Rock ChMS  
complete Developer

Reference   
*(movED TO WIKI)*

DRAFT Version: 0.0.

Nick Airdo

David Turner

Jon Edmiston

Last updated: 11/8/2012

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

**Table of Contents**

System Structure Overview 4

Rock (Framework) Project 4

RockWeb WebSite project 6

Rock.DataTransferObjects 7

The Other Projects 7

The Core Rock Components 12

Blocks 12

Pages 12

Themes / Layouts 13

Themes 14

Developing Core Classes 15

Code First 15

Helper Methods 16

Entity Change Logging 16

Developing Custom Blocks 18

Block Instance Properties (BIP) 18

Relative Paths 18

Adding to the Document Head 18

Sharing Objects Between Block Instances 19

Page\_Init vs. OnInit 19

OnInit vs. OnLoad 19

Popup Windows 19

Caching 20

Exception Handling 21

Error Pages 21

Notifications 21

Performance Related Considerations 22

Transactions 22

Global Attributes 24

Merge Fields 24

Namespaces and Conventions 25

Custom Tables 25

Custom Classes 25

Custom API 25

UI Toolkit 26

Rock:Grid 26

UI Standards and Guidelines 28

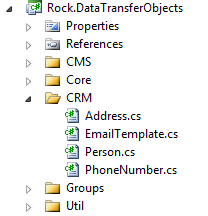
Internals 29

Core Attributes 29

Defined Types and Values 31

Context Aware 31

## 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.

### 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:



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.”

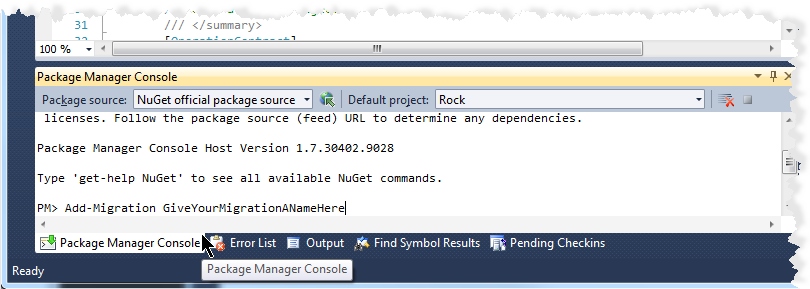
# Developing Core Classes

## 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](http://blogs.msdn.com/b/adonet/archive/2012/02/09/ef-4-3-code-based-migrations-walkthrough.aspx) to better understand 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 that implements 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](http://www.screencast.com/t/Qfz12VkXT) on this topic.

## Helper Methods

### 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.

# 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.

## Context Aware

It is possible to create a Rock Block that has the ability to retrieve data for an entity that exists on the current page. By passing a *fully qualified class name of an entity* to the PageInstance.GetCurrentContext() method, the model for that entity will be returned if it exists (in the page context).

To see this in action, take a look at the Core/ContextAttributeValues block. That block can load attribute values for an entity as seen here:



For example, in the above code you see the entity string (perhaps “Rock.CRM.Person”) is passed to the GetCurrentContext() and if a valid context was provided for the page (such as ?context=*< context\_key>* where *<context\_key>* is an encoded key for a particular person), then the model for that particular person will be returned and the attributes instance values for that person will be added to the page.

### Context Key

An entity’s context key can be obtained via the ContextKey property (inherited from the base Model.cs class).