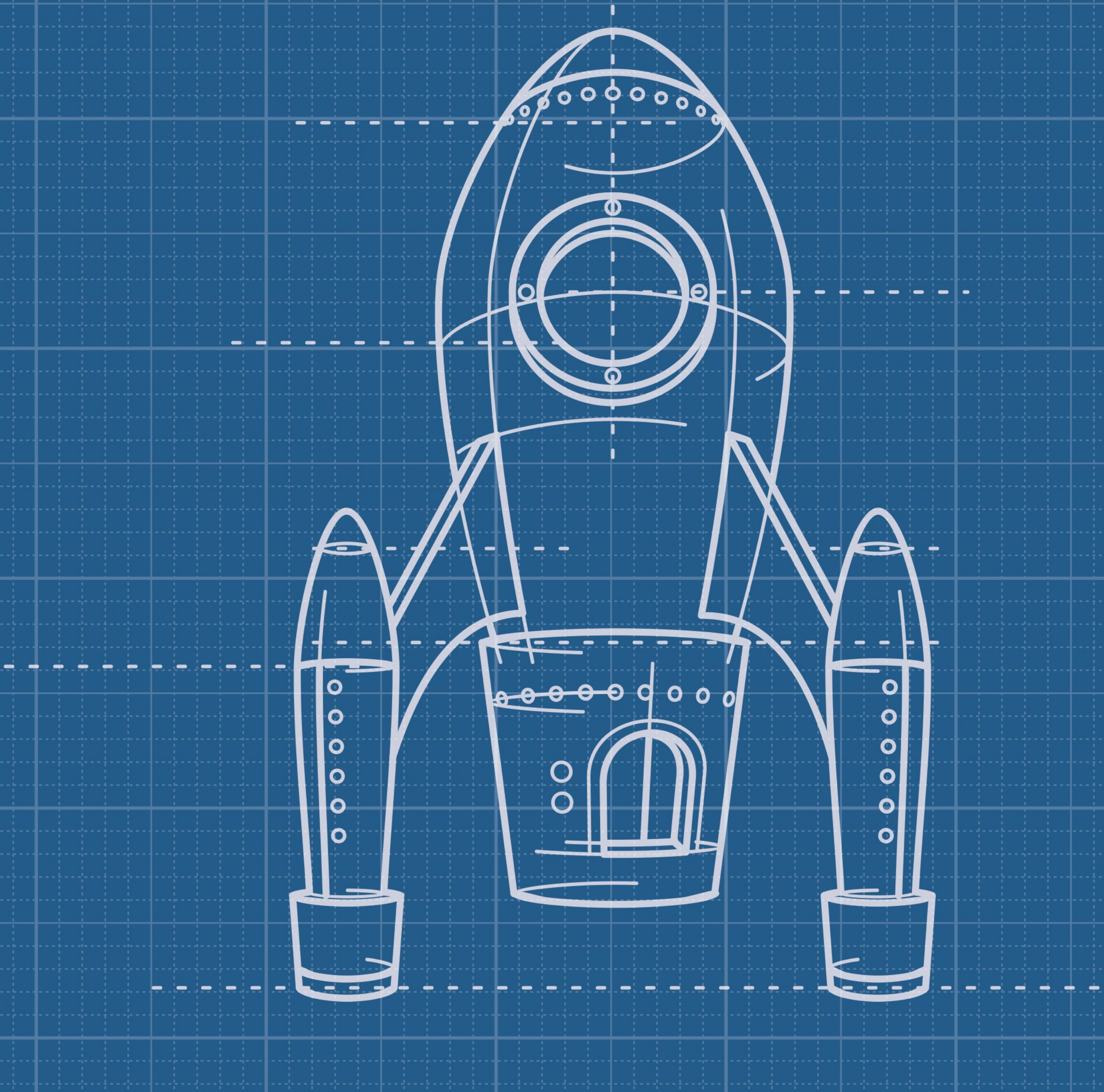
ROCKIT ACADEMY

202 IGNITION

GOING FURTHER WITH ROCK DEVELOPMENT



GOING FURTHER WITH ROCKIT

202

More Than Blocks

There is more to Rock than blocks. Under the covers Rock is a framework full of rich entities that let you manage people, families, groups, campuses, locations, dictionary-like-lists, and much more. And if those things don't meet your needs, you can even create your own custom entities within the framework too.

In this book we'll tell you a little about the major Rock entities, and then wrap up with details you'll need to develop your own custom entities. But first let's talk more about the concept of an entity.

What Are Entities?

The word "entity" is just a generic term that refers to a code class that models and manages a particular type of data. These classes are built on something called the Entity Framework (EF). EF is Microsoft's recommended data access technology -- but don't worry! You do not need to know all the ins-and-outs of EF unless you really want to. We only mention it because some of the conventions you'll read below come from that framework. Ok, back to our story...

> **Going Deep**
>
> If you are new to Entity Framework and want to learn more about what's going on behind the scenes in Rock, you can read the Get Started with Entity Framework and/or the Working With DbContext MSDN articles.

Let's get real by talking specifics. And, since everyone knows what a person is, we'll use it as we describe how entities work.

The Person *entity* is actually the `Person.cs` class. That class models the properties of a person including FirstName, LastName, BirthDate, Gender, and Email just to name a few. So, when code somewhere creates a new Person object and saves it, a new record is added to the database which contains all the person's property values... mostly.

Some classes have *virtual* properties (such as the Person class *Age* property). These virtual properties decorated with the [NotMapped] attribute don't actually store their values in the database. Instead they are basically computed via code in a getter method as seen here:

```
[NotMapped]
[DataMember]
```

```
public virtual int? Age
{
    get
    {
        if ( BirthYear.HasValue )
        {
            DateTime? bd = BirthDate;
            if ( bd.HasValue )
            {
                DateTime today = RockDateTime.Today;
                int age = today.Year - bd.Value.Year;
                if ( bd.Value > today.AddYears( -age ) ) {age--};
                return age;
            }
        }
        return null;
    }
    private set { }
}
```

> Note:
>
> Virtual properties are almost always decorated with the `[NotMapped]` data annotation (or C# attribute) which is a signal to Entity Framework to *not store* the value in the database. The `[DataMember]` annotation is used to tell the framework to serialize these properties because we've decorated our classes with the `[DataContract]` attribute. If this sort of thing interests you, read about it on the MSDN site.

Some virtual properties hold other entities or whole collections of entities. For example, the person PrimaryAlias property holds a PersonAlias entity, but the PhoneNumbers property holds a *collection* of PhoneNumber entities.

```
[DataMember]
public virtual ICollection<PhoneNumber> PhoneNumbers
{
    get { return _phoneNumbers; }
    set { _phoneNumbers = value; }
}
private ICollection<PhoneNumber> _phoneNumbers;
```

For the virtual properties that are entities *but not collections*, you will see two parts. One property holds the entity as a "navigation property" and the other property holds the associated Id. Let's look at the person's Photo property. Here you see a Photo and a PhotoId property:

```
[DataMember]
public virtual BinaryFile Photo { get; set; }

[DataMember]
public int? PhotoId { get; set; }
```

This is very cool because EF will Lazy Load the Photo property-entity if you need to refer to it. In those cases, you have full access to that entity as shown here:

```
// assuming you have a person object/instance already...
var personPhotoFile = person.Photo.FileName;
```

I don't know about you, but that just makes me a little giddy inside.

> ### Going Deep
> While it is cool, lazy loading comes at a performance cost: meaning behind the scenes the framework has to issue another database query to get the value. So, if you know you're going to need it, tell the framework to eagerly pre-load your navigable object properties as discussed in the 101 Launchpad book on Loading Entities.

To make it work like that we had to tell EF about this relationship via a PersonConfiguration (*Entity* Configuration) class in our `Person.cs` class:

```
///

/// Person Configuration class.
///

public partial class PersonConfiguration : EntityTypeConfiguration<Person>
{
    public PersonConfiguration()
    {
        /// ...
        this.HasOptional( p => p.Photo ).WithMany()
            .HasForeignKey( p => p.PhotoId ).WillCascadeOnDelete( false );
        /// ...
    }
```

> ### Don't Worry
> Again, you don't really need to understand this last part unless you're creating your own custom entities. We'll cover that in the last several chapters of this book.

Common Entities

Now that we've got some of the inner workings out of the way, let's learn a little about each of the primary entities in Rock. This section might seem a bit tricky because entities often refer to other entities that we haven't explained yet -- please bear with us. If you read something that's not quite making sense, just keep reading and it should become clear.

> **Help Us Help Everyone**
>
> If you think we've missed the mark trying to explain something, please let us know. Use the little megaphone "Improve" feature (available online on the right side of the page) and give us the details on how you might write it. We love it when you help us help everyone.

Person

What can we tell you about this that you don't already know? This is a pretty straightforward entity except perhaps its relationship to the *PersonAlias* that we touched on in the last book. The *PersonAlias* is almost always the glue that connects other entities to the Person entity. You'll do well to not forget this tidbit.

Here the more commonly used properties you should consider when working with people in your code:

- **FirstName, NickName, LastName, FullName, FullNameFormal, FullNameReversed** - You get the drift. These are their various names in various formats. Many of these are virtual properties that are just computed from the first and last name properties.
- **EmailPreference** - Tells you whether when it's ok or not ok to use their email (ok, never, or no mass/bulk emails, etc.)
- **BirthDay, BirthMonth, BirthYear** - Of the date of birth. Sometimes you won't have the BirthYear (because it can be a sensitive topic for some people -- ahem.)
- **GraduationDate** - This is the date they will have completed their secondary schooling (not their higher education or vocational schooling). In the United States this is their High School graduation date.
- **PhotoUrl** - This is a URL to their photo or if no photo exists, it will be a URL to the generic male/female/child SVG image.
- **RecordStatusValue** - This is the status of their record. You know...whether they're active, inactive, or pending. This value is actually a DefinedValue which we'll explain

in greater detail in the DefinedType DefinedValue section.
- **IsDeceased** - A Boolean flag that would indicate they are deceased.
- **ConnectionStatusValue** - This tells you the nature of their connection with your organization. This is also a DefinedValue which comes from the list of possible options of the Connection Status DefinedType.
- **PrimaryAlias** - This is the person's primary PersonAlias.
- **Aliases** - A collection of all of the person's PersonAlias records. This collection slowly builds over time as a person's duplicate records are merged together.

Some properties are not required, so keep in mind you may not have this data for some people:

- **Email** - The person's email address.
- **PhoneNumbers** - A collection of their PhoneNumber entities.
- **Users** – A collection of UserLogin entities for the person. Similar to the Aliases, a person can have one or more UserLogins as their duplicates records are merged.

Here are a few very handy methods you can use on a person or via the Person class:

- **GetAnchorTag(...)** – This builds a HTML anchor/link tag to the person's profile. You just need to pass it the URL to Rock's root. You'll see this commonly called like this: `GetAnchorTag( ResolveRockUrl( "/" ) )`
- **GetEmailTag(...)** – This builds an HTML email anchor tag for sending an email to the person through Rock's communication system. The HTML will also note whether or not the person has an email address, allows email, allows bulk emails, etc.
- **GetPersonPhotoImageTag(...)** – This will build the markup necessary for a photo of the given person or person alias.

  ```
  // Get the HTML for the person's photo as a 50x50 image.
  lPersonIconHtml.Text = Person.GetPersonPhotoImageTag( aPersonAlias, 50, 50 );
  ```

- **GetHomeLocation()** – This method makes it trivial to fetch a person's home address:

  ```
  var homelocation = person.GetHomeLocation();
  ```

> ### That's It?
> We don't want to bore you to death by listing out every single property a Person or the other entities have. You can find the complete list in the Model Map block under Power Tools which shows the Rock *model entities*. Additionally, there is now a REST "API Docs" reference (found under your Rock system's Power Tools) which shows you all entities that have REST endpoints you can use with your custom code.

Attributes

Before we move on to the next entity, we should mention one very important thing: *Attributes*. Attributes can be tied to any entity, which means you can track specific values for any instance of that entity. For the Person entity, you'll notice under Admin

Tools > General Settings we set up the *Person Attributes* page. This lets you add/manage custom attributes for the Person entity.

Loading Attribute Values

Before you try to *access* an entity's attribute values **you must first load the attributes** as seen in this example:

```
person.LoadAttributes();
string personAbilityLevel = person.GetAttributeValue( "AbilityLevel" );
```

Saving Attribute Values

Before you *save* an entity's attribute values **you have to load the attributes** as seen in this example:

```
group.LoadAttributes( rockContext );
// After you've loaded the attributes... you can set and save values
group.SetAttributeValue( "Topic", smallGroupTopicDefinedValue.Guid.ToString() );
group.SaveAttributeValues( rockContext );
```

Group

Groups represent collections of people. Serving teams are groups, as are home and neighborhood "Small Group" groups. In fact, as you will read below, you will see that security roles and families are groups too.

Groups will always have a *Name* and a *GroupType*. There are all kinds of GroupTypes in Rock. Your code will use this to distinguish and handle things differently for different types of groups. (Read more about GroupTypes below.)

Unless it is a top level group, it will probably have a ParentGroup, and unless it's a leaf group, it will have a Groups property which holds the collection of its child groups. A group will also generally have a Members property which holds a collection of GroupMember entities, and will sometimes have a GroupLocations property that holds a collection of locations (GroupLocation) associated with the group.

The rest of the properties are pretty self-explanatory too.

Code Recipes

Here are a few code recipes for some typical situations you may be in...

When adding a new person, if you also wish to create their new family (group) too, you can use the *PersonService*'s handy SaveNewPerson() method. It will return the newly created group in case you want to do something with it.

```
// once you have your person object setup...
var familyGroup = PersonService.SaveNewPerson( person, rockContext, campusId, false );
```

When adding a person to a group you will usually want to set their group member status and the role that is the default for that particular group type.

```
// Adds a person (using an Id) to a group with the default role.
GroupMember groupMember = new GroupMember
{
    PersonId = personId,
    GroupMemberStatus = GroupMemberStatus.Active,
    GroupRoleId = aGroup.GroupType.DefaultGroupRoleId.Value
};
aGroup.Members.Add( groupMember );
```

Listing all addresses of a group:

```
var html = new StringBuilder();
var groupId = 68;
var group = new GroupService( new RockContext() ).Get( groupId );
foreach ( var groupLocation in group.GroupLocations )
{
    var location = groupLocation.Location;
    html.AppendFormat( "
• {0}{1}{2}
{3}, {4} {5}
",
        location.Street1,
        (string.IsNullOrEmpty(location.Street2) ? string.Empty : "
"),
        location.Street2,
        location.City,
        location.State,
        location.PostalCode );
}

lList.Text = string.Format( "
      {0}

", html.ToString() );
```

GroupType

Every group has a specific group type. The group type establishes attributes and has properties that helps control part of the group's behavior. For example, any groups that are of type "Check in by Grade" will have a "Grade Range" attribute which is inherited from that group type. For an example of this in action (controlling behavior), refer to the check-in related blocks which use a group's GroupTypePurposeValue to decide whether or not to show/use a particular group for check-in purposes.

Group Types will *always* have:

- **Name** – The name by which the group is known.
  GroupTerm – This defines the term used to describe groups of this type. Examples are: group, community, class, family, etc.
- **GroupMemberTerm** – This is the term used to describe the member. Examples are family member, team member, member, employee, etc.

Group Types may have:

- **AllowMultipleLocations** - Boolean that controls whether or not these groups are allowed to have multiple locations.
  DefaultGroupRole - The default role that a new member should have when placed into this group type.
- **InheritedGroupType** - If set, indicates which GroupType it inherits settings and properties from.
- **ShowInGroupList** - Indicates whether these groups are shown in GroupList blocks.

- **ShowInNavigation** - Indicates whether these groups are shown in navigation (such as TreeViews and menus).
- **TakesAttendance** - Flag that indicates whether or not this type supports taking attendance.
- **SendAttendanceReminder** - Flag that indicates if an attendance reminder should be sent to group leaders.

GroupMember

A group member is the entity that connects a person and a group. It holds any attribute values the member has and a property that points to the actual group and person instances.

Group Members will *always* have:

- **Group** - The group to which the person belongs.
  Person - The person who is in the group.
- **GroupRole** - The role the person has within the group (leader, member, etc.). These come from the GroupType and are configurable.
- **GroupMemberStatus** - Indicates whether the person is Inactive (0), Active (1), or Pending (2). This is an enum in Rock.

Code Recipes

Adding a person to a group:

```
var groupMemberService = new GroupMemberService( rockContext );
var groupMember = new GroupMember();
groupMember.PersonId = person.Id;
groupMember.GroupId = group.Id;
groupMember.GroupRoleId = groupRoleId.Value;
groupMember.GroupMemberStatus = GroupMemberStatus.Active;
if ( groupMember.IsValid )
{
    groupMemberService.Add( groupMember );
    rockContext.SaveChanges();
}
```

It's rather important that you remember to set the group member's role and status as seen above.

GroupLocation

This holds all the location details for a group. Examples could include a Person/Family's address, a Business' address, a room where a Bible study meets, etc. Pretty much any place where a group of people meet or where they are located.

A GroupLocation should *always* have:

- **Group** – The group to which the address belongs.
  Location – The postal/mailing/geographical details for the address. This is another object class described in the next section.
- **GroupLocationTypeValue** – This will be one of the values from the Location Type *DefinedType* for Groups (representing either Home, Meeting Location, Work, or Previous Address).

In certain cases it may also have:

- **Schedule –** The date/time details about when the group is at that location.

Location

A location could be a street address, building, floor, room, kiosk location, etc. A location is also stackable/hierarchical. For example, for a church's campus can have multiple buildings or facilities, each building can be multi story and a story can have multiple rooms.

If the location is a place on a campus, it should have:

- **Name** - The common name the location is known by.
- **CampusId** - The Id of the location's campus.
- **ParentLocation** - This will be another location that contains this location. If the location is a campus, this value will be null.
- **LocationTypeValueId** - This will be one of the Id values from the *Location Type* DefinedType (representing either Campus, Building, or Room).

If it is a mailing address it will usually have:

- **Street1** - The first line of the mailing address.
- **Street2** - The second (optional) line of a mailing address.
- **City** - The city.
- **State** - The state.
- **PostalCode** - The Zip/postal code of the mailing address.
- **Country** - The optional value indicating the country of the address.

Once geo-coded, a location will have:

- **GeoPoint** - The DbGeography object that holds the latitude and longitude of the location. This will be null if the location is a GeoFence.
- **GeoFence** - A DbGeography object that holds the set of *coordinates* that make up the fence (or perimeter) contains the location.
- **Latitude** - If it's a GeoPoint, this will give you the latitude of the point.
- **Longitude** - If it's a GeoPoint, this will give you the longitude of the point.
- **GeocodeAttemptedResult** - The resulting code given by the service when the geocode took place. This depends on the service. For example, SmartyStreets will use their "precision" result described here: https://smartystreets.com/docs/address
- **GeocodeAttemptedDateTime** - The date the geocoding took place.
- **GeocodeAttemptedServiceType** - The name of the service that performed the geocoding.

Once "standardized" by one of the Address Standardization services defined under System Settings > Location Services (if enabled), a location will have:

- **StandardizeAttemptedDateTime** - Represents when the most recent address standardization attempt was made. If this is not applicable to this location, or if the address has not been standardized, this value will be null.
- **StandardizedDateTime** - The date and time that the Location's address was successfully standardized.

- **StandardizeAttemptedServiceType** - The component name of the service that last attempted to standardize this Location's address.
- **StandardizeAttemptedResult** - The result code that was returned by the address standardization service.

Campus

This entity represents the physical or logical site where the organization holds its activities. For multi-site organizations there should be a campus record for each site. Many of the primary entities in Rock can be tied to either a single or multiple campuses.

A campus will always have a **Name** and will usually have:

- **Location** - The actual geographical details about the campus.
- **PhoneNumber** - The main, public office phone number for the campus.
- **ServiceTimes** - The delimited data you put in here is primarily intended to be displayed on your campus service time's web page. It's not actually structured data – for that, use the Schedule entity.

Single Campus Special Considerations

Starting with Rock v10, when a Rock system has only 1 campus defined, we want to ensure that no one using that system has to worry about seeing or picking that single campus. We want the system to seemlessly just use that single campus and never really expose it (except in very rare, specific places).

Schedule

This holds the date and time details for some sort of event. It is used with the check-in system, the group locator system, and the event calendar.

A schedule will usually have:

- **Name** - A friendly name for the schedule.
- **iCalendarContent** - This is the iCal representation of the date/time. Since iCal is quite sophisticated, you may wish to get the full DDay.iCal.Event object using the `GetCalenderEvent()` method. From that object you can programatically access the properties of the schedule.
- **EffectiveStartDate** - The date that the schedule becomes active.
- **EffectiveEndDate** - The date that the schedule becomes inactive.
- **WeeklyDayOfWeek** - This is the day of the week that the weekly item takes place.
- **WeeklyTimeOfDay** - This is the time of day that the weekly item take place.

> DDay.iCal Info
>
> Although this project has been moved to a new code base, you may still find the examples and information about iCal from this old repository useful, namely the Event class.

Role

This is a special entity that represents a security role and has a collection of users that have the role. Behind the scenes its data is stored in the Group and GroupMember tables and it is primarily used by Rock’s security system including the Authorization class.

A role will have:

- **Name** - One ore two words that quickly defined what the role represents.

The Role class also has a method that is used to quickly determine if the given person has the role:

- **IsPersonInRole(...)** - Returns true if the person is in the role; false otherwise..

DefinedType & DefinedValue

A DefinedType is a dictionary of mostly-unchanging values for a particular thing in Rock. The thing's individual item values are referred to as DefinedValues. Therefore, if Marital Status was the DefinedType, then its DefinedValues are Single, Married, Unknown, Widowed, etc.

Several classic examples of DefinedTypes are *Phone Type*, *Shirt Sizes*, *Small Group Topic*, *Countries*, etc.

Defined Values can be categorized, ordered and can be furthered specified by a FieldType. And, of course they can have *Attributes* which means you can expand on them quite a bit. Attributes and AttributeValues are covered in the next chapter.

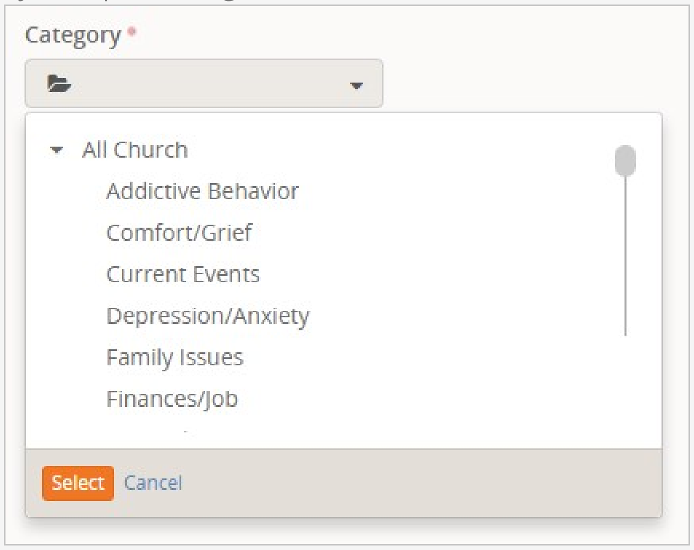
The benefit of managing these values in this way is that it avoids having to create new Entity types for each defined type/value that you want to create. Similar to attributes, these can be created as the need arises without having to change the Rock core or add a plug-in just to provide additional lookup data.

If you are looking for a good case study showing how far you can go with the DefineType and DefinedValue, check out the DISC Results DataType. It has nine attributes of varying datatypes and they are used programmatically by the *DiscResults* block.

Category

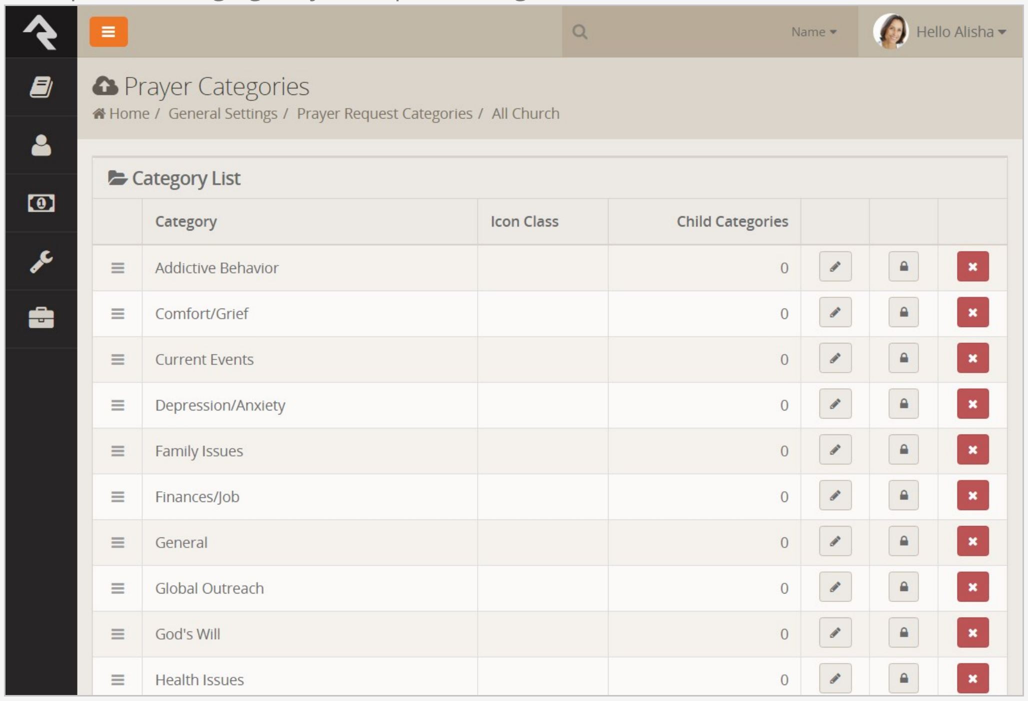
A category is another fairly generic entity type in Rock which can be used to... well... categorize things. Each category will belong to one Entity type, and categories can be created in a parent-child hierarchy as well as ordered.

Example of Prayer Request Categories



There is even a *Categories* block in Rock that can be configured and bound to any Entity in order to manage categories for that entity.

Example of Managing Prayer Request Categories



Prayer Categories

Home / General Settings / Prayer Request Categories / All Church

Category List

Category	Icon Class	Child Categories
Addictive Behavior		0
Comfort/Grief		0
Current Events		0
Depression/Anxiety		0
Family Issues		0
Finances/Job		0
General		0
Global Outreach		0
God's Will		0
Health Issues		0

Advanced Entity Guide

Now let's look at several entities that are a little less common or a little more obscure than the day to day entities.

Attendance

This represents an instance where a person attended or was scheduled to attend a group or event. An attendance will essentially always have the following properties:

- **PersonAliasId** - The alias id of the person who did/was to attend.
- **StartDateTime** - The date and time the person checked in.
- **RSVP** - An enum indicating whether or not the person had RSVP'd. (0 = No, 1 = Yes, 2 = Maybe)
- **DidAttend** - A Boolean indicating whether or not the person attended.
- **ScheduleId** - The Schedule Id for which the person did/was to attend.
- **GroupId** - The Group Id for which the person was related to when the check-in occurred.
- **SundayDate** - The date of the Sunday for the week the attendance occurred.

This represents an instance where a person attended or was scheduled to attend a group or event. An attendance will essentially always have the following properties:

- **AttendanceCodeId** - The Id to the AttendanceCode that has the security token that was created for the attendance record.
- **LocationId** - The Location Id where the person did/was to attend.
- **DeviceId** - The device the person used when they checked in.
- **CampusId** - The campus where the attendance occurred.
- **DidNotOccur** - a Boolean which will be set to true if the event for which the person was to attend did not occur.

Attribute & AttributeValue

As mentioned elsewhere, nearly every entity type can have Attributes and the entity instances can store AttributeValues. Every Attribute has a particular field type such as Text, Date, Date Time, Date Range, Day of Week, Person, Group, etc. and this controls what kind of data can be stored in the AttributeValue. There are about one hundred field types so you should be able to store just about any kind of data you can imagine.

Attributes have the following properties:

- **Name** - This is the friendly name of the attribute.
- **Key** - This is the key that you may use in certain Lava filters to manage your

attribute/values.
- **EntityTypeId** - Since an attribute is tied to an Entity Type, this holds the value of that type.
- **EntityTypeQualifierColumn** - This is only used in those cases where the attribute applies to only a subset of the EntityType.
- **EntityTypeQualifierValue** - This is only used in those cases where the attribute applies to only a subset of the EntityType.
- **FieldTypeId** - This indicates exactly which field type the attribute is using.
- **IsMultiValue** - This boolean controls whether or not the attribute can have multiple assigned values.
- **IsRequired** - This boolean controls whether or not an attribute value must be supplied when editing.
- **IsGridColumn** - This boolean controls whether the attribute/value should appear in grids where the entities are listed.
- **Description** - This is the friendly administrative text that helps you remember what the attribute is use for.
- **Order** - The integer stored here indicates the ordinal when they are displayed in a list.

AttributeValues have the following properties:

- **AttributeId** - This indicates exactly which attribute this value belongs to.
- **EntityId** - This indicates exactly which entity the value belongs to.
- **Value** - This is the string representation of the attribute value for the entity.
- **ValueAsNumeric** - This is the numeric (decimal) representation of the value.
- **ValueAsDateTime** - This is the DateTime representation of the value.
- **ValueAsBoolean** - This is the boolean representation of the value.
- **ValueAsPersonId** - This is the person Id of the Value (if the value is a person alias guid).

Out of the box Rock includes UI for adding/deleting attributes for Person, Group, GroupType, and a few others entities, but there is an "Attributes" block that's very easy to configure to manage new custom attributes for just about any *entity*. However, not every entity "Detail" block currently has code to manage the Attribute *Values* for that entity but, if you look at the code history (33a7217 and 70f3e16) of the Finance AccountDetail block, you will see the code needed to add this feature. *Be sure to check the latest code/pattern in the AccountDetail block since some patterns have changed slightly since those two commits were made (for example: 760f7d7).*

There is another code pattern for enabling "List" blocks to show attribute values on the grid when the attribute has the "Show in Grid" (IsGridColumn) checkbox checked. (See 2c59a7f)

Code Recipes

There isn't a quick way to load attributes in one round trip for a list of entities, but using the “.Where” clause on attribute values for a queryable you can do something like this:

```
var rockContext = new RockContext();

// get groups that have a favorite color attribute of blue
```

```
IQueryable<Group> groupsWithAttributeValues = new GroupService( rockContext )
    .Queryable().WhereAttributeValue( rockContext, "FavoriteColor", "Blue" );
```

Group attributes are the most complicated to load since they can inherit attributes from their parent GroupType(s) and the above snippet wouldn't work if a group inherited an attribute value from a GroupType, but in most cases, this would do the trick.

You can even do some complex queries using the `WhereAttributeValue` method and the ValueAsBoolean, ValueAsDateTime, ValueAsNumeric forms of the Value. These properties are automatically computed using the Value property but be sure your value stores a Date before trying to use the ValueAsDateTime property, for example.

```
var rockContext = new RockContext();

// Get cool people.
 var catBaptismCutoff = RockDateTime.Now.AddYears( -3 );
 IQueryable<Person> personsThatAreCool = new PersonService( rockContext ).Queryable()
     .WhereAttributeValue( rockContext, av =>
         (av.Attribute.Key == "LovesStarWars" && av.ValueAsBoolean == true)
      || (av.Attribute.Key == "CatBaptismDate" && av.ValueAsDateTime > catBaptismCutoff ));
```

BinaryFile

A binary file is basically any kind of file that needs to be stored in Rock (or elsewhere). The file might be an image, a check-in print label, or an uploaded PDF. Using different "Storage Providers" the contents of the file can be stored to the web server's file system, the database or places such as Azure, Amazon S3, etc. The BinaryFile entity will always have the following properties:

- **IsTemporary** - This Boolean lets Rock know if the storage of the file was only temporarily necessary. In other words, Rock is free to delete any that are temporary.
- **FileName** - This is the name of the file, including any extensions. This name is usually captured when the file is uploaded to Rock and this same name will be used when the file is downloaded.
- **MimeType** - No clowning around, these are official. The value recorded here tells you if the file is a PDF, an image, text, video, word document, etc. You can see a rather extensive official list of valid Mime Types here.

Although not technically required, a BinaryFile will also typically have the following:

- **BinaryFileTypeId** - This lets you determine the BinaryTypeType which controls the security requirements on the file, the storage provider to use for storing the file, and whether or not the file can be cached.
- **StorageEntityTypeId** - This tells you which Storage Provider is being used to store the file. This is the Id of record from the EntityType table.
- **Path** - This is the path to the file resource as determined by the storage provider.
- **URL** - Somewhat related to the Path, the value stored here is the public URL used to retrieve the file from the storage provider.

Communication

A communication represents an email, an SMS message or something similar. Oddly enough there are not many required properties on a communication but they will usually have these:

- **Subject** - Yes, it's what you think it is.
- **Status** - This is one of the following: 0 – Transient, 1 – Draft, 2 – Pending Approval, 3 – Approved, 4 – Denied. Transient communications are those created somehow by the system but not yet edited by the user. They will be removed by the Rock cleanup job periodically.
- **MediumEntityTypeId** - This is what tells you if the communication is an email, SMS text, etc. This is the Id of an item in the EntityType table.
- **SenderPersonAliasId** - This is the alias id of the person who is considered the "sender" of the communication.
- **IsBulkCommunication** - Recommended to be set to true, if the communication is being sent to a large amount of people who are probably not expecting it.
- **Recipients** - This is a collection of CommunicationRecipient entities which gives you the particular details about who is going to (or has) received the message.

CommunicationRecipient

This goes along with a Communication and holds the details about the person who received or should have received the communication.

- **Status** - This is one of the following and tells you whether or not the message was sent, received and/or opened: 0 – Pending, 1 – Delivered, 2 – Failed, 3 – Cancelled, 4 – Opened, 5 – Sending.
- **StatusNote** - This will hold any free-form text regarding the message delivery or reason why it could not be delivered.

ContentChannel & ContentChannelItem

Content channels and the items they contain have become the main vehicle for putting structured content (HTML) on your Rock powered website. When you look at certain pages on the main rockrms.com website such as https://www.rockrms.com/Rock/Connect or the promotion ads on the stock home page of the demo site you will see them in action.

In actuality, Content Channel's have an underlying Content Channel Type where base, inherited channel attributes and channel item attributes can be put. In this way, Content Channel Types and Content Channels are similar to Group Types and Groups.

You may not be interacting with Content Channels in your code too often, but if you do you'll want to study the ContentChannelView block to see how.

Note & NoteType

Notes represent an entity that holds user typed information about another entity and Rock provides some blocks to help with their usage. To prevent you from re-creating the wheel, let's make sure you understand the power of the Notes entity. Notes can be seen in action on the person profile page as the "Timeline" feature via the Notes block and on the Group Member Details page. Although Notes are relatively simple, having only a few properties listed here, they can be bound to any entity type in Rock and each Note Type can be secured allowing only certain roles/people to view, add or edit them. The properties of a Note are:

- **NoteTypeId** - Each note has an underlying type that tells Rock which entity type it is tied to and what security it has.
- **EntityId** - This is the Id of the entity that the note is bound to. The Note's type tells you which entity type the Id belongs to.
- **IsAlert** - This Boolean lets Rock know if this special note should be highlighted in some way to bring special attention to the viewer.
- **IsPrivateNote** - This Boolean flag indicates that the note should only be visible to the person who created the note.
- **Caption** - This is a brief, optional title for the note which is typically programmatically generated on the fly by Rock's internal code. For example, on a private note, the caption is set to "You - Personal Note".
- **Text** - This is the main content of the note.

Putting Notes on an Entity

Regardless of whether you are interacting with notes in your own code, if you find the need to have notes on a new entity consider using the Note block with the following setup:

1. Add a new Note Type
2. Add the Notes block to your entity details page and configure its Context with the entity type of your choosing.
3. Edit the page Advanced Settings and set the Context Parameters so that Rock puts the correct entity into the page context for the Note block to operate against.

Workflow & WorkflowType

A Workflow is an instance of a WorkflowType. Abstractly speaking, a WorkflowType defines a set of Activities and Actions to perform each time a new workflow instance is created. Typically you won't be trying to work too directly with Workflows and WorkflowTypes in your code, but you may want to start a workflow from your code.

Here's an example of programatically launching a new Workflow using a known workflowType (as found in the PersonUpdate.Kiosk block):

```
if ( workflowType != null && ( workflowType.IsActive ?? true ) )
{
    var workflowService = new WorkflowService( rockContext );
    var workflow = Rock.Model.Workflow.Activate( workflowType, "Kiosk Update Info" );

    // set attributes
    workflow.SetAttributeValue( "PersonId", hfPersonId.Value );
    workflow.SetAttributeValue( "FirstName", tbFirstName.Text );
    workflow.SetAttributeValue( "LastName", tbLastName.Text );
    workflow.SetAttributeValue( "StreetAddress", acAddress.Street1 );
    workflow.SetAttributeValue( "City", acAddress.City );

    // ...

    // lauch workflow
    List<string> workflowErrors;
    workflowService.Process( workflow, out workflowErrors );
}
```

Saving Custom Data

There's so much more we can do with just blocks, but we suspect you want to create your own custom entities that can be saved to the database. Since Rock uses Microsoft's Entity Framework we'll show you how simple this is with a "code first" approach.

> **Prerequisites**
>
> Although what we're about to do is pretty simple, we can't turn you into an experienced C# web-developer. You should have a basic understanding of ASP.NET Web Forms otherwise you'll probably be a bit lost in this chapter.

Referral Agency Sample Project

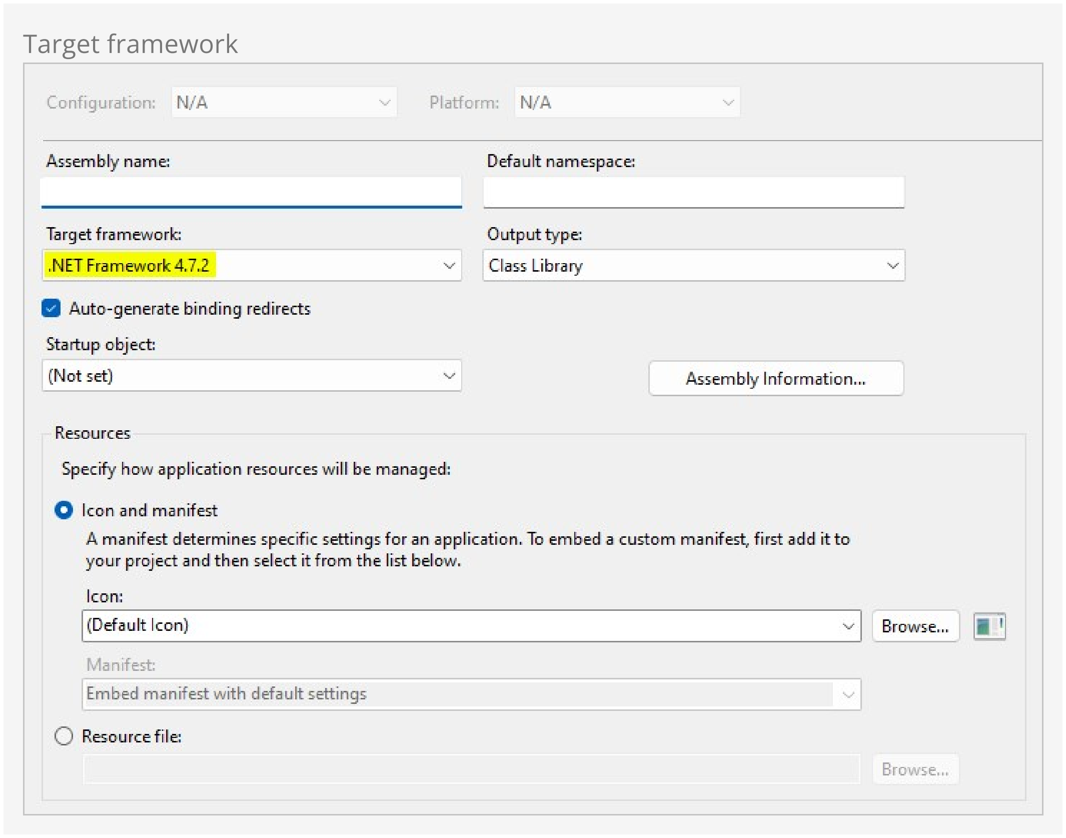
The Rockit SDK comes with a reference project called "Sample Project". It includes a custom entity called *ReferralAgency* and also included in your plugin folder are two custom blocks, one to list existing referral agencies and one to add/edit/view the details of a referral agency. In this section we'll walk you through the code for the custom *ReferralAgency* entity that belongs to the generic *org.rocksolidchurch.SampleProject* so you have a solid foundation for creating your own custom entities.

Let's say we want to keep track of agencies that your church/organization refers people to. We are going to create a `ReferralAgency` class that models an agency with its properties and a very simple `ReferralAgencyService` to act as our liason with the database. These classes will rely on the base Rock classes but We won't go too deeply into those inner workings so we can keep this chapter simple. Then we're going to create two blocks to help us manage our *ReferralAgency* items.

> **Rock Groups Rock**
>
> Depending on your exact needs you might be able to use Rock's super-flexible *Groups* system along with Rock's general purpose *Attributes* system. If so, it will mean you won't have to write any code. However for the purpose of this chapter we'll pretend you really needed to create your own custom model.

Step 1 - Add a Project

Let's create a class library project to hold our model classes. Make sure nothing is running in Visual Studio and then right-click the solution in the Solution Explorer. Select 'Add > New Project...' Create it as a new Visual C# Class Library (Target framework .NET Framework 4.7.2) with a name of `org.rocksolidchurch.SampleProject`.


Target framework

Configuration: N/A
Platform: N/A

Assembly name:
Default namespace:

Target framework: .NET Framework 4.7.2
Output type: Class Library

☑ Auto-generate binding redirects

Startup object: (Not set)

Resources

Specify how application resources will be managed:

● Icon and manifest

A manifest determines specific settings for an application. To embed a custom manifest, first add it to your project and then select it from the list below.

Icon: (Default Icon)

Manifest: Embed manifest with default settings

○ Resource file:

Let's create some folders to keep our stuff organized. Right-click the project in the Solution Explorer and select 'Add > New Folder' and create:

- Migrations
- Model
- Rest
- SystemGuids

We're going to be using several other libraries in our classes so let's add references to a few key assemblies.

- From the Framework Assemblies, select:
  - System.ComponentModel.DataAnnotations
  - System.Runtime.Serialization
- From the Browse, navigate to your RockWeb/bin folder and select the following items or add them from the Tools | NuGet Package Manager | Manage NuGet Packages for Solution (except the Rock and Rock.Rest assemblies, you'll need to add them from RockWeb/bin):
  - EntityFramework.SqlServer.dll
  - EntityFramework.dll
  - Microsoft.Data.Edm.dll
  - Microsoft.Data.OData.dll
  - Newtonsoft.Json.dll

- DotLiquid.dll
- Rock.dll
- Rock.Lava.Shared.dll
- Rock.Rest.dll

Step 2 - Build a Model

Open your org.rocksolidchurch.SampleProject and create a class called *ReferralAgency.cs* under the Model folder. Put this class in your `org.rocksolidchurch.SampleProject.Model` namespace and have it extend `Rock.Data.Model` with the type ReferralAgency and extend `Rock.Security.ISecured` as shown below. You'll also want to add the `[DataContract]` class decorator in order to explicitly control serialization of your class properties.

```
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
using System.Data.Entity.ModelConfiguration;
using System.Linq;
using System.Runtime.Serialization;

using Rock.Data;
using Rock.Model;

namespace org.rocksolidchurch.SampleProject.Model
{
    [DataContract]
    public class ReferralAgency : Model<ReferralAgency>, IRockEntity
    {
    }
}
```

Before we add the properties for our class, let's tell the framework which table to use to store our data. Do this by adding the `[Table( "TABLE-NAME" )]` decorator above the class name. Use proper Rock Naming Conventions and name it like this:

```
[Table( "_org_rocksolidchurch_SampleProject_ReferralAgency" )]
// That line goes right above the class definition...
[DataContract]
public class ReferralAgency : Model<ReferralAgency>, IRockEntity
{
    //...
```

Now we can add properties for contact name, phone number, website, campus and agency type. We're going to tie our campus and agency type properties to the existing Rock *Campus* and *DefinedValue* entities. When we do this we'll create "virtual" properties to hold the reference to the object and regular int properties to store the related entity object's Id.

A DefinedValue what?

A *DefinedValue* represents one of the possible values for a *DefinedType*. And, you can think of a DefinedType like a custom field. You can review the details about DefinedTypes in an earlier chapter.

In our case, we'll create a referral agency 'type' that holds our possible pre-set values such as counseling, financial assistance, crisis hotline, food and clothing, etc. Later below, we'll force the block that manages our agencies to only use this particular DefinedType when setting the *type* of an agency.

```
public class ReferralAgency : Model<ReferralAgency>, IRockEntity
{
    // Now we'll add all our classes properties
    // except Id, Guid, CreatedByPersonAliasId, CreatedDateTime,
    // ModifiedByPersonAliasId, and ModifiedDateTime because
    // the Rock base Model class implements them for all models.

    [MaxLength( 100 )]
    [Required( ErrorMessage = "Name is required" )]
    [DataMember]
    public string Name { get; set; }

    [DataMember]
    public string Description { get; set; }

    [MaxLength( 100 )]
    [DataMember]
    public string ContactName { get; set; }

    [MaxLength( 100 )]
    [DataMember]
    public string PhoneNumber { get; set; }

    [MaxLength( 100 )]
    [DataMember]
    public string Website { get; set; }

    [DataMember]
    public int? CampusId { get; set; }

    [DataMember]
    public int? AgencyTypeValueId { get; set; }

    public virtual Campus Campus { get; set; }

    [DataMember]
    public virtual DefinedValue AgencyTypeValue { get; set; }

    //...
}
```

That's all there is to our class, but we do need a configuration class that tells Entity Framework how the virtual Campus and AgencyTypeValue properties relate to the int properties of our class. Just add this to the *ReferralAgency.cs* file just after the closing brace of your class (but before the closing brace of the namespace).

```
public partial class ReferralAgencyConfiguration : EntityTypeConfiguration<ReferralAgency>
```

```
{
    public ReferralAgencyConfiguration()
    {
        this.HasOptional( r => r.Campus ).WithMany().HasForeignKey( r => r.CampusId).WillCa
scadeOnDelete( false );
        this.HasOptional( r => r.AgencyTypeValue ).WithMany().HasForeignKey( p => p.AgencyT
ypeValueId ).WillCascadeOnDelete( false );

        // IMPORTANT!!
        this.HasEntitySetName( "ReferralAgency" );
    }
}
```

Note:

If you're new to the Entity Framework and you want to learn more about what's going on behind the scenes in Rock, you should read the Get Started with Entity Framework and/or the Working With DbContext MSDN articles.

Step 3 - Service Class

Now we'll create a `ReferralAgencyService` class which extends the `Rock.Data.Service` and uses the RockContext to communicate with the database. This is the class that you'll use in your blocks to fetch and store the referral agencies. Don't worry if this seems complicated to understand. This is just boilerplate code that you don't really have to worry too much about. It just glues our models to Rock's models.

```
using Rock.Data;

namespace org.rocksolidchurch.SampleProject.Model
{
    public class ReferralAgencyService : Service<ReferralAgency>
    {
        public ReferralAgencyService( RockContext context ) : base( context ) { }

        public bool CanDelete( ReferralAgency item, out string errorMessage )
        {
            errorMessage = string.Empty;
            return true;
        }
    }
}
```

It's pretty hard to believe, but that's really all there is to it. Rock's Entity Framework and LINQ does all the heavy lifting. Now, if you wanted to implement a method that only fetches a very specific set of data using your own custom LINQ, this is where you would put it.

Let's also create a constant to use in our code to refer to our new referral agency type *DefinedType*. We'll put these into a static class called DefinedType in our `org.rocksolidchurch.SampleProject.SystemGuid` namespace under the *SystemGuid* folder in our project.

```
using System;

namespace org.rocksolidchurch.SampleProject.SystemGuid
```

```
{
    public static class DefinedType
    {
        /// <summary>
        /// Types of Referral Agencies
        /// </summary>
        public const string REFERRAL_AGENCY_TYPE = "150478D4-3709-4543-906F-1F9496B4E7D0";
    }
}
```

> Note on Guids
>
> The Guid you generate for your constants are permanent and they will follow your application wherever it goes. You will include them (along with any other data that's needed in your new application) in something called a data "migration". Rock uses your data migration when it installs your application into Rock. You'll learn more about data migrations later.
>
> You can generate Guids any way you wish as long as they're unique.

Check your work by building the project in Visual Studio. `Ctrl+Shift+B`

Now go to the RockWeb/bin folder and add a reference to your new org.rocksolidchurch.SampleProject project.

In the next section we'll use what we learned in the previous guides to build blocks to add and edit agencies and to list them.

Step 4 - Back to Blocks: ReferralAgencyDetail Block

Following the Rock block convention, we need a block to add/view/edit an agency and one block to show the list of agencies. We'll also make these blocks follow common Rock UI patterns.

Find your *RockWeb\Plugins\org_rocksolidchurch\SampleProject* folder and create a `ReferralAgencyDetail` web usercontrol with an asp:UpdatePanel and with the necessary `using` statements to include your new data and model classes as shown here:

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

using Rock;
using Rock.Constants;
using Rock.Data;
using Rock.Model;
using Rock.Web.Cache;
using Rock.Web.UI;
using Rock.Web.UI.Controls;
using Rock.Attribute;

using org.rocksolidchurch.SampleProject.Model;
```

```
namespace RockWeb.Plugins.org_rocksolidchurch.SampleProject
{
    [DisplayName( "Referral Agency Detail" )]
    [Category( "rocksolidchurch > Sample Project" )]
    [Description( "Displays the details of a Referral Agency." )]

    public partial class ReferralAgencyDetail : Rock.Web.UI.RockBlock
    {
    }
}
```

```
<%@ Control Language="C#" AutoEventWireup="true" CodeFile="ReferralAgencyDetail.ascx.cs"
            Inherits="RockWeb.Plugins.org_rocksolidchurch.SampleProject.ReferralAgencyDetai
l" %>
<asp:UpdatePanel ID="upnlContent" runat="server">
    <ContentTemplate>

        <asp:Panel ID="pnlDetails" runat="server" Visible="false">

        </asp:Panel>

    </ContentTemplate>
</asp:UpdatePanel>
```

Markup (ReferralAgencyDetail.ascx)

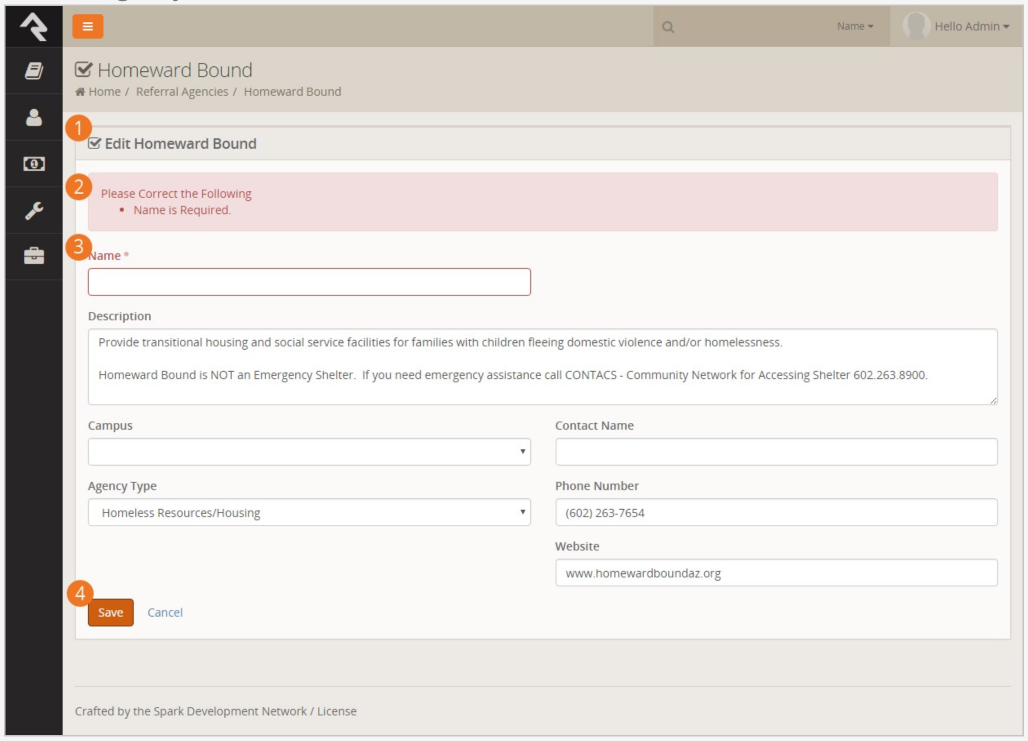
This block will show the details of a selected agency, so we need some usercontrols and markup inside the pnlDetails panel of our .ascx to display:

- a hidden field to keep track of the id of the agency
- the action the user's performing
- a validation summary area
- notification boxes for a possible warning/error and edit mode message
- a field for each of our agency's properties
- the save and cancel action buttons

Rock comes with a variety of user controls we'll use for making perfectly styled form fields. Use the `Rock:NotificationBox` for the notification boxes, a `Rock:CampusPicker` for the campus selector and `Rock:DataTextBox` for most of the remaining property fields. The generic `Rock:DataDropDownList` will be good to use for selecting the agency type.

> **Validation Tip!**
> The DataTextBox is great because it can perform automatic validation based on your entity model. So, if you have a `[MaxLength( 100 )]` attribute decorator on your Name property, it won't let the user enter more than 100 characters.

ReferralAgencyDetail



Homeward Bound

Home / Referral Agencies / Homeward Bound

Edit Homeward Bound

Please Correct the Following
- Name is Required.

Name *

Description

Provide transitional housing and social service facilities for families with children fleeing domestic violence and/or homelessness.

Homeward Bound is NOT an Emergency Shelter. If you need emergency assistance call CONTACS - Community Network for Accessing Shelter 602.263.8900.

Campus	Contact Name
Agency Type: Homeless Resources/Housing	Phone Number: (602) 263-7654
	Website: www.homewardboundaz.org

Crafted by the Spark Development Network / License

1 **Action Title Banner**
Shows the action and the name of the item being edited.

2 **Validation and Notification Area**
Displays any validation errors and edit or warning messages.

3 **Form Fields**
The main area for the edit fields.

4 **Action Buttons**
Location for the save, cancel, done, etc. buttons.

```
<asp:HiddenField ID="hfReferralAgencyId" runat="server" />

<div class="banner">
    <h1><asp:Literal ID="lActionTitle" runat="server" /></h1>
</div>

<asp:ValidationSummary ID="valSummaryTop" runat="server" HeaderText="Please Correct the Fol
lowing" CssClass="alert alert-danger" />
<Rock:NotificationBox ID="nbWarningMessage" runat="server" NotificationBoxType="Warning" />
<Rock:NotificationBox ID="nbEditModeMessage" runat="server" NotificationBoxType="Info" />

<div class="row">
    <div class="col-md-6">
        <Rock:DataTextBox ID="tbName" runat="server" SourceTypeName="org.rocksolidchurch.Sa
mpleProject.Model.ReferralAgency, org.rocksolidchurch.SampleProject" PropertyName="Name" />
    </div>
    <div class="col-md-6">
    </div>
</div>
```

```
<Rock:DataTextBox ID="tbDescription" runat="server" SourceTypeName="org.rocksolidchurch.SampleProject.Model.ReferralAgency, org.rocksolidchurch.SampleProject" PropertyName="Description" TextMode="MultiLine" Rows="4" />

<div class="row">
    <div class="col-md-6">
        <Rock:CampusPicker ID="cpCampus" runat="server" Label="Campus" />
        <Rock:DefinedValuePicker ID="dvpAgencyType" runat="server" Label="Agency Type" />
    </div>
    <div class="col-md-6">
        <Rock:DataTextBox ID="tbContactName" runat="server" SourceTypeName="org.rocksolidchurch.SampleProject.Model.ReferralAgency, org.rocksolidchurch.SampleProject" PropertyName="ContactName" />
        <Rock:DataTextBox ID="tbPhoneNumber" runat="server" SourceTypeName="org.rocksolidchurch.SampleProject.Model.ReferralAgency, org.rocksolidchurch.SampleProject" PropertyName="PhoneNumber" />
        <Rock:DataTextBox ID="tbWebsite" runat="server" SourceTypeName="org.rocksolidchurch.SampleProject.Model.ReferralAgency, org.rocksolidchurch.SampleProject" PropertyName="Website" />
    </div>
</div>

<div class="actions">
    <asp:LinkButton ID="btnSave" runat="server" Text="Save" CssClass="btn btn-primary" OnClick="btnSave_Click" />
    <asp:LinkButton ID="btnCancel" runat="server" Text="Cancel" CssClass="btn btn-link" CausesValidation="false" OnClick="btnCancel_Click" />
</div>
```

Now the real coding begins.

Code (ReferralAgencyDetail.ascx.cs)

Create an `OnInit` method and add the code to bind our agency type drop-down list to the particular referral agency type *DefinedType* that we defined in our `org.rocksolidchurch.SystemGuid.REFERRAL_AGENCY_TYPE` constant. We'll read it from Rock's cache for increased performance.

```
protected override void OnInit( EventArgs e )
{
    base.OnInit( e );

    this.BlockUpdated += Block_BlockUpdated;
    this.AddConfigurationUpdateTrigger( upnlContent );

    var definedType = DefinedTypeCache.Get( org.rocksolidchurch.SampleProject.SystemGuid.DefinedType.REFERRAL_AGENCY_TYPE.AsGuid() );
    if (definedType != null)
    {
        dvpAgencyType.DefinedTypeId = definedType.Id;
    }
}

protected void Block_BlockUpdated( object sender, EventArgs e )
{
    ShowDetail();
}
```

In the `OnLoad` event we'll just bind the campuses to the campus picker and then call a `ShowDetail()` method that we will create next to show all the details of the selected agency.

```
protected override void OnLoad( EventArgs e )
{
    base.OnLoad( e );

    if ( !Page.IsPostBack )
    {
        var campuses = CampusCache.All();
        cpCampus.Campuses = campuses;
        cpCampus.Visible = campuses.Any();

        ShowDetail();
    }
}
```

The `ShowDetail()` will need to:

- Fetch agency id given in the querystring. Use the `PageParameter` method to do this.
- Load the agency object using our *ReferralAgencyService* class
- Set the page and action title on the page.
- Bind the hidden id value and all the edit field values to the respective agency property values.
- Check the viewer's *EDIT* authorization.
- If not authorized to edit, display the read-only notification box, set the edit fields `ReadOnly`, property to true and hide the `Save` button.

```
private ReferralAgency _referralAgency = null;

private void ShowDetail()
{
    pnlDetails.Visible = true;

    int? referralAgencyId = PageParameter( "referralAgencyId" ).AsIntegerOrNull();
    int? campusId = PageParameter( "campusId" ).AsIntegerOrNull();
    int? agencyTypeValueId = PageParameter( "agencyTypeId" ).AsIntegerOrNull();

    ReferralAgency referralAgency = null;
    if (referralAgencyId.HasValue)
    {
        referralAgency = _referralAgency ?? new ReferralAgencyService( new RockContext() ).
Get( referralAgencyId.Value );
    }

    if (referralAgency != null)
    {
        RockPage.PageTitle = referralAgency.Name;
        lActionTitle.Text = ActionTitle.Edit( referralAgency.Name ).FormatAsHtmlTitle();
    }
    else
    {
        referralAgency = new ReferralAgency { Id = 0, CampusId = campusId, AgencyTypeValueI
d = agencyTypeValueId };
        RockPage.PageTitle = ActionTitle.Add( ReferralAgency.FriendlyTypeName );
        lActionTitle.Text = ActionTitle.Add( ReferralAgency.FriendlyTypeName ).FormatAsHtml
Title();
    }

    hfReferralAgencyId.Value = referralAgency.Id.ToString();
    tbName.Text = referralAgency.Name;
    tbDescription.Text = referralAgency.Description;
    cpCampus.SelectedCampusId = referralAgency.CampusId;
    dvpAgencyType.SetValue( referralAgency.AgencyTypeValueId );
    tbContactName.Text = referralAgency.ContactName;
    tbPhoneNumber.Text = referralAgency.PhoneNumber;
    tbWebsite.Text = referralAgency.Website;

    bool readOnly = false;

    nbEditModeMessage.Text = string.Empty;
    if ( !IsUserAuthorized(Rock.Security.Authorization.EDIT) )
    {
        readOnly = true;
        nbEditModeMessage.Text = EditModeMessage.ReadOnlyEditActionNotAllowed( ReferralAgen
cy.FriendlyTypeName );
    }

    if (readOnly)
    {
        lActionTitle.Text = ActionTitle.View( ReferralAgency.FriendlyTypeName );
        btnCancel.Text = "Close";
    }

    tbName.ReadOnly = readOnly;
    tbDescription.ReadOnly = readOnly;
    tbContactName.ReadOnly = readOnly;
    tbPhoneNumber.ReadOnly = readOnly;
    tbWebsite.ReadOnly = readOnly;

    btnSave.Visible = !readOnly;
}
```

You may have noticed we also declared a new `_referralAgency` private ReferralAgency property for this block, and we try getting the agency object from there first before we attempt to load it using our ReferralAgencyService. We're doing this for performance reasons. As you'll see in a few minutes, we may have already loaded it inside the `GetBreadCrumbs` method we're going to create.

Now let's write the `btnSave_Click` code that handles the `Save` button click event. In here we need to:

- Create a database context. We'll use this dataContext with our ReferralAgencyService to save the changes we're about to make to a referral agency.
- If we're editing a new agency we use the service's `.Add(object)` method, otherwise we fetch the agency fresh from the database using the service's `.Get(int)` method.
- Set the agency's property values using the values in the edit fields.
- Check to see if the agency is valid and the page is also valid.
- Save our changes to the database and navigate back to the parent page.

```
protected void btnSave_Click( object sender, EventArgs e )
{
    ReferralAgency referralAgency;
    var dataContext = new RockContext();
    var service = new ReferralAgencyService( dataContext );

    int referralAgencyId = int.Parse( hfReferralAgencyId.Value );

    if ( referralAgencyId == 0 )
    {
        referralAgency = new ReferralAgency();
        service.Add( referralAgency );
    }
    else
    {
        referralAgency = service.Get( referralAgencyId );
    }

    referralAgency.Name = tbName.Text;
    referralAgency.Description = tbDescription.Text;
    referralAgency.CampusId = cpCampus.SelectedCampusId;
    referralAgency.AgencyTypeValueId = dvpAgencyType.SelectedValueAsId();
    referralAgency.ContactName = tbContactName.Text;
    referralAgency.PhoneNumber = tbPhoneNumber.Text;
    referralAgency.Website = tbWebsite.Text;

    if ( !referralAgency.IsValid || !Page.IsValid )
    {
        // Controls will render the error messages
        return;
    }

    dataContext.SaveChanges();

    NavigateToParentPage();

}
```

Let's tap into Rock's breadcrumb system. Just override the `GetBreadCrumbs` method and add the name of the agency we're displaying or editing into the breadcrumbs. When we're just adding a new agency we'll set the crumb's name to the default 'add' action title for the general type name of the *ReferralAgency* class. The code looks like this:

```
public override List<BreadCrumb> GetBreadCrumbs( Rock.Web.PageReference pageReference )
{
    var breadCrumbs = new List<BreadCrumb>();

    string crumbName = ActionTitle.Add( ReferralAgency.FriendlyTypeName );

    int? referralAgencyId = PageParameter( "referralAgencyId" ).AsIntegerOrNull();
    if ( referralAgencyId.HasValue )
    {
        _referralAgency = new ReferralAgencyService( new RockContext() ).Get( referralAgenc
yId.Value );
        if ( _referralAgency != null )
        {
            crumbName = _referralAgency.Name;
        }
    }

    breadCrumbs.Add( new BreadCrumb( crumbName, pageReference ) );

    return breadCrumbs;
}
```

Lastly, if someone clicks the `Cancel` button we'll just write a handler to navigate back to the parent page.

```
protected void btnCancel_Click( object sender, EventArgs e )
{
    NavigateToParentPage();
}
```

Step 5 - ReferralAgencyList Block

Now we need a block to list the agencies. Find your *RockWeb\Plugins\org_rocksolidchurch\SampleProject* folder and create a `ReferralAgencyList` web usercontrol with an asp:UpdatePanel and with this template code in your code-behind file as shown here:

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

using Rock;
using Rock.Attribute;
using Rock.Data;
using Rock.Model;
using Rock.Web.Cache;
using Rock.Web.UI.Controls;

using org.rocksolidchurch.SampleProject.Model;

namespace RockWeb.Plugins.org_rocksolidchurch.SampleProject
{
    [DisplayName( "Referral Agency List" )]
    [Category( "rocksolidchurch > Sample Project" )]
    [Description( "Lists all the Referral Agencies." )]

    [LinkedPage( "Detail Page" )]
    public partial class ReferralAgencyList : Rock.Web.UI.RockBlock
    {
    }
}
```

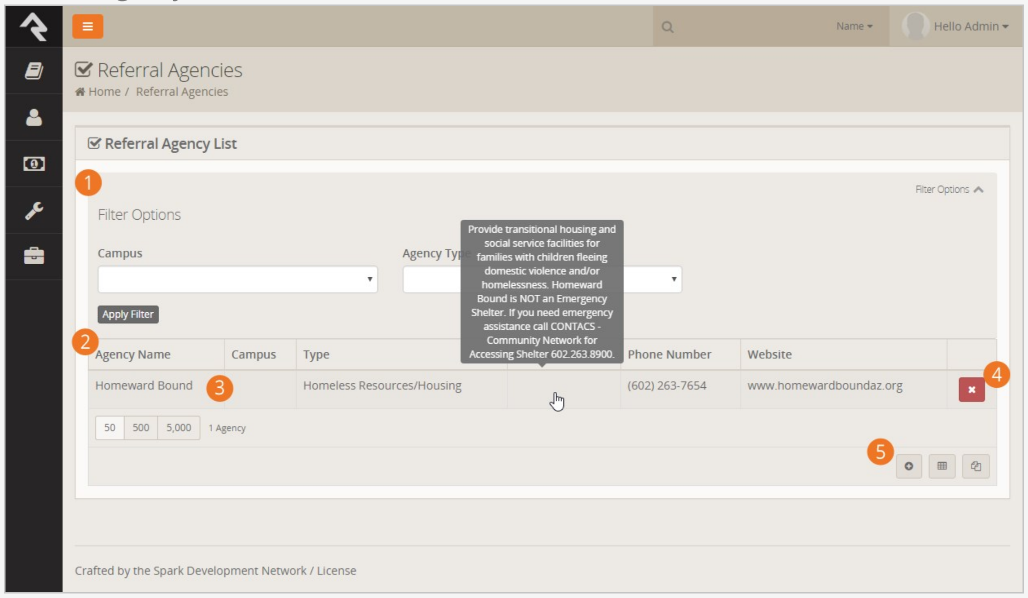
Notice that we've added a `LinkedPage` block attribute. Once it's on a page, we'll configure it to the page that has an instance of the *ReferralAgencyDetail* block we just created.

Markup (ReferralAgencyList.ascx)

This block will show a list of agencies, so we need a Grid and let's also add a GridFilter to make it easy to filter agencies by type. This block will also allow for:

- deleting agencies
- navigating to an agency detail when selected
- adding a new agency
- saving the viewer's filter settings

ReferralAgencyList


Name
Hello Admin
Referral Agencies
Home / Referral Agencies
Referral Agency List
1
Filter Options
Filter Options
Campus
Agency Type
Apply Filter
Provide transitional housing and social service facilities for families with children fleeing domestic violence and/or homelessness. Homeward Bound is NOT an Emergency Shelter. If you need emergency assistance call CONTACS - Community Network for Accessing Shelter 602.263.8900.
2
Agency Name
Campus
Type
Phone Number
Website
Homeward Bound
3
Homeless Resources/Housing
(602) 263-7654
www.homewardboundaz.org
4
50
500
5,000
1 Agency
5
Crafted by the Spark Development Network / License

1. **Rock GridFilter**
   Collapsible region that holds the filter for the grid that includes a Rock CampusPicker and a RockDropDownList.

2. **Rock Grid**
   A Rock Grid showing a list of items.

3. **A row with ASP BoundFields**
   A selectable row showing desired BoundFields for an item and with the Grid's TooltipField set to show the item Description on hover-over.

4. **Rock DeleteField**
   Standard delete button for deleting an item in a grid row.

5. **Grid Action Bar**
   Location for any actions associated with the grid items including, add, export, etc.

```
<%@ Control Language="C#" AutoEventWireup="true" CodeFile="ReferralAgencyList.ascx.cs" Inhe
rits="RockWeb.Plugins.org_rocksolidchurch.SampleProject.ReferralAgencyList" %>
<asp:UpdatePanel ID="upnlContent" runat="server">
    <ContentTemplate>

        <Rock:GridFilter ID="gfSettings" runat="server">
            <Rock:CampusPicker ID="cpCampus" runat="server" Label="Campus" />
            <Rock:DefinedValuePicker ID="dvpAgencyType" runat="server" Label="Agency Type"
/>
        </Rock:GridFilter>

        <Rock:ModalAlert ID="mdGridWarning" runat="server" />

        <Rock:Grid ID="gAgencies" runat="server" AllowSorting="true" OnRowSelected="gAgenci
es_Edit" TooltipField="Description">
            <Columns>
                <asp:BoundField DataField="Name" HeaderText="Agency Name" SortExpression="N
ame" />
```

```
                <asp:BoundField DataField="Campus.Name" HeaderText="Campus" SortExpression=
"Campus.Name" />
                <asp:BoundField DataField="AgencyTypeValue.Value" HeaderText="Type" SortExp
ression="AgencyTypeValue.Value" />
                <asp:BoundField DataField="ContactName" HeaderText="Contact Name" SortExpre
ssion="ContactName" />
                <asp:BoundField DataField="PhoneNumber" HeaderText="Phone Number" SortExpre
ssion="PhoneNumber" />
                <asp:BoundField DataField="Website" HeaderText="Website" SortExpression="We
bsite" />
                <Rock:DeleteField OnClick="gAgencies_Delete" />
            </Columns>
        </Rock:Grid>

    </ContentTemplate>
</asp:UpdatePanel>
```

Code (ReferralAgencyList.ascx.cs)

In the `OnInit` method we will:

- Check the viewer's *EDIT* authorization.
- If authorized to edit, enable the Add button on the grid and the row's' delete item button.
- Register grid filter event handlers for when the `Apply` button is pressed and for when the grid filter summary is being built.
- Tell the grid which key/identifier property to use from the row's item.
- Tell the grid the friendly name of the items that will be displaying.
- Bind the agency types and campuses in the filter.

```
protected override void OnInit( EventArgs e )
{
    base.OnInit( e );

    bool canEdit = IsUserAuthorized( Rock.Security.Authorization.EDIT );

    gfSettings.ApplyFilterClick += gfSettings_ApplyFilterClick;
    gfSettings.DisplayFilterValue += gfSettings_DisplayFilterValue;

    gAgencies.Actions.ShowAdd = canEdit;
    gAgencies.IsDeleteEnabled = canEdit;
    gAgencies.Actions.AddClick += gAgencies_Add;

    gAgencies.RowItemText = "Agency";
    gAgencies.DataKeyNames = new string[] { "id" };

    BindFilter();
}
```

The `BindFilter()` method binds the campuses to the CampusPicker control and the agency type drop-down list to our `REFERRAL_AGENCY_TYPE`.

```
private void BindFilter()
{
    var campuses = CampusCache.All();
    cpCampus.Campuses = campuses;
    cpCampus.Visible = campuses.Any();

    var definedType = DefinedTypeCache.Get( org.rocksolidchurch.SampleProject.SystemGuid.De
finedType.REFERRAL_AGENCY_TYPE.AsGuid() );
```

```
    if ( definedType != null )
    {
        dvpAgencyType.DefinedTypeId = definedType.Id;
    }
}
```

The `OnLoad` method uses any previously saved filter settings to select the correct filter items before it calls a `BindGrid()` method.

```
protected override void OnLoad( EventArgs e )
{
    base.OnLoad( e );

    if ( !Page.IsPostBack )
    {
        var preferences = GetBlockPersonPreferences();
        cpCampus.SetValue( preferences.GetValue( "Campus" ).AsIntegerOrNull() );
        dvpAgencyType.SelectedValue = preferences.GetValue( "Agency Type" );

        BindGrid();
    }
}
```

Our `BindGrid()` method uses our *ReferralAgencyService* to fetch the agencies and the grid filter settings to intelligently exclude any that don't match our filter. It also sorts the data using any sort option set on the grid by the viewer.

```
private void BindGrid()
{
    var service = new ReferralAgencyService( new RockContext() );
    SortProperty sortProperty = gAgencies.SortProperty;

    var query = service.Queryable( "Campus,AgencyTypeValue" );

    var preferences = GetBlockPersonPreferences();
    int? campusId = preferences.GetValue( "Campus" ).AsIntegerOrNull();
    if ( campusId.HasValue )
    {
        query = query.Where( a => a.CampusId == campusId.Value );
    }

    int? definedValueId = preferences.GetValue( "Agency Type" ).AsIntegerOrNull();
    if ( definedValueId.HasValue )
    {
        query = query.Where( a => a.AgencyTypeValueId == definedValueId.Value );
    }

    // Sort results
    if ( sortProperty != null )
    {
        gAgencies.DataSource = query.Sort( sortProperty ).ToList();
    }
    else
    {
        gAgencies.DataSource = query.OrderBy( a => a.Name ).ToList();
    }

    gAgencies.DataBind();
}
```

We'll implement the `gfSettings_ApplyFilterClick` handler to save the user's filter preferences when they press the `Apply` button in the filter before calling `BindGrid()`.

```
protected void gfSettings_ApplyFilterClick( object sender, EventArgs e )
{
    gfSettings.SetFilterPreference( "Campus", ( cpCampus.SelectedCampusId != null ?
        cpCampus.SelectedCampusId.Value.ToString() : string.Empty ) );
    gfSettings.SetFilterPreference( "Agency Type", dvpAgencyType.SelectedValue );

    BindGrid();
}
```

You wouldn't want the filter summary to show you're currently filtering on agency type "3", right? For this reason, it's the job of the `gfSettings_DisplayFilterValue` handler to turn the selected *id* for each filter item into a user readable "name".

```
protected void gfSettings_DisplayFilterValue( object sender, GridFilter.DisplayFilterValueA
rgs e )
{
    switch ( e.Key )
    {
        case "Campus":
            {
                if ( !string.IsNullOrWhiteSpace( e.Value ) )
                {
                    e.Value = CampusCache.Get( int.Parse( e.Value ) ).Name;
                }
                break;
            }

        case "Agency Type":
            {
                var preferences = GetBlockPersonPreferences();
                int? valueId = preferences.GetValue( "Agency Type" ).AsIntegerOrNull();
                if ( valueId.HasValue )
                {
                    var definedValue = DefinedValueCache.Get( valueId.Value );
                    if ( definedValue != null )
                    {
                        e.Value = definedValue.Value;
                    }
                }
                break;
            }

        default:
            {
                e.Value = string.Empty;
                break;
            }
    }
}
```

The `gAgencies_Add` and `gAgencies_Edit` methods are similar except we'll pass a "0" to the detail page when we're about to add a *new* agency and we'll pass the selected row's agency Id to edit an existing agency.

```
protected void gAgencies_Add( object sender, EventArgs e )
{
    NavigateToDetailPage( 0 );
}
```

```
protected void gAgencies_Edit( object sender, RowEventArgs e )
{
    NavigateToDetailPage( e.RowKeyId );
}
```

Our `NavigateToDetailPage` method will build an appropriate querystring and then navigate to the detail page. The detail page is determined by the LinkedPage block attribute we named "DetailPage".

```
private void NavigateToDetailPage( int referralAgencyId )
{
    var preferences = GetBlockPersonPreferences();
    var queryParams = new Dictionary<string, string>();
    queryParams.Add( "referralAgencyId", referralAgencyId.ToString() );
    queryParams.Add( "campusId", preferences.GetValue( "Campus" ) );
    queryParams.Add( "agencyTypeId", preferences.GetValue( "Agency Type" ) );
    NavigateToLinkedPage( "DetailPage", queryParams );
}
```

We're almost done! The `gAgencies_Delete` handler will check the selected agency to verify we can delete it and show a warning if we can't. Otherwise it uses a database context and our *ReferralAgencyService* once again but this time to delete and save the changes. After saving, we'll rebind the grid to reflect the change.

```
protected void gAgencies_Delete( object sender, RowEventArgs e )
{
    var dataContext = new RockContext();
    var service = new ReferralAgencyService( dataContext );
    var referralAgency = service.Get( (int)e.RowKeyValue );
    if ( referralAgency != null )
    {
        string errorMessage;
        if ( !service.CanDelete( referralAgency, out errorMessage ) )
        {
            mdGridWarning.Show( errorMessage, ModalAlertType.Information );
            return;
        }

        service.Delete( referralAgency );
        dataContext.SaveChanges();
    }

    BindGrid();
}
```

All we have to do for our `gAgencies_GridRebind` handler is rebind the data to the grid.

```
protected void gAgencies_GridRebind( object sender, EventArgs e )
{
    BindGrid();
}
```

Step 6 - Page and Block Setup

The last thing we'll do is add two pages where we can see the list of all agencies and the details for a selected (or new) agency.

1. Add a new page under your favorite test-menu-page and call it "Referral Agencies".
2. Under it, add a child page called "Referral Agency Details".

3. Add the ReferralAgencyDetail block to this *Referral Agency Details* page.
4. Add the ReferralAgencyList block back on the *Referral Agencies* parent page.
5. Configure this ReferralAgencyList block instance. Set the LinkedPage to the *Referral Agency Details* page.
6. *Don't try to load the block yet- you may need to include a migration that we'll cover in the next section to avoid getting an exception*

This is a very common parent-child page pattern in Rock. It makes the navigation between list and detail a predictable and easy thing to code using the `NavigateToParentPage()` method.

> **Need code?**
> The code for this chapter is already in your Rockit SDK. You're welcome. :)

The Data Migration

There's one more important piece of the puzzle. It's the part that adds any new tables, schema, and data to the database needed for your plugin/application. We call them *data migrations*.

If you had tried running the example code from the previous chapter without the corresponding data migration, you would have run into trouble. We sorta snuck them in there for you, but now it's time to learn how to handle this yourself.

Behind the Scenes

When Rock starts it looks for any *Migration* classes inside any plugin assemblies. Without trying to explain all the magic that goes on, let's just say it figures which ones have not yet been run and calls their `Up()` methods.

All you really need to know is:

- Migrations are ordered.
- Migrations can be marked to require a minimum version of Rock.
- Migrations must have an `Up()` and a `Down()` method.
- Your `Up()` method must add any needed tables and data.
- Your `Down()` method must remove any corresponding data and tables it added.
- You should not use a RockContext to accomplish any of your data movement.

> RockContext
>
> This rule is 100% true for the core Entity Framework migrations but we've had mixed concerns with trying to use a RockContext in a plugin migration. So, we highly advise not doing it because it may not work in the future.

Anatomy of a Migration

Migrations should be added into your org.rocksolidchurch.SampleProject under the *Migrations* folder. The files should also follow the 000_ClassName.cs naming convention so they are sorted in your folder appropriately.

The class name may be what ever you wish but it must extend Rock's base `Rock.Plugin.Migration` class as seen here:

```
namespace org.rocksolidchurch.SampleProject.Migrations
{
    [MigrationNumber( 1, "1.0.13" )]
    public class CreateDb : Migration
    {
        public override void Up()
        {
            Sql( @"..." );
        }

        public override void Down()
        {
            Sql( @"..." );
        }
    }
}
```

The `[MigrationNumber(1, "1.0.13")]` indicates it is the first migration and it requires Rock version 1.0.13 in order to be installed.

You can include any general SQL statements in the `Sql( string )` calls and you can also use one of the many helper methods Rock has for adding pages, adding blocks, attributes, entities, etc. You'll find a list in the Data Migration Helper Methods section of the next book or by examining the methods in the *RockMigrationHelper* class.

001_CreateDb Migration

For our ReferralAgency class we created a migration to add a new table, columns, constraints and foreign key references as seen in this `Up()` method below.

You will notice several columns listed below that were not part of the model we defined. Properties for each of these comes from the Rock base Model class and they are required to be defined in your table's columns:

- Id [int] IDENTITY(1,1) NOT NULL
- Guid [uniqueidentifier] NOT NULL
- CreatedDateTime [datetime] NULL
- ModifiedDateTime [datetime] NULL
- CreatedByPersonAliasId [int] NULL
- ModifiedByPersonAliasId [int] NULL
- ForeignKey [nvarchar](50) NULL
- ForeignGuid [uniqueidentifier](50) NULL
- ForeignId [nvarchar](50) NULL

```
public override void Up()
{
        Sql( @"
CREATE TABLE [dbo].[_org_rockSolidChurch_SampleProject_ReferralAgency](
    [Id] [int] IDENTITY(1,1) NOT NULL,
    [Name] [nvarchar](100) NOT NULL,
    [Description] [nvarchar](max) NULL,
    [ContactName] [nvarchar](100) NULL,
    [PhoneNumber] [nvarchar](100) NULL,
    [Website] [nvarchar](100) NULL,
    [CampusId] [int] NULL,
    [AgencyTypeValueId] [int] NULL,
    [Guid] [uniqueidentifier] NOT NULL,
    [CreatedDateTime] [datetime] NULL,
    [ModifiedDateTime] [datetime] NULL,
    [CreatedByPersonAliasId] [int] NULL,
    [ModifiedByPersonAliasId] [int] NULL,
 [ForeignKey] [nvarchar](50) NULL,
 [ForeignGuid] [uniqueidentifier] NULL,
    [ForeignId] [nvarchar](50) NULL,
 CONSTRAINT [PK_dbo._org_rockSolidChurch_SampleProject_ReferralAgency] PRIMARY KEY CLUSTERE
D
(
    [Id] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS
 = ON, ALLOW_PAGE_LOCKS = ON)
)

ALTER TABLE [dbo].[_org_rockSolidChurch_SampleProject_ReferralAgency]  WITH CHECK ADD  CONS
TRAINT [FK_dbo._org_rockSolidChurch_SampleProject_ReferralAgency_dbo.DefinedValue_ReferralA
gencyTypeValueId] FOREIGN KEY([AgencyTypeValueId])
REFERENCES [dbo].[DefinedValue] ([Id])

ALTER TABLE [dbo].[_org_rockSolidChurch_SampleProject_ReferralAgency] CHECK CONSTRAINT [FK_
dbo._org_rockSolidChurch_SampleProject_ReferralAgency_dbo.DefinedValue_ReferralAgencyTypeVa
lueId]

ALTER TABLE [dbo].[_org_rockSolidChurch_SampleProject_ReferralAgency]  WITH CHECK ADD  CONS
TRAINT [FK_dbo._org_rockSolidChurch_SampleProject_ReferralAgency_dbo.Campus_CampusId] FOREI
GN KEY([CampusId])
REFERENCES [dbo].[Campus] ([Id])

ALTER TABLE [dbo].[_org_rockSolidChurch_SampleProject_ReferralAgency] CHECK CONSTRAINT [FK_
dbo._org_rockSolidChurch_SampleProject_ReferralAgency_dbo.Campus_CampusId]

ALTER TABLE [dbo].[_org_rockSolidChurch_SampleProject_ReferralAgency]  WITH CHECK ADD  CONS
TRAINT [FK_dbo._org_rockSolidChurch_SampleProject_ReferralAgency_dbo.PersonAlias_CreatedByP
ersonAliasId] FOREIGN KEY([CreatedByPersonAliasId])
REFERENCES [dbo].[PersonAlias] ([Id])

ALTER TABLE [dbo].[_org_rockSolidChurch_SampleProject_ReferralAgency] CHECK CONSTRAINT [FK_
dbo._org_rockSolidChurch_SampleProject_ReferralAgency_dbo.PersonAlias_CreatedByPersonAliasI
d]

ALTER TABLE [dbo].[_org_rockSolidChurch_SampleProject_ReferralAgency]  WITH CHECK ADD  CONS
TRAINT [FK_dbo._org_rockSolidChurch_SampleProject_ReferralAgency_dbo.PersonAlias_ModifiedBy
PersonAliasId] FOREIGN KEY([ModifiedByPersonAliasId])
REFERENCES [dbo].[PersonAlias] ([Id])

ALTER TABLE [dbo].[_org_rockSolidChurch_SampleProject_ReferralAgency] CHECK CONSTRAINT [FK_
dbo._org_rockSolidChurch_SampleProject_ReferralAgency_dbo.PersonAlias_ModifiedByPersonAlias
Id]
" );
}
```

The `Down()` method in that migration drops the constraints and table:

```
public override void Down()
{
    Sql(@"
ALTER TABLE [dbo].[_org_rockSolidChurch_SampleProject_ReferralAgency] DROP CONSTRAINT [FK_d
bo._org_rockSolidChurch_SampleProject_ReferralAgency_dbo.PersonAlias_ModifiedByPersonAliasI
d]
ALTER TABLE [dbo].[_org_rockSolidChurch_SampleProject_ReferralAgency] DROP CONSTRAINT [FK_d
bo._org_rockSolidChurch_SampleProject_ReferralAgency_dbo.PersonAlias_CreatedByPersonAliasId
]
ALTER TABLE [dbo].[_org_rockSolidChurch_SampleProject_ReferralAgency] DROP CONSTRAINT [FK_d
bo._org_rockSolidChurch_SampleProject_ReferralAgency_dbo.Campus_CampusId]
ALTER TABLE [dbo].[_org_rockSolidChurch_SampleProject_ReferralAgency] DROP CONSTRAINT [FK_d
bo._org_rockSolidChurch_SampleProject_ReferralAgency_dbo.DefinedValue_ReferralAgencyTypeVal
ueId]
DROP TABLE [dbo].[_org_rockSolidChurch_SampleProject_ReferralAgency]
" );
}
```

002_AddSystemData Migration

In a second migration we added the data we wanted to include with our plugin. We wanted to:

1. add some new pages
2. set some page settings
3. explicitly add our new block types to Rock (versus letting Rock auto add them)
4. add instances of the new block types to our new pages
5. set block attributes on those new block instances
6. add our "Referral Agency Type" DefinedType
7. add the possible values for our new DefinedType

```
namespace org.rocksolidchurch.SampleProject.Migrations
{
    [MigrationNumber( 2, "1.0.8" )]
    public class AddSystemData : Migration
    {
        public override void Up()
        {
            RockMigrationHelper.AddPage( "7F2581A1-941E-4D51-8A9D-5BE9B881B003", "D65F783D-
87A9-4CC9-8110-E83466A0EADB", "Referral Agencies", "", "223AC4F2-CBED-4733-807A-188CFBBFA0C
8", "" ); // Site:Rock RMS
            RockMigrationHelper.AddPage( "223AC4F2-CBED-4733-807A-188CFBBFA0C8", "D65F783D-
87A9-4CC9-8110-E83466A0EADB", "Referral Agency Details", "", "4BF8FA57-AE86-4103-B07E-80ECE
0000AEE", "" ); // Site:Rock RMS

            // Since the Referral Agency Details block handles displaying the breadcrumb fo
r the page, we need to turn off the default breadcrumb rendered by the page
            Sql( @"
    UPDATE [Page] SET [BreadCrumbDisplayName] = 0 WHERE [Guid] = '4BF8FA57-AE86-4103-B07E-8
0ECE0000AEE'
" );
            RockMigrationHelper.UpdateBlockType( "Referral Agency Detail", "Displays the de
tails of a Referral Agency.", "~/Plugins/org_rocksolidchurch/SampleProject/ReferralAgencyDe
tail.ascx", "rocksolidchurch > Sample Project", "2F130DF6-1EE4-45CE-9410-CBB0517EB33E" );
            RockMigrationHelper.UpdateBlockType( "Referral Agency List", "Lists all the Ref
erral Agencies.", "~/Plugins/org_rocksolidchurch/SampleProject/ReferralAgencyList.ascx", "r
ocksolidchurch > Sample Project", "53F447CE-4B91-470A-A15D-B60DCAAB29CB" );
            // Add Block to Page: Referral Agencies, Site: Rock RMS
            RockMigrationHelper.AddBlock( "223AC4F2-CBED-4733-807A-188CFBBFA0C8", "", "53F4
47CE-4B91-470A-A15D-B60DCAAB29CB", "Referral Agency List", "Main", "", "", 0, "A0B53736-413
2-4D1B-8300-9F9FB1A5DC21" );
            // Add Block to Page: Referral Agency Details, Site: Rock RMS
            RockMigrationHelper.AddBlock( "4BF8FA57-AE86-4103-B07E-80ECE0000AEE", "", "2F13
0DF6-1EE4-45CE-9410-CBB0517EB33E", "Referral Agency Detail", "Main", "", "", 0, "B69EBD0E-A
1B4-47C5-AAE7-B40BEA37965F" );
            // Attrib for BlockType: Referral Agency List:Detail Page
            RockMigrationHelper.AddBlockTypeAttribute( "53F447CE-4B91-470A-A15D-B60DCAAB29C
B", "BD53F9C9-EBA9-4D3F-82EA-DE5DD34A8108", "Detail Page", "DetailPage", "", "", 0, @"", "5
B480350-663C-4789-BF4D-33EC8DF882E8" );
            // Attrib Value for Block:Referral Agency List, Attribute:Detail Page Page: Ref
erral Agencies, Site: Rock RMS
            RockMigrationHelper.AddBlockAttributeValue( "A0B53736-4132-4D1B-8300-9F9FB1A5DC
21", "5B480350-663C-4789-BF4D-33EC8DF882E8", @"4bf8fa57-ae86-4103-b07e-80ece0000aee" );
            RockMigrationHelper.AddDefinedType( "Global", "Referral Agency Type", "The type
 of agency (e.g. Counseling, Food, Financial Assistance, etc.)", "150478D4-3709-4543-906F-1
F9496B4E7D0");
            RockMigrationHelper.AddDefinedValue( "150478D4-3709-4543-906F-1F9496B4E7D0", "C
ounseling and Therapy", "", "83F9A59C-DBE5-4E1A-B33C-F701FA8175E1", false );
            RockMigrationHelper.AddDefinedValue( "150478D4-3709-4543-906F-1F9496B4E7D0", "F
inancial Assistance or Counseling", "", "7A30D312-996E-4823-B1FF-AA27C1806521", false );
            RockMigrationHelper.AddDefinedValue( "150478D4-3709-4543-906F-1F9496B4E7D0", "2
4 Hour Crisis Hotlines", "", "EDBE6DCE-313F-4648-8D97-A39520A54BFC", false );
            RockMigrationHelper.AddDefinedValue( "150478D4-3709-4543-906F-1F9496B4E7D0", "Y
outh Resources", "", "BB666FA1-5391-40B1-B334-3A27575AD9D5", false );
            RockMigrationHelper.AddDefinedValue( "150478D4-3709-4543-906F-1F9496B4E7D0", "F
ood and Clothing", "", "E15AE7DE-3555-437B-99B0-B28601C4EA2D", false );
            RockMigrationHelper.AddDefinedValue( "150478D4-3709-4543-906F-1F9496B4E7D0", "H
omeless Resources/Housing", "", "F6E4D78C-E05A-4AEF-AF8C-09B3B8FDDEBF", false );
            RockMigrationHelper.AddDefinedValue( "150478D4-3709-4543-906F-1F9496B4E7D0", "S
ubstance Abuse", "", "AD01D370-7CB6-4261-ACF6-8EE21CB353AA", false );
            RockMigrationHelper.AddDefinedValue( "150478D4-3709-4543-906F-1F9496B4E7D0", "R
esidential Drug Treatment Centers", "", "57F4BCC8-B80F-48E5-93E2-A76E3F572C0C", false );
            RockMigrationHelper.AddDefinedValue( "150478D4-3709-4543-906F-1F9496B4E7D0", "D
omestic Violence Resources", "", "AE95FD8A-FD9E-4EDD-9689-5491725FEFE6", false );
            RockMigrationHelper.AddDefinedValue( "150478D4-3709-4543-906F-1F9496B4E7D0", "M
ediation", "", "40C66BE2-CE13-4E7D-980A-A2D66968CE57", false );
            RockMigrationHelper.AddDefinedValue( "150478D4-3709-4543-906F-1F9496B4E7D0", "M
iscellaneous", "", "62AB4A35-6E72-4BCD-BF6A-5B0D2052BACA", false );
        }

        public override void Down()
        {
            Sql( @"
    UPDATE [_org_rockSolidChurch_SampleProject_ReferralAgency] SET [AgencyTypeValueId] = NU
LL
" );
            RockMigrationHelper.DeleteDefinedType( "150478D4-3709-4543-906F-1F9496B4E7D0" )
;

            // Attrib for BlockType: Referral Agency List:Detail Page
            RockMigrationHelper.DeleteAttribute( "5B480350-663C-4789-BF4D-33EC8DF882E8" );

            // Remove Block: Referral Agency Detail, from Page: Referral Agency Details, Si
te: Rock RMS
            RockMigrationHelper.DeleteBlock( "B69EBD0E-A1B4-47C5-AAE7-B40BEA37965F" );

            // Remove Block: Referral Agency List, from Page: Referral Agencies, Site: Rock
 RMS
            RockMigrationHelper.DeleteBlock( "A0B53736-4132-4D1B-8300-9F9FB1A5DC21" );
            RockMigrationHelper.DeleteBlockType( "53F447CE-4B91-470A-A15D-B60DCAAB29CB" );
// Referral Agency List
            RockMigrationHelper.DeleteBlockType( "2F130DF6-1EE4-45CE-9410-CBB0517EB33E" );
// Referral Agency Detail
            RockMigrationHelper.DeletePage( "4BF8FA57-AE86-4103-B07E-80ECE0000AEE" ); // Pa
ge: Referral Agency DetailsLayout: Full Width, Site: Rock RMS
            RockMigrationHelper.DeletePage( "223AC4F2-CBED-4733-807A-188CFBBFA0C8" ); // Pa
ge: Referral AgenciesLayout: Full Width, Site: Rock RMS

        }
    }
}
```

Tip

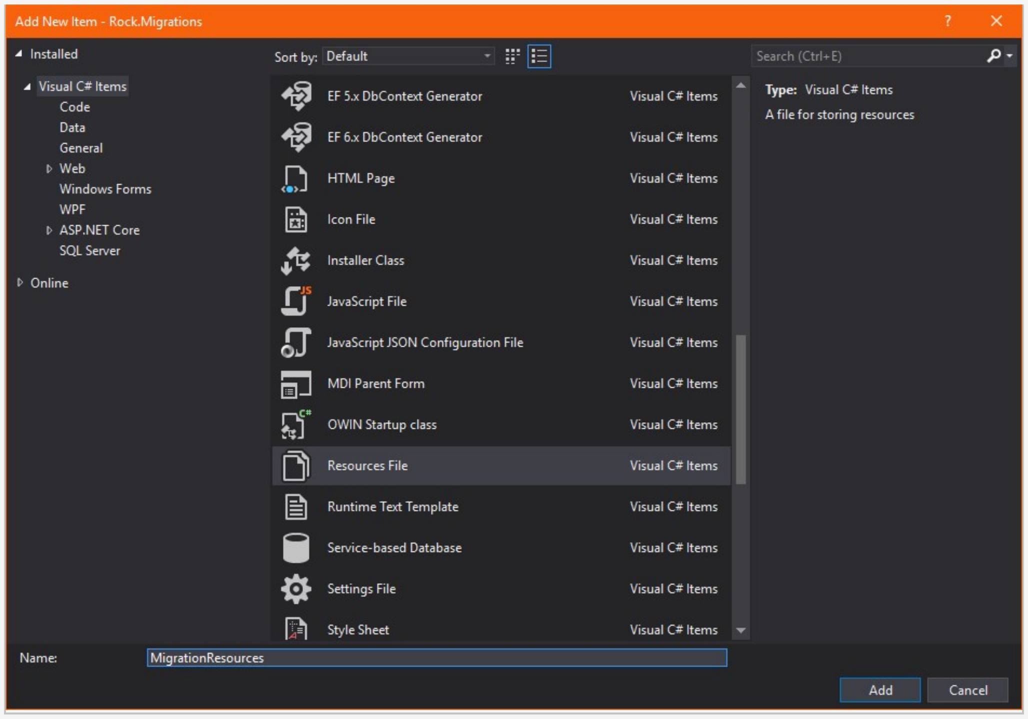
You generally want to explicitly add new block types so that you can set well-known Guid values for them. Doing that let's you maintain control -- so you can find them in other Rock installations, refer to them by guid when you're adding instances of them to pages, or setting attributes for them, etc. In fact, you should think about doing this for anything you may need to update at a later date.

Migration Resources

Although the examples above show stuff like inline SQL directly in the migration class file, there is another way which lets you avoid the need to escape everything for a string.

It is the Resource File. Just right-click your Migration folder and choose Add New Item... Select *Resource File* and name it MigrationResources.

Add New Item... Resource File



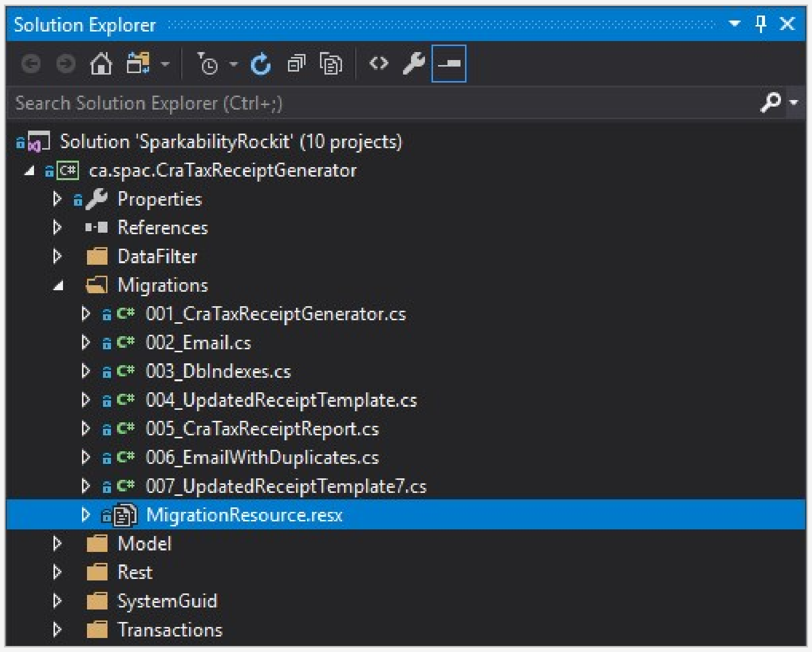
Add New Item - Rock.Migrations

Name: MigrationResources

Type: Visual C# Items

A file for storing resources

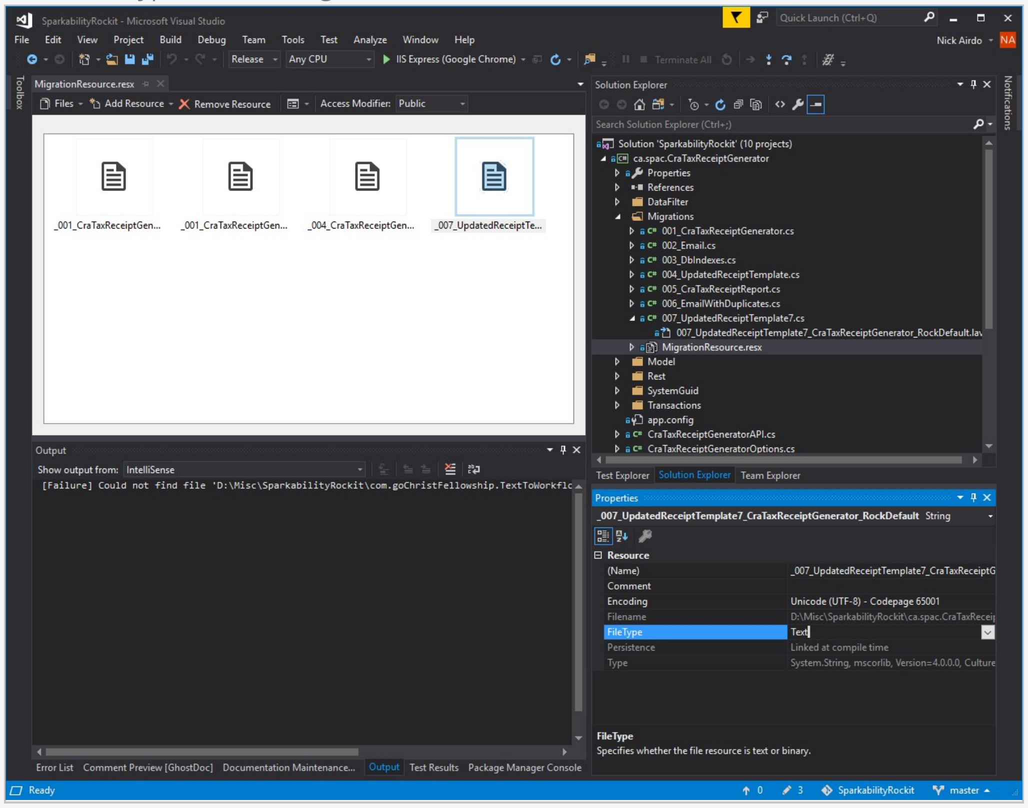
Your New Resource File


Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'SparkabilityRockit' (10 projects)
ca.spac.CraTaxReceiptGenerator
Properties
References
DataFilter
Migrations
001_CraTaxReceiptGenerator.cs
002_Email.cs
003_DbIndexes.cs
004_UpdatedReceiptTemplate.cs
005_CraTaxReceiptReport.cs
006_EmailWithDuplicates.cs
007_UpdatedReceiptTemplate7.cs
MigrationResource.resx
Model
Rest
SystemGuid
Transactions

Next, just put your raw SQL or Lava file next to the migration you want to include it in. We recommend you prefix it with a name that matches the first part of your migration name. So if your my migration was called `007_UpdatedReceiptTemplate7`, name the SQL/Lava file `007_UpdatedReceiptTemplate7_MyLavaFile.lava`.

Now, double-click to open your MigrationResource.resx file and drag your `007_UpdatedReceiptTemplate7_MyLavaFile.lava` into the resx. If Visual Studio doesn't know what type of file the resource is, it may default to a byte[] array. If the file is actually just text (such as a .lava file) then change the FileType from byte[] to Text.

Check FileType after adding a new resource



```
[Failure] Could not find file 'D:\Misc\SparkabilityRockit\com.goChristFellowship.TextToWorkflc
```

Properties: _007_UpdatedReceiptTemplate7_CraTaxReceiptGenerator_RockDefault String

Resource	
(Name)	_007_UpdatedReceiptTemplate7_CraTaxReceiptG
Comment	
Encoding	Unicode (UTF-8) - Codepage 65001
Filename	D:\Misc\SparkabilityRockit\ca.spac.CraTaxReceip
FileType	Text
Persistence	Linked at compile time
Type	System.String, mscorlib, Version=4.0.0.0, Culture

FileType
Specifies whether the file resource is text or binary.

That's everything you need to know to get started. Now you can use the details in the Packaging Plugins & Themes to start sharing all your amazing plugins in the RockShop.

Data Migration Helper Methods

Creating data migrations can be a bit of a chore, but we created some migration helper methods to give you super powers. These methods will help you to add/register new blocks, add pages, add blocks to pages, add groups, add defined types, etc. So instead of writing something like this:

```
Sql( string.Format( @"
DECLARE @FieldTypeId int
SET @FieldTypeId = (SELECT [Id] FROM [FieldType] WHERE [Guid] = '{1}')
DELETE [Attribute]
WHERE [EntityTypeId] IS NULL
AND [Key] = '{2}'
AND [EntityTypeQualifierColumn] = '{8}'
AND [EntityTypeQualifierValue] = '{9}'
INSERT INTO [Attribute] (
    [IsSystem],[FieldTypeId],[EntityTypeId],
    [EntityTypeQualifierColumn],[EntityTypeQualifierValue],
    [Key],[Name],[Description],
    [Order],[IsGridColumn],[DefaultValue],[IsMultiValue],
    [IsRequired],[Guid])
VALUES(
    1, @FieldTypeId, NULL,
    '{8}', '{9}',
    '{2}', '{3}', '{4}',
    {5}, 0, '{6}', 0,
    0, '{7}')
    ",
    "",
    "C28C7BF3-A552-4D77-9408-DEDCF760CED0",
    "Safe Sender Domains".Replace( " ", string.Empty ),
    "Safe Sender Domains",
    "Delimited list of domains that can be used to send emails. It's simple.".Replace( "'",
 "''" ),
    0,
    "",
    "CDD29C51-5D33-435F-96AB-2C06BA772F88",
    "",
    "")
);
```

You can just write it like this:

```
RockMigrationHelper.AddGlobalAttribute( "C28C7BF3-A552-4D77-9408-DEDCF760CED0", "", "", "Sa
fe Sender Domains", "Delimited list of domains that can be used to send emails. It's simple
.", 0, "", "CDD29C51-5D33-435F-96AB-2C06BA772F88" );
```

Helper Method Reference

We've listed all the helper methods for you by entity type for your review.

ActionTypeAttributeValue

- **AddActionTypeAttributeValue(string actionTypeGuid, string attributeGuid, string value)**
  Adds the action type attribute value.

ActionTypePersonAttributeValue

- **AddActionTypePersonAttributeValue(string actionTypeGuid, string attributeGuid, string value)**
  Adds an action type person attribute value. Because there's not a way to link to another person in the target database, person attribute values are just set to the first person alias record in the target database which will most likely be the Admin, Admin record.

Attribute

- **DeleteAttribute(string guid)**
  Deletes the attribute.

AttributeQualifier

- **AddAttributeQualifier(string attributeGuid, string key, string value, string guid)**
  Adds the attribute qualifier.

AttributeValue

- **AddAttributeValue(string attributeGuid, int entityId, string value, string guid)**
  Adds a new attribute value for the given attributeGuid if it does not already exist.

BinaryFileType

- **UpdateBinaryFileType(string storageEntityTypeId, string name, string description, string iconCssClass, string guid, bool allowCaching, bool requiresViewSecurity)**
  Updates the type of the binary file.

Block

- **AddBlock(string pageGuid, string layoutGuid, string blockTypeGuid, string name, string zone, string preHtml, string postHtml, int order, string guid)**
  Adds a new Block of the given block type to the given page (optional) and layout (optional), setting its values with the given parameter values. If only the layout is given, edit/configuration authorization will also be inserted into the Auth table for the admin role (GroupId 2).
- **DeleteBlock(string guid)**

Deletes the block and any authorization records that belonged to it.

BlockAttribute

- **DeleteBlockAttribute(string guid)**
  Deletes the block Attribute.

BlockAttributeValue

- **AddBlockAttributeValue(string blockGuid, string attributeGuid, string value, bool appendToExisting)**
  Adds a new block attribute value for the given block guid and attribute guid, deleting any previously existing attribute value first.
- **DeleteBlockAttributeValue(string blockGuid, string attributeGuid)**
  Deletes the block attribute value.

BlockType

- **AddBlockType(string name, string description, string path, string category, string guid)**
  Adds a new BlockType.
- **UpdateBlockType(string name, string description, string path, string category, string guid)**
  Updates the BlockType by path (if it exists); otherwise it inserts a new record. In either case it will be marked IsSystem.
- **DeleteBlockType(string guid)**
  Deletes the BlockType.

BlockTypeAttribute

- **AddBlockTypeAttribute(string blockTypeGuid, string fieldTypeGuid, string name, string key, string category, string description, int order, string defaultValue, string guid, bool isRequired)**
  Adds a new BlockType Attribute for the given blocktype and key.
- **UpdateBlockTypeAttribute(string blockTypeGuid, string fieldTypeGuid, string name, string key, string category, string description, int order, string defaultValue, string guid)**
  Updates the BlockType Attribute for the given blocktype and key (if it exists); otherwise it inserts a new record.

Category

- **UpdateCategory(string entityTypeGuid, string name, string iconCssClass, string description, string guid, int order)**
  Updates the category.
- **DeleteCategory(string guid)**
  Deletes the category.

DefinedType

- **AddDefinedType(string category, string name, string description, string guid, string helpText)**
  Adds a new DefinedType.
- **DeleteDefinedType(string guid)**
  Deletes the DefinedType.

DefinedTypeAttribute

- **AddDefinedTypeAttribute(string definedTypeGuid, string fieldTypeGuid, string name, string key, string description, int order, string defaultValue, string guid)**
  Adds the defined type attribute.

DefinedValue

- **AddDefinedValue(string definedTypeGuid, string value, string description, string guid, bool isSystem)**
  Adds a new DefinedValue for the given DefinedType.
- **UpdateDefinedValue(string definedTypeGuid, string value, string description, string guid, bool isSystem)**
  Updates (or Adds) the defined value for the given DefinedType.
- **DeleteDefinedValue(string guid)**
  Deletes the DefinedValue.

DefinedValueAttributeValue

- **AddDefinedValueAttributeValue(string definedValueGuid, string attributeGuid, string value)**
  Adds the defined value attribute value.
- **UpdateDefinedValueAttributeValue(string definedValueGuid, string attributeGuid, string value)**
  Adds the defined value attribute value.

DefinedValueAttributeValueByValue

- **AddDefinedValueAttributeValueByValue(string definedTypeGuid, string definedValueValue, string attributeKey, string value)**
  Adds the name of the defined value attribute value by.

DefinedValueByValue

- **UpdateDefinedValueByValue(string definedTypeGuid, string value, string description, int order, bool isSystem)**
  Updates the name of the defined value by.

EntityAttribute

- **AddEntityAttribute(string entityTypeName, string fieldTypeGuid, string entityTypeQualifierColumn, string entityTypeQualifierValue, string name,**

string category, string description, int order, string defaultValue, string guid)
Adds a new EntityType Attribute for the given EntityType, FieldType, and name (key).

- **UpdateEntityAttribute(string entityTypeName, string fieldTypeGuid, string entityTypeQualifierColumn, string entityTypeQualifierValue, string name, string description, int order, string defaultValue, string guid)**
  Updates the Entity Attribute for the given EntityType, FieldType, and name (key). otherwise it inserts a new record.

EntityType

- **UpdateEntityType(string name, string guid, bool isEntity, bool isSecured)**
  Updates the EntityType by name (if it exists); otherwise it inserts a new record.
- **UpdateEntityType(string name, string friendlyName, string assemblyName, bool isEntity, bool isSecured, string guid)**
  Updates the EntityType by name (if it exists); otherwise it inserts a new record.
- **DeleteEntityType(string guid)**
  Deletes the EntityType.

EntityTypeMultiValueFieldType

- **UpdateEntityTypeMultiValueFieldType(string entityTypeName, string fieldTypeGuid)**
  Updates the EntityType MultiValueFieldType

EntityTypeSingleValueFieldType

- **UpdateEntityTypeSingleValueFieldType(string entityTypeName, string fieldTypeGuid)**
  Updates the EntityType SingleValueFieldType

FieldType

- **UpdateFieldType(string name, string description, string assembly, string className, string guid, bool IsSystem)**
  Updates the FieldType by assembly and className (if it exists); otherwise it inserts a new record.
- **DeleteFieldType(string guid)**
  Deletes the FieldType.

GlobalAttribute

- **AddGlobalAttribute(string fieldTypeGuid, string entityTypeQualifierColumn, string entityTypeQualifierValue, string name, string description, int order, string defaultValue, string guid)**
  Adds a global Attribute for the given FieldType, entityTypeQualifierColumn, entityTypeQualifierValue and name (key). Note: This method delets the Attribute first if it had already existed.

Group

- **DeleteGroup(string guid, bool orphanAnyChildren)**
  Deletes the group.

GroupMemberAttributeDefinedValue

- **AddGroupMemberAttributeDefinedValue(string groupGuid, string name, string description, int order, string defaultValue, bool isGridColumn, bool isMultiValue, bool isRequired, string definedTypeGuid, string guid, bool isSystem)**
  Adds or updates a group member Attribute for the given group for storing a particular defined value. The defined values are constrained by the given defined type.
- **UpdateGroupMemberAttributeDefinedValue(string groupGuid, string name, string description, int order, string defaultValue, bool isGridColumn, bool isMultiValue, bool isRequired, string definedTypeGuid, string guid, bool isSystem)**
  Adds or updates a group member Attribute for the given group for storing a particular defined value. The defined values are constrained by the given defined type.

GroupType

- **DeleteGroupType(string guid)**
  Deletes the GroupType.

GroupTypeGroupAttribute

- **AddGroupTypeGroupAttribute(string groupTypeGuid, string fieldTypeGuid, string name, string description, int order, string defaultValue, string guid)**
  Adds a new GroupType "Group Attribute" for the given GroupType using the given values.

GroupTypeRole

- **AddGroupTypeRole(string groupTypeGuid, string name, string description, int order, int? maxCount, int? minCount, string guid, bool isSystem, bool isLeader, bool isDefaultGroupTypeRole)**
  Adds or Updates the GroupTypeRole for the given guid (if it exists); otherwise it inserts a new record. Can also set the role as the default for the given GroupType if isDefaultGroupTypeRole is set to true.
- **UpdateGroupTypeRole(string groupTypeGuid, string name, string description, int order, int? maxCount, int? minCount, string guid, bool isSystem, bool isLeader, bool isDefaultGroupTypeRole)**
  Adds or Updates the GroupTypeRole for the given guid (if it exists); otherwise it inserts a new record. Can also set the role as the default for the given GroupType if isDefaultGroupTypeRole is set to true.
- **DeleteGroupTypeRole(string guid)**

Deletes the GroupTypeRole.

HtmlContentBlock

- **UpdateHtmlContentBlock(string blockGuid, string htmlContent, string guid)**
  Add or Updates the HTML content for an HTML Content Block

Layout

- **AddLayout(string siteGuid, string fileName, string name, string description, string guid)**
  Adds a new Layout to the given site.
- **DeleteLayout(string guid)**
  Deletes the Layout.

Page

- **AddPage(string parentPageGuid, string layoutGuid, string name, string description, string guid, string iconCssClass, string insertAfterPageGuid)**
  Adds a new Page to the given parent page. The new page will be ordered as last child of the parent page.
- **MovePage(string pageGuid, string parentPageGuid)**
  Moves the Page to the new given parent page.
- **DeletePage(string guid)**
  Deletes the Page and any PageViews that use the page.

PageContext

- **AddPageContext(string pageGuid, string entity, string idParameter)**
- **UpdatePageContext(string pageGuid, string entity, string idParameter, string guid)**
  Adds or Updates PageContext to the given page, entity, idParameter
- **DeletePageContext(string guid)**
  Deletes the page context.

PageRoute

- **AddPageRoute(string pageGuid, string route)**
  Adds a new PageRoute to the given page but only if the given route name does not exist.

PersonAttribute

- **UpdatePersonAttribute(string fieldTypeGuid, string categoryGuid, string name, string key, string iconCssClass, string description, int order, string defaultValue, string guid)**
  Updates the BlockType Attribute for the given blocktype and key (if it exists); otherwise it inserts a new record.

PersonAttributeCategory

- **UpdatePersonAttributeCategory(string name, string iconCssClass, string description, string guid, int order)**
  Updates the person attribute category.

PersonBadge

- **UpdatePersonBadge(string name, string description, string entityTypeName, int order, string guid)**
  Updates the PersonBadge by Guid (if it exists); otherwise it inserts a new record.

PersonBadgeAttribute

- **AddPersonBadgeAttribute(string personBadgeGuid, string fieldTypeGuid, string name, string key, string description, int order, string defaultValue, string guid)**
  Adds (or Deletes and Adds) the person badge attribute.

PersonBadgeAttributeValue

- **AddPersonBadgeAttributeValue(string personBadgeGuid, string attributeGuid, string value)**
  Adds/Updates the person badge attribute value.

Report

- **AddReport(string categoryGuid, string dataViewGuid, string entityTypeGuid, string name, string description, string guid, int? fetchTop)**
  Adds a report.
- **DeleteReport(string guid)**
  Deletes the report

ReportField

- **DeleteReportField(string guid)**
  Deletes the report field.

RestAction

- **AddRestAction(string controllerName, string controllerClass, string actionMethod, string actionPath)**
  Adds the rest action.

RestController

- **AddRestController(string controllerName, string controllerClass)**
  Adds the rest controller.

SecurityAuth

- **AddSecurityAuth(string entityTypeName, string action, string groupGuid, string authGuid)**
  Adds the security auth record for the given entity type and group.
- **DeleteSecurityAuth(string guid)**
  Deletes the security auth record.

SecurityAuthForAttribute

- **AddSecurityAuthForAttribute(string attributeGuid, int order, string action, bool allow, string groupGuid, int specialRole, string authGuid)**
  Adds the attribute security authentication. Set GroupGuid to null when setting to a special role
- **DeleteSecurityAuthForAttribute(string attributeGuid)**
  Deletes the security authentication for page.

SecurityAuthForBinaryFileType

- **AddSecurityAuthForBinaryFileType(string binaryFileTypeGuid, int order, string action, bool allow, string groupGuid, Rock.Model.SpecialRole specialRole, string authGuid)**
  Adds the binaryfiletype security authentication. Set GroupGuid to null when setting to a special role

SecurityAuthForBlock

- **AddSecurityAuthForBlock(string blockGuid, int order, string action, bool allow, string groupGuid, Rock.Model.SpecialRole specialRole, string authGuid)**
  Adds the page security authentication. Set GroupGuid to null when setting to a special role
- **DeleteSecurityAuthForBlock(string blockGuid)**
  Deletes the security authentication for block.

SecurityAuthForCategory

- **AddSecurityAuthForCategory(string categoryGuid, int order, string action, bool allow, string groupGuid, int specialRole, string authGuid)**
  Adds the category security authentication. Set GroupGuid to null when setting to a special role
- **DeleteSecurityAuthForCategory(string categoryGuid)**
  Deletes the security authentication for category.

SecurityAuthForEntityType

- **AddSecurityAuthForEntityType(string entityTypeName, int order, string action, bool allow, string groupGuid, int specialRole, string authGuid)**
  Adds the security auth record for the given entity type and group.

SecurityAuthForGroupType

- **AddSecurityAuthForGroupType(string groupTypeGuid, int order, string action, bool allow, string groupGuid, Rock.Model.SpecialRole specialRole, string authGuid)**
  Adds the page security authentication. Set GroupGuid to null when setting to a special role
- **DeleteSecurityAuthForGroupType(string groupTypeGuid)**
  Deletes the security authentication for groupType.

SecurityAuthForPage

- **AddSecurityAuthForPage(string pageGuid, int order, string action, bool allow, string groupGuid, int specialRole, string authGuid)**
  Adds the page security authentication. Set GroupGuid to null when setting to a special role
- **DeleteSecurityAuthForPage(string pageGuid)**
  Deletes the security authentication for page.

SecurityAuthForRestAction

- **AddSecurityAuthForRestAction(string restActionMethod, string restActionPath, int order, string action, bool allow, string groupGuid, Rock.Model.SpecialRole specialRole, string authGuid)**
  Adds the security authentication for rest action.

SecurityAuthForRestController

- **AddSecurityAuthForRestController(string restControllerClass, int order, string action, bool allow, string groupGuid, Rock.Model.SpecialRole specialRole, string authGuid)**
  Adds the security authentication for rest controller.

SecurityRoleGroup

- **AddSecurityRoleGroup(string name, string description, string guid)**
  Adds the security role group.
- **DeleteSecurityRoleGroup(string guid)**
  Deletes the security role group.

Site

- **AddSite(string name, string description, string theme, string guid)**
  Adds a new Layout to the given site.
- **DeleteSite(string guid)**
  Deletes the Layout.

SystemEmail

- **DeleteSystemEmail(string guid)**
  Deletes the SystemEmail.

WorkflowActionEntityAttribute

- **UpdateWorkflowActionEntityAttribute(string actionEntityTypeGuid, string fieldTypeGuid, string name, string key, string description, int order, string defaultValue, string guid)**
  Updates the workflow action entity attribute.

WorkflowActivityTypeAttribute

- **UpdateWorkflowActivityTypeAttribute(string workflowActivityTypeGuid, string fieldTypeGuid, string name, string key, string description, int order, string defaultValue, string guid)**
  Updates the workflow activity type attribute.

WorkflowTypeAttribute

- **UpdateWorkflowTypeAttribute(string workflowTypeGuid, string fieldTypeGuid, string name, string key, string description, int order, string defaultValue, string guid)**
  Updates the workflow type attribute.

> Gotcha!
> When creating your SQL migrations be sure to watch out for the common gotchas in the next section.

Double The Quotes

Be sure to double any quotes that you have within your own SQL as seen here:

```
Sql( @"
  UPDATE
     [Attribute]
  SET
     [Description] = 'He said ""Rock"" is fun. Don''t you agree?'
  WHERE
     [Guid] = 'ABCDEFG9-1111-2222-3333-1213456789ABC'
" );
```

Don't Quote Your Nulls

Just pass the null keyword as seen below:

```
RockMigrationHelper.UpdateGroupTypeRole( "E0C5A0E2-B7B3-4EF4-820D-BBF7F9A374EF", "Facebook
Friend", "A Facebook friend.",
0, null, null, "AB69816C-4DFA-4A7A-86A5-9BFCBA6FED1E" );
```

Migration Generation Tools

You may find yourself creating a new page with child pages that use your new blocks, or your block may use a new custom workflow that you need to distribute with your package. Depending on the situation, creating a migration by hand can be a daunting task. We've felt that pain too and created a few more helper tools.

If you look in the *Rock\Dev Tools\Sql* folder, you'll notice several sql scripts that start with the prefix `CodeGen*_`. These scripts can help generate many of the needed MigrationHelper methods for your stuff.

For example, when executed, the `CodeGen_PagesBlocksAttributesMigration_ForAPage.sql` script takes the PageId parameter (which you set to the id of your choice):

```
DECLARE @PageId int = 226
```

...and outputs the needed MigrationHelper methods for the Up() and Down() methods of your migration.

```
// MigrationUp
// -----------
// Page: Layout Detail
RockMigrationHelper.AddPage("A2991117-0B85-4209-9008-254929C6E00F","D65F783D-87A9-4CC9-8110
-E83466A0EADB","Layout Detail","","E6217A2B-B16F-4E84-BF67-795CA7F5F9AA","fa fa-th"); // Si
te:Rock RMS
RockMigrationHelper.UpdateBlockType("Layout Detail","Displays the details for a specific la
yout.","~/Blocks/Cms/LayoutDetail.ascx","CMS","68B9D63D-D714-473A-89F2-62EB1602E00A");
RockMigrationHelper.UpdateBlockType("Layout Block List","Lists blocks that are on a given s
ite layout.","~/Blocks/Cms/LayoutBlockList.ascx","CMS","CD3C0C1D-2171-4FCC-B840-FC6E6F72EEE
F");
RockMigrationHelper.AddBlock("E6217A2B-B16F-4E84-BF67-795CA7F5F9AA","","68B9D63D-D714-473A-
89F2-62EB1602E00A","Layout Detail","Main","","",0,"C04C6905-C156-49D3-832D-D09F3B0E1BF1");

RockMigrationHelper.AddBlock("E6217A2B-B16F-4E84-BF67-795CA7F5F9AA","","CD3C0C1D-2171-4FCC-
B840-FC6E6F72EEEF","Layout Block List","Main","","",1,"5FB1CC3B-4550-4099-8C83-044FF57CEAD8
");

// MigrationDown
// -------------
RockMigrationHelper.DeleteBlock("5FB1CC3B-4550-4099-8C83-044FF57CEAD8");
RockMigrationHelper.DeleteBlock("C04C6905-C156-49D3-832D-D09F3B0E1BF1");
RockMigrationHelper.DeleteBlockType("CD3C0C1D-2171-4FCC-B840-FC6E6F72EEEF");
RockMigrationHelper.DeleteBlockType("68B9D63D-D714-473A-89F2-62EB1602E00A");
RockMigrationHelper.DeletePage("E6217A2B-B16F-4E84-BF67-795CA7F5F9AA"); //  Page: Layout De
tail
```

Caution!

Don't forget to really look closely at the code these scripts generate. It's always a good idea to verify that it did not include any extra bits, pages, items, etc.

You may need to experiment with each one to become familiar with how it works, but they all work similarly. Scripts like `CodeGen_WorkflowTypeMigration.sql` can save you tons of hours, but you need to know it works a little differently. That script outputs all Workflow related records *except* the ones defined in its `#knownGuidsToIgnore` table.