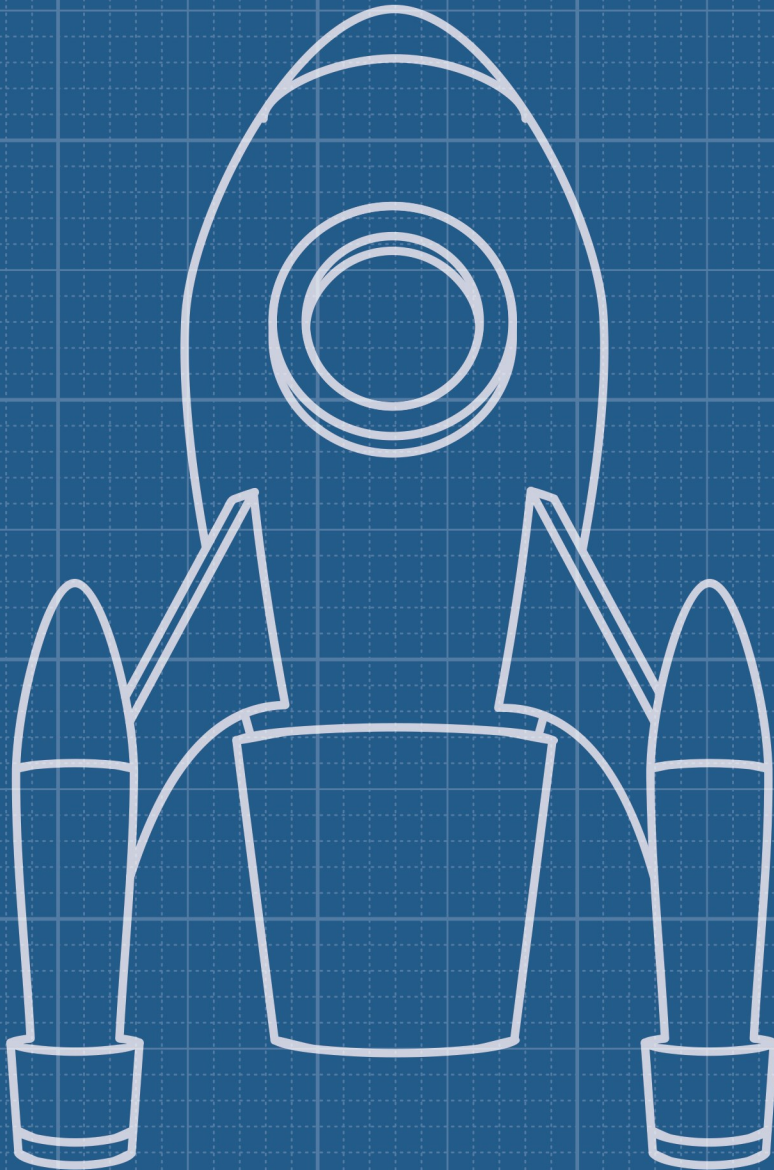





101 LAUNCHPAD

GETTING STARTED WITH ROCK DEVELOPMENT



GETTING STARTED WITH ROCKIT			
		101	

Introduction

Hopefully you've worked through the [QuickStart Tutorials](#) to get exposed to a few of the basics. Rock has so much to offer developers so we've broken down the remaining essentials into a 101, 202 and 303 series.

This book, *101 - Launchpad*, will cover just about everything you would want to know about Rock's most fundamental building component – the Block. We'll also describe the details on how to load and save data for any of Rock's built-in data entities.

In the second book, we'll show you how to save your own custom data. Then we'll explain the primary Rock *Entities* you'll want to become familiar with, as well as important performance topics. Lastly, we'll touch on the other components you can develop for Rock.

In the last book, we'll cover the advanced features to help you go deep with your Rock development.

When finished, you'll have your Rock developer diploma and be ready to start contributing to the Rock store.

A Basic Block

When you look at Rock you're basically viewing a collection of blocks on a page. This was described a bit in the [Designing and Building Websites](#) guide. A well-built block can be used over and over again for slightly different situations. Let's get started with the basics.

A block is simply a small, reusable chunk of functionality, and it can do nearly anything you can dream up. It can be added to one or more pages, and depending how it's designed, it can be added multiple times to the same page. In ASP.NET terms, a block is really just a *User Control* and it is comprised of a file containing the HTML/markup (*.ascx) and a file containing some code (*.ascx.cs).

Example of a block's markup file

```
<%@ Control Language="C#" AutoEventWireup="true" CodeFile="HelloWorld.ascx.cs" Inherits="RockWeb.Plugins.org_RockSolidChurch.Utility.HelloWorld" %>
<h1>Hello World</h1>
```

Example of a block's code-behind file.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;

namespace RockWeb.Plugins.org_RockSolidChurch.Utility
{
    [DisplayName( "Hello World" )]
    [Category( "org_RockSolidChurch > Utility" )]
    [Description( "This is where you put your block's description." )]

    public partial class Plugins_org_RockSolidChurch_HelloWorld : Rock.Web.UI.RockBlock
    {
        protected void Page_Load(object sender, EventArgs e)
        {
            // This is where your code normally goes.
            // Don't use this as an example block! Try looking at the
            // StarkDetail block under the Blocks\Utility folder.
        }
    }

    // more code would be here...
}
```

As seen in the [QuickStart Tutorials](#), all custom Rock blocks inherit from the *Rock.Web.UI.RockBlock* base class. This allows them to be used on a page and have access to properties that tell: what page it's on, who's viewing the block, how to navigate to the parent page, etc. We'll cover those things a bit later in the [Class Properties and Class Methods](#) section.

Those three lines that come before the block's class declaration are important for giving your block a friendly/public name, description and for categorizing your block in Rock's admin screens. The category is especially important for keeping your custom blocks grouped together inside Rock.

Note:

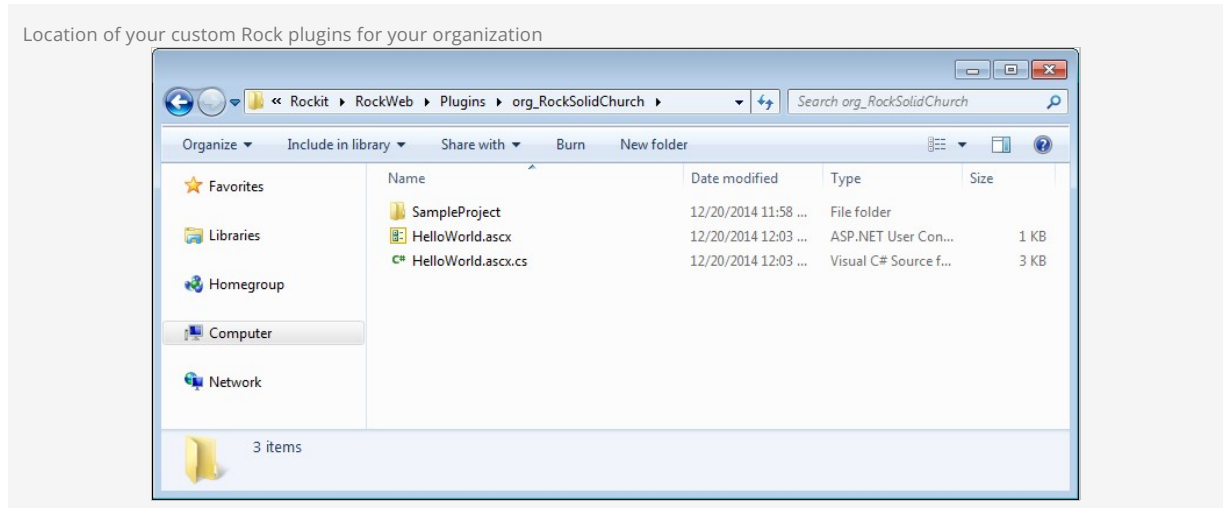
Be sure to follow these [Naming Conventions](#) when creating your own blocks to avoid namespace or filesystem collisions.

Example?

For a more complete example of a block check out the example in the [202 Ignition](#) guide, or the [StarkDetail](#) block, or one of the hundreds of other blocks in the [Blocks](#) folder of your Rock install. Always follow the patterns.

FileSystem Location

Your custom blocks should be stored under the RockWeb\Plugins folder in a sub folder for your organization/domain.



Put your javascripts in a Scripts folder and your CSS stylesheets in a Styles folder as depicted here:

```
RockWeb
 \--Plugins
  \--org_rocksolidchurch
   +--MyProject
    | | WidgetBlock.ascx
    | |
    | +--Assets
    | +--Scripts
    | | widget.js
    | |
    | \--Styles
    | | widget.css
    | |
    +--Scripts
    \--Styles
       myshared.css
```

You can share your common styles and scripts across all your projects by putting them into a **Styles** and **Scripts** folder at your *domain root folder*. However, you will probably want to keep things simple since it could affect the dependencies and complexity of packaging and sharing your work later.

Class Properties

All blocks have the following properties. Some of them, such as **CurrentPerson**, are especially important and you'll find yourself using them often.

- **BlockId** - The Id of the current instance of the block.
- **BlockValidationGroup** - The unique validation group that the framework assigned to all the controls on the block.
- **BreadCrumbs** - The Breadcrumbs created by the page. A block can add additional breadcrumbs as described in the Breadcrumbs section.
- **CurrentPageReference** - URL for the page where the current block is located.
- **CurrentPerson** - The currently authenticated (logged in) person. If this is null, then the person viewing the block is not logged in yet.
- **CurrentPersonAlias** - The alias of the current person. In general you'll want to use the PersonAlias when storing references to people. See the Using PersonAlias vs Person section below for additional details.
- **CurrentPersonAliasId** - The Id of the primary alias of the current person.
- **CurrentPersonId** - The Id of the current person.
- **CurrentUser** - The currently authenticated (logged in) user. This property has other properties such as: UserName and Person.
- **RockPage** - The page the block is currently on.
- **RootPath** - The fully qualified URL path to the root of the website running Rock.

Class Methods

The table below lists all the public methods available to use in your blocks. You can review them quickly since we'll cover some of the more important methods in the sections below.

- **AddConfigurationUpdateTrigger(UpdatePanel upanel)**
Adds an update trigger to the given panel for when the block properties are updated.

- **AddHistory(string key, string state, string title)**
Adds a history point to the ScriptManager.
- **ContextEntity(string entityTypeName)**
Returns the ContextEntity of the entityType specified.
- **ContextEntity()**
Return the ContextEntity for blocks that are designed to have at most one ContextEntity.
- **GetAdministratedControls(bool canConfig, bool canEdit)**
Adds icons to the configuration area of a Block instance. Can be overridden to add additional icons.
- **GetAttributeValue(string key)**
Gets the value of an attribute key.
- **GetAttributeValues(string key)**
Gets the value of an attribute key - splitting that delimited value into a list of strings.
- **GetBreadCrumbs(PageReference pageReference)**
Returns breadcrumbs specific to the block that should be added to navigation based on the current page reference. This function is called during the page's OnInit to load any initial breadcrumbs.
- **GetUserPreference(string key) (Obsolete; see Person Preferences)**
Returns the user preference value for the current user for a given key.
- **GetUserPreferences(string keyPrefix) (Obsolete; see Person Preferences)**
Gets the preferences for the current user where the key begins with the specified value.
- **HideSecondaryBlocks(bool hidden)**
Sets the visibility of the secondary blocks on the page.
- **IsUserAuthorized(string action)**
Evaluates if the CurrentPerson is authorized to perform the requested action.
- **LinkedPageUrl(string attributeKey, Dictionary<string, string> queryParams)**
Builds and returns the URL for a linked page from a 'linked page attribute' and any necessary query parameters.
- **LogException(Exception ex)**
Logs the given exception.
- **NavigateToLinkedPage(string attributeKey, Dictionary<string, string> queryParams)**
Navigate/redirect to a linked page with the given parameters.
- **NavigateToLinkedPage(string attributeKey, string itemKey, int itemKeyValue, string itemParentKey, int? itemParentValue)**
Navigate/redirect to a linked page with the given key, key-value and parent key, parent key-value. This is useful when navigating to a page that has a treeview that needs to expand the given parent item.
- **NavigateToPage(Guid pageGuid, Dictionary<string, string> queryString)**
Navigate/redirect to the page specified by the provided Guid.
- **NavigateToPage(Guid pageGuid, Guid pageRouteGuid, Dictionary<string, string> queryString)**
Navigate/redirect to the Page specified by the given page Guid using the PageRoute specified by the given page route Guid.
- **NavigateToPage(PageReference pageReference)**
Navigate/redirect to the page.
- **NavigateToParentPage(Dictionary<string, string> queryString)**
Navigates/redirects to the parent Page.
- **PageParameter(PageReference pageReference, string name)**
Returns a specified page parameter from the specified PageReference. If a match is not found, an empty string is returned.
- **PageParameter (string name)**
Returns the specified page parameter value. The page's PageRoute is checked first and then query string values. If a match is not found an empty string is returned.
- **PageParameters()**
Returns a Dictionary {String, Object} representing all of the page's page parameters.
- **ResolveRockUrl(string url)**
Resolves a rock URL. Similar to the System.Web.UI.Control ResolveUrl method except that you can prefix a Url with '~/' to indicate a virtual path to Rock's current theme root folder.
- **ResolveRockUrlIncludeRoot(string url)**
Resolves the rock URL and includes its web root.
- **SaveAttributeValues()**
Saves the block attribute values.
- **SetAttributeValue(string key, string value)**
Sets the value of a block attribute key in memory. Once values have been set, use the SaveAttributeValues() method to save all values to database.
- **SetUserPreference(string key, string value) (Obsolete; see Person Preferences)**
Sets a user preference for the current user with the specified key and value.
- **SetValidationGroup(ControlCollection controls, string validationGroup)**
Sets the validation group for the given collection of controls.

Different Ways to Get Input Values

Besides getting input values from regular HTML form controls, your blocks can get input from the QueryString and from stored configuration values we call Block Attributes (aka Block Properties).

QueryString

You're probably pretty comfortable with the QueryString, but we ask that you use the `PageParameter(string)` method as a unified approach for fetching values from the QueryString, page route, etc. This method hides the complexity of having to find the value in any of those places.

```
if ( !Page.IsPostBack )
{
    string itemId = PageParameter( "CategoryId" );
    if ( !string.IsNullOrEmpty( itemId ) )
    {
        ShowDetail( "CategoryId", int.Parse( itemId ) );
    }
    else
    {
        pnlDetails.Visible = false;
    }
}
```

Block Attributes

The Rock framework gives you an amazingly easy way to keep custom configuration settings for each Block. This feature lets you develop Blocks that are flexible, generic and configurable. We call these Block Attributes.

When a Block class is decorated with one or more *Rock.Attribute* field attributes, administrators can set values for each instance of the Block. (And thankfully Rock automatically builds the admin UI for setting the values.)

For example, let's say you were creating a block to cache some data and wanted to let the administrator decide how many minutes to keep it cached. Just add an *IntegerField* attribute above your block class declaration:

```
using Rock.Attribute;
[IntegerField( "Cache Duration", "Number of seconds to cache the content.", false, 0 )]
public partial class HtmlContent : RockBlock
{
    // ...
}
```

Rock's Framework UI allows admins to configure block attributes

Block Properties

Basic Settings Advanced Settings

Name *

Blog Post

Enable Debug ⓘ

Yes

Feed

Cache Duration ⓘ

0

Include RSS Link ⓘ

Yes

RSS Feed Uri *

http://sparkblogs.azurewebsites.net/rock/syndication.axd

Layout

CSS File ⓘ

Template ⓘ *

```
1 {% include '~/Assets/Lava/RSSFeed.lavaItem %}
```

Save Cancel

And in your code, use the `GetAttributeValue(attributeName)` method to get the configured value.

```
int duration = Convert.ToInt32( GetAttributeValue( "CacheDuration" ) );
```

Notice how the key, `CacheDuration`, is simply the *name* specified in the attribution declaration with the spaces removed? This is a convention in Rock.

There are more than 30 different types of Block Attributes to get values such as booleans, campuses, dates, groups, memo and text fields – to name a few.

Note:

You can learn much more about Block Attributes and get a full list of each available type in the [Block Attributes](#) reference guide (coming soon).

Person Preferences

Want to delight the individuals using your block? Save their preference the first time you ask for it. The Rock framework has a mechanism for storing and retrieving preferences (or settings) for the currently logged in person or tracked visitor. For example, let's say your block has options for sorting or filtering, and you'd like to "remember" how the user set them each time they use the block. This is an ideal case for setting and getting *Person Preferences*.

Preferences are not limited to just blocks. You can associate preferences with any entity in Rock. There are also "global" preferences. These are still unique to each person, but they are not attached to any specific entity. Global preferences make a good place to store information that should be shared between many different blocks.

For consistency and future compatibility, keys should be in kebab case. Meaning `grid-page-size-preference` instead of `GridPageSizePreference`.

Accessing Preferences

Accessing preferences is handled by an instance of `PersonPreferenceCollection`. You don't need to create this yourself, it will be handled automatically by the `RockBlock` base class.

`GetBlockPersonPreferences()`

This is the method you will probably use most to access person preferences. It will return a collection configured to store your preferences on the specific block.

`GetBlockTypePersonPreferences()`

There might be cases where you want preferences to be shared between all instances of your block. This method will do that for you. The preference collection that is returned will be scoped to the `BlockTypeId` for you. This means different instances of the same block will be accessing the same preference collection.

`GetGlobalPersonPreferences()`

You can also store preferences globally. They are still unique to the person, but they are not attached to any specific block or entity. These can be useful if you have a preference that needs to be shared between different block types. When using global preferences remember that other developers might be storing data here too so ensure your keys are unique.

`GetScopedPersonPreferences(IEntity scopedEntity)`

If you need to attach preferences to some other entity (rare) you can use this method to scope those preferences to any entity in the system. In fact, if you call this method and pass in the block instance, it is the same as calling `GetBlockPersonPreferences()`. When using preferences returned by this method remember to make the keys unique. Other developers might be storing preferences on these entities too.

Person Preference Collection

Once you have your preference collection, all you need to do is call the methods below to access and update the values.

`SetValue(string key, string value)`

Use this to store a setting value for a given key.

In this example you can see the *Rock:Grid* saving a collection of the current block's filter preferences previously stored for the current user. This preference is specific to the person as well as scoped to the current block instance.

```
int pageSize;
// ...
var preferences = GetBlockPersonPreferences();
preferences.SetValue( "grid-page-size-preference", pageSize.ToString() );
preferences.Save();
```

`GetValue(string key)`

A person's individual preference setting can be retrieved using the `GetValue(string)` method. Here we see the preferred page size setting being fetched. Now the *Rock:Grid* will automatically reselect the last page size the person had selected.

```
// grid-page-size-preference will come from the block instance.
var preferences = GetBlockPersonPreferences();
string pageSize = preferences.GetValue( "grid-page-size-preference" );
```

In another example, we see a block retrieving a *global* preference setting. This would be an example of a preference value that we want to apply to all blocks across the entire system.

```
// default-grid-page-size will come from the individual's global preferences.
var preferences = GetGlobalPersonPreferences();
string defaultPageSize = preferences.GetValue( "default-grid-page-size" );
```

Save()

Save the preference values that have been modified by previously calling `SetValue(string key, string value)`. You don't need to worry about tracking if the value actually changed. That is all done automatically for you. If nothing has changed then `Save()` won't do anything.

This allows you to set a number of preference values and then save them all at once. Otherwise, you would be generating a database call each time you call `SetValue(string key, string value)`.

In this example you can see the *Rock:Grid* saving multiple preference values at once.

```
int pageSize;
bool showDescription;
// ...
var preferences = GetBlockPersonPreferences();
preferences.SetValue( "grid-page-size-preference", pageSize.ToString() );
preferences.SetValue( "grid-show-description", showDescription.ToString() );
preferences.Save();
```

Securing Access to Your Blocks

A block need not worry about hiding itself if a user shouldn't be allowed to view it. The page framework handles that. However, it does have to check security for other situations. Thankfully securing functionality access within your block is easy to do.

To test whether the current user is allowed to perform a certain action, use the `IsUserAuthorized (string action)` method where action is one of "View", "Edit", or "Administrate".

Consider a block that might allow the user to edit some content. Before it allows this, or even shows the edit button, it should use the `IsUserAuthorized(Rock.Security.Authorization.EDIT)` method. Consider this example:

```
if ( IsUserAuthorized(Authorization.EDIT ) || IsUserAuthorized( Authorization.ADMINISTRATE ) )
{
    rGrid.Actions.ShowAdd = true;
    // ...
}
else
{
    message = "You are not allowed to edit this content.";
}
```

Since the `IsUserAuthorized(...)` method is also available on many securable entities in Rock besides Blocks, it can be called to check for authorization against a particular entity. Consider this example where authorization is being checked for a particular group in addition checking general block security:

```
// user must have EDIT to both the Block and the group
if ( IsUserAuthorized( Authorization.EDIT ) &&
    group.IsAuthorized( Authorization.EDIT, this.CurrentPerson ) )
{
    grid.Actions.ShowAdd = true;
}
```

Did you notice how we called our "group" object's `IsAuthorized()` method too?

Note:

You will need to include `using Rock.Security;` in your block. Once you do this, you can then use the `IsUserAuthorized(string action)` method to verify user authorization.

Here are the standard security actions and their meanings:

Standard security action names

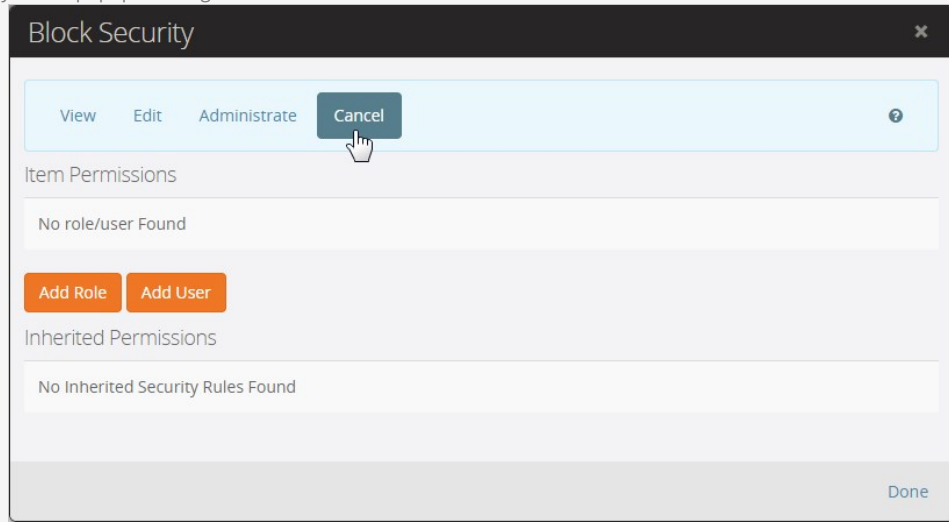
Name	Description
View	Grants the ability to view the item's public properties.
Edit	Includes view access and the ability to change the item's name and other properties.
Administrate	This means the block's security and block's settings can be changed.
Approve	Authorization to approve the item (html, prayer, ads, etc).

If you need to define additional action names to control your custom functionality, you can simply decorate your block with `[SecurityAction(...)]` like this:

```
[SecurityAction( "Cancel", "The roles and/or users that have access to cancel existing orders." )]
```

This will also cause Rock to include your new action in the Block Security settings window so you can allow or deny particular roles or users to the action.

Block Security modal popup showing a custom "Cancel" action.



Validation

When validating a user's input, you'll need to provide some feedback to let them know when they've entered something incorrectly. Use a `ValidationSummary` control at the top of an edit panel with the Bootstrap standard "alert alert-danger" CSS classes:

```
<asp:ValidationSummary ID="ValidationSummary1" runat="server" CssClass="alert alert-danger" />
```

The `RockBlock` base class automatically adds a `ValidationGroup` property unique to each block instance for any `RockControls`, `Validators`, `ValidationSummary` controls, and `Buttons` that you have on your block. If one of these has already had a `ValidationGroup` declared, the `RockBlock` will update it so that it is prefixed with its unique `ValidationGroup` name.

Because of this, you should only need to add a `ValidationGroup` to any areas of your block that are validated separately from the main block (i.e. Modal Dialogs, or Panels that are shown and hidden).

Note:

See the `GroupTypeDetail` block for a good example of how to use validation group for modal dialogs.

Also, while the ASP.NET validators will perform client-side validation, any validation done by Entity Framework (i.e. data annotations and the `DataValidator` used by the `DataTextBox`, and `DataDropDownList` controls) is only done server-side. So if you are validating input from a `ModalDialog`, you may need keep that dialog shown through a postback so that the validation summary can be displayed to the user.

Preventing Validation

You can prevent a button, link, etc. from causing validation by setting the `CausesValidation` property to false:

```
<asp:LinkButton ID="btnCancel" runat="server" Text="Cancel"
  CssClass="btn btn-link" CausesValidation="false" OnClick="btnCancel_Click" />
```

You'll usually want to do this on cancel buttons, etc.

Custom Validation Group

If you need to add your own custom validators on controls, you should set the `ValidationGroup` property on the `ValidationSummary` control and then use that group name in controls on your block:

```
<asp:ValidationSummary ID="valSearchOrganization" runat="server" ValidationGroup="SearchOrg"
  HeaderText="Please Correct the Following" CssClass="alert alert-danger block-message error" />

<Rock:RockTextBox ID="tbSearchPostalCode" runat="server"
  Label="Zip/Postal Code" Required="true"
  RequiredErrorMessage="Enter a zip or postal code to search"
  ValidationGroup="SearchOrg" />

<asp:RegularExpressionValidator ID="regSearchPostalCode" runat="server"
  ErrorMessage="Enter at least 4 characters" ControlToValidate="tbSearchPostalCode"
  ValidationGroup="SearchOrg" ValidationExpression="\w{4,12}"
  CssClass="validation-error" Display="None" />

...
<div class="actions">
  <asp:LinkButton ID="lbSearch" runat="server" Text="Search" CssClass="btn btn-primary" OnClick="lbSearch_Click" ValidationGroup="SearchOrg" />
</div>
```

Loading Entities

Next let's look at how your block will fetch existing entities from Rock. For this we'll use a *Service* class for entity we're trying to load.

Service Layer

Nearly every entity in Rock has a corresponding *<EntityName>Service* class that handles loading (and storing) data from the database. When you use a service class, you'll need to give it a *RockContext* database context so it can communicate with the database.

The service classes will always have a `Queryable()` method for returning an *IQueryable* you can query using LINQ, but they will often have other methods for loading specific sets of data.

Examples

Let's use the *PersonService* class to illustrate a few ways you can get a person or a set of people.

```
// create a person service to perform different queries
var rockContext = new RockContext();
var personService = new PersonService( rockContext );

// a single person based on either their Id or Guid
var person = personService.Get( personId );
var person2 = personService.Get( personGuid );

// a collection of all people (including deceased)
var allPeople = personService.Queryable( includeDeceased: true );

// all males
var allMales = personService.Queryable().Where( p => p.Gender == Gender.Male );

// all people with the given email address
var matchingPeople = personService.GetByEmail( "ted.decker@rocksolidchurch.org" );

// all people with the matching partial phone number
var peopleWithPhone = personService.GetByPhonePartial( "1212" );
```

Certain service classes also return things other than collections of its corresponding entity. Some return a single entity, *related* entities, strings, or whatever is appropriate for that method. For example, the *PersonService* class has a method for getting a collection of families (Groups) for the given person. It also has methods for getting other things related to a person such as *GroupMember*, a *GroupLocation*, a *PhoneNumber*, etc.

```
// a collection of families for the given "person" object
var familyGroups = personService.GetFamilies( person.Id );

// get the person's home phone number
var phone = personService.GetPhoneNumber( person, DefinedValueCache.Read( Rock.SystemGuid.DefinedValue.PERSON_PHONE_TYPE_HOME.AsGuid() ) );
```

Tip!

If you'll be using many services, it is more efficient to create a single instance of a *RockContext* and just pass it along to each service you use.

Eager Loading Properties

You know how some entities have properties that are actually other entities? For example, a *Group* has a property called *Members* which is a collection of *GroupMembers* and a *GroupMember* has a *Person* property. If you know you're going to be querying and using those properties, you'll see significant performance improvements if you pass the property names to the `Queryable()` method.

Consider this example where the group members and member's names will be used to decide which groups you are fetching. Notice how the properties are passed as an *include* string by comma delimiting them if there are more than one. Also notice how you can refer to sub-properties via dot notation.

```
var rockContext = new RockContext();
var groupService = new GroupService( rockContext );

var query = groupService.Queryable( "Members,Members.Person" );
```

```
// only select groups whose members have someone with a nick-name
// that starts with the given value.
if ( !string.IsNullOrEmpty( nickName ) )
{
    query = query.Where( g => g.Members.Any( m => m.Person.NickName.StartsWith( nickName ) ) );
}
}
```

Read Only Queries

You should always add `AsNoTracking()` on your queryable when you're not saving the collection back to the database. Doing so will significantly improve performance since it allows the Entity Framework to ignore certain bookkeeping activities. You'll need to include a `using System.Data.Entity;` to see this method.

```
var query = new PersonService( new RockContext() ).Queryable().AsNoTracking();
```

Just remember - you cannot do this if you are going to call `SaveChanges()` on the service.

ToList() Your DataSource

Make sure to call `ToList()` on your queryable collection if you're going to bind it to a DataGrid or use it as a DataSource somewhere.

```
var query = new PersonService( new RockContext() ).Queryable();

gPeople.DataSource = query.ToList();
gPeople.DataBind();
```


Saving Entities

We're just guessing, but at some point you'll probably need to save an edited entity or add a new one back to the Rock database. The *RockContext* and the service classes we covered in the last chapter are used for doing this. It's pretty simple so we'll just illustrate with a couple of examples.

If you're *updating an existing* entity that you just fetched via a service method, all you need to do is call the `SaveChanges()` method on the *rockContext* you used to construct the service.

```
var rockContext = new RockContext();
var prayerRequestService = new PrayerRequestService( rockContext );

// Fetch an existing prayer request that has the given id
PrayerRequest prayerRequest = prayerRequestService.Get( prayerRequestId );

// Change the approved flag to true and then save it.
prayerRequest.IsApproved = true;
rockContext.SaveChanges();
```

If you're adding a *new* entity you just add it to the service method and then call the `SaveChanges()` method on the *rockContext* you used to construct the service.

```
var rockContext = new RockContext();
var prayerRequestService = new PrayerRequestService( rockContext );

PrayerRequest prayerRequest = new PrayerRequest {
    IsActive = true,
    IsApproved = false,
    AllowComments = false,
    EnteredDateTime = RockDateTime.Now,
    // ...
};
prayerRequestService.Add( prayerRequest );
rockContext.SaveChanges();
```

Caution When Saving Then Attempting to View Entity Properties

When you save a new entity using the service layer, be aware that Entity Framework will not automatically hydrate the related entity properties unless you use a *new service* (with a new *RockContext*) to re-fetch the item.

For example, a *PrayerRequest* has a relationship to a *Category* entity. Consider the case when we're setting a new prayer request's category by setting its `CategoryId` property and then saving it:

```
var rockContext = new RockContext();
var prayerRequestService = new PrayerRequestService ( rockContext );
PrayerRequest prayerRequest = new PrayerRequest {
    CategoryId = 9,
    // ...
};
prayerRequestService.Add( prayerRequest );
rockContext.SaveChanges();
```

You might wish that the prayer request's *Category* property would now reflect the correct category entity, however it does not. It is null. Even if you try to load it again using the same service after saving it, it will be null:

```
// Does not work
prayerRequest = prayerRequestService.Get( prayerRequest.Id ); // Warning!
var category = prayerRequest.Category; // THIS IS NULL
```

Instead, you need to fetch the item using a *new service* if you need the updated property as a fully hydrated entity:

```
// Will work
prayerRequest = new PrayerRequestService( new RockContext() ).Get( prayerRequest.Id ); // Good.
var category = prayerRequest.Category; // Now it's there.
```

Using the Grid and GridFilter

Although the Rock Grid and GridFilter are just two reusable UI controls out of more than a hundred we created for you, they are worth mentioning here since you'll probably use them pretty often.

The markup for a Grid is simple but you also have many other rock controls that can be bound in your grid columns. Here's an example from the SiteList block:

```
<div class="grid grid-panel">
  <Rock:Grid ID="gSites" runat="server" AllowSorting="true" OnRowSelected="gSites_Edit">
    <Columns>
      <Rock:RockBoundField DataField="Name" HeaderText="Name" SortExpression="Name" />
      <Rock:RockBoundField HeaderText="Description" DataField="Description" SortExpression="Description" />
      <Rock:RockTemplateField HeaderText="Domain(s)">
        <ItemTemplate><%=# GetDomains( (int)Eval("Id") ) %></ItemTemplate>
      </Rock:RockTemplateField>
      <Rock:RockBoundField HeaderText="Theme" DataField="Theme" SortExpression="Theme" />
      <Rock:BoolField DataField="IsSystem" HeaderText="System" SortExpression="IsSystem" />
      <Rock:SecurityField TitleField="Name" />
      <Rock:DeleteField OnClick="gSites_Delete" />
    </Columns>
  </Rock:Grid>
</div>
```

In your code you bind the grid's `DataSource` to a set of data as seen below. As you'll see, we're also using the Grid's `SortProperty` to sort the data:

```
SiteService siteService = new SiteService( new RockContext() );
SortProperty sortProperty = gSites.SortProperty;
var qry = siteService.Queryable();
if ( sortProperty != null )
{
  gSites.DataSource = qry.Sort(sortProperty).ToList();
}
else
{
  gSites.DataSource = qry.OrderBy( s => s.Name ).ToList();
}
gSites.DataBind();
```

When it's rendered it looks like this:

Example of a rendered Rock Grid

Name	Description	Domain(s)	Theme	System		
External Website	Site used by attendees to access Rock features.		Stark			
Rock Check-in	The Rock default check-in site		CheckinPark			
Rock Check-in Manager	Rock site to manage check-in		Rock			
Rock RMS	The internal Rock administration site.	localhost	Rock			

50 500 5,000 4 Sites

Tip!

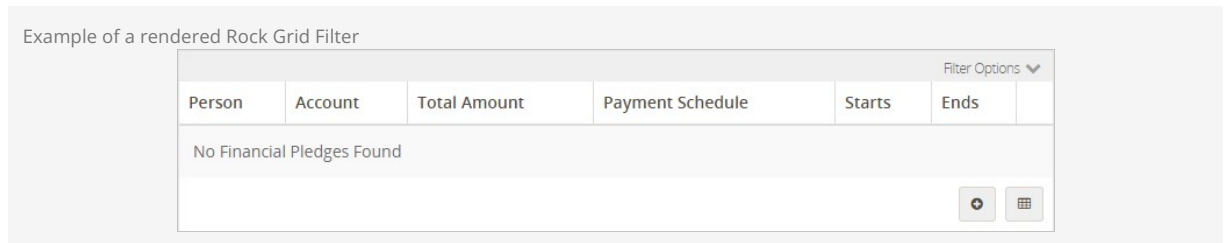
You'll typically want to wrap your Grid in a div with the `grid grid-panel` classes for proper grid styling.

Filter Options

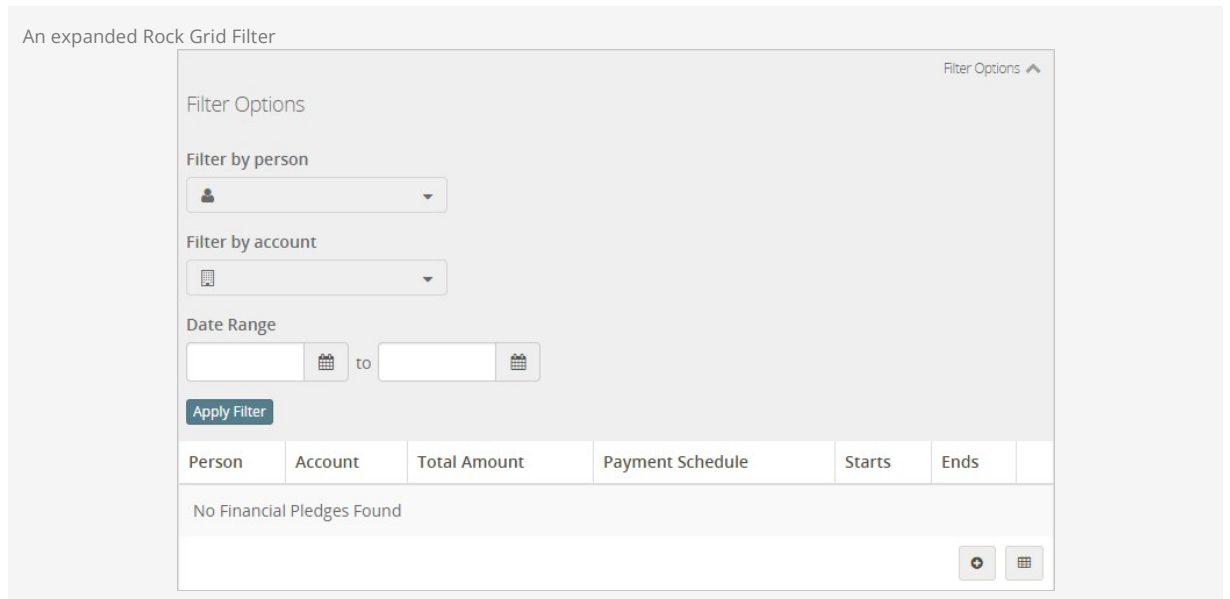
If your grid potentially has too much data, you'll want to include a GridFilter control above your grid. Inside the GridFilter add any controls you'll use to collect filtering criteria from the user. Here is an example as seen in the PledgeList block which uses a PersonPicker to allow filtering by person, an account picker, and a date range picker.

```
<div class="grid grid-panel">
  <Rock:GridFilter ID="gfPledges" runat="server">
    <Rock:PersonPicker ID="ppFilterPerson" runat="server" Label="Filter by person" IncludeBusinesses="true"/>
    <Rock:AccountPicker ID="apFilterAccount" runat="server" Label="Filter by account" AllowMultiSelect="True"/>
    <Rock:DateRangePicker ID="drpDates" runat="server" Label="Date Range" />
  </Rock:GridFilter>
  <Rock:Grid ID="gPledges" runat="server" AutoGenerateColumns="False" AllowSorting="True" AllowPaging="True" OnRowSelected="gPledges_Edit">
    <Columns>
      ...
    </Columns>
  </Rock:Grid>
</div>
```

It will render a small Filter Options menu item in the top right corner above the grid.



Once expanded, it displays all the inner controls in the filter.



Of course, you still need to actually use the selected criteria to filter your data set – but don't worry it's pretty simple. Here's an example from the PledgeList block showing the user's criteria saved in a person preference and then used to filter the data in a `BindGrid()` method:

```
protected void gfPledges_ApplyFilterClick( object sender, EventArgs e )
{
    gfPledges.SetFilterPreference( "date-range", drpDates.DelimitedValues );
    gfPledges.SetFilterPreference( "person", ppFilterPerson.PersonId.ToString() );
    gfPledges.SetFilterPreference( "accounts", apFilterAccount.SelectedValues.ToList().AsDelimited(",") );
    BindGrid();
}

private void BindGrid()
{
    var pledgeService = new FinancialPledgeService( new RockContext() );
    var sortProperty = gPledges.SortProperty;
    var pledges = pledgeService.Queryable();
    int? personId = gfPledges.GetFilterPreference("person").AsIntegerOrNull();
    if ( personId.HasValue )
    {
        pledges = pledges.Where( p => p.PersonAlias.PersonId == personId );
    }
}
```

```
var accountIds = gfPledges.GetFilterPreference( "accounts" ).Split( ',' ).AsIntegerList();
if ( accountIds.Any() )
{
    pledges = pledges.Where( p => p.AccountId.HasValue && accountIds.Contains( p.AccountId.Value ) );
}
// Date Range
var drp = new DateRangePicker();
drp.DelimitedValues = gfPledges.GetFilterPreference( "date-range" );
var filterStartDate = drp.LowerValue ?? DateTime.MinValue;
var filterEndDate = drp.UpperValue ?? DateTime.MaxValue;
// exclude pledges that start after the filter's end date or end before the filter's start date
pledges = pledges.Where( p => !( p.StartDate > filterEndDate ) && !( p.EndDate < filterStartDate ) );
gPledges.DataSource = sortProperty != null ? pledges.Sort( sortProperty ).ToList() : pledges.OrderBy( p => p.AccountId ).ToList();
gPledges.DataBind();
}
```

Note:

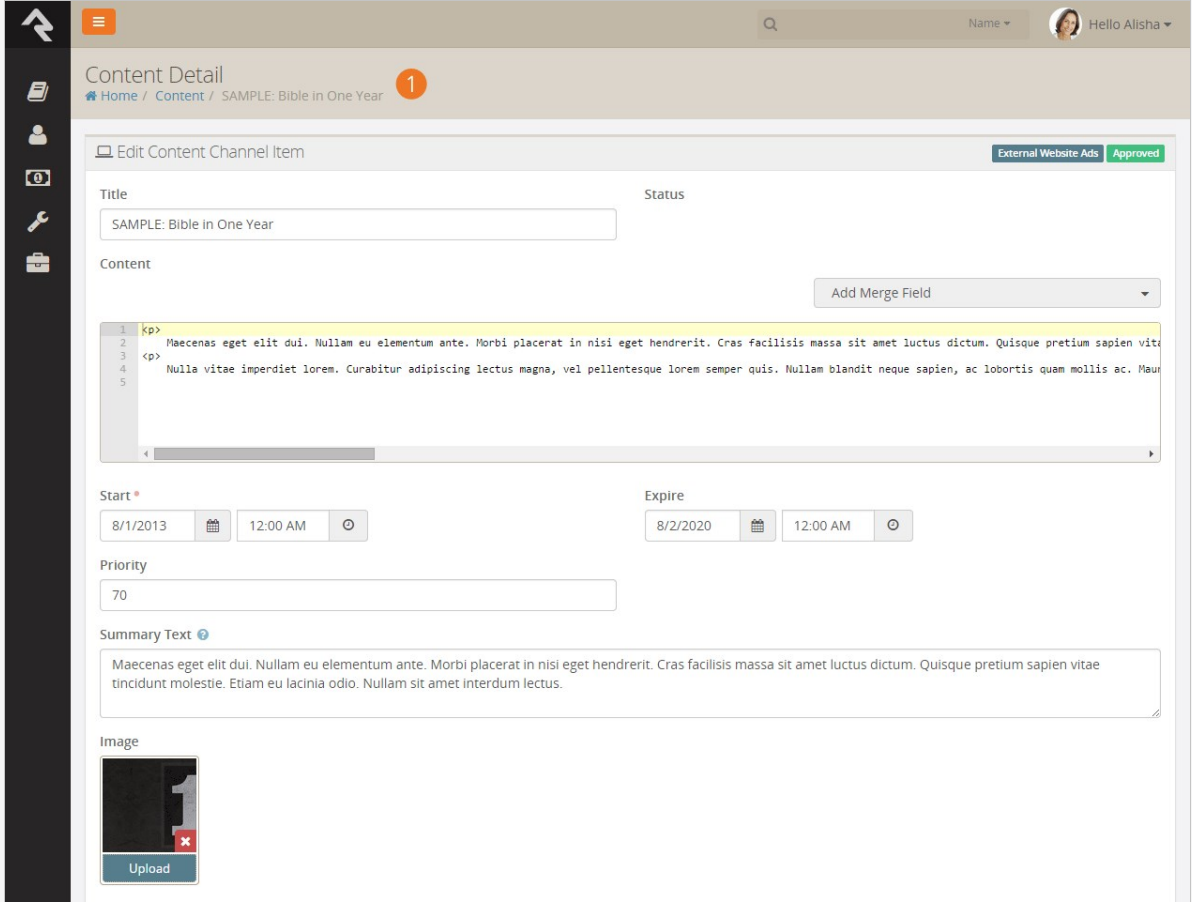
For full details on using the Rock Grid and Grid Filter see the UI Toolkit.

Using Breadcrumbs

Rock has a breadcrumb system which understands the normal parent-child page hierarchy and renders navigation breadcrumbs as you'd expect. You can also control the breadcrumb trail and extend it further as your data/model requires.

For example, consider the case of the Content Channel Item View block (Tools > Content > External Website Ads). As you click on one of the sample items, you'll notice that the last item in the breadcrumb trail changes to the name (title) of the item you clicked.

A block with Breadcrumbs



1 Breadcrumb
Shows a custom breadcrumb that includes the name of the content channel item being edited.

This is possible by overriding the `GetBreadcrumbs(PageReference pageReference)` method in your block.

Override GetBreadcrumbs

Your `GetBreadcrumbs()` implementation will use the given page reference to get the id of the item in question, use that id to get that item's title, and then add a `new Breadcrumb()` onto a list of breadcrumbs that is returned to the caller.

You can see this happening in the `ContentChannelItemView` block:

```
public override List<breadcrumb> GetBreadcrumbs( PageReference pageReference )
{
    var breadCrums = new List<breadcrumb>();
    int? contentItemId = PageParameter( pageReference, "contentItemId" ).AsIntegerOrNull();
```

```

if ( contentItemId != null )
{
    ContentChannelItem contentItem = new ContentChannelItemService( new RockContext() ).Get( contentItemId.Value );
    if ( contentItem != null )
    {
        breadcrumbs.Add( new Breadcrumb( contentItem.Title, pageReference ) );
    }
    else
    {
        breadcrumbs.Add( new Breadcrumb( "New Content Item", pageReference ) );
    }
}
else
{
    // don't show a breadcrumb if we don't have a pageparam to work with
}
return breadcrumbs;
}

```

Controlling Page Breadcrumbs Display Settings

Breadcrumbs can be disabled on a page and you can also control other aspects of the breadcrumbs under the Page Properties > Display Settings.

Controlling Breadcrumbs via Page Properties

The screenshot shows a 'Page Properties' dialog box with the 'Display Settings' tab selected. The settings are as follows:

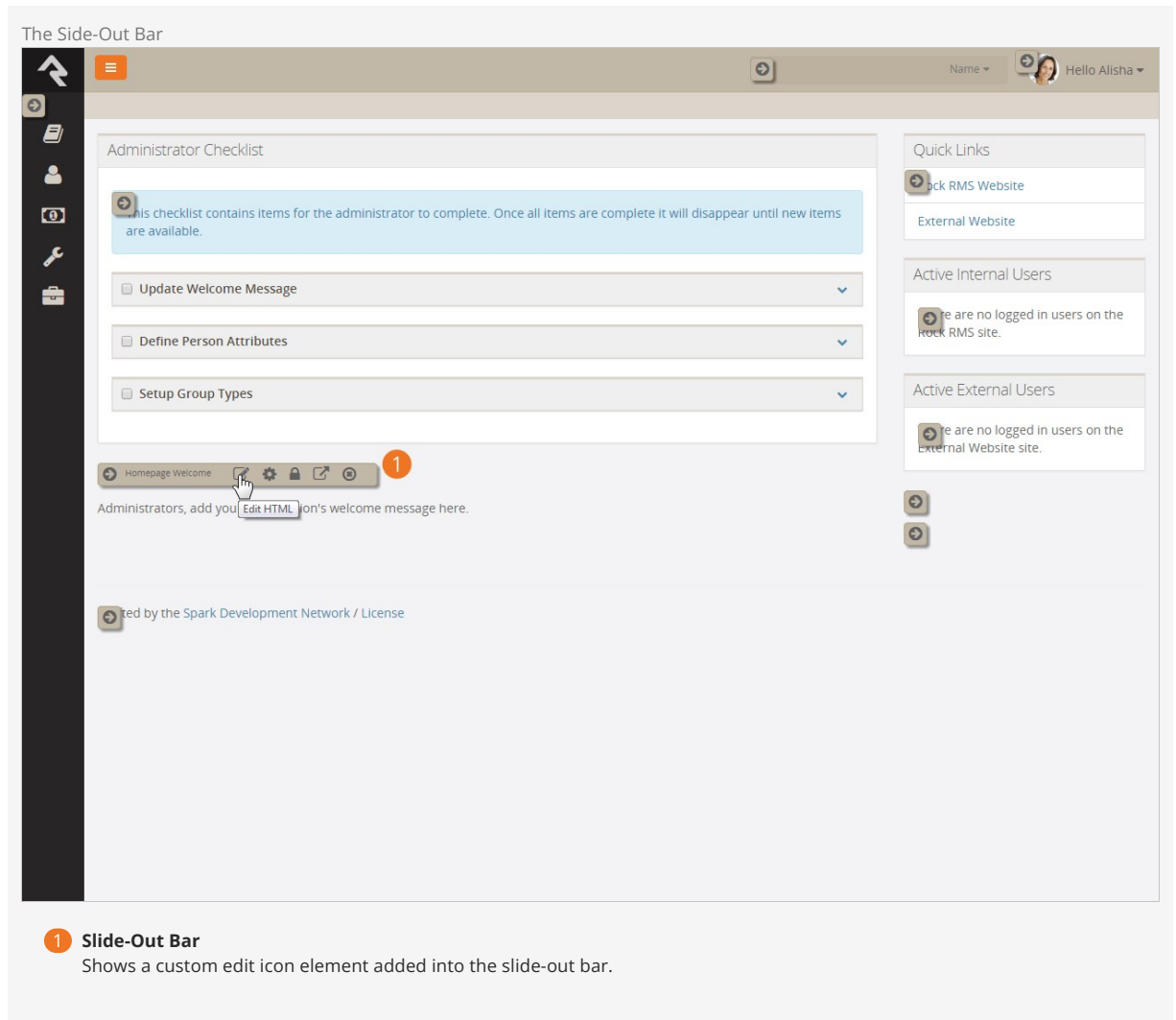
- Page:**
 - Show Title on Page
 - Show Breadcrumbs on Page
 - Show Icon on Page
 - Show Description on Page
- Menu:**
 - Display When: When Allowed
 - Show Description
 - Show Child Pages
- Breadcrumbs:**
 - Show Name in Breadcrumb
 - Show Icon in Breadcrumb

Buttons: Save, Cancel

The Block Configuration Slide Out Bar

Your Own Edit Button

To use the edit button in the slide-out bar that opens when you click on the Block Configuration toolbar you can extend the `RockBlockCustomSettings` class instead of the usual `RockBlock` class. Once you do this, you only need to override and implement your own `ShowSettings()` method. That method will be called by Rock when a person with edit access clicks the edit button.



The Side-Out Bar

1 **Slide-Out Bar**
Shows a custom edit icon element added into the slide-out bar.

This example code from the DynamicReport block illustrates the concept. Here the block loads some data, binds some filters and then calls the `Show()` method of a model where all the custom stuff exists.

```
protected override void ShowSettings()
{
    pnlConfigure.Visible = true;
    LoadDropDowns();
    ddlReport.SetValue( this.GetAttributeValue( "Report" ).AsGuidOrNull() );
    txtResultsTitle.Text = this.GetAttributeValue( "ResultsTitle" );
    txtResultsIconCssClass.Text = this.GetAttributeValue( "ResultsIconCssClass" );
    txtFilterTitle.Text = this.GetAttributeValue( "FilterTitle" );
    txtFilterIconCssClass.Text = this.GetAttributeValue( "FilterIconCssClass" );
    BindDataFiltersGrid( false );
    ddlPersonIdField.SetValue( this.GetAttributeValue( "PersonIdField" ) );
}
```

```
mdConfigure.Show();  
}
```

Additionally, any block property you've given a category of *CustomSetting* will be hidden in the normal block settings. This is handy when you want to do something beyond the basics provided by the normal block settings area.

I Need More Control, Captain!

If you need more than a single edit pencil button, you can insert your own controls into the slide-out bar by overriding the `GetAdministratedControls()` method. This is exactly how the `RockBlockCustomSettings` class did it to get an edit button in the slide-out bar for you in the simple case.

This example code from a block illustrates the details:

```
public override List<control> GetAdministratedControls( bool canConfig, bool canEdit )  
{  
    List<control> configControls = new List<control>();  
    // add edit icon to config controls if user has edit permission  
    if ( canConfig || canEdit )  
    {  
        LinkButton lbEdit = new LinkButton();  
        lbEdit.CssClass = "edit";  
        lbEdit.ToolTip = "Edit HTML";  
        lbEdit.Click += new EventHandler( lbEdit_Click );  
        configControls.Add( lbEdit );  
        HtmlGenericControl iEdit = new HtmlGenericControl( "i" );  
        lbEdit.Controls.Add( iEdit );  
        lbEdit.CausesValidation = false;  
        iEdit.Attributes.Add("class", "fa fa-pencil-square-o");  
        ScriptManager.GetCurrent( this.Page ).RegisterAsyncPostBackControl( lbEdit );  
    }  
    configControls.AddRange( base.GetAdministratedControls( canConfig, canEdit ) );  
    return configControls;  
}
```

In a nutshell, here's what's going on:

1. Create an empty list of controls.
2. Use the `canConfig` and `canEdit` boolean flags to decide if you want to add your items.
3. Create an appropriate control with an event handler.
4. Add it to the list of controls.
5. Register the control with the script manager.
6. IMPORTANT: add the standard set of controls by calling the `base.GetAdministratedControls()` method (unless you decide it's not appropriate for your block).
7. Return the list of controls.

That's all there is to it -- well, almost. You still have to write the code to actually do something when someone clicks the button you've added. In the example above, you would need to implement the `lbEdit_Click()` method.

Dates and Times

When developing a framework that works across time zones and internationally, you've got to be careful with how you store and present dates and times. This chapter will show you what you need to know about all things date and time in Rock.

RockDateTime vs DateTime

Because a Rock *server* is not always running or is not configured to be in the same time zone of the organization using Rock, you should *always* avoid creating new date/times or the current time (aka ".Now") using the standard system *DateTime*

Use the `RockDateTime.Now` or `RockDateTime.Today` static class methods when creating new date/times.

```
// Get the current date time
DateTime now = RockDateTime.Now;

// Get only the date component for today (wherever today is)
DateTime today = RockDateTime.Today;
```

If you're in need of checking the date/time of a file on the file-system, you should use the `ConvertLocalDateTimeToRockDateTime(...)` method to convert the time to a correctly adjusted time.

```
DateTime fileDateTime = File.GetCreationTime( file );
DateTime adjustedDateTime = RockDateTime.ConvertLocalDateTimeToRockDateTime( fileDateTime );
```

These methods always work because organizations set their particular time zone when they first install Rock. Behind the scenes, Rock stored a `OrgTimeZone` property into their *web.config* which is used by the Rock DateTime helper methods.

DateTime Formatting

For people, there is not one *correct* way to format dates and times. In some countries "MM/DD/YYYY" is the standard while in others it's "DD/MM/YYYY" or "YYYY-MM-DD" or one of hundreds of other styles. However, using the incorrect format isn't just a small issue, in some cases it will change the meaning of the date value to the viewer.

For the most part, the framework displays dates and times using the culture setting of the viewer's browser. However, to achieve this, when a date is stored in a `string` field (such as a string Attribute) you must set that string using the "Round-trip" (aka ISO 8601) format specifier "o" as shown here:

```
// set the value to the current time
thing.SetAttributeValue( attrKey, RockDateTime.Now.ToString( "o" ) );
```

Rock.DateTime for SQL Migrations

If you're using `Rock.DateTime` for SQL migrations, be sure to use the ISO format for datetime strings before writing to SQL, especially in interpolated strings. For example, use `'{RockDateTime.Now:s}'` instead of `'{RockDateTime.Now}'`.

Date, Time, DateTime Comparison

When writing LINQ (or SQL) queries where there is a Date, Time, Date/Time, or Numeric comparison involved, it's important to be consistent in how they are written. For example, if the user wants to filter the a list of records using a Date/Time range, do you use `>=` and `<` or `>` and `<=?`.

In general, we say let your *start* be "inclusive" and your *end* be "exclusive" as illustrated in each of the examples below.

Warning: There's No BETWEEN

Don't use the `BETWEEN` operator in SQL because it is fully inclusive and is in conflict with our "let your end be exclusive" rule.

DateTime range

```
var qry = new MyService().Queryable();

DateTime startDateTime = DateTime.Parse("11/1/2012 01:00 pm");
DateTime endDateTime = DateTime.Parse("11/2/2012 01:00 pm");
```

```
// Get the records equal to and greater than the StartDateTime, but just less than (but not equal)
// the EndDateTime.
qry = qry.Where( a => a.DateTime >= startDateTime && a.DateTime < endDateTime );
```

Date range

```
var qry = new MyService().Queryable();

DateTime startDate = DateTime.Parse("11/1/2012");
DateTime endDate = DateTime.Parse("11/2/2012");

// For a little extra safety, since we are doing a Date comparison (not a DateTime comparison)
// get just the Date portion without the time (just in case).
startDate = startDate.Date;
endDate = endDate.Date;

// Get the records equal to and greater than the StartDate, but add a whole day to
// the selected endDate since users will expect to see all the stuff that happened
// on the endDate up until the very end of that day.

// calculate the query endDate before including it in the qry statement to avoid Linq error
endDate = endDate.AddDays(1);

qry = qry.Where( a => a.DateTime >= startDate && a.DateTime < endDate );
```

Specific Date (of a DateTime column)

```
var qry = new MyService().Queryable();

DateTime startDate = DateTime.Parse("11/1/2012");

// For a little extra safety, since we are doing a Date comparison (not a DateTime comparison)
// get just the Date portion without the time.
// (just in case)
startDate = startDate.Date;

// When querying for stuff that occurred on a specific date, when the data is a DateTime,
// just add a day to the specific date to get records for the entire day.

// calculate the query endDate before including it in the qry statement to avoid Linq error
DateTime endDate = startDate.AddDays(1);

qry = qry.Where( a => a.DateTime >= startDate && a.DateTime < endDate );
```

Specific Date (of a Date column)

```
var qry = new MyService().Queryable();

DateTime specificDate = DateTime.Parse("11/1/2012");

// Since we are doing a Date comparison (not a DateTime comparison) get just the Date
// portion without the time (just in case).
specificDate = specificDate.Date;

qry = qry.Where( a => a.LogDate == specificDate );
```

Time Range (when querying off of a Time column)

```
var qry = new MyService().Queryable();

TimeSpan starttime = TimeSpan.Parse("01:00 pm");
TimeSpan endTime = TimeSpan.Parse("02:00 pm");

// Get the records equal to and greater than the startTime, but just less
// than (but not equal) the endTime.
qry = qry.Where( a => a.Time >= startTime && a.Time < endTime );
```

Time Range (when querying off of a DateTime column)

```
var qry = new MyService().Queryable();

TimeSpan starttime = TimeSpan.Parse("01:00 pm");
TimeSpan endTime = TimeSpan.Parse("02:00 pm");

// We can't do DateTime.TimeOfDay in a queryable, so fetch it into a list first.
var list = qry.ToList();

// Get the records equal to and greater than the startTime, but just less than
// (but not equal) the endTime.
list = list.Where( a => a.DateTime.TimeOfDay >= startTime && a.DateTime.TimeOfDay < endTime)
```

Numeric range

```
var qry = new MyService().Queryable();

int startValue = int.Parse("100");
int endValue = int.Parse("150");

// Unless the UI has some text that explicitly states how the values are going to be
```

```
// compared, simply get the records equal to and greater than the startValue and
// less than or equal to the endValue. In other words, for integer comparison, do
// an "inclusive" compare

qry = qry.Where( a => a.Rating >= startValue && a.Rating <= endValue; )
```

Referencing Files and Images

There are two ways to fetch (reference) files and images from storage: using `GetFile.ashx` or `GetImage.ashx` while passing either the ID or GUID as shown in this example:

```
# Getting a file
GetFile.ashx?guid=735A97A3-7A9A-4D3F-BE86-B3874E85E141

# Getting an image
GetImage.ashx?id=3
```

The final reference might look like this `` in your block.

Using the `GetFile.ashx` handler to fetch large files is fast and efficient because the file is streamed from the storage asynchronously and is not loaded into memory to do so.

Using the `GetImage.ashx` handler gives you additional features such as resizing, rotating, as well as it will cache those transformations. To rotate an image simply pass `rotate` with the number of degrees. To resize an image, simply pass `height` and `width` parameters or `maxheight` and `maxwidth`. Using the latter will maintain the aspect ratio while using the former will force the image into the provided dimensions.

```
# Rotate 45 degrees
GetImage.ashx?guid=735A97A3-7A9A-4D3F-BE86-B3874E85E141&rotate=45

# Resize forcing into 300 x 200
GetImage.ashx?id=3&width=300&height=200
```

Open-Source Rocks!

Image resizing and rotation are provided by the ImageResizer project (v4). Their complete reference can be found on the ImageResizer project website.

QR Codes

There is also a `GetQRCode.ashx` handler to dynamically generate QR codes that can be embedded in web pages, printed on materials, etc. The QR codes are generated based on the text or URL you provide through the query string parameters.

To use this handler, you need to provide the `data` parameter in the query string, which will be encoded into the QR code.

The following parameters are supported by this handler:

- **data** - The text or URL to encode in the QR code. This is a required parameter.
- **outputType** - The image type to generate; either `svg` or `png`. (Default is `png`)
- **pixelsPerModule** - The pixel size each b/w module is drawn. (Default is 20)
- **foreground** - The color hex value to use for the foreground (b) color. Transparency is supported using RGBA values. (Default is `#000000`) v17.0
- **background** - The color hex value to use for the background (w) color. Transparency is supported using RGBA values. (Default is `#FFFFFF`) v17.0

```
/GetQRCode.ashx?data=ABC&background=%23FF5C0077
```

The example above generates a QR code that encodes the text `ABC` to output as a PNG with a semi-transparent orange background color.

UI Toolkit

As mentioned earlier, we've created over a hundred reusable UI controls to help you write-less-code(TM) and get on with the task of solving your "business" problems. There are just too many to cover so we decided to put them in their own book. You'll definitely need to spend some time becoming familiar with these before you start writing any serious blocks.

UI Toolkit

Note:



The UI Toolkit book isn't finished yet. In the mean time you can check out the [Control Gallery](#) block on your Rock instance or on the [Demo site](#).

Using PersonAlias vs Person

Over the course of time, duplicate person records happen and then they're merged. Because of this, avoid storing the *Id* of a Person. Instead, use the Id of their PersonAlias. *In other words your custom data models should reference the PersonAlias and not the Person object.* PersonAlias records will always exist (even after a merge) and each one has a PersonId which points to the correct Person.


Consider this example where code is referencing PersonAlias Ids 100 and 101 which point to two Tom Miller records with Person Id 16 and 17, respectively.

A representation of two duplicate person records and their aliases.

Your Reference	PersonAlias Id -> Person Id	Person (Id)
100	100 -> 16	 (16)
101	101 -> 17	 (17)

After the two Tom Miller records are merged, the PersonAlias references are intact and point to the Tom Miller whose PersonId is 16.

A representation of the person records and their aliases after a merge.

Your Reference	PersonAlias Id -> Person Id	Person (Id)
100	100 -> 16	 (16)
101	101 -> 16	

Comment:

The fact that person Id 16 survived the merge is somewhat arbitrary. It's based on the record that was selected as the 'primary' when they were merged. The important thing to note is this: if you were referencing 17 in your code, things would not be good.

So, in code, when you've got a person's alias id (aka AliasPersonId) and you need to find the person record you can use the PersonAliasService's `GetByAliasId(int aliasId)` method.

```
// use the PersonAliasService
var rockContext = new RockContext();
var personAliasService = new PersonAliasService( rockContext );

// get the PersonAlias record using one of their known aliasIds
var personAlias = personAliasService.GetByAliasId( 101 );

// the Id of the PersonAlias (which you will usually want to associate to another entity)
int id = personAlias.Id;

// the person's Person record
var person = personAlias.Person;
```

Extension Methods

Extension methods are a great way to add functionality to existing .Net classes and we made some to make your life easier. Just include the "using Rock;" in your code. You should definitely review the full list in the `Rock.ExtensionMethods` class, but we've highlighted most of them here for your reference.

Boolean Extensions

- **Bit()** - Returns a numeric 1 (if true) or 0 (if false).
- **ToYesNo()** - Returns either "Yes" or "No".
- **ToTrueFalse()** - Returns the string "True" or "False".

CheckBoxList Extensions

- **SetValues()** - Sets the Selected property of each item to true for each given matching values.

Control Extensions

- **LoadControl()** - Loads a user control using the constructor with the parameters specified.
- **RockBlock()** - Gets the parent RockBlock.
- **ParentUpdatePanel()** - Returns the parent update panel for the given control (or null if none is found).
- **ControlsOfTypeRecursive<T>()** - Gets all controls of Type recursively <http://stackoverflow.com/questions/7362482/c-sharp-get-all-web-controls-on-page>.

DateTime Extensions

- **Age()** - Returns the age at the current date
- **TotalMonths()** - The total months.
- **TotalYears()** - The total years.
- **ToElapsedString()** - Returns a friendly elapsed time string.
- **ToRelativeDateString()** - Returns a string in FB style relative format (x seconds ago, x minutes ago, about an hour ago, etc.) or if max days has already passed in FB datetime format (February 13 at 11:28am or November 5, 2011 at 1:57pm).
- **ToJavaScriptMilliseconds()** - Converts the date to an Epoch of milliseconds since 1970/1/1.
- **ToMonthDayString()** - Converts the date to a string containing month and day values (culture-specific).

Dictionary (liquid) extension methods

- **Update()** - Adds a new key/value to dictionary or if key already exists will update existing value.

Dictionary extension methods

- **AddOrReplace<TKey, TValue>()** - Adds or Replaces an item in a Dictionary.
- **AddOrIgnore<TKey, TValue>()** - Adds an item to a Dictionary if it doesn't already exist in Dictionary.

Enum Extensions

- **ConvertToString()** - Converts to the enum value to it's string value
- **GetDescription()** - Gets the enum description.
- **ConvertToInt()** - Converts to int.
- **ConvertToEnum<T>()** - Converts a string value to an enum value.

```
// Assuming you've got an enum such as:
public enum Status
{
    Success,
    Fail,
    EpicFail,
}

// elsewhere, you can get the enum if you've only got the string value:
Status s = "Epic Fail".ConvertToEnum(); // w00t!
```

- **ConvertToEnumOrNull<T>()** - Converts to enum or null.

GenericCollection Extensions

- **AsDelimited<T>()** - Concatonate the items.
- **AsGuidList()** - Converts a List to List only returning items that could be converted to a guid.
- **AsIntegerList()** - Converts a List to List only returning items that could be converted to a int.
- **Join()** - Joins a dictionary of items.

Geography extension methods

- **Coordinates ()** - Coordinatases the specified geography.

HiddenField Extensions

- **ValueAsInt()** - Values as int.
- **SetValue()** - Sets the value.
- **IsZero()** - Determines whether the specified hidden field is zero.

HtmlControl Extensions

- **AddCssClass()** - Adds a CSS class name to an html control.
- **RemoveCssClass()** - Removes a CSS class name from an html control.

IEntity extensions

- **RemoveEntity<T>()** - Removes the entity.

IHasAttributes extensions

- **LoadAttributes()** - Loads the attributes.
- **SaveAttributeValues()** - Saves the attribute values.
- **CopyAttributesFrom()** - Copies the attributes.

Int Extensions

- **DefinedValue()** - Gets the Defined Value name associated with this id.

ListControl Extensions

- **SetReadOnlyValue()** - Sets the read only value.
- **BindToEnum<T>()** - Binds to enum.
- **BindToDefinedType()** - Binds to the values of a definedType.

```
ListControl ddl = new ListControl();  
ddl.BindToDefinedType( DefinedTypeCache.Read( myConfiguredDefinedTypeGuid ) );
```

- **SelectedValueAsInt()** - Returns the Value as Int or null if Value is.
- **SelectedValueAsId()** - Returns the value of the currently selected item. It will return NULL if either or is selected.
- **SelectedValueAsEnum<T>()** - Selecteds the value as enum.
- **SelectedValueAsEnumOrNull<T>()** - Selecteds the value as enum or null.
- **SelectedValueAsGuid()** - Selecteds the value as unique identifier.

Object Extensions

- **Tojson()** - Converts object to JSON string.
- **GetPropertyValue()** - Gets the property value.
- **ToStringSafe()** - Safely ToString() this item, even if it's null.
- **JavaDebugInfo()** - Returns an html representation of object that is available to Lava.

Route Extensions

- **AddPageRoute()** - Adds the page route.

Stream extension methods

- **ReadBytesToEnd()** - Reads entire stream and converts to byte array.

String Extensions

- **RemoveSpecialCharacters()** - Removed special characters from strings.
- **SplitCase()** - Splits a Camel or Pascal cased identifier into seperate words.

- **SplitDelimitedValues()** - Returns a string array that contains the substrings in this string that are delimited by any combination of whitespace, comma, semi-colon, or pipe characters.
- **GetListItems()** - Returns a List of ListItems that contains the values/text in this string that are formatted as either 'value1,value2,value3,...' or 'value1^text1,value2^text2,value3^text3,...'
- **ReplaceCaseInsensitive()** - Replaces every instance of oldValue (regardless of case) with the newValue. from <http://www.codeproject.com/Articles/10890/Fastest-C-Case-Insensitive-String-Replace> .
- **ReplaceWhileExists()** - Replaces every instance of oldValue with newValue. Will continue to replace values after each replace until the oldValue does not exist.
- **EscapeQuotes()** - Adds escape character for quotes in a string.
- **Quoted()** - Adds Quotes around the specified string and escapes any quotes that are already in the string.
- **Left()** - Returns the specified number of characters, starting at the left side of the string.
- **Truncate()** - Truncates a string after a max length and adds ellipsis. Truncation will occur at first space prior to maxLength.
- **Pluralize()** - Pluralizes the specified string.
- **Singularize()** - Singularizes the specified string.
- **AsNumeric()** - Removes any non-numeric characters.
- **AsBoolean()** - Returns True for 'True', 'Yes', 'T', 'Y', '1' (case-insensitive).
- **AsBooleanOrNull()** - Returns True for 'True', 'Yes', 'T', 'Y', '1' (case-insensitive), null for emptystring/null.
- **AsInteger()** - Attempts to convert string to integer. Returns 0 if unsuccessful.
- **AsIntegerOrNull()** - Attempts to convert string to an integer. Returns null if unsuccessful.
- **AsGuid()** - Attempts to convert string to Guid. Returns Guid.Empty if unsuccessful.
- **AsGuidOrNull()** - Attempts to convert string to Guid. Returns null if unsuccessful.
- **IsEmpty()** - Determines whether the specified unique identifier is Guid.Empty.
- **AsDecimal()** - Attempts to convert string to decimal. Returns 0 if unsuccessful.
- **AsDecimalOrNull()** - Attempts to convert string to decimal. Returns null if unsuccessful.
- **AsDouble()** - Attempts to convert string to double. Returns 0 if unsuccessful.
- **AsDoubleOrNull()** - Attempts to convert string to double. Returns null if unsuccessful.
- **AsDateTime()** - Attempts to convert string to DateTime. Returns null if unsuccessful.
- **AsTimeSpan()** - Attempts to convert string to TimeSpan. Returns null if unsuccessful.
- **ResolveMergeFields()** - Use DotLiquid to resolve any merge codes within the content using the values in the mergeObjects.
- **HasMergeFields()** - Determines whether [has merge fields] [the specified content].
- **FormatAsHtmlTitle()** - Converts string to a HTML title "first-word rest of string".
- **ConvertCrLfToHtmlBr()** - Converts CR (carriage return) LF (line feed) to non-encoded html breaks (br).
- **ConvertBrToCrLf()** - Converts the HTML br to cr lf.
- **EncodeHtml()** - HTML Encodes the string.
- **SanitizeHtml()** - Sanitizes the HTML by removing tags. If strict is true, all html tags will be removed, if false, only a blacklist of specific XSS dangerous tags and attribute values are removed.
- **ScrubHtmlAndConvertCrLfToBr()** - Scrubs any html from the string but converts carriage returns into html suitable for web display.
- **IsValidEmail()** - Returns true if the given string is a valid email address.
- **AsType<T>()** - Converts the value to Type, or if unsuccessful, returns the default value of Type.
- **Masked()** - Masks the specified value if greater than 4 characters (such as a credit card number). For example, the return string becomes "*****6789".
- **EnsureTrailingBackslash()** - Ensures the trailing backslash. Handy when combining folder paths.
- **EnsureTrailingForwardslash()** - Ensures the trailing forward slash. Handy when combining url paths.
- **IfEmpty()** - Evaluates string and if null or empty returns nullValue instead.
- **CompareTo()** - Compares to the given value returning true if comparable.

TimeSpan Extensions

- **ToTimeString()** - Returns a TimeSpan to HH:MM AM/PM. Examples: 1:45 PM, 12:01 AM

Type Extensions

- **GetFriendlyTypeName()** - Gets the name of the friendly type.

Where Extensions

- **Where<T>()** - Queries a list of items that match the specified expression.
- **OrderBy<T>()** - Orders the list by the name of a property.
- **OrderByDescending<T>()** - Orders the list by the name of a property in descending order.
- **ThenBy<T>()** - Then Orders the list by the name of a property.
- **ThenByDescending<T>()** - Then Orders the list by a a property in descending order.
- **Sort<T>()** - Sorts the object by the specified sort property.
- **WhereAttributeValue<T>()** - Filters a Query to rows that have matching attribute value.

Internal Features

Rock Internal

If a class, method, or property is decorated with the [RockInternal] attribute, then it should not be used.

Naming Conventions

Here are some rules to follow so your stuff doesn't collide with other's stuff. We've decided to standardize on using your organizations's domain name as the main naming prefix. Keep reading, you'll see what we mean below.

Quick Reference

Lowercase Domain

In all cases below the "domain" (reversed.domain, etc.) MUST be lowercase.

Item	Rule	Example
Project Naming	<reversed.domain>.<Project Name>	org.mychurch.MyProjectName (domain is always lowercase)
Tables	_ <u>reversed_domain</u> _.<project>.<tablename>	_org_mychurch_MyProjectName_Book
Stored Procedures	_ <u>reversed_domain</u> _.sp<project and/or component>	_org_mychurch_spWidgetGet
User Functions	_ <u>reversed_domain</u> _.unf<project and/or component>	_org_mychurch_unfWidgetCalc
Action Categorization	<Organization Name>: <category name>	[Category("My Church: MyActionCategory")] or include a locality/identifier if your organization's name is common [Category("My Church (AZ): MyActionCategory")]
Lava Shortcodes	<reversed_domain>.<shortcode tag-name>	{[org_mychurch_supermap ...]}
Block Categorization	<Organization Name> > <category name>	[Category("My Church > My Category")] or include a locality/identifier if your organization's name is common [Category("My Church (AZ) > My Category")]
Block Location	RockWeb/Plugins/<reversed_domain>/<project>/	RockWeb/Plugins/org_mychurch/MyProject/
Block Namespace	RockWeb.Plugins.<reversed_domain>.<project>	RockWeb.Plugins.org_mychurch.MyProjectName
Class Namespace	<reversed.domain>.<project>	org.mychurch.MyProjectName
Assemblies	<reversed.domain>.<project>.dll	org.mychurch.MyProjectName.dll
API path	api/<reversed.domain>/	api/org.mychurch/
API v2 path	api/v2/plugins/<reversed.domain>/<project>/ api/v2/plugins/<reversed.domain>/<project>/controls/ api/v2/plugins/<reversed.domain>/<project>/models/	api/v2/org.mychurch/myplugin/ api/v2/org.mychurch/myplugin/controls/ api/v2/org.mychurch/myplugin/models/
Attribute "Key" names (except Block Attributes)	<reversed_domain>.<variable>	org_mychurch_AmazingWidgetFoo
Private Class Property	_ <u>property</u> >	_fooId
Querystring Parameters	use field/property case (Pascal Case)	GroupId
Webhook filename	_ <u>reversed_domain</u> _.ashx or <org initials>.<...> (again, to prevent collisions with other plugin developers)	_org_mychurch_Mailgun.ashx Or mc_Mailgun.ashx
Or Webhook folder	Webhooks/<reversed_domain>/Mailgun.ashx	Webhooks/org_mychurch/Mailgun.ashx

Note:

The domain name is always in lowercase.

For blocks, block location, attributes, and sql stuff, the reversed domain uses an underscore (_) instead of a dot. This is intentional to help prevent namespace resolution conflicts. Also, using a dot can be problematic when used in Lava.

Note to Core Team:

To avoid collisions with admin created attributes, the Rock core team will prefix attribute's attribute [Key] with "core_". Example:
`core_LastSendDate`

SQL: Custom Tables, Stored Procedures, User Functions

Custom tables should be **prefixed with an underscore** followed by your reversed, lowercase domain name, then your ProjectName -- replacing all dots with underscores.

Custom Namespaces, Classes, & Assemblies

Prefix your namespaces/classes & assemblies with your reversed, lowercase domain name, then your ProjectName.

```
namespace org.rocksolidchurch.MyProjectName
{
    public class Book
    {
        public int Id { get; set; }
        // ...
    }
}
```

Private class properties should be **prefixed with an underscore** as illustrated here.

```
#region Fields

private int? _fooId;
private Note _note;

#endregion
```

Custom API

When developing custom API extensions, you must use a folder convention `api/<com.domain>/` (such as `api/org.rocksolidchurch/`) to avoid collisions with other custom developer APIs.

Standard Control Variable Naming

If you really want to make the core team happy, you can follow these naming conventions on your entity property/fields:

Control Type	Prefix	Example
AccountPicker	<code>acctp</code>	
AutoCompleteDropDown	<code>ac</code>	acPersonSelect
AttributeEditor	<code>edt</code>	edtGroupMemberAttributes
Badge	<code>badge</code>	badgeNotice
BinaryFilePicker	<code>bfp</code>	
BinaryFileTypePicker	<code>bftp</code>	
BirthDayPicker	<code>bdayp</code>	
BootstrapButton	<code>bbtn</code>	bbtnSearch
Button	<code>btn</code>	btnSendLogin
ButtonDropDownList	<code>bddl</code>	
CampusesPicker	<code>mcamp</code>	
CampusPicker	<code>camp</code>	
CategoryPicker	<code>catp</code>	catpCategory
CheckBox	<code>cb</code>	cbUnlisted

Control Type	Prefix	Example
CheckBoxList	<code>cbl</code>	
CodeEditor	<code>ce</code>	
CompareValidator	<code>coval</code>	
ComponentPicker	<code>comp</code>	
ConfirmPageUnload	<code>conpu</code>	
CustomValidator	<code>cval</code>	
DataDropDownList	<code>ddl</code>	ddlDataView
DataPager	<code>dpgr</code>	
DataTextBox	<code>dtb</code>	dtbDescription
DatePicker	<code>dp</code>	dpAnniversaryDate
DateRangePicker	<code>drp</code>	
DateTimePicker	<code>dtp</code>	dtpFutureSendAt
DropDownList	<code>ddl</code>	
EntityTypePicker	<code>etp</code>	
FieldTypeList	<code>ftl</code>	
FileUpload	<code>fup</code>	
FileUploader	<code>fupr</code>	
GeoPicker	<code>geop</code>	
Grid	<code>g</code>	gMembers
GridFilter	<code>gf</code>	gfSettings
GroupPicker	<code>gp</code>	
GroupRolePicker	<code>grp</code>	
GroupTypePicker	<code>gtp</code>	
HelpBlock	<code>hb</code>	
HiddenField	<code>hf</code>	hfValue
HighlightLabel	<code>hbl</code>	hblUrgent
HtmlEditor	<code>html</code>	
HyperLink	<code>hl</code>	
Image	<code>img</code>	
ImageButton	<code>imb</code>	
ImageUploader	<code>imgup</code>	imgupPhoto
Label	<code>lbl</code>	lblApprovedByPerson
LinkButton	<code>btn</code>	btnLoginLogout
ListView	<code>lv</code>	
Literal	<code>l</code>	lPostText
LocationItemPicker	<code>locip</code>	
LocationPicker	<code>locp</code>	
MergeFieldPicker	<code>mfp</code>	
ModalAlert	<code>ma</code>	maDeleteWarning

Control Type	Prefix	Example
ModalDialog	md	mdPreview
ModalPopupExtender	mpe	
MonthDayPicker	mdp	
MonthYearPicker	myy	
NewFamilyMembers	nfm	nfmMembers
NoteContainer	note	noteComments
NotificationBox	nb	nbError
NumberBox	numb	
NumberRangeEditor	nre	
PagePicker	pagep	pagepRedirectTo
Panel	pnl	pnlValue
PanelWidget	pnlw	pnlwDisplay
PersonPicker	pp	ppGroupMemberPerson
PersonProfileBadgeList	badge1	
Placeholder	ph	phSuccess
PostBackTrigger	trgr	
RadioButtonList	rbl	
Repeater	rpt	rptCompletedPledges
RockBulletedList	blst	
RockCheckBox	cb	cbValue
RockCheckBoxList	cb1	
RockControlWrapper	wrap	
RockDropDownList	ddl	ddlGender
RockLiteral	l	
RockRadioButtonList	rbl	rblStatus
RockTextBox	tb	tbEmail
ScheduleBuilder	schedb	
SecurityButton	sbtn	
SlidingDateRangePicker	sdrp	sdrpFeeDateRange
StateDropDownList	statep	
Table	tbl	
TagList	tag1	
TermDescription	termd	termdTransactionCode
TextBox	tb	tbNewNote
TimePicker	timep	timepStartTime
Toggle	tgl	
UpdatePanel	upnl	upnlSettings
ValidationSummary	val	valSummaryTop
Xml	xml	xmlContent

Note:

Items shown in bold are the most frequently used controls.

Standard Field/Property Naming

We'd also recommend these naming conventions on your entity property/fields:

Data	Field Name Convention	SQL Datatype	C# Datatype	Note
Name	Name	nvarchar(100)	string	Grids, Unique, Not Null, Not Empty
Description	Description	nvarchar(max)	string	NoGrid, NotUnique, Optional
Date "Key"	..DateKey	int	int	This is <i>essentially</i> a FK to the AnalyticsSourceDate.DateKey (yyyymmdd) table/column for additional date analytics (useful for cases like 'select only Sunday dates').
Date/Time	..DateTime	DateTime	DateTime	
Date	..Date	DateTime	DateTime	
Time	..Time	DateTime	DateTime	
Url	..Url	nvarchar(2000)	string	http://stackoverflow.com/questions/417142/what-is-the-maximum-length-of-a-url
Birthdate - Month	BirthMonth	int	int	
Birthdate - Day	BirthDay	int	int	
Birthdate - Year	BirthYear	int	int	optional, null means not disclosed
email	..Email	nvarchar(254)	string	http://stackoverflow.com/questions/386294/maximum-length-of-a-valid-email-address
boolean	Is.. (unless obvious)	bit	bool	0 = false, 1 = true (try to avoid double negatives, for example "NotEnabled")
First,Middle,Last name	..Name	nvarchar(50)	string	
FullName	..FullName	nvarchar(152)		
Guid	Guid	uniqueidentifier	Guid	Unique, Not Displayed, Not Null, Not Empty, Required Column for Rock Tables
Duration (OBSOLETE)	..Duration	int	int	a field that implies a number of seconds or minutes of something
Duration (as of v11)	..Duration {UnitOfTime}	int, decimal (18,2) or float	int, decimal or double	where {UnitOfTime} is Seconds, Milliseconds, Days, Hours, Weeks, etc.
Path/FileName	..Path/Filename	nvarchar(260)	string	http://msdn.microsoft.com/en-us/library/aa365247.aspx
Order	..Order	int	int	
Currency	..Amount	decimal (18,2)	decimal	US Dollar Only
Percent				TBD 100%, 1 or 100. 100 reads better
Password	Password	nvarchar(128)	string	
Html	..Html	nvarchar(max)	string	
PhoneNumber	PhoneNumber	nvarchar(20)	string	store unformatted, no spaces, no dashes, no parentheses
PhoneNumberExtension	[PhoneNumber]Extension	nvarchar(20)	string	store unformatted, no spaces, no dashes, no parentheses
PrimaryKey	Id	int	int	
ForeignKey	[optional] + <ParentTableName> +Id	int	int	ex. see Note's NoteTypeId

Data	Field Name Convention	SQL Datatype	C# Datatype	Note
DefinedValue Foreign Key	<DefinedType.Name> +ValueId			a ref to a DefinedValue of a DefinedType
BinaryFile Foreign Key	[optional] +BinaryFileId or [optional] +FileId			ex. Page's IconBinaryFileId

Utility Class Naming Conventions

Lastly, here are a few conventions for any UI controls you might create:

Class Name Convention	Description	Example
..Picker	users selecting and picking an item	CampusPicker
..Uploader	editor control for selecting or uploading	ImageUploader
..List	used for simple list/grid-like controls	ButtonDropDownList