How to make a Custom Rock Library Plugin

# Summary

This is will show how to make a simple Custom Rock Library Plugin that can be used to extend the functionality of the core Rock system, without needing to change any code in the core.

# Prerequisites

* Proficiency in C# and a familiarity with Entity Framework.
* An installation of Rock, either from the Rock Installer or by building the Rock core in your development environment (see <https://github.com/SparkDevNetwork/Rock-ChMS/wiki/Installing-rock-chms>)
* A version of Rock with Entity Framework 6.0.
* The Rock Requirements (see <https://github.com/SparkDevNetwork/Rock-ChMS/wiki/Rock-Requirements>)

# Special Install Instructions

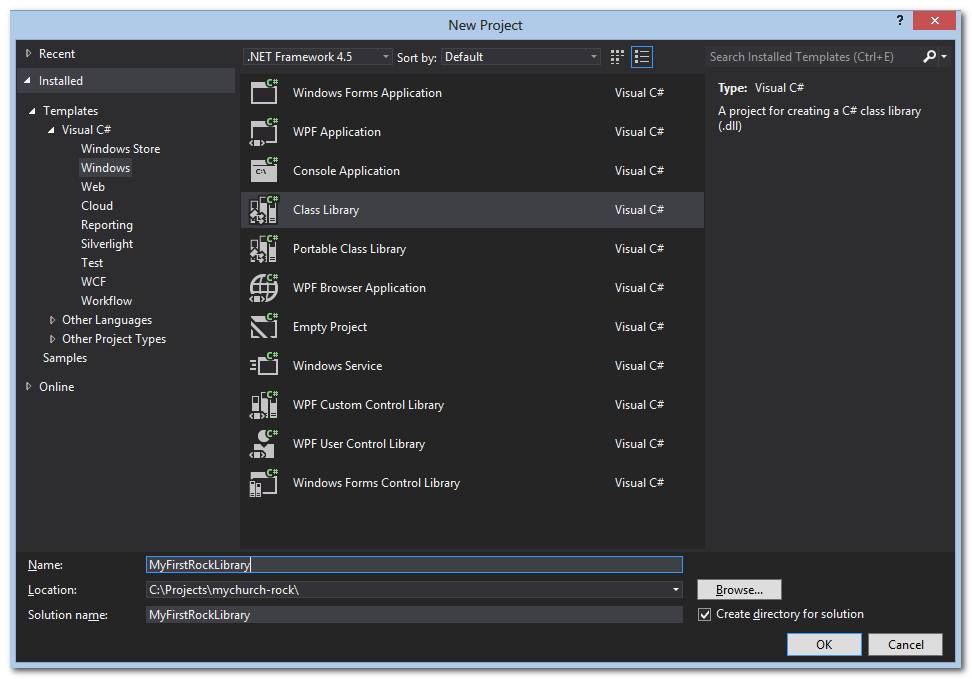
* Develop your plugin in your developer database environment, not your production database. For example, do the following
  + Create a folder for your development environment, for example c:\projects\Rock-ChMS\. Install Rock there.
  + Edit web.ConnectionStrings.config (in the root of the RockWeb folder) (see <https://github.com/SparkDevNetwork/Rock-ChMS/wiki/Installing-rock-chms> if you don’t have one yet)
    - Change the connectionString so that Initial Catalog is set to RockChMS\_developer. You may also want to change Data Source to localhost.
  + After changing the connection string, you’ll need to start Rock to create the core Rock database. NOTE: If you are doing this in Visual Studio, you should restart Visual Studio after changing the connection string to ensure you get a clean connection to the new database.
* Make sure that your RockWeb’s webconfig has AutoMigrateDatabase set to True
* Now, start Rock, which will create your core Rock Database and update it to the current Rock version. (see instructions in <https://github.com/SparkDevNetwork/Rock-ChMS/wiki/Installing-rock-chms>)

# Prepare your Development Environment

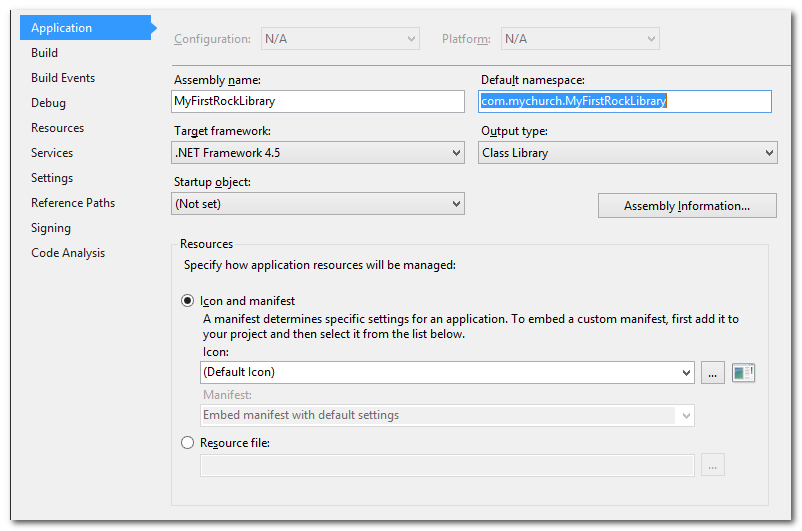
* Create a new folder for your custom Solution.
  + Optional: Now would also be a good time to implement some sort of Source Control for your code. For example, you could use <http://github.com>, create an account and make a repository for your project.
* Name your folder using the Rock naming conventions (see <https://github.com/SparkDevNetwork/Rock-ChMS/wiki/Naming-Conventions>)
  + For example
    - c:\projects\mychurch-rock\ as the solution folder of your custom project(s)
    - c:\projects\mychurch-rock\MyCustomProject\ as the folder for your Custom Rock Library Plugin project
* Copy and Paste the RockWeb folder from your c:\projects\Rock-ChMS\ (or production environment) into your c:\projects\mychurch-rock\ folder. This is helpful to make sure you don’t impact your production site until you are ready to deploy. This will also make it much easier to manage your database connection string.
* Now, again, make sure that the web.ConnectionStrings.config in this folder (C:\Projects\mychurch-rock\RockWeb) is configured to connect to your development database, not your production environment (Data Source=localhost;Initial Catalog=RockChMS\_developer;...)

# Create new Visual Studio Solution

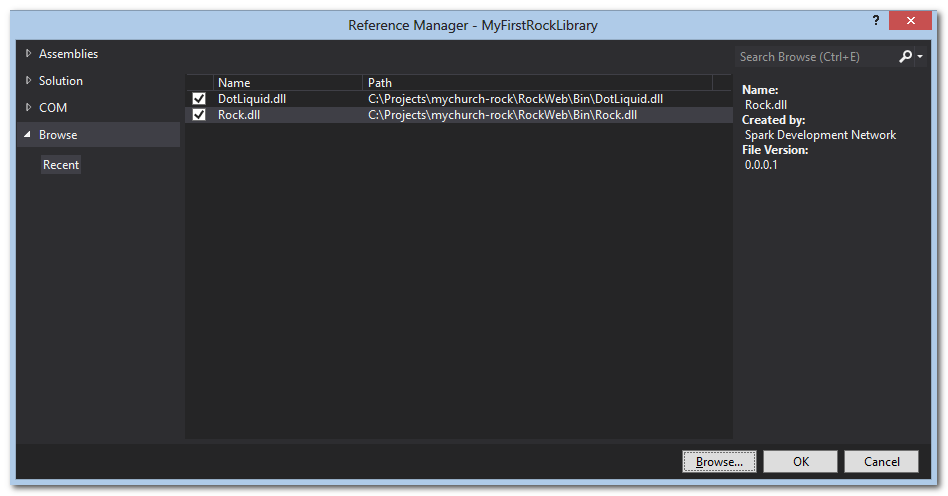
* Using Visual Studio, create a new Class Library project
  + Choose C:\Projects\mychurch-rock\ as the Location, and name your Library



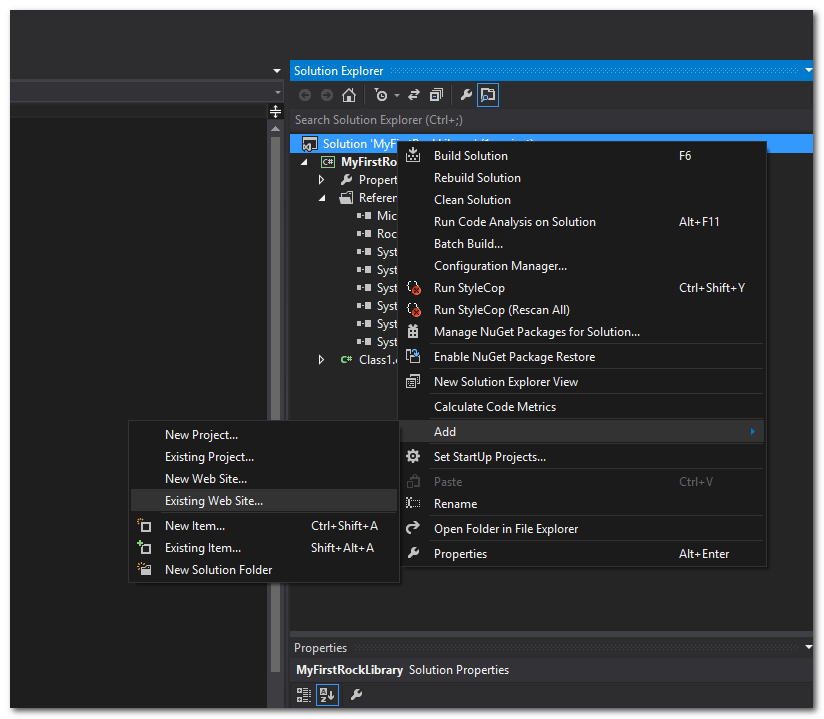
* Set your default Namespace using the Rock naming conventions (see <https://github.com/SparkDevNetwork/Rock-ChMS/wiki/Naming-Conventions>)
  + For example, com.mychurch.MyFirstRockLibrary

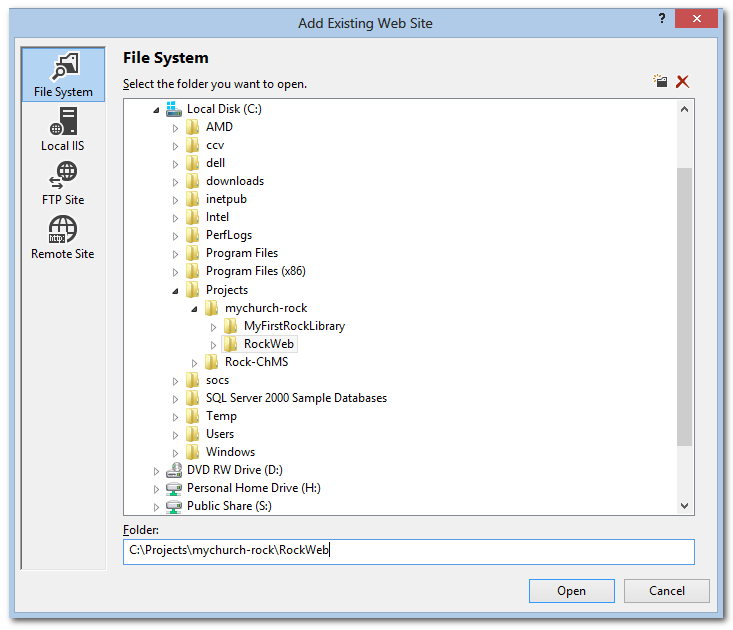


* Add Rock.Dll and DotLiquid.dll to your project’s References
  + Get Rock.Dll from C:\Projects\mychurch-rock\RockWeb\Bin
  + DotLiquid.dll is a 3rd party library that Rock.Dll requires. It is also in C:\Projects\mychurch-rock\RockWeb\Bin

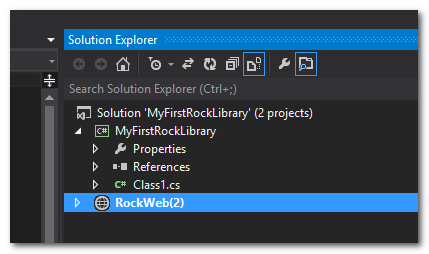


* Add Existing Website, C:\Projects\mychurch-rock\RockWeb, to your Solution

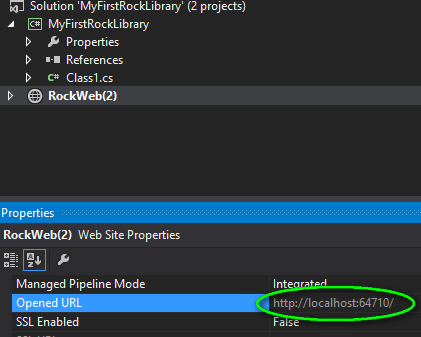


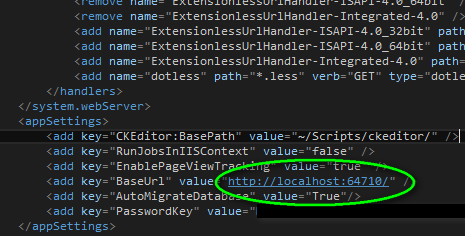


* Enable IIS-Express for the RockWeb project
  + In your Solution tree, Right Click on c:\...\RockWeb, then select “Use IIS Express…”
    - Click Yes on the “Do you want to…” prompt
* Right-Click again on the RockWeb project and select “Set as Startup Project”
* Now, your Solution tree should look like this



* + Note: Your RockWeb might just be called “RockWeb”, or have a different number in parenthesis. This is just IIS Express making sure there is a uniquely named entry for RockWeb depending on how many other RockWeb projects you have.
* Edit RockWeb’s web.config again, this time, edit BaseUrl to match the RockWeb project’s “Opened URL”. This would also be a good time to double-check that AutoMigrationDatabase is True.





# Start Rock

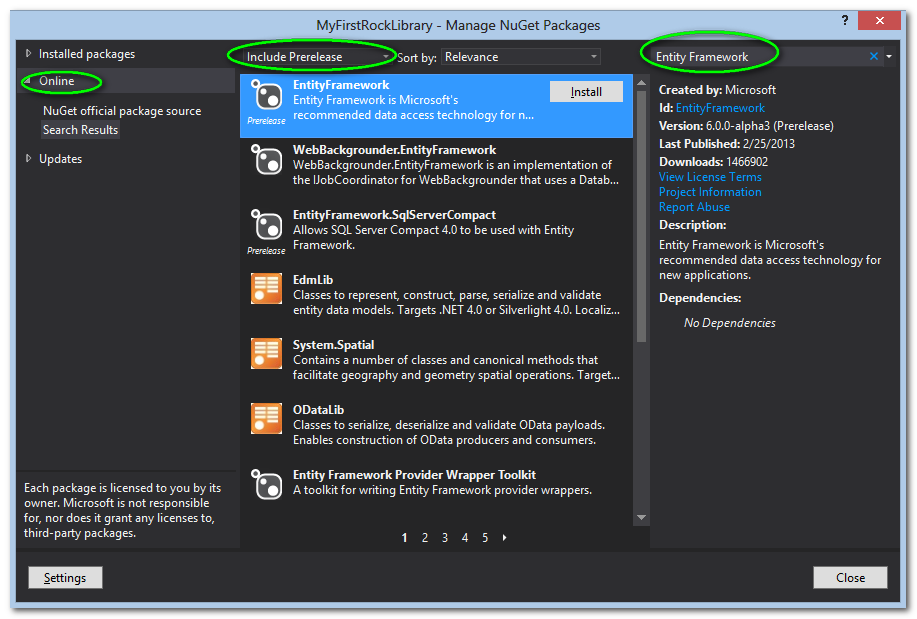
* Now, to make sure we have everything ready, start Rock. (Press F5)
* This might take up to a minute or so the first time. It is creating and migrating the database.
* If you get errors, the most common reasons are the WebConfig settings. Go back a step or so in these directions and make sure those are correct.

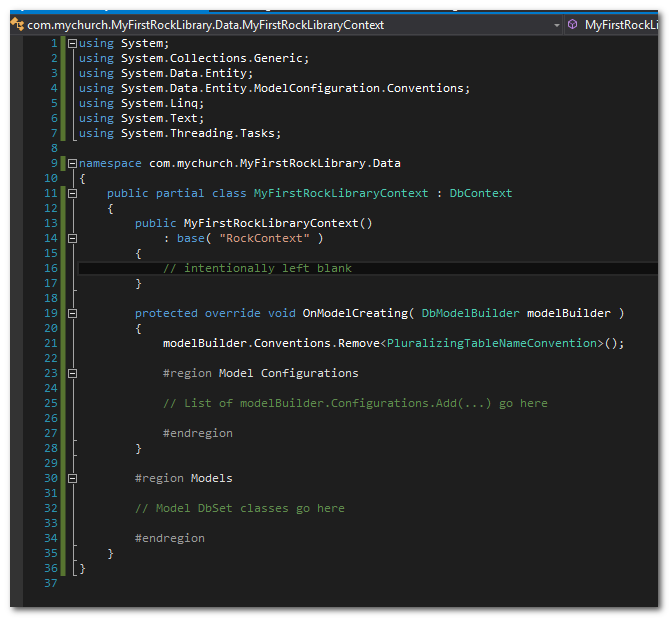
# Breathe a Sigh of Relief

* If you’ve made it to this point, now would be a good time to your C:\Projects\mychurch-rock into Source Control (or at least zip it up and save it somewhere). This is a good point to go back to in case you want to start over.

# Setup your Custom Rock Library project for Entity Framework

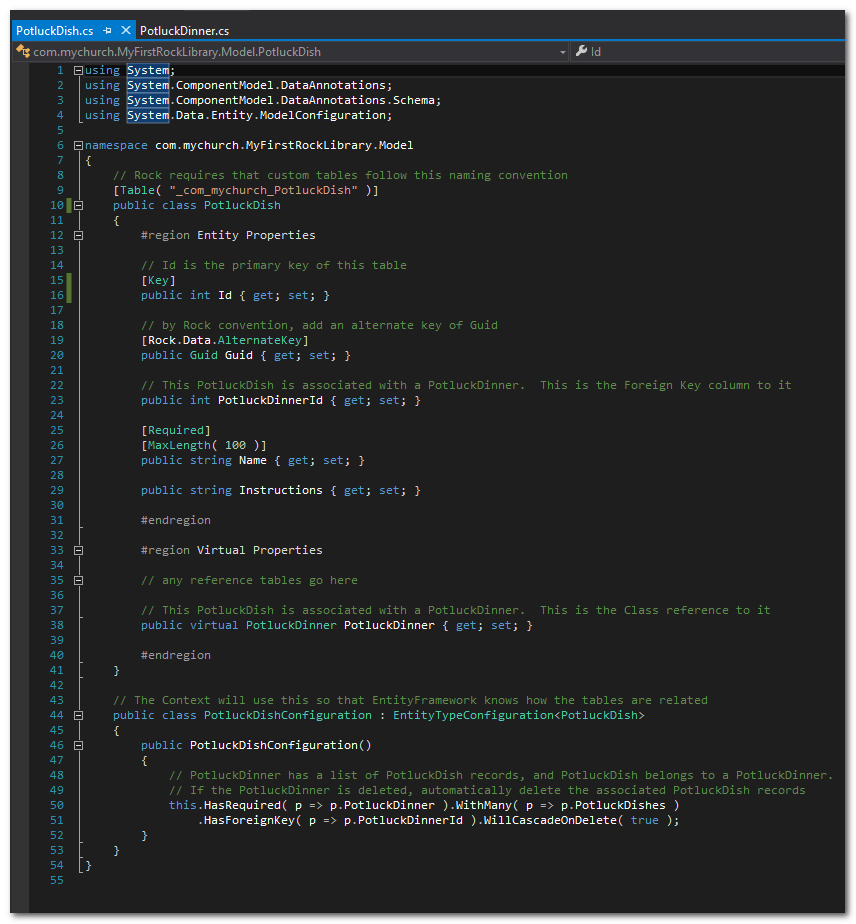
* In the Solution tree, Right-Click on your Custom Rock Library and choose “Manage NuGet Packages…”
  + Search for EntityFramework 6.0 and Install it



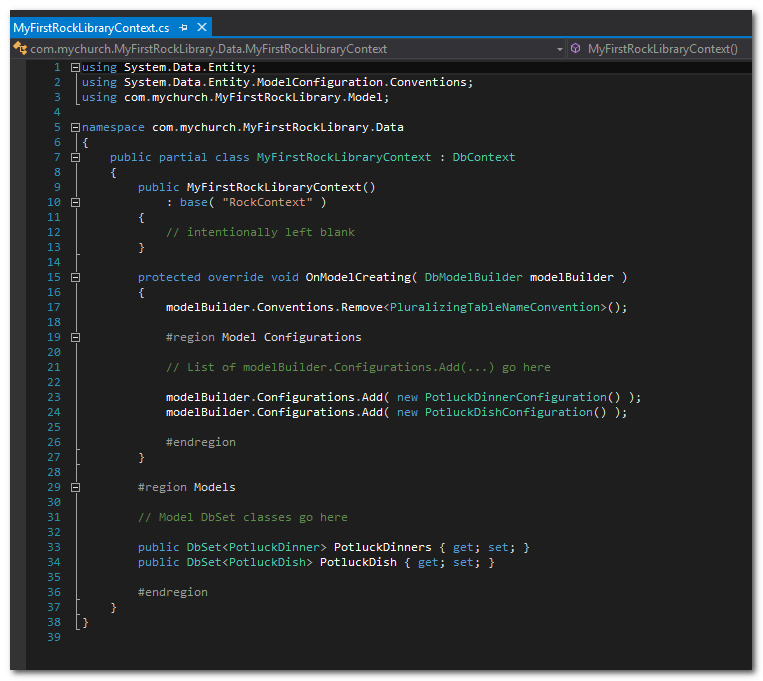
* Add two folders to your Project
  + Data
  + Model
* Create a Context class in the Data folder named <YourProjectName>Context
* This class should inherit from DbContext. Add the methods, regions, comments, etc as shown in this example
  + 
  + Take special note of the constructor. See how it uses the base constructor with a parameter of “RockContext”. That’s important. Your context class should do the exact same thing. This lets Entity Framework know that your connection string name is “RockContext”, and we want to use the exact same connection name as RockWeb.
* Create your first model classes
  + Note: Rock uses the Code-First convention of Entity Framework, a model class is what ends up as a table in your database.
  + Let’s say you want Custom Rock Plugin to add a Potluck feature, but you need a couple of tables to store your data
    - Create a class called PotluckDinner along with a PotluckDinnerConfiguration class as shown. Don’t worry if PotluckDish can’t be found. We’ll add that next.



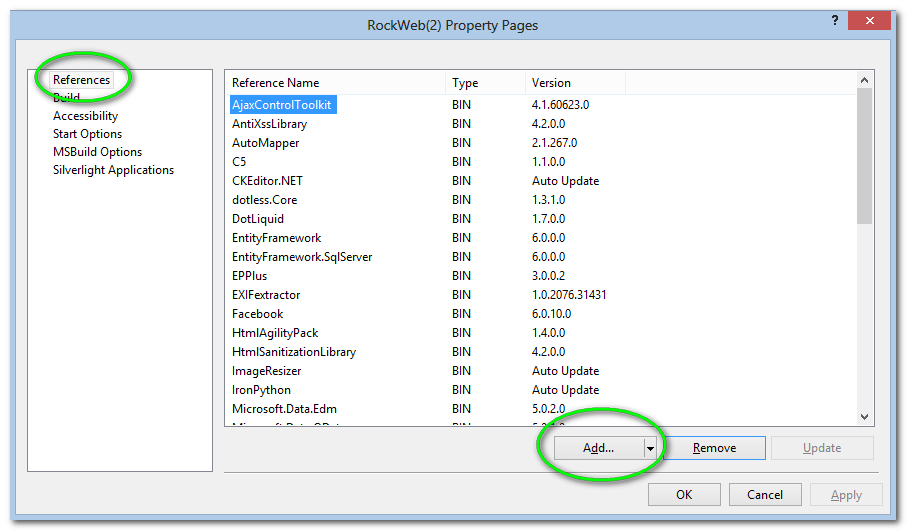
* + - Next, create the PotluckDish class as shown



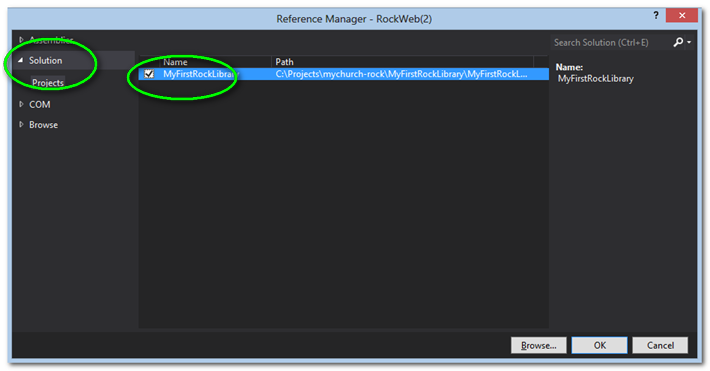
* + Now that we have a good start on our tables, let’s hook them up to our Context. Add the Model DbSet classes and Configuration classes to MyFirstRockLibraryContext.cs as shown



* + Now, we are almost ready to get our new tables into the database, but first, make sure you can Build the solution.
  + After a successful build, you need to add your project’s dll to RockWeb
  + In the Solution tree, right-click on RockWeb and select Properties
    - Now add your Project’s dll to RockWeb’s References

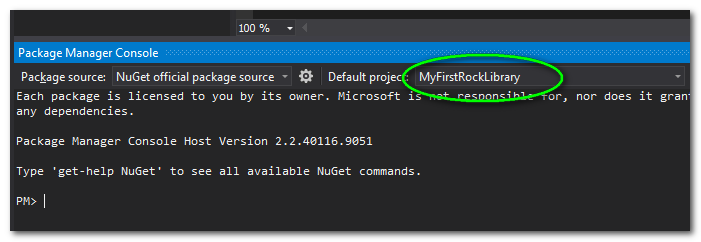


* + - In the Add dialog, select Solution, then select your Project, then Press OK a couple of times to get back to the main solution

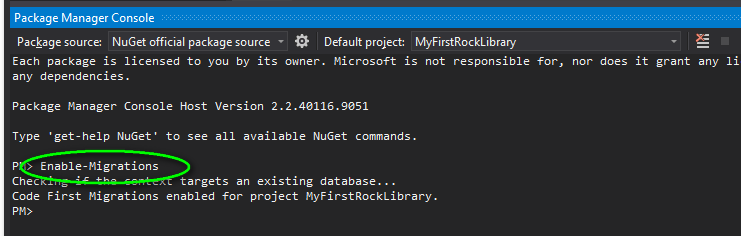


# Setup Entity Framework Migrations for your project

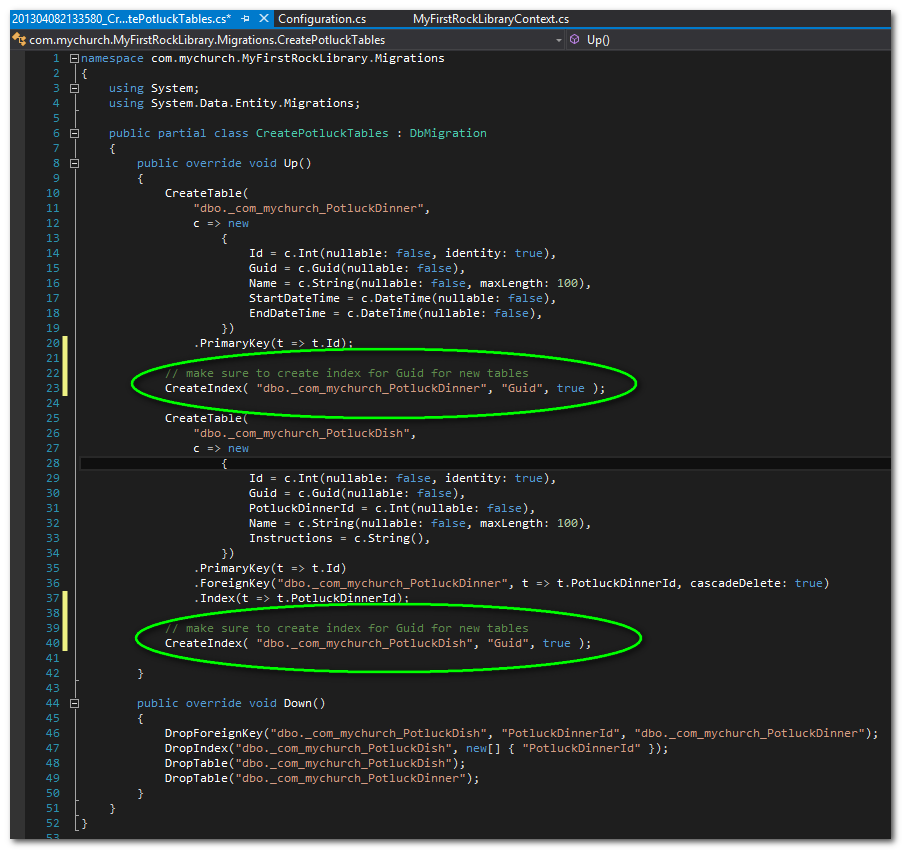
* Open Visual Studio’s Package Manager Console (View | Other Windows | Package Manager Console)
* In the Package Manager Console window, change the Default project to your project’s name



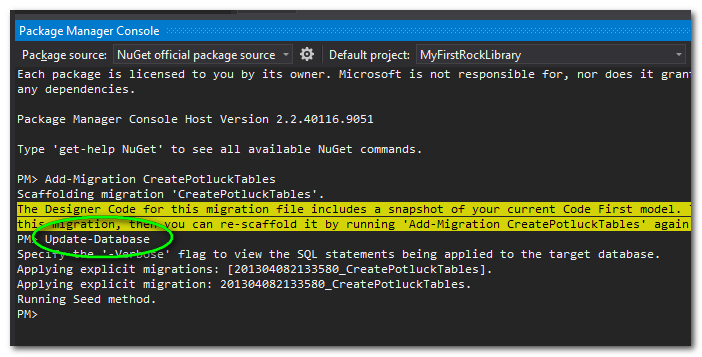
* Run “Enable-Migrations”

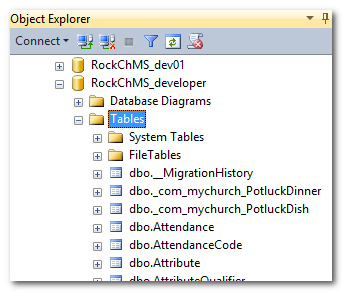


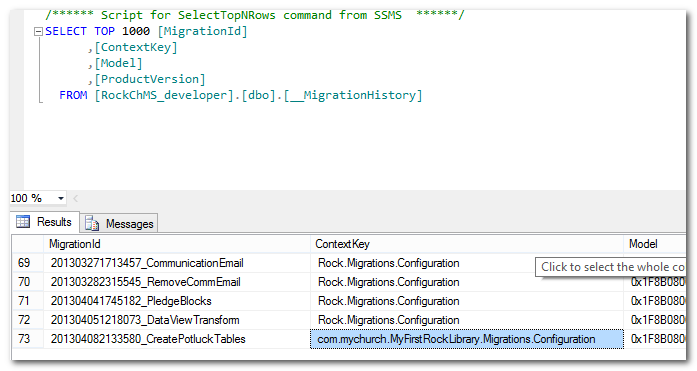
* If this worked, a couple of things should happen
  + The Package Manager Console should say that Code First Migrations are enabled for your project
  + A “Migrations” folder appears in your Project with a Configuration class
* Now you can create your first Migration
  + Note: The syntax for adding a migration is “Add-Migration <migration name>”.
  + In the Package Manager Console, run “Add-Migration CreatePotluckTables”
  + This creates a DbMigration class in the Configurations folder called CreatePotluckTables
    - Note that this creates two methods called Up() and Down(), and gives you starter code that Entity Framework generated from your Context and Model classes
    - Sometimes, the code generated for Up() and Down() is all you need, but usually you’ll have to add some additional code
      * In this case, we wanted our Guid columns to have a Unique Index, since those are our Alternate Keys, so we’ll have to add that manually
    - When you are done, the migration will be the code generated by Entity Framework along with a couple of lines of code that we have to add manually.



* Update your database to be in sync with your latest Migration
  + First, make sure you do a Save-All for your solution so that the Migration class will get saved.
  + In the Package Manager Console, run “Update-Database”



* + This will make an entry in EntityFramework’s \_\_MigrationHistory table in your database, and run the migration which created the tables, foreign keys, etc for you.
  + Now, use Sql Server Management Studio to see what happened.
    - Navigate to RockChMS\_developer, and expand tables to see the \_\_MigrationHistory table, your new tables, followed by Rock’s core tables
    - 
    - If you are curious, you can query the \_\_MigrationHistory table to see how EntityFramework stores the Migration data. Notice how your Migration(s) have a different ContextKey than the core Rock Migrations. This is a new feature that started in Entity Framework 6.0, that allows Rock to have multiple migrations coming from multiple plugins (Multi-Tenant Migrations).

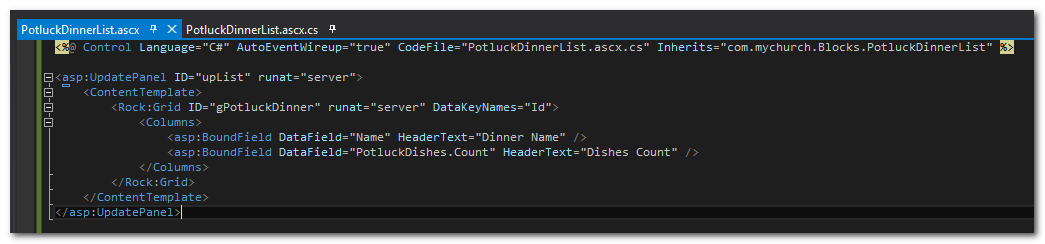


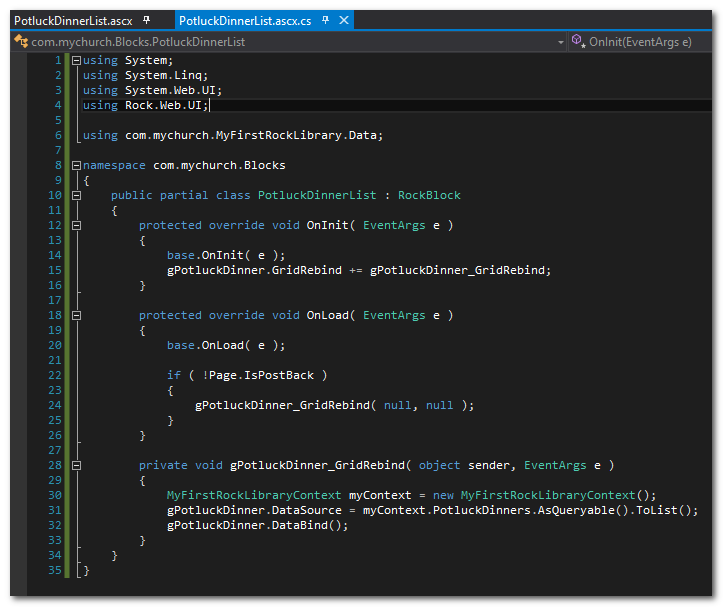
# Using your new Library in new Rock Block

Now that we have our Customer Rock Library Plugin ready, we can use it in some new Rock Blocks. To do that, for the basics of creating a Rock Block, first read and understand <https://github.com/SparkDevNetwork/Rock-ChMS/wiki/Building-your-first-custom-block>, and then come back here for specifics on how to use your new library plugin

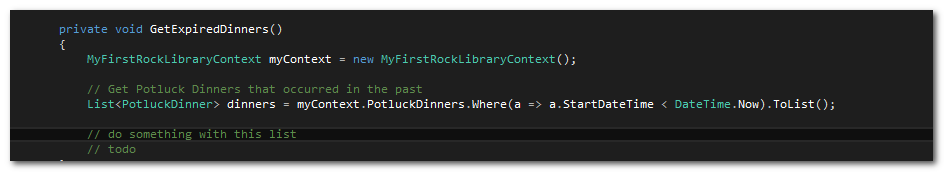
OK, now that you’re familiar with the basics of a Rock Block, let’s use our new Library Plugin with one. The main differences between using built in Rock tables versus our custom tables is that our custom tables come from an instance of our Context (MyFirstRockLibraryContext). To illustrate, here are some examples

Here’s an example of a List page using our new context and PotluckDinner table

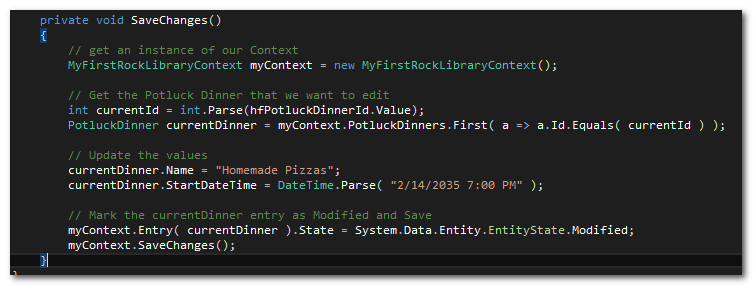




Here is a Linq query against our PotluckDinners table



This is the basic idea on how to save changes to the database using our context



# Conclusion

So, that’s about it! Congratulations, you are now a Rock Plugin Developer.