Writing a LINQPad Data Context Driver

Joseph Albahari
Last updated 2016-4-6

Contents

[Introduction 2](#_Toc443730582)

[Who Is This Guide For? 2](#_Toc443730583)

[Why Write a Data Context Driver? 2](#_Toc443730584)

[How Does it Work, From the User’s Perspective? 3](#_Toc443730585)

[Is Writing a Driver Difficult? 4](#_Toc443730586)

[Are There Any Special Terms and Conditions? 4](#_Toc443730587)

[What Framework Versions are Supported? 5](#_Toc443730588)

[Concepts 5](#_Toc443730589)

[Terminology 5](#_Toc443730590)

[Static vs Dynamic Drivers 5](#_Toc443730591)

[How LINQPad Queries Work 6](#_Toc443730592)

[Autocompletion 6](#_Toc443730593)

[Setting up a Project 6](#_Toc443730594)

[Versioning 7](#_Toc443730595)

[Writing a Driver 8](#_Toc443730596)

[Writing a Static Driver 9](#_Toc443730597)

[Writing a Dynamic Driver 9](#_Toc443730598)

[Application Domains & Loading Assemblies 10](#_Toc443730599)

[Assembly Resolution 11](#_Toc443730600)

[Fine-Tuning 12](#_Toc443730601)

[Passing Arguments into a Data Context’s Constructor 12](#_Toc443730602)

[Working with Databases and Connection Strings 12](#_Toc443730603)

[Performing Additional Initialization / Teardown on the Data Context 13](#_Toc443730604)

[Populating the SQL Translation Tab 13](#_Toc443730605)

[Importing Additional Assemblies and Namespaces 14](#_Toc443730606)

[Overriding AreRepositoriesEquivalent 15](#_Toc443730607)

[Overriding GetLastSchemaUpdate (dynamic drivers) 15](#_Toc443730608)

[Supporting SQL Queries 16](#_Toc443730609)

[Clearing Connection Pools 16](#_Toc443730610)

[Application Configuration Files 16](#_Toc443730611)

[Customizing the Icon 17](#_Toc443730612)

[Custom Features 17](#_Toc443730613)

[Customizing Output 17](#_Toc443730614)

[Overriding PreprocessObjectToWrite 17](#_Toc443730615)

[Implementing ICustomMemberProvider 18](#_Toc443730616)

[Customizing Output to Data Grids 20](#_Toc443730617)

[Troubleshooting 21](#_Toc443730618)

[Exception Logging 21](#_Toc443730619)

[Debugging 21](#_Toc443730620)

[API Reference 22](#_Toc443730621)

[IConnectionInfo 22](#_Toc443730622)

[IDatabaseInfo 22](#_Toc443730623)

[ICustomTypeInfo 23](#_Toc443730624)

[ExplorerItem 23](#_Toc443730625)

[ExplorerItemKind 24](#_Toc443730626)

[ExplorerIcon 25](#_Toc443730627)

# Introduction

## Who Is This Guide For?

This guide is for programmers interested in extending LINQPad to support other data sources. In other words, this guide describes how to add new drivers to the following dialog:



## Why Write a Data Context Driver?

Without a custom data context driver, LINQPad can query any data source, but the user must manually reference libraries, import custom namespaces, and formulate queries like this:

 var dataSource = new CustomersData();

 (from c in dataSource.Customers

 where c.Name.StartsWith ("A")

 select new { c.Name, c.Purchases }).Dump();

instead of simply:

 from c in Customers

 where c.Name.StartsWith ("A")

 select new { c.Name, c.Purchases }

A custom data context driver overcomes these problems. Furthermore:

* The types being queried can appear in the Schema Explorer.
* The types being queried can optionally be built on the fly (with a *dynamic* driver) instead of being defined or imported by the user.
* A driver can control how LINQPad renders any type in the output window. This is useful, for instance, when a type has lazily evaluated properties that are expensive to walk.
* A driver can populate the SQL translation tab in the output pane, and make numerous additional tweaks to optimize the querying experience.

## How Does it Work, From the User’s Perspective?

When the user clicks “Add Connection”, they get the following dialog:



If the user clicks “View More Drivers”, the following dialog appears (assuming he or she is online):



Clicking on a library from the gallery (which is, in fact, a web page) downloads a driver from the Internet. Clicking the “Browse” buttons lets users import a *.lpx* file that they’ve downloaded themselves. Once clicked, the driver will become visible in the previous dialog, and the user can begin querying.

## Is Writing a Driver Difficult?

In most cases, writing a driver is easy. The basic steps are as follows:

1. Choose between writing a dynamic or static driver (more on this soon)
2. Write a class library project and reference LINQPad.exe
3. Subclass **DynamicDataContextDriver** or **StaticDataContextDriver**
4. Implement a handful of abstract methods (and optionally, some virtual methods)
5. Zip up your library (and any dependencies) and change the extension from *.zip* to *.lpx*
6. (Optionally) submit your *.lpx* file to the LINQPad Drivers Gallery

The extensibility model has been designed such that it’s quick to write a driver with basic functionality. Be sure to check out the demo project: this comprises two drivers that illustrate most aspects of the process.

The types comprising the extensibility model in LINQPad are not obfuscated. You are encouraged to use ILSpy or Reflector if you want to look deeper into these types.

## Are There Any Special Terms and Conditions?

No—unless you choose to submit your driver to LINQPad’s web gallery (so that it’s visible directly from LINQPad’s “More Drivers” page). In which case:

1. There’s no obligation to accept a particular submission. In general, a driver should be for a popular product and should already have been tested by users.
2. You must provide a support URI (this can be a peer support forum or FAQ page).

## What Framework Versions are Supported?

LINQPad has separate builds to target different .NET Framework versions. In general, if you target a lower framework version, your driver will work with higher versions as well.

# Concepts

## Terminology

A *connection* corresponds to what the user enters when they click ‘Add Connection’. This is broader than the concept of a database connection in that a LINQPad connection can point to other kinds of data sources such a web services URI. Further, a LINQPad connection can include data context-specific details such as pluralization and capitalization options. A LINQPad connection is represented by the **IConnectionInfo** interface.

A *typed* *data context* is a class with properties/fields/methods that the user can query. A classic example is a typed LINQ to SQL DataContext (or a typed ObjectContext in Entity Framework):

 public class TypedDataContext : DataContext

 {

 public IQueryable<Customer> Customers
 { get { return this.GetTable<Customer>(); } }

 public IQueryable<Order> Orders
 { get { return this.GetTable<Orders>(); } }

 }

A typed data context does not need base class. The following is perfectly valid:

 public class TypedDataContext

 {

 public IEnumerable<string> CustomerNames

 public int[] Numbers;
 public void DoSomething() { … }

 }

A typed data context is mandatory if you want to write a LINQPad Data Context Driver. There are two ways to obtain a typed data context:

* Your driver can build one of the fly (*Dynamic* Driver)
* You can consume a typed data context already defined by the user (*Static* Driver)

## Static vs Dynamic Drivers

When you click ‘Add connection’ in LINQPad, a dialog appears with a list of data context drivers from which to choose. These are split into two lists:

“Build Data Context Automatically” — *Dynamic* data context drivers

“Use a typed data context from your own assembly” — *Static* data context drivers

A dynamic driver builds the typed data context on the fly. It does this either by generating code and then compiling it, or by using Reflection.Emit.

A static driver requires that the user supply the typed data context. The connection dialog that you write will prompt the user for the path to a custom assembly containing the typed data context, and the name of the type.

The advantage of a dynamic driver is that it allows the end user to query without having to first write classes in a Visual Studio project. The advantage of a static driver is that it gives users a finer degree of control, and also full compatibility with typed data contexts in projects that they’ve already written.

You can implement both kinds of driver in the same assembly.

## How LINQPad Queries Work

Recall that users write queries in LINQPad without explicitly referring to a data context:

 **from c in Customers**

 **where c.Name.StartsWith ("A")**

 **select new { c.Name, c.Purchases }**

To make this work, LINQPad *subclasses* your typed data context, writing the user’s query into a method as follows:

 public class UserQuery : TypedDataContext

 {

 public UserQuery (*parameters...*) : base (*parameters...*) { }

 void RunUserAuthoredQuery()

 {

 (

 **from c in Customers**

 **where c.Name.StartsWith ("A")**

 **select new { c.Name, c.Purchases }**

 )

 .Dump();

 }

 }

LINQPad then calls the C# or VB compiler service on the class, compiles it into a temporary assembly, instantiates the class, and then calls **RunUserAuthoredQuery**. The same principle holds with both dynamic and static drivers.

It is therefore important that your typed data context class is not sealed and has a public constructor.

We’ll discuss later how to feed parameters into the constructor, in “Fine-Tuning”.

## Autocompletion

Autocompletion feeds entirely on the type system, so no special work is required to support it.

# Setting up a Project

To begin, create a new class library in Visual Studio. Set the project properties as follows:

* Build | Platform target = “Any CPU”
* (Optional) Signing | Sign the Assembly. Strong-name signing is no longer mandatory if your users are running LINQPad 5.06 or higher. A valid reason not to strong-name your assembly is if your driver references other weakly named assemblies.

Then add a reference to LINQPad.exe and subclass **DynamicDataContextDriver** or **StaticDataContextDriver** (or both) as described in the following sections. The easiest way to begin is to copy and paste one of the samples from the demo project.

When you’re done, zip up your target *.dll* (and any dependencies) and add a file called **header.xml** with the following content:

 <?xml version="1.0" encoding="utf-8" ?>

 <DataContextDriver>

 <MainAssembly>*YourAssembly*.dll</MainAssembly>

 <SupportUri>*http://mysite.com*</SupportUri>

 </DataContextDriver>

*YourAssembly.dll* should be name of the assembly containing the drivers. There can be any number of driver classes in this assembly; LINQPad looks for all **public non-abstract** types that are based on **DynamicDataContextDriver** or **StaticDataContextDriver**.

Once you’ve got a zip file, change its extension from *.zip* to *.lpx*. You’ll then be able to import this into LINQPad by clicking ‘Add Connection’, ‘More Drivers’ and ‘Browse’.

When you import a driver into LINQPad, all that LINQPad does is unzip the driver’s contents into the a folder rooted in the following location:

 %localappdata%\LINQPad\Drivers\DataContext\4.6\

This is true for LINQPad 5 (which targets .NET Framework 4.6). For LINQPad 4 (which targets .NET Framework 4.0/4.5), the following directory is used instead:

 **%programdata%**\LINQPad\Drivers\DataContext\**4.0**\

(And for LINQPad 2, it was %programdata%\LINQPad\Drivers\DataContext\3.5\).

The driver itself sits in a subfolder whose name comprises the name of the assembly plus its public key token in parenthesis. For example:

 %localappdata%\LINQPad\Drivers\DataContext\4.6\**MyDriver (ff414cf4a100c74d)**\

which expands out to something like:

 C:\Users\joe\AppData\Local\LINQPad\Drivers\DataContext\4.6\MyDriver (ff414cf4a100c74d)\

If your assembly has no public key (not strongly named), the folder will instead look like this:

 C:\Users\joe\AppData\Local\LINQPad\Drivers\DataContext\4.6\MyDriver (**no-strong-name**)\

After importing your driver into LINQPad, locate this directory. Then, set up a post-build event in Visual Studio to copy your output assemblies to your driver folder: this will let you make and test changes without having to re-import the driver within LINQPad. There’s a batch file in the demo project called **devdeploy.bat** that does exactly this—just edit the directories within this file and then call it from the project post-build event.

## Versioning

Whatever you put in your assembly’s **AssemblyFileVersion** attribute will appear in LINQPad’s dialog when the user selects a driver. This helps users in knowing whether they’re running the latest version.

Users can update drivers simply by re-import the *.lpx* file. Existing files are overwritten.

To support the side-by-side loading of multiple drivers versions, you must either give the new driver a distinct class name (e.g. Foo20), or put the class into a library with a different assembly name or signature. You should also change the value returned by the **DataContextDriver.Name** property to make the distinction clear to the user (see next section).

The **AssemblyVersion** attribute is ignored.

# Writing a Driver

Both **DynamicDataContextDriver** and **StaticDataContextDriver** are based on a common base class called **DataContextDriver** (in LINQPad.Extensibility.DataContext) which defines the following abstract methods:

/// <summary>User-friendly name for your driver.</summary>

public abstract string Name { get; }

/// <summary>Your name.</summary>

public abstract string Author { get; }

/// <summary>Returns the text to display in the root Schema Explorer node for a given

connection info.</summary>

public abstract string GetConnectionDescription (IConnectionInfo cxInfo);

/// <summary>Displays a dialog prompting the user for connection details. The isNewConnection

/// parameter will be true if the user is creating a new connection rather than editing an

/// existing connection. This should return true if the user clicked OK. If it returns false,

/// any changes to the IConnectionInfo object will be rolled back.</summary>

public abstract bool ShowConnectionDialog (IConnectionInfo cxInfo, bool isNewConnection);

The first step is to implement these abstract methods. The only substantial method here is **ShowConnectionDialog**, which must display a (modal) WPF or Windows Forms dialog prompting the user for connection information. The samples should get you started.

If you choose Windows Forms to write the UI, beware that LINQPad calls **user32.SetProcessDPIAware**, so you’ll need to be high-DPI friendly; the general rule is that *every* control should be either **docked** or in a **table layout panel**, and have auto-sizing enabled. If you’re unsure, use WPF instead.

The most significant member of **IConnectionInfo** is **DriverData** (of type **XElement**). This lets you store and retrieve arbitrary data to feed the dialog.

**DataContextDriver** also exposes a number of virtual methods that you can override to provide additional functionality. These are covered later, in “Fine-Tuning”.

The type also provides the following helper methods for your convenience:

/// <summary>Returns a friendly name for a type, suitable for use in the Schema Explorer.</summary>

public static string FormatTypeName (Type t, bool includeNamespace)

{

 return TypeUtil.FormatTypeName (t, includeNamespace);

}

/// <summary>Returns the folder containing your driver assembly.</summary>

public string GetDriverFolder ()

{

 return Path.GetDirectoryName (GetType().Assembly.Location);

}

There’s also a method called **LoadAssemblySafely**, which we discuss later.

## Writing a Static Driver

To write a static data context driver, subclass **StaticDataContextDriver**. You will need to implement the abstract methods described in the previous section, plus the following method:

/// <summary>Returns a hierarchy of objects describing how to populate the Schema Explorer.</summary>

public abstract List<ExplorerItem> GetSchema (IConnectionInfo cxInfo, Type customType);

The best way to write this method is to start with the code in the **UniversalStaticDriver** class in the demo project and tweak it as necessary—you might find that it’s already 90% there. This code relies purely on reflecting the typed data context. You can also (or instead) infer the schema from what you’ve stored in **cxInfo**.

Note that the code in the **UniversalStaticDriver** sample won’t populate additional schema objects such as stored procedures and functions. The following code illustrates the use of **ExplorerItemKind** and **ExplorerIcon** in creating nodes for stored procedures:

var sprocs = new ExplorerItem ("Stored Procs", ExplorerItemKind.Category, ExplorerIcon.StoredProc)

{

 Children = new List<ExplorerItem>

 {

 new ExplorerItem ("UpdateCustomerName", ExplorerItemKind.QueryableObject, ExplorerIcon.StoredProc)

 {

 Children = new List<ExplorerItem>

 {

 new ExplorerItem ("ID", ExplorerItemKind.Parameter, ExplorerIcon.Parameter),

 new ExplorerItem ("Name", ExplorerItemKind.Parameter, ExplorerIcon.Parameter),

 }

 }

 }

};

LINQPad calls all driver methods in an isolated application domain. In the case of **GetSchema**, LINQPad destroys the domain immediately after the method runs. This means you can freely load assemblies into memory without worrying about locking assemblies or affecting subsequent assembly resolution. (See “Application Domains” for more information.)

There’s a helper method on IConnectionInfo for getting the names of public types in the user’s typed DataContext assembly:

 string[] customTypes = cxInfo.CustomTypeInfo.GetCustomTypesInAssembly();

## Writing a Dynamic Driver

To write a static data context driver, subclass **DynamicDataContextDriver.** In addition to implementing the standard abstract methods in **DataContextDriver**, you’ll need to implement this:

/// <summary>

/// Builds an assembly containing a typed data context, and returns data for the Schema Explorer.

/// </summary>

/// <param name="cxInfo">Connection information, as entered by the user</param>

/// <param name="assemblyToBuild">Name and location of the target assembly to build</param>

/// <param name="nameSpace">The suggested namespace of the typed data context. You must update this

/// parameter if you don't use the suggested namespace.</param>

/// <param name="typeName">The suggested type name of the typed data context. You must update this

/// parameter if you don't use the suggested type name.</param>

/// <returns>Schema which will be subsequently loaded into the Schema Explorer.</returns>

public abstract List<ExplorerItem> GetSchemaAndBuildAssembly (IConnectionInfo cxInfo,

 AssemblyName assemblyToBuild, ref string nameSpace, ref string typeName);

This method must do two things:

* Dynamically build an assembly containing a typed data context
* Return the schema to display in the Schema Explorer

In the demo project there’s a complete example of a dynamic driver for OData 3 (WCF Data Sservices). This is functionally similar to LINQPad’s built-in driver for the same.

There are no restrictions on the kinds of members that a typed data context can expose. Your design goal should be to provide the best querying experience for the end user.

Building the **List<ExplorerItem>** for the Schema Explorer is just as with a static driver. However, there are two ways to source the raw information:

* Build the schema from the same metadata that you used to build the typed data context
* First build the typed data context, and then reflect over the typed data context to build the **List<ExplorerItem>**.

The advantage of the first approach is that you have more data on hand. This extra information can help, for instance, in distinguishing many:1 from many:many relationships.

# Application Domains & Loading Assemblies

To provide isolation between queries, LINQPad runs each query **in its own process or application domain** (we call these “*query domains*”). If a user creates five queries, each runs its own *query domain*—even if they all use the same data context.

The following virtual driver methods all execute in a query domain (these methods are explained in “Fine-Tuning”):

GetContextConstructorArguments
InitializeContext
TearDownContext
OnQueryFinishing
GetCustomDisplayMemberProvider
PreprocessObjectToWrite
DisplayObjectInGrid
GetProviderFactory
GetIDbConnection
ExecuteESqlQuery
ClearConnectionPools

When running in a query domain, LINQPad ensures that assemblies that the user might want to rebuild in Visual Studio are “shadowed” to a temporary folder, so that they’re not locked while the query is open. This includes everything in a static data context folder, as well as any other reference the user has brought in that aren’t part of **ProgramFiles** or the **Windows** directory. You don’t have to do anything special to take advantage of shadowing, unless you want to load an assembly explicitly with **Assembly.LoadFrom**/**LoadFile**, in which case you should instead call DataContextDriver’s helper method, **LoadAssemblySafely**. This will ensure that (a) shadowed assemblies are loaded from the correct location, and (b) you don’t end up with multiple copies of the same assembly in memory.

Driver methods that aren’t in the above list (such as **GetConnectionDescription**) don’t run in a query domain—because a query doesn’t exist when they’re called. Instead, LINQPad creates an app domain per driver (a “*driver domain*”) in which to run them. If you call **LoadAssemblySafely** from a driver domain, it simply thunks to Assembly.LoadFrom. LINQPad recycles the driver domain after calling GetSchema, GetSchemaAndBuildAssembly and ShowConnectionDialog. This means you can load user assemblies directly in these methods and not worry about the effects of locking them.

## Assembly Resolution

You may need to reference other assemblies that are not part of the .NET Framework. In general, you can simply package them in the *.lpx* file and LINQPad will load them automatically as needed. If you want to *explicitly* load an assembly, use the **LoadAssemblySafely** method, together with **GetDriverFolder** to locate the file:

 var a = LoadAssemblySafely (Path.Combine (GetDriverFolder(), "stuff.dll"));

/// <summary>Loads an assembly with safeguards to avoid locking user assemblies and

/// ensure that duplicate assemblies do not end up in memory. Always use this method

/// to load additional non-framework assemblies. You must provide a full valid path

/// although LINQPad will not always load the assembly from the path you specify.</summary>

public static Assembly LoadAssemblySafely (string fullFilePath)

**GetDriverFolder** is the means by which you can locate other files, too:

 string xmlPath = Path.Combine (GetDriverFolder(), "data.xml"));

If you’re writing a static driver, an interesting scenario arises if the user’s typed DataContext folder contains another copy of an assembly that’s in your driver folder. This happens typically with ORMs; for instance:

 ...\LINQPad\Drivers\DataContext\4.6\MyDriver (ff414cf4a100c74d)\MyDriver.dll ...\LINQPad\Drivers\DataContext\4.6\MyDriver (ff414cf4a100c74d)\**MyOrm.dll**

 c:\source\projectxyz\MyCustomDataContext.dll
 c:\source\projectxyz\**MyOrm.dll**

(This clash occurs only with methods that run in the *query domain*, not the *driver domain*. With methods/properties that run in the driver domain, there is no user data context folder to clash with.)

LoadAssemblySafely resolves the clash by favoring **user assemblies** ahead of your own. This works even if the assemblies in question (MyOrm.dll, in this case) have different **AssemblyVersion** attributes, and will not cause trouble as long as there are no functional incompatibilities (in which case, you need to write separate drivers for different versions of your ORM.)

Hence with static drivers, your driver may be talking to any version of MyOrm.dll, not necessarily the one that shipped with your driver.

*Always* use **LoadAssemblySafely** if you need to explicitly load non-Framework assemblies.

**Assembly.LoadFrom** / **AssemblyLoadFile** will land you in **DLL hell**. Remember that the user can hit F4 and add any assembly references they like, including ones that *conflict* with your own. LINQPad jumps through some hoops (by applying its own fusion rules) to smooth things over, but only if you use **LoadAssemblySafely**.

Remember that in most cases, you don’t need to explicitly load assemblies at all: just statically reference them and ship them with *.lpx* file. LINQPad resolves references automatically by handling the application domain’s **AssemblyResolve** event and forwarding the resolution to **LoadAssemblySafely**.

It’s possible to ship a static driver without any ORM assemblies and rely purely on the assemblies in the user data context folder. However, you must then be careful not to consume any ORM types from methods/properties that execute in the driver domain, such as ShowConnectionDialog, Name, Version, Author, etc.

# Fine-Tuning

## Passing Arguments into a Data Context’s Constructor

To pass arguments into a data context class constructor, override the following two methods:

/// <summary>Returns the names & types of the parameter(s) that should be passed into your data

/// context's constructor. Typically this is a connection string or a DbConnection. The number

/// of parameters and their types need not be fixed - they may depend on custom flags in the

/// connection's DriverData. The default is no parameters.</summary>

public virtual ParameterDescriptor[] GetContextConstructorParameters (IConnectionInfo cxInfo)

{

 return null;

}

/// <summary>Returns the argument values to pass into your data context's constructor, based on

/// a given IConnectionInfo. This must be consistent with GetContextConstructorParameters.</summary>

public virtual object[] GetContextConstructorArguments (IConnectionInfo cxInfo) { return null; }

Refer to the **AstoriaDriver** in the demo project for an example.

A typical scenario is passing a connection string to a data context’s constructor. This avoids the need to hard-code the connection string into the typed data context, and avoids the need for application configuration files. (If you do want to rely on application configuration files supplied by the user, refer to the **UniversalStaticDriver** example).

## Working with Databases and Connection Strings

As just described, feeding a connection string to the data context’s constructor is a common scenario. If you do this, you’ll need to prompt the user for that connection string in the dialog. There are two ways to proceed:

* (Less work) Prompt the user for the provider name and connection string using a combo box and multiline text box. Save the provider invariant name to **cxInfo.DatabaseInfo.Provider** and the connection string to **cxInfo.DatabaseInfo.CustomCxString**.

Tip: you can populate the provider combo box as follows:

DbProviderFactories.GetFactoryClasses ().Rows

 .OfType<DataRow> ()

 .Select (r => r ["InvariantName"])

 .ToArray ()

* (More work) Write a friendly connection dialog that prompts the user for the server, database, authentication details, etc. If you’re supporting only SQL Server and SQL CE, you’ll find numerous properties on **cxInfo.DatabaseInfo** to store your data; if you populate these correctly you can call **GetCxString** / **GetConnection** to get a valid connection string / **IDbConnection**.

If you want to support other databases, however, you’ll need to save the details to custom elements in **cxInfo.DriverData**. You should then build the connection string yourself and write it to **cxInfo.DatabaseInfo.CustomCxString** (if you fail to take this step, LINQ queries will work but users won’t be able to write old-fashioned SQL queries, unless you override **GetIDbConnection**—see “Supporting SQL Queries”).

## Performing Additional Initialization / Teardown on the Data Context

You might want to assign properties on a newly created data context—or call methods to perform further initialization. To do so, override **InitializeContext**. You can also perform teardown by overriding **TearDownContext**:

/// <summary>This virtual method is called after a data context object has been instantiated, in

/// preparation for a query. You can use this hook to perform additional initialization work.</summary>

public virtual void InitializeContext (IConnectionInfo cxInfo, object context,

 QueryExecutionManager executionManager) { }

/// <summary>This virtual method is called after a query has completed. You can use this hook to

/// perform cleanup activities such as disposing of the context or other objects.</summary>

public virtual void TearDownContext (IConnectionInfo cxInfo, object context,

 QueryExecutionManager executionManager,

 object[] constructorArguments) { }

**TearDownContext** does not run if the user calls the **Cache** extension method or the **Util.Cache** method to preserve results between query runs.

A useful application of overriding **InitializeContext** is to set up population of the SQL translation tab.

When a query runs, LINQPad preserves the same DataContextDriver object from the time it calls InitializeContext to when it calls TearDownContext. This means store state in fields that you define in your data context driver class. You can safely access this state in GetCustomDisplayMemberProvider, PreprocessObjectToWrite and OnQueryFinishing.

There’s also an **OnQueryFinishing** method that you can override. Unlike TearDownContext, this runs just *before* the query ends, so you can Dump extra output in this method. You can also block for as long as you like—while waiting on some background threads to finish, for instance. If the user gets tired of waiting, they’ll hit the Cancel button in which case your thread will be aborted, and the TearDownContext method will then run. (The next thing to happen is that your application domain/process will be torn down and recreated, unless the user’s requested otherwise in Edit | Preferences | Advanced, or has cached objects alive).

/// <summary>This method is called after the query's main thread has finished running the user's code,

/// but before the query has stopped. If you've spun up threads that are still writing results, you can

/// use this method to wait out those threads.</summary>
public virtual void OnQueryFinishing (IConnectionInfo cxInfo, object context,
 QueryExecutionManager executionManager) { }

­Another way to extend a query’s “life” is to call Util.KeepRunning(). Calling this puts the query into an “asynchronous” state upon completion until you dispose the token. This is, in fact, how LINQPad deals with IObservables and C# 5.0’s asynchronous functions.

## Populating the SQL Translation Tab

In overriding **InitializeContext**, you can access properties on the **QueryExecutionManager** object that’s passed in as a parameter. One of these properties is called **SqlTranslationWriter** (type TextWriter) and it allows you to send data to the SQL translation tab.

Although this tab is intended primary for SQL translations, you can use it for other things as well. For example, with WCF Data Services, it makes sense to write HTTP requests here:

public override void InitializeContext (IConnectionInfo cxInfo, object context,
 QueryExecutionManager executionManager)
{
 var dsContext = (DataServiceContext)context;

 dsContext.SendingRequest += (sender, e) =>
 executionManager.SqlTranslationWriter.WriteLine (e.Request.RequestUri);
}

## Importing Additional Assemblies and Namespaces

You can make queries automatically reference additional assemblies and import additional namespaces by overriding the following methods:

/// <summary>Returns a list of additional assemblies to reference when building queries. To refer to

/// an assembly in the GAC, specify its fully qualified name, otherwise specified the assembly's full

/// location on the hard drive. Assemblies in the same folder as the driver, however, don't require a

/// folder name. If you're unable to find the necessary assemblies, throw an exception, with a message

/// indicating the problem assembly.</summary>

public virtual IEnumerable<string> GetAssembliesToAdd (IConnectionInfo cxInfo)

/// <summary>Returns a list of additional namespaces that should be imported automatically into all

/// queries that use this driver. This should include the commonly used namespaces of your ORM or

/// querying technology .</summary>

public virtual IEnumerable<string> GetNamespacesToAdd (IConnectionInfo cxInfo)

LINQPad references the following assemblies automatically:

"System.dll",

 "Microsoft.CSharp.dll", (in version 4.x)

 "System.Core.dll",

 "System.Data.dll",

 "System.Transactions.dll",

 "System.Xml.dll",

 "System.Xml.Linq.dll",

 "System.Data.Linq.dll",

 "System.Drawing.dll",

 "System.Data.DataSetExtensions.dll"

 "LINQPad.exe"

LINQPad imports the following namespaces automatically:

 "System",

 "System.IO",

 "System.Text",

 "System.Text.RegularExpressions",

 "System.Diagnostics",

 "System.Threading",

 "System.Reflection",

 "System.Collections",

 "System.Collections.Generic",

 "System.Linq",

 "System.Linq.Expressions",

 "System.Data",

 "System.Data.SqlClient",

 "System.Data.Linq",

 "System.Data.Linq.SqlClient",

 "System.Transactions",

 "System.Xml",

 "System.Xml.Linq",

 "System.Xml.XPath",

 "LINQPad"

You can prevent LINQPad from importing any of these namespaces by overriding this method:

/// <summary>Returns a list of namespace imports that should be removed to improve the autocompletion

/// experience. This might include System.Data.Linq if you're not using LINQ to SQL.</summary>

public virtual IEnumerable<string> GetNamespacesToRemove (IConnectionInfo cxInfo)

Removing the **System.Data.Linq** namespace makes sense if you’re writing driver for an ORM, because you might otherwise conflict with LINQ to SQL’s type names.

## Overriding AreRepositoriesEquivalent

After you’ve got everything else working, a nice (and easy) touch is to override **AreRepositoriesEquivalent**. This ensures that if a user runs a LINQ query created on another machine that references a different (but equivalent) connection, you won’t end up with multiple identical connections in the Schema Explorer.

Here’s the default implementation:

/// <summary>Returns true if two <see cref="IConnectionInfo"/> objects are semantically equal.</summary>

public virtual bool AreRepositoriesEquivalent (IConnectionInfo c1, IConnectionInfo c2)

{

 if (!c1.DatabaseInfo.IsEquivalent (c2.DatabaseInfo)) return false;

 return c1.DriverData.ToString() == c2.DriverData.ToString();
}

The call to **DriverData.ToString()** can lead to false positives, as it’s sensitive to XML element ordering. Here’s an overridden version for the AstoriaDynamicDriver (WCF Data Services):

public override bool AreRepositoriesEquivalent (IConnectionInfo r1, IConnectionInfo r2)

{

 // Two repositories point to the same endpoint if their URIs are the same.

 return object.Equals (r1.DriverData.Element ("Uri"), r2.DriverData.Element ("Uri"));
}

## Overriding GetLastSchemaUpdate (dynamic drivers)

Another nice touch with dynamic drivers for databases is to override **GetLastSchemaUpdate**. This method is defined in **DynamicDataContextDriver**:

/// <summary>Returns the time that the schema was last modified. If unknown, return null.</summary>

public virtual DateTime? GetLastSchemaUpdate (IConnectionInfo cxInfo) { return null; }

LINQPad calls this after the user executes an old-fashioned SQL query. If it returns a non-null value that’s later than its last value, it automatically refreshes the Schema Explorer. This is useful in that quite often, the reason for users running a SQL query is to create a new table or perform some other DDL.

Output from this method may also be used in the future for caching data contexts between sessions.

With static drivers, no action is required: LINQPad installs a file watcher on the target assembly. When that assembly changes, it automatically refreshes the Schema Explorer.

## Supporting SQL Queries

If you’re writing a driver for databases, LINQPad lets users run old-fashioned SQL queries, by setting the query language to “SQL”. If it makes sense for your driver to support this, you can gain more control over how connections are created by overriding the following methods:

/// <summary>Allows you to override the default factory, which is obtained by calling

/// DbProviderFactories.GetFactory on DatabaseInfo.Provider. This can be useful if you want

/// to use uninstalled database drivers. This method is called if the user executes a query

/// with the language set to 'SQL'. Overriding GetIDbConnection renders this method redundant.</summary>

public virtual DbProviderFactory GetProviderFactory (IConnectionInfo cxInfo)

{

 try { return DbProviderFactories.GetFactory (cxInfo.DatabaseInfo.Provider); }

 catch (ArgumentException ex)

 {

 throw new DisplayToUserException (ex.Message, ex); // Not installed

 }

}

/// <summary>Instantiates a database connection for queries whose languages is set to 'SQL'.

/// By default, this calls cxInfo.DatabaseInfo.GetCxString to obtain a connection string,

/// then GetProviderFactory to obtain a connection object. You can override this if you want

/// more control over creating the connection or connection string.</summary>

public virtual IDbConnection GetIDbConnection (IConnectionInfo cxInfo)

{

 string cxString = cxInfo.DatabaseInfo.GetCxString ();

 if (string.IsNullOrEmpty (cxString))

 throw new DisplayToUserException ("A valid database connection string could not be obtained.");

 var cx = GetProviderFactory (cxInfo).CreateConnection ();

 cx.ConnectionString = cxInfo.DatabaseInfo.GetCxString ();

 return cx;

}

Overriding **GetIDbConnection** means you don’t have to populate the connection string in **DatabaseInfo**. There’s also a method that you can override to support “ESQL” queries, although this is really only relevant to Entity Framework:

public virtual void ExecuteESqlQuery (IConnectionInfo cxInfo, string query)

{

 throw new Exception ("ESQL queries are not supported for this type of connection");

}

## Clearing Connection Pools

If your driver creates database connections, you can override the following method, which is called when the user right-clicks a connection and chooses “Clear all connections”.

public virtual void ClearConnectionPools (IConnectionInfo cxInfo)

{

}

## Application Configuration Files

With static data context drivers, a user’s assembly may rely on an application configuration file. You can specify its location either by writing it to **IConnectionInfo.AppConfigPath**, or by overriding the following driver method:

public virtual string GetAppConfigPath (IConnectionInfo cxInfo)

{

 return cxInfo.AppConfigPath;

}

## Customizing the Icon

You can provide a custom 16x16 icon for your data contexts. Just include two files in your driver folder: **Connection.png** and **FailedConnection.png** (the latter is applied when a connection is in error). These are fed into a 16x16 ImageList and are upscaled in high-DPI scenarios.

## Custom Features

The follow methods currently do nothing. They are to support specialized options in the future without breaking driver compatibility:

public object InvokeCustomOption (string optionName, params object [] data)

{

 return null;

}

public virtual object OnCustomEvent (string eventName, params object [] data)

{

 return null;

}

# Customizing Output

LINQPad’s output window works by walking object graphs that you Dump, emitting XHTML which it then displays in an embedded web browser. This is the normal “Rich Text” output mode (LINQPad also lets you display results to data grids; we cover this later).

There are three reasons for wanting to customize LINQPad’s output.

* If your objects expose lazily evaluated navigation properties, LINQPad will (in standard output mode) walk them **eagerly** in rendering the output, resulting in additional queries. (And if those entities themselves contain lazily evaluated properties, it can go exponential!)
* You might want to hide fields and properties in your entities that are uninteresting and create clutter.
* You might want to *transform* properties or create new ones, or replace the entire HTML rendering for an object to improve the output.

There are two ways to control output formatting. The first is to override **PreprocessObjectToWrite**: the idea here is that you simply replace the object in question with another one that has the members that you want to render (or else, simply, the desired HTML). This behaves like the ToDump method introduced in LINQPad 5.09 ([click here](http://www.linqpad.net/CustomizingDump.aspx) for details).

Overriding **PreprocessObjectToWrite** is conceptually simple, but creating a proxy with the right members (used to be) awkward if the members need to be chosen at runtime. The solution, in the past, has been to override **GetCustomDisplayMemberProvider** and implement **ICustomMemberProvider**, which lets you return an array of values to display, along with their names and types. However, since LINQPad 5.09, you can now stick with **PreprocessObjectToWrite** and provide any level of customization by returning a **System.Dynamic.ExpandoObject** ([click here](http://www.linqpad.net/CustomizingDump.aspx) for details).

## Overriding PreprocessObjectToWrite

/// <summary>This lets you replace any non-primitively-typed object with another object for

/// display. The replacement object can be a System.Dynamic.ExpandoObject.</summary>

public virtual void PreprocessObjectToWrite (ref object objectToWrite, ObjectGraphInfo info) { }

LINQPad calls **PreprocessObjectToWrite** before writing all non-primitive types, including enumerables and other objects. You can replace **objectToWrite** with anything you like; it can be an object specially designed for output formatting (effectively a proxy), or even a System.Dynamic.ExpandoObject. For columns that are expensive to evaluate, or that cause server round-tripping, you can return a Lazy<T>, which renders in LINQPad as a hyperlink that evaluates when the user clicks on it.

With Util.RawHtml, you can even output HTML directly:

 if (objectToWrite is MySpecialEntity) objectToWrite = Util.RawHtml ("<h1>foo</h1>");

(In the following section, there’s a more elaborate example on how to detect entities and entity collections.)

To “swallow” the object entirely so that nothing is written, setting **objectToWrite** to **null** might seem reasonable, but it won’t work because ‘null’ will be then written in green. Instead, do this:

objectToWrite = info.DisplayNothingToken;

## Implementing ICustomMemberProvider

Since LINQPad 5.09, ICustomMemberProvider is no longer necessary and you can use **PreprocessObjectToWrite** instead (see preceding two sections). This section still contains a useful discussion on how to detect entities and entity collections.

Another way to control output formatting is to override the following driver method:

/// <summary>Allows you to change how types are displayed in the output window - in particular, this

/// lets you prevent LINQPad from endlessly enumerating lazily evaluated properties. Overriding this

/// method is an alternative to implementing ICustomMemberProvider in the target types. See

/// http://www.linqpad.net/FAQ.aspx#extensibility for more info.</summary>

public virtual ICustomMemberProvider GetCustomDisplayMemberProvider (object objectToWrite)

{

 return null;

}

If **objectToWrite** is not an entity whose output you want to customize, return null. Otherwise, return an object that implements **ICustomMemberProvider**:

 public interface ICustomMemberProvider

 {

 // Each of these methods must return a sequence

 // with the same number of elements:

 IEnumerable<string> GetNames();

 IEnumerable<Type> GetTypes();

 IEnumerable<object> GetValues();

 }

Here’s an example of implementing **ICustomMemberProvider**:

 public class Foo : ICustomMemberProvider

 {

 public string FirstName, LastName, Junk;

 IEnumerable<string> ICustomMemberProvider.GetNames()

 => new string[] { "FirstName", "LastName" };

 IEnumerable<Type> ICustomMemberProvider.GetTypes ()

 => new Type[] { typeof(string), typeof(string) };

 IEnumerable<object> ICustomMemberProvider.GetValues ()

 => new object[] { FirstName, LastName };

 }

An alternative to overriding **GetCustomDisplayMemberProvider** is to [implement ICustomMemberProvider](http://www.linqpad.net/FAQ.aspx#extensibility) in your entity type itself; this ensures that your custom output formatting takes effect whether or not your driver is in use. This can be done without taking a dependency on LINQPad.exe, as described in the link.

If you’re implementing **ICustomMemberProvider** in the type itself, you might want to substitute objectToWrite with another object (in other words, you might want the same functionality as **PreprocessObjectToWrite**). The good news is that you can do this with **ICustomMemberProvider**—in a slightly roundabout way. To give an example, let’s suppose you want the object to be rendered as a simple string. Here’s the solution:

 public class Foo : ICustomMemberProvider

 {

 public string FirstName, LastName, Junk;

 IEnumerable<string> ICustomMemberProvider.GetNames()

 => new string[] { "" };

 IEnumerable<Type> ICustomMemberProvider.GetTypes ()

 => new Type[] { typeof(string) };

 IEnumerable<object> ICustomMemberProvider.GetValues ()

 => new object[] { FirstName + " " + LastName };

 }

(Notice that **GetNames** returns an empty string. This tells LINQPad to “collapse” the containing object.)

If you’re writing a driver for an ORM, and have generic types such as MyEntity<T>, you can identify what are your entities by looking for the base type. For instance, suppose all entities are based on **Entity<T>**, and entity collections are of some type which implements **IEnumerable<T>**, where T is an entity:

/// <summary>Ensure that the output window ignores nested entities and entity collections.</summary>

public override LINQPad.ICustomMemberProvider GetCustomDisplayMemberProvider (object objectToWrite)

{

 if (objectToWrite != null && EntityMemberProvider.IsEntity (objectToWrite.GetType ()))

 return new EntityMemberProvider (objectToWrite);

 return null;

}

class EntityMemberProvider : LINQPad.ICustomMemberProvider

{

 public static bool IsEntity (Type t)

 {

 while (t != null)

 {

 if (t.IsGenericType && t.GetGenericTypeDefinition () == typeof (Entity<>)) return true;

 t = t.BaseType;

 }

 return false;

 }

 public static bool IsEntityOrEntities (Type t)

 {

 // For entity collections, switch to the element type:

 if (t.IsGenericType)

 {

 Type iEnumerableOfT = t.GetInterface ("System.Collections.Generic.IEnumerable`1");

 if (iEnumerableOfT != null) t = iEnumerableOfT.GetGenericArguments () [0];

 }

 return IsEntity (t);

 }

 object \_objectToWrite;

 PropertyInfo [] \_propsToWrite;

 public EntityMemberProvider (object objectToWrite)

 {

 \_objectToWrite = objectToWrite;

 \_propsToWrite = objectToWrite.GetType ().GetProperties ()

 .Where (p => p.GetIndexParameters ().Length == 0 && !IsEntityOrEntities (p.PropertyType))

 .ToArray ();

 }

 public IEnumerable<string> GetNames ()

 {

 return \_propsToWrite.Select (p => p.Name);

 }

 public IEnumerable<Type> GetTypes ()

 {

 return \_propsToWrite.Select (p => p.PropertyType);

 }

 public IEnumerable<object> GetValues ()

 {

 return \_propsToWrite.Select (p => p.GetValue (\_objectToWrite, null));

 }

}

Note the following predicate in EntityMemberProvider’s constructor:

 p => p.GetIndexParameters ().Length == 0

This is important in that we don’t want to enumerate indexers.

## Customizing Output to Data Grids

If the user chooses “Results to Data Grids”, the output customizations in the above sections (PreprocessObjectToWrite and GetCustomDisplayMemberProvider) do not apply. To customize data grid display, you instead override the following driver method:

 public virtual void DisplayObjectInGrid (object objectToDisplay, GridOptions options)

Here’s **GridOptions**:

 public class GridOptions

 {

 public string PanelTitle { get; set; }

 public string[] MembersToExclude { get; set; }

 }

In general, you don’t need to exclude (or make lazy) properties that are expensive to evaluate or that cause round-tripping. This is because LINQPad works differently when rendering grids: unlike with HTML formatting, it does not eagerly walk object graphs (it doesn’t need to in order to display data in a flat grid). Instead, upon encountering a non-primitive object, it displays a hyperlink in the grid and evaluates that object only when the user clicks the link. Hence you can (and will want to) expose all lazily evaluated properties.

You might still want to remove extraneous members from the output, though, or perhaps take over the rendering entirely with your own UI control. Both are possible (and easy) by overriding

Here’s how to tell LINQPad to remove the fields/properties named “\_context”, “ChangeTracker” and “EntityState”:

 public override void DisplayObjectInGrid (object objectToDisplay, GridOptions options)

 {

 if (IsEntityOrEntities (objectToDisplay.GetType()))

 options.MembersToExclude = "\_context ChangeTracker EntityState".Split ();

 base.DisplayObjectInGrid (objectToDisplay, options);

 }

(The **IsEntityOrEntities** method comes from the example in the preceding section.)

You can swap out the object to render simply by calling **base.DisplayObjectInGrid** with a different object. You can use the trick of returning a **System.Dynamic.ExpandoObject** to create columns on the fly ([see here](http://www.linqpad.net/CustomizingDump.aspx)).

You can also tell LINQPad to display your own control in place of its DataGrid, simply by not calling the base method at all and instead dumping a WPF or Windows Forms control:

 public override void DisplayObjectInGrid (object objectToDisplay, GridOptions options)

 {

 if (IsEntityOrEntities (objectToDisplay.GetType()))

 new System.Windows.Forms.DataGrid { DataSource = objectToDisplay }.Dump (options.PanelTitle);

 else

 base.DisplayObjectInGrid (objectToDisplay, options);

 }

(You can gain more control over how LINQPad displays a Windows Forms control or WPF element by using the methods on LINQPad’s static **PanelManager** class instead of dumping the control.)

# Troubleshooting

## Exception Logging

If your driver throws an exception, LINQPad writes the exception details and stack trace to its log file. The log file sits in %localappdata%\linqpad\logs\

which is normally:

 C:\Users\*UserName*\AppData\Local\LINQPad\logs

## Debugging

To debug your driver:

* Start LINQPad
* From Visual Studio, go to **Debug | Attach to Process** and locate LINQPad.exe
* Set desired breakpoints in your project
* Enable break on exception in **Debug | Exceptions** if desired.

You can insert breakpoints in queries by calling methods on System.Diagnostics.Debugger (Launch, Break). You can also write queries that use reflection to display information about the current typed data context:

GetType().GetProperties()

Another trick is to run a query that calls ILSpy or .NET Reflector to examine the LINQPad-generated query:

Process.Start (@"c:\reflector\reflector.exe", GetType().Assembly.Location);

or the typed data context class:

Process.Start (@"c:\reflector\reflector.exe", GetType().BaseType.Assembly.Location);

This is particularly useful if using Reflection.Emit to dynamically build a typed data context.

# API Reference

**DynamicDataContextDriver** and **StaticDataContextDriver** are covered in previous sections. Following are some notes on the other types in the extensibility model.

## IConnectionInfo

/// <summary>

/// Describes a connection to a queryable data source. This corresponds to what

/// the user sees when they click 'Add Connection'.

/// </summary>

public interface IConnectionInfo

{

 /// <summary>Details of a database connection, if connecting to a database. This currently

 /// supports only SQL Server and SQL CE. If you want to support other databases, use

 /// use DriverData to supplement its properties.</summary>

 IDatabaseInfo DatabaseInfo { get; }

 /// <summary>Details of the custom type supplied by the user that contains the typed

 /// data context to query. This is relevant only if you subclass StaticDataContextDriver

 /// rather than DynamicDataContextDriver.</summary>

 ICustomTypeInfo CustomTypeInfo { get; }

 /// <summary>Standard options for dynamic schema generation. Use <see cref="DriverData"/>

 /// for any additional options that you wish to support.</summary>

 IDynamicSchemaOptions DynamicSchemaOptions { get; }

 /// <summary>Full path to custom application configuration file. Prompting the user for

 /// this is necessary if you're using, for instance, an ORM that obtains connection

 /// strings from the app.config file.</summary>

 string AppConfigPath { get; set; }

 /// <summary>Whether or not to save the connection details for next time LINQPad is

 /// started. Default is true.</summary>

 bool Persist { get; set; }

 /// <summary>Custom data. You can store anything you want here and it will be

 /// saved and restored. </summary>

 XElement DriverData { get; set; }

 // Helper methods

 /// <summary>Encrypts a string using Windows DPAPI. A null or empty string returns

 /// an empty string. This method should be used for storing passwords.</summary>

 string Encrypt (string data);

 /// <summary>Decrypts a string using Windows DPAPI. A null or empty string returns

 /// an empty string.</summary>

 string Decrypt (string data);

}

## IDatabaseInfo

public interface IDatabaseInfo

{

 /// <summary>The invariant provider name, as returned by

 /// System.Data.Common.DbProviderFactories.GetFactoryClasses().

 /// If this is not System.Data.SqlClient or System.Data.SqlServerCe.\*,

 /// you must populate CustomCxString. </summary>

 string Provider { get; set; }

 /// <summary>If this is populated, it overrides everything else except Provider.</summary>

 string CustomCxString { get; set; }

 string Server { get; set; }

 string Database { get; set; }

 bool AttachFile { get; set; }

 string AttachFileName { get; set; }

 bool UserInstance { get; set; }

 bool SqlSecurity { get; set; }

 string UserName { get; set; }

 string Password { get; set; }

 /// <summary>For SQL CE</summary>

 int MaxDatabaseSize { get; set; }

 // Helper methods:

 bool IsSqlServer { get; }

 bool IsSqlCE { get; }

 System.Data.Common.DbProviderFactory GetProviderFactory ();

 string GetCxString ();

 IDbConnection GetConnection ();

 string GetDatabaseDescription ();

 /// <summary>Returns true if another IDatabaseInfo refers to the same database.

 /// This ignores Password, for instance.</summary>

 bool IsEquivalent (IDatabaseInfo other);

}

## ICustomTypeInfo

public interface ICustomTypeInfo

{

 /// <summary>Full path to assembly containing custom schema.</summary>

 string CustomAssemblyPath { get; set; }

 /// <summary>Full type name (namespace + name) of custom type to query.</summary>

 string CustomTypeName { get; set; }

 /// <summary>Metadata path. This is intended mainly for Entity Framework.</summary>

 string CustomMetadataPath { get; set; }

 // Helper methods

 string GetCustomTypeDescription ();

 bool IsEquivalent (ICustomTypeInfo other);

 /// <summary>Returns an array of all public types in the custom assembly, without loading

 /// those types into the current application domain.</summary>

 string [] GetCustomTypesInAssembly ();

 /// <summary>Returns an array of all public types in the custom assembly, without loading

 /// those types into the current application domain.</summary>

 string [] GetCustomTypesInAssembly (string baseTypeName);

}

## ExplorerItem

[Serializable]

public class ExplorerItem

{

 public ExplorerItem (string text, ExplorerItemKind kind, ExplorerIcon icon)

 {

 Text = text;

 Kind = kind;

 Icon = icon;

 }

 public ExplorerItemKind Kind { get; set; }

 public string Text { get; set; }

 public string ToolTipText { get; set; }

 /// <summary>The text that appears when the item is dragged to the code editor.</summary>

 public string DragText { get; set; }

 public ExplorerIcon Icon { get; set; }

 /// <summary>If populated, this creates a hyperlink to another ExplorerItem. This is intended

 /// for association properties.</summary>

 public ExplorerItem HyperlinkTarget { get; set; }

 public List<ExplorerItem> Children { get; set; }

 /// <summary>Set to true to get the context menu to appear with query snippets such as

 /// Customers.Take(100). In general, this should be set to true with all items of kind

 /// QueryableObject except scalar functions.</summary>

 public bool IsEnumerable { get; set; }

 /// <summary>You can use this to store temporary data to help in constructing the object

 /// graph. The content of this field is not sent back to the host domain.</summary>

 [NonSerialized]

 public object Tag;

 /// <summary>For drivers that support SQL queries, this indicates the name of the underlying

 database object. This is shown when the user changes the query language to SQL.</summary>

 public string SqlName { get; set; }

 /// <summary>For drivers that support SQL queries, this indicates the type of the underlying

 database object (applicable to columns). This is shown when the user changes the query

 language to SQL.</summary>

 public string SqlTypeDeclaration { get; set; }

}

## ExplorerItemKind

**ExplorerItemKind** determines how an **ExplorerItem** will appear (and to some extent, behave) in the Schema Explorer treeview:

public enum ExplorerItemKind

{

 QueryableObject, Category, Schema, Parameter, Property, ReferenceLink, CollectionLink

}

**QueryableObject** is for objects that the user might want to query. Right now, this is the only member of the enum that has special *behavior*, which is to display a context menu with query snippets such as Customers.Take(100) when the user right-clicks on it. Note that you must also set the ExplorerItem’s **IsEnumerable** property set to true, otherwise the context menu won’t appear.

**Category** is for organizing groups of items, such as “Stored Procedures” or “Views”. See “Writing a Static Driver” for an example. **Schema** is for grouping into multiple schemas.

**Parameter** is for method/function parameters and **Property** is for simple objects or colums.

**ReferenceLink** displays the item underlined in blue. Use it for hyperlinked objects that point to a non-enumerable object (e.g., many:one associations). Use **CollectionLink** for a hyperlinked object that points to a collection, and for one:many and many:many relationships. The ExplorerItem’s **HyperlinkTarget** property indicates where to go when the user clicks on the link.

## ExplorerIcon

This determines an **ExplorerItem**’s icon in the Schema Explorer. It has no other functional significance.