14      remote: "http://localhost:8080/cs/wemresources/
        wemcontext.html"
15    }, /** The interface configuration */
16    {
17      remote: {

```



```
18         getSiteName : {},
19 ...
20
21     }
22 }, /**The onReady handler*/ function() {
23     // This function will be loaded as soon as the page is
        loaded
24     populateAttributes();
25 });
26 }
27 </script>
28
29 <script type="text/javascript">
30     /** Define local methods for accessing remote methods
        */
31     function getSiteName() {
32         remote.getSiteName(function(result) {
33             alert("result = " + result);
34         });
35     }
36 ...
37 </script>
```

Methods Available in Context Object

Return Type	Method Name and Description
Object	<code>getAttribute (attributename)</code> Returns attribute value for the given attribute name.
Object	<code>getAttributeNames ()</code> Returns all the attribute names.
Object	<code>getCookie (name)</code> Returns cookie value for the given name. Has all restrictions of the normal browser cookie.
Object	<code>getCookies ()</code> Returns all the cookies.
Object	<code>getLocale ()</code> Returns locale.
Object	<code>getSiteId ()</code> Returns the site id.
Object	<code>getSiteName ()</code> Returns the site name.
Object	<code>getUser ()</code> Returns user object.
Object	<code>getUserName ()</code> Returns user name.
void	<code>removeCookie (name, properties)</code> Removes cookie.
void	<code>setAttribute (attributename, attributevalue)</code> Sets attribute. These attributes can be accessed in other applications.
void	<code>setCookie (name, value, expiredays, properties)</code> Sets the cookie.

Registration Code

Registration exposes applications in WEM, as explained on [page 18](#). Registering an application creates an asset of type `FW_Application` and an asset of type `FW_View` for each view associated with the application. The asset types are enabled on AdminSite. Their attributes are defined in the *REST API Bean Reference*. Programmatic registration is the preferred method. (For an example of manual registration, see [Appendix A](#).)

This section contains the following topics:

- [Registering Applications with an iframe View](#)
- [Registering Applications with JavaScript and HTML Views](#)

Registering Applications with an iframe View

The section uses code from the “Articles” sample application to illustrate the registration process. “Articles” has a single view of type `iframe`. The same steps apply to JavaScript and HTML views.

To register an application

1. Create or get an icon to represent your application. (The icon will be displayed in the applications banner.)

(The “Articles” sample application uses the `articles.png` image file located in: `/sample app/articles/src/main/webapp/images/`)

2. Create a file that specifies the layout of the application in HTML, i.e., for each view, create a placeholder element to hold the content rendered by the view. Applications and views are related as shown in [Figure 9](#), on [page 44](#).

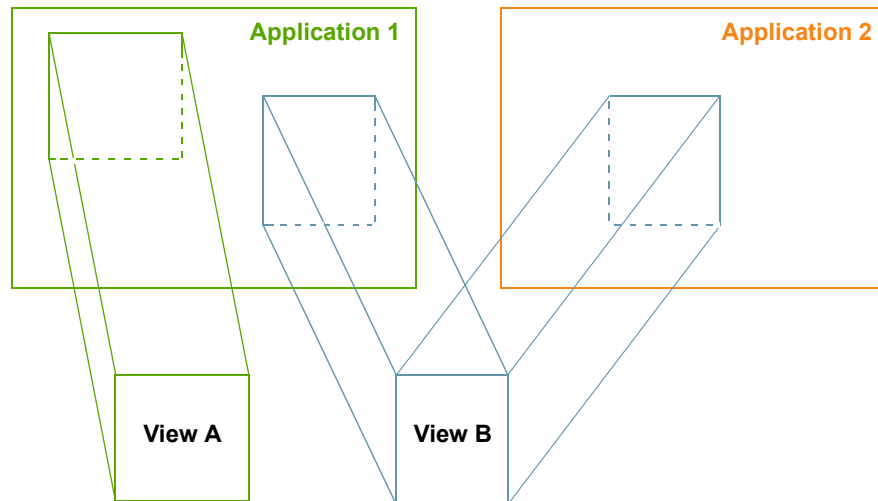
For example, `layout.jsp` for the “Articles” sample application contains the following line:

```
<div id="articles" style="float:left;height:100%;width:100%"  
    class="wemholder"></div>
```

The view’s content will be rendered within the placeholder element when the application is displayed (`layout.app` renders the application’s layout; `home.app` renders the view).

Note

When creating the layout file, specify a unique `id` for the placeholder element. You will specify the same `id` for the `parentnode` attribute when coding the view object. Use `class="wemholder"` for the placeholder elements.

Figure 9: Applications and Views

The relationship between applications and views is many-to-many (Figure 9). One application can have multiple views and each view can be used by many applications. Only registered views can be shared (through their asset IDs). If the asset ID is omitted, the view will be created within the context of its application. In the basic case, an application has only one view associated with it.

3. Invoke the PUT `wem/applications/{applicationid}` REST service and specify your application bean. Populate the bean with the view asset and application asset.

For an iframe view, use the code of the “Articles” sample application, i.e., `InstallController.java` (locate the comment lines `// Create a new view object` and `// Create a new application object`). Set the `layouturl` attribute to specify the URL of the application’s layout page.

In the “Articles” application, the `layouturl` attribute points to the URL of `layout.app` (implemented by `LayoutController.java`):

```
app.setLayouturl(config.getArticlesUrl() + "/layout.app");
```

You can test the results of your registration process by logging in to the WEM Admin interface as a general administrator and selecting **Apps** on the menu bar. Your application should be listed on that page.

Registering Applications with JavaScript and HTML Views

For applications that use HTML and JavaScript views, follow the steps in the previous section, but use the sample code and attributes listed below:

- [JavaScript View](#)
- [HTML View](#)

JavaScript View

Note

JavaScript specified in the view will be rendered (executed) when the application is rendered. Make sure that the JavaScript does not conflict with other views.

Sample code:

```
window.onload = function () {
    if (GBrowserIsCompatible()) {
        var map = new
        GMap2(document.getElementById("map_canvas"));
        map.setCenter(new GLatLng(37.4419, -122.1419), 13);
        map.setUIToDefault();
    }
}
```

- **Rendering the JavaScript view from a source URL**

Set the following attributes:

- name: Name of the view
- parentnode: ID of the placeholder element (from [step 2 on page 43](#))
- viewtype: `fw.wem.framework.ScriptRenderer`, which renders JavaScript into the placeholder element.
- sourceurl: Path of the `.js` file, which provides content for the view. For example: `http://myhost.com:8080/js/drawTree.js`

- **Rendering the JavaScript view from source code**

Set the following attributes:

- name: Name of the view
- parentnode: ID of the placeholder element (from [step 2 on page 43](#))
- viewtype: `fw.wem.framework.ScriptRenderer`, which renders JavaScript into the placeholder element
- javascriptcontent: JavaScript code (sample provided above. The code must not contain `<script>` tags.)

HTML View

Note

HTML specified in the view will be rendered (executed) when the application is rendered.

Sample code:

```
<object width="480" height="385">
  <param name="movie" value="http://www.localhost:8080/jsp/
    flash_slider_main.swf"></param>
  <param name="allowFullScreen" value="true"></param>
  <embed src=" http://www.localhost:8080/jsp/
    flash_slider_main.swf"
    type="application/x-shockwave-flash"
    allowscriptaccess="always" allowfullscreen="true"
    width="480" height="385">
  </embed>
</object>
```

- **Rendering the HTML view from a source URL**

Set the following attributes:

- name: Name of the view
- parentnode: ID of the placeholder element (from [step 2 on page 43](#))
- viewtype: `fw.wem.framework.IncludeRenderer`, which renders JavaScript into the placeholder element
- sourceurl: Path to the HTML file that provides content for the view. For example: `http://myhost.com:8080/js/drawTree.jsp`

- **Rendering the HTML view from source code**

Set the following attributes:

- view: Name of the view
- parentnode: ID of the placeholder element (from [step 2 on page 43](#))
- viewtype: `fw.wem.framework.IncludeRenderer`, which renders JavaScript into the placeholder element
- includecontent: HTML content (sample provided above. The code must not contain `<html>` or `<body>` tags.)

Chapter 5

Developing Custom REST Resources

- [‘Recommendations’ Sample Application](#)
- [Creating REST Resources](#)

'Recommendations' Sample Application

- [Overview](#)
- [Building and Deploying the Application](#)
- [Testing the Application](#)

Overview

The “Recommendations” sample application demonstrates how to create REST resources for Content Server and Satellite Server. The application registers a new REST resource `sample/recommendations/<id>` with GET and POST operations, which allow for retrieval and modification of static list recommendations. The application also demonstrates how it is possible to leverage the Satellite Server caching system.

Building and Deploying the Application

1. The “Recommendations” sample application is located in the `Samples` folder under your Content Server installation directory. Navigate to `recommendations` and edit the `build.properties` file. Specify the correct paths for `cs.webapp.dir` and `ss.webapp.dir` properties.
2. Run Apache ant while in the `recommendations` folder. This will build and deploy your sample application.
3. Launch the `catalogmover` application. Use the **Server > Connect** menu to connect to Content Server. Go to **Catalog > Auto Import Catalog(s)** and select `src\main\schema\elements.zip` file. Append `xceladmin, xceleeditor` when specifying the list of ACLs.
4. Go to the Content Server web application folder. Edit the `WEB-INF/classes/custom/RestResource.xml` file. Uncomment `recommendationService`, `recommendationConfig` and `resourceConfigs` beans.
5. Go to the Satellite Server web application folder. Edit `WEB-INF/classes/custom/RestResource.xml` file. Uncomment `recommendationService`, `recommendationConfig`, and `resourceConfigs` beans.
6. Restart both Content Server and Satellite Server.

Testing the Application

Use the existing static list recommendation id (or create a new recommendation) for the URL `http://<hostname>:<port>/<contextpath>/REST/sample/recommendations/<recommendationid>`. Use the same URL for both Content Server and Satellite Server installations. For example, use `http://localhost:8080/cs/REST/sample/recommendations/1266874492697`. See the XML response for both Content Server and Satellite Server.

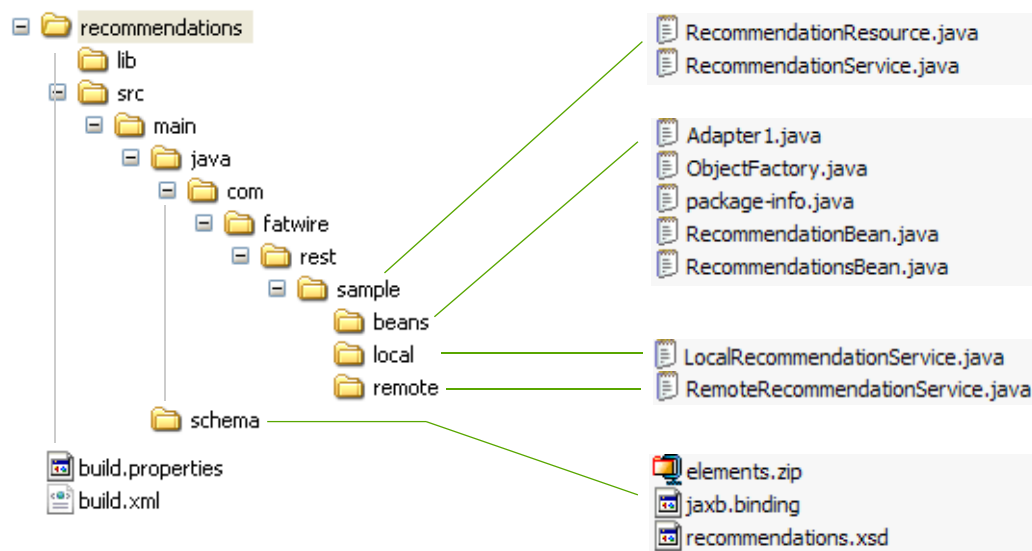
Creating REST Resources

- [Application Structure](#)
- [Steps for Implementing Custom REST Resources](#)

Application Structure

The “Recommendations” sample application was created to guide you through the process of creating your own REST resources.

Figure 10: “Recommendations” Sample Application



- Schema files: `src/main/schema`
 - `elements.zip` contains a sample element, which is used by Satellite Server for caching purposes.
 - `jaxb.binding` is a customization for the default JAXB bindings used during the bean generation process.
 - `recommendation.xsd` is an XML schema for the `RecommendationService` beans.
- Java source files: `src/main/java/.../sample`
 - `RecommendationResource` contains the REST resource implementation. It is used on both Content Server and Satellite Server.
 - `RecommendationService` is an interface that provides the functionality for the `RecommendationResource` class. It is implemented differently, depending on where the resource is hosted: locally (on Content Server) or remotely (on Satellite Server).
 - `beans/*` classes are generated using Java `xjc` compiler. They are pre-packaged with the application. If you want to regenerate beans (i.e., when changing the

recommendation.xsd file) you can run “generate” ant’s task from build.xml.

- LocalRecommendationService is a local (Content Server) implementation for the RecommendationService interface.
- RemoteRecommendationService is a remote (Satellite Server) implementation for the RecommendationService interface.

Steps for Implementing Custom REST Resources

1. Write your XSD file describing your REST service (recommendations.xsd file).
2. Generate beans using the JAXB xjc utility (“generate” ant’s task).
3. Create your REST interface, which will be implemented differently for Content Server and Satellite Server.
4. Implement the REST interface by extending the following classes:
com.fatwire.rest.BaseLocalService
com.fatwire.rest.BaseRemoteService
5. This step is optional in case you decide to leverage Satellite Server caching:
Create elements on the Content Server side, which load the same assets as the local implementation does.
6. Create your REST resource class by extending the com.fatwire.rest.BaseResource class.
7. Register your REST service and configuration in WEB-INF/classes/custom/RestResources.xml file on both Content Server and Satellite Server sides.

The custom/RestResources.xml file contains the following components:

- The only mandatory bean is the bean with resourceConfigs id. The resourceConfigs property contains references to all REST configurations used.

Note

If custom resourceConfigs is uncommented, then embeddedConfig bean should be referenced. Otherwise, the default REST resource, which is provided with the WEM installation will not be registered.

- Resource configurations must be of type com.fatwire.rest.ResourceConfig. Typically only one instance of this class is registered (multiple services can be registered per configuration).

Note

For multiple services, create a new configuration for each disjoint group of your REST services, usually identified by separate XSD files.

- The resourceClasses property contains the list of all resources used.
- beanPackage contains the Java package name specified for the output beans when running the xjc utility.
- schemaLocation is the xsi:schemaLocation attribute to be put in all output XML files produced by your REST service.

Chapter 6

Single Sign-On for Production Sites

- [SSO Sample Application](#)
- [Deploying the SSO Sample Application](#)
- [Application Structure](#)
- [Implementing Single Sign-On](#)
- [Implementing Single Sign-Out](#)

SSO Sample Application

Our SSO sample application is driven by a delivery use case. Given that out-of-the-box CAS cannot be used to secure applications on production sites, we provide a simple example of how to enable single sign-on and sign-out for applications on live sites.

Deploying the SSO Sample Application

1. Unpack the `wem-ss0-api-cas-sample.war` file (to the `/sso-sample` folder, for example). The application is located in Content Server's `/Samples/WEM Samples/WEM Sample applications/` directory.
2. Modify the `applicationContext.xml` file in the `WEB-INF` folder by setting the following properties:
 - `casUrl`: Point to the CAS server base path:
`http://localhost:8080/cas`
 - `casLoginPath`: Include the login form template hosted by the SSO sample application:
`/login?wemLoginTemplate=http%3A%2F%2Flocalhost%3A9080%2Fsso-cas-sample%2Ftemplate.html`
3. Deploy the modified SSO sample application to your application server.
4. Access the application.

The SSO sample application consists of the following pages:

- **Protected area** – a page that is protected by the WEM SSO filter. This page contains two single sign-out links (Figure 11).

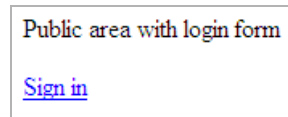
Figure 11: Protected page with single sign-out links



single sign-out links

The first link (single sign-out with redirect) is an HTML link that performs single sign-out on the CAS side and redirects the user back to the home page. The second link (single sign-out without redirect) is also an HTML link that performs single sign-out on the CAS side, but without leaving or reloading the current page.

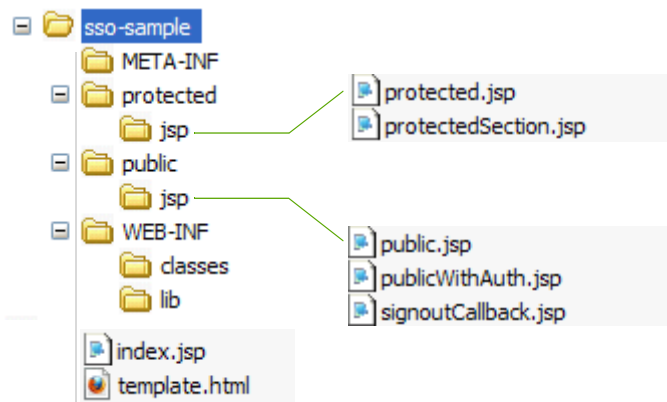
- **Public area** – a page that is excluded from the protection filter.
- **Public area with login form** – this page is excluded from the protection filter, but has a login form, which allows performing a sign-in operation without leaving or reloading the current page.

Figure 12: Public area with “Sign in” link

Application Structure

The SSO sample application provides you with the basic code for utilizing single sign-on and sign-out functionality to protect applications on production sites. The following components provide access to the SSO sample application:

- `index.jsp` – starting page. This page contains links to the pages described as **Protected area**, **Public area**, and **Public area with login form pages** (see “[Deploying the SSO Sample Application](#),” on page 52).
- `template.html` – used to provide a custom sign-in form for CAS. Its path is referenced in the `wemLoginTemplate` parameter in `casLoginPath` in the `applicationContext.xml` file.



Configuration Files: `/sso-sample/WEB-INF`

`WEB-INF` contains the following configuration files:

- `applicationContext.xml` – Spring web application configuration file, which configures the SSO subsystem.
- `web.xml` – web application deployment descriptor.

Protected Files: `/sso-sample/protected/jsp`

Files in this area are protected by the SSO filter. By default, the following files are included in this folder:

- `protected.jsp` – A page protected by the SSO filter. This page hosts two links for performing single sign-out. The first link leads to the CAS sign-out page with a redirect to the application’s home page when sign-out is complete. The second link embeds an `iframe` into this page, which calls the CAS sign-out page with a redirect to

the `signoutCallback.jsp` page. The `protected.jsp` page also prints out all attributes from the `Assertion` object, which describes the current logged in user.

- `protected/jsp/protectedSection.jsp` – Page that is referenced from the `public.jsp` page, when the **Sign in** link is clicked in an embedded iframe. As this page is protected, a login screen is presented in the embedded iframe.

Public Files: `/sso-sample/public/jsp`

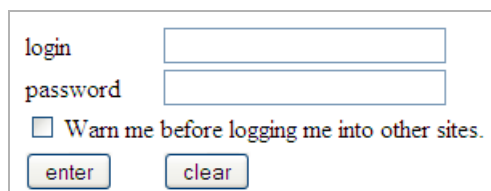
Files in this area are not protected by the SSO filter. By default, the following sample files are included in the `/public/jsp/` folder:

- `public.jsp` – this page not protected by the CAS filter
- `publicWithAuth.jsp` – this page displays the **Sign in** link. Clicking the link embeds an iframe into the `publicWithAuth.jsp` with the iframe pointing to the `protectedSection.jsp` page. As the page is protected, a login screen is presented in the embedded iframe.
- `signoutCallback.jsp` – this page is called from the `protected.jsp` page upon sign-out completion when using iframe.

Implementing Single Sign-On

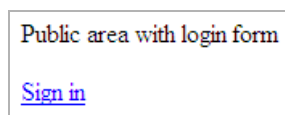
Implementing single sign-on on a web site amounts to implementing a sign-in form. The sign-in form can be presented to site visitors in one of two ways:

- The sign-in form is presented when the visitor tries to access a protected page. This is the default sign-in implementation. This sign in form could be either a default sign-in form shipped with CAS or a custom form provided by an application.



A screenshot of a login form. It contains two text input fields: the first is labeled 'login' and the second is labeled 'password'. Below these fields is a checkbox with the text 'Warn me before logging me into other sites.' To the right of the checkbox are two buttons: 'enter' and 'clear'.

- The sign-in form is embedded into a public page, and the sign-in function is performed without the user leaving the current page. This behavior can be implemented by embedding the iframe that points to a protected page. As the page is being protected, the sign-in form is presented to the visitor.



A screenshot of a public area. It shows the text 'Public area with login form' and a blue, underlined link labeled 'Sign in'.

Implementing Single Sign-Out

When implementing single sign-out on a web page, you can do one of the following:

- Retrieve the “single sign-out” URL by invoking the following method:

```
getSignoutUrl() or getSignoutUrl(String callbackUrl) method of  
com.fatwire.wem.sso.SSO.getSSOSession() object.
```

After performing single sign-out, CAS can optionally redirect to the visitor-supplied URL, which is set in the `callbackUrl` parameter.

- Use an `iframe`-embedding technique if the sign-out is to be performed without leaving the current page. This technique involves embedding an `iframe` with the single sign-out URL as source. When the `iframe` is loaded, the sign-out URL is called (this is done primarily to avoid cross-domain restrictions in browsers).

Chapter 7

Using REST Resources

- [Authentication for REST Resources](#)
- [Configuring CAS](#)
- [REST Authorization](#)
- [Managing Assets Over REST](#)

Authentication for REST Resources

FatWire WEM Framework uses the SSO mechanism built on top of CAS (<http://www.jasig.org/cas>) for authentication purposes. The system behaves differently when the REST API is used from a browser or programmatically.

When accessing the REST API from a browser, the user is redirected to the CAS login page and, upon successful login, back to the original location with the `ticket` parameter, which is validated to establish the user's identity. When accessing the REST API programmatically, the developer must supply either the `ticket` or `multiticket` parameter.

Both the `ticket` and `multiticket` parameters could be acquired by using either the FatWire SSO API if making calls from Java, or simply by using the HTTP protocol if making calls from any other language. The difference between `ticket` and `multiticket` is that a `ticket` is acquired per each REST resource and can be used only once (as the name implies, think of a train or a theater ticket, which is valid for one ride or one play), while a `multiticket` could be used multiple times for any resource. Both the `ticket` and `multiticket` parameters are limited in time, but the typical usage pattern differs. As a `ticket` is acquired per each call, there is no need to worry about its expiration time. However, reusing the same `multiticket` will eventually lead to its expiration and getting an HTTP 403 error. The application must be able to recognize such behavior and fall back to the `multiticket` re-acquisition procedure in such a case. The decision to use either `ticket` or `multiticket` is up to the application developer.

Acquiring Tickets from Java Code

The FatWire SSO API is implemented in an authentication provider-independent manner. Users will not be able to register their own SSO authentication providers. Support for a new authentication provider can be implemented only by FatWire. Switching between providers involves only changing the SSO configuration files.

All SSO calls originate at the SSO front-end class `SSO`. It is used to get the `SSOSession` object. `SSOSession` is acquired per each SSO configuration. It is a single configuration in the web application case, which is loaded using the Spring Web application loader or a configuration loaded from a configuration file in the case of a standalone application.

Web Application

```
SSO.getSession().getTicket(String service, String username, String password)
SSO.getSession().getMultiTicket(String username, String password)
```

Standalone Application

```
SSO.getSession(String configName).getTicket(String service, String username, String password)
SSO.getSession(String configName).getMultiTicket(String username, String password)
```

Acquiring Tickets from Other Programming Languages (Over HTTP)

The CAS REST API is used to acquire a ticket and/or multiticket in the delivery environment. Two HTTP POST calls should be performed to acquire either ticket or multiticket. The difference between ticket and multiticket is that the `service` parameter is “*” for multiticket, while it is an actual REST resource you are trying to access for the ticket parameter.

The example below demonstrates the calls to be made to the CAS server to get a ticket to the `http://localhost:8080/cs/REST/sites` service with `fwadmin/xceladmin` credentials:

1. Call to get Ticket Granting Ticket

Request

```
POST /cas/v1/tickets HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Content-Length: 35
```

```
username=fwadmin&password=xceladmin
```

Response

```
HTTP/1.1 201 Created
Location: http://localhost:8080/cas/v1/tickets/TGT-1-
    ej2biTUFoCNBwA5X4lJn4PjYLRcLtLYg2QhLHclInfQqUk3au0-cas
Content-Length: 441
...
```

2. Call to get a Service ticket

Request

```
POST /cas/v1/tickets/TGT-1-
    ej2biTUFoCNBwA5X4lJn4PjYLRcLtLYg2QhLHclInfQqUk3au0-cas
HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Content-Length: 57
```

```
service=http%3A%2F%2Flocalhost%3A8080%2Fcs%2FRESt%2Fsites
```

Response

```
HTTP/1.1 200 OK
Content-Type: text/plain
Content-Length: 29
```

```
ST-1-7xsHEMYR9ZmKdyNuBz6W-cas
```

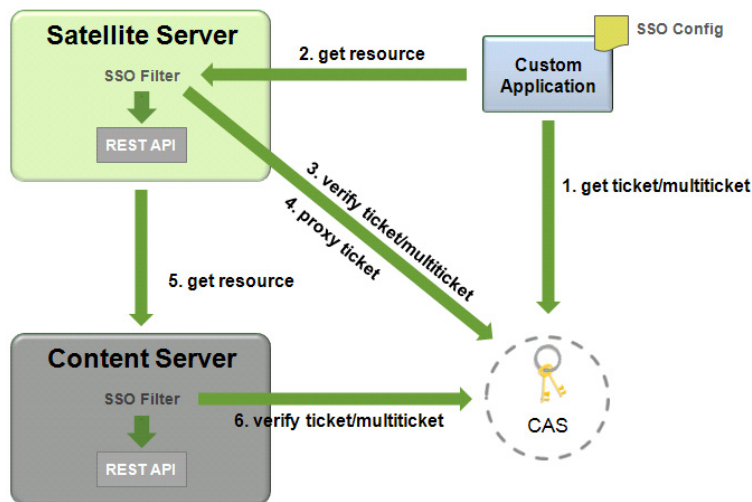
The protocol is fairly straightforward. First a call to get Ticket Granting Ticket (TGT) is made by passing the username and password parameter in `application/x-www-form-urlencoded` POST request. The Response will contain the `Location` HTTP header, which should be used to issue a second `application/x-www-form-urlencoded` POST request with `service` parameter. The response body will contain the actual ticket.

Using Tickets and Multitickets

To use the generated ticket/multiticket, supply the `ticket/multiticket` URL query parameter. For example:

```
http://localhost:8080/cs/REST/sites?ticket=ST-1-7xsHEMYR9ZmKdyNuBz6W-cas
```

```
http://localhost:8080/cs/REST/sites?multiticket=ST-2-Bhen7VnZBERxXcepJZaV-cas
```



1. The application performs a call to get the ticket/multiticket.
 - Input: service, username, password
 - Output: ticket /multiticket
2. The application performs call to Remote Satellite Server to get the resource.
 - Input: ticket, resource input data
 - Output: resource output data
3. Remote Satellite Server performs a call to validate the resulting 'assertion'. The assertion contains user information. Satellite Server also maintains a time-based cache of multitickets, so that subsequent calls do not incur the cost of validation.
 - Input: ticket/multiticket
 - Output: assertion
4. This step is optional. If the `proxyTickets` parameter in the `SSOConfig.xml` file parameter is set to `true` on the Satellite Server side, it also proxies the ticket.
 - Input: ticket
 - Output: proxied ticket
5. Remote Satellite Server performs a call to Content Server.
 - Input: assertion (in serialized form), resource input data
 - Output: resource output data

6. This step is optional. If security is enabled on the Content Server side, it performs a call to validate the ticket.
 - Input: ticket/multiticket
 - Output: assertion

By default the communication channel between Content Server and Remote Satellite Server is not trusted. The `proxyTickets` parameter in the `SSOConfig.xml` file on Remote Satellite Server is set to `true`, which forces Remote Satellite Server to proxy the ticket supplied by the application that is being accessed.

For optimal performance, the system can be configured for authentication by Satellite Server alone. The security check should be disabled on the Content Server side by excluding the REST and Content Server elements used by the REST API from the SSO filter; the `proxyTickets` parameter in the `SSOConfig.xml` file on Remote Satellite Server should be set to `false`. In this mode it is possible to leverage multitickets. Note that the Content Server installation should be hosted inside a private network in this mode, and the communication channel between Content Server and Remote Satellite Server should be trusted.

SSO Configuration for Standalone Applications

The single sign-on module relies on the Spring configuration. The only required bean is `ssoprovider`, which references the `ssoconfig` bean.

Beans and Properties

```
id="ssolistener",
class="com.fatwire.wem.sso.cas.listener.CASListener"
```

Property	Description
No properties for this bean.	

```
id="ssofilter",
class="com.fatwire.wem.sso.cas.filter.CASFilter"
```

Property	Description
<code>config</code>	Required. SSO configuration reference. Sample value: <code>ssoconfig</code>
<code>provider</code>	Required. SSO provider reference. Sample value: <code>ssoprovider</code>

```
id="provider",
class="com.fatwire.wem.sso.cas.CASProvider"
```

Property	Description
config	SSO configuration reference. Sample value: ssoconfig

```
id="config",
class="com.fatwire.wem.sso.cas.conf.CASConfig"
```

Property	Description
applicationProxy CallbackPath	Proxy callback path, relative to casUrl. Default value: /proxycallback
authRedirect	Use this property to specify the default behavior on unauthenticated access to protected pages. <code>true</code> redirects the user to the CAS login page; <code>false</code> displays a 403 error if users are not unauthenticated. This setting could be overridden by the <code>Pragma: auth-redirect</code> HTTP header. Default value: <code>true</code>
casLoginPath	Login page path, relative to casUrl. Can accept additional query parameters: <ul style="list-style-type: none"> <code>wemLoginTemplate</code>, points to the page containing the HTML login template to be used instead of the default template. The template must have two input fields: <code>username</code> and <code>password</code>. Note, that the HTML <code><form></code> tag should not be used in the template. <code>wemLoginCss</code>, points to the CSS page containing style declarations used on the login form. Default value: /login
casRESTPath	CAS REST servlet path, relative to casUrl. Default value: /v1
casSignoutPath	Logout page path, relative to casUrl. Default value: /logout
casUrl	Required property. CAS URL prefix. Example: <code>http://localhost:8080/cas</code>
gateway	If <code>true</code> , the request to protected pages will be redirected to CAS. If a ticket-granting cookie is present, then the user will be implicitly authenticated; if not, the user will be redirected back to the original location. This is used primarily to allow implicit authentication if the user is already logged in to another application.

```
id="config",
class="com.fatwire.wem.sso.cas.conf.CASConfig" (continued)
```

Property	Description
gateway (continued)	<p>Be careful when enabling the redirect behavior to occur by default. Make sure that the clients are able to follow the redirects. Otherwise, <code>gateway=false</code> URL query parameter should be used to override the default behavior. For example, while processing <code>wemLoginTemplate</code> and <code>wemLoginCss</code> parameters, CAS does not follow redirects; you will have to prepend <code>gateway=false</code> to URLs when turning this setting on.</p> <p>Default value: <code>false</code></p>
multiticketTimeout	<p>Multiticket timeout in msec.</p> <p>Default value: <code>600000</code></p>
protectedMappingExcludes	<p>List of mappings that should be excluded. Regular expressions are allowed.</p> <p>Allowed value: See <code>protectedMappingIncludes</code></p>
protectedMappingIncludes	<p>List of protected mappings. Regular expressions are allowed.</p> <p>Allowed value: <code>path?[name=value,#]</code></p> <p>path is a URL path part. It may contain asterisks (<code>*</code> and <code>**</code>). The single asterisk <code>*</code> symbolizes any character sequence up to the forward slash character (<code>/</code>), while <code>**</code> applies to the entire path.</p> <p>Example</p> <p><code>/folder1/folder2</code> matches against <code>/folder1/*</code>, while <code>/folder1/folder2/folder3</code> does not.</p> <p><code>/folder1/folder2</code> matches against <code>/folder1/**</code>, as well as <code>/folder1/folder2/folder3</code>.</p> <p><code>?[...]</code> block is optional. Query parameters can be specified inside the block. Parameters are comma separated. The special character <code>#</code> means that the specified parameters are a subset of those from the request; omitting <code>#</code> requires the request parameters to exactly match the specified parameters.</p> <p>Parameters may contain only <code>name</code>. The match will be done against <code>name</code> only, or against <code>name=value</code> (i.e., both <code>name</code> and <code>value</code>). A parameter can take multiple values. In this case, the match test will pass if any of the specified parameter values match the corresponding parameter value from the request.</p> <p>Example</p> <p><code>/file1[size=1 2]</code> matches against <code>/file1?size=2</code>, but not against <code>/file1?size=2&author=admin</code></p> <p><code>/file1[size=1 2,name=file1,#]</code> matches against <code>/file1?size=2</code> and <code>/file1?size=2&author=admin</code>, but not against <code>/file1?size=3</code></p>

```
id="config",
class="com.fatwire.wem.sso.cas.conf.CASConfig" (continued)
```

Property	Description
protectedMapping Includes (continued)	To make custom REST resources in an application available via remote Satellite Server, specify the following value: /ContentServer? [pagename=rest/<path toCSElement>, #] Example /ContentServer? [pagename=rest/sample/recommendation, #] for custom REST resources in the “Recommendation” sample application (Chapter 5).
proxyTickets	Specifies whether to proxy tickets. Set this property to <code>false</code> for the last server in the call chain for optimal performance. Set this property to <code>true</code> if you need to call another CAS-protected application from this application on behalf of the currently logged-in user. This results in the ability to call the following method: <code>SSO.getSSOSession().getTicket(String service, String username, String password)</code> Default value: <code>true</code>
useMultiTickets	Specifies whether to use multitickets. Default value: <code>true</code>

Query Parameters Processed by SSO Filter

Query Parameters Processed by SSO Filter

Property Name	Description
ticket	Used to verify user identity. Can be used only during some limited period of time for one resource and only once. Type: <query parameter> Value: <random string>
multiticket	Used to verify user identity. Can be used only during some limited period, multiple times for any resource. Type: <query parameter> Value: <random string>

Query Parameters Processed by SSO Filter (*continued*)

Property Name	Description
gateway	<p>If this property is set to <code>true</code>, the request for public pages will be redirected to CAS. If the ticket granting cookie is present, then the user will be implicitly authenticated; if not, the user will be redirected back to the original location. This is primarily to allow implicit authentication if the user is already logged in to another application.</p> <p>Type: <query parameter> Value: <code>true</code> <code>false</code></p>
auth-redirect	<p>Used to specify the default behavior on unauthenticated access to protected pages. If this property is set to <code>true</code>, the user will be redirected to the CAS login page; if <code>false</code>, a 403 error will be presented.</p> <p>Type: <Pragma HTTP header> Value: <code>true</code> <code>false</code></p>

Configuring CAS

Information about CAS clustering can be found in the following sources:

- For information about CAS architecture, use the following link:
<http://www.jasig.org/cas/about>
- For information about CAS clustering, see the *Content Server Rollup Guide*.
- For information about configuring CAS with LDAP providers, use the following link:
<http://www.jasig.org/cas/server-deployment/authentication-handler>

REST Authorization

This section is for developers who are interested in administrators' authorization processes. REST authorization is the process of granting privileges to perform REST operations on applications' resources (which map to objects in Content Server). REST authorization uses the "deny everything by default" model. If a privilege is not explicitly granted to a particular group, that privilege is denied.

Security Model

The WEM security model is based on objects, groups, and actions. Security must be configured per object type in Content Server's Advanced interface:

Objects of a given type are accessible to a user only if the user belongs to at least one group with privileges to perform specified actions on objects of the given type.

The Security Configuration node is accessible from the Admin tab in CS Advanced.

- **Object** is a generic term that refers to any entity in WEM such as a site, a user, or an asset. Protected objects are of the following types:
 - Asset Type
 - Asset
 - Index
 - Site
 - Role
 - User
 - User Locale
 - ACL
 - Application
 - **Security groups** are used to gather users for the purpose of managing their permissions (to operate on objects) simultaneously.
 - An **action** is a security privilege: LIST, READ, UPDATE, CREATE, DELETE. LIST provides GET permission on services that list objects (such as /types), whereas READ provides GET permission on services that retrieve individual objects in full detail (such as /types/{assettype}).
- Privileges are assigned to groups to operate on allowed objects. Some objects, such as ACLs, are read-only (they can be created directly in Content Server, but not over REST).

A security configuration is an array, such as shown above, which specifies:

- The protected object type and object(s)

- Groups that are able to access the objects
- Actions that groups (and their members) can perform on the objects

Possible security configurations are summarized in the *WEM Framework Administrator's Guide*.

Using the Security Model to Access REST Resources

Object types and objects in Content Server map to REST resources in WEM. For example, the `Asset Type` object maps to:

- `<BaseURI>/types/ resource` (which lists all asset types in the system)
- `<BaseURI>/types/<assettype> resource` (which displays information about the selected asset type), and so on.

Actions in Content Server map to REST methods in WEM. For example, granting the `READ` privilege to group `Editor` to operate on asset type `Content_C` gives users in the `Editor` group permission to use `GET` and `HEAD` methods on the REST resource `/types/Content_C`.

- The `LIST` action allows group members to use `GET` methods on REST resources.
- The `READ` action allows group members to use `GET` and `HEAD` methods on REST resources.
- The `UPDATE` action allows group members to use `POST` methods on REST resources.
- The `CREATE` action allows group members to use `PUT` methods on REST resources.
- The `DELETE` action allows group members to use `DELETE` methods on REST resources.

For comprehensive information, see the *REST API Resource Reference*.

Configuring REST Security

Procedures for configuring REST security are available in the *WEM Framework Administrator's Guide*.

Privilege Resolution Algorithm

When configuring a security privilege, you can specify that the privilege applies to all objects of a certain type or a single object of a certain type. For example, granting the privilege to `UPDATE (POST)` any site allows users in the group to modify the details of all sites in WEM. Granting the privilege to `UPDATE (POST)` the `FirstSiteII` sample site allows users in the group to modify this site's details in WEM.

The `Asset` object type requires you to specify the site to which the security setting applies, as assets are always accessed from a particular site. The `AssetType` object can be extended by specifying a subtype, which is used to make the security configuration more granular. For example, setting the `DELETE` privilege on asset type `Content_C` in allows a `DELETE` request to be performed on the REST resource `/types/Content_C` (i.e., to delete the `Content_C` asset type from the system).

Because privileges can be granted only to groups, a user's total privileges are not obvious until they are computed across all of the user's groups. WEM provides a privilege resolution algorithm. Its basic steps are listed below:

1. REST finds the groups in which the user has membership.
2. REST determines which groups can perform which REST operations on which REST resources. If site or subtype is specified, each is taken into account.
3. REST compares the results of steps 1 and 2. If at least one of the groups from step 1 is in the list of groups from step 2, then access is granted. Otherwise, access is denied.

Managing Assets Over REST

Sample code illustrating management of assets via the Content Server REST API is available in your Content Server installation directory, in the following paths:

```
Samples/WEM Samples/REST API samples/Basic Assets/com/fatwire/  
rest/samples/basic/
```

```
Samples/WEM Samples/REST API samples/Basic Assets/com/fatwire/  
rest/samples/flex/
```

The subfolders `basic` and `flex` each contain the following set of files:

- `CreateAsset.java`
- `DeleteAsset.java`
- `ReadAsset.java`
- `UpdateAsset.java`

The code is richly documented with step-by-step instructions. Examples of basic asset management use the `HelloAssetWorld` sample site. Examples of flex asset management use the `FirstSite II` sample site. All information regarding the required asset types and assets can be found in the `java` files.

Chapter 8

Customizable Single Sign-On Facility

This chapter provides information about the FatWire Customizable Single Sign-On facility, which enables developers to implement custom authentication behavior. For example, you can use an external authentication authority to authenticate Content Server users.

This chapter contains the following sections:

- [Customizing Login Behavior for the WEM Framework](#)
- [Configuring and Deploying Custom SSO Behavior](#)
- [Running the CSSO Sample Implementation](#)

Customizing Login Behavior for the WEM Framework

WEM Framework authentication, which is built over the CAS framework, includes a customization layer called the FatWire Customizable Single Sign-On facility, also called *CSSO*. The *CSSO* facility contains authentication extensions that you can use to create a custom SSO solution, without directly modifying the CAS configuration. Instead, the Spring configuration directs the injection of these extensions into the CAS configuration to implement the desired login behavior.

The *CSSO* facility provides pre-packaged classes that can be extended to implement a custom SSO solution. It also provides a default Spring configuration file which identifies the classes to Spring for instantiation. Customizing WEM SSO enables you to use a different login screen, require credentials other than a username/password pair, or use an external authentication authority to authenticate Content Server users. A custom SSO implementation consists of:

- Three Java classes (which extend the default classes)
- A configuration file that exposes the new classes to the framework

The default *CSSO* classes defer all credential discovery and authentication to the standard WEM SSO implementation. These classes are instantiated by the `customdefaultWEMSSObean.xml` Spring configuration file. Extending the default *CSSO* classes enables you to define methods which specify the behavior of your custom SSO solution. For example you can create a different authentication for browser access, REST, and/or thick client authentication. When you extend the default *CSSO* classes, you must create a custom Spring configuration file that identifies the custom classes and exposes them to the WEM Framework.

The *CSSO* facility provides a complete SSO sample (including Java source files) that replaces the default WEM login behavior with custom login behavior. The sample SSO implementation demonstrates two different types of authentication – username/password pair (with an additional domain field) and external user identifier. (The external identifier maps a user authenticated by an external authentication authority to a Content Server system user.)

The rest of this chapter provides information about the default components of the *CSSO* facility and instructions on implementing a custom SSO solution. If you wish to see an example of a custom SSO solution, the end of this chapter provides information about the *CSSO* sample, and instructions for running the sample.

Components of the Default CSSO Implementation

This section provides information about the default components provided by the CSSO facility. These components are your starting point for customizing your own SSO implementation.

The `com.fatwire.wem.sso.cas.custom.basis` package (shown in [Table 1](#)) contains the default classes that are included in the CSSO facility. The default Spring configuration file (`customdefaultWEMSSObeans.xml`) instantiates these classes to implement the default WEM login behavior.

Note

The CSSO facility provides a complete SSO sample that replaces the default WEM login behavior with custom login behavior. For more information, see [“Running the CSSO Sample Implementation,”](#) on page 81.

Table 1: `com.fatwire.wem.sso.cas.custom.basis`

Class	Description
<code>CustomAuthenticator.java</code>	<p>Implements the <code>CustomAuthentication</code> interface. This class controls the behavior of the login sequence and handles authentication requests. By default, it returns to WEM to complete the authentication by displaying the standard WEM login form.</p> <p>For information about extending this class, see “Extending the Default CSSO Classes,” on page 73.</p>
<code>CustomConfiguration.java</code>	<p>Provides access to the properties that are set in the default Spring configuration file. You can extend this class when additional properties are required for a custom SSO implementation.</p> <p>For information about extending this class, see “Extending the Default CSSO Classes,” on page 73.</p>
<code>CustomCredentials.java</code>	<p>Provides a standard set of credential values for custom authentication. You can extend this class when additional attributes are needed for a custom SSO implementation.</p> <p>For information about extending this class, see “Extending the Default CSSO Classes,” on page 73.</p>

The `com.fatwire.wem.sso.cas.custom.interfaces` package (shown in [Table 2](#)) defines the custom authentication interfaces.

Table 2: `com.fatwire.wem.sso.cas.custom.interfaces`

Class	Description
<code>CustomAuthentication.java</code>	Defines the interfaces that must be implemented by any custom SSO solution. For more information, see “Extending the Default CSSO Classes,” on page 73.
<code>CustomRestCodec.java</code>	Defines the interfaces that must be implemented to encode and decode a custom REST authentication token that is not username/password based.

Configuring and Deploying Custom SSO Behavior

To configure and deploy custom SSO behavior you must first extend the default classes that are included in the CSSO facility. You then identify the new Java classes to Spring by creating a custom Spring configuration file which instantiates the classes, exposing them to the CSSO framework. These are your basic steps:

1. Extend the default CSSO classes – `CustomAuthenticator.java`, `CustomConfiguration.java`, and `CustomCredentials.java` (contained within the `com.fatwire.wem.sso.cas.custom.basis` package):
 - a. Create new Java classes that extend the default CSSO classes.
 - b. Package the Java classes you created in a `jar` file, then place the `jar` file in the classpath of the CAS servlet (in `cas/WEB-INF/lib`).

For more information, see [“Extending the Default CSSO Classes,”](#) on page 73.

2. Identify your new Java classes to Spring for instantiation:
 - a. Create a Spring configuration file that contains all the custom class names and properties for your SSO implementation.
 - b. Place the custom Spring configuration file in the `spring-configuration` folder (in `cas/WEB-INF/`).
 - c. Remove the `.xml` extension from the default Spring configuration file (`customDefaultWEMSSObeans.xml`).

For more information, see [“Identifying Your Java Classes to Spring for Instantiation,”](#) on page 75.

3. If an external authentication authority is used to authenticate a user, map the external user identifier to the appropriate Content Server system user name, unique identifier, and ACLs. For instructions, see [“Mapping External User Identifiers to Content Server Credentials,”](#) on page 78.
4. Restart the CAS web application. For more information, see [“Restarting the CAS Web Application,”](#) on page 80.

The rest of this section provides detailed information for the steps outlined above.

Extending the Default CSSO Classes

An SSO implementation is a set of called methods that are specified in the default CSSO classes `CustomAuthenticator.java`, `CustomConfiguration.java`, and `CustomCredentials.java`. To replace the default WEM login behavior with custom behavior, you must create new Java classes for it that extend the default CSSO classes. By extending the CSSO classes, the methods specified in the default CSSO classes are replaced by the methods specified in the custom classes for the functionality you wish to change.

The three classes (located in the `com.fatwire.wem.sso.cas.custom.basis` package) that must be extended to implement a custom SSO solution are:

- **CustomConfiguration.java** – Provides access to the externally defined properties that are specified in the default Spring configuration file. By default, this class exists only as a placeholder for injecting properties into the SSO configuration from the Spring configuration file. Extend this class if you wish to include additional properties, such as URLs or other configuration information, that are specific to your custom SSO implementation.
- **CustomCredentials.java** – Provides a standard set of credential values for custom authentication. This class is built and populated by the web-flow handler or the custom REST authenticator. By default, this class defines the standard `UsernamePasswordCredentials` object (provided by CAS), which collects all information required to complete user authentication in the following properties – `username`, `userId`, and `currentACL`. The values of these properties populate the attributes map used by the authenticator (`CustomAuthenticator.java`), to perform the actual user authentication.

Extend this class if you wish to require additional credentials for your custom SSO solution. For an example of how this class passes user information to the authenticator to complete user authentication, refer to the code of the sample CSSO class `SampleCredentials.java` (located in the `ContentServer/Samples/WEM/Samples/CustomizableSSO/lib` folder).

- **CustomAuthenticator.java** – Implements the `CustomAuthentication` interface. This class controls the behavior of the login sequence and handles authentication requests. By default, it returns to WEM to complete the authentication by displaying the standard WEM login form.

Note

The default `CustomAuthenticator.java` class is the most important class because it contains all the authentication methods for an SSO implementation.

All authentication decisions and CAS web-flow actions are directed to this class for action. CAS web-flow performs a number of steps, one of which invokes the `performLoginAction` method. This method displays a login form or communicates with an external authentication authority.

This class also defines the static method `callCsResolverPage` which maps an external user to a Content Server user. If your custom SSO implementation uses an external authentication authority to authenticate users, the `callCsResolverPage` method must define the unique name for the CSSO authenticator. For more

information, see [“Mapping External User Identifiers to Content Server Credentials,” on page 78.](#)

The following is a complete interface description of the methods this class implements:

```
static final int SUCCESS = 0;
static final int GOTOWEM = 1;
static final int FAILURE = 2;
static final int REDIRECT = 3;
static final int ERROR = 4;
static final int REPEAT = 5;

/**
 * Called from UserAuthentication handler to check for
 * alternate
 * credentials and validate appropriately.
 * @param userCredentials
 * @return
 */
public int authenticate(com.fatwire.wem.sso.cas.custom.basis.
    CustomCredentials userCredentials);

/**
 * Called from CSAAuthenticationHandler to check for REST user
 * credentials and validate appropriately.
 */
public int authenticateRest(UsernamePasswordCredentials
    restCredentials);

/**
 * Called from CSAAuthenticationHandler to check is username/
 * password
 * combination is detected.
 */
public boolean checkRestCredentials(String token);

/**
 * Called from CSAttributeDAO to check for encoded credentials
 * and
 * if so then return the correct username for DAO processing.
 * @param username
 * @return
 */
public String resolveRestUsername(String username);

/**
 * Called from LoginViewAction to handle login view processing.
 * This
 * method allows the calling of internal CAS methods.
 * @param context
 * @param userAuthentication
 * @param centralAuthenticationService
```

```
* @return
*/
public int performLoginAction(RequestContext context,
    CustomAuthentication userAuthentication,
    CentralAuthenticationService
    centralAuthenticationService);

/**
 * Called from casLogoutView to perform sign in cleanup
 * @param request
 * @param response
 */
public void performLogoutAction(HttpServletRequest request,
    HttpServletResponse response);
```

Identifying Your Java Classes to Spring for Instantiation

All customization settings for an SSO implementation are specified in a single Spring configuration file, located in the `spring-configuration` folder (in `cas/WEB-INF`).

The rest of this section contains the following topics:

- [Creating a Spring Configuration File](#)
- [Placing Your Spring Configuration File](#)

Creating a Spring Configuration File

The classes and properties for the default SSO implementation are defined by the Spring configuration file `customDefaultWEMSSObeans.xml`, which is located in the `spring-configuration` folder (in `cas/WEB-INF`). When customizing CSSO, you can either create a new Spring configuration file or customize the classes and properties referenced in the default Spring configuration file. The rest of this section focuses on the second option.

The default Spring configuration file contains several bean identifiers that reference the classes and properties required for the default SSO implementation. The `customUserConfiguration` bean references the `CustomConfiguration.java` class and the `customUserAuthenticator` bean references the `CustomAuthenticator.java` class. These classes are instantiated by the Spring configuration file, which uses them to create the persistent objects for the SSO implementation's authentication process. To create a custom SSO solution, you must reference your custom Java classes within these beans.

Note

The `CustomCredentials.java` class is **not** referenced by the Spring configuration file. Instead, you provide the code that instantiates this object in the `performLoginAction` method, defined in the default CSSO `CustomAuthenticator.java` class. This method creates a custom credentials object for every login request and passes it into CAS for authentication.

The `customUserConfiguration` bean also identifies the configuration properties which supply system information to the default SSO implementation. These properties are set with values of the environment on which you are deploying the SSO implementation. When you customize the Spring configuration file, you must modify the values of the properties to match the custom SSO implementation's environment, or include additional properties required by the custom SSO implementation.

Extending the `CustomConfiguration.java` class enables you to define additional properties in the Spring configuration file's `customUserConfiguration` bean. For example, if you created a JSP file that provides a custom login form for your SSO implementation, create a property that specifies the location of the JSP file by extending the `CustomConfiguration.java` class.

The rest of this section analyzes the classes and properties that are referenced in the default Spring configuration file (`customDefaultWEMSSObean.xml`).

The default Spring configuration file

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <beans xmlns="http://www.springframework.org/schema/beans"
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-
4          instance"
5      xmlns:flow="http://www.springframework.org/schema/
6          webflow-config"
7      xmlns:p="http://www.springframework.org/schema/p"
8      xsi:schemaLocation="http://www.springframework.org/
9          schema/beans http://www.springframework.org/schema/
10         beans/spring-beans-2.0.xsd
11         http://www.springframework.org/schema/webflow-config
12         http://www.springframework.org/schema/webflow-
13         config/spring-webflow-config-1.0.xsd">
14  <!-- Custom SSO Bean definitions. This file defines either
15         the default CAS/SSO configuration or a special
16         user implementation. No other CAS configuration
17         files are modified for a custom implementation -->
18  <!-- This bean is never modified. It defines the web-flow
19         controller which always passes control into
20         the custom authenticator -->
21  <bean id="customUserLoginAction"
22         class="com.fatwire.wem.sso.cas.web.CustomLoginViewA
23         ction"
24         p:centralAuthenticationService-
25             ref="centralAuthenticationService"
26         p:customAuthentication-
27             ref="customUserAuthenticator"
28     />
29  <!-- This bean is usually not modified. Override it when
30         there needs to be a custom encoding for
31         information passed between the web-flow and any
32         external component -->
33  <bean id="customRestCoder"
34         class="com.fatwire.wem.sso.cas.custom.basis.CustomRest
35         TokenCoding"
36     />
```

```
17 <!-- Modify this bean with a custom configuration
    implementation class when additional parameters are
    needed for a custom implementation -->
18 <bean id="customUserConfiguration"
    class="com.fatwire.wem.sso.cas.custom.basis.CustomConf
    uration"
19     p:casLoginUrl="http://localhost:7080/cas/login"
20     p:resolverUrl="http://localhost:8080/cs/custom/
    customCsResolver.jsp"
21     p:resolverUsername="fwadmin"
22     p:resolverPassword="xceladmin"
23     p:traceFlag="false"
24     />
25 <!-- Modify this bean with a customAuthentication class for
    a custom implementation. -->
26 <bean id="customUserAuthenticator"
    class="com.fatwire.wem.sso.cas.custom.basis.CustomAuth
    enticator"
27     p:customConfiguration-ref="customUserConfiguration"
28     p:customRestCoder-ref="customRestCoder"
29     />
30 </beans>
```

Analyzing the default Spring configuration file

- **Line 18** is the `customUserConfiguration` bean, which references the default CSSO `customConfiguration.java` class. (For information about this class, see [“Extending the Default CSSO Classes,”](#) on page 73.) This bean also contains the required properties for the default SSO implementation:
 - **Line 19** references the `casLoginURL` property. This property specifies the base URL to the CAS login function. The domain and port number settings require modification if the values specified are different from the CAS server installation.
 - **Lines 20 – 22** reference the external authentication properties – `resolverURL`, `resolverUsername`, and `resolverPassword`. If Content Server is used to authenticate users, these properties do not need to be referenced. If an external authentication authority is used to authenticate users, these properties must be referenced. When these properties are referenced they enable you to implement mapping from an external identifier to a Content Server system user.
 - **Line 20** references the `resolverURL` property. If an external authentication authority is used to authenticate a user, this property must specify the full URL to the `customCsResolver` page, located on Content Server. The `customCsResolver` page obtains a user’s external identifier and queries the Content Server database to retrieve the user’s Content Server credentials. The domain and port number specified in this property, must be modified if the values specified are different from the Content Server installation.
 - **Line 21** references the `resolverUsername` property. If an external authentication authority is used to authenticate a user, this property must specify the username of a Content Server user who has permissions to read the `SystemUserAttr` table. This username is used when the `customCsResolver` page needs to query the Content Server database to resolve an external user identifier into a registered Content Server user.

- [Line 22](#) references the `resolverPassword` property. If an external authentication authority is used to authenticate a user, this property must specify the password of the user identified by the `resolverUsername` property ([line 21](#)).

For information about implementing mapping, see “[Mapping External User Identifiers to Content Server Credentials](#),” on page 78.

- [Line 23](#) references the `traceFlag` property. This property specifies whether the trace log, which provides information about the custom SSO layer, is enabled or disabled. This property can either be set to `True` or `False`.
- [Line 26](#) is the `customUserAuthenticator` bean, which references the default CSSO `CustomAuthenticator.java` class. (For information about this class, see “[Extending the Default CSSO Classes](#),” on page 73.)

Placing Your Spring Configuration File

The default Spring configuration file, which specifies the classes and properties for the default WEM login behavior, is located in the `spring-configuration` folder (in `cas/WEB-INF`). Placing your own file into the same location requires deactivating the default file (by removing or changing the file’s `.xml` extension). This is because Spring loads all Spring configuration files contained in the `spring-configuration` folder (in `cas/WEB-INF`) and merges those files into a single configuration. As both the custom and the default files specify the same bean identifiers, only one of the files can be recognized by the Spring configuration. Duplicate bean identifiers result in initialization failure.

Note

Avoid deleting `customDefaultWEMSSObeans.xml`. Instead, remove or change the file’s `.xml` extension. This way you can restore the file if you wish to return to using the default WEM login screen.

Mapping External User Identifiers to Content Server Credentials

The CSSO facility enables you to use an external authentication authority to authenticate Content Server users. When the external authentication authority validates the user’s credentials, it associates a unique external identifier with that user. To complete WEM authentication, the user’s external identifier must be mapped to the corresponding Content Server system username, unique identifier, and ACLs by using the method `callCsResolverPage` (defined as a static method in the default CSSO class `CustomAuthenticator.java`).

To map an external identifier to a Content Server system user, make sure you have set the external authentication properties in the Spring Configuration file (see, “[Analyzing the default Spring configuration file](#),” on page 77). To implement mapping from an external identifier to the appropriate Content Server system credentials, do the following:

To implement mapping

1. Define a unique CSSO authenticator name for the external authentication authority of your custom SSO implementation in the `callCsResolverPage` method (defined in your extended `CustomAutheticator.java` class).

For example, the following `callCsResolverPage` method (defined in the Sample CSSO class `SampleAutheticator.java`) defines "sampleSSO" as the unique authenticator name:

```
Map<String,String>csTokens=callCsResolverPage(externalUserId,
    "sampleSSO")
```

2. Access the Content Server Advanced interface as a general administrator (for example, `fwadmin/xceladmin`).
3. In the **Admin** tab, expand the **Content Server Management Tools** node and double-click **User**.
4. Select the user whose external identifier you wish to map to Content Server credentials:
 - a. In the "Enter User Name" field, enter the name of the user.
 - b. In the "Select Operation" section, select the **Modify User Attributes** radio button.
 - c. Click **OK**.

The "Modify User" form is displayed:

Modify User	
Select the user to modify:	
User Name	ACL
fwadmin	TableEditor, Visitor, VisitorAdmin, UserEditor, UserReader, xceladmin, Browser, xceleeditor, xcelpublish, PageEditor, ElementEditor, RemoteClient, WSUser, WSEditor, WSAdmin

5. In the "User Name" column, click the name of the user whose external identifier you wish to map to Content Server credentials.

The following form is displayed:

User Attributes:UTF-8	
Attribute Name	Attribute Values
<input type="text"/>	<input type="text"/> <small>Add new attribute and value .</small>
<input type="button" value="Modify"/>	

6. In the form, fill in the fields:
 - In the "Attribute Name" field, enter the unique CSSO authenticator name (the name used to identify the external authentication authority). This name must match the unique name of the CSSO authenticator defined in the `callCsResolverPage` method (in [step 1](#)).
 - In the "Attribute Values" field, enter the user's external identifier provided by the external authentication authority.
7. Click **Modify** to store the new attribute and value in the Content Server `SystemUserAttr` database table.
8. Repeat [steps 3 – 7](#) for all users associated with an external identifier.

Analyzing the Mapping Process

When the `callCsResolverPage` method is called to map an external identifier to a Content Server system user, it defines the unique CSSO authenticator name for your custom SSO implementation. The method uses the external identifier and the unique CSSO authenticator name to map the external user to the Content Server system user. This map contains the following items, which are placed in the associated properties of the `CustomCredentials` object:

- `username` – The user's Content Server username.
- `currentUser` – The user's Content Server unique identifier.
- `currentACL` – The user's ACLs.

The `CustomCredentials` object passes the `username`, `currentUser`, and `currentACL` values to the `authenticate` method, defined in the `CustomAuthenticator.java` class. The `authenticate` method uses these values to build the response map, which identifies the Content Server user.

Restarting the CAS Web Application

To deploy your custom SSO implementation, restart the CAS web application. Once CAS has been restarted, it uses the classes defined in the custom Spring configuration file, located in the `spring-configuration` folder (in `cas/WEB-INF`) to provide the custom login behavior.

Running the CSSO Sample Implementation

The CSSO facility provides a working example of a custom SSO implementation (including Java source files). The sample replaces the default WEM login behavior with custom login behavior, which includes the standard username and password fields, an additional field for a user to specify a domain name, and a field for an external user identifier. This demonstrates two different types of authentication – username/password pair (with an additional domain field) and user authentication through an external authentication authority.

Note

The CSSO sample does not enforce any validation rules that apply to the fields on the login form. Fields are not checked for completeness and incorrect values are not reported. If authentication fails, the form is re-displayed without comment. If you implement this form in a production environment, you must ensure that all rules are enforced with suitable diagnostic messages if an error occurs.

For information about all the sample components included in the CSSO facility, see [“Sample CSSO Components,” on page 85](#).

To run the sample SSO implementation

1. Deploy the `customizable-sso-1.0.jar` (`ContentServer/Samples/WEM/Samples/CustomizableSSO`) by placing it in the CAS classpath (`cas/WEB-INF/lib` folder). This file contains the sample CSSO classes.
For more information about the sample classes, see [“Sample CSSO Classes,” on page 82](#).
2. Create a `fatwire` folder in the CAS web application context folder. Copy the `SampleLoginform.jsp` file into the `fatwire` folder.
3. Identify the classes contained in the `customizable-sso-1.0.jar` file to Spring for instantiation:
 - a. Copy the `customSampleSSObeans.xml` configuration file into the `spring-configuration` folder.
 - b. Modify the properties in the `customSampleSSObeans.xml` file to match your operation environment.
 - c. Remove the `.xml` extension from the `customDefaultWEMSSObeans.xml` configuration file’s name, located in the `spring-configuration` folder.

- For more information about the sample Spring configuration file, see [“Sample Spring Configuration File,” on page 83](#).
4. If you wish to use the external identifier credentials to validate users, define the mapping relationship between the external user identifier and the user’s Content Server system credentials by adding the appropriate entry to the `SystemUserAttr` table.
For instructions, see [“Mapping External User Identifiers to Content Server Credentials,” on page 78](#).
 5. Restart the CAS web application.

The sample login form looks as follows:



Sample CSSO Classes

The CSSO sample contains three Java classes which extend the default CSSO classes, providing the methods for the sample SSO implementation's login behavior:

- **SampleConfiguration.java** – This class extends the default CSSO `CustomConfiguration.java` class to include a domain property (`sampleDomain`) which will be validated by an external authentication authority when a user provides a value for this field on the login form. The `sampleDomain` property is injected into the CSSO configuration by Spring.

This class also includes the `sampleFormURL` property which defines the sample login form that is called to retrieve a user's credentials. Standard and custom properties for this class are supplied through the sample Spring configuration file.

- **SampleCredentials.java** – This class extends the default CSSO `CustomCredentials.java` class and collects all information required to complete user authentication. The `SampleAuthenticator` class uses the `UsernamePasswordCredentials` object when a user supplies a username and password on the login form. If a user supplies an external identifier on the login form instead of username and password credentials, the `SampleCredentials` object is created to provide that information to the authenticator (in this example, sample SSO class `SampleAuthenticator.java`).

In CAS, the type of credentials object that is created controls which authenticator is used (either standard or custom). If username and password credentials are supplied on the login form, the standard WEM username and password authenticator is used automatically. If an external identifier is supplied on the login form, the custom authenticator is called to authenticate the `SampleCredentials` object.

- **SampleAuthenticator.java** – This class extends the default CSSO `CustomAuthenticator.java` class and contains all the authentication methods that are called by the CSSO framework. When the sample is deployed, all authentication decisions and web-flow actions, during CAS authentication, are directed to this class for action.

The `performLoginAction` method (extended by this class) displays the sample login form. When a user submits his credentials on the form, CAS returns to this method to process the input fields. Depending on the credentials that require verification, the method creates either a `UsernamePasswordCredentials` object or a `SampleCredentials` object, populated with the user's assigned credentials. The credentials object is then inserted into the CAS context (provided by CAS) and a TGT is requested. The TGT request triggers authentication of the credentials object. If

authentication is denied, a ticket exception results in the login form being redisplayed. If the authentication is successful, the next action in the web-flow occurs. For example, acquire a ticket, append the ticket to the original service URL (Content Server's URL), and redirect back to the original service.

There are two authentication methods in this class. One handles authentication using `SampleCredentials` and the other authenticates REST requests, which are usually username/password based. The sample introduces the `sampleDomain` value as a new value to be authenticated. In this case, the `performLoginAction` method encodes the username, password, and `sampleDomain` values provided by the user and passes the encoded values to the `UsernamePasswordCredentials` object. The default WEM authentication handler detects the `sampleDomain` value and passes that credential to the `authenticationRest` method. This method decodes the `sampleDomain` value from the other values and verifies that the correct domain has been specified. If the value is incorrect, authentication fails. If the value is correct, this method encodes the username and password back into the credentials object, and the default WEM authentication handler validates the username and password.

Sample Spring Configuration File

The classes and properties for the sample SSO implementation are defined by the sample Spring configuration file `customSampleSSObeans.xml` (located in `ContentServer/Samples/WEM Samples/CustomizableSSO/src/main/webapp/WEB-INF/spring-configuration`).

The rest of this section contains the following topics:

- [Analyzing the Sample Spring Configuration File](#)
- [Placing the Sample Spring Configuration File](#)

Analyzing the Sample Spring Configuration File

The sample Spring configuration file contains the same bean identifiers as the default Spring configuration file (see, “[Creating a Spring Configuration File](#),” on page 75). However, the property values are modified to implement the sample login behavior. For example, the `customUserConfiguration` bean references the `SampleConfiguration.java` class and the `customUserAuthenticator` bean references the `SampleAuthenticator.java` class.

The `customUserConfiguration` bean also identifies the configuration properties which supply system information to the sample SSO implementation. For example, since the `SampleLoginForm.jsp` file provides the browser form that is used by the sample to obtain a user's credentials, the `SampleConfiguration.java` class is extended to include the `sampleFormURL` property. This property specifies the full URL of the login page for the sample SSO implementation. The domain name and port number match the CAS server installation, and the path points to where this page was placed during set up.

The following is the sample Spring configuration file's code. For more information about the properties referenced by this file, see “[Analyzing the default Spring configuration file](#),” on page 77.

The sample Spring configuration file

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
xmlns:flow="http://www.springframework.org/schema/
webflow-config"
xmlns:p="http://www.springframework.org/schema/p"
xsi:schemaLocation="http://www.springframework.org/
schema/beans
http://www.springframework.org/schema/beans/spring-beans-
2.0.xsd
http://www.springframework.org/schema/webflow-config
http://www.springframework.org/schema/webflow-config/
spring-webflow-config-1.0.xsd">

<!-- Custom SSO Bean definitions. This file defines either
the default CAS/SSO configuration or a special user
implementation. No other CAS configuration files are
modified for a custom implementation -->

<!-- This bean is never modified. It defines the web-flow
controller which always passes control into the custom
authenticator -->
<bean id="customUserLoginAction"
class="com.fatwire.wem.sso.cas.web.CustomLoginViewAction"
    p:centralAuthenticationService-
        ref="centralAuthenticationService"
    p:customAuthentication-ref="customUserAuthenticator"
/>

<!-- This bean is usually not modified. Override it when
there needs to be a custom encoding for information passed
between the web-flow and any external component -->
<bean id="customRestCoder"
class="com.fatwire.wem.sso.cas.custom.basis.
CustomRestTokenCoding"
/>

<!-- Modify this bean with a custom configuration class
when additional parameters are needed for a custom
implementation -->
<bean id="customUserConfiguration"
class="com.fatwire.wem.sso.cas.sample.SampleConfiguration"
    p:casLoginUrl="http://localhost:7080/cas/login"
    p:resolverUrl="http://localhost:8080/cs/custom/
customCsResolver.jsp"
    p:resolverUsername="fwadmin"
    p:resolverPassword="xceladmin"
    p:traceFlag="false"
    p:sampleDomain="mydomain"
    p:sampleFormUrl="http://localhost:7080/cas/
SampleLoginForm.jsp"
/>

<!-- Modify this bean with a customAuthentication class for
a custom implementation. -->
<bean id="customUserAuthenticator"
class="com.fatwire.wem.sso.cas.sample.SampleAuthenticator"
```

```

        p:customConfiguration-ref="customUserConfiguration"
        p:customRestCoder-ref="customRestCoder"
    />
</beans>

```

Placing the Sample Spring Configuration File

To instantiate the sample classes, place the sample Spring configuration file in the `spring-configuration` folder (in `cas/WEB-INF`) and remove the `.xml` extension from the default Spring configuration file. For more information, see [“Placing Your Spring Configuration File,” on page 78](#).

Sample CSSO Components

The sample CSSO implementation’s components are located in the `WEM Samples/CustomizableSSO` folder. The following folders are included with the sample CSSO implementation:

Folder	Description
<code>ContentServer/Samples/WEM Samples/CustomizableSSO</code>	Contains the <code>customizable-ss-1.0.jar</code> file. This jar file provides the classes of the executable code for the sample. If you wish to deploy the sample SSO implementation, place this jar file in the CAS classpath (<code>cas/WEB-INF/lib</code> folder).
<code>ContentServer/Samples/WEM Samples/CustomizableSSO/lib</code>	Contains all the third-party jar files required to compile the Java source files for the sample SSO implementation.
<code>ContentServer/Samples/WEM Samples/CustomizableSSO/src/main/dist</code>	Contains Word documents that explain the individual source components and operations of the sample implementation. Note: We recommend reviewing these documents before viewing the sample’s source code.
<code>ContentServer/Samples/WEM Samples/CustomizableSSO/src/main/java</code>	The root folder for the Java source files.
<code>ContentServer/Samples/WEM Samples/CustomizableSSO/src/main/webapp/fatwire</code>	Contains <code>SampleLoginForm.jsp</code> . The JSP provides the browser form that is used by the sample to obtain a user’s login credentials. Implementing the sample requires creating a <code>fatwire</code> folder in the CAS application context folder and copying the <code>SampleLoginForm.jsp</code> to that folder.

Folder	Description
ContentServer/Samples/WEM Samples/CustomizableSSO/src/ main/webapp/WEB-INF/spring- configuration	<p>Contains the sample Spring configuration file <code>customSampleSSObeans.xml</code>, which defines the Spring bean definitions required by the sample SSO implementation. This file must be placed in the <code>spring-configuration</code> folder (in <code>cas/WEB-INF</code>). The file that exists in the <code>spring-configuration</code> folder (<code>customDefaultWEMSSObeans.xml</code>) must be given an extension other than <code>.xml</code> or removed.</p> <p>Note: Save a copy of the <code>customDefaultWEMSSObeans.xml</code> file so it can be restored when you wish to return to the standard WEM login screen.</p>

Chapter 9

Buffering

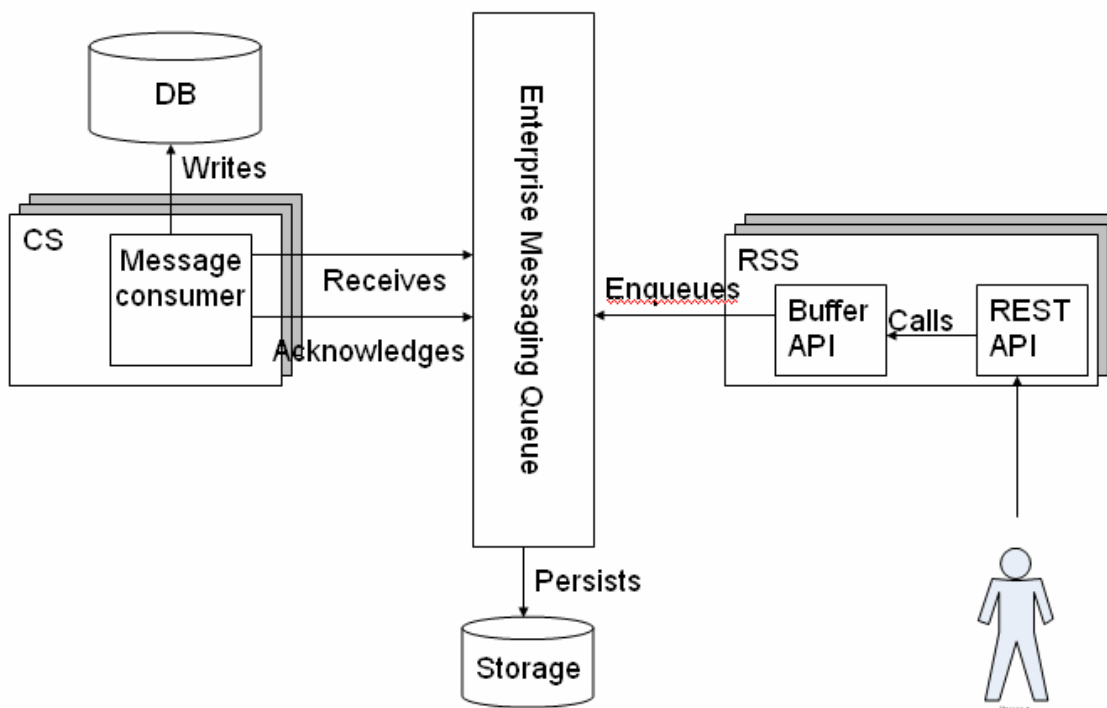
- [Introduction](#)
- [Architecture](#)
- [Using Buffering](#)

Introduction

Asset create, update, and delete operations are much slower than the read operation. Sometimes, it is acceptable to delay these operations to occur at a future time with the guarantee of eventual consistency. That is, if a delayed (buffered) operation was performed, it is guaranteed that the Content Server platform will receive the change at some finite, undetermined period of time. Although buffering operations are extremely fast, they do not speed up the total time that is needed to create, update, and delete assets in the platform.

Architecture

The current implementation of buffering subsystem relies on Java Messaging Service (JMS) technology.



Buffering consists of the following components:

- Buffering Producer, which produces messages and puts them into the Messaging Queue (MQ).
- Buffering Consumer, which picks messages from MQ and persists them in the platform.

The buffering producer can be used on both Content Server and Remote Satellite Server, where the asset REST service `<BaseURI>/sites/<sitename>/types/`

`<assettype>/assets/<id>` is available. When used on Remote Satellite Server, the buffering producer does not communicate with Content Server, which ensures linear scalability of the entire system.

Note

The buffering consumer is available only on Content Server. We recommend enabling the buffering consumer only on the primary cluster member. Enabling on multiple cluster members cannot guarantee that the sequence of CRUD operations will be preserved.

Using Buffering

1. Install the JMS provider if one is not already available. (For supported providers, see the *Supported Platform Document*, available at: <http://support.fatwire.com>)
2. Configure `BufferingConfig.xml` on Content Server and optionally on Remote Satellite Server.

```
id="bufferingManager"
class="com.fatwire.cs.core.buffering.jms.JmsBufferingManager"
```

Property name	Description
<code>jmsConnectionFactory</code>	Required. Instance of <code>javax.jms.ConnectionFactory</code>
<code>jmsDestination</code>	Required. Instance of <code>javax.jms.Destination</code>
<code>messageConsumers</code>	List of <code>com.fatwire.cs.core.buffering.IMessageConsumer</code> implementations.

3. Specify `buffer=true` when invoking the REST asset service `<BaseURI>/sites/<sitename>/types/<assettype>/assets/<id>`.

Note

Buffering does not return the result of PUT and POST operations in the response. Instead, an empty payload is sent. Developers should be aware of this behavior when coding the client application.

The default `BufferingConfig.xml` file, provided with Content Server, contains the sample configuration for Apache ActiveMQ. The `BufferingConfig.xml` file is similar for both Content Server and Remote Satellite Server, except that the list of message consumers for Remote Satellite Server is empty.

Appendix A

Registering Applications Manually

- [Registration Steps](#)
- [Reference: Registration Asset Types](#)

Registration Steps

Registration exposes applications in WEM, as described on [page 18](#). Registering an application manually requires using Content Server's Advanced interface to create an asset for the application, create an asset for each of its views, and associate the view assets with the application asset. The registration asset types `FW_Application` and `FW_View` are enabled on AdminSite.

To manually register an application and view

The section uses code from the “Articles” sample application to illustrate the registration process. “Articles” has a single view of type `iframe`. The same steps apply to JavaScript and HTML views.

1. Create or get an icon to represent your application. (The icon will be displayed in the WEM banner.)

(The “Articles” sample application uses the `articles.png` image file located in: `/sample_app/articles/src/main/webapp/images/`)

2. Create a file that specifies the layout of the application in HTML, i.e., for each view, create a placeholder element to hold the content rendered by the view. Applications and views are related as shown in [Figure 9](#), on [page 44](#).

For example, `layout.jsp` (for the “Articles” sample application) contains the following line:

```
<div id="articles" style="float:left;height:100%;width:100%"  
    class="wemholder"></div>
```

The view's content will be rendered within the placeholder element when the application is displayed (`layout.app` renders the application's layout; `home.app` renders the view).

Note

When creating the layout file, specify a unique `id` for the placeholder element. You will specify the same `id` for the Parent Node attribute when creating the view asset. Use `class="wemholder"` for the placeholder elements.

3. Register the view and application.

- a. Log in to Content Server's Advanced interface as a general administrator, navigate to the AdminSite and click the **Admin** tab, where the `FW_View` and `FW_Application` asset types are enabled.

(We assume you will create the view and application assets in the same session, in which case both assets will be listed on the **History** tab. When creating the application asset, you will select the view asset from the **History** tab and associate it with the application asset. The **History** tab is volatile; it is cleared at the end of the user's session. Assets can be permanently placed on the **Active List** tab. For instructions, see the *Content Server Administrator's Guide*.)

b. Create an instance of the `FW_View` asset type:

Click **New**, select **New FW_View**, and set attributes as shown below this figure. (This figure displays attribute values for the view asset of the “Articles” sample application.)

FW_View:

Cancel Save

*Name: ArticlesView

Description:

Subtype: (no subtype)

Start Date: Format: yyyy-mm-dd hh:mm:ss

End Date: Format: yyyy-mm-dd hh:mm:ss

Category: General

Filename:

Path:

Parent Node: articles

*View Type: iframe

Source Url: http://localhost:9080/articles-0.1/home.app

JavaScript:

Content:

Cancel Save

Name: Enter a short descriptive name for this view asset.

Parent Node: Enter the `id` of the placeholder element (defined in [step 2 on page 92](#)) that will hold the content rendered by the view.

View Type: Select one of the following options to specify how the view’s content should be rendered in the placeholder:

- `Iframe` – renders the view in an `iframe` into the placeholder element
- `IncludeHTML` – renders HTML into the placeholder element
- `IncludeJavaScript` – renders JavaScript into the placeholder element

Source URL: Enter the URL that provides content for the view. For example, Source URL for the “Articles” sample application takes the following value:
`http://localhost:9080/articles-1.0/home.app`

- c. Create an instance of the FW_Application asset type:

Click **New**, select **New FW_Application**, and set attributes as shown below this figure. (This figure displays attribute values for the application asset of the “Articles” sample application.)

FW_Application:

Cancel Save

*Name: Articles

Description:

Subtype: (no subtype)

Start Date: Format: yyyy-mm-dd hh:mm:ss

End Date: Format: yyyy-mm-dd hh:mm:ss

Category: General

Filename:

Path:

Short Description: ?

Tooltip: ?

Icon URL: http://localhost:9080/articles-1.0/images/articles.png ?

Hover Icon URL: ?

Click Icon URL: ?

Active Icon URL: ?

*Layout type: Layout Renderer

Layout URL: http://localhost:9080/articles-1.0/layoutapp ?

Site Access Roles: Browse...

Related: Associated FW_Application: extends (none) Add Selected Items

Associated FW_View: views

Current Contents:

Select Assets from the Tree and click 'Add Selected Items'

Add Selected Items Remove

Cancel Save

Name: Enter a short descriptive name for this application asset.

ToolTip: Enter the text that will be displayed over the application’s icon when users mouse over the icon.

Icon URL: Enter the URL of the icon that represents the application. The icon will be displayed on the login page and at the top of the WEM interface. For example, the Icon URL for the “Articles” sample application takes the following value: `http://localhost:9080/articles-1.0/images/articles.png`

Hover Icon: Enter the URL of the icon that represents the application when users mouse over the icon.

Click Icon: Enter the URL of the icon that represents the application when users click on the icon.

Active Icon: Enter the URL of the icon that represents the application when it is in use.

Layout Type: `LayoutRenderer` (the default and only value). Layout Type is used by the UI container to render the application's views by using the application's layout page (specified below in the Layout URL attribute).

Layout URL: Enter the URL of the page that displays the application's layout. The layout page has only HTML placeholder elements (such as `div`) for placing the view(s).

For example, Layout URL for the "Articles" sample application takes the following value: `http://localhost:9080/articles-1.0/layout.app` (rather than `http://.../layout.jsp`, given the Spring MVC framework.)

Related: Associated FW_View: views: Select the view asset created on [page 93](#) (click the **History** tab, select the view asset, and click **Add Selected Items**).

Reference: Registration Asset Types

- [FW_View Asset Type](#)
- [FW_Application Asset Type](#)

FW_View Asset Type

This asset type is used to register the views of an application. For each view, create an instance of `FW_View`. Attributes of `FW_View` are listed below as they appear in the Content Server Advanced interface and in the *REST API Bean Reference*. Shading indicates a required attribute. This asset type is enabled on the site named ‘AdminSite.’

Table A-1: `FW_View` Asset Type Attributes

Attribute:		Description
CS Interface	REST API	
Name	name	Short descriptive name for this view asset.
Description	description	Description of this view asset.
Parent Node	parentnode	ID of the placeholder element in the application’s layout file. The placeholder element will hold the content rendered by the view. The layout file has only HTML placeholder elements (such as <code>div</code>) for placing the views.
View Type	viewtype	How the view should be rendered. The following view types are available: <ul style="list-style-type: none"> • <code>Iframe</code> – renders the view in an <code>iframe</code> into the placeholder element • <code>IncludeHTML</code> – renders HTML into the placeholder element • <code>IncludeJavaScript</code> – renders JavaScript into the placeholder element
Source URL	sourceurl	URL that provides content for the view.
JavaScript	javascriptcontent	Required if <code>IncludeJavaScript</code> is the view type and Source URL is not specified. The content specified by this attribute is included in a script tag if <code>IncludeJavaScript</code> is specified as the view type. If <code>IncludeJavaScript</code> is specified, either Source URL must be specified, or code must be provided for the JavaScript attribute.
Content	includecontent	Required if <code>IncludeHTML</code> is the view type and Source URL is not specified. The content specified by this attribute is included in the placeholder element tag if <code>IncludeHTML</code> is specified as the view type. If <code>IncludeHTML</code> is specified, either the Source URL must be specified or code must be provided for the Content attribute.

FW_Application Asset Type

This asset type is used to register the application. The asset type is enabled on AdminSite. Attributes of `FW_Application` are listed below as they appear in the Content Server Advanced interface and in the *REST API Bean Reference*. Shading indicates a required attribute.

Table A-2: `FW_Application` Asset Type Attributes

Attribute:		Description
CS Interface	REST API	
Name	name	Short descriptive name for this application asset.
Description	description	Description of this application asset.
Tooltip	tooltip	Text that will be displayed on the application's icon when users mouse over the icon.
Icon URL	iconurl	URL of the icon that represents the application in WEM.
Hover Icon URL	iconurlhover	URL of the icon that represents the application when users mouse over the icon.
Click Icon URL	clickiconurl	URL of the icon that represents the application when users click on the icon.
Active Icon URL	iconurlactive	URL of the icon that represents the application while it is in use.
Layout Type	layouttype	Type of layout. The value is <code>LayoutRenderer</code> . Layout Type is responsible for rendering the application's views by using the application's layout page (specified in the Layout URL attribute, below).
Layout URL	layouturl	URL of the page where the application's layout is displayed. This page has only HTML placeholder elements (such as <code>div</code>) for placing the views.
Related: Associated FW_Application: extends	parentnode	Parent application which the current application extends.
Related: Associated FW_View: views	views	List of view assets used in this application.

