

WRITING A LINQPAD DATA CONTEXT DRIVER

Joseph Albahari

Version 1.2

Last updated 2010-3-11

Send feedback to mail@albahari.com

Introduction

Who is this Guide for?

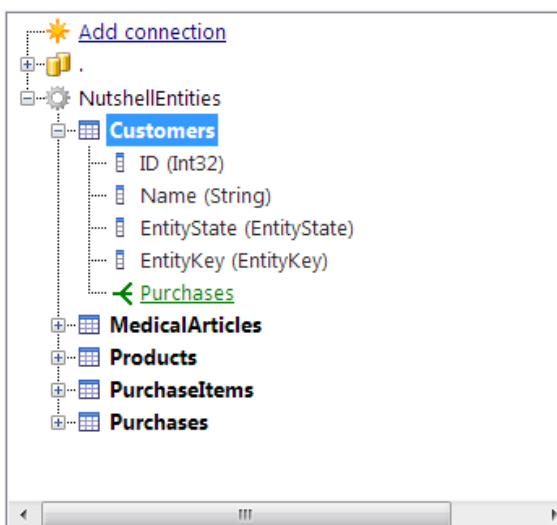
This guide is for programmers interested in adding first-class support to LINQPad for third-party object-relational mappers and other querying sources.

Why Write a Data Context Driver?

Without a data context driver, LINQPad can query any data source, but it requires users to manually reference appropriate libraries and import custom namespaces with each query. Further:

- The schema of the custom data source doesn't show in the Schema Explorer.
- A pre-compiled typed DataContext-equivalent is required
- Users must instantiate a DataContext-equivalent and possibly perform other administrative tasks such as assigning connection strings with each query.
- Users **must implement ICustomMemberProvider** (<http://www.linqpad.net/FAQ.aspx#extensibility>) in their entities to prevent LINQPad's output window from endlessly walking lazily evaluated properties.

These problems can all be overcome by writing a custom Data Context Driver. With such a driver, when the user clicks **Add Connection**, they're first presented with a friendly custom dialog prompting for connection details. Those details are then saved and the schema of the chosen query source shows in the Schema Explorer:



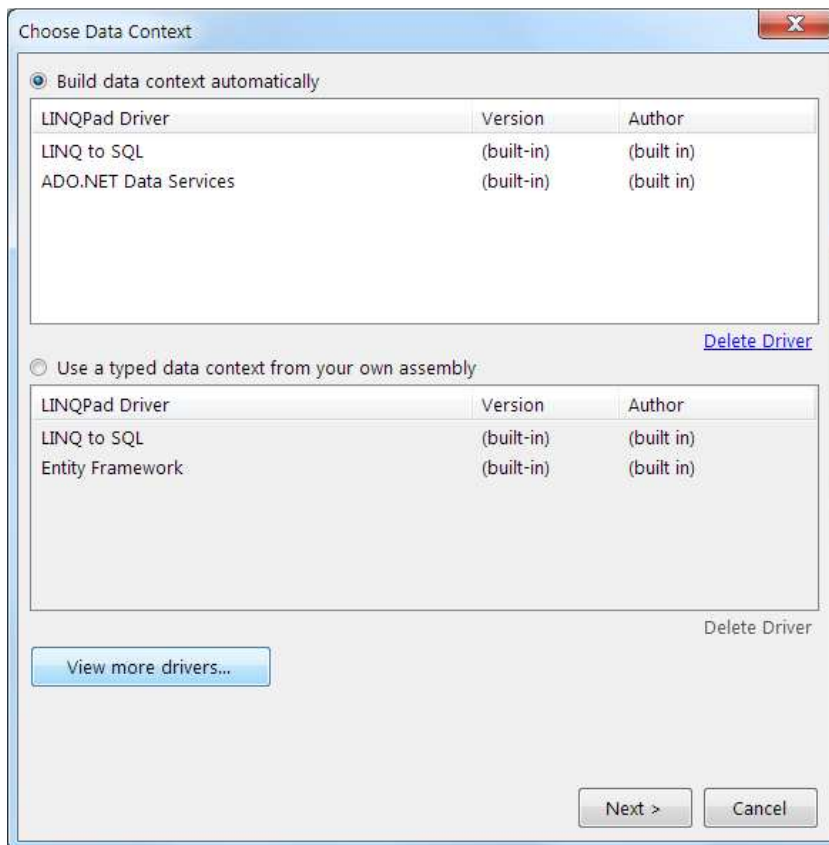
The user can write queries without needing to explicitly instantiate a DataContext-equivalent:

```
from c in Customers
where c.Name.StartsWith ("A")
select new { c.Name, c.Purchases }
```

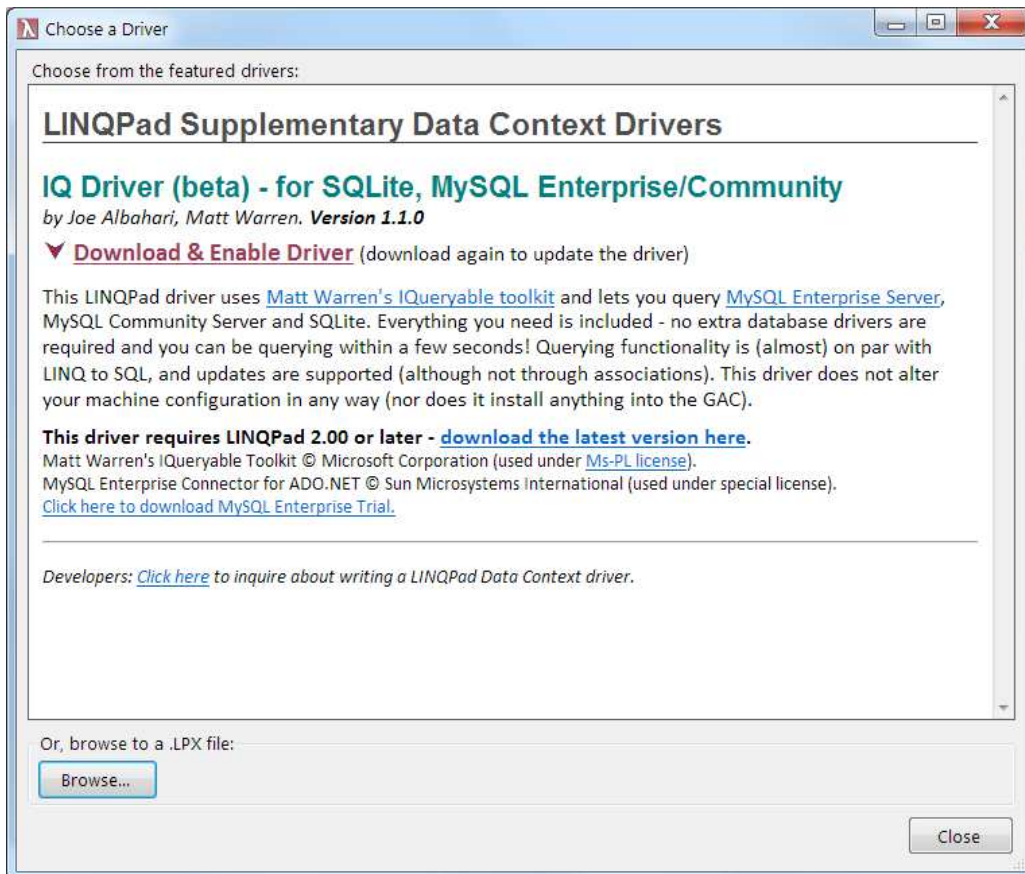
In writing the driver, you can also instruct LINQPad to skip over the lazily evaluated association properties when displaying entities such as **Customer** and **Purchase** in the output window. You can even populate the SQL translation tab in the output pane.

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 immediately visible in the previous dialog, and the user can enjoy first-class querying support.

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 .NET 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 the conditions will include the following:

1. There's no obligation to accept a particular submission.
2. Drivers must maintain compliance with LINQPad's "zero-impact" policy. That is, they must not make changes to settings on users' computers.
3. You must provide a support URI (this can be a peer support forum or FAQ page).

What's the Story with Framework 4.0?

There are two versions of the LINQPad executable: one for Framework 3.5 and one for Framework 4.0. In general, if you target Framework 3.5, your driver will work with both versions.

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 as 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 *data context* is some class that performs the role of the LINQ to SQL's **DataContext** class (or Entity Framework's **ObjectContext** class).

A *typed data context* is just like a typed **DataContext/ObjectContext** in LINQ to SQL and Entity Framework. In other words, it's a subclassed data context with properties for each of the queryable collections. For example:

```
public class TypedContext : DataContext
{
    public IQueryable<Customer> Customers
    { get { return this.GetTable<Customer>(); } }

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

The presence of 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

A Dynamic Data Context Driver provides the simplest experience for the end user, and mimics the default behavior of LINQPad. When the connection dialog appears, the user simply points to a database (or a URI in the case of something like ADO.NET Data Services). LINQPad then queries that database (or URI) for the schema and builds a typed data context on the fly which is then used for querying.

In contrast, a Static Data Context Driver requires that the user supplies the schema in the form of a custom typed data context. The connection dialog (which you supply) prompts the user for the path to a custom assembly containing the typed data context, and the name of the type. LINQPad doesn't then need to build anything—it simply uses the specified typed data context for querying. LINQPad has shipped with two built-in static drivers for

some time: one for LINQ to SQL and one for Entity Framework. (You can select one of these by changing from 'Automatic' to 'Custom Data Context' when adding a connection).

The advantage of a dynamic driver is end-user simplicity: the user can query without first writing a project in Visual Studio. The advantage of a static driver is twofold:

- In most cases, it's easier to write.
- It allows the user to query higher-level domain models which cannot be automatically inferred from the low-level schema.

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, IUserQuery
{
    public UserQuery (parameters...) : base (parameters...) { }

    void IUserQuery.RunUserQuery()
    {
        (
            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 `IUserQuery.RunUserQuery`. The same principle holds with both dynamic and static drivers.

It is therefore important that your typed data context class is not sealed.

We'll discuss later how to feed parameters into the constructor, in "Advanced Features".

Autocompletion

Autocompletion feeds entirely on the type system, so no special work is required to offer first-class support.

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"
- Signing | Sign the Assembly. Your assembly must be strong-name signed.

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 based in the following location:

```
Path.Combine (
  Environment.GetFolderPath (Environment.SpecialFolder.CommonApplicationData),
  @"LINQPad\Drivers\DataContext\3.5\" )
```

"3.5" refers to the .NET Framework version. If you're writing a 4.0 driver, this will be "4.0" instead. Tacked onto the end of this path is a folder comprising the name of the assembly and its public key token in parenthesis. For example, if you're running Vista or Windows 7, the driver files might end up in the following directory:

```
c:\ProgramData\LINQPad\Drivers\DataContext\3.5\MyDriver (ff414cf4a100c74d)\
```

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 an 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, note that LINQPad calls **user32.SetProcessDPIAware** so you should take care to be high-DPI friendly; exclusive use of table layout panels and auto-sizing controls and forms is recommended). The most significant member of **IConnectionInfo** is **DriverData** (of type **XElement**). This lets you store and retrieve arbitrary data to feed the dialog.

All methods are called from an isolated application domain. This domain is cached as LINQPad runs, however, so you should avoid loading end-user assemblies into memory (as this will stop the end user from rebuilding those assemblies). If you want to inspect assemblies using reflection, either create a separate application domain for this, or use the helper method on **IConnectionInfo**:

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

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

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);
}
```

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 abstract 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 proceed in writing 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) inspect metadata objects based on information in **cxInfo**.

Note that the code in **UniversalStaticDriver** 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.

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 ADO.NET Data Services. (This is functionally almost identical to LINQPad’s new built-in driver for Data Services).

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, 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.

Assembly Resolution

You may need to reference other assemblies. These fall into two categories:

- assemblies that are consumed purely by your driver, and packaged in the .lpx file
- assemblies that are part of your custom object relational mapper (assuming you are writing a driver for an ORM or something similar)

LINQPad will resolve assemblies in the first category automatically if implicitly loaded. If you want to *explicitly* load an assembly (or any other file), you should use the **GetDriverFolder** method on **DataContextDriver** to locate the file:

```
var a = Assembly.LoadFrom (Path.Combine (GetDriverFolder(), "stuff.dll"));
string xmlPath = Path.Combine (GetDriverFolder(), "data.xml");
```

If you're writing a dynamic driver, assemblies in the second category are analogous. Simply package them with the .lpx file and LINQPad will ensure that they resolve when referenced.

If you're writing a static driver, assemblies in the second category create the possibility of ambiguity. This is because the user's typed data context assembly will itself reference (other copies of) your object relational mapper assemblies:

```
c:\ProgramData\LINQPad\Drivers\DataContext\3.5\MyDriver (ff414cf4a100c74d)\MyDriver.dll
c:\ProgramData\LINQPad\Drivers\DataContext\3.5\MyDriver (ff414cf4a100c74d)\MyOrm.dll

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

In general, LINQPad favors the user's assemblies ahead of your own assemblies. In other words, if the same assembly exists in both the directory containing the user's typed data context and your driver directory, LINQPad will load the former rather than the latter. This is necessary to ensure that you don't end up with multiple versions of the same ORM assembly in memory at once. The catch, though, is that you might be talking to *any* (CLR-compatible) version of your ORM—not necessarily the one you ship with your driver!

Specifically, LINQPad favors the user's assemblies when it does any of the following:

- calls any of the following methods in your driver: **GetSchema**, **InitializeContext** (see "Advanced Features"), **TeardownContext**, **GetContextConstructorArguments**, **GetCustomDisplayMemberProvider**
- compiles the user's query
- executes the user's query

The consequence is that when implementing the above methods, you must **avoid statically binding** to your ORM assemblies if all of the following conditions are true:

- You're writing a static driver for an ORM (or any API with assemblies upon which you need to take a dependency)
- The ORM assemblies are strongly named
- You want the same driver to support multiple ORM builds that have different **AssemblyVersion** attribute values

There are three ways to avoid static binding:

- Use reflection to access ORM types and members
- (in CLR 4.0) Use dynamic binding to access ORM types and members
- (if you're the author of the ORM) Write and ship an 'interop' assembly with your ORM that acts as an adapter and whose AssemblyVersion number doesn't change over time. Your driver then talks to that adapter and doesn't directly reference the ORM assemblies.

There's an example of using reflection in "Populating the SQL Translation Tab" in the "Advanced Features" section. Dynamic binding in CLR 4.0 is the simpler approach, but it does restrict you to classes, structs and delegates: you can't dynamically bind to an interface in CLR 4.0.

Under what circumstances will LINQPad load the ORM assemblies that ship with the driver?

This happens when LINQPad calls methods not listed above, such as **ShowConnectionDialog**. In this case, it can't load the user's assemblies because it doesn't know where they are.

Is it possible to ship a static driver without any ORM assemblies?

Yes: as long as you don't consume any ORM types from within ShowConnectionDialog, or other members not listed above, such as Name, Version, Author, etc.

Advanced Features

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).

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) { }
```

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

Tip: 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.

In LINQPad 2.12 and above, there's also an **OnQueryFinishing** method that you can override. Unlike TearDownContext, this runs before the query actually ends, so you can Dump extra output in this method. You can also block for as long as you like—while waiting 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, if the user hit Cancel, is that your application domain will be torn down and recreated).

```
/// <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) { }
```

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 ADO.NET 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);
}
```

Note for static driver authors: Here's how to write the above code using reflection (see the earlier section "Assembly Resolution" for a discussion on when you'd want to do this):

```
// Let's suppose DataServiceContext was not part of the .NET Framework, but defined
// in an assembly whose version# changes with each minor release:

var dsContext = context;

EventInfo eventInfo = dsContext.GetType().GetEvent ("SendingRequest");
if (eventInfo != null)
{
    EventHandler<EventArgs> handler = (sender, e) =>
    {
        var pi = e.GetType().GetProperty ("Request");
        if (pi != null)
        {
            var request = pi.GetValue (e, null) as WebRequest;
            if (request != null)
                executionManager.SqlTranslationWriter.WriteLine (request.RequestUri);
        }
    };
    var typedDelegate = Delegate.CreateDelegate (eventInfo.EventHandlerType, handler.Method);
    eventInfo.GetAddMethod().Invoke (dsContext, new[] { typedDelegate });
}
```

Importing additional assemblies and namespaces

LINQPad imports **System.Data.Linq** by default, since LINQ to SQL is its default ORM. You might want to eschew this namespace and import your own namespaces (and assemblies) instead, by overriding the following methods:

```

/// <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 () { return null; }

/// <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 () { return null; }

/// <summary>Returns a list of additional assemblies to reference when building queries. Assemblies
/// should be named as they would when calling the compiler (including the .dll extension). Assemblies
/// in the same folder as the driver, or in the GAC don't require a folder name. Assemblies in other
/// locations should include the full path. 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 () { return null; }

```

Note: if `GetAssembliesToAdd` returns multiple dependent assemblies, return them in the order of least derived to most derived. This will ensure that autocompletion works without glitches.

Implementing `ICustomMemberProvider`

LINQPad's output window eagerly walks object graphs, and this makes it slow to display entities with lazily evaluated navigation properties. And if the walked sub-entities themselves contain lazy navigation properties, it can take forever!

The standard solution is for users to [implement `ICustomMemberProvider`](#) in their entities, stripping out the lazily evaluated properties. However, this is inconvenient and must be implemented centrally to be practical.

With a custom data context driver, you can instead 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;
}

```

Simply return null if `objectToWrite` is not an entity whose output you want to customize. Otherwise, return any object that with a suitable implementation of `ICustomMemberProvider` for that object.

You can identify entities via attributes or by looking for a base type, depending on your ORM. 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.

Overriding PreprocessObjectToWrite

From LINQPad 2.12, you can go even further in controlling how objects are displayed by overriding **PreprocessObjectToWrite**:

```

/// <summary>This lets you replace any non-primitively-typed object with another object for
/// display. The replacement object can optionally implement ICustomMemberProvider for further
/// control of output formatting.</summary>
public virtual void PreprocessObjectToWrite (ref object objectToWrite, ObjectGraphInfo info) { }

```

This method is called 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), and it can implement **ICustomMemberProvider** if you want to dynamically control the properties it has.

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

```
objectToWrite = info.DisplayNothingToken;
```

An example of when you might do this is if writing a driver for Reactive Framework. When a user dumps an `IObservable<T>`, you’d want to subscribe to the observable and have your subscription methods Dump output rather than emitting output there and then.

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 (ADO.NET 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 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.

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

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 the LINQPad folder under the following directory:

```
Environment.SpecialFolder.LocalApplicationData
```

For Windows 7 and Vista, this is normally:

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

For Windows XP:

```
C:\Documents and Settings\UserName\Local Settings\Application Data\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 .NET Reflector to examine the LINQPad-generated query:

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

or the typed data context class:

```
Process.Start (@":\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 classes 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(50). In general, this should be set to true with all items of kind
    /// IQueryableObject 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;
}
```