Rock ChMS  
complete Developer

Reference

Version: 1.0.

Nick Airdo

David Turner

Last updated: 7/11/2011

# Table of Contents

Table of Contents 2

System Structure 3

Rock.Framework Project 3

Cms Entities 3

RockWeb application project 5

The Core Components 6

Blocks 6

Adding to the Document Head 6

Sharing Objects Between Block Instances 6

Page\_Init vs. OnInit 7

OnInit vs. OnLoad 7

Pages 7

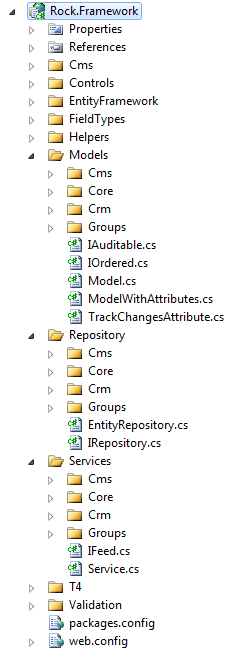
Caching 9

Themes 10

# **System Structure**

There are two projects in the Rock ChMS solution: Rock.Framework and RockWeb.

## Rock.Framework Project

This project has all the EF (entity framework) Models, Repository, Services, etc. It’s anything that does not belong in the web project.

**Models** – Auto generated using the T4 template, each class under the folders (Cms, Core, Crm, etc.) represents an EF (entity framework) entity whose data is persisted to a particular database table using a corresponding repository class described next.

While some classes will inherit from Model, most all custom and core entities will inherit from either the ModelWithAttributes class

**Repository** – These classes handle fetching/persisting the entity data to the database. Using the Repository Pattern allows us to perform some testing using a mock database and not the actual database. These classes are also auto-generated using the T4 template.

**Services** – These classes hold the “business logic” for the Rock application and are also auto-generated using the T4 template. Generally speaking, most everything outside of the Rock Framework will/should access Rock core entities/objects via the services layer.

### Cms Entities

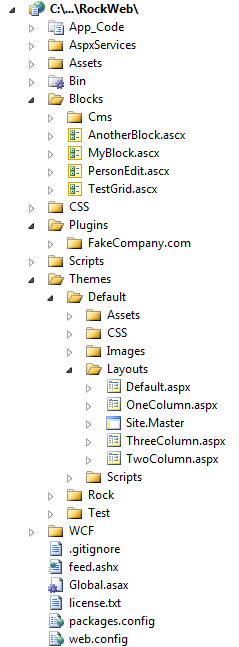
The CMS entities are the parts that make up the Content Management System of Rock. These are primarily Sites, Pages, and Blocks.

**Sites** – These typically correspond to a unique website or domain and are comprised of a collection of pages.

**Page** – A page belongs to a site and also has a layout which defines its structure or zones (ie, header, footer, main, etc.). A page can have a parent page and can also have one or more child pages. A page can also be configured to cache its rendered output for performance considerations (when appropriate) by setting our custom OutputCacheDuration property to anything greater than 0 (seconds).

**Blocks** – These “building blocks” represent reusable pieces of functionality (ASP.NET UserControls). Blocks can be added to a page by adding them a zone on a page or by adding them to a zone in a layout. See Blocks for more details.

## RockWeb application project

This project holds the reusable building core Blocks, Themes, and 3rd party Plugins.

**Themes (Layouts)** – For now, these are represented by physical files that are defined in a Theme (found in the RockWeb application project) and define one or more “Zones”. For example, the Rock Default theme has a default layout that defines two zones: head and main. Additionally, Layouts can also use ASP.NET Master Pages to further control layout.

**Blocks** – These “building blocks” represent reusable pieces of Rock’s core functionality (ASP.NET UserControls). Blocks can be added to a page by adding them a zone on a page or by adding them to a zone in a layout.

**Plugins** – The Plugins folder is where 3rd party developer plugins are stored. Plugins are complete pieces of functionality which are typically comprised of blocks, assets, etc.

# The Core Components

## Blocks

Blocks can be added to a page by adding them a zone on *a page* or by adding them to a zone in *a layout*. Adding a block to a zone in *a layout* will cause all pages which use that layout to automatically have an instance of that block.

Here are several basic things to know about when developing your own custom blocks:

* Blocks can also use the **ThemePath** property as a prefix for any theme-specific things (such as images, css, etc.).
* Blocks can also control how long they are cached by using the **OutputCacheDuration** property.
* The cache methods (**AddCacheItem**(), **GetCacheItem**(), **FlushCacheItem**()) can be used to cache custom data across requests. By default the item’s cache key will be unique to the block instance, but if caching more than one item in your block, you can specify a different key for each item.
* The **AttributeValue**(attributeName) method can be used to get the value of any attribute associated to the instance of the block
* The **UserAuthorized**(actionName) method can be used to test whether the current user (if there is one) is allowed to perform the requested action
* If a block needs data from the page routing/path information (such as the action value or site ID) it can use the **PageParameter**() method to fetch the value.
* The **CurrentPerson** property represents the currently authenticated (logged in) person and the **CurrentPersonId** is that person’s ID.

### Adding to the Document Head

When a block needs to add a reference into the page Head for another asset (javascript, CSS, etc.) it should use one of these methods from the PageInstance class.

**Javascript** - PageInstance.AddScriptLink( this.Page, “../../../scripts/ckeditor/ckeditor.js” );

**CSS** - PageInstance.AddCSSLink( this.Page, “../..//css/cms-core.css” );

**Custom** – PageInstance.AddHtmlLink( this.Page, linkObject );

Example Usage:

System.Web.UI.HtmlControls.HtmlLink rssLink = new System.Web.UI.HtmlControls.HtmlLink();  
rssLink.Attributes.Add( ”type”, ”application/rss+xml”);  
rssLink.Attributes.Add(”rel”, ”alternate” );  
rssLink.Attributes.Add( ”href”, blog.PublicFeedAddress );  
rssLink.Attributes.Add( ”title”, ”RSS” );  
PageInstance.AddHtmlLink( this.Page, rssLink );

### Sharing Objects Between Block Instances

Blocks can communicate with each other through the sharing of objects . The base CmsBlock class has a PageInstance object that is a reference to the current cms page object. This object has two methods for saving and retrieving shared objects specific to current page request. Within your block, you can call

* PageInstance.SaveSharedItem( string key, object item )
* PageInstance.GetSharedItem( string key )

Example Usage:

// try loading the blog object from the page cache  
Rock.Models.Cms.Blog blog = PageInstance.GetSharedItem( “blog” ) as Rock.Models.Cms.Blog;  
   
if ( blog == null )  
{  
 blog = blogService.GetBlog( blogId );  
 PageInstance.SaveSharedItem( “blog”, blog );  
}

It’s worth noting that the order in which loaded blocks modify these shared objects cannot be guaranteed without further preparation and coordination.

### Page\_Init vs. OnInit

There’s not really any big difference besides preference. Overriding the base method (OnInit) may be slightly faster than invoking an event delegate (Page\_Init), and it also doesn’t require using the AutoEventWireup feature, but essentially it comes down to preference. My preference is to override the event. (i.e. use OnInit or OnLoad instead of Page\_Init or Page\_Load). [This Article](http://weblogs.asp.net/infinitiesloop/archive/2008/03/24/onload-vs-page-load-vs-load-event.aspx) discusses this in detail.

### OnInit vs. OnLoad

There’s a significant difference between putting code into the OnInit (Page\_Init) method compared to the OnLoad (Page\_Load) method, specifically in how it affects ViewState. Any change you make to a control in the Init portion of the page life cycle does not need to be added to ViewState, however, if changed in the Load portion it does. Consider a dropdown box of all the states. If you load the dropdown in the OnLoad method, all of the 50 items of the dropdown box will be added to the ViewState collection, but if you load it in the OnInit method, they will not. For performance sake, we want to keep ViewState as small as possible. So whenever possible set the properties of controls in the OnInit method. Please read [this article](http://weblogs.asp.net/infinitiesloop/archive/2006/08/03/Truly-Understanding-Viewstate.aspx).

## Pages

The following are some basic properties and methods of the Page class that you will find useful.

* The **AttributeValue**(attributeName) method can be used to get the value of any attribute associated to the instance of the page.
* The **CurrentPerson** property represents the currently authenticated (logged in) person and the **CurrentPersonId** is that person’s ID.
* **DisplayInNavWhen** – Determines when a page should be listed in navigation. Valid Values are:  
  0 = When Security Allows (default)  
  1 = Always (always shows up. If you don’ have security when you click it, it will ask you to log in. This keeps you from having to make redirect pages)  
  2 = Never (no matter what it won’t show up)
* **MenuDisplayDescription** – Tells the drop down menu to add the description to the page’s listing.
* **MenuDisplayIcon** – Tells the drop down menu to add the icon to the page’s listing.
* **MenuDisplayChildPages** – Tells the drop down menu to add a list of child pages to the page’s listing.

# Caching

# Themes