ROCK CHMS COMPLETE DEVELOPER REFERENCE

DRAFT

Version: 1.0.26

Nick Airdo

David Turner

Last updated: 11/21/2011

TABLE OF CONTENTS

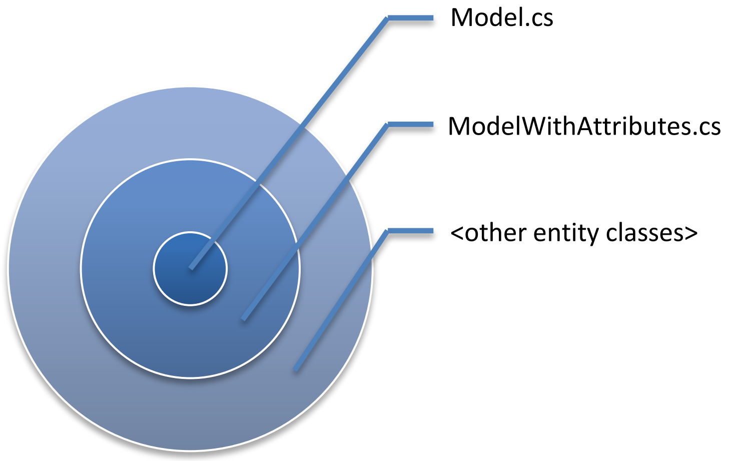
SYSTEM STRUCTURE OVERVIEW

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.

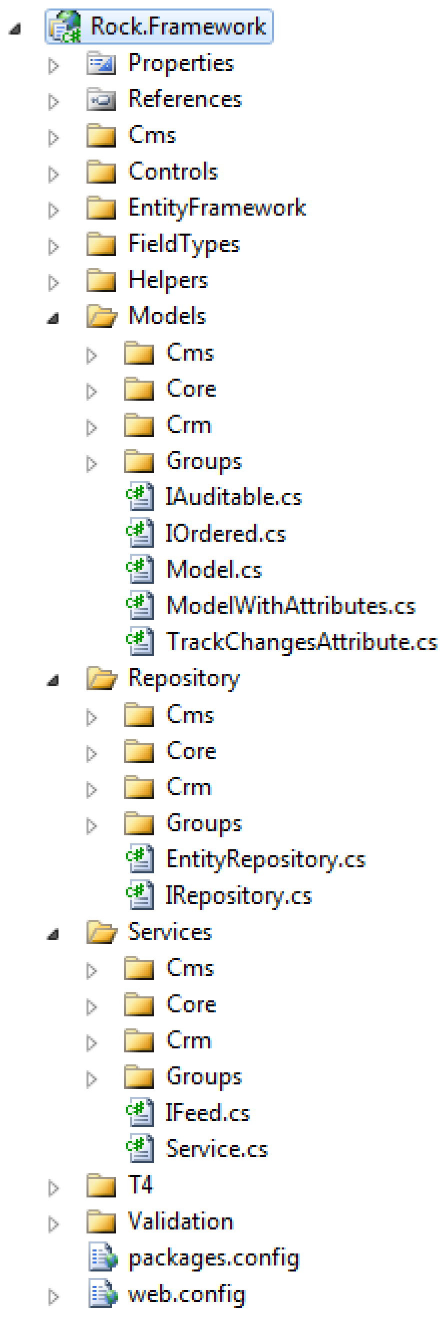
Models – Auto generated using the T4 template, each class under the folders (Cms, Core, Crm, etc.) represents an EF (entity framework) entity whose data is persisted to a particular database table using a corresponding repository class described next.



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.



```
Rock.Framework
  Properties
  References
  Cms
  Controls
  EntityFramework
  FieldTypes
  Helpers
  Models
    Cms
    Core
    Crm
    Groups
    IAuditable.cs
    IOrdered.cs
    Model.cs
    ModelWithAttributes.cs
    TrackChangesAttribute.cs
  Repository
    Cms
    Core
    Crm
    Groups
    EntityRepository.cs
    IRepository.cs
  Services
    Cms
    Core
    Crm
    Groups
    IFeed.cs
    Service.cs
  T4
  Validation
  packages.config
  web.config
```

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 (header, footer, main, etc.). A page can have a parent page and can also have one or more child pages.

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.

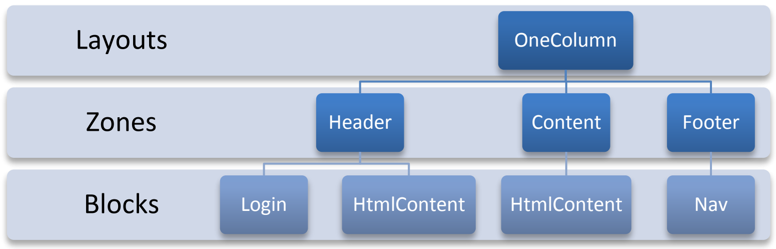
ROCKWEB APPLICATION PROJECT

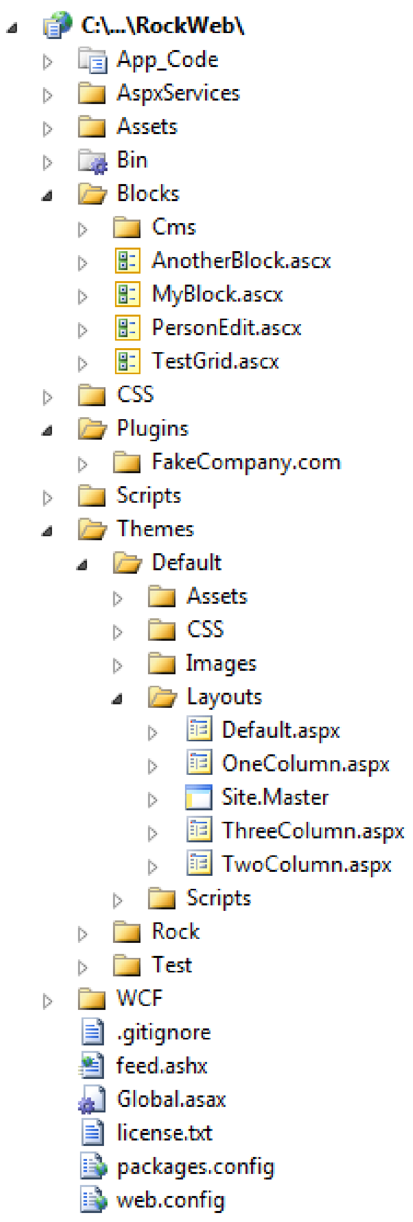
This project holds the reusable building core Blocks, Themes, and 3rd party Plugins.

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.





```
C:\...\RockWeb\
  App_Code
  AspxServices
  Assets
  Bin
  Blocks
    Cms
    AnotherBlock.ascx
    MyBlock.ascx
    PersonEdit.ascx
    TestGrid.ascx
  CSS
  Plugins
    FakeCompany.com
  Scripts
  Themes
    Default
      Assets
      CSS
      Images
      Layouts
        Default.aspx
        OneColumn.aspx
        Site.Master
        ThreeColumn.aspx
        TwoColumn.aspx
      Scripts
    Rock
    Test
  WCF
  .gitignore
  feed.ashx
  Global.asax
  license.txt
  packages.config
  web.config
```

THE CORE ROCK COMPONENTS

BLOCKS

In Rock ChMS, 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**:

- Instances of Blocks can have admin/user controlled, configurable properties. These can be used to change the behavior or functionality of a Block. See Block Instance Properties (BIP) for details.
- The **AttributeValue**(attributeName) method can be used to get the value of any block instance property .
- Blocks can also use the **ThemePath** property as a prefix for any theme-specific resources (images, css, etc.). To reference resources that are not part of a Theme use **ResolveUrl**(path).
- 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 **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.

PAGES

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

- Setting the **OutputCacheDuration** property to anything greater than 0 (seconds) will cache its rendered output for performance considerations (use when appropriate).
- 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.

THEMES / LAYOUTS

THEMES

DEVELOPING CORE CLASSES

CODE GENERATION VIA T4 TEMPLATE

The bulk of the framework classes are auto-generated from SQL tables using the T4 template Rock.Framework/T4/Model.tt, including these namespaces: Api, EntityFramework, Models, Repository, Services.

In the event that manually written code needs to be added for a class, simply create a partial class next to the auto-generated class with the naming convention <classname>.partial.cs.

Enums

To have model properties created as an enumeration, make sure the column has a datatype of int and add text to the description in the format enum[NameOfEnum]. This text can be anywhere in the description. Because EF does not natively support enums the models will be generated with two properties to support each enum field. Below is an example of how an enum field is generated.

```
[DataMember(Name = "DisplayInNavWhen")]
internal int DisplayInNavWhenInternal { get; set; }

[NotMapped]public DisplayInNavWhen DisplayInNavWhen
{
    get { return (DisplayInNavWhen)this.DisplayInNavWhenInternal; }
    set { this.DisplayInNavWhenInternal = (int)value; }
}
```

HELPER METHODS

There are several classes/methods in the Rock.Framework project under the Helpers folder

Rock.Helpers.Reflection

FindTypes() – Static method that will return a sorted dictionary object of all types that inherit from a specified base type. Will search through the Rock.Framework.dll and any other dll in the same folder (web/bin) that have a pattern of Rock.*.dll

ClassName() – Static method that will return the [Description] attribute value of a given class type, or the class name if the attribute does not exist.

Misc

When creating new classes for the Rock.Framework or some other 3rd party framework layer that uses EF, keep in mind the following:

- Use “[Table("<*TABLENAME*>")]” to specify the name of your class’s persistence table.
- Add the “[NotMapped]” attribute on any properties that are not mapped to a column in the database.

TBD

DEVELOPING CUSTOM BLOCKS

BLOCK INSTANCE PROPERTIES (BIP)

When a Block class is decorated with a “BlockInstanceProperty” attribute, instances of the Block can store a user provided value for the property. For example, this ‘Root Page’ block instance property would be found on a Block whose purpose is to generate navigation, and it will store the value of a page.

```
[BlockInstanceProperty( "Root Page", "The root page to use for the navigation" )]
```

In this case a simple textbox is used to collect the value from the user; however other field types can be specified to control this aspect in addition to specifying a default value for the BIP as shown here:

```
BlockInstancePropertyAttribute( string name, string key, string description, string default
Value, string fieldTypeAssembly, string fieldTypeClass)
```

RELATIVE PATHS

Both the cmsBlock and the cmsPage objects have a public “ThemePath” property that can be used in either a block or template file to get the resolved path to the current theme folder. Here’s an example of how to use this property:

Markup – `src=‘<%= ThemePath %>/Images/avatar.gif’`

Code Behind – `myImg.ImageUrl = ThemePath + “/Images/avatar.gif”;`

If trying to reference a resource that is not in the theme folder, you can use the ResolveUrl() property of the System.Web.UI.Control object. For example:

```
<link type=“text/css” rel=“stylesheet” href=“<%# ResolveUrl(”~/CSS/reset-core.css") %>" />
```

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. The path should be relative to the layout template.

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 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.

POPUP WINDOWS

In Rock ChMS we’ve abstracted the jQuery plugin used for displaying popup windows to standardize its look (animation settings, size, etc) by creating our own “popup” jQuery plugin. It’s located in

RockWeb\Scripts\rock\popup.js. It currently implements the colorbox plugin but if we later decide to switch from colorbox to something new, it will be an easy swap (provided all Rock Blocks are using our popup plugin).

To implement a popup, you'll first need to create an anchor tag where the href attribute is either the id of a div element on the same page, or an external page's url. When using the id of a div, it's important to include the '#' character. The plugin evaluates the first character of the href property and will set things up differently (inline div vs. external page) based on the presence of this character.

You can call the plugin for your anchors like so:

```
$(document).ready(function () {
        $('a.zone-blocks').popup();
    });
```

This is all that is needed to display a popup with the default values. Any of the default values can be overriden though. Here's an example overriding the width and onClosed:

```
$('a.zone-blocks').popup({height: '80%', onClosed:function(){ location.reload(true); }});
```

Note: When using an *inline* div, your div should be wrapped within another div that has the display:none css style.

CACHING

TBD

NAMESPACES AND CONVENTIONS

TBD

UI STANDARDS AND GUIDELINES

In order to ensure consistent UI and extensible CSS it is important that all forms be coded in the following manner:

```
<fieldset>
      <legend>Account Information</legend>
      <ol>
         <li>
            <asp:Label ID="UserNameLabel" runat="server"
AssociatedControlID="UserName">Username:
               <asp:TextBox ID="UserName" runat="server"></asp:TextBox></asp:Label>
         </li>
         <li>
            <asp:Label ID="PasswordLabel" runat="server"
AssociatedControlID="Password">Password:
                        <asp:TextBox ID="Password" runat="server" CssClass="passwordEntry"
TextMode="Password"></asp:TextBox></asp:Label>
          </li>

     </ol>
</fieldset>
```