COMPLETE DEVELOPER

REFERENCE

DRAFT Version: 0.0.67

Nick Airdo

David Turner

Jon Edmiston

Last updated: 6/18/2012

TABLE OF CONTENTS

SYSTEM STRUCTURE OVERVIEW

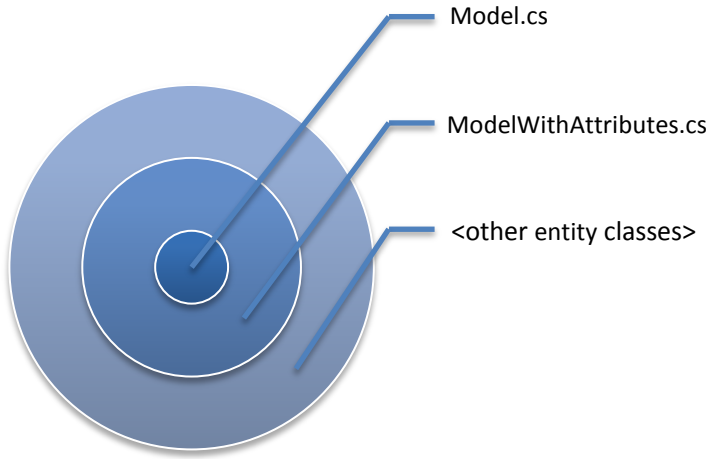
The two main projects in the Rock solution are Rock and RockWeb. The REST API can be seen while running the RockWeb project at by accessing ~/REST/help.

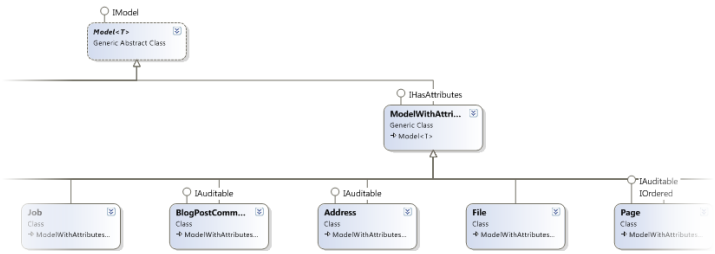
ROCK (FRAMEWORK) PROJECT

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

Models –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 custom and core entities will inherit from the ModelWithAttributes class.





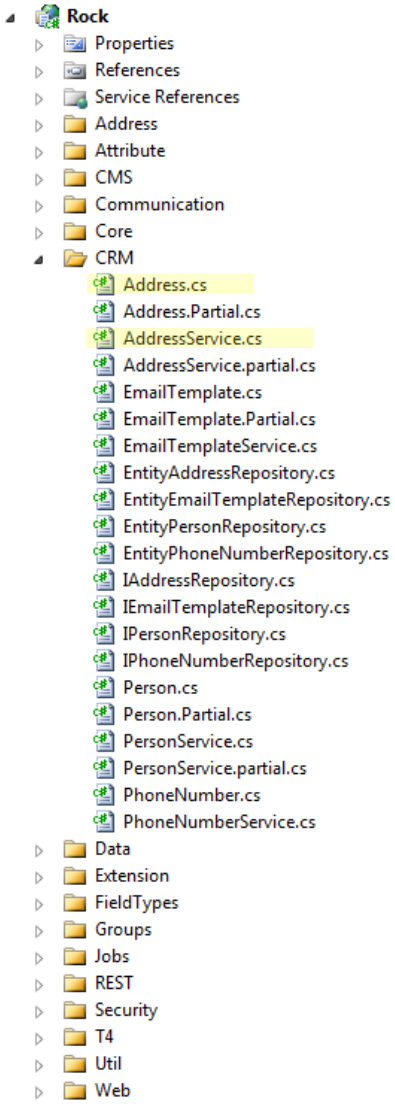
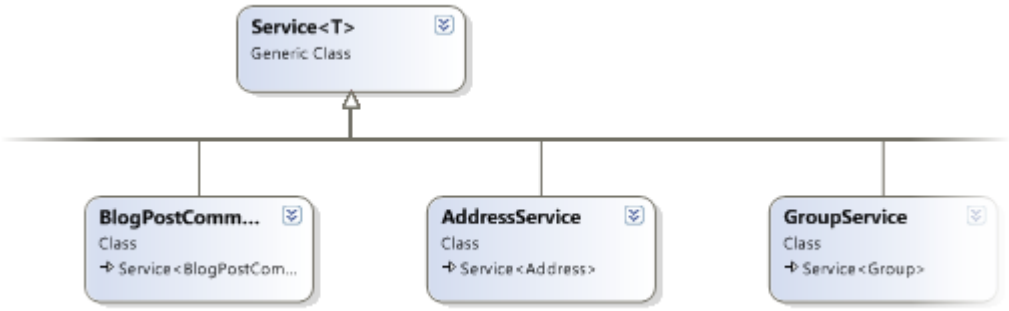


Figure 1 - Rock framework project highlighting the "Address" entity.

Services – These classes (such as AddressService.cs) hold the "business logic" for the Rock application. 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.

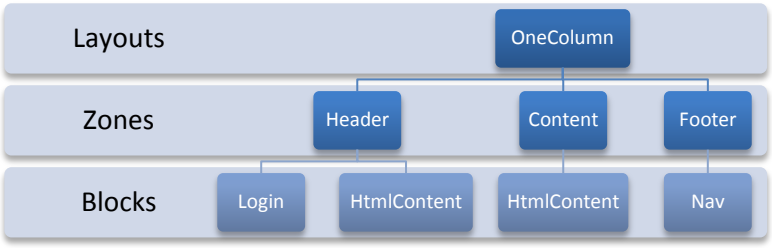
ROCKWEB WEBSITE PROJECT

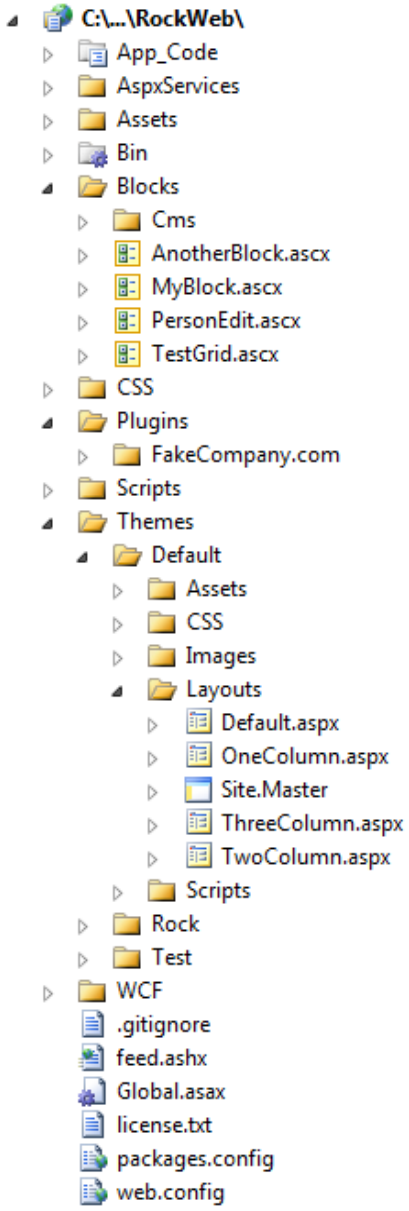
This project holds the reusable building core Blocks, Themes, and 3rd party Plugins.

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.

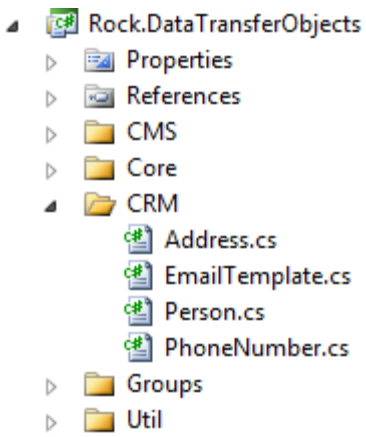




ROCK.DATATRANSFEROBJECTS

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 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 (*v2.0 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

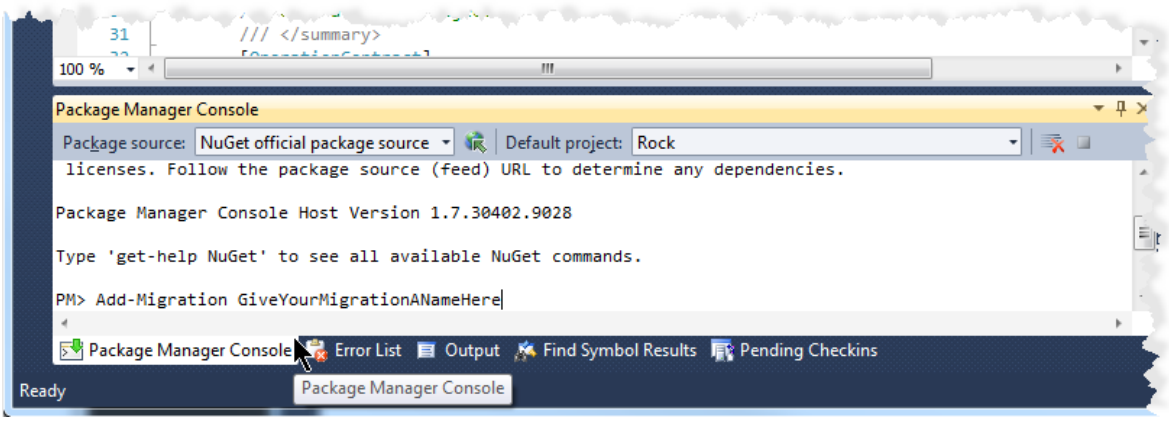
DEVELOPING CORE CLASSES

CODE FIRST

Rock is now a code-first project. In other words, EF is now responsible for managing the database layer. Changes to the database are largely managed via "Migrations". This is a good article to read/understand about code-first migrations.

Adding/Removing/Changing Model Properties

After making a change to an existing model, you'll need to generate the migration by opening the Package Manager Console (View -> Other Windows -> Package Manager Console) and typing the Add-Migration command and giving it a name:



A migration will be created under the Rock/Migrations folder. An Up and a Down method will be created which implement the changes you've made to the model for the database level. The next time you run solution, the database changes will be implemented.

We want to see only one migration per feature (and not end up with a bunch of little migrations) so please feel free to re-run the Add-Migration ***using the same name*** in order to have them all included into the same migration. In fact, we plan to merge all migrations into one single migration for each release of Rock.

In the event that you need to back out your migration you simply target a specific migration and the framework will set the database back to the correct state (migration).

```
Update-Database –TargetMigration:"<TheNameOfTheMigration>"
```

You can watch a video by David T on this topic.

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.

ENTITY CHANGE LOGGING

If you would like to log or track changes made to your custom entities, you can use the “TrackChanges” decorator attribute on a model’s properties as seen in this example:

```
[Table( "crmPerson" )]
public partial class Person : ModelWithAttributes<Person>, IAuditable
{
        …

        /// <summary>
        /// Gets or sets the Nick Name.
        /// </summary>
        /// <value>
        /// Nick Name.
        /// </value>
        [MaxLength( 50 )]
        [TrackChanges]
        [DataMember]
        public string NickName { get; set; }
        …
```

When the Save() method is called, the framework will automatically log any changes that were made to that property to the coreEntityChange table:

Id	Change...	ChangeType	EntityType	EntityId	Property	OriginalValue	CurrentValue	CreatedDateTime	CreatedByPersonId	Guid
1	FFB85A...	Added	Person	0	FirstName		Jonathan	NULL	NULL	0000000...
2	C1C2F7...	Added	Person	0	NickName		Jon	NULL	NULL	FC86D3A...
3	C1C2F7...	Added	Person	0	FirstName		Jonathan	NULL	NULL	5DFE1B9...
4	C1C2F7...	Added	Person	0	LastName		Edmiston	NULL	NULL	22EED5E...
5	E70AA7...	Added	Person	4	FirstName		Jon	NULL	NULL	E1488F8...
6	9FE7FC...	Modified	Page	0	Name	Main Page	Main Page Test	2011-12-10 07:26:12.873	1	B7A68B5...
7	B76C9A...	Modified	Page	0	Name	Create Account	New Account	2012-01-17 06:09:09.917	1	B936EBF...
8	0A8F41...	Modified	Page	0	Name	Organization Values	Global Values	2012-01-25 06:05:09.357	1	B9BD246...
9	F8EFAE...	Modified	Page	0	Name	Organization Settings	Global Attributes	2012-01-25 06:05:50.060	1	CA32B25...

DEVELOPING CUSTOM BLOCKS

BLOCK INSTANCE PROPERTIES (BIP)

When a Block class is decorated with a "BlockInstanceProperty" attribute, instances of the Block can store an admin provided value for the property. For example, a 'Root Page' block instance property might be found on a Block (whose purpose is to generate navigation) to store the value of a page id.

```
[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 administrator; 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 defaultValu
e, string fieldTypeAssembly, string fieldTypeClass)
```

There is different kind of configurable property, called **Global Attributes**, which are *not block instance specific* but instead are used to store configurable values for *any and all* blocks and code (Jobs, Transactions, etc.). See the Global Attributes section for more information about these settings.

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 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

EXCEPTION HANDLING

Rock has a built in exception handling mechanism. Most exceptions should be caught and appropriately handled in the Blocks, however any unhandled exceptions will be logged by the core framework and an error page will be displayed. A few things worth noting:

- Exceptions are logged in the coreExceptionLog table.
- The RockCleanup job will clean this log/table while keeping N number of days' worth of exceptions.
- There is an organization global attribute 'Log404AsException' that will log any 404 File Not Found errors into the same log (no error will be displayed to the user). By default it is disabled since it adds overhead to the processing of the page. It's there for webmasters to occasionally enable in order to see and fix 404 errors.

ERROR PAGES

In Rock each site can be configured to use a custom error page in the event of an exception. If no value is provided, error.aspx will be shown which is skinned to match the Rock Theme. This standard error page will display the details of the exception if the logged in user is a part of the **Rock Administrators** security group.

New, custom error pages should be very simple – even static HTML. If one decides to make it more robust (i.e. by adding logic to display the error) it should be careful not to generate an exception itself because that would cause an infinite loop. A query parameter has been added to the error page to help catch these loops. If the parameter is not '1' then processing should not be done as it is causing an error.

NOTIFICATIONS

There is a global attribute (EmailExceptionsList) that controls who will receive exception notifications. Its value is a comma delimited list of email addresses.

PERFORMANCE RELATED CONSIDERATIONS

Speed is a primary feature of Rock ChMS. Before writing any code think about performance, and when you write code, code for performance.

TRANSACTIONS

Every effort should be made to return a page back to the user as quickly as possible. Any processing that can be done out-of-process should consider using transactions.

Rock has a built-in transaction queue to handle out-of-process execution of code. A block can create a transaction, add it to the queue and move on. An example usage is the implementation of page analytics. To capture data for pages that have been viewed, a transaction is added to the queue instead of writing to the database directly while the user waits. In many cases you can see nearly 100x increase in responsiveness[1].

Using Transactions

A transaction type class must be created for *type* of transaction. These must inherit from ITransaction which has one method called 'Execute'. For example, to implement the page analytics feature described above, a PageViewTransaction.cs class was created with an Execute method consisting of:

```
    /// <summary>
    /// Execute method to write transaction to the database.
    /// </summary>
    public void Execute()
    {
        string directory = AppDomain.CurrentDomain.BaseDirectory;
        directory = Path.Combine( directory, "Logs" );
        // check that directory exists
        if ( !Directory.Exists( directory ) )
            Directory.CreateDirectory( directory );

        // create full path to the fie
        string filePath = Path.Combine( directory, "pageviews.csv" );

        // write to the file
        StreamWriter w = new StreamWriter( filePath, true );
        w.Write( "{0},{1},{2},{3},{4},{5}\r\n", DateViewed.ToString(),  PageId.ToString(),
SiteId.ToString(), PersonId.ToString(), IPAddress, UserAgent);
        w.Close();
    }
```

[1] Through performance testing using this feature we found that inserting 100 records from a block into the database took 1123ms however adding 100 corresponding transactions to the queue only took 15ms.

To use this transaction type on a block you would simply instantiate an object, set its properties, and add it to the transaction queue using the RockQueue.TransactionQueue's Enqueue method. Using our working example, this is how the Rock page loader uses the PageViewTransaction to record page views:

```
        PageViewTransaction transaction = new PageViewTransaction();
        transaction.DateViewed = DateTime.Now;
        transaction.PageId = PageInstance.Id;
        transaction.SiteId = PageInstance.Site.Id;
        if ( CurrentPersonId != null )
            transaction.PersonId = (int)CurrentPersonId;
        transaction.IPAddress = Request.UserHostAddress;
        transaction.UserAgent = Request.UserAgent;

        RockQueue.TransactionQueue.Enqueue( transaction );
```

The Rock queue manager will wake up (currently every 60 seconds) and drain the queue by calling the each transaction's Execute method through the interface.

Sample code can be found in Rock.Transactions. In general though this is very simple, it is also very powerful.

CONSIDERATION

Transactions are meant for short running tasks and are **not recommended** for very long running tasks. They are not cost-free processing. They still operate in the IIS context and *still use* processing and memory. Longer running tasks should be developed with other alternatives, such as Arena Jobs.

GLOBAL ATTRIBUTES

Rock has a place where your custom blocks and other code (Jobs, Transactions, etc.) can access globally configured setting values. To retrieve a value, use the Rock.Web.Cache.GlobalAttributes.Value method while passing in an appropriate key as seen here:

```
// determine if 404's should be tracked as exceptions
bool track404 = Convert.ToBoolean( Rock.Web.Cache.GlobalAttributes.Value( "Log404AsException" ) );
```

MERGE FIELDS

The *values* of global attributes can also have merge fields in them that contain *other* global attributes. For example, say you are working on an email templates and several of the templates should have the same header and footer (common for styling). These headers and footers however may have settings in them like the background color or logo url. This is all possible with 'global attribute nesting'. The 'header' attribute can include the 'background-color' attribute and be used in the email template. This nesting can be n levels deep, however, simple is better. The administrator must be very careful to not create circular references (A includes B which includes A) as these will cause the system to enter a loop.

NAMESPACES AND CONVENTIONS

When you write custom stuff please adhere to the rules below to avoid collisions with other developer's stuff. Below you'll see reference to your organization's *<OID>*. This means some unique string such as your organization's name, acronym or domain name.

Examples: **Moz** – for Mozilla, **JordanRift** or **JRift** – for Jordan Rift, **CCV** – for Christ's Church of the Valley,

CUSTOM TABLES

Custom tables should be prefixed with an underscore followed by your *<OID>* such as:

_mozTable1 or _ccvTableXYZ

CUSTOM CLASSES

We recommend you place your custom code into a Rock.Custom.*<OID >*. namespace such as:

Rock.Custom.CCV.Api or Rock.Custom.JordanRift.App1

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>
```

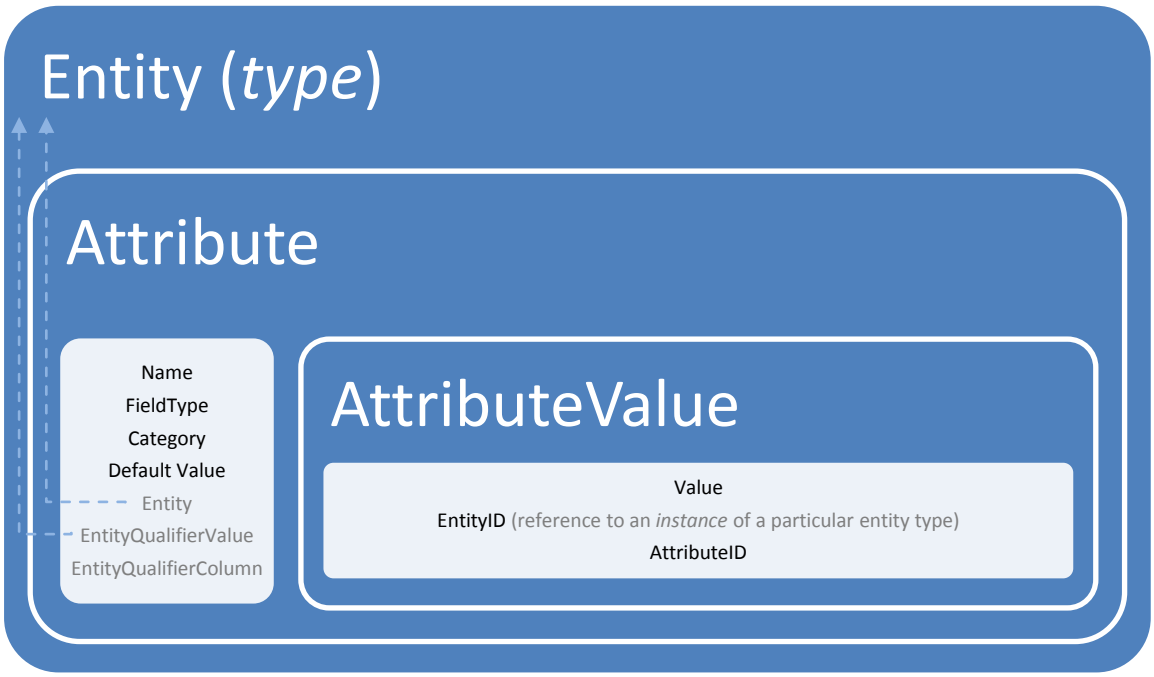
Additional UI guidelines are TBD

INTERNALS

This section is meant for Core developers who want to understand how some internal piece of Rock works. It's primarily to help us wrap our brains around some of the more complex entity/database relationships when where in these early stages of Rock development.

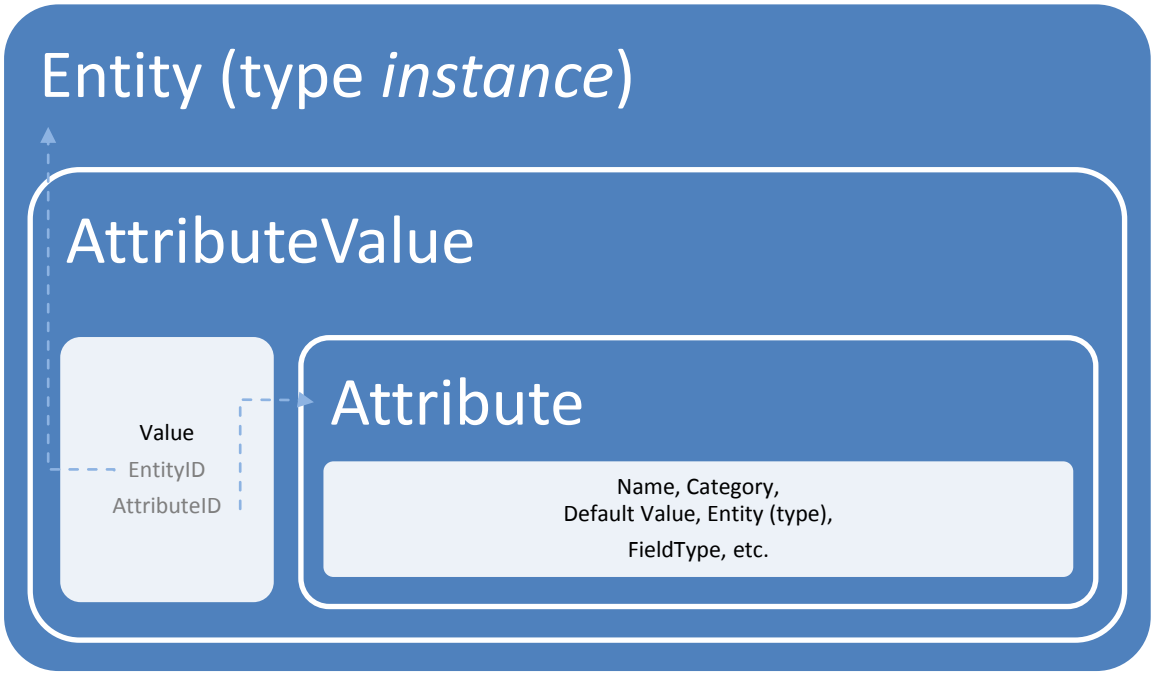
CORE ATTRIBUTES

Attributes (aka Global Attributes) are related to various entities in Rock. Taking an Entity *type* centric viewpoint, they can be seen in this way:



Attributes are named, are of a particular FieldType (i.e. data type), and belong to a particular type of entity (Block, Page, Group, "the system", etc.) as recorded in the Entity property. When necessary, they are further qualified by the EntityQualifierValues and EntityQualifierColumn property (*as deemed/used by the entity type*). An Attribute's Category value, along with some of the other properties, is used when organizing the attribute property UI. Attribute*Values* will have an EntityID which is a reference to a particular *instance* of an entity type (such as the HTML Content block, for example) when deemed necessary by the entity type. Some Attributes have no relationship to entity type *instances* and still other Attributes have no relationship to any entity and therefore can be thought of as global attributes tied to the Rock ChMS system.

When Attributes are related to entity type *instances*, taking an entity type instance centric viewpoint, attributes might be viewed of in this way:

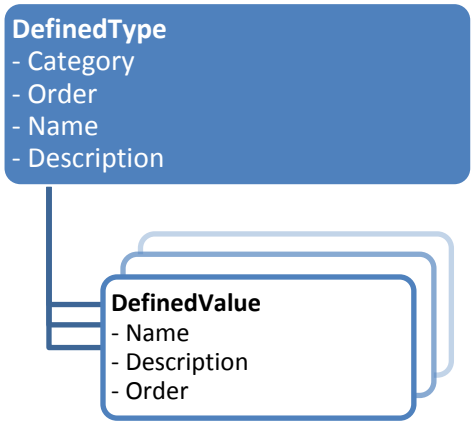


An entity type instance can have one or more AttributeValues of a particular type of Attribute. These Attribute will have a name, category, default value, etc. and specify the particular kind of entity (Page, Block, Group, etc.) to which they belong via the Entity, EntityQualifierValue (the ID of an entity type instance) and EntityQualifierColumn properties.

To use a concrete example, an HTML Content block "entity type instance" has the following Attributes: Pre Text (of fieldtype text), Post Text (of fieldtype text), and Cache Duration (of fieldtype integer) – to name a few; and each of these will have an Entity value of "Rock.CMS.BlockInstance", an EntityQualifierColumn of "BlockID" and EntityQualifierValue that holds the ID of *the* HTML Content block *type*. Each particular *instance* of a HTML Content block will have these AttributeValues and each will store its HTML Content block *instance ID* in the EntityID field/column.

DEFINED TYPES AND VALUES

In Rock, developers can define types of reusable fields and their possible values in a common place (DefinedType and DefinedValue). For example, there is a well-known DefinedType called “Record Status” which has the following DefinedValues: Active, Inactive, and Pending.



CONTEXT AWARE BLOCKS