Rock ChMS  
complete Developer

Reference

DRAFT Version: 0.0.47

Nick Airdo

David Turner

Last updated: 2/7/2012

http://www.sparkdevelopmentnetwork.com/assets/images/spark-logo.png

# **Table of Contents**

Table of Contents 2

System Structure Overview 3

Rock (Framework) Project 3

Cms Entities 5

RockWeb WebSite project 6

Rock.DataTransferObjects 7

The Other Projects 7

Quartz 7

RockJobSchedulerService 7

RockJobSchedulerServiceInstaller 7

The Core Rock Components 8

Blocks 8

Pages 8

Themes / Layouts 9

Themes 10

Developing Core Classes 11

Code Generation via T4 Template 11

Enums 11

Helper Methods 11

Rock.Helpers.Reflection 11

Misc 11

Developing Custom Blocks 13

Block Instance Properties (BIP) 13

Relative Paths 13

Adding to the Document Head 13

Sharing Objects Between Block Instances 14

Page\_Init vs. OnInit 14

OnInit vs. OnLoad 14

Popup Windows 14

Caching 15

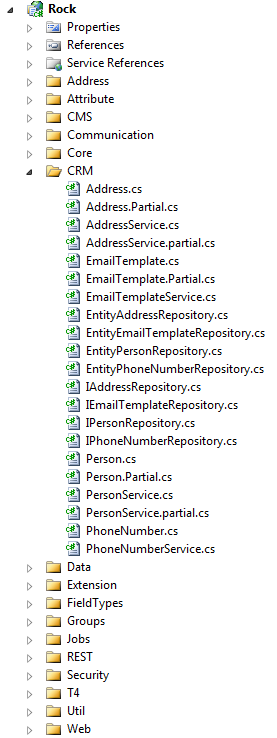
Namespaces and Conventions 16

UI Standards and Guidelines 17

# **System Structure Overview**

The two main projects in the Rock solution are Rock and RockWeb. The REST API can be seen while running the RockWeb project at by accessing ~/REST/help.

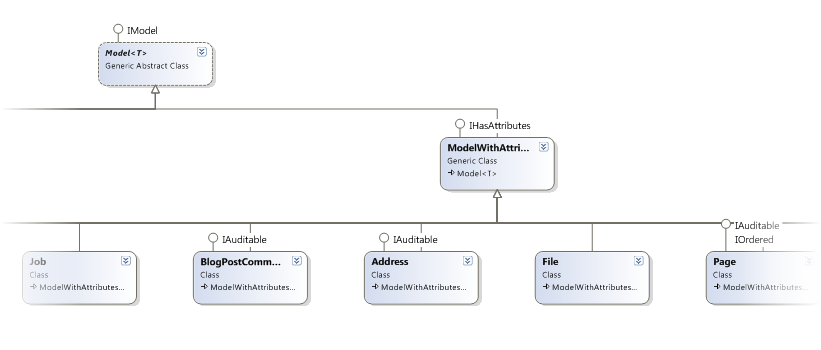
## Rock (Framework) Project

This project contains the entity framework (EF) Models, Repository, Services, etc.

**Models** – Auto generated using the T4 template, each class is put under a logical folder whose name comes from the database table prefix (ie, the Address entity from the table crmAddress goes into the Crm folder) and represents an EF entity whose data is persisted using a corresponding repository class described next.

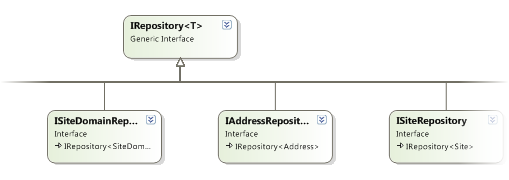
While some classes will inherit from Model (such as Attribute, AtrributeQualifier, FieldType, etc), most all custom and core entities will inherit from the ModelWithAttributes class.

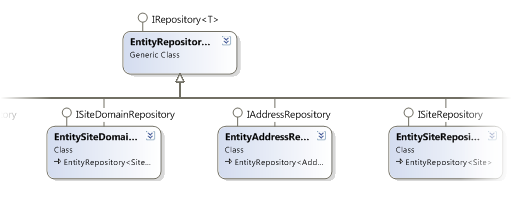
Figure - Rock framework project highlighting the "Address" entity.



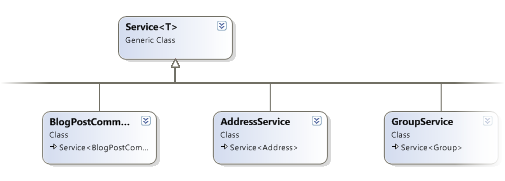
Code that is not auto-generated is put in a partial class next to the main entity (ie, Address.Partial.cs).

**Repository** – These classes (such as EntityAddressRepository and its corresponding interface, IAddressRepository) handle fetching/persisting the entity data to the database. Each entity and corresponding interface class inherits from the EntityRepository and IRepository base classes, respectively. 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 (such as AddressService.cs) hold the “business logic” for the Rock application and are also auto-generated using the T4 template. They inherit from the Service base class (found under \Data\). 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. Other notable entities are Auth and User.

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

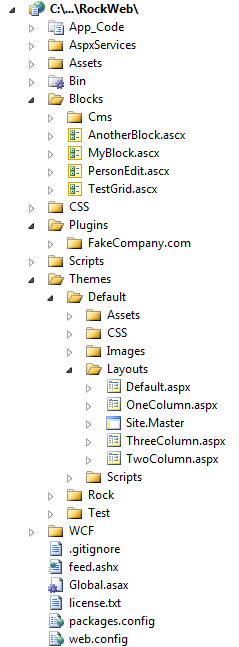
**Page** – A page belongs to a site and also has a layout which defines its structure or zones (header, footer, main, etc.). A page can have a parent page and can also have one or more child pages.

**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.

**Auth** – Are used to manage authorization (who can do what) of various Rock entities.

**User** – This represents the authenticated user viewing the website (or Rock application). It typically goes hand-in-hand with the Auth class.

## RockWeb WebSite project

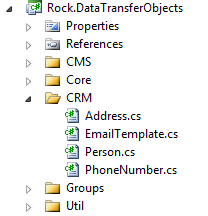
This project holds the reusable building core Blocks, Themes, and 3rd party Plugins.

**Themes (Layouts)** – For now, these are represented by physical files that are defined in a Theme 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.

## Rock.DataTransferObjects

The classes in this project represent lightweight (disconnected from context, etc.) representations of the entities and are used in situations such as serializing the object for use by the REST API.

They are also automatically created by the T4 template and are put into folders corresponding to their table prefix.

## The Other Projects

The other projects in the solution are for managing other aspects of Rock, such as job/task scheduling.

### Quartz

This is a customized version of the open source [Quartz.NET job scheduling system](http://quartznet.sourceforge.net/) (v*2.0 beta 1 for .NET 4.0* at the time of this writing) which was modified so that it can run in Medium Trust environments, which is required by many hosting companies. Quartz is the integrated job/task scheduling system in Rock.

### RockJobSchedulerService

Normally the job/tasks scheduling will run from within IIS (RunJobsInIISContext appsetting key in the web.config is “true”); however in environments where users/administrators have ownership of the server, it may be more desirable to run the job/tasks scheduling system as a Windows Service. This project represents the source to build that service. The RunJobsInIISContext key value should be set to “false” when running as a Windows Service.

### RockJobSchedulerServiceInstaller

This is the installer application for installing the RockJobSchedulerService.

### Rock.REST

This is a web project for only the REST API. The default page is the help interface (as mentioned earlier as seen via accessing ~/REST/help while running the RockWeb project.)

**Using the REST API**

Example accessing the REST API via client JavaScript (as seen in RockWeb\Scripts\Rock\page-admin.js):

// Get the current block instance object

$.ajax({ type: 'GET', contentType: 'application/json', dataType: 'json',  
 url: rock.baseUrl + 'REST/Cms/BlockInstance/' + blockInstanceId,  
 success: function (getData, status, xhr) {

// Update the new zone  
 getData.Zone = zoneName;

// Do something  
 // …

// Save the updated block instance  
 $.ajax({ type: ‘PUT’, contentType: ‘application/json’, dataType: ‘json’,  
 data: JSON.stringify(getData),  
 url: rock.baseUrl + ‘REST/Cms/BlockInstance/Move/’ + blockInstanceId,  
 success: function (data, status, xhr) {  
 // …  
 },  
 error: function (xhr, status, error) {  
 alert(status + ’ [’ + error + ’]: ’ + xhr.responseText);  
 }  
 });  
 },  
 error: function (xhr, status, error) {  
 alert(status + ’ [’ + error + ’]: ’ + xhr.responseText);  
 }

});

Example accessing the REST API via C# code (as seen in Rock.Custom.CCV.ClientTestApp\Rock.Custom.CCV.ClientTestApp\Form1.cs):

Rock.CRM.DTO.Address address = new Rock.CRM.DTO.Address { Street1 = tbStreet1.Text,

Street2 = tbStreet2.Text, City = tbCity.Text, State = tbState.Text, Zip = tbZip.Text

};

WebClient proxy = new System.Net.WebClient();

proxy.Headers["Content-type"] = "application/json";

MemoryStream ms = new MemoryStream();

DataContractJsonSerializer serializer = new DataContractJsonSerializer( typeof( Rock.CRM.DTO.Address ) );

serializer.WriteObject(ms, address);

try

{

byte[] data = proxy.UploadData( string.Format( "http://localhost:6229/RockWeb/REST/CRM/Address/{0}/{1}", service, APIKEY ),

"PUT", ms.ToArray() );

Stream stream = new MemoryStream( data );

return serializer.ReadObject( stream ) as Rock.CRM.DTO.Address;

}

catch ( WebException ex )

{

using ( Stream data = ex.Response.GetResponseStream() )

{

string text = new StreamReader( data ).ReadToEnd();

MessageBox.Show( string.Format( "Response: {0}\n{1}",

( ( System.Net.HttpWebResponse )ex.Response ).StatusDescription, text ), service + " Error" );

}

return null;

}

**Extending the REST API**

Rock uses MEF to find all of the available REST WCF services. To extend the REST API by adding your own service, create a class that implements the Rock.REST.IService interface. The class must include the MEF [Export] and [ExportMetadata] attributes. Best practice would be to include both an interface and implementation class for your WCF REST class as seen here:

[ServiceContract]   
public partial interface IPageService  
{  
 [OperationContract]  
 Rock.DataTransferObjects.Cms.Page Get( string id );  
 [OperationContract]  
 Rock.DataTransferObjects.Cms.Page ApiGet( string id, string apiKey );

}

[Export(typeof(IService))]  
[ExportMetadata(“RouteName”, “Cms/Page”)]  
[AspNetCompatibilityRequirements( RequirementsMode = AspNetCompatibilityRequirementsMode.Allowed )]  
public partial class PageService : IPageService, IService

{

[WebGet( UriTemplate = “{id}” )]

public Rock.DataTransferObjects.Cms.Page Get( string id )

{

var currentUser = System.Web.Security.Membership.GetUser();

if ( currentUser == null )

throw new WebFaultException(“Must be logged in”, System.Net.HttpStatusCode.Forbidden);

// …

}

[WebGet( UriTemplate = “{id}/{apiKey}” )]

public Rock.DataTransferObjects.Cms.Page ApiGet( string id, string apiKey )

{

using (Rock.Helpers.UnitOfWorkScope uow = new Rock.Helpers.UnitOfWorkScope())

{

Rock.Services.Cms.UserService userService = new Rock.Services.Cms.UserService();

Rock.Models.Cms.User user = userService.Queryable().Where( u => u.ApiKey == apiKey ).FirstOrDefault();

if (user != null)

{

// …

}

else

throw new WebFaultException( “Invalid API Key”, System.Net.HttpStatusCode.Forbidden );

}

}

}

Notice also that it’s best practice to include two methods for each action (one that expects an API Key and one that doesn’t). For the method that doesn’t get the API Key, you can ensure that the current user is logged in if necessary. The method with the API Key can be used by third-party applications that need access to your API, while the other can be used by the ROCK blocks since they have will be running on the same ASP.NET site and can be “logged in.”

# 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/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 useful classes/methods in the Rock project you may want to become familiar with.

### Rock.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.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 defaultValue, 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](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).

## 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 overridden 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

When you write custom stuff, please adhere to the following rules below to avoid collisions of various types.

## Custom Tables and Entities

Custom tables should be prefixed with an underscore followed by your organization’s acronym such as:

\_moz or \_ccv

## Custom API

When developing custom API extensions, developers must use a folder convention api/*<com.domain>*/ to avoid collisions with other custom developer APIs.

# 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>