GETTING THE MOST FROM LINQPAD

SUCCINCTLY

BY JOSÉ ROBERTO OLIVAS MENDOZA

SUCCINCTLY EBOOK SERIES

Syncfusion®

www.dbooks.org

Getting the Most from LINQPad Succinctly

By
José Roberto Olivas Mendoza

Foreword by Daniel Jebaraj



Copyright © 2019 by Syncfusion, Inc.

2501 Aerial Center Parkway
Suite 200
Morrisville, NC 27560
USA
All rights reserved.

Important licensing information. Please read.

This book is available for free download from www.syncfusion.com on completion of a registration form.

If you obtained this book from any other source, please register and download a free copy from www.syncfusion.com.

This book is licensed for reading only if obtained from www.syncfusion.com.

This book is licensed strictly for personal or educational use.

Redistribution in any form is prohibited.

The authors and copyright holders provide absolutely no warranty for any information provided.

The authors and copyright holders shall not be liable for any claim, damages, or any other liability arising from, out of, or in connection with the information in this book.

Please do not use this book if the listed terms are unacceptable.

Use shall constitute acceptance of the terms listed.

SYNCFUSION, SUCCINCTLY, DELIVER INNOVATION WITH EASE, ESSENTIAL, and .NET ESSENTIALS are the registered trademarks of Syncfusion, Inc.

Technical Reviewer: James McCaffrey

Copy Editor: Courtney Wright

Acquisitions Coordinator: Tres Watkins, content development manager, Syncfusion, Inc.

Proofreader: Graham High, senior content producer, Syncfusion, Inc.

www.dbooks.org

Table of Contents

The Story behind the *Succinctly* Series of Books 7

About the Author 9

Who Is This Book For? 10

Chapter 1 A Quick Tour of LINQPad 11

What is LINQPad? 11

LINQPad specifications 11

Getting LINQPad 5 12

LINQPad 5 installation 12

How LINQPad is deployed in user's computer 12

LINQPad user interface 13

LINQPad basics: Queries 14

Setting up the queries and external projects location 15

Chapter summary 15

Chapter 2 LINQPad and Entity Framework 17

Creating an ADO.NET Entity Data Model 17

Connecting to an Entity Data Model 27

Creating an Entity Data Model connection 27

A few highlights about Entity Framework connections 32

Chapter summary 33

Chapter 3 LINQPad Scripting 34

Lprun command-line options 34

Output formatting 35

Output redirection 39

Error handling 40
Compilation options 40
Plain-text scripts 41
Managing connections 42
Managing assemblies and namespaces 46
Passing arguments to scripts 47
The CMD Boolean symbol 47
Calling a script from another script 48
Chapter summary 49

Chapter 4 LINQPad Extensibility 51

Writing custom extensions 51
String and DateTime extensions sample 52
Enumerating the properties and values for an object 54
Packaging custom extension methods as a plugin 56
Creating a custom visualizer 58
Displaying rows from a LINQ to SQL entity in a grid view 59
Stacking WPF elements 64
The DumpContainer object 68
Chapter summary 70

Chapter 5 Custom Data Context Drivers 72

What is a data context driver? 72
Why is writing custom data context drivers useful? 72
Data context drivers from the user's perspective 73
Basic steps for writing a data context driver 75
Special terms and conditions 76
Basic concepts about data context drivers 76

www.dbooks.org

Connection 76
Typed data contexts 76
Dynamic and static drivers 77
How LINQPad works with queries 78
Writing a data context driver 79
Setting up a project 79
The header file 80
Making room for dynamic and static context drivers 83
Adding necessary references 83
Writing the driver 84
Deploying the data context driver 98
Installing the data context driver 98
Testing our static data context driver 100
Selecting a custom assembly 101
Creating the connection and viewing the entities 103
Displaying the custom driver description 103
Executing a query 103
Debugging the driver 105
Exception logging 105
Credits 110
Chapter summary 110
Conclusion 112

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

www.dbooks.org

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!





About the Author

I'm an IT businesses entrepreneur, a software developer, and a huge technology fan. My company went to market in 1990, focused mainly in custom software development. We started with COBOL as our main programming language, and we've evolved along the years up to .NET and Microsoft Office technologies. We recently started to adopt some of Microsoft's multiplatform technologies, such as .NET Core. Throughout these years, I have been researching cutting-edge technologies and searching for new tools that can help us automate processes for our products' development lifecycles.

In 2010, electronic invoicing was set as a requirement for business transactions in Mexico. This event forced my company to upgrade all our business-related products within a six-month window. At that time, the effort of our development team was huge. Fortunately, we delivered all our products right on time.

Electronic invoicing has evolved substantially since 2010, and the continuous changes have forced us to enhance our development process. Continuous delivery is very important for us because the Mexican tax authorities have been publishing improvements to electronic invoicing several times per year.

We put LINQPad into action in 2015, as a solution for testing our .NET reusable libraries related to electronic invoicing. These tests were made with small pieces of code that were written and executed directly in LINQPad, making it inefficient to create Visual Studio projects for testing purposes. As a result, testing for those libraries was made in a fraction of the time, compared to our previous effort.

As a huge tech fan, I've been exploring the product since we started to use it. Its LINQ support allowed our development team to improve their skills in the language. As a result, the amount of code for our projects was reduced by almost 30 percent. And since LINQPad can be used as a scratchpad, pieces of code can be tested very quickly, and we can gather these pieces to make an entire project.

LINQPad offers awesome features that make it a powerful code-testing tool. The purpose of this book is to explain these features and make it possible for you to apply them in your development process.

www.dbooks.org

Who Is This Book For?

This book is for all developers who work in the .NET platform (C# or VB.NET) and want to accelerate the development lifecycle by using LINQPad as a better, more powerful testing tool. Basic knowledge of LINQPad is required. If you're not familiar with the product, I suggest reading my previous ebook, *LINQPad Succinctly*, which you can download here. C# programming skills and basic knowledge of LINQ are also recommended.

The book starts with a brief overview of LINQPad and its top features. The next chapter focuses on using Entity Framework with LINQPad to get a database model testing tool. After that, the book explains how to get the most from LINQPad by using scripts and extensibility features. Finally, the last chapter of the book explains how to build a custom data context driver to be used by LINQPad.

The LINQPad 5 free edition and the C# programming language are used for the purposes of this book. Visual Studio 2017 will be employed to write some of the code examples, with the .NET Framework 4.6 being the target framework for building assemblies. Microsoft SQL Server Management Studio will be used for database treatment. All samples described in this book can be downloaded here.

I hope that you enjoy this material, and that it helps you improve your development lifecycle and deliver your applications in less time.

Chapter 1 A Quick Tour of LINQPad

What is LINQPad?

LINQPad is a free software utility developed by Joseph Albahari to be used as part of .NET application development. It's used to interactively query SQL databases using LINQ, as well as to interactively write C# code without needing an IDE such as Visual Studio. This feature expands its use as a "test tool" where C# code can be quickly prototyped outside the IDE. It also supports the VB.NET, SQL, and F# languages. LINQPad supports the following dialects:

- Entity Framework
- LINQ to Objects
- LINQ to SQL
- LINQ to XML

There are several versions of the product; for the purposes of this book, we will focus on LINQPad 5 version 5.36.03, which is the current stable release at the time of writing.

LINQPad specifications

LINQPad offers a series of benefits to the user, such as:

- Full support for C# 6 and later, and VB 14 in LINQPad 5.
- Reporting of execution time in its status bar, allowing performance testing.
- Cloning of code snippets with a keyboard shortcut (Ctrl+Shift+C), allowing their execution side by side.
- Custom extension writing, which allows users to code methods and classes to make them available for all LINQPad queries.
- Custom assembly referencing.
- Building typed data contexts on the fly, including drivers and formatters for SQL Server, SQL Azure, SQLCE, Oracle, SQLite, and MySQL.
- Object rendering with the embedded **`Dump`** method.
- Access control for the **`Dump`** method in users' custom types.
- Full command-line experience with the lprun.exe utility.
- Support for writing custom visualizers.
- Support for presentation mode, allowing use of cordless presenter devices.



Note: A detailed explanation of LINQPad versions and other features can be found in my ebook LINQPad Succinctly.

www.dbooks.org

Getting LINQPad 5

You can download LINQPad 5 here.

LINQPad 5 installation

Once you've downloaded LINQPad 5, double-click the setup program file **LINQPad5Setup.exe** to open it. For most users, this setup program can be found in the **Downloads** folder.

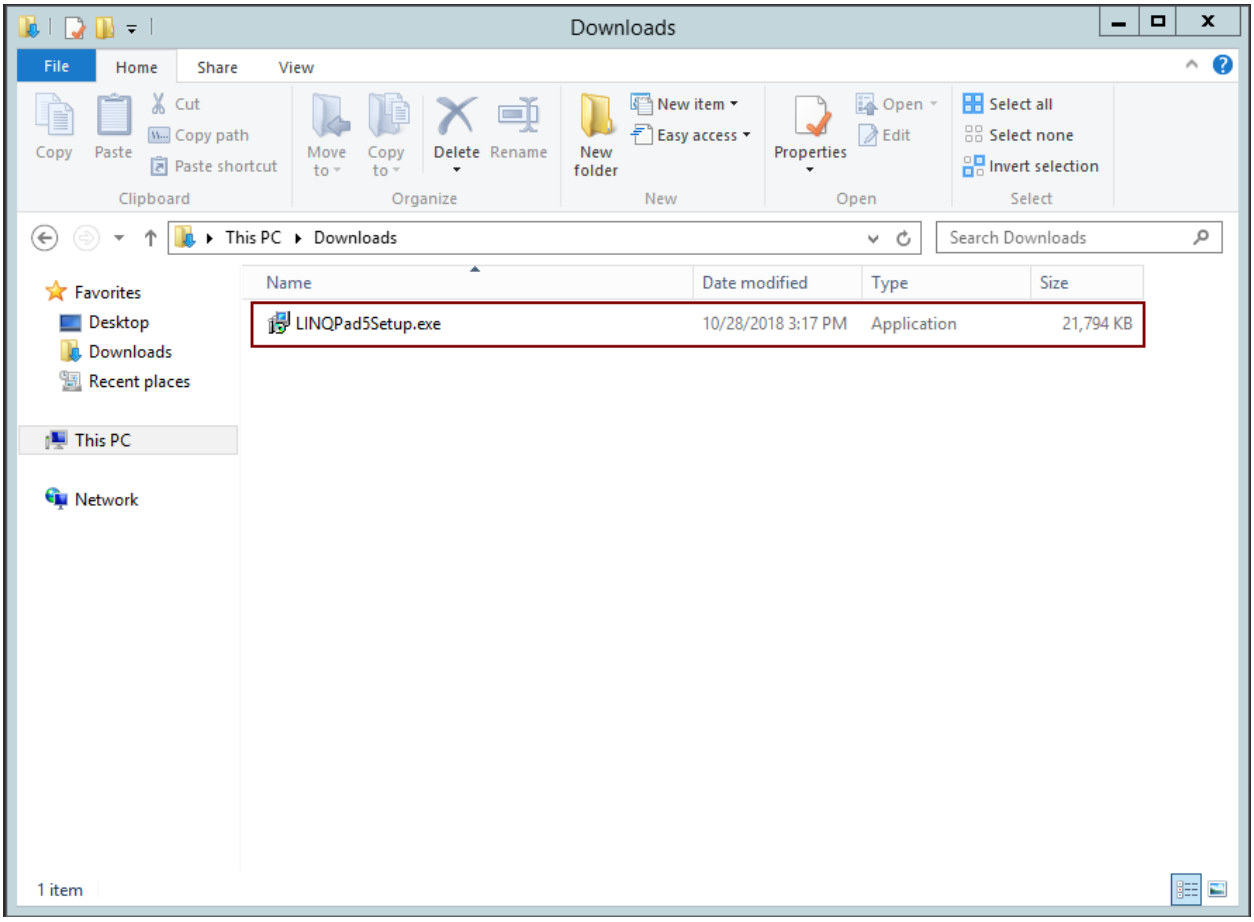


Figure 1: LINQPad Installation Program in User's Downloads Folder

How LINQPad is deployed in user's computer

After LINQPad is installed, a set of folders and files is created. These folders and files hold all the necessary elements that make LINQPad work properly, and are displayed in the following table.

Table 1: LINQPad folder structure.

Folder	Purpose
Documents\LINQPad Queries	Holds all queries created by the user. LINQPad uses this folder to display those queries in a tree view named My Queries.
Documents\LINQPad Plugins	Holds any custom assembly referenced by all or some queries.
Documents\LINQPad Snippets	Holds custom code snippets (doesn't apply for free edition).
%LocalAppData%\LINQPad\drivers	Holds custom data context drivers which can be used in all queries.
%AppData%\LINQPad\ConnectionsV2.xml	Stores all connections that appear in the Connections tree view.
%AppData%\LINQPad\DefaultQuery.xml	Stores all assembly references that will be used by default in queries.
%AppData%\LINQPad\RoamingUserOptions.xml	Stores the user preferences that dictate LINQPad behavior.



Note: You can override the LINQPad default storage locations by creating a directory structure in the root install location of the LINQPad executable (which is *C:\Program Files(x86)\LINQPad5* ***by default). For example, if you create a directory*** *C:\Program Files(x86)\LINQPad5\Documents\LINQPad Queries* ***and save queries there, when you ask LINQPad to open a query, it will search the new directory before searching the default directory.***

LINQPad user interface

The LINQPad user interface is where you will do most of your work. For the purposes of this book, we'll identify 14 graphic elements of this interface. These elements are:

1. **LINQPad File menu**: Holds all available LINQPad commands for working with the program.
2. **Connection's tree view area**: Shows all connections created by the user with the connection manager.
3. **Queries and Samples tree view area**: Shows all queries and extensions created by the user. It also shows all available samples within LINQPad, which are displayed in a separate tab.
4. **Query code tab window area:** Where the user writes query code.
5. **Query code tab caption**: Shows the name of the query being edited.

www.dbooks.org

6. **Add New Tab button**: Adds a new tab for query editing.
7. **Execute Query button**: Executes the query that is currently selected.
8. **Stop Execution button**: Stops query execution.
9. **Results View Type button**: Allows the user to switch between Rich Text and Data Grid for viewing query results.
10. **Language Selector combo box**: Allows the user to select the programming language in which queries will be written.
11. **Connection Selector combo box**: Allows the user to select a previously created custom connection.
12. **Close Tab button**: Closes the tab currently selected.
13. **Premium features activation link**: In the free version of LINQPad, this link allows the user to open the product's webpage to acquire a license for activating the paid license's features.
14. **Status Bar**: Shows several messages sent by the program, including query's time performance.

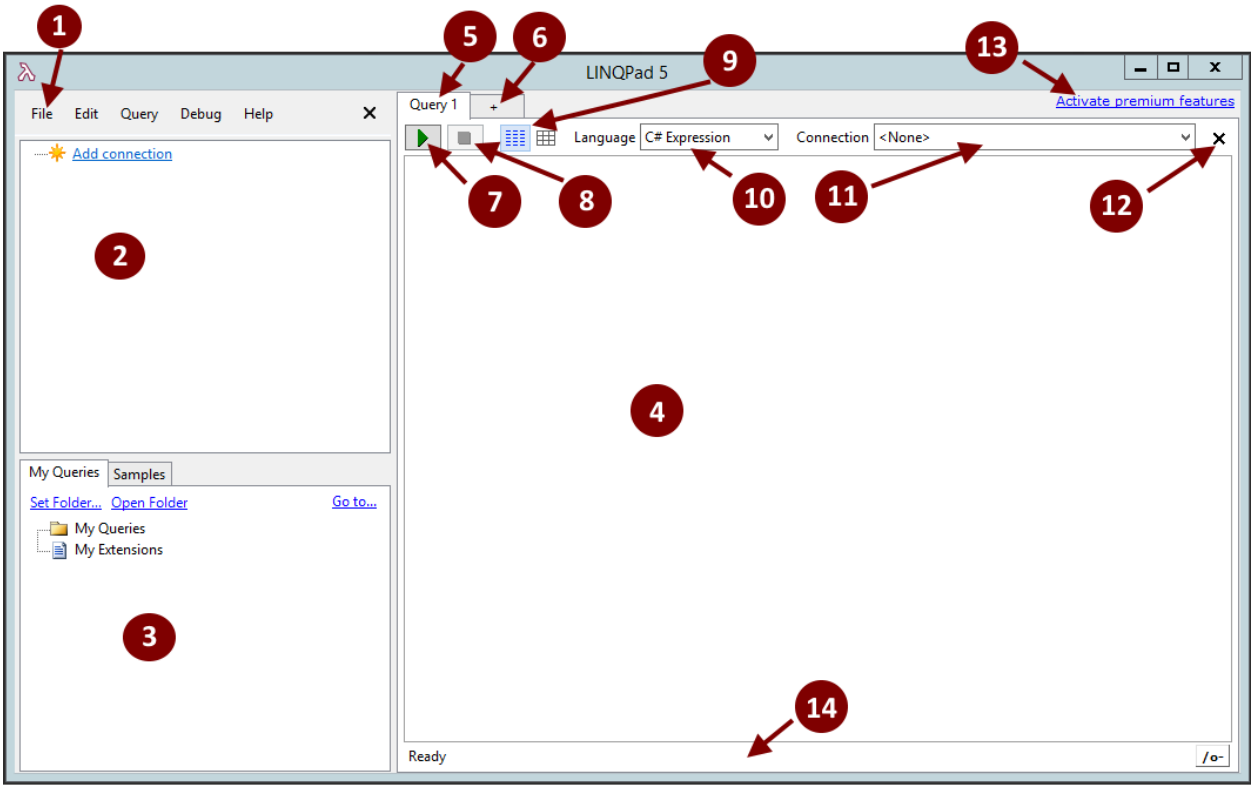


Figure 2: LINQPad User Interface and Its Elements

LINQPad basics: Queries

LINQPad uses the term *query* for every piece of code that is written in the editor. So from now on, this word will be used within this book for pointing to every code sample displayed in it.

We can save every one of these pieces of code for future use. By default, queries are saved in the **LINQPad Queries** folder located in the current user's **Documents** folder. Also, after a query is saved, its name appears in the Queries and Samples tree view area of the user interface.

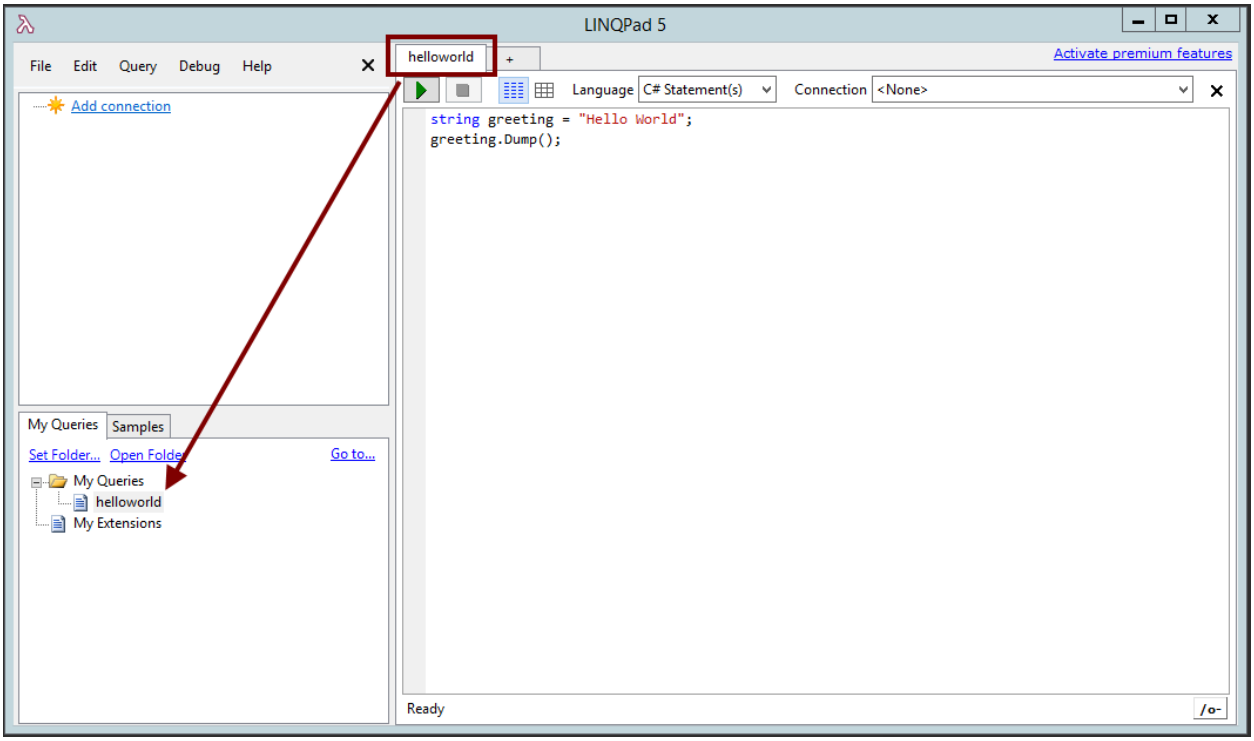


Figure 3: A LINQPad Query in the Queries and Samples Tree View Area

Setting up the queries and external projects location

For the purposes of this book, every query written in LINQPad will be saved in a folder named **LINQPad Samples**. In my case, this folder will be located at D:\, but you can place this folder in any convenient directory.

Chapter summary

LINQPad is a software utility developed by Joseph Albahari to be used as a part of .NET application development. It is used for interactively querying SQL databases using LINQ, as well as interactively writing C# code without the need for an IDE such as Visual Studio.

LINQPad supports the VB.NET, SQL, and F# languages. It also supports Entity Framework, LINQ to Objects, LINQ to SQL, and LINQ to XML dialects.

There are several versions of LINQPad, but for the purposes of this book, we will focus on the LINQPad 5 free version 5.36.03, which is the latest stable version according to the product's website at the time of this writing.

www.dbooks.org

LINQPad 5 can be downloaded here.

LINQPad 5 can be installed by downloading the product from the LINQPad website and double-clicking the setup program's icon. Most users can find the installation program file in the **Downloads** folder.

After LINQPad is installed, a set of folders and files is created in order to make sure LINQPad works properly. These folders and files hold all elements needed by LINQPad, such as queries, plugins, code snippets, data context drivers, connections, default references, and user preferences.

The LINQPad user interface is where you will do most of your work. We have identified fourteen graphic elements in the LINQPad user interface for the purposes of this book. These elements are: the LINQPad main menu, Connection's tree view area, Queries and Samples tree view area, Query code tab window area, Query code tab caption, Add New Tab button, Execute Query button, Stop Execution button, Results View Type button, Language Selector combo box, Connection Selector combo box, Close Tab button, Premium Features activation link, and Status Bar.

LINQPad uses the term *query* for every piece of code that is written in the editor and allows us to save this code onto disk. LINQPad stores queries in a folder named **LINQPad Queries** by default, located into the user's **Documents** folder. All queries saved are displayed beneath a tree view node called **My Queries**.

For the purposes of this book, every query written in LINQPad will be saved in a folder named **LINQPad Samples**.

Chapter 2 LINQPad and Entity Framework

LINQPad lets the user query Entity Framework models defined in Visual Studio. The process used to accomplish this action will be explained in this chapter. The examples in this chapter assume you have an existing SQL Server database named **`uspostalcodes`**. If you wish to run the examples, you can find the SQL scripts to create and populate the database here.

Creating an ADO.NET Entity Data Model

To explore the ability of LINQPad to query Entity Framework models, we need to create an ADO.NET Entity Data Model project in Visual Studio. To do this, open Visual Studio and select **New Project** from the **File** menu. Next, select the **Class Library** project type, as displayed in the following figure.

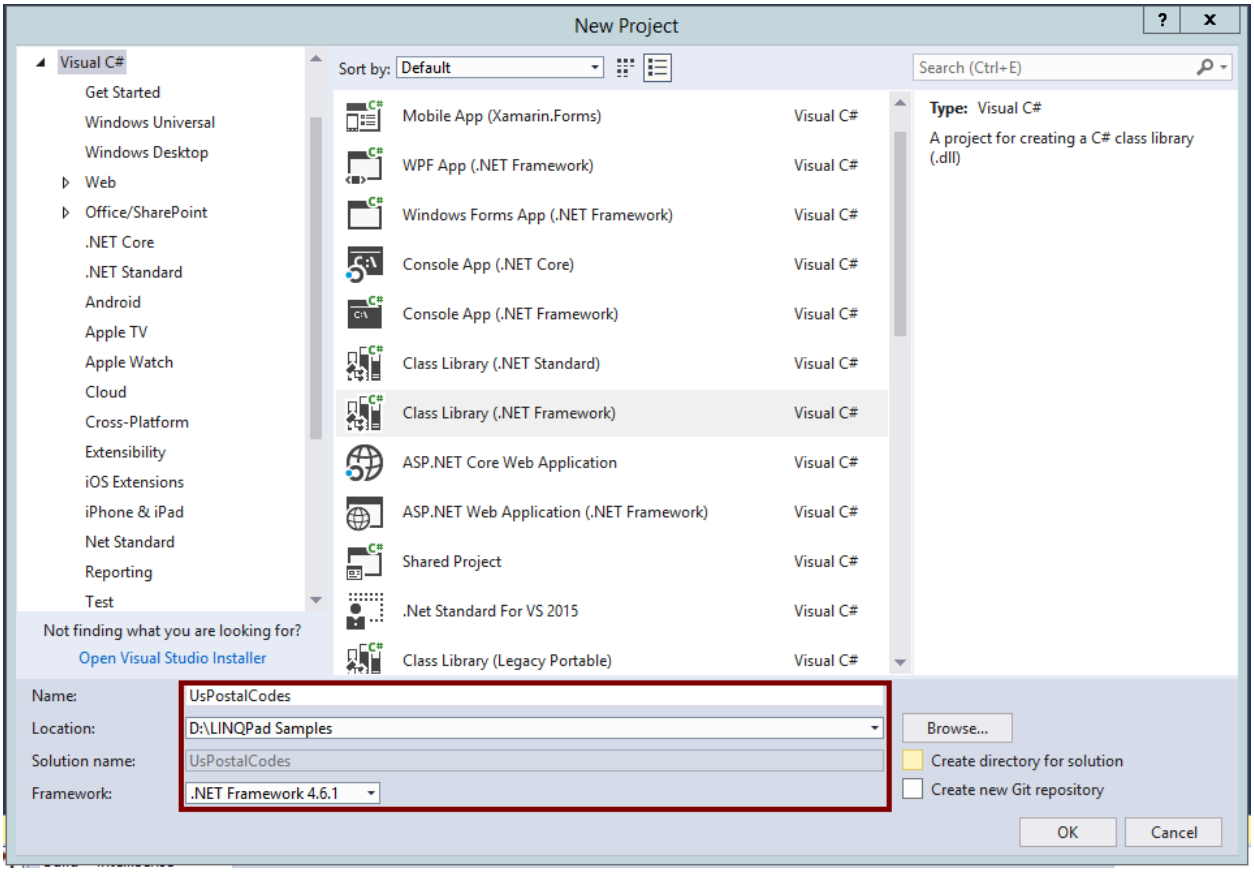


Figure 4: The UsPostalCodes Class Library Project

The project will be named UsPostalCodes, which is the name of the database that will be used to create the model. This project should be saved in the **D:\LINQPad_Samples** folder created for saving the samples discussed in this book.

www.dbooks.org

Once the project is created, right-click the project's name in the Solution Explorer and select **Add** > **New Item**.

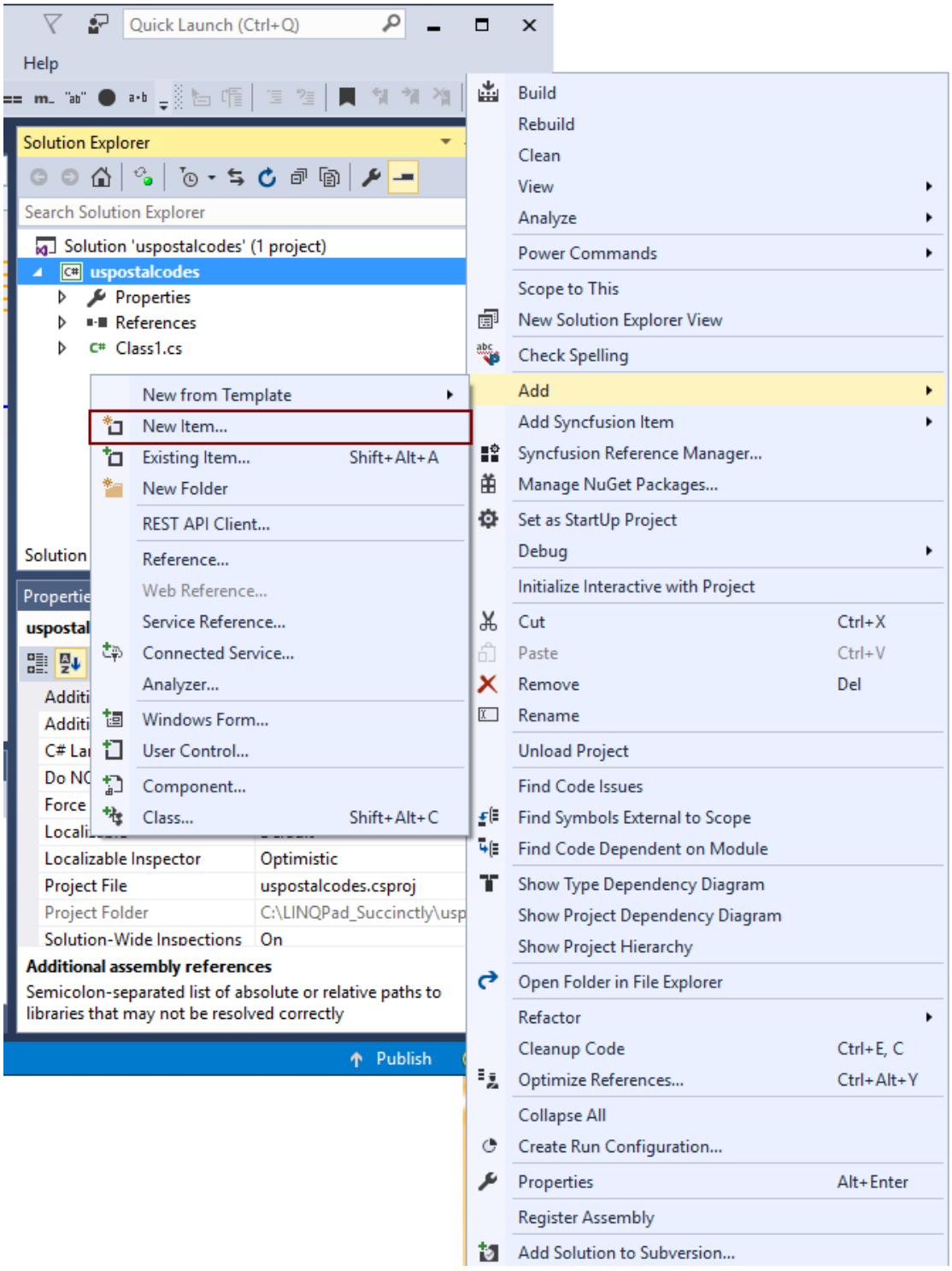


Figure 5: The New Item Command

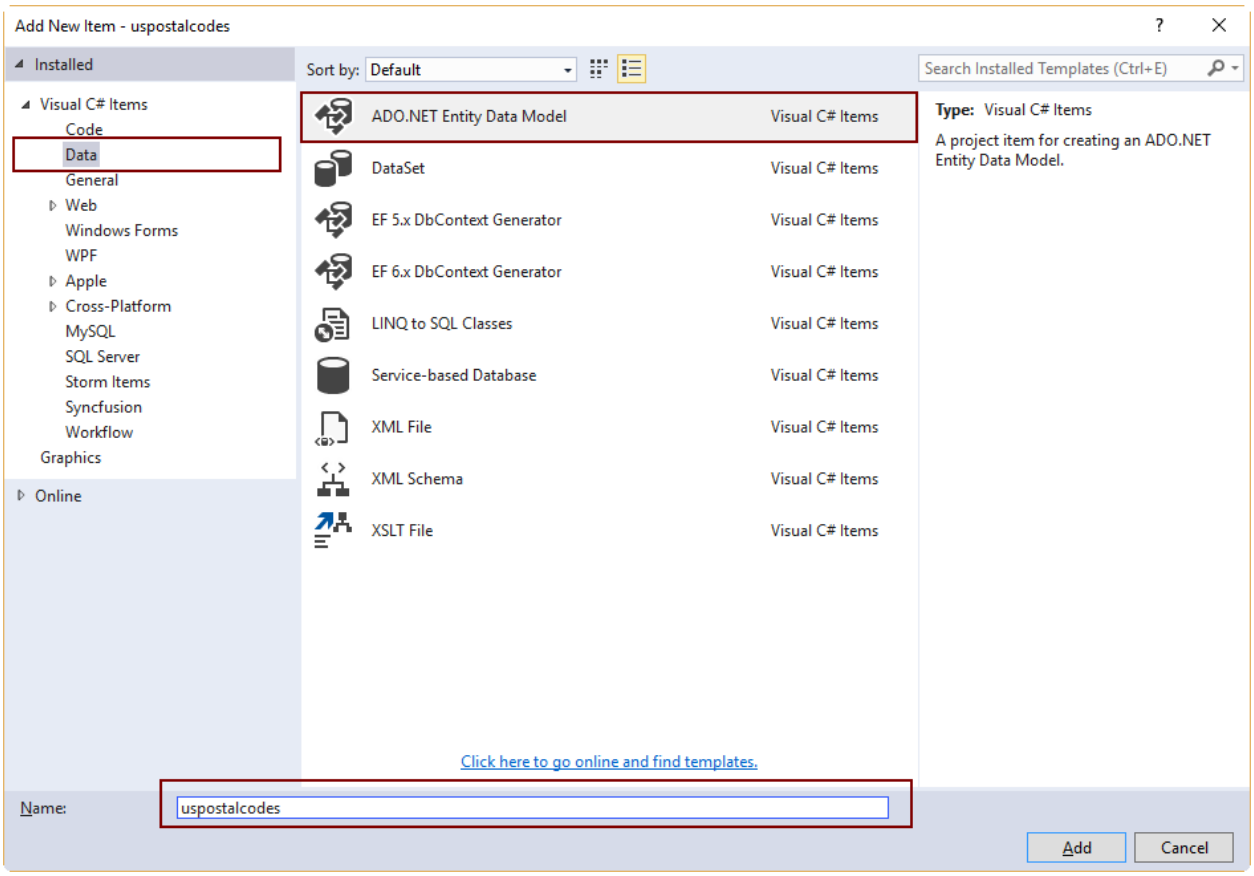


Figure 6: The Add New Item Dialog Box

In the **Add New Item** dialog box, select **Data** from the template groups list located on the left. Then, select **ADO.NET Entity Data Model** from the templates displayed at the right. The name for the model will be **uspostalcodes**, as will the database and project names. Click **Add** to continue.

www.dbooks.org

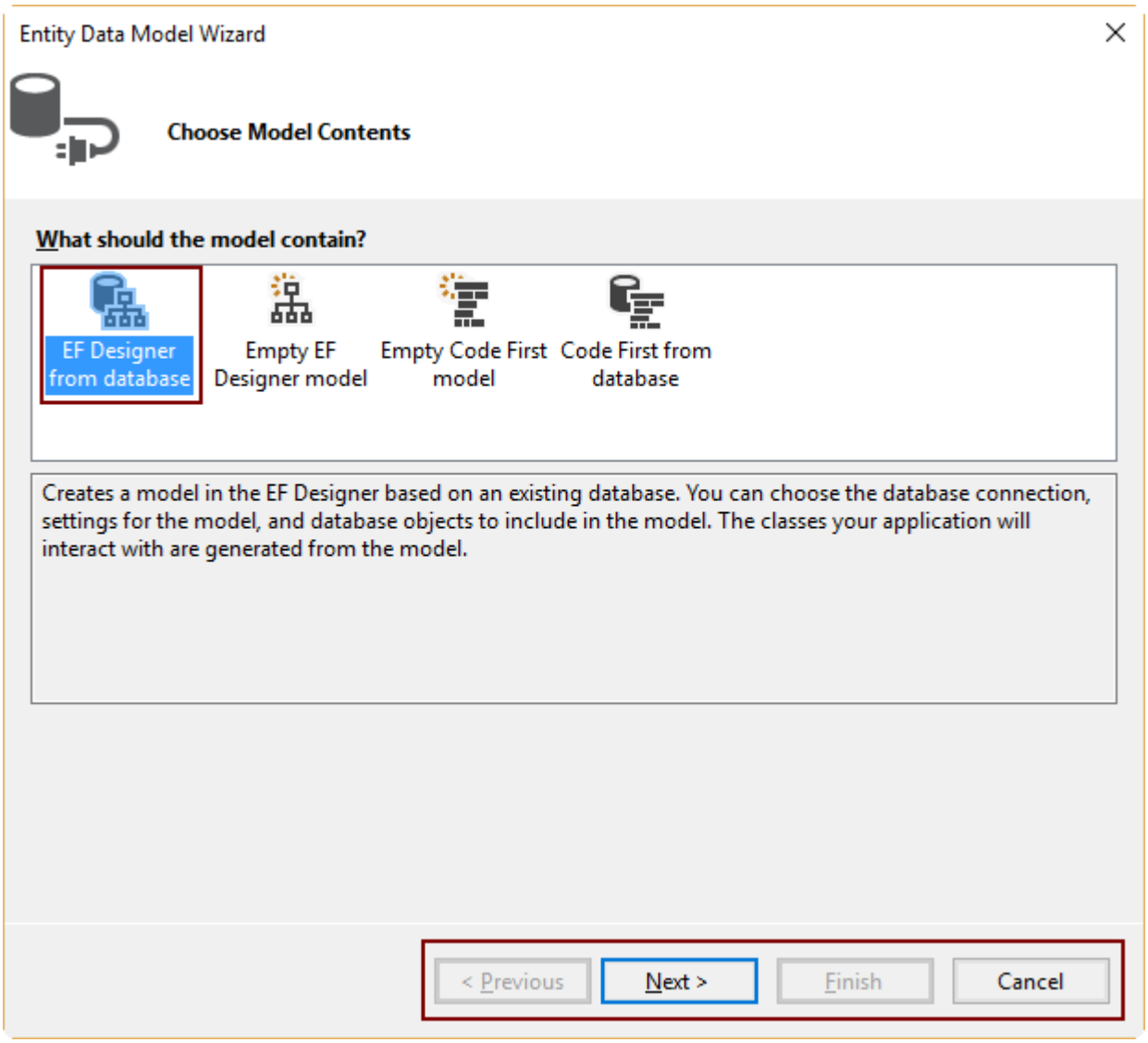


Figure 7: The Entity Data Model Wizard

The Entity Data Model Wizard dialog will appear and will ask for the contents of the model. For the purposes of this project, the model contents will be **EF Designer from database**. A brief explanation about this kind of content appears in a frame located under the content options. After selecting **EF Designer from database**, click **Next** to continue.

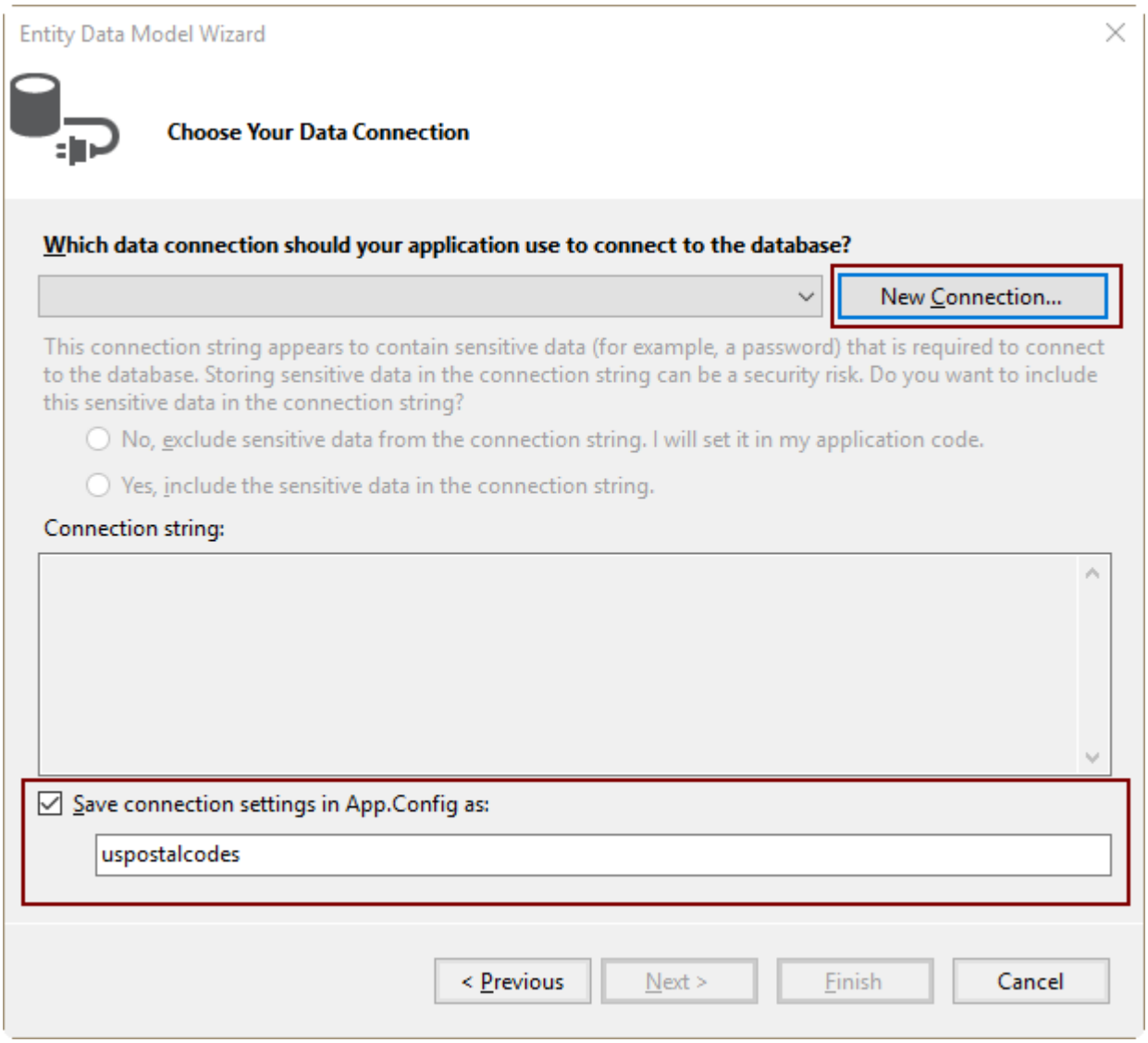


Figure 8: Choosing the Data Connection

Now, the wizard will ask about the data connection that will be used to generate the model. Since a connection isn't defined yet, click **New Connection**. Then, enter the name **uspostalcodes** in the text box that will be displayed in order to save the connection settings in the App.Config file.

www.dbooks.org

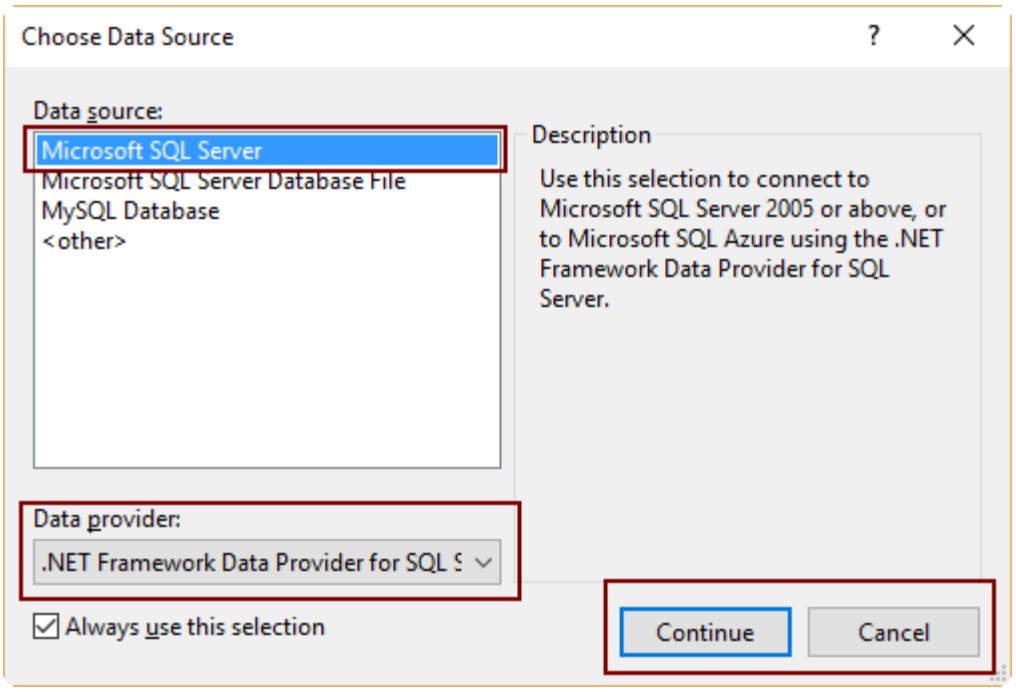


Figure 9: Choose Data Source Dialog Box

The next step is choosing the data source for the connection. In this case, we'll use **Microsoft SQL Server** and the **.NET Framework Data Provider for SQL Server**. Click **Continue** to proceed.

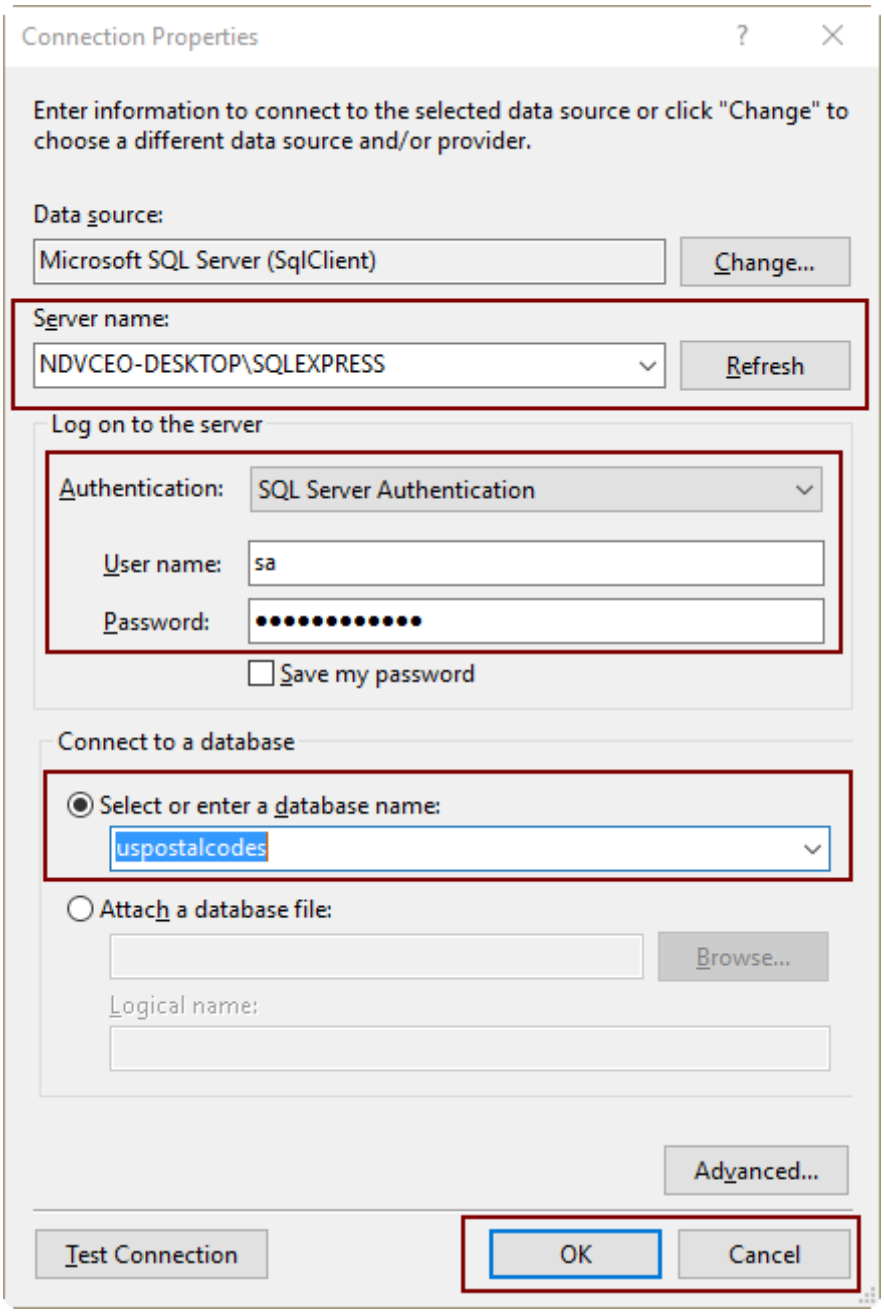


Figure 10: Connection Properties Dialog Box

Now we will set the connection properties. In this case, the connection will use an instance of SQL Server installed on the local computer, with **SQL Server Authentication** mode and the **uspostalcodes** database. Click **OK** to continue the process.

www.dbooks.org

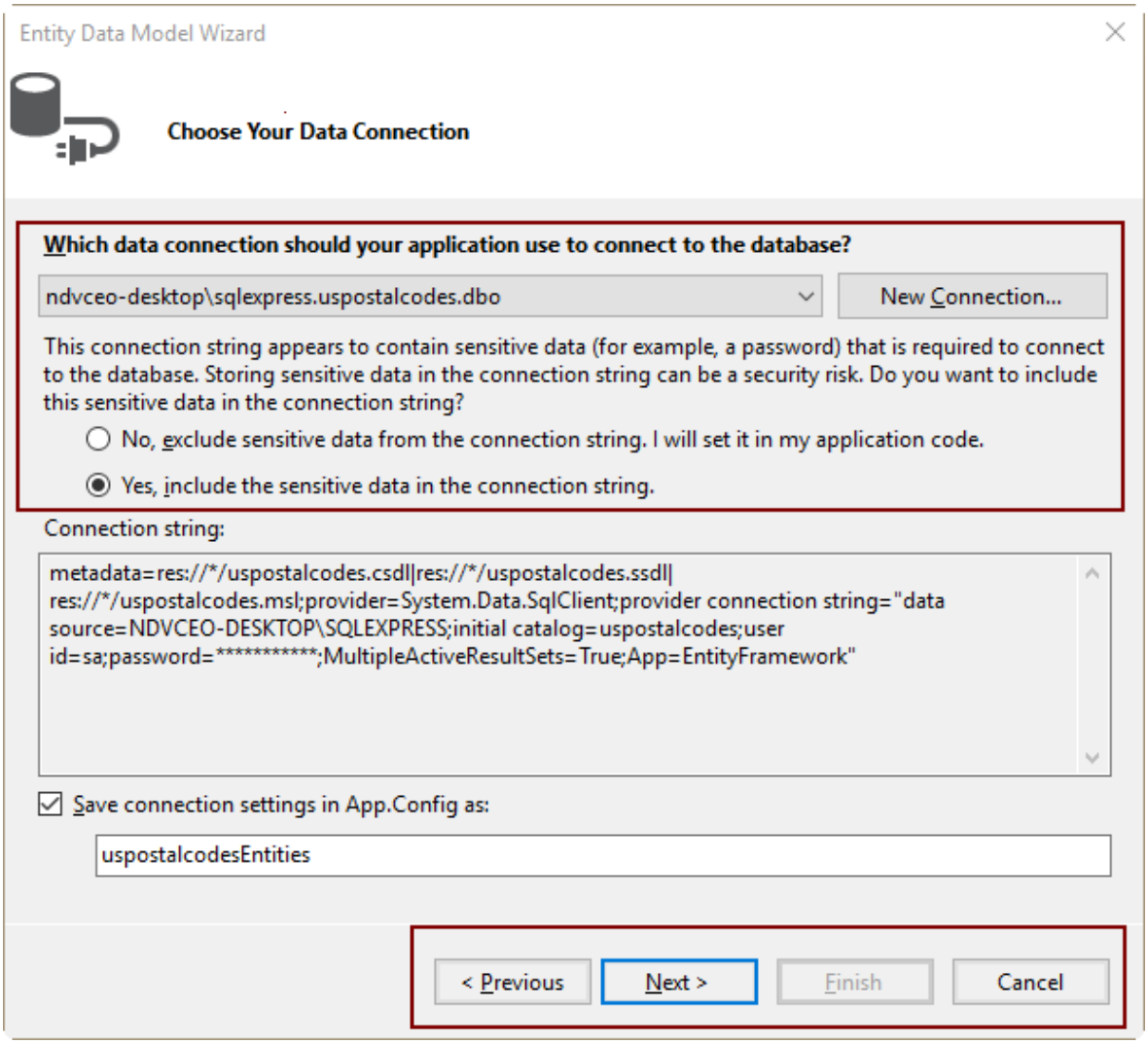


Figure 11: Entity Data Model Wizard Displaying the Connection String

The Entity Data Model Wizard will display the name of the connection selected, and the connection string that will be used to access the database. A group of option buttons and descriptive text appear under the name of the connection. The text suggests that sensitive data (such as the password to connect to the server) shouldn't be included in the connection string. This is highly recommended for production environments, but for the purposes of this book, you can choose to include password information. Click **Next** to continue the process.

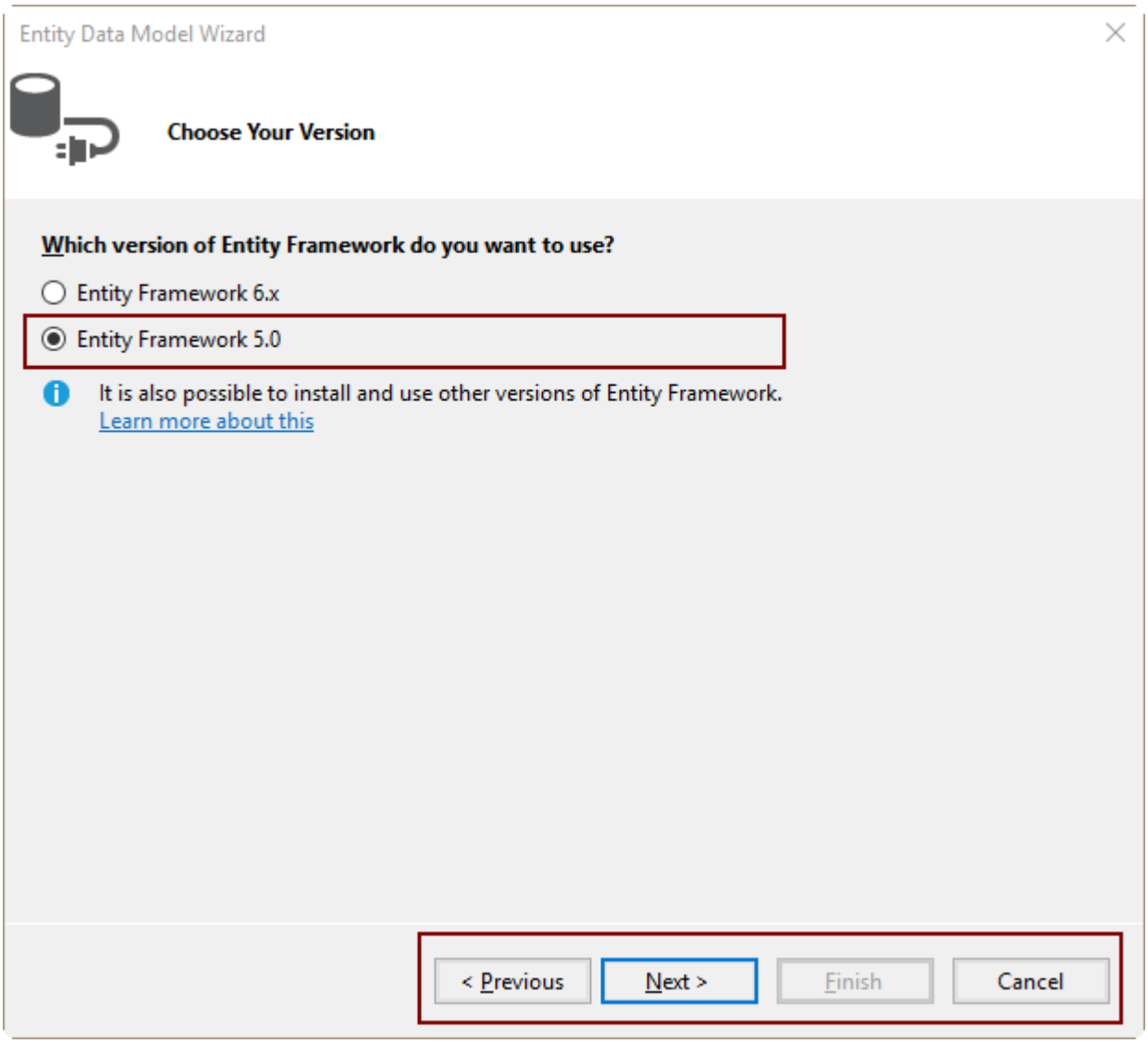


Figure 12: Selecting Entity Framework Version

The next dialog box asks for the version of Entity Framework to be used for model building. Choose **Entity Framework 5.0** for this project.

www.dbooks.org

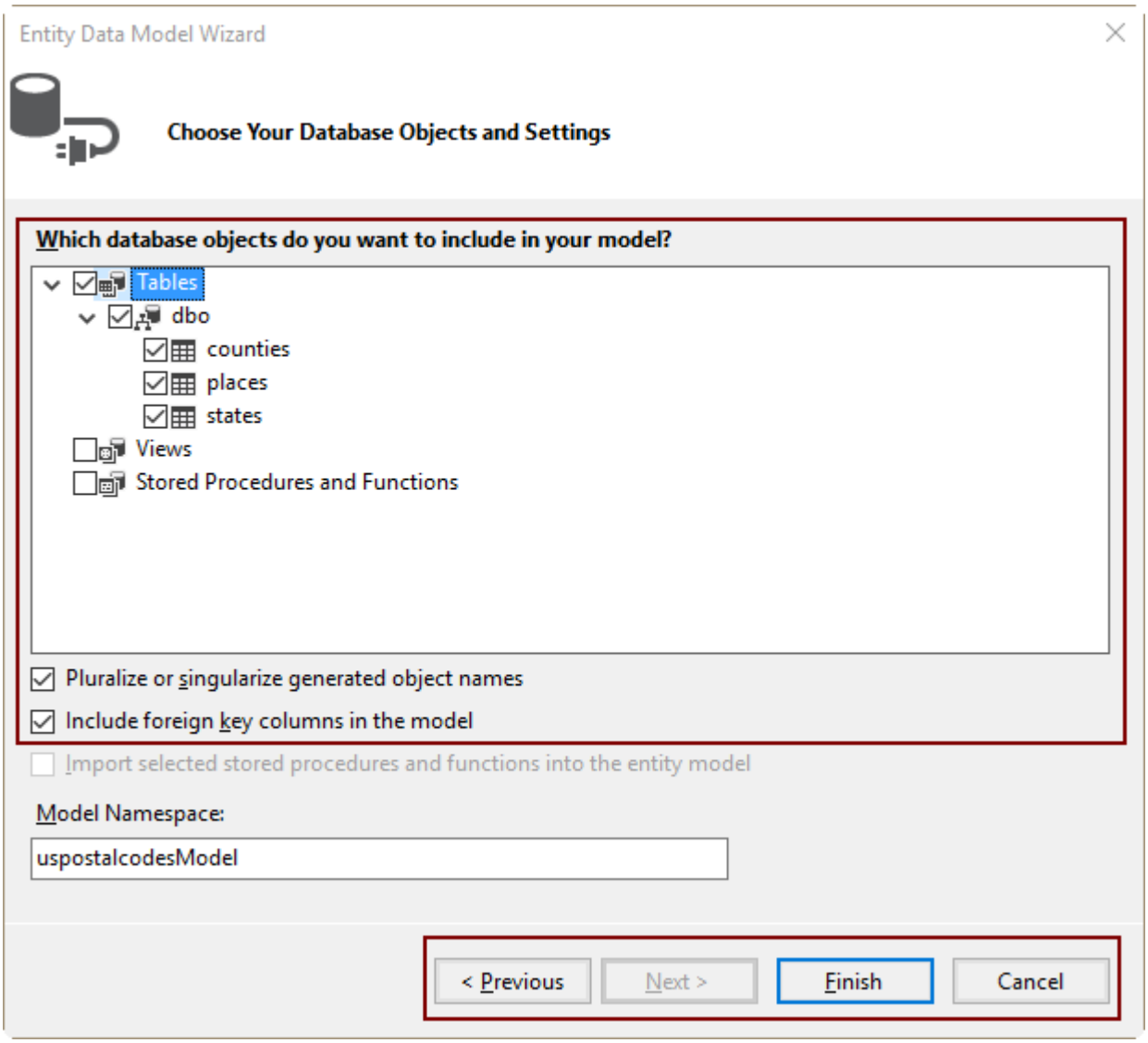


Figure 13: Choosing Database Objects

The wizard asks us to choose our database objects and settings to include in the model. This project will use the three tables that are part of the database (counties, places, and states). As in LINQPad connections, the wizard can pluralize object names, and include the foreign key columns in the model. Both actions are checked by default. To start model building, click **Finish**.

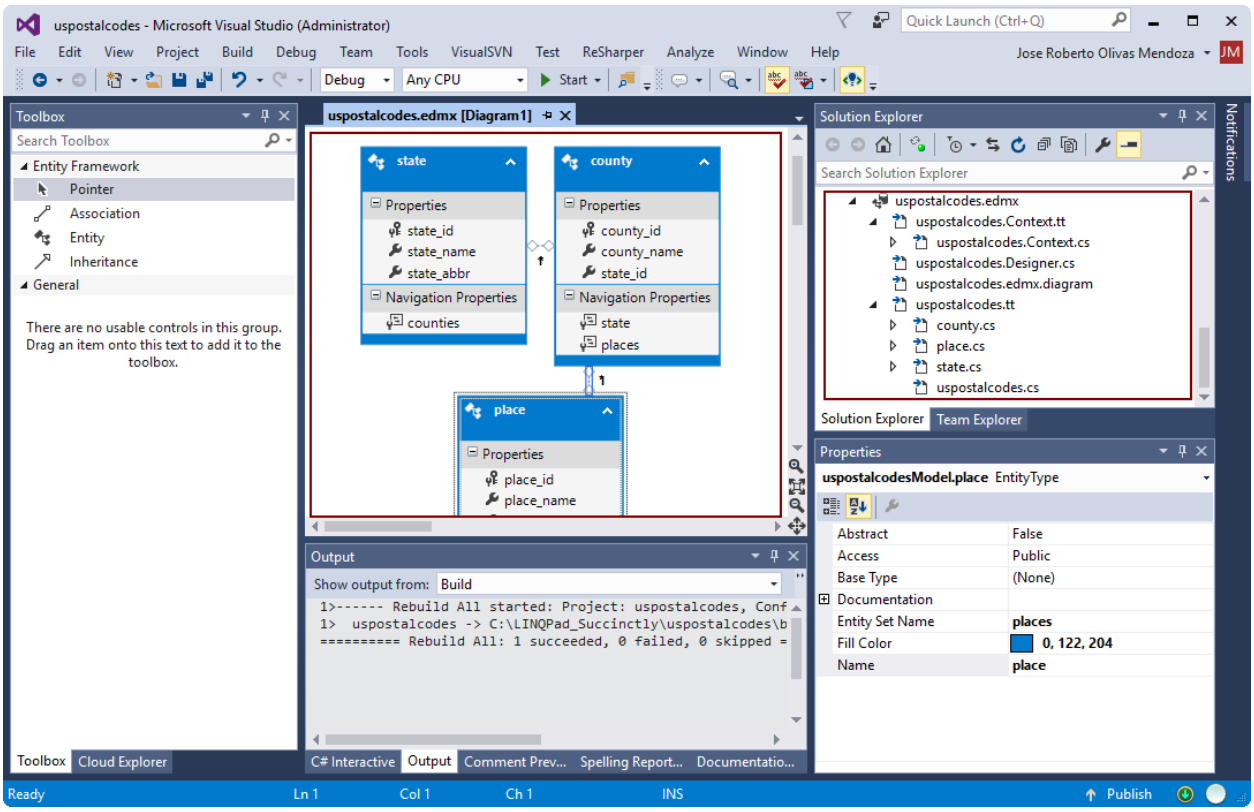


Figure 14: The Model Built as Part of the uspostalcodes Project

When the model finishes building, it is added to the project, appearing as a diagram, and all its components are displayed in the Solution Explorer.

The final step is building the assembly. To do this, click **Build** > **Build Solution** or **Rebuild Solution** in the Visual Studio menu bar.

Connecting to an Entity Data Model

LINQPad can connect to an Entity Data Model by adding a connection, just like LINQ to SQL connections.

Creating an Entity Data Model connection

Start by launching LINQPad, and then click the **Add Connection** hyperlink to bring up the Choose Data Context dialog.

www.dbooks.org

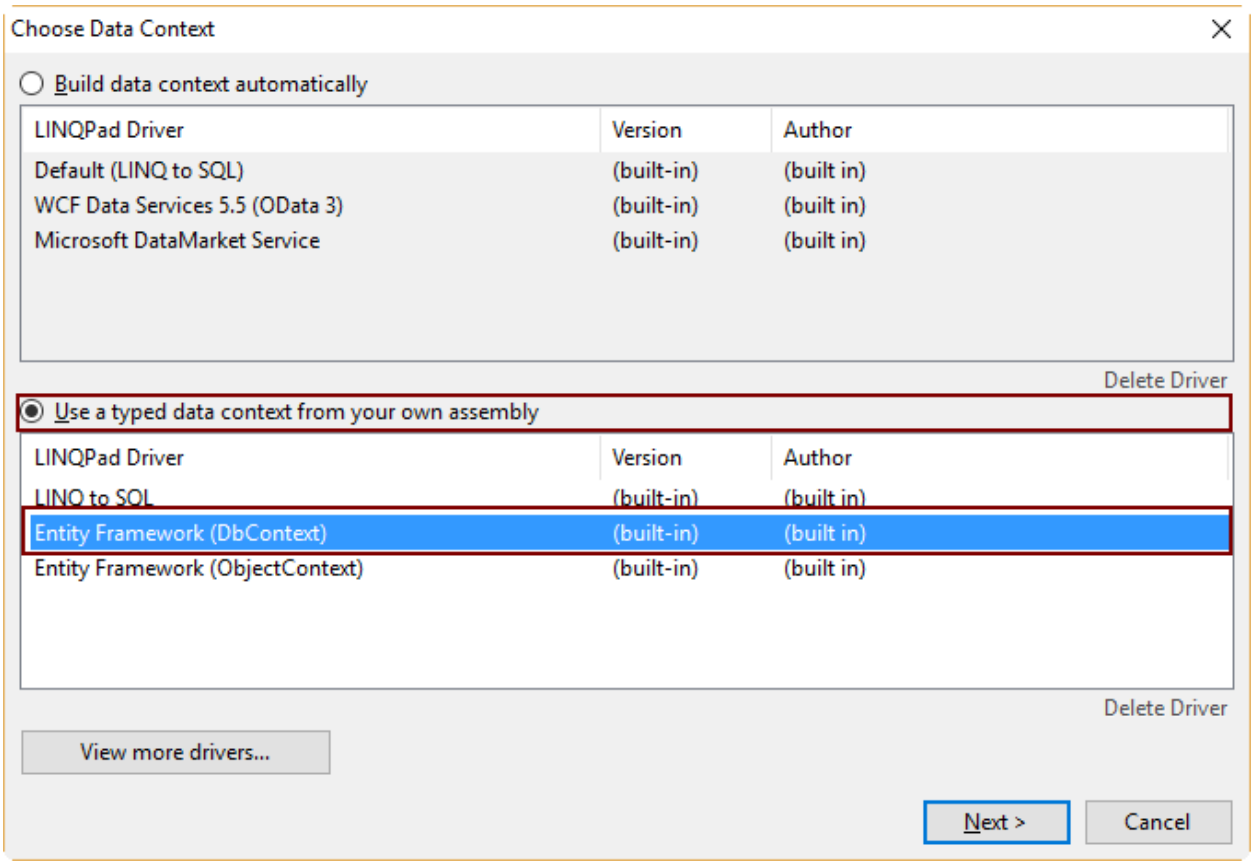


Figure 15: The Choose Data Context Dialog Settings to Query an Entity Framework Model

Select the **Use a typed data context from your own assembly** option, and then select the **Entity Framework (DbContext)** option from the list displayed under the option button. Click **Next** to continue.

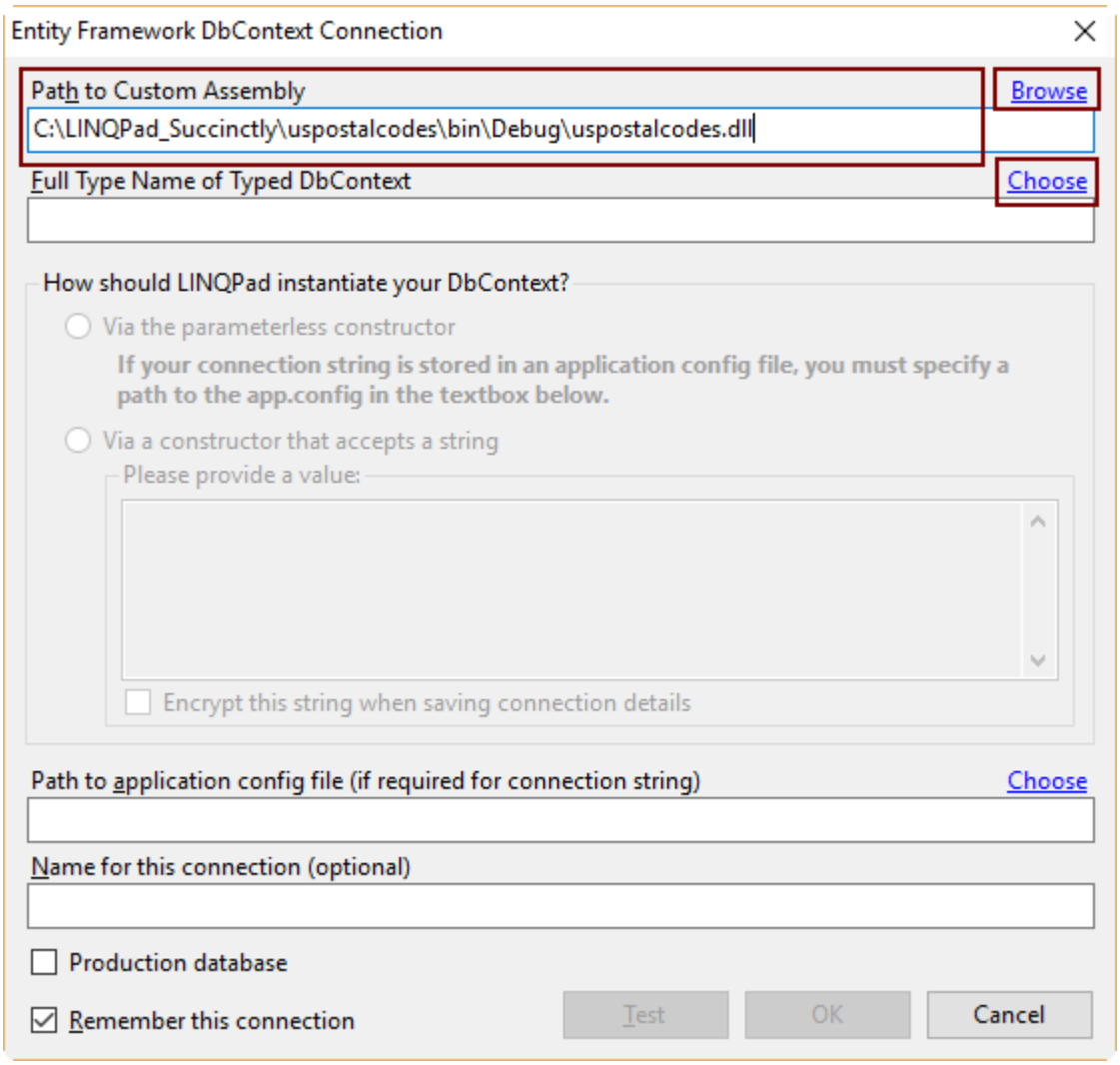


Figure 16: The Entity Framework DbContext Connection Dialog Box

Now, the Entity Framework DbContext Connection dialog box appears. To select the assembly containing the data model, you can either type the full path where the assembly is located, or click on the **Browse** hyperlink to locate the assembly file.

After the custom assembly is specified, select the typed DbContext from the **Choose Custom Type** dialog box. This dialog box appears automatically if you browse for the assembly. If you typed the path for the assembly directly, click the **Choose** hyperlink that is located to the right of the dialog box.

www.dbooks.org

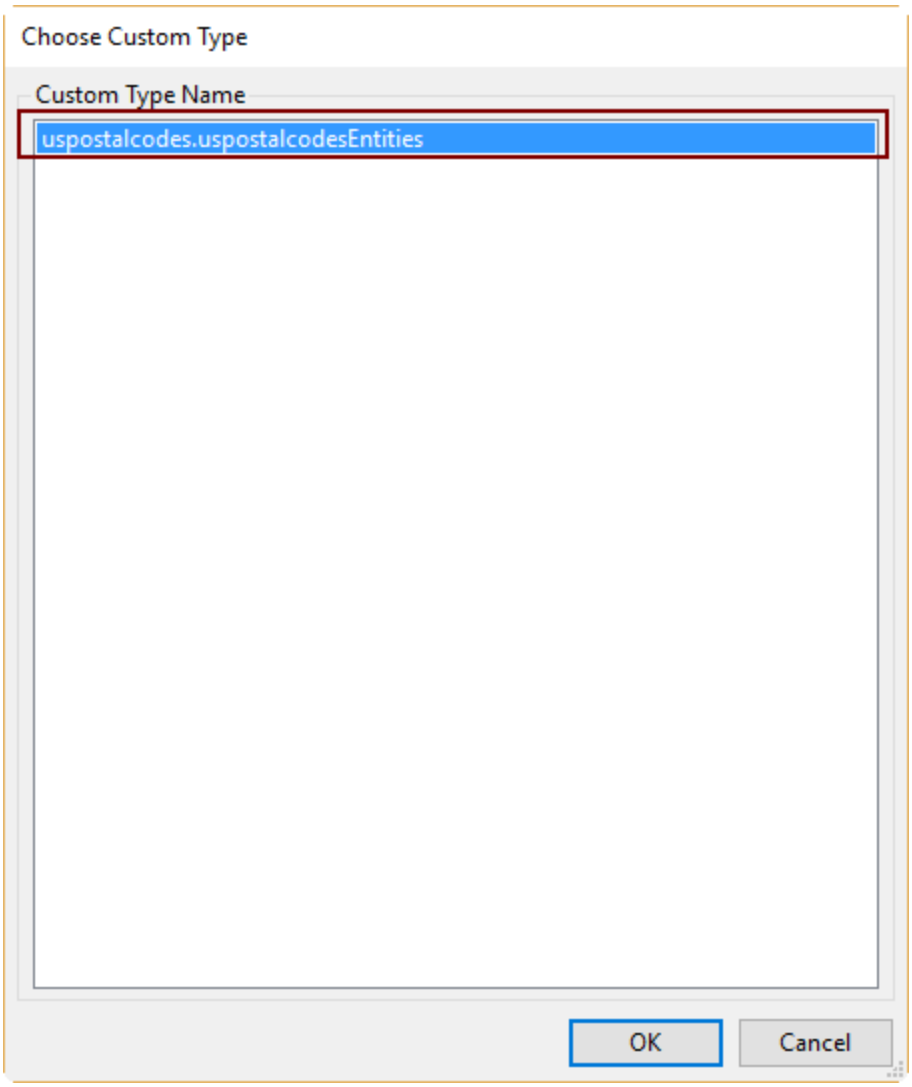


Figure 17: Choosing the Custom Type for the Data Model

The Choose Custom Type dialog box will display all typed DataContexts available in the assembly. For the **uspostalcodes** project, there's only one typed DataContext, so it will appear selected by default, as displayed in Figure 17. You can select the DataContext by clicking **OK**.

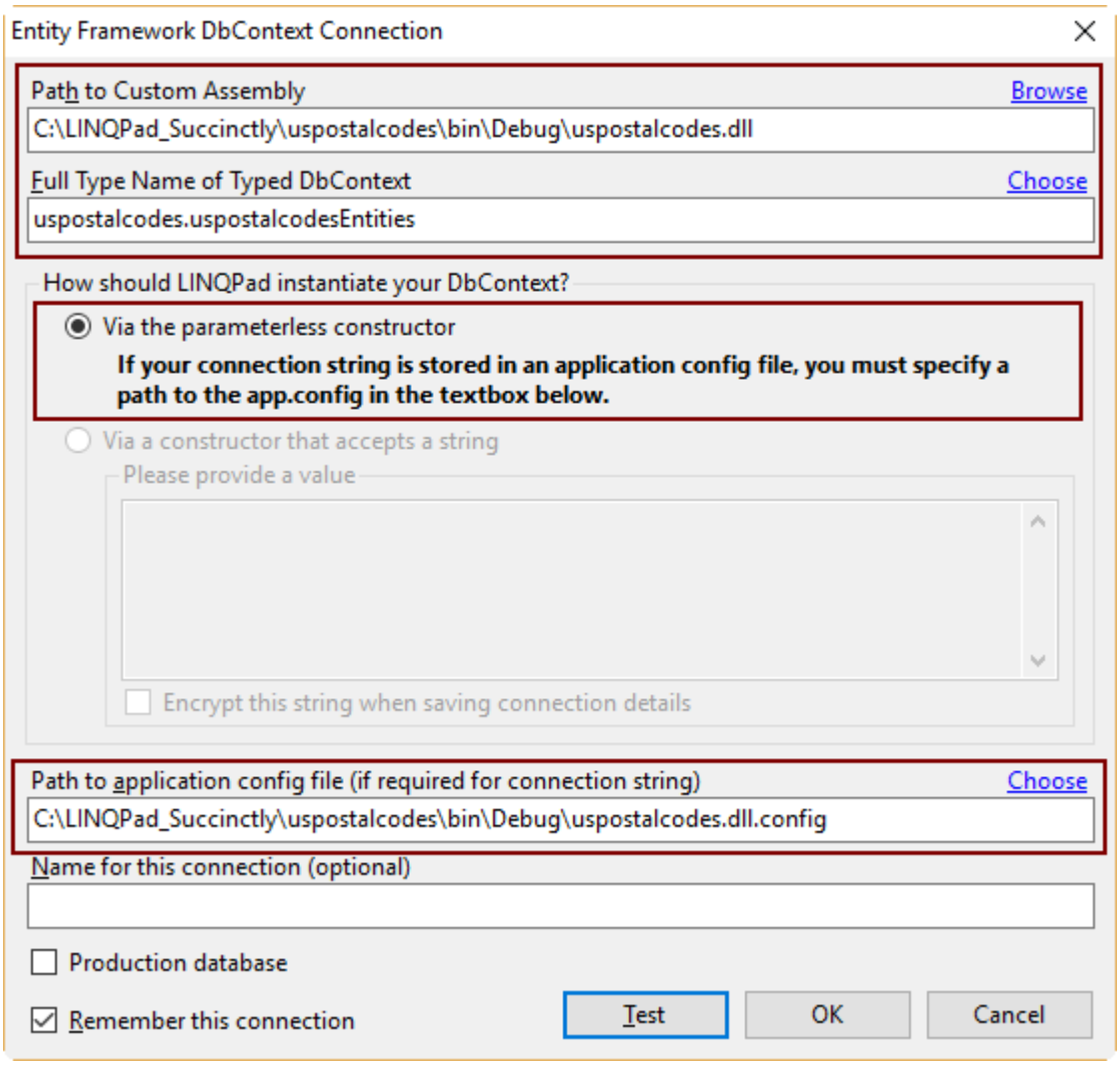


Figure 18: The Entity Framework DbContext Connection Dialog, after Typed DataContext Is Selected

The Entity Framework DbContext Connection dialog box displays the name of the custom type name and enables an option button that indicates LINQPad will instantiate the DbContext by creating a parameters constructor.

As indicated in the descriptive text below the option button, you must specify the path for the app.config file associated with the assembly. As noted in the previous figure, this config file is located in the same folder where the assembly is stored.

Click **OK** to save the connection. The connection is added to the Connection's tree view area.

www.dbooks.org

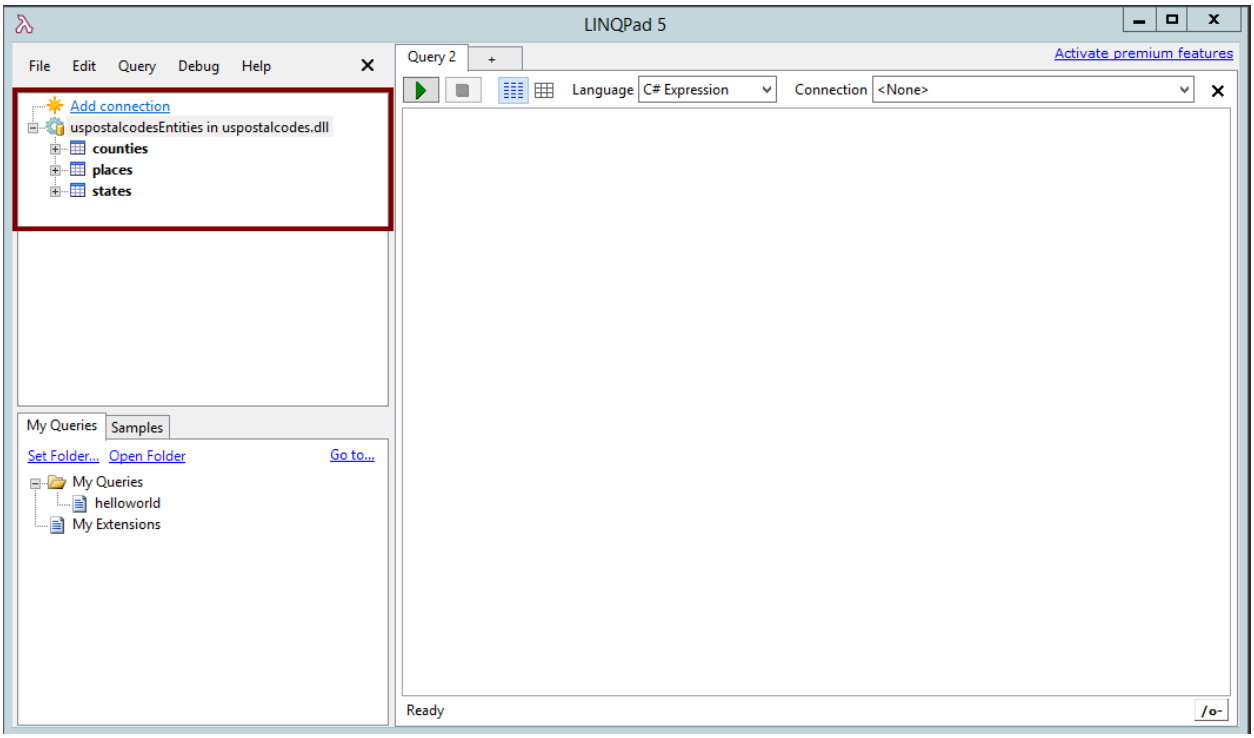


Figure 19: The Entity Framework Connection Displayed in LINQPad

The code for Entity Framework queries is like LINQ to SQL code. The following example selects the **`county_id`** and **`county_name`** columns from the **`counties`** entity, and then sorts the result by using the value stored in the **`county_name`** column.

Code Listing 1: An Entity Framework Query Example

```
var countiesTable = counties.Select(row => new {row.county_id,
row.county_name});
countiesTable.OrderBy(row => row.county_name);
countiesTable.Dump();
```

A few highlights about Entity Framework connections

When an Entity Framework connection is used, LINQPad offers the following features:

- The user can return to Visual Studio at any time and rebuild the assembly, because LINQPad doesn't lock the file.
- LINQPad automatically refreshes the tree view and the auto-completion cache (applies for paid versions) whenever the assembly changes.
- If the assembly references other assemblies, LINQPad will pick these up too.

Chapter summary

This chapter explained how to query Entity Framework models created with Visual Studio.

An ADO.NET Entity Data Model project was created, using the **uspostalcodes** database. The result for this project was a custom assembly named **uspostalcodes.dll**, containing a custom typed DataContext named **`uspostalcodes.uspostalcodesEntities`**.

LINQPad can connect to an Entity Data Model by adding a connection, just like LINQ to SQL connections. To do so, click the **Add Connection** hyperlink to bring up the Choose Data Context dialog. Then, select the option **Use a typed data context from your own assembly**, along with the **Entity Framework (DbContext)** option. Next, specify the assembly containing the data model, either by typing the full path where the assembly is located, or by browsing the disk to locate it. After that, select the custom typed DbContext from the **Choose Custom Type** dialog box. To complete the process, you must specify the path for the app.config file associated with the assembly containing the data model.

The code for Entity Framework queries is like LINQ to SQL code, so all samples written for LINQ to SQL queries should work with Entity Framework, with a few changes.

Because LINQPad doesn't lock the file, you can update and rebuild the assembly in Visual Studio at any time. If the assembly changes, LINQPad refreshes the Connection's tree view automatically.

www.dbooks.org

Chapter 3 LINQPad Scripting

LINQPad is shipped with a utility program called lprun.exe (LINQPad run). This program offers a full command-line experience, and also allows the user to write scripts. These scripts can be executed from the command line or invoked from another query.

Lprun command-line options

The lprun.exe tool can be executed with the following syntax.

Code Listing 2: Lprun.exe Execution Syntax

```
lprun [<options>] <scriptfile> [<script-args>]

Where:
<options> are a series of option values that control lprun behavior.
<scriptfile> is the name for a query script or a query (.linq) file.
<script-args> is a white-space separated list with the arguments passed to
the script or query.
```

The options that control lprun.exe behavior will be explained in the following sections. The script file can be a plain text file with any extension (even .txt), or a previously saved query file with the .linq extension. The difference between them is an XML header that LINQPad places at the beginning of every .linq file created with the program, which is absent in a plain-text script file.

A script file can receive parameters as a series of arguments. These parameters can be declared after the name of the script file, in a white-space separated list.

The following figure shows the execution of the helloworld.linq query using the lprun.exe tool.

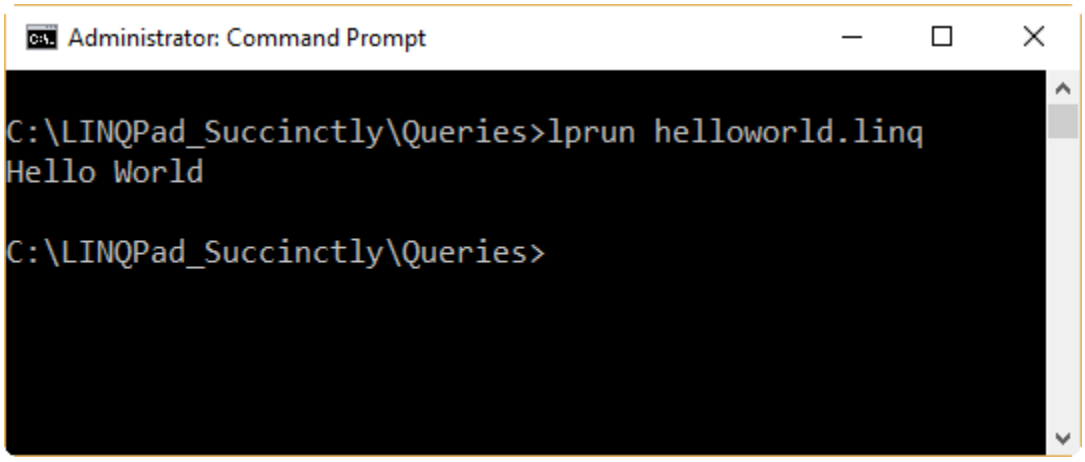


Figure 20: Running the helloworld.linq Query

Code Listing 3: helloworld.linq

```
<Query Kind="Statements" />
Console.WriteLine("Hello World");
```

Output formatting

By default, the lprun.exe tool outputs the result for a query in plain text format. Lists and non-basic objects are formatted as JSON. The output format can be controlled by the **-format** option, which has the following syntax.

Code Listing 4: The -format Option Syntax

```
-format={text|html|htmlfrag|csv|csvi}
```

The values for the **-format** option are:

- **text**: Generates output in plain text. This is the default behavior.
- **html**: Returns a complete HTML output. This output can be written to a file.
- **htmlfrag**: Returns an HTML fragment that can be inserted into an existing HTML document.
- **csv**: Renders the output to a CSV (Excel-friendly) format. This is useful when simple lists are returned.
- **csvi**: Same as **csv**, but forcing to a culture-insensitive formatting.

The following sample executes the **helloworld.linq** query with an HTML format output.

Code Listing 5: Execution of helloworld.linq Example

```
lprun -format=html helloworld.linq
```

The output generated is an HTML text, like the following sample:

Code Listing 6: Helloworld.linq Output Generated

```
!DOCTYPE HTML>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html;charset=utf-8" />
      <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="Generator" content="LINQ to XML, baby!" />
    <style type='text/css'>
body {
      margin: 0.3em 0.3em 0.4em 0.4em;
      font-family: Verdana;
      font-size: 80%;
      background: white;
```


www.dbooks.org

```
}

p, pre {
	margin:0;
	padding:0;
	font-family: Verdana;
}

table {
	border-collapse: collapse;
	border: 2px solid #17b;
	margin: 0.3em 0.2em;
}

table.limit {
	border-collapse: collapse;
	border-bottom: 2px solid #c31;
}

td, th {
	vertical-align: top;
	border: 1px solid #aaa;
	padding: 0.1em 0.2em;
	margin: 0;
}

th {
	text-align: left;
	background-color: #ddd;
	border: 1px solid #777;
	font-family: tahoma;
	font-size:90%;
	font-weight: bold;
}

th.member {
	padding: 0.1em 0.2em 0.1em 0.2em;
}

td.typeheader {
	font-family: tahoma;
	font-size: 100%;
	font-weight: bold;
	background-color: #17b;
	color: white;
	padding: 0 0.2em 0.15em 0.1em;
}

td.n { text-align: right }
```

```
a:link.typeheader, a:visited.typeheader, a:link.extenser,
a:visited.extenser {
      font-family: tahoma;
      font-size: 90%;
      font-weight: bold;
      text-decoration: none;
      background-color: #17b;
      color: white;
      float:left;
}

a:link.extenser, a:visited.extenser {
      float:right;
      padding-left:2pt;
      margin-left:4pt
}

span.typeglyph, span.typeglyphx {
      padding: 0 0.2em 0 0;
      margin: 0;
}

span.extenser, span.extenserx {
      margin-top:1.2pt;
}

span.typeglyph, span.extenser {
      font-family: webdings;
}

span.typeglyphx, span.extenserx {
      font-family: arial;
      font-weight: bold;
      margin: 2px;
}

table.group {
      border: none;
      margin: 0;
}

td.group {
      border: none;
      padding: 0 0.1em;
}

div.spacer { margin: 0.6em 0; }
```


www.dbooks.org

```
table.headingpresenter {
	border: none;
	border-left: 3px dotted #1a5;
	margin: 1em 0em 1.2em 0.15em;
}

th.headingpresenter {
	font-family: Arial;
	border: none;
	padding: 0 0 0.2em 0.5em;
	background-color: white;
	color: green;
	font-size: 110%;
}

td.headingpresenter {
	border: none;
	padding: 0 0 0 0.6em;
}

td.summary {
	background-color: #def;
	color: #024;
	font-family: Tahoma;
	padding: 0 0.1em 0.1em 0.1em;
}

td.columntotal {
	font-family: Tahoma;
	background-color: #eee;
	font-weight: bold;
	color: #17b;
	font-size:90%;
	text-align:right;
}

span.graphbar {
	background: #17b;
	color: #17b;
	margin-left: -2px;
	margin-right: -2px;
}

a:link.graphcolumn, a:visited.graphcolumn {
	color: #17b;
	text-decoration: none;
	font-weight: bold;
	font-family: Arial;
	font-size: 110%;
```

```
      letter-spacing: -0.2em;
      margin-left: 0.3em;
      margin-right: 0.1em;
}

a:link.collection, a:visited.collection { color: green }

a:link.reference, a:visited.reference { color:blue }

i { color: green; }

em { color: red; }

span.highlight { background: #ff8; }

code { font-family: Consolas }

code.xml b { color:blue; font-weight:normal }
code.xml i { color:maroon; font-weight:normal; font-style:normal }
code.xml em { color:red; font-weight:normal; font-style:normal }
    </style>

  </head>
<body>Hello World<br /></body>
</html>
```

Output redirection

The standard **>** operator can be used to redirect the output generated by lprun.exe. The following code writes the result of the **helloworld.linq** query to a file called **helloworld.html**.

Code Listing 7: Redirecting helloworld.linq Output

```
lprun -format=html helloworld.linq > helloworld.html
```

Another way to write the output to a file is by making the script itself to do the job. The following code writes the output for the query to an HTML file, which will take the same name of the query file, except for the .html extension.

Code Listing 8: Writing Query Output to a File from the Query Itself

```
string filename = Util.CurrentQueryPath.Replace(".linq",".html");
string greeting = "Hello World";
File.WriteAllText(filename,Util.ToHtmlString(greeting));
```


www.dbooks.org

If you save the query as **helloworldhtml.linq**, when this code is executed using the lprun.exe tool, a file called helloworldhtml.html will be found in the default queries folder.

The following figure shows how to execute the previous query using the lprun.exe tool.

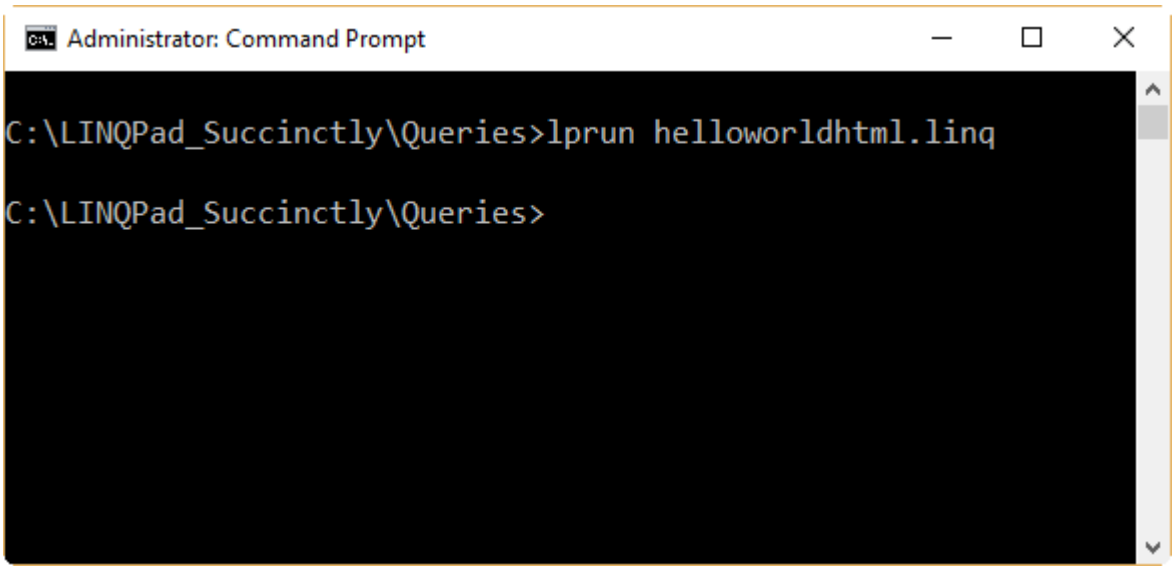


Figure 21: Helloworldhtml.linq Creating the HTML File Itself

Note: Error messages and warnings are written to the `stderr`, so they're not redirected.

Error handling

Error handling for the lprun.exe tool can be managed by using the **`%errorlevel%`** variable since all errors and warnings are written to the **`stderr`** (**`Console.Error`**). The following sample can be saved into a batch file in order to know if the query was executed with no errors.

Code Listing 9: Lprun.exe Error Handling

```
lprun helloworldhtml.linq
if %errorlevel% neq 0 echo Error executing the query!
```

Compilation options

There are three options for controlling the way lprun.exe compiles queries:

- **`-optimize`**: Enables compile optimizations. This produces a slightly faster execution but less accurate error reporting.
- **`-warn`**: Outputs the compiler warnings to the **`stderr`** (**`Console.Error`**).
- **`-compileonly`**: Tells lprun.exe that the query will be compiled only, without running anything.

The following sample tells lprun.exe that the **helloworldhtml.linq** query should be only compiled, not executed.

Code Listing 10: Compiling a Query

```
lprun -compileonly helloworldhtml.linq
```

Plain-text scripts

Lprun.exe can execute plain text files as scripts. These files are known as plain-text scripts, and as explained in the "Lprun command-line options" section of this chapter, their difference compared to a regular .linq file is the absence of the XML header found at the beginning of .linq files.

In the case of plain-text scripts, lprun.exe forces the user to tell which language will be used to compile the code. This is done with the **-lang** option of lprun.exe, and the possible values this option can take are the following:

- **Expression**: A C# expression with no semicolon at the end of the line.
- **Statements**: Several C# statements ending with a semicolon.
- **Program**: A C# program that begins with a **Main** method declaration.
- **VBExpression**: A Visual Basic .NET expression.
- **VBStatements**: Server Visual Basic .NET statements.
- **VBProgram**: A Visual Basic .NET program with a **Main** sub declaration.
- **FSharpExpression**: An F# expression.
- **FSharpProgram**: An F# program.
- **SQL**: A SQL language query.
- **ESQL**: An Entity SQL language query.

The first three values can be abbreviated to their first letter: **e** can be used for **Expression**, **s** can be used for **Statements**, and **p** can be used for **Program**.

The following code displays the value of 44. This code should be saved as **expressionscript.txt** in the LINQPad queries default folder.

Code Listing 11: Code for Displaying a Numeric Constant

```
32+12
```

To display the result in the command window, lprun.exe should be executed in the following way.

Code Listing 12: Executing expressionscript.txt with Lprun.exe

```
lprun -lang=e expressionscript.txt
```

The following figure shows the output displayed.

www.dbooks.org

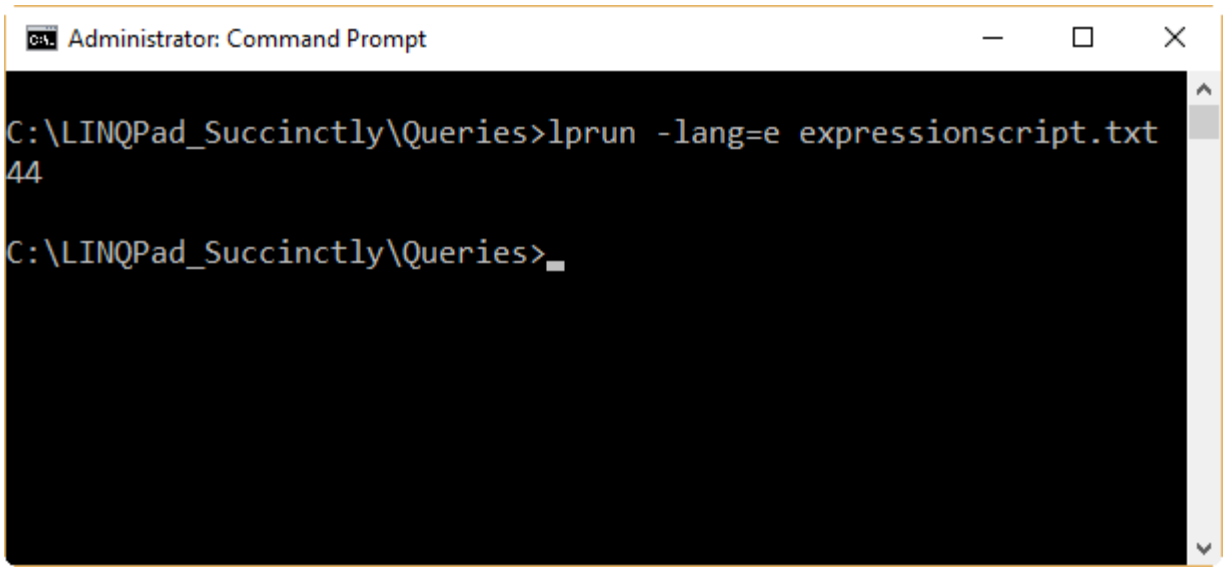


Figure 22: Executing an Expression Plain-Text Script

Managing connections

For plain-text scripts that require connections, the **`-cxnname`** option of lprun.exe should be used to specify the connection's name.

The following sample takes the first 20 rows from the **`Places`** table, which belongs to the connection called **`uspostalcodesEntities`**. The code should be saved as **placestake20.txt** in the LINQPad queries default folder.

Code Listing 13: A Script that Uses a Connection

```
Places.Take(20)
```

The placestake20.txt script should be executed in the following way.

Code Listing 14: Executing a Script with the -cxname Option

```
lprun -lang=e -cxname= uspostalcodesEntities placestake20.txt
```

The output displayed by this script is shown in the following listing.

Code Listing 15: Output Generated by placestake20.txt

```
[
  {
    "Place_id": 1,
    "Place_name": "Portsmouth",
    "Postal_code": "00210",
    "Latitude": 43.0059,
```

```
    "Longitude": -71.0130,
    "County_id": 1787
  },
  {
    "Place_id": 2,
    "Place_name": "Portsmouth",
    "Postal_code": "00211",
    "Latitude": 43.0059,
    "Longitude": -71.0130,
    "County_id": 1787
  },
  {
    "Place_id": 3,
    "Place_name": "Portsmouth",
    "Postal_code": "00212",
    "Latitude": 43.0059,
    "Longitude": -71.0130,
    "County_id": 1787
  },
  {
    "Place_id": 4,
    "Place_name": "Portsmouth",
    "Postal_code": "00213",
    "Latitude": 43.0059,
    "Longitude": -71.0130,
    "County_id": 1787
  },
  {
    "Place_id": 5,
    "Place_name": "Portsmouth",
    "Postal_code": "00214",
    "Latitude": 43.0059,
    "Longitude": -71.0130,
    "County_id": 1787
  },
  {
    "Place_id": 6,
    "Place_name": "Portsmouth",
    "Postal_code": "00215",
    "Latitude": 43.0059,
    "Longitude": -71.0130,
    "County_id": 1787
  },
  {
    "Place_id": 7,
    "Place_name": "Pleasantville",
    "Postal_code": "00401",
    "Latitude": 41.1381,
    "Longitude": -73.7840,
```


www.dbooks.org

```
    "County_id": 1852
  },
  {
    "Place_id": 8,
    "Place_name": "Holtsville",
    "Postal_code": "00501",
    "Latitude": 40.9223,
    "Longitude": -72.6370,
    "County_id": 1904
  },
  {
    "Place_id": 9,
    "Place_name": "Holtsville",
    "Postal_code": "00544",
    "Latitude": 40.9223,
    "Longitude": -72.6370,
    "County_id": 1904
  },
  {
    "Place_id": 10,
    "Place_name": "Agawam",
    "Postal_code": "01001",
    "Latitude": 42.0702,
    "Longitude": -72.6220,
    "County_id": 1213
  },
  {
    "Place_id": 11,
    "Place_name": "Amherst",
    "Postal_code": "01002",
    "Latitude": 42.3671,
    "Longitude": -72.4640,
    "County_id": 1214
  },
  {
    "Place_id": 12,
    "Place_name": "Amherst",
    "Postal_code": "01003",
    "Latitude": 42.3919,
    "Longitude": -72.5240,
    "County_id": 1214
  },
  {
    "Place_id": 13,
    "Place_name": "Amherst",
    "Postal_code": "01004",
    "Latitude": 42.3845,
    "Longitude": -72.5130,
    "County_id": 1214
```

```
  },
  {
    "Place_id": 14,
    "Place_name": "Barre",
    "Postal_code": "01005",
    "Latitude": 42.4097,
    "Longitude": -72.1080,
    "County_id": 1220
  },
  {
    "Place_id": 15,
    "Place_name": "Belchertown",
    "Postal_code": "01007",
    "Latitude": 42.2751,
    "Longitude": -72.4100,
    "County_id": 1214
  },
  {
    "Place_id": 16,
    "Place_name": "Blandford",
    "Postal_code": "01008",
    "Latitude": 42.1829,
    "Longitude": -72.9360,
    "County_id": 1213
  },
  {
    "Place_id": 17,
    "Place_name": "Bondsville",
    "Postal_code": "01009",
    "Latitude": 42.2061,
    "Longitude": -72.3400,
    "County_id": 1213
  },
  {
    "Place_id": 18,
    "Place_name": "Brimfield",
    "Postal_code": "01010",
    "Latitude": 42.1165,
    "Longitude": -72.1880,
    "County_id": 1213
  },
  {
    "Place_id": 19,
    "Place_name": "Chester",
    "Postal_code": "01011",
    "Latitude": 42.2794,
    "Longitude": -72.9880,
    "County_id": 1213
  },
```


www.dbooks.org

```
  {
    "Place_id": 20,
    "Place_name": "Chesterfield",
    "Postal_code": "01012",
    "Latitude": 42.3923,
    "Longitude": -72.8250,
    "County_id": 1214
  }
]
```

Managing assemblies and namespaces

LINQPad plain-text scripts support a directive named **`ref`**. This directive is used for referencing additional assemblies. Also, unlike .linq queries, plain-text scripts support using directives to import additional namespaces.

The following sample creates a custom button class derived from the **`System.Windows.Forms.Button`** class.

Code Listing 16: Referencing Assemblies in a Script

```
ref System.Windows.Forms.dll;
ref System.Drawing.dll;

using System.Windows.Forms;
using System.Drawing;

void Main()
{
      var custombutton = new CustomButton();
      custombutton.Dump();
}

public class CustomButton : Button
{
    public CustomButton()
      {
           FlatStyle = FlatStyle.Flat;
             BackColor = Color.FromArgb(244,152,31);
             ForeColor = Color.Black;
         Font = new Font("Segoe UI",12);
         Text = "Custom Button Class Flat Version";
      }
}
```

As noted in the previous code, all assemblies' references and namespaces importing must be at the top of the script. The user can specify each assembly either by its file name (including the absolute path) or its fully qualified name, in case the assembly is in the GAC.



Note: `ref` directives must precede `using` directives.

The previous script must be executed as a Program script, as shown in the following example.

Code Listing 17: Executing a Script as a Program Script

```
lprun -lang=p -format=html custombutton.txt > custombutton.html
```

This code executes the script, formatting the result in HTML. The result is saved into the custombutton.html file.

Passing arguments to scripts

To pass arguments to scripts from the command line, a space-separated list should be typed after the file name of the script. To pick those arguments, a Program query can be written, declaring a string array parameter at the **`Main`** method.

The following example dumps all arguments passed to the query.

Code Listing 18: A Script that Receives Arguments from the Command Line

```
void Main(string[] args)
{
   args.Dump();
}
```

Assuming that the previous code was saved in a file named passingarguments.txt, the following command will display **`["Hello", "World"]`**.

Code Listing 19: Executing a Script with Arguments

```
lprun -lang=p passingarguments.txt Hello World
```

The CMD Boolean symbol

Sometimes it is necessary to know if a query is being executed from the command line or from the GUI. In order to do this, LINQPad provides a Boolean symbol called **`CMD`**. This symbol should be used in conjunction with the **`#if`** directive, as shown in the following sample.

www.dbooks.org

Code Listing 20: Using the CMD Symbol

```
void Main()
{
   var custombutton = new CustomButton();

#if CMD
   string filename = Util.CurrentQueryPath.Replace(".linq",".html");
   File.WriteAllText(filename,Util.ToHtmlString(custombutton));
#else
   custombutton.Dump();
#endif
}

public class CustomButton : Button
{
    public CustomButton()
      {
           FlatStyle = FlatStyle.Flat;
             BackColor = Color.FromArgb(244,152,31);
             ForeColor = Color.Black;
         Font = new Font("Segoe UI",12);
         Text = "Custom Button Class Flat Version";
      }
}
```

As noted in the previous code, the **CMD** directive helps LINQPad choose whether to save the contents of the **CustomButton** object into an HTML file, or to display the button in the Results panel.

The code should be saved as a .linq query file (for book purposes, cmdsymbolsample.linq), referencing the **System.Drawing.dll** and **System.Windows.Forms.dll** assemblies, and also importing the **System.Drawing** and **System.Windows.Forms** namespaces.

Now, if the query is executed as a script using lprun.exe, a file named cmdsymbolsample.html will be created in the default queries folder. Otherwise, if the query is executed from the LINQPad user interface, the **CustomButton** object will be displayed in the Results panel.

Calling a script from another script

Scripts can be combined in order to execute one script from another. To do this, use the **Util.Run** method. The following sample takes the name of any query as an argument from the command line, and then runs that query getting the result in HTML format. The result is displayed in the command window.

Code Listing 21: Executing a Script from Another Script

```
void Main (string[] args)
```

```
{
   if (args.Length != 1)
   {
      "Usage:\r\nlprun -lang=p querytohtml.txt <queryname>".Dump();
   }
   else
   {
      if (!File.Exists(args[0]))
        {
          var message = "The file " + args[0] + " doesn't exist.";
            message.Dump();
        }
        else
        {
        string htmlResult =
Util.Run(args[0],QueryResultFormat.Html).AsString();
          htmlResult.Dump();
        }
   }
}
```

This code expects a one-element length array with the name of the query file to run. If no arguments or more than one argument are passed to the query, a help message is displayed. Also, the query validates if the file exists in the filesystem. If not, an error message is displayed. Once all validation tasks are passed, the **`Util.Run`** method executes the query, formatting the result as an HTML string, using the **`AsString`** method. The result of the query is stored in the **`htmlResult`** variable, which is dumped to the command window.

Assuming that the previous script was saved in a file named querytohtml.txt, the following sample runs the imagerendering.linq query from within this script.

Code Listing 22: Executing the Scripts

```
lprun -lang=p querytohtml.txt imagerendering.linq
```

Chapter summary

LINQPad is shipped with a utility program called lprun.exe, which offers a full command-line experience and also allows the user to write scripts to be executed from the command line or from within another query.

A script can be written using plain-text files or LINQPad .linq query files. A script saved in plain-text format doesn’t have the XML header found in .linq files.

The lprun.exe tool has a series of options that can control its behavior. These options control the output formatting, the way a script is compiled, the database connection that a script should use, and the language used by the script.

www.dbooks.org

Arguments can be passed to a script from the command line. To do this, you should write the values for these arguments using a space-separated list after the name of the script. The arguments are trapped by the query using a string array parameter in the **`Main`** method of a C# Program query.

You can determine if a script is being executed either from the command line or from the user interface by using the **`CMD`** Boolean symbol. This symbol takes the value of **`true`** when a script is executed by lprun.exe. Otherwise, a value of **`false`** is stored in that symbol.

Finally, the **`Util.Run`** method allows you to execute a script from another one. The output format for the result can be established by the **`QueryResultFormat`** enumeration, and converted to a string using the **`AsString`** method.

Chapter 4 LINQPad Extensibility

LINQPad provides elements to extend its power by letting you write your own classes or methods in order to improve several features included in the product. This chapter will focus on writing custom extensions and visualizers.

Writing custom extensions

LINQPad allows you to write your own methods in order to make them available for all queries. This is done by using the **My Extensions** node tab, which is located beside the **My Queries** tree view in the user interface.

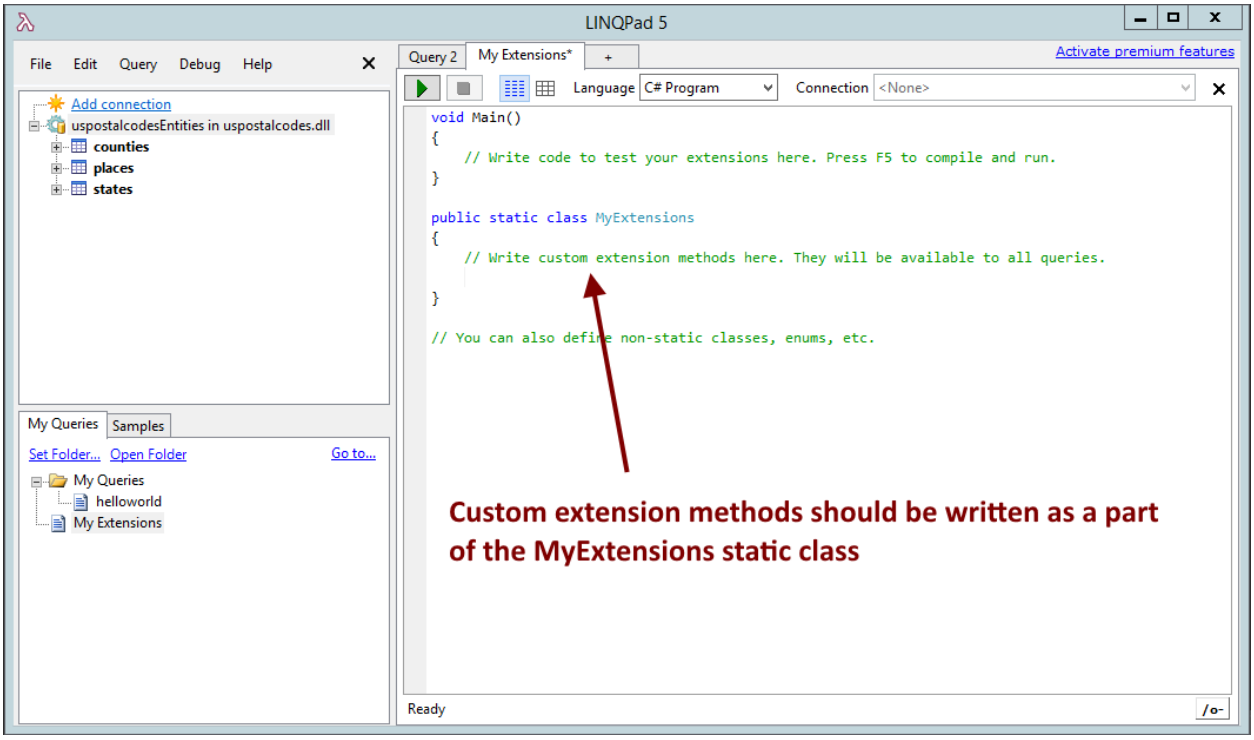


Figure 23: My Extensions Node and Its Default Code

The first time you click on the My Extensions node, an empty code template is built automatically by LINQPad. This empty code template can also be activated using Ctrl+Shift+Y. As noted in Figure 23, this is a C# Program query that contains a **`Main`** method and an empty definition for a static class named **`MyExtensions`**. All custom extension methods should be coded within this class, and when they're ready, you should press **F5** or click the **Execute Query** button to make LINQPad compile the query. This compilation will create a custom assembly called MyExtensions.FW40.dll in the LINQPad Plugins folder, and make all methods available for all queries.

www.dbooks.org

String and DateTime extensions sample

The first example of custom extensions will expose a series of methods to manage **`string`** and **`DateTime`** value types.

Code Listing 23: An Example of Custom Extensions

```
void Main()
{
      // Write code to test your extensions here. Press F5 to compile and
run.
}

public static class MyExtensions
{
        public static string Proper(this string inputString)
        {
            if (inputString == string.Empty) return string.Empty;

            if (inputString.Contains(" "))
            {
                var result = string.Empty;
                var wordsList = inputString.Split(' ');
                result = wordsList.Aggregate(result, (current, word) =>
current + (word.Substring(0, 1).ToUpper() + word.Substring(1).ToLower() + "
"));
                return result.Trim();
            }

            return inputString.Substring(0, 1).ToUpper() +
inputString.Substring(1).ToLower();
        }
      // Write custom extension methods here. They will be available to all
queries.

      public static DateTime BackNDays(this DateTime currentDateTime, int
daysToGoBack)
        {
            return
(currentDateTime.Subtract(TimeSpan.FromDays(daysToGoBack)));
        }

     public static DateTime ForwardNDays(this DateTime currentDateTime, int
daysToGoBack)
        {
            return (currentDateTime.Add(TimeSpan.FromDays(daysToGoBack)));
        }
```

```
        public static string ToXmlSchemaDateTime(this DateTime
currentDateTime)
        {
            var xmlschemaDatetime = XmlConvert.ToString(DateTime.Now,
XmlDateTimeSerializationMode.Unspecified);
            xmlschemaDatetime = xmlschemaDatetime.Substring(0,
xmlschemaDatetime.IndexOf(".", StringComparison.Ordinal));

            return (xmlschemaDatetime);
        }

}

// You can also define non-static classes, enums, etc.
```

This code exposes four extension methods:

- **Proper**: Converts the first letter of every word found in the string, passed as a parameter, to an uppercase letter, and the rest of that word to lowercase.
- **BackNDays**: Returns the **DateTime** value that results from subtracting the number of days, passed as an integer parameter, from the **DateTime** value, also passed as a parameter.
- **ForwardNDays**: Returns the **DateTime** value that results from adding the number of days, passed as an integer parameter, to the **DateTime** value, also passed as a parameter.
- **ToXmlSchemaDateTime**: Returns a string representing the **DateTime** value, passed as a parameter, according to the XML Schema **DateTime** value definition.

The following query tests the extension methods written in the previous code.

Code Listing 24: Testing Custom Extensions

```
var properName = "THIS IS A PROPER NAME";
properName.Proper().Dump();
DateTime.Now.BackNDays(3).Dump();
DateTime.Now.ForwardNDays(5).Dump();
DateTime.Now.ToXmlSchemaDateTime().Dump();
```

The output displayed by this query is shown in the following figure.

www.dbooks.org

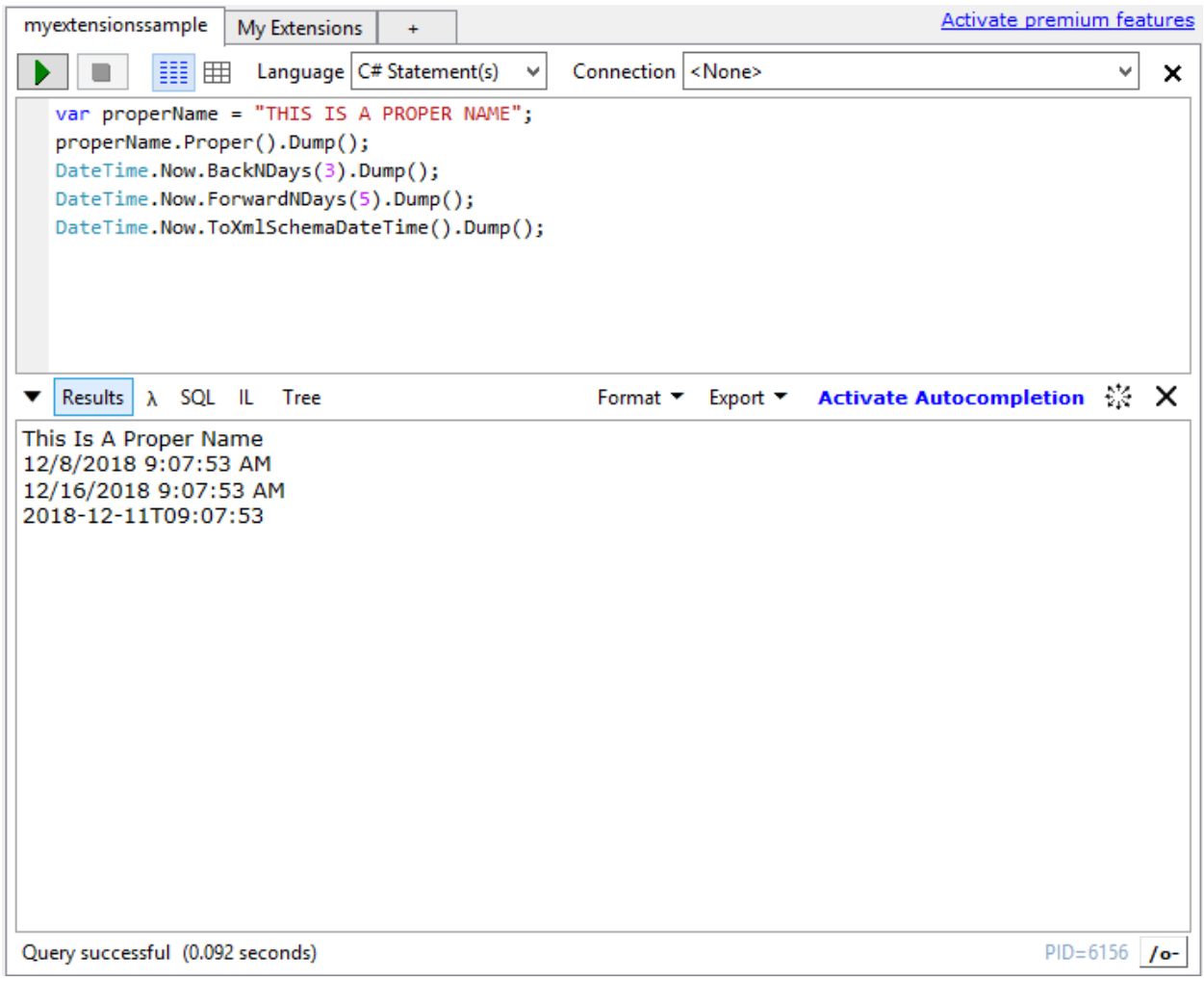


Figure 24: Testing the Custom Extension Methods

Enumerating the properties and values for an object

Another useful application for extension methods is an object-properties enumerator, including their values. To do this, add the following method to the **`MyExtensions`** class.

Code Listing 25: Enumerating Properties and Values for an Object

```
public static Dictionary<string, object> Properties(this object myself)
{
  return myself.GetType().GetProperties().Where(prop =>
prop.CanRead).ToDictionary(prop => prop.Name, prop =>
prop.GetValue(myself));
}
```

The first thing this code does is retrieve the type of the object passed as a parameter with the **`GetType`** method. Then, the **`GetProperties`** method returns a **`PropertyInfo`** array with all

properties belonging to the object, including their values. The LINQ **Where** method is employed to choose those properties with a **Get** accessor available (**CanRead** equals to **true**). Then, the **ToDictionary** method transforms those properties in a **<string, object>** collection, where the name of each property is used as a key, and each property value corresponds to each dictionary item's value.

The following query will be used to test this extension method.

Code Listing 26: Testing Enumeration of Properties

```
void Main()
{
      var custombutton = new CustomButton();
      var properties = custombutton.Properties();
      properties.Dump();
}

// Define other methods and classes here
public class CustomButton : System.Windows.Forms.Button
{
    public CustomButton()
      {
         Text = "Custom Button Class";
         Font = new System.Drawing.Font("Segoe UI",12);
      }
}
```

As noted in the previous code, the query doesn't display the button itself, but its properties. A partial view of the output is shown in the following figure.

www.dbooks.org

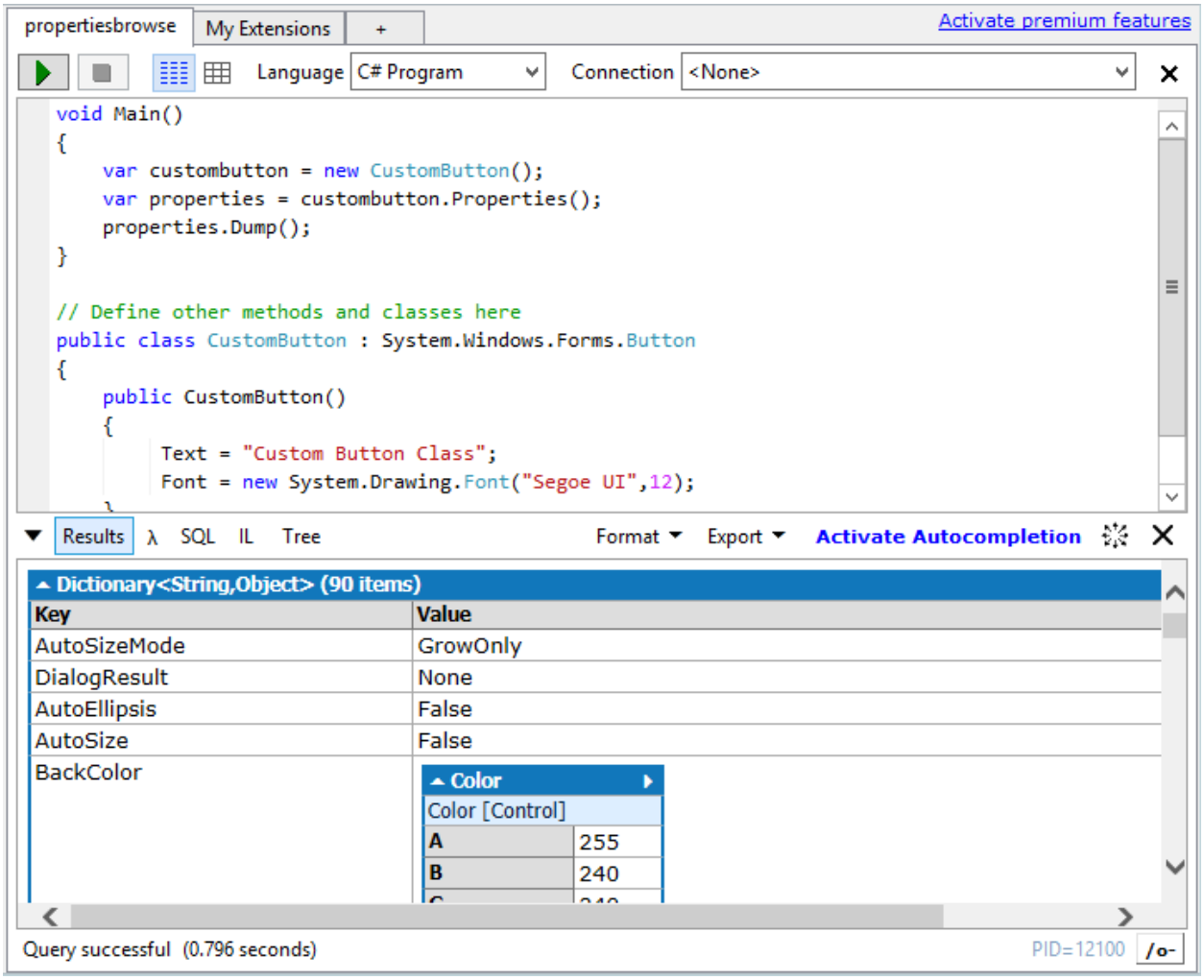


Figure 25: Displaying the Properties for an Object

Packaging custom extension methods as a plugin

It's possible to package custom extension methods as a plugin for LINQPad. This is done by creating a class library project in Visual Studio and adding a reference to LINQPad.exe into that project. Once the project is built, you should copy the assembly DLL and any dependencies into LINQPad's plugins folder (usually My Documents\LINQPad Plugins).

For the purposes of this book, a project called Linqpadextensions will be created, and all the extension methods explained in the previous sections of this chapter will be included.

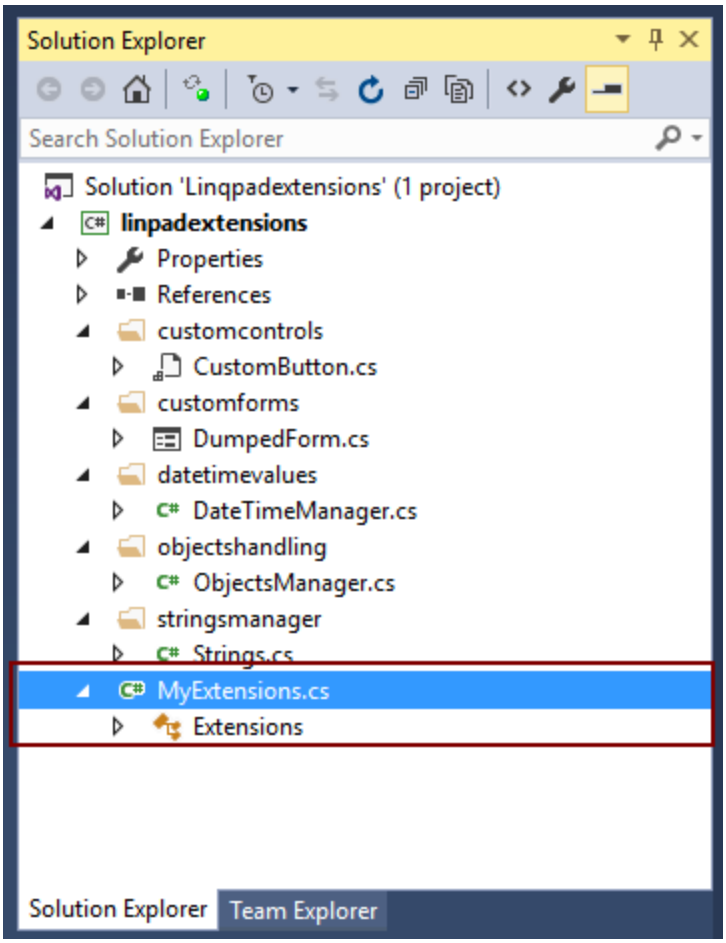


Figure 26: Linqpadextensions Project in Visual Studio's Solution Explorer

As highlighted in the previous figure, a custom class named **`MyExtensions`** is defined in a program called MyExtensions.cs. The code for this program is as follows.

Code Listing 27: MyExtensions.cs Program Listing

```
using System;
using System.Collections.Generic;
using LINQPad.datetimevalues;
using LINQPad.objectshandling;
using LINQPad.stringsmanager;

namespace LINQPad
{
    public static class MyExtensions
    {
        public static string ToXmlSchemaDateTime(this DateTime
currentDateTime)
        {
            return new
DateTimeManager().ToXmlSchemaDateTime(currentDateTime);
```


www.dbooks.org

```
        }

        public static DateTime BackNDays(this DateTime inputDateTime, int
daysNumber)
        {
            return (new DateTimeManager().BackNDays(inputDateTime,
daysNumber));
        }

        public static DateTime ForwardNDays(this DateTime inputDateTime,
int daysNumber)
        {
            return (new DateTimeManager().ForwardNDays(inputDateTime,
daysNumber));
        }

        public static string Proper(this string inputString)
        {
            return (Strings.Proper(inputString));
        }

        public static Dictionary<string, object> Properties(this object
mySelf)
        {
            return ObjectsManager.Properties(mySelf);
        }
    }
}
```

The most important thing about this code is the **`namespace`** declaration. As noted in the program, the default namespace is declared as **`LINQPad`**. This ensures that all classes defined in the custom assembly will be available when queries are executed, if the DLL is copied into LINQPad's plugins folder.

Also, for maintenance reasons, the actual methods that return the results are defined in separate namespaces and classes. These methods are called from the **`MyExtensions`** class.



Note: The explanation for all programs that are part of the Linqpadextensions project are beyond the scope of this book. The source code for the project is supplied as part of the examples available at GitHub.

Creating a custom visualizer

LINQPad can display the output generated by a query in two ways. The first and most common way is calling the Dump extension method, which creates an instance of the **`DumpContainer`** object for displaying the results. The second way is by using the **`DisplayControl`** and

DisplayWpfElement methods of the **PanelManager** static class, which creates an instance of an **OutputPanel** object and returns it to the calling process.

Both the **PanelManager** static class and the **DumpContainer** object can be accessed from LINQPad queries, and can be used in custom extension methods declared in **MyExtensions** or packed into a custom assembly. This means that you can customize them to meet a particular need. In the case of custom assembly packaging, a reference to LINQPad.exe must be added to the project in order to gain access to the **PanelManager** static class and the **DumpContainer** object.

The following sections will explain some ways to use the **PanelManager** static class and the **DumpContainer** object.

Displaying rows from a LINQ to SQL entity in a grid view

The **DisplayControl** method of the **PanelManager** class takes a Windows Forms control and shows it in an output panel. The method accepts an optional string parameter to identify the instance of the output panel created. The string is also displayed in the Results panel.

Because the **DisplayControl** method can take any Windows Forms control, it's possible to show the contents of a data table using a **DataGridView**. An extension method can be written to feed the **DataGridView** control with the results of a LINQ to SQL query.

Writing the extension method

The following extension method assigns the contents of a **DataTable** into a **DataGridView**, and displays the control using a custom **OutputPanel**.

Code Listing 28: Using a custom OutputPanel to display a DataTable

```
public static void DisplayInGrid(this DataTable dataTable,string title =
null)
 {
     if (string.IsNullOrEmpty(title)) title = "&Custom";

     var dgrid = new DataGridView { DataSource = dataTable };
     PanelManager.DisplayControl(dgrid, title);
 }
```

The first line within the braces sets the title variable to **&Custom** if no string value is passed when the method is invoked. Then, the second line creates a **DataGridView** instance and places it into a variable called **dgrid**. The **DataSource** property takes the contents of the **dataTable** parameter to fill the **DataGridView**. Finally, the method calls **DisplayControl** and passes the values of the **dgrid** and **title** variables.

www.dbooks.org

Solving the IQueryable<T> issue

At this point, everything seems easy with the previous extension method. But when the following query is executed, LINQPad throws an exception.

Code Listing 29: Displaying Data in a Custom OutputPanel

```
var states = States.Select(row => new {row.State_id,row.State_name,
row.State_abbr});
states.DisplayInGrid("Custom Panel");
```

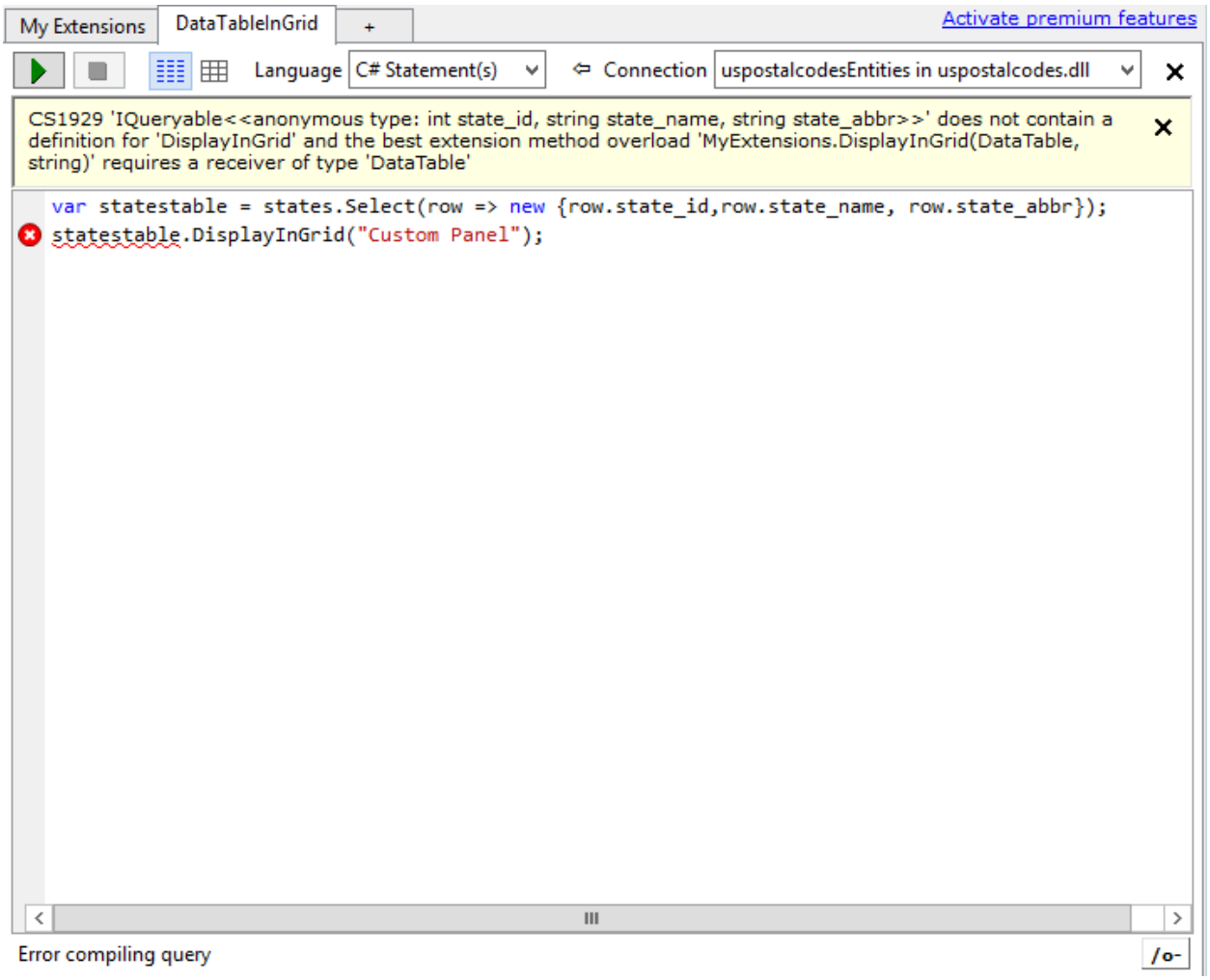


Figure 27: The DisplayInGrid Extension Method Fails

The exception displayed in Figure 27 is fired because a LINQ to SQL query returns an **`IQueryable<T>`** object, and the extension method is built to be used by a **`DataTable`** object.

To solve this, an extension method for an **`IQueryable<T>`** object and an auxiliary method should be implemented in order to make a **`DataTable`** object available for the **`DisplayInGrid`** method.

Code Listing 30: Transforming IQueryable<T> to a DataTable

```
public static DataTable LinqToDataTable<T>(this IQueryable<T> queryData)
 {
   return (EnumToDataTable(queryData.AsEnumerable()));
 }

private static DataTable EnumToDataTable<T>(IEnumerable<T> queryData)
 {
            var result = new DataTable();
            if (queryData == null) return result;

            PropertyInfo[] queryDataProps = null;

            foreach (var queryRecord in queryData)
            {
                if (queryDataProps == null)
                {
                    queryDataProps = queryRecord.ReadProperties();

                    foreach (var property in queryDataProps)
                    {
                        var columnType = property.PropertyType;

                        if (columnType.IsGenericType &&
(columnType.GetGenericTypeDefinition() == typeof(Nullable<>)))
                        {
                            columnType =
columnType.GetGenericArguments()[0];
                        }
                        result.Columns.Add(new DataColumn(property.Name,
columnType));
                    }
                }

                var resultRow = result.NewRow();

                foreach (var property in queryDataProps)
                {
                    resultRow[property.Name] =
property.GetValue(queryRecord, null) ?? DBNull.Value;
                }

                result.Rows.Add(resultRow);
            }

  return (result);
}
```


www.dbooks.org

The **LinqToDataTable** method shown in Code Listing 30 is the actual extension method. This method returns a **DataTable** object from an **IQueryable<T>** object. To perform this conversion, the method calls a private method named **EnumToDataTable**, passing to it the **IQueryable<T>** object as an enumeration (the **AsEnumerable** method is used for this purpose).

The **EnumToDataTable** method creates a **DataTable** object named **result**. The columns for the **DataTable** are created using the property name and the property type of each property belonging to the first enumeration item. The process is performed this way because these properties are the same for all items in the enumeration. After that, the method creates a **DataRow** object called **resultRow** for each item in the enumeration, and copies each value contained in the item by iterating its properties, and passing the value of each one to the corresponding column in the **DataRow** object. Each column name in the **DataRow** object matches the name of each property of the enumeration item.

Finally, the **DataRow** object is added to the **DataTable**, and this is returned to the calling process after the loop ends.

Getting the expected results

Now, by making a few changes in the query…

Code Listing 31: Using LinqToDataTable() Extension Method

```
var states = States.Select(row => new {row.State_id,row.State_name,
row.State_abbr}).LinqToDataTable();
states.DisplayInGrid("Custom Panel");
```

…the following output is displayed:

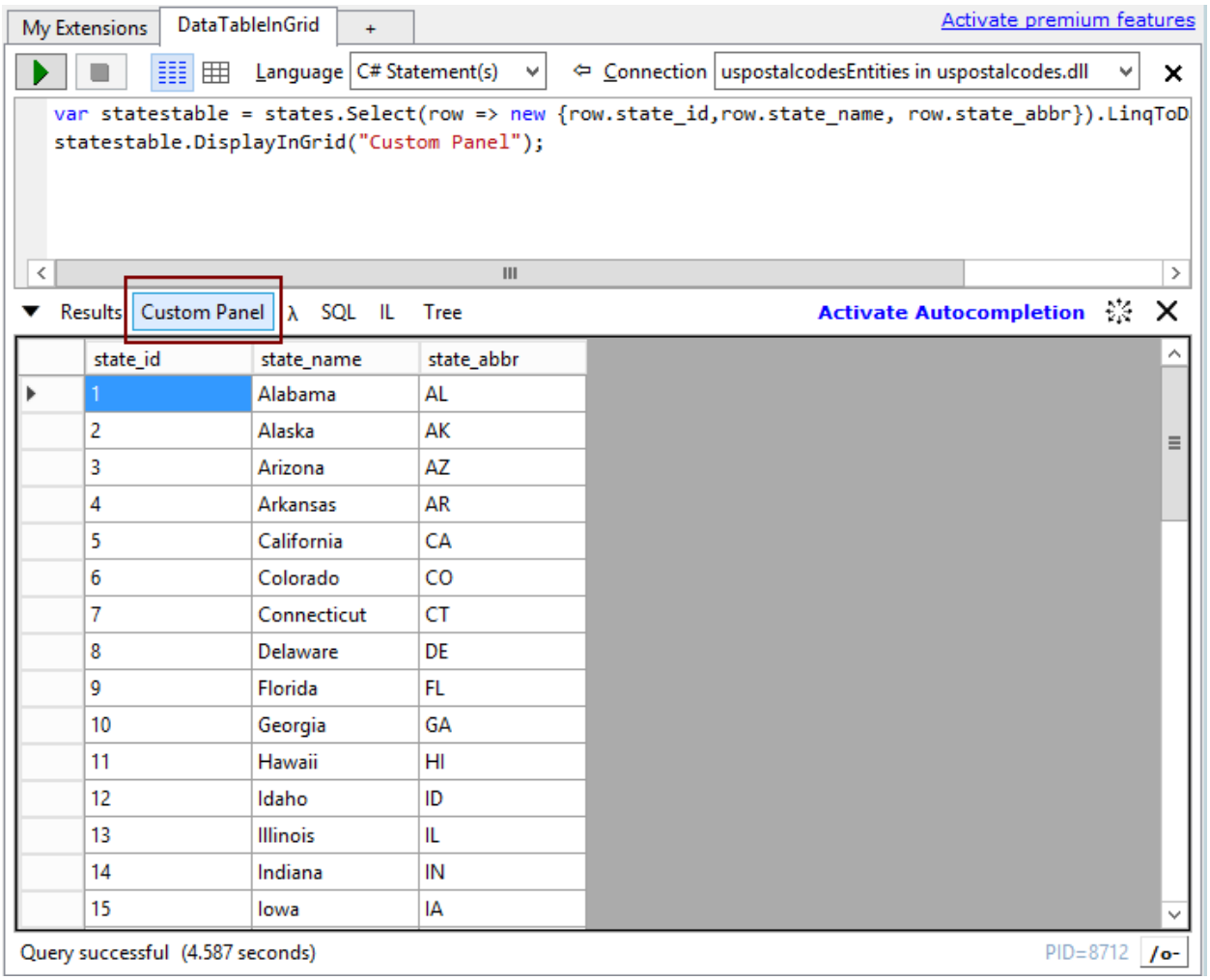


Figure 28: The DisplayInGrid Extension Method Working as Expected

The desired **`DataViewGrid`** control is displayed in a custom output panel, as seen in Figure 28. This is the result expected from the **`DisplayInGrid`** extension method. The highlighted portion shows the title for the **`OutputPanel`** instance created.

Overloading the DisplayInGrid extension method

The solution given to the **`IQueryable<T>`** issue might not be suitable for practical usages. Sometimes, the user can forget to employ the **`LinqToDataTable`** extension method before calling **`DisplayInGrid`**, causing LINQPad to throw the **`IQueryable<T>`** exception and crashing the query.

The problem can be solved in a better way by overloading the **`DisplayInGrid`** extension method, allowing an **`IQueryable<T>`** object as a parameter.

www.dbooks.org

Code Listing 32: Overloading DisplayInGrid to Avoid LinqToDataTable()

```
public static void DisplayInGrid<T>(this IQueryable<T> queryData,string
title = null)
 {
    if (string.IsNullOrEmpty(title)) title = "&Custom";
    DisplayInGrid(EnumToDataTable(queryData.AsEnumerable()),title);
 }
```

As noted in this code, the overloaded **DisplayInGrid** method turns out as a generic method. This is required because the **IQueryable** object passed to the method can be the result of any LINQ to SQL query, and the type for this result is unknown at the time of the assembly development. At the end, the overloaded method employs the **DataTable** version of itself. To pass the **IQueryable<T>** object as a **DataTable**, it uses the **EnumToDataTable** method, which returns a **DataTable** object from an **IQueryable<T>** object.

Now, if the query is written in this way…

Code Listing 33: The Original DisplayInGrid Example

```
var states = States.Select(row => new {row.State_id,row.State_name,
row.State_abbr});
states.DisplayInGrid("Custom Panel");
```

…which is the original query that caused LINQPad to throw the exception, the same result for the **DisplayInGrid** method will be shown as expected.

Stacking WPF elements

Multiple elements can be displayed on a single output panel. This can be done by using the **StackWpfElement** method of the **PanelManager** class. The method can be invoked from any LINQPad query, or from a custom extension method for that purpose.

Creating the extension method

To create an extension method for WPF elements, add the following code to the **Linqpadextensions** project.

Code Listing 34: StackWpfControl Extension Method

```
public static void StackWpfControl(this System.Windows.UIElement element,
string title = null)
  {
     if (string.IsNullOrEmpty(title)) title = "&Custom";
     PanelManager.StackWpfElement(element, title);
  }
```

The extension method shown in this code receives a **`UIElement`** object, and a string to be displayed as a title for the output panel. This method can be used with any WPF element.

The following query…

Code Listing 35: Testing StackWpfControl

```
var button = new System.Windows.Controls.Button {Content = "My Button"};
button.StackWpfControl();
```

…produces this output:

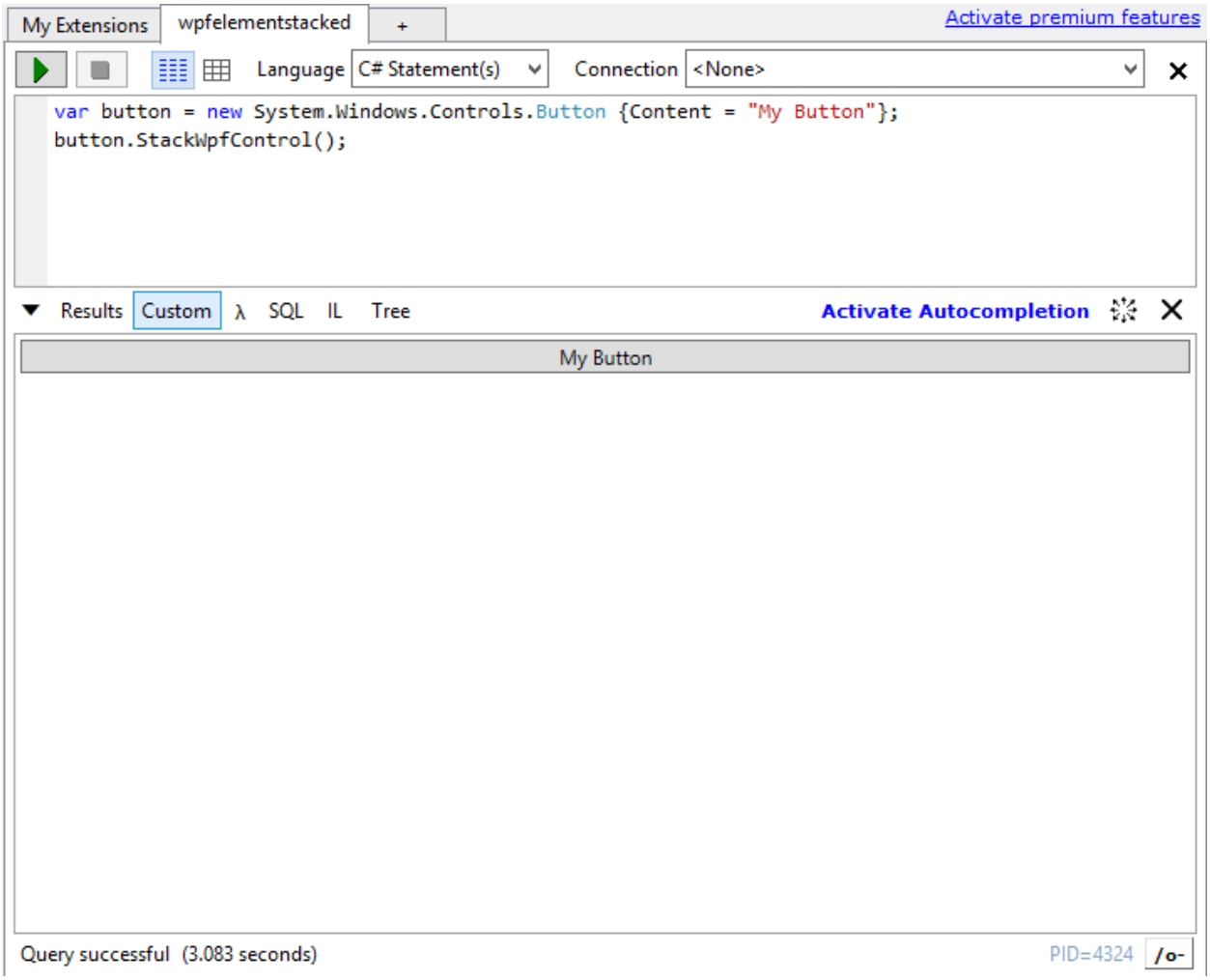


Figure 29: A WPF Element Displayed in an OutputPanel.

Stacking several elements at once

The **`StackWpfControl`** extension method can be called repeatedly to stack several WPF elements in an output panel. However, it could be created as an extension method that stacks several objects at once.

www.dbooks.org

Code Listing 36: Stacking Several Controls at Once

```
public static void StackWpfControls(this List<System.Windows.UIElement>
elements, string title = null)
  {
            elements.ForEach(element => StackWpfControl(element,title));
  }
```

The previous code defines an extension method that receives a **`List`** of **`System.Windows.UIElement`** objects, and a string to be used as a title for the output panel. The **`ForEach`** LINQ method iterates the list and calls the **`StackWpfControl`** method for each one of its elements.

Now, this query…

Code Listing 37: Testing StackWpfControls

```
List<System.Windows.UIElement> elements = new
List<System.Windows.UIElement> {new System.Windows.Controls.Button {Content
= "My Button"},  new System.Windows.Controls.Label {Content = "My Label"},
new System.Windows.Controls.Expander {Header = "More"}};
elements.StackWpfControls();
```

…displays the following output:

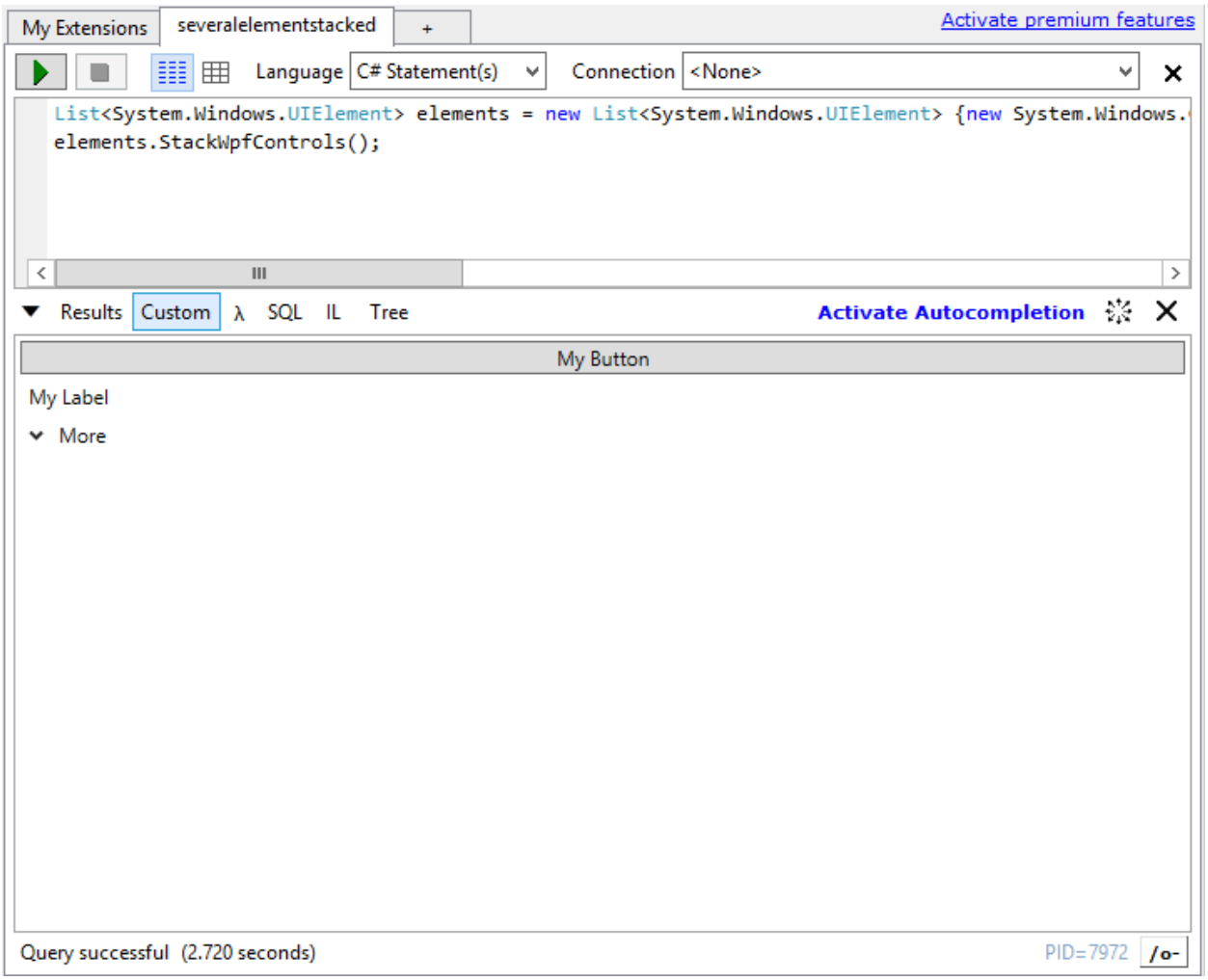


Figure 30: Several WPF Elements Displayed at Once

Catching element events

Events associated to WPF elements can be caught in LINQPad queries. The following code uses the **`Click`** event of a WPF button to close the **`OutputPanel`** that displays the button.

Code Listing 38: Trapping Element Events

```
void Main()
{
	var button = new System.Windows.Controls.Button {Content = "My
Button"};
	button.Click += catchedEvent;
	button.StackWpfControl("Events");
}
private static void catchedEvent(object sender, System.EventArgs args)
{
	PanelManager.GetOutputPanel("Events").Close();
```


www.dbooks.org

```
}

// Define other methods and classes here
```

When the previous query is executed, an output panel named **Events** will be displayed with the WPF button in it. The button will respond to the **Click** event after the query has finished executing. Clicking on the button will close the **Events** output panel.



Note: A reference to PresentationCore.dll, PresentationFramework.dll, System.Xaml.dll, and WindowsBase.dll must be added to the queries and to the Linqpadextensions project in order to get the expected results from all WPF elements samples.

The DumpContainer object

The common way query results are displayed is by employing the **Dump** extension method. Every time **Dump** is invoked, a **DumpContainer** object is created to display query results. This object has some properties that can be used to control the mode in which results are shown. The following table summarizes these properties.

Table 2: DumpContainer Properties

Property	Description
Content	An object instance that contains the results to be displayed.
DumpDepth	Establishes the maximum number of levels Dump will display, in case of a query result with multiple nested levels. The default value is **5**.
Style	A string that tells **DumpContainer** how to display the results. This string is expressed in CSS.

The **DumpContainer** object presents the results of a query by using its own **Dump** method, which can take two optional parameters:

- **Description** (string): Displays a description at the top of the results.
- **ToDataGrid** (Boolean): If a value of true is passed using this parameter, the results will be displayed in a data grid instead of Rich Text Format. The default value for this parameter is false.

Creating a custom Dump method

A custom **Dump** method can be created using the following code.

Code Listing 39: Custom Dump Method

```
public static void MyDump(this object content, string title = null, string
style = null,bool toDataGrid = false)
  {
      if (string.IsNullOrEmpty(title)) title = string.Empty;
      if (string.IsNullOrEmpty(style)) style = string.Empty;

      var myContainer = new DumpContainer {Content = content, Style =
style};
      myContainer.Dump(title,toDataGrid);
  }
```

The custom method defined in the previous sample is called **MyDump**. The method takes three optional parameters to store the title that will be displayed at the top of the results, to set the style (in CSS) that will be used to display the results, and a Boolean value indicating whether the results will be presented in a data grid or in Rich Text Format. When no title or styles are passed, the code ensures the **DumpContainer**'s **Dump** method receives empty strings for those parameters. In the case of the **toDataGrid** parameter, the default value passed to the method is **false** when no value is specified for the parameter.

Using the custom Dump method in a query

The following query uses the custom **Dump** method to change the background and the foreground colors on the fly for the results displayed.

Code Listing 40: Testing Custom Dump

```
var states = States.Take(20);
states.MyDump("Custom Dump","background-color: #F4D7A3; color:
#DE8900;",false);
```

As shown in this code, the string passed to the style parameter contains CSS code that defines colors for the background and for the text that will be displayed for the query results. These will be used instead of the colors defined in the default style sheet for text results (which can be customized in the **Results** tab of the **Preferences** dialog box).

Now, when the query is executed, the following output is displayed.

www.dbooks.org

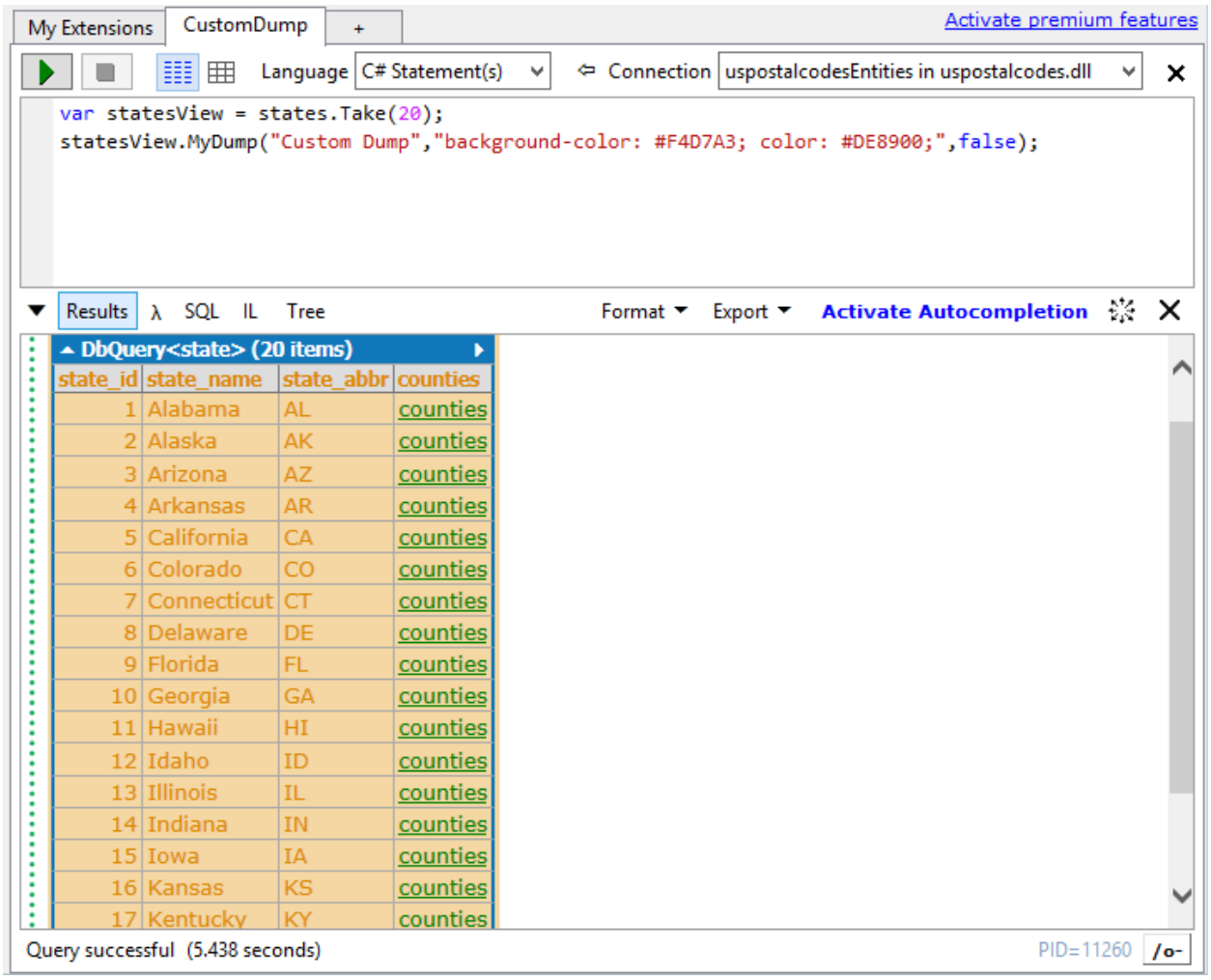


Figure 31: The Custom Dump Method in Action

Tip: The `font-family` CSS style property can be used as a part of the style string to change the font used for displaying results.

Chapter summary

LINQPad provides elements to extend its power by letting you write your own classes or methods to improve several features included in the product. Those elements include custom extensions and custom visualizers.

LINQPad allows you to write custom methods and make them available for all queries. This is done by using the **My Extensions** node tab, which is located beside the **My Queries** tree view. This node creates a query with a static class definition called **`MyExtensions`**, where all custom methods should be defined. The query should be compiled in order to make those methods available for all queries. This compilation will create a custom assembly called MyExtensions.FW40.dll inside the LINQPad Plugins folder.

It's possible to package custom extension methods as a plugin for LINQPad. This is done by creating a class library project in Visual Studio and adding a reference to LINQPad.exe into that project. Once the project is built, you should copy the assembly DLL, and any dependencies into LINQPad's plugins folder (usually Documents\LINQPad Plugins).

LINQPad can display the output generated by a query in two ways. The first and most common way is by calling the **`Dump`** extension method, which creates an instance of the **`DumpContainer`** object for displaying the results. The second way is by using the **`DisplayControl`** and **`DisplayWpfElement`** methods of the **`PanelManager`** static class, which creates an instance of an **`OutputPanel`** object and returns it to the calling process.

Both the **`PanelManager`** static class and the **`DumpContainer`** object can be accessed from LINQPad queries, and can also be used in custom extension methods declared in **`MyExtensions`** or packed into a custom assembly.

The **`PanelManager`** static class allows you to display Windows Forms controls or WPF elements as part of the query results, and the events associated to these controls can be trapped in queries, allowing the user to interact with them.

The **`DumpContainer`** object allows you to create a customized version of the **`Dump`** extension method. This customized version can be used, for example, to dynamically change the style used to display results.

www.dbooks.org

Chapter 5 Custom Data Context Drivers

What is a data context driver?

A data context driver is a mechanism that allows programmers to extend LINQPad in order to support several data sources. In other words, it is the way to add new drivers to the following dialog.

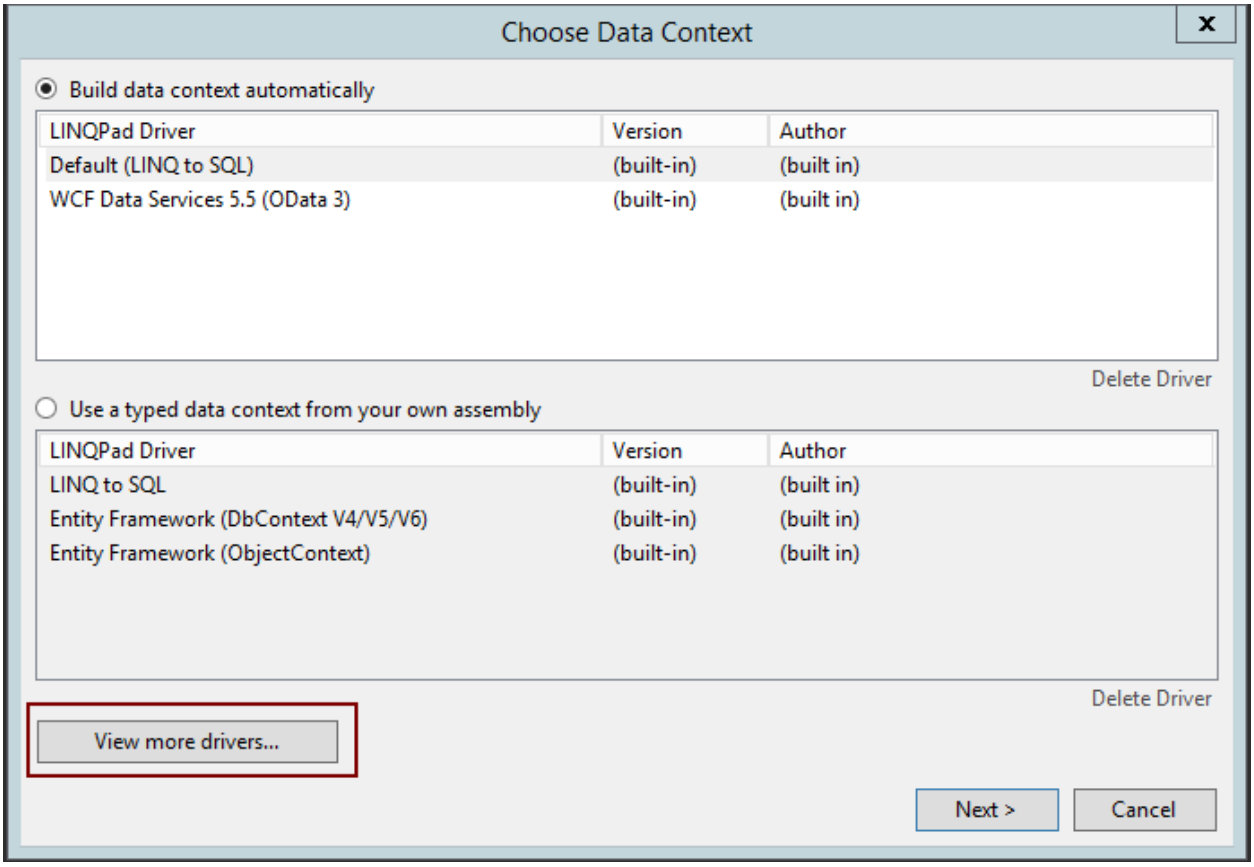


Figure 32: The Choose Data Context Dialog

Why is writing custom data context drivers useful?

LINQPad can query any data source without a custom data context driver, but in this case, the user must manually reference libraries, import custom namespaces, and formulate all queries like the following example.

Code Listing 41: Querying Data without a Custom Data Context Driver

```
var dataSource = new ItemsData();
```

```
(from i in dataSource.Items
   where i.Name.StartsWith ("D")
   select new { i.Name, i.UnitPrice }).Dump();
```

Instead, by using a custom data context driver, the query from the previous example turns into the following code.

Code Listing 42: Querying Data with a Custom Data Context Driver

```
(from i in dataSource.Items
   where i.Name.StartsWith ("D")
   select new { i.Name, i.UnitPrice }).Dump();
```

If we compare Code Listing 41 and Code Listing 42, the difference between them is the instantiation of the **`ItemsData()`** class. It might not seem like a big deal for small queries, but in large queries, the use of a data context driver helps you avoid writing many extra lines of code.

Data context drivers from the user's perspective

A data context driver is employed every time a user adds a connection to LINQPad. Those connections appear on the Connection's tree view area of the user interface.

To add a connection, click the **Add Connection** hyperlink located at the top of the Connection's tree view area. The dialog shown in Figure 32 pops up on the screen. If you click the **View More Drivers** button (which is highlighted in the same figure), the dialog displayed in the following figure appears.

www.dbooks.org

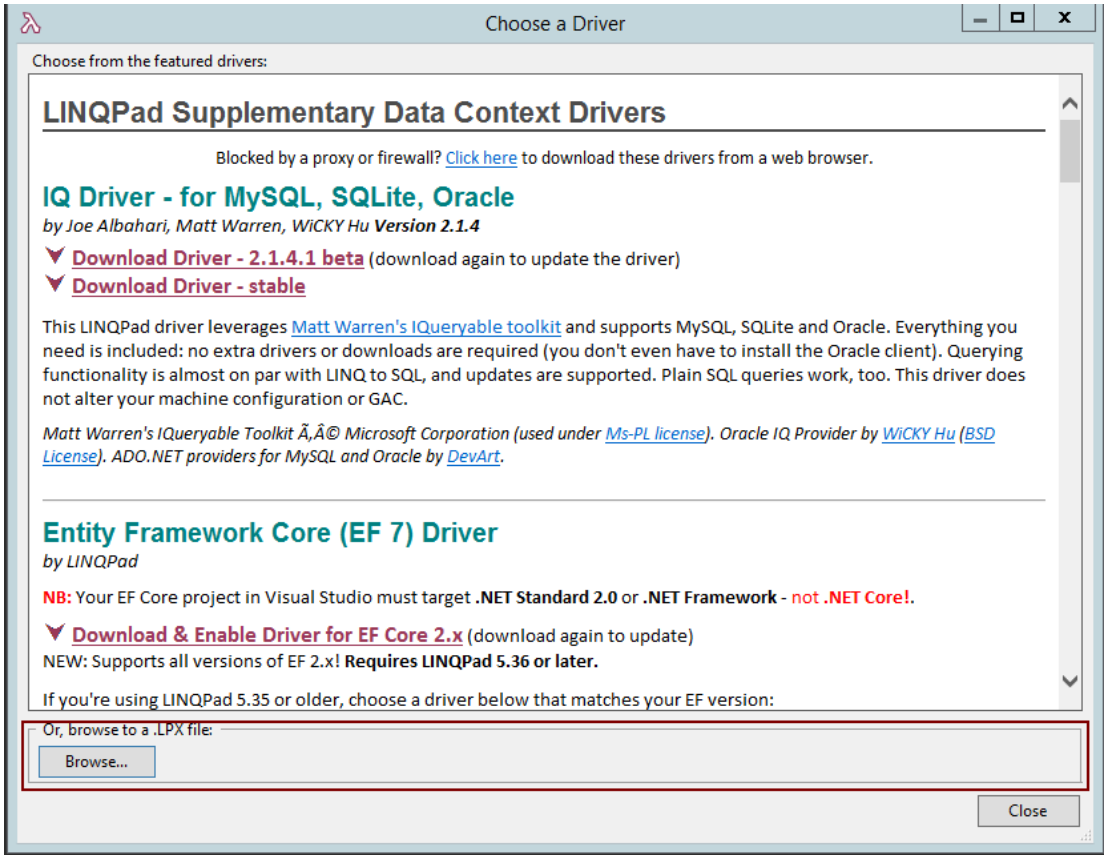


Figure 33: The Choose a Driver Dialog

The Drivers Gallery is the first item shown in the dialog from Figure 33. It's assumed that you're online—if you're not, this gallery is not displayed. You can click the **Download** hyperlink of any driver from the gallery to download and install it onto your computer. Clicking the **Browse** button located near the bottom of the dialog allows you to choose a driver previously deployed onto your computer. LINQPad asks for the location of the driver file in the following dialog.

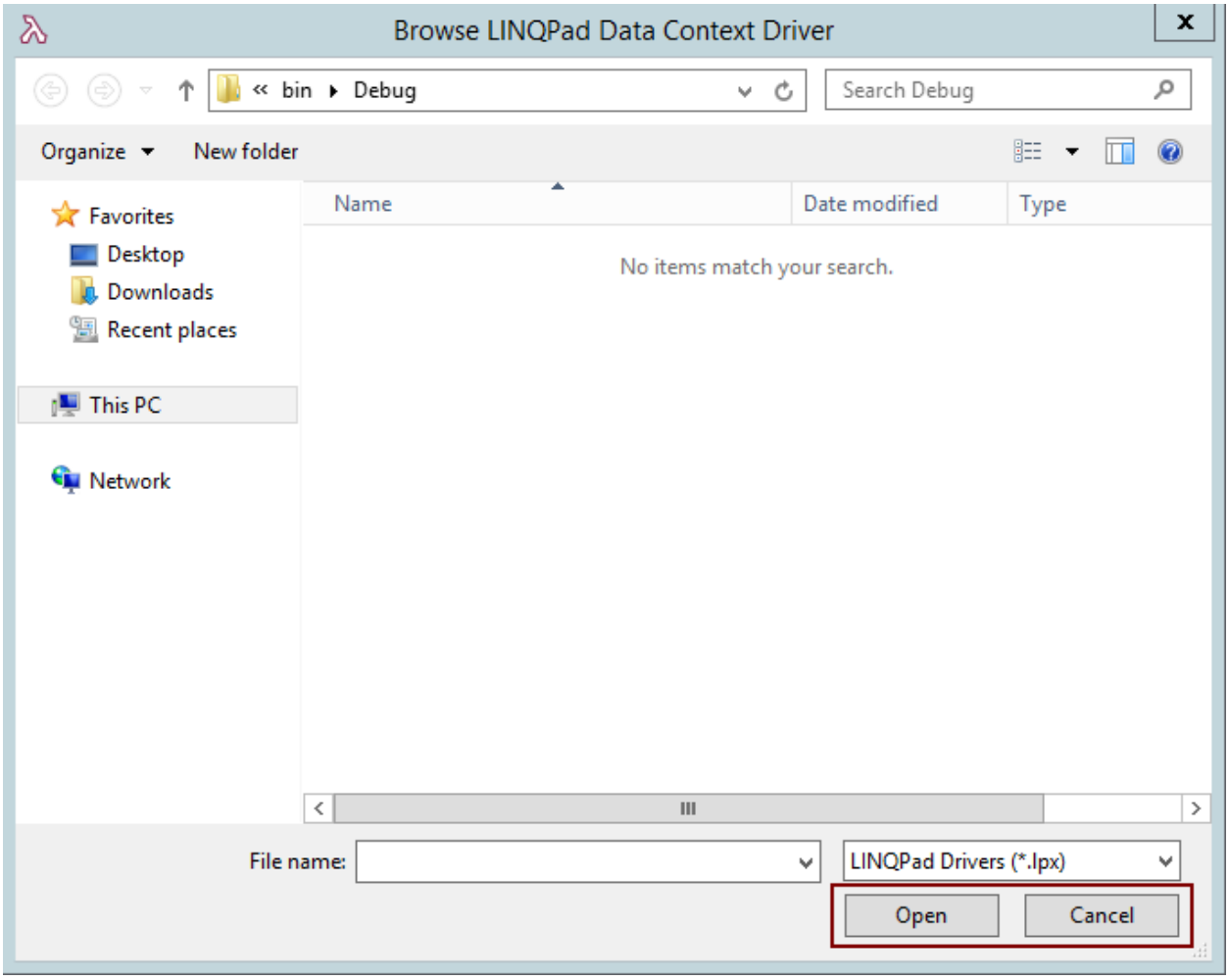


Figure 34: The Browse LINQPad Data Context Driver Dialog

Once the driver is located, you can click the file name, and then click **Open**. If you don't want to install a driver, you can abort the process by clicking **Cancel**.

Note: As shown in Figure 34, the extension for a data context driver file is .lpx (LINQPad extension).

When the driver is properly installed, it will become visible in the dialog shown in Figure 33.

Basic steps for writing a data context driver

Writing a data context driver is not trivial, but the process isn't as complex as you might guess. The basic steps for building a driver are:

1. Choose between writing a dynamic or a static driver (discussed later in this book).
2. Build a class project in Visual Studio that references LINQPad.

www.dbooks.org

3. Create classes derived from **`DynamicDataContextDriver`** or **`StaticDataContextDriver`**.
4. Implement a few abstract methods (and some virtual methods, optionally).
5. Zip the files generated by the class project (and all their dependencies) and change the extension from .zip to .lpx
6. Place the file in any place on your computer, so it can be found by using the Browse button shown in Figure 33.

LINQPad's extensibility model has been designed in a way that makes it quick to write a data context driver with basic functionality.

Special terms and conditions

There are no special terms or conditions for writing a data context driver, unless you plan to submit the driver to the LINQPad Driver Gallery, which is outside the scope of this book.

Basic concepts about data context drivers

There are some basics to cover before building a data context driver, to give you some perspective on how LINQPad treats data in the background. These basics are discussed in the following sections.

Connection

A connection relates to what you'll enter after clicking **Add Connection**. This is wider than the classic concept of a database connection; a LINQPad connection can refer to other types of data sources, such a web service URI. In addition, a LINQPad connection can include data context-specific details, such as pluralization and capitalization options.



Note: A LINQPad connection is represented by the IConnectionInfo interface.

Typed data contexts

A typed data context is a class with properties, fields, and methods that can be queried by the user. A typical example is a typed LINQ to SQL **`DataContext`** or a typed **`ObjectContext`** in Entity Framework, both of which are shown in the following code listing.

Code Listing 43: A Typical Data Context Example

```
public class TypedDataContext : DataContext
   {
      public IQueryable<Product> Products
      { get { return this.GetTable<Product>(); } }
```

```
      public IQueryable<InventoryEntry> InventoryEntries
      { get { return this.GetTable<InventoryEntry>();   } }
   }
```

We can define a **`DataContext`** without using a base class, as shown in the following code example.

Code Listing 44: A Typed Data Context with No Base Class

```
public class TypedDataContext
   {
      public IEnumerable<string> ProductNames
      public int[] Numbers;
      public void DoYourThings() { … }
   }
```

When writing data context drivers, it's mandatory to define a typed data context. There are two techniques for getting a typed data context:

- The driver can build one on the fly (dynamic driver).
- The driver can consume a typed data context already defined (static driver).

Dynamic and static drivers

When you click the **Add Connection** hyperlink in LINQPad, the dialog that appears shows two lists from which you can choose a driver:

- **Build Data Context Automatically**: This list shows the dynamic data context drivers currently installed in LINQPad.
- **Use a typed data context from your own assembly**: This list shows the static data context drivers installed in LINQPad.

So, why the separate lists? This is because different kinds of drivers work in different ways. The following list explains how dynamic and static drivers work:

- A **dynamic** driver builds the data context on the fly, either by generating code and compiling it after, or by using **`Reflection.Emit`**.
- A **static** driver requires the data context to be supplied by the user. So, when the user attempts to use a static driver, the connection dialog will prompt for a path where a custom assembly containing the typed data context is located, and after that, for the name of the type.

The advantage of using a dynamic driver is that it allows the user to query data without having to first write classes in a Visual Studio project. The advantage of using a static driver is that it gives the user a finer degree of control, and compatibility with typed data contexts written in previous projects.

www.dbooks.org

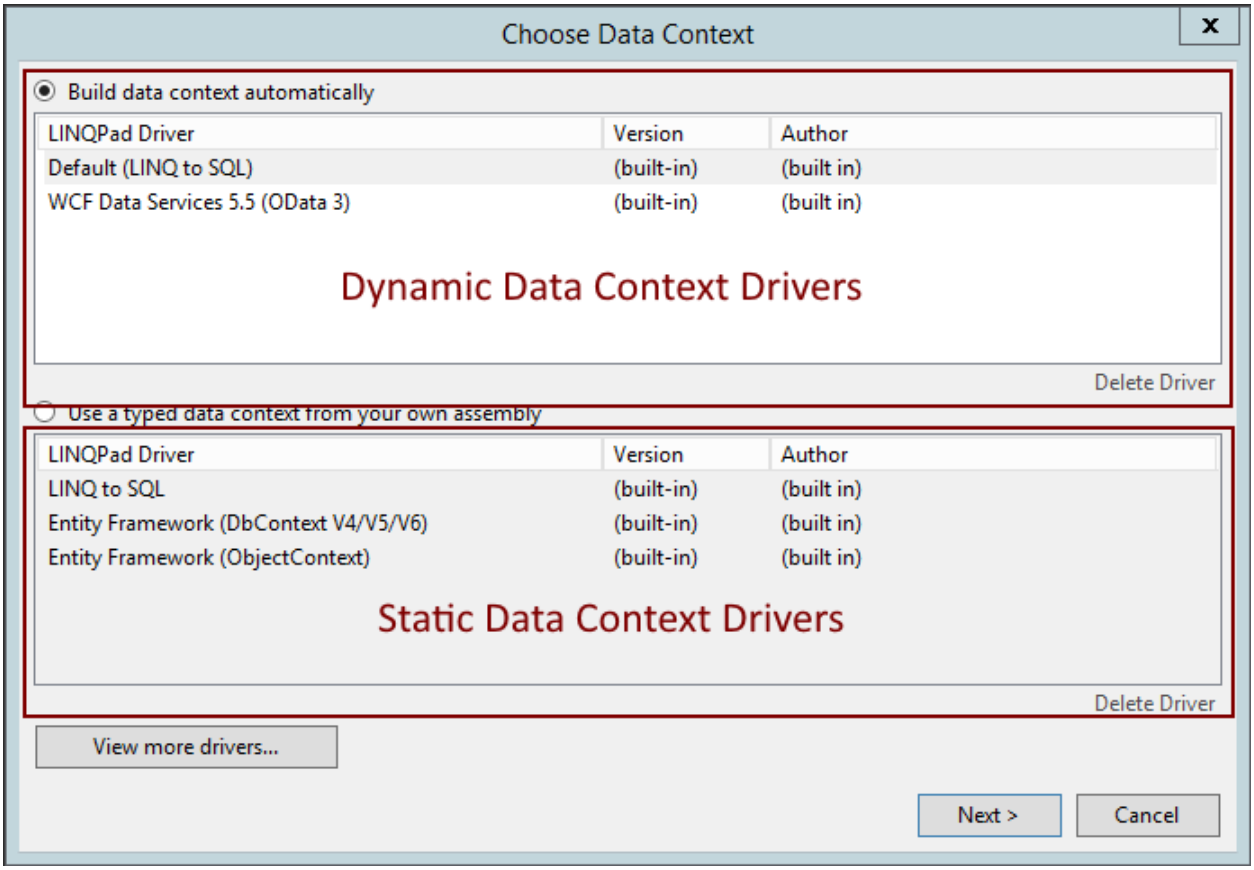


Figure 35: The Choose Data Context Dialog Showing the Two Kinds of Data Context Drivers

Note: Both kinds of drivers can be implemented in the same project.

How LINQPad works with queries

Every time the user writes a query, it doesn't explicitly refer to a particular data context. Instead, the user writes queries as shown in the following code example.

Code Listing 45: A Typical Query

```
from p in Products
   where p.Name.Contains ("CARD")
   select new { p.Name, p.UnitPrice }
```

When the user runs this query, LINQPad *subclasses* the typed data context by transforming the user's query into a method, as seen in the following code example.

Code Listing 46: A Query After Being Transformed by LINQPad

```
   public class UserQuery : TypedDataContext
```

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

   void RunUserAuthoredQuery()
   {
      (
         from p in Products
          where p.Name.Contains ("CARD")
         select new { p.Name, p.UnitPrice }
      )
      .Dump();
   }
}
```

After that, LINQPad calls the C# or VB compiler service (depending on the language selected in the user interface), compiles the code into a temporary assembly, creates an instance of the class, and finally, calls **`RunUserAuthoredQuery`**. This principle applies to both dynamic and static drivers.



Note: The `TypedDataContext` class of a data context driver must not be a sealed class and must have a public constructor.

Writing a data context driver

All the concepts discussed previously will be combined in the following sections in order to create a custom data context driver. Let's have some fun!

Setting up a project

The first step is to set up a Visual Studio project with the following properties:

- Project type: Class library (.NET Framework)
- Name: MyDataContextDriver
- Location: D:\LINQPad Samples
- Framework: .NET Framework 4.6.1
- Create directory for solution: Unchecked

The following figure shows the project configuration.

www.dbooks.org

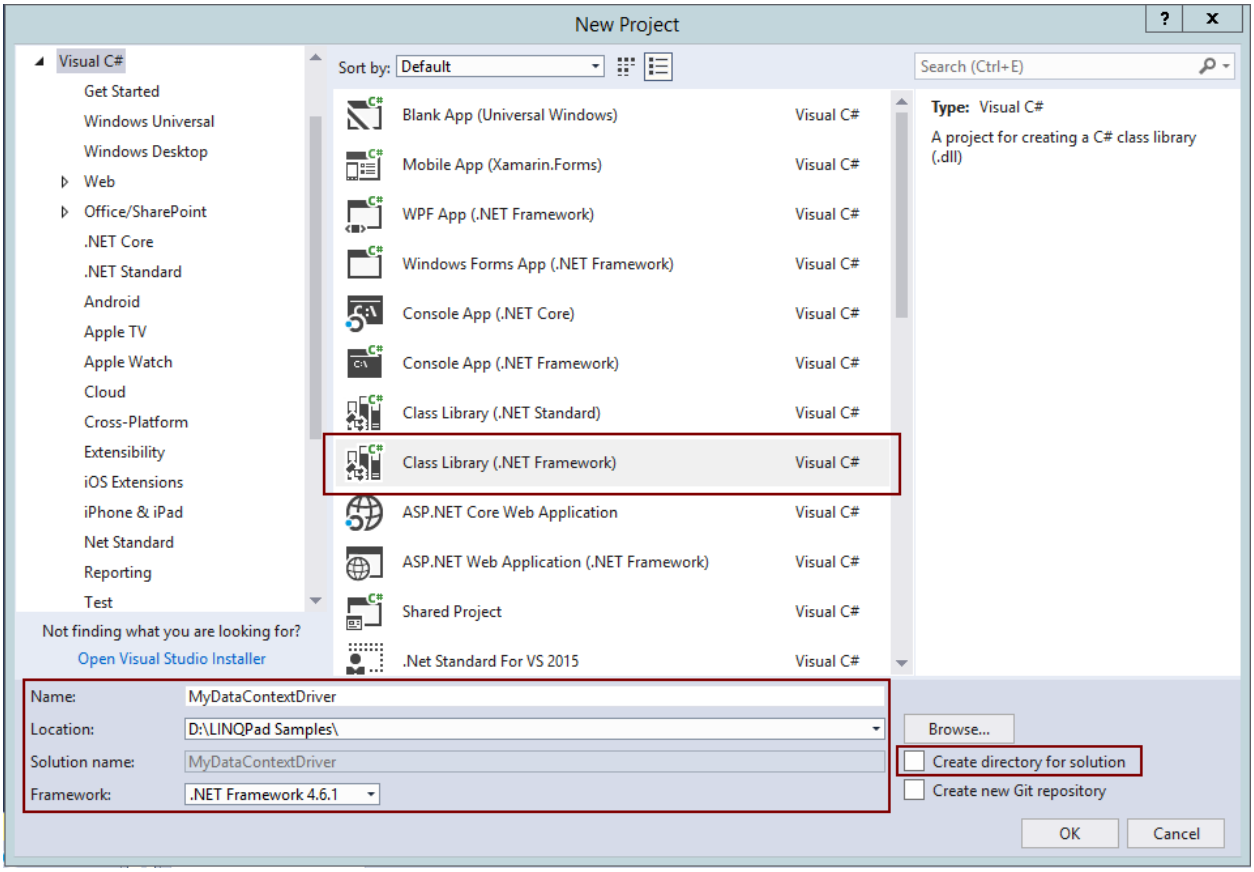


Figure 36: The Data Context Driver Project Properties

After configuring the project as shown in Figure 36, click **OK** to create the project files, which will be located at D:\LINQPad Samples\MyDataContextDriver.

The header file

Every time LINQPad tries to install a data context driver, it looks for a file named header.xml. If this file is not present in the deployment package, the installation process will fail. So, the first thing to do is add the corresponding **header.xml** file to the project. This file can be created in a text editor (like Notepad), according to the structure shown in the following code example.

Code Listing 47: The header.xml File

```
<?xml version="1.0" encoding="utf-8" ?>
<DataContextDriver>
   <MainAssembly> MyDataContextDriver.dll</MainAssembly>
   <SupportUri>http://YourURIHere</SupportUri>
</DataContextDriver>
```

The file should be saved at the project's root directory (in this case D:\LINQPad Samples\MyDataContextDriver). Next, it should be added to the project by right-clicking on the

project's name node in the Solution Explorer, and then clicking the **Add** > **Existing Item** option from the context menu.

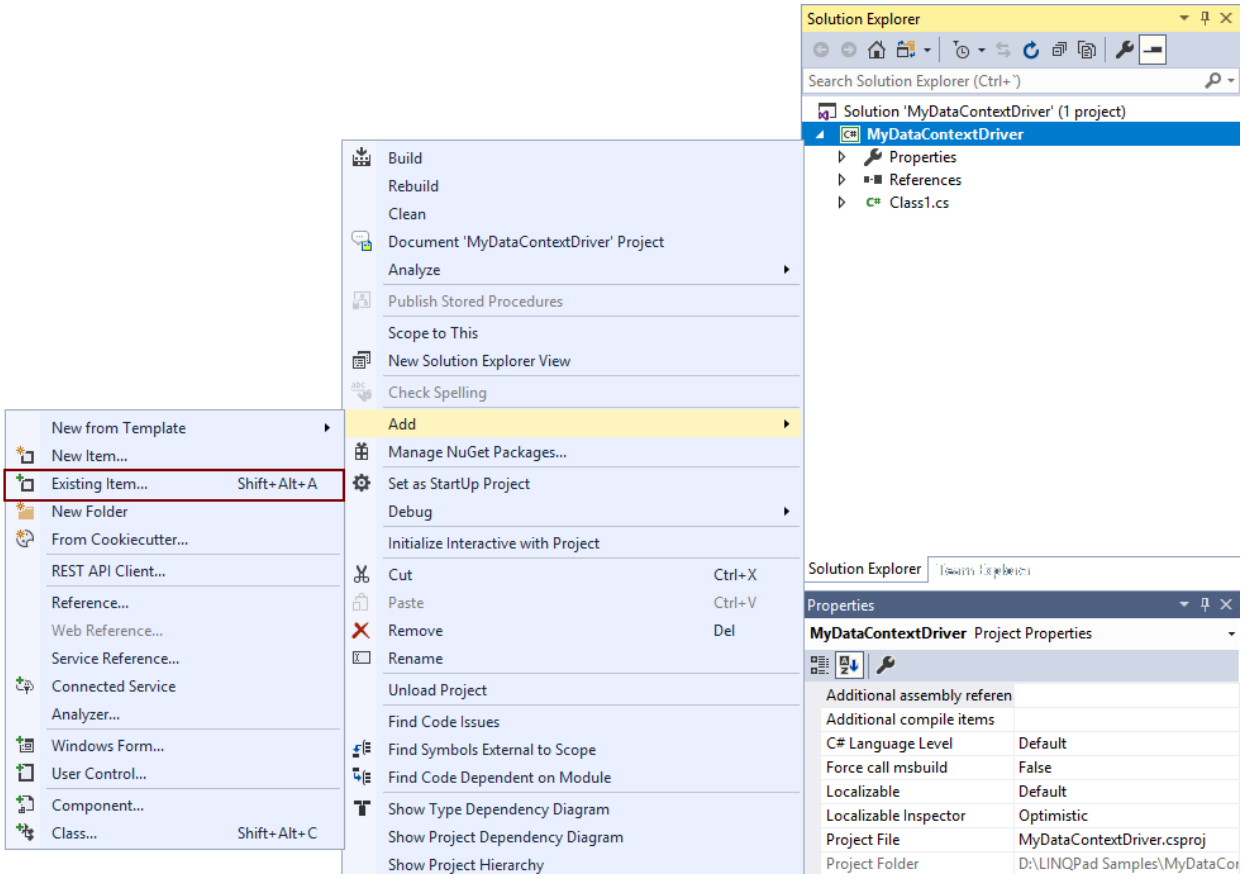


Figure 37: Adding an Existing Item to The Project

The Add Existing Item dialog will appear. Select **All Files (*.*)** in the file types list, and then go to the **D:\LINQPad Samples\MyDataContextDriver** folder to select the **header.xml** file. Finally, click the **Add** button to complete the process and make the file visible in the solution items tree.

As mentioned previously, the header.xml file is necessary for deploying the data context driver to ensure that the file will be copied to the project's output directory, along with the assembly files and dependencies. To accomplish this, change the file's properties by right-clicking the file name and going to the **Properties** window located at the bottom of the Solution Explorer. Set the **Build Action** property to **Content** and the **Copy to Output Directory** property to **Copy if newer**.

These steps are shown in the following figures.

www.dbooks.org

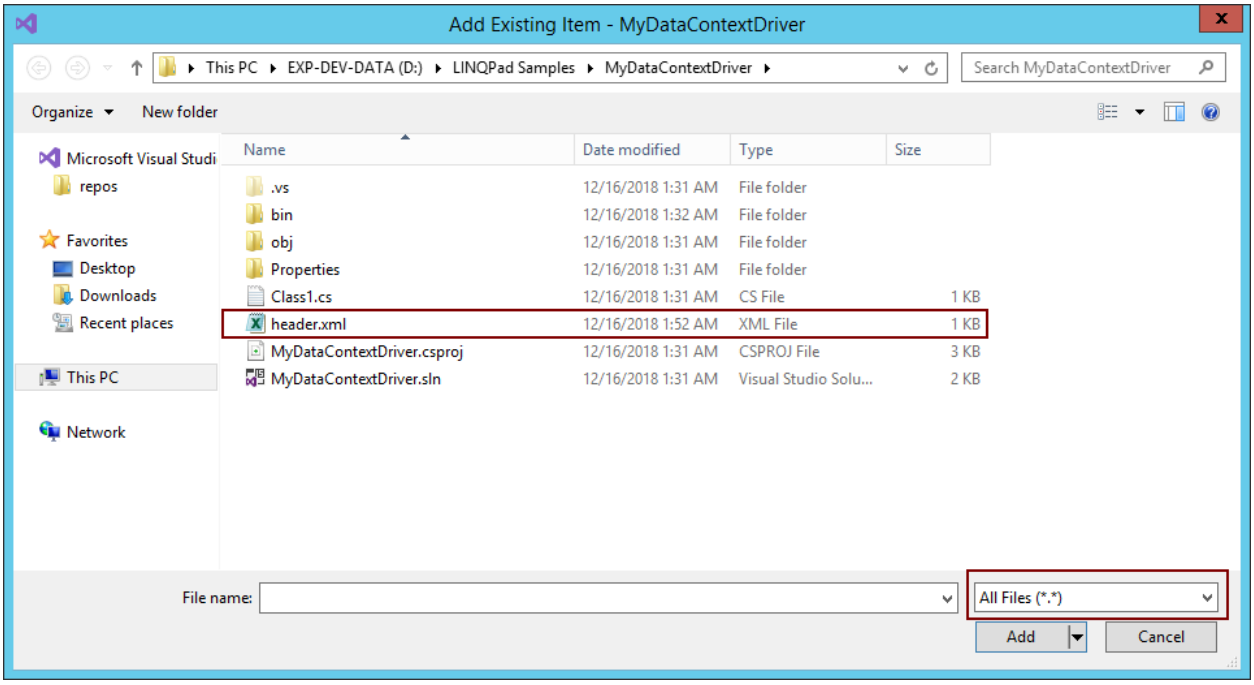


Figure 38: The Add Existing Item Dialog

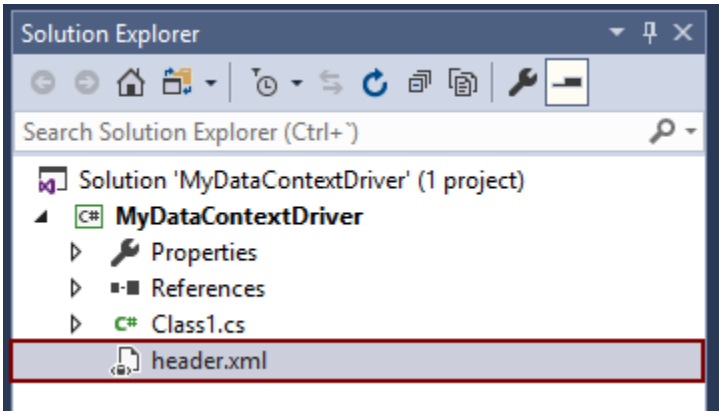


Figure 39: The header.xml File in the Project

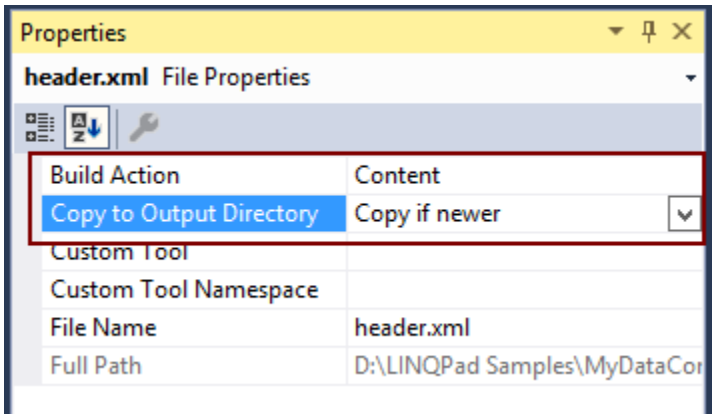


Figure 40: The header.xml File Properties

Making room for dynamic and static context drivers

As mentioned previously, a context driver project can hold both dynamic and static context drivers. To accomplish this, we'll create a pair of folders named **mydynamicdriver** and **mystaticdriver** within the project.

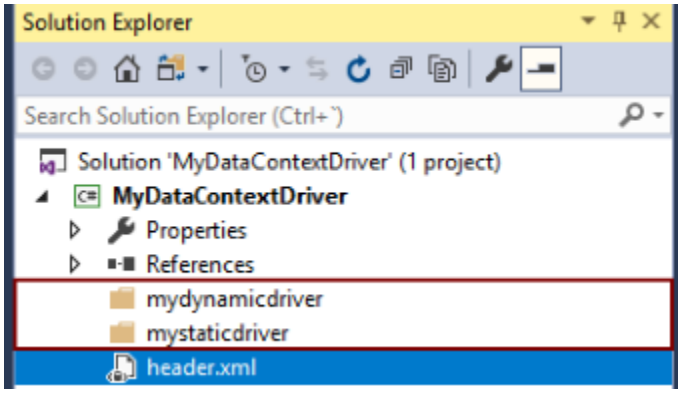


Figure 41: The Dynamic and Static Context Drivers Folders

Adding necessary references

At this point, the only reference needed is the one that points to LINQPad.exe. Let's add this reference to the project.

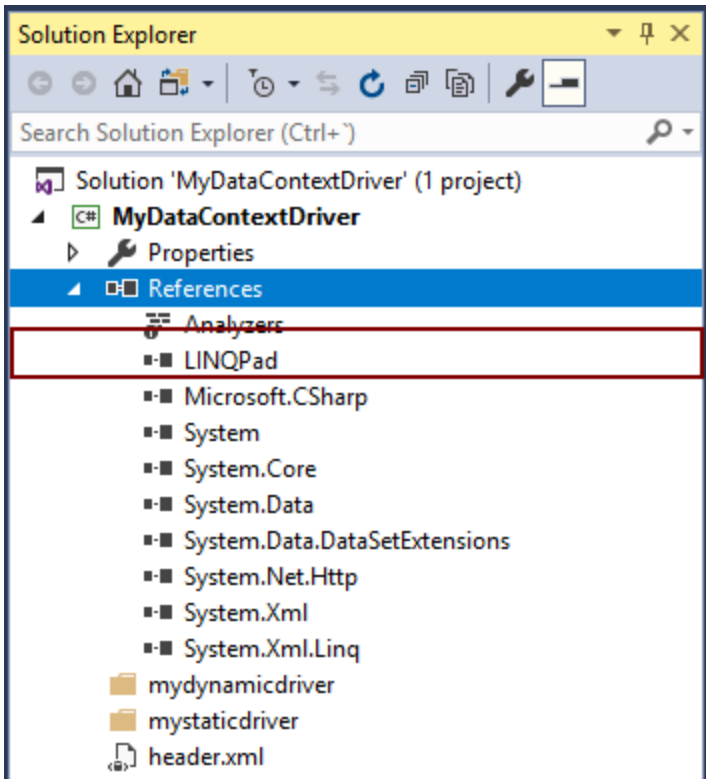


Figure 42: The Reference to LINQPad Added into the Project

www.dbooks.org

Writing the driver

There are two base classes for context drivers: **DynamicDataContextDriver** and **StaticDataContextDriver**. The name of each type tells us the kind of driver we can build with each one of them. Both types are derived from the base class **DataContextDriver**, which is defined in the **LINQPad.Extensibility.DataContext** namespace. This class defines some abstract members that will be implemented during project creation.

The first thing to do is add a code file for each kind of data context driver. These files will be named **MyDynamicDataContextDriver.cs** and **MyStaticDataContextDriver.cs**, and will define the **MyDynamicDataContextDriver** and **MyStaticDataContextDriver** classes, respectively. Since both classes are derived from the **DataContextDriver** class, Visual Studio automatically asks us to implement the abstract members defined in **DataContextDriver**. We simply let Visual Studio generate the implementation code automatically. Now, the code for both classes looks like the following examples.

Code Listing 48: MyDynamicDataContextDriver.cs

```
using System.Collections.Generic;
using System.Reflection;
using LINQPad.Extensibility.DataContext;

namespace MyDataContextDriver.mydynamicdriver
{
    public class MyDynamicDataContextDriver : DynamicDataContextDriver
    {
        public override string GetConnectionDescription(IConnectionInfo
cxInfo)
        {
            throw new System.NotImplementedException();
        }

        public override bool ShowConnectionDialog(IConnectionInfo cxInfo,
bool isNewConnection)
        {
            throw new System.NotImplementedException();
        }

        public override string Name => "My Dynamic Data Context Driver
(Demo)";
        public override string Author => "Getting the Most from LINQPad
Succinctly";

        public override List<ExplorerItem>
GetSchemaAndBuildAssembly(IConnectionInfo cxInfo, AssemblyName
assemblyToBuild, ref string nameSpace,ref string typeName)
        {
            throw new System.NotImplementedException();
        }
```

```
    }
}
```

Code Listing 49: MyStaticDataContextDriver.cs

```
using System;
using System.Collections.Generic;
using LINQPad.Extensibility.DataContext;

namespace MyDataContextDriver.mystaticdriver
{
    public class MyStaticDataContextDriver : StaticDataContextDriver
    {
        public override string GetConnectionDescription(IConnectionInfo
cxInfo)
        {
            throw new NotImplementedException();
        }

        public override bool ShowConnectionDialog(IConnectionInfo cxInfo,
bool isNewConnection)
        {
            throw new NotImplementedException();
        }

        public override string Name => "My Static Data Context Driver
(Demo)";
        public override string Author => "Getting the Most from LINQPad
Succinctly";

        public override List<ExplorerItem> GetSchema(IConnectionInfo
cxInfo, Type customType)
        {
            throw new NotImplementedException();
        }
    }
}
```

The code shown in the previous examples has already implemented the **Name** and **Author** abstract members. The **Name** member returns the name for the driver that is displayed in the LINQPad Driver column of the Choose Data Context dialog. The **Author** member, as it suggests, displays the name of the driver's author in the corresponding column of the Choose Data Context dialog.

www.dbooks.org

Writing the static driver

At this point, we will focus on building the static data context driver defined in **`MyStaticDataContextDriver.cs`**. To do so, we're going to follow the series of steps explained in the following sections.

The GetSchema method

The first step is implementing the **`GetSchema`** abstract method shown in Code Listing 48. This method is used to return a hierarchy of objects that will be displayed in the Schema Explorer (which is the Connection's tree view area of the user interface). This implementation is presented in the following code listing.

Code Listing 50: GetSchema Method Implementation

```
       public override List<ExplorerItem> GetSchema(IConnectionInfo cxInfo,
Type customType)
        {
            //First, we iterate throught all top level properties of
custumType
            var topLevelProps =
            (
                from prop in customType.GetProperties()
                where prop.PropertyType != typeof (string)

                // Get and display all properties of IEnumerable<T>
                let ienumerableOfT = prop.PropertyType.GetInterface
("System.Collections.Generic.IEnumerable`1")
                where ienumerableOfT != null

                orderby prop.Name

                select new ExplorerItem (prop.Name,
ExplorerItemKind.QueryableObject, ExplorerIcon.Table)
                {
                    IsEnumerable = true,
                    ToolTipText = FormatTypeName (prop.PropertyType,
false),

                    // Store entity type to the Tag property. This will be
used later.
                    Tag = ienumerableOfT.GetGenericArguments()[0]
                }

            ).ToList ();

            // Create a lookup element, associating each element type to
the properties of that type.
```

```
            // This will allow to build hyperlinks which let the user click
between relationships.
            var elementTypeLookup = topLevelProps.ToLookup (tp =>
(Type)tp.Tag);

            // Populate the properties of each entity
            foreach (var table in topLevelProps)
                table.Children = ((Type)table.Tag)
                    .GetProperties()
                    .Select (childProp => GetChildItem (elementTypeLookup,
childProp))
                    .OrderBy (childItem => childItem.Kind)
                    .ToList ();

            return topLevelProps;
        }

        private ExplorerItem GetChildItem (ILookup<Type, ExplorerItem>
elementTypeLookup, PropertyInfo childProp)
        {
            //if the property's type is in the list of entities, then we're
going to assume that it's a Many:1 or
            //1:1 referefence. It's not reliable to identify 1:1s
relationships purely from reflection.
            if (elementTypeLookup.Contains (childProp.PropertyType))
                return new ExplorerItem (childProp.Name,
ExplorerItemKind.ReferenceLink, ExplorerIcon.ManyToOne)
                {
                    HyperlinkTarget = elementTypeLookup
[childProp.PropertyType].First (),
                    ToolTipText = FormatTypeName (childProp.PropertyType,
true) // FormatTypeName is a LINQPad's helper method that returns a nicely
formatted type name.
                };

            //We're going to check if the property's type is a collection
of entities
            var ienumerableOfT = childProp.PropertyType.GetInterface
("System.Collections.Generic.IEnumerable`1");

            // If it isn't we return the Name and Type of the property as
an ExplorerItem
            if (ienumerableOfT == null) return new
ExplorerItem(childProp.Name + " (" + FormatTypeName(childProp.PropertyType,
false) + ")",ExplorerItemKind.Property, ExplorerIcon.Column);

            //Now, we're going to check if it is a 1:Many relationship
            var elementType = ienumerableOfT.GetGenericArguments()[0];
```


www.dbooks.org

```
            if (elementTypeLookup.Contains(elementType))
                return new ExplorerItem (childProp.Name,
ExplorerItemKind.CollectionLink, ExplorerIcon.OneToMany)
                {
                    HyperlinkTarget = elementTypeLookup [elementType].First
(),
                    ToolTipText = FormatTypeName (elementType, true)
                };

            //If it isn't, this is an ordinary property.
            return new ExplorerItem (childProp.Name + " (" + FormatTypeName
(childProp.PropertyType, false) + ")",
                ExplorerItemKind.Property, ExplorerIcon.Column);
        }
```

The first thing this code does is get a list of all top-level properties of the **`Type`** passed to the method. In the case of a database type, the entities (tables) of the database will be the top-level properties.

The next step to complete is the association of the elements for each top-level type (again, in the case of a database type, the elements will be the columns of each entity) into the **`Children`** property of each top-level element. The **`GetChildItem`** private method formats each descendant element in order to create a hyperlink when one of these descendants is involved in a relationship with another entity. The result of this implementation can be seen in the following figure.

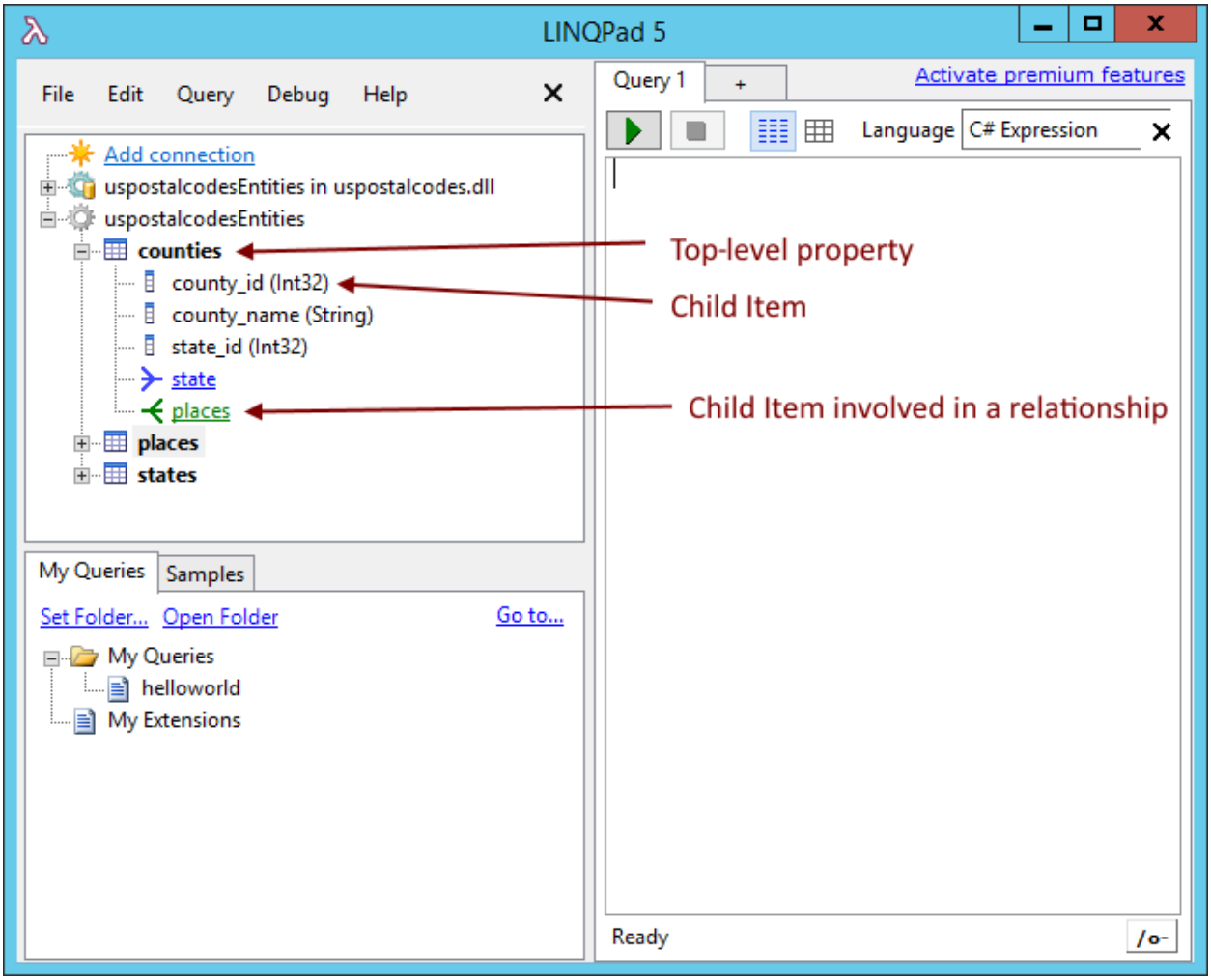


Figure 43: The GetSchema Method in Action

The ShowConnectionDialog method

This method must display a modal WPF dialog that will prompt the user for connection information. Since we're going to use WPF in the project, a reference to **`System.Xaml`** must be added to this. After that, we're going to right-click the **mystaticdriver** tree view's node and select the **Add** > **New Item** option from the context menu. Now, in the **Add New Item** dialog, look for the WPF section. The dialog should look like the following figure.

www.dbooks.org

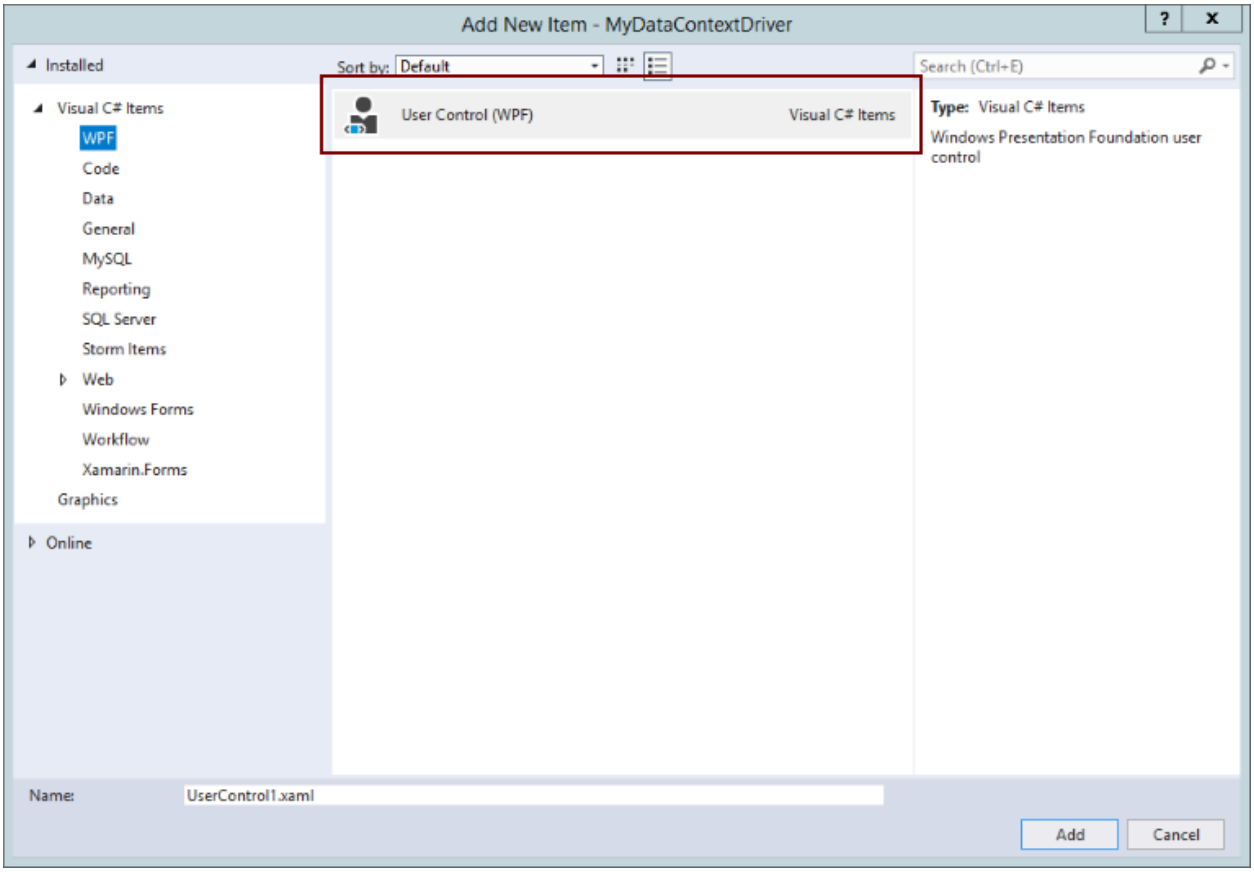


Figure 44: The WPF Section in the Add New Item Dialog

We can see in Figure 44 that only one option is available in the WPF section, and it's not a form, but a WPF User Control. To solve this problem, we're going to make a little tweak to the **MyDataContextDriver.csproj** file. Using a text editor, we'll add the following line to the global **`<PropertyGroup>`** section in the file.

Code Listing 51: Code Line for Tweaking the MyDataContextDriver.csproj File

```
<ProjectTypeGuids>{60dc8134-eba5-43b8-bcc9-bb4bc16c2548};{FAE04EC0-301F-
11D3-BF4B-00C04F79EFBC}</ProjectTypeGuids>
```

The **`<ProjectTypeGuids>`** tag tells Visual Studio which kind of project it's dealing with. In this case, **`{60dc8134-eba5-43b8-bcc9-bb4bc16c2548}`** stands for WPF, and **`{FAE04EC0-301F-11D3-BF4B-00C04F79EFBC}`** stands for a C# project. Now, the global **`<PropertyGroup>`** section of **`MyDataContextDriver.csproj`** should look like the following code example.

Code Listing 52: The Global <PropertyGroup> Section Tweaked

```
  <PropertyGroup>
    <Configuration Condition=" '$(Configuration)' == ''
">Debug</Configuration>
    <Platform Condition=" '$(Platform)' == '' ">AnyCPU</Platform>
```

```
    <ProjectGuid>{59F6858D-5BAC-4A9F-9D84-40E012AFC36C}</ProjectGuid>
    <ProjectTypeGuids>{60dc8134-eba5-43b8-bcc9-bb4bc16c2548};{FAE04EC0-
301F-11D3-BF4B-00C04F79EFBC}</ProjectTypeGuids>
    <OutputType>Library</OutputType>
    <AppDesignerFolder>Properties</AppDesignerFolder>
    <RootNamespace>MyDataContextDriver</RootNamespace>
    <AssemblyName>MyDataContextDriver</AssemblyName>
    <TargetFrameworkVersion>v4.6.1</TargetFrameworkVersion>
    <FileAlignment>512</FileAlignment>
    <Deterministic>true</Deterministic>
  </PropertyGroup>
```

Now, when we select **Add** > **New Item** from the context menu, the dialog will look like the following figure.

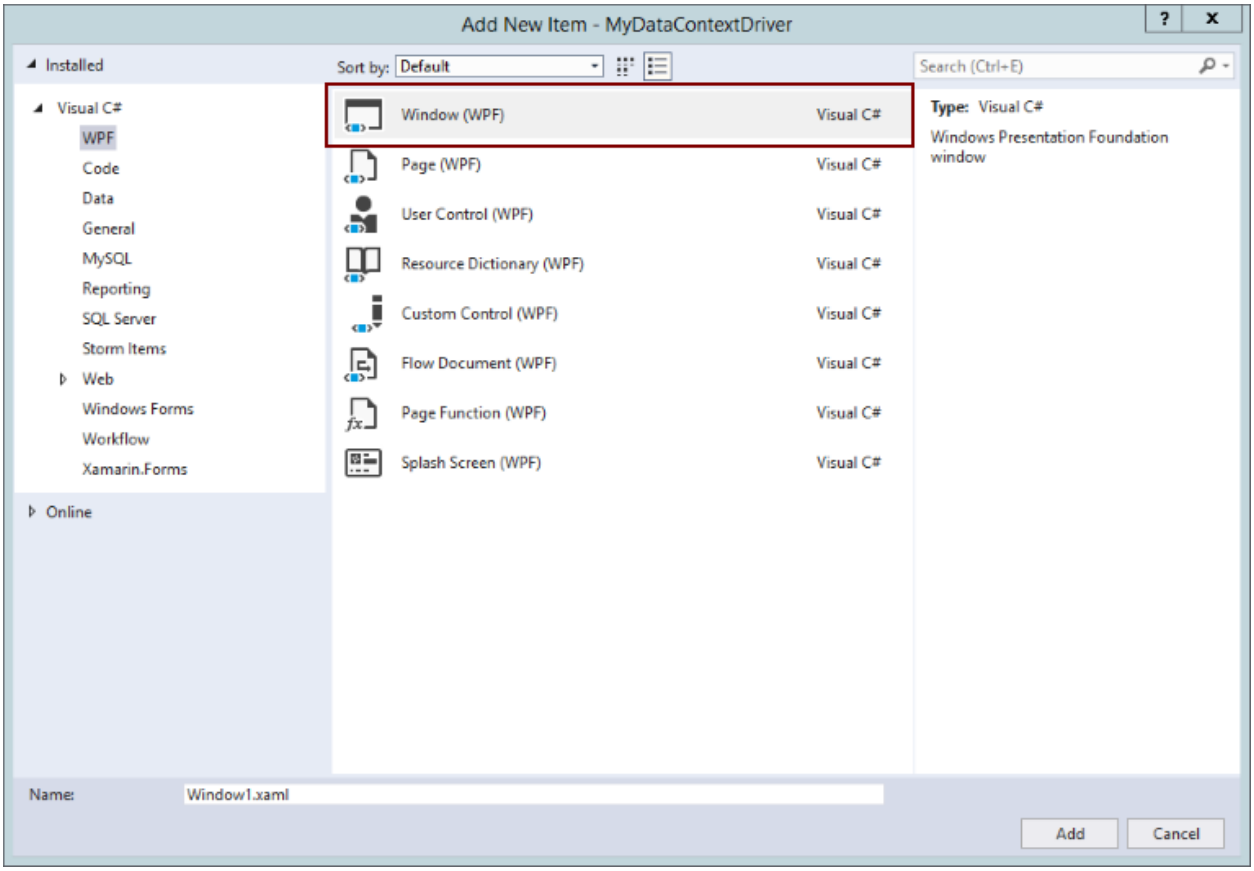


Figure 45: The WPF Section After Tweaking MyDataContextDriver.csproj

To add the WPF form, we're going to type **ConnectionDialog** into the **Name** text box, and then click the **Add** button. Now, a ConnectionDialog.xaml file entry will be displayed in the Solution Explorer. To add the necessary controls to enter connection information, we're going to right-click the entry and select **View Designer** from the context menu. Once the designer is displayed on the screen, we'll place those controls using the designer's toolbox, dragging each one of

www.dbooks.org

them into the form. The final result of these actions will be a form that will look like the following figure.

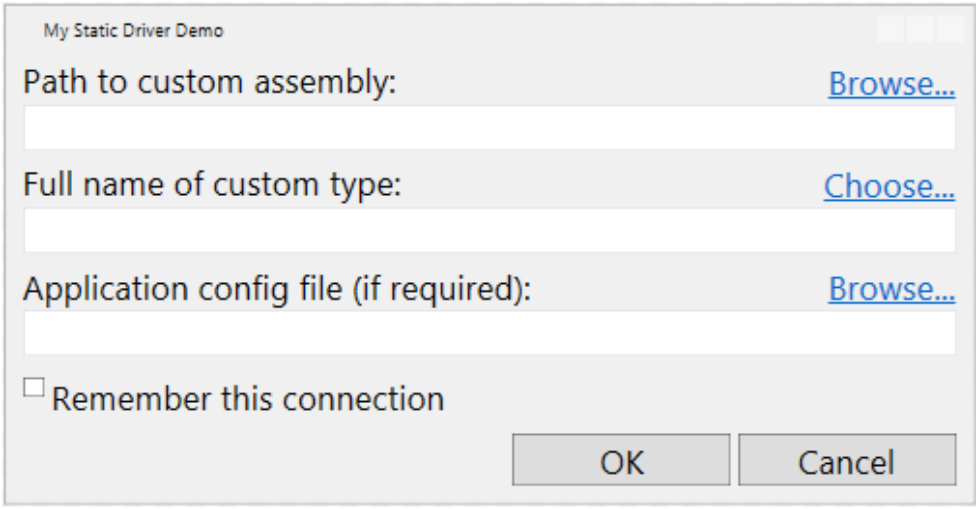


Figure 46: The Connection Dialog in the WPF Form's Designer

Now, we will create the event handlers for every control in the form (except labels). The resulting code is shown in the following example.

Code Listing 53: Interaction Logic for ConnectionDialog

```
using System;
using System.Diagnostics.CodeAnalysis;
using System.IO;
using System.Windows;
using LINQPad.Extensibility.DataContext;

namespace MyDataContextDriver.mystaticdriver
{
    /// <inheritdoc cref="Window" />
    /// <summary>
    /// Interaction logic for ConnectionDialog.xaml
    /// </summary>
    public partial class ConnectionDialog
    {
        readonly IConnectionInfo _cxInfo;

        public ConnectionDialog(IConnectionInfo cxInfo)
        {
            _cxInfo = cxInfo;
            DataContext = cxInfo.CustomTypeInfo;
            InitializeComponent();
        }

        private void BrowseAssembly(object sender, RoutedEventArgs e)
```

```
        {
            var dialog = new Microsoft.Win32.OpenFileDialog()
            {
                Title = "Choose custom assembly",
                DefaultExt = ".dll",
            };

            if (dialog.ShowDialog() == true)
_cxInfo.CustomTypeInfo.CustomAssemblyPath = dialog.FileName;
        }

        [SuppressMessage("ReSharper", "CoVariantArrayConversion")]
        private void ChooseType(object sender, RoutedEventArgs e)
        {
            var assemPath = _cxInfo.CustomTypeInfo.CustomAssemblyPath;
            if (assemPath.Length == 0)
            {
                MessageBox.Show("First enter a path to an assembly.");
                return;
            }

            if (!File.Exists(assemPath))
            {
                MessageBox.Show("File '" + assemPath + "' does not
exist.");
                return;
            }

            string[] customTypes;
            try
            {
                customTypes =
_cxInfo.CustomTypeInfo.GetCustomTypesInAssembly();
            }
            catch (Exception ex)
            {
                MessageBox.Show("Error obtaining custom types: " +
ex.Message);
                return;
            }

            if (customTypes.Length == 0)
            {
                MessageBox.Show("There are no public types in that
assembly.");
                return;
            }
```


www.dbooks.org

```
            var result =
(string)LINQPad.Extensibility.DataContext.UI.Dialogs.PickFromList("Choose
Custom Type", customTypes);
            if (result != null) _cxInfo.CustomTypeInfo.CustomTypeName =
result;
        }

        private void BrowseAppConfig(object sender, RoutedEventArgs e)
        {
            var dialog = new Microsoft.Win32.OpenFileDialog()
            {
                Title = "Choose application config file",
                DefaultExt = ".config",
            };

            if (dialog.ShowDialog() == true) _cxInfo.AppConfigPath =
dialog.FileName;
        }

        private void BtnOK_Click(object sender, RoutedEventArgs e) =>
DialogResult = true;
    }
}
```

The most significant member of this code is **_cxInfo**, which is an **IConnectionInfo** type. This object will hold the connection information using the attributes detailed in the following table.

Table 3: IConnectionInfo Attributes

Attribute	Description
CustomTypeInfo.CustomAssemblyPath	Path to the custom assembly file containing the entity types that will be displayed in the Schema Explorer.
CustomTypeInfo.GetCustomTypesInAssembly()	Retrieves a list of all public custom types exposed by the custom assembly.
CustomTypeInfo.CustomTypeName	Name of the custom type that will be used to populate the Schema Explorer.
AppConfigPath	Path to the configuration file associated with the custom assembly being used (.config file extension).

Once the form's design is finished, we'll use it in the **ShowConnectionDialog** method, as shown in the following code example.

Code Listing 54: ShowConnectionDialog Implemented

```
public override bool ShowConnectionDialog(IConnectionInfo cxInfo, bool
isNewConnection) => new ConnectionDialog(cxInfo).ShowDialog() == true;
```

Now, if you add a connection using our driver, the connection dialog will appear, asking for the information needed to create it. If you click **Cancel**, the **ShowDialog** method will return **false**, and the process will be terminated. Otherwise, LINQPad will populate the Schema Explorer with the custom type's entities and their attributes.

The GetConnectionDescription method

Finally, we need to implement the **GetConnectionDescription** method in order to display the selected custom type's name in the Schema Explorer. This will be accomplished using the code in the following example.

Code Listing 55: GetConnectionDescription Method Implemented

```
// We'll use the description of the custom type and its
// assembly for static drivers

public override string GetConnectionDescription(IConnectionInfo cxInfo) =>
cxInfo.CustomTypeInfo.GetCustomTypeDescription();
```

We can see the use of **CustomTypeInfo.GetCustomTypeDescription**, which is the method that retrieves the name of the custom type selected for populating the Schema Explorer.

The completed static data context driver

The static data context driver will be obtained as a result of completing all the previous steps. The final code is shown in the following example.

Code Listing 56: Completed MyStaticDataContextDriver

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Reflection;
using LINQPad.Extensibility.DataContext;

namespace MyDataContextDriver.mystaticdriver
{
    public class MyStaticDataContextDriver : StaticDataContextDriver
    {
        // We'll use the description of the custom type and its assembly
for Static Drivers
```


www.dbooks.org

```
        public override string GetConnectionDescription(IConnectionInfo
cxInfo) => cxInfo.CustomTypeInfo.GetCustomTypeDescription();

        public override bool ShowConnectionDialog(IConnectionInfo cxInfo,
bool isNewConnection) => new ConnectionDialog(cxInfo).ShowDialog() == true;

        public override string Name => "My Static Data Context Driver
(Demo)";
        public override string Author => "Getting the Most from LINQPad
Succinctly";

        public override List<ExplorerItem> GetSchema(IConnectionInfo
cxInfo, Type customType)
        {
            // First, we iterate through all top-level properties of
customType
            var topLevelProps =
            (
                from prop in customType.GetProperties()
                where prop.PropertyType != typeof (string)

                // Get and display all properties of IEnumerable<T>
                let ienumerableOfT = prop.PropertyType.GetInterface
("System.Collections.Generic.IEnumerable`1")
                where ienumerableOfT != null

                orderby prop.Name

                select new ExplorerItem (prop.Name,
ExplorerItemKind.QueryableObject, ExplorerIcon.Table)
                {
                    IsEnumerable = true,
                    ToolTipText = FormatTypeName (prop.PropertyType,
false),

                    // Store entity type to the Tag property. This will be
used later.
                    Tag = ienumerableOfT.GetGenericArguments()[0]
                }

            ).ToList ();

            // Create a lookup element, associating each element type to
the properties of that type.
            // This will allow us to build hyperlinks that let the user
click between relationships.
            var elementTypeLookup = topLevelProps.ToLookup (tp =>
(Type)tp.Tag);
```

```
            // Populate the properties of each entity
            foreach (var table in topLevelProps)
                table.Children = ((Type)table.Tag)
                    .GetProperties()
                    .Select (childProp => GetChildItem (elementTypeLookup,
childProp))
                    .OrderBy (childItem => childItem.Kind)
                    .ToList ();

            return topLevelProps;
        }

        private ExplorerItem GetChildItem (ILookup<Type, ExplorerItem>
elementTypeLookup, PropertyInfo childProp)
        {
            // If the property's type is in the list of entities, then
we're going to assume that it's a Many:1 or
            // 1:1 reference. It's not reliable to identify 1:1
relationships purely from reflection.
            if (elementTypeLookup.Contains (childProp.PropertyType))
                return new ExplorerItem (childProp.Name,
ExplorerItemKind.ReferenceLink, ExplorerIcon.ManyToOne)
                {
                    HyperlinkTarget = elementTypeLookup
[childProp.PropertyType].First (),
                    ToolTipText = FormatTypeName (childProp.PropertyType,
true) // FormatTypeName is LINQPad's helper method that returns a nicely
formatted type name.
                };

            // We're going to check if the property's type is a collection
of entities
            var ienumerableOfT = childProp.PropertyType.GetInterface
("System.Collections.Generic.IEnumerable`1");

            // If it isn't, we return the Name and Type of the property as
an ExplorerItem
            if (ienumerableOfT == null) return new
ExplorerItem(childProp.Name + " (" + FormatTypeName(childProp.PropertyType,
false) + ")",ExplorerItemKind.Property, ExplorerIcon.Column);

            // Now, we're going to check if it is a 1:Many relationship
            var elementType = ienumerableOfT.GetGenericArguments()[0];

            if (elementTypeLookup.Contains(elementType))
                return new ExplorerItem (childProp.Name,
ExplorerItemKind.CollectionLink, ExplorerIcon.OneToMany)
                {
```


www.dbooks.org

```
                    HyperlinkTarget = elementTypeLookup [elementType].First
(),
                    ToolTipText = FormatTypeName (elementType, true)
                };

            // If it isn't, this is an ordinary property.
            return new ExplorerItem (childProp.Name + " (" + FormatTypeName
(childProp.PropertyType, false) + ")",
                ExplorerItemKind.Property, ExplorerIcon.Column);
        }
    }
}
```

Deploying the data context driver

To deploy the data context driver, we need to perform the following steps:

1. Build the assembly by selecting **Build** > **Build Solution** or **Rebuild Solution** from the Visual Studio menu bar.
2. Zip MyDataContextDriver.dll and header.xml files into a single file named mycontextdatadriver.zip.
3. Rename mycontextdatadriver.zip to mycontextdatadriver.lpx.
4. Copy **mycontextdatadriver.lpx** to any desired folder in the file system.

Installing the data context driver

Now, it's time to install our developed driver into LINQPad. To complete this action, it's mandatory to try adding a new connection by clicking the **Add Connection** hyperlink in the user interface. As seen at the beginning of this chapter, LINQPad will show the Choose Data Context dialog. In order to work with custom drivers, we need to click the **View More Drivers** button. This will bring up the **Choose a Driver** dialog, as seen in the "Data context drivers from the user's perspective" section. Once the dialog is displayed, we'll click the **Browse** button, and the Browse LINQPad Data Context Driver dialog will be shown.

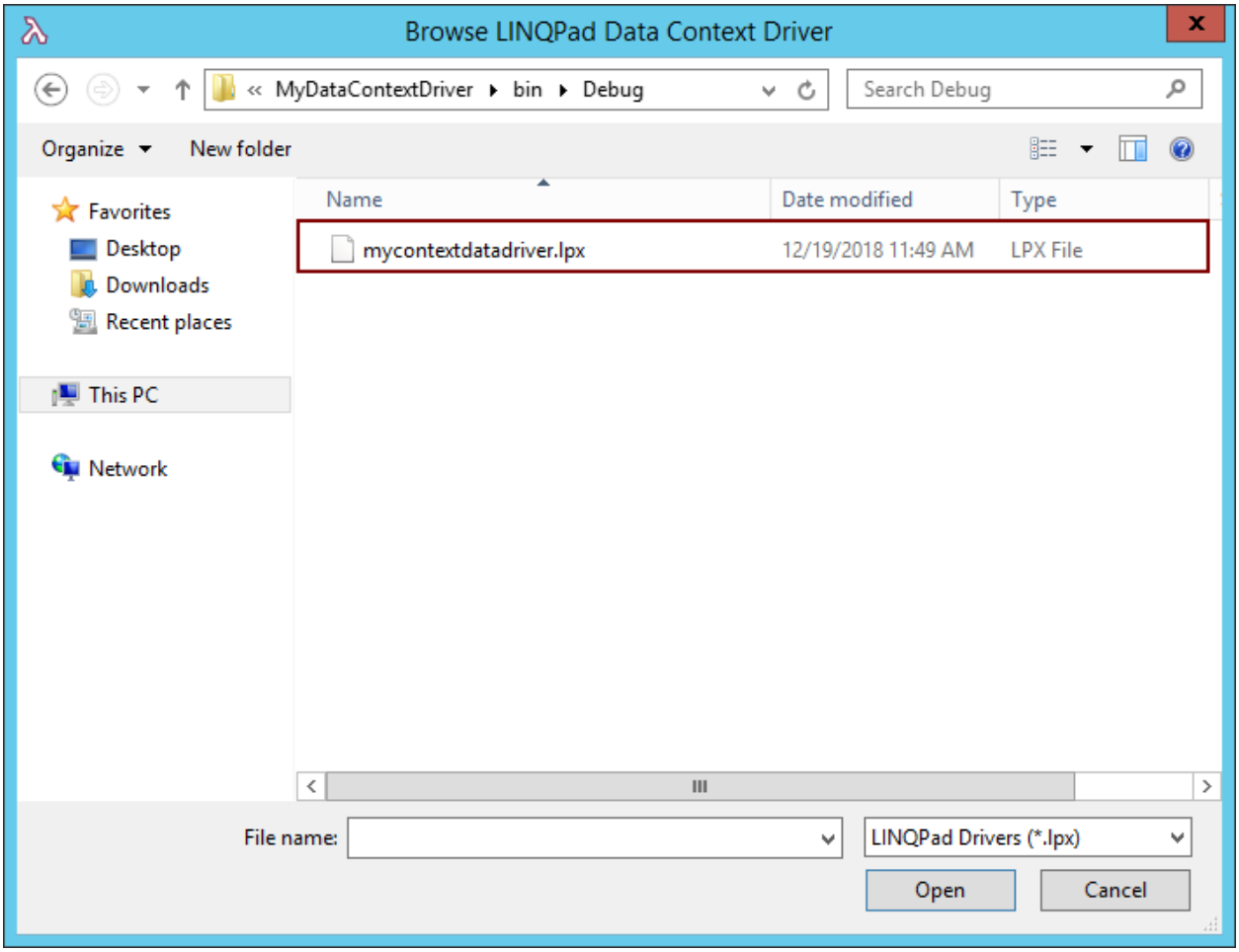


Figure 47: Browse LINQPad Data Context Driver Dialog Displaying Our Custom Context Data Driver

Next, we need to locate the file for our custom driver and double-click its name. This will open the file and install the driver. When finished, LINQPad will show the following dialog.

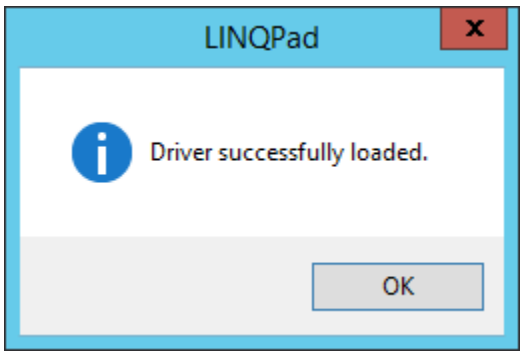


Figure 48: Context Data Driver Successfully Installed

Now, the **Choose Data Context** dialog will display our custom data context drivers as available options.

www.dbooks.org

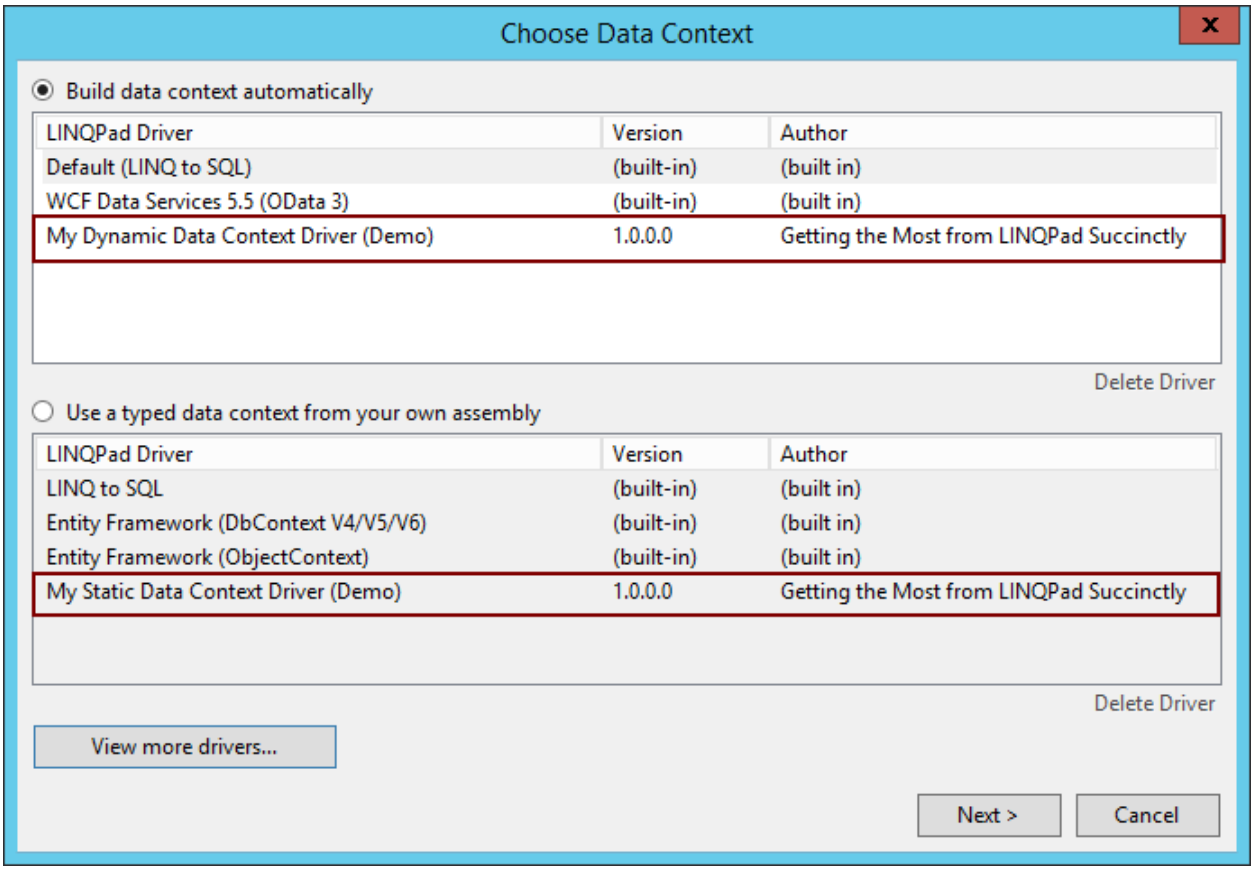


Figure 49: Our Custom Context Data Drivers in the Choose Data Context Dialog

Testing our static data context driver

To test our driver, select the **Use a typed data context from your own assembly** option from the **Choose Data Context** dialog, and choose **My Static Data Context Driver (Demo)** from the drivers list. Click **Next** to continue, and the connection dialog will be displayed.

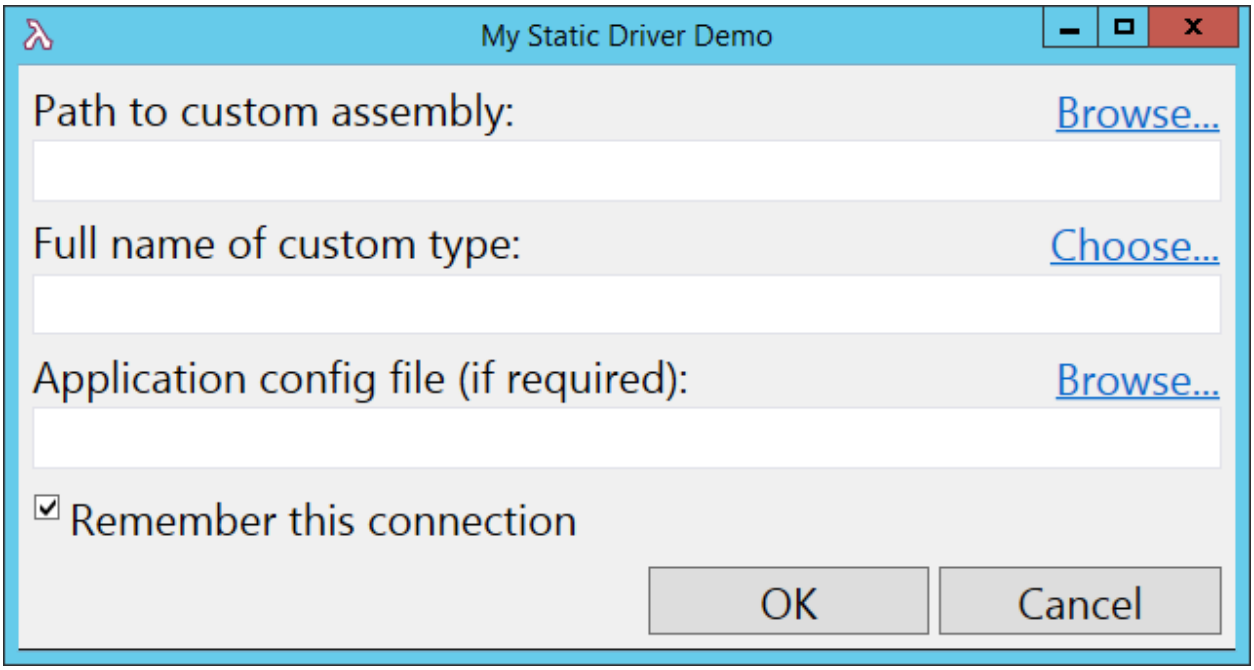


Figure 50: The Connection Dialog of Our Custom Context Data Driver

Selecting a custom assembly

We're going to use the Entity Data Model assembly created in Chapter 2 to test our driver. So, after we browse the file system and select the **uspostalcodes.dll** assembly, the connection dialog should look like the following figure.

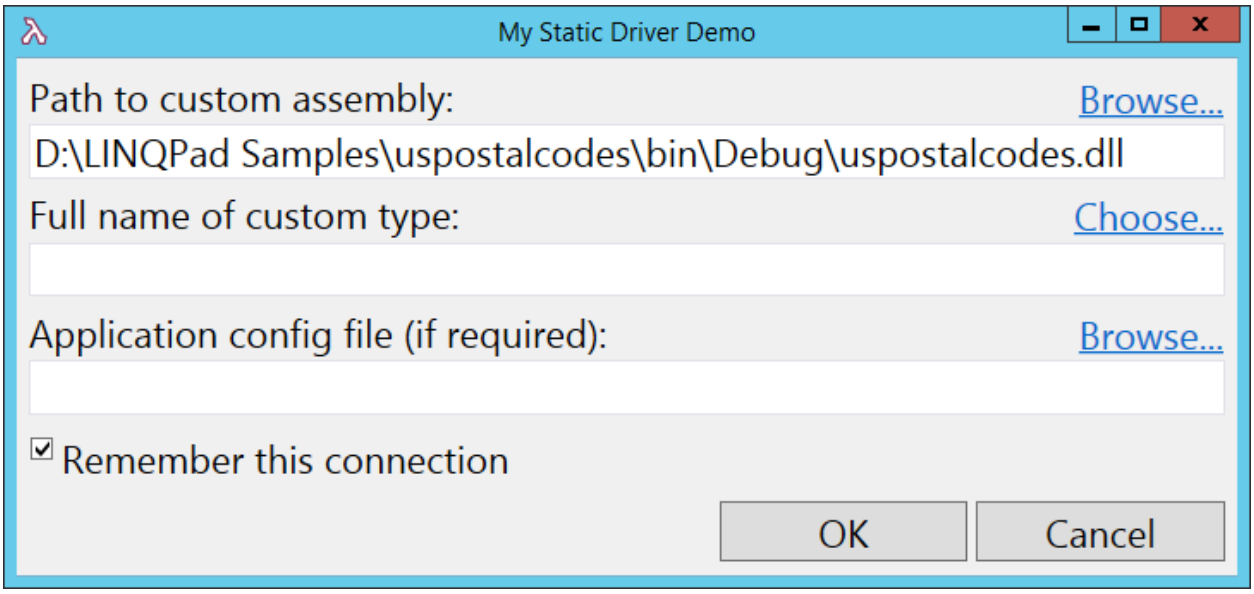


Figure 51: The uspostalcodes.dll Assembly Selected

Now, we're going to browse the file system and locate the **uspostalcodes.dll.config** file, which is the application configuration file required by the connection dialog. Finally, we're going to

www.dbooks.org

choose the **uspostalcodes.uspostalcodesEntities** custom type. This type will be used to populate the Schema Explorer.

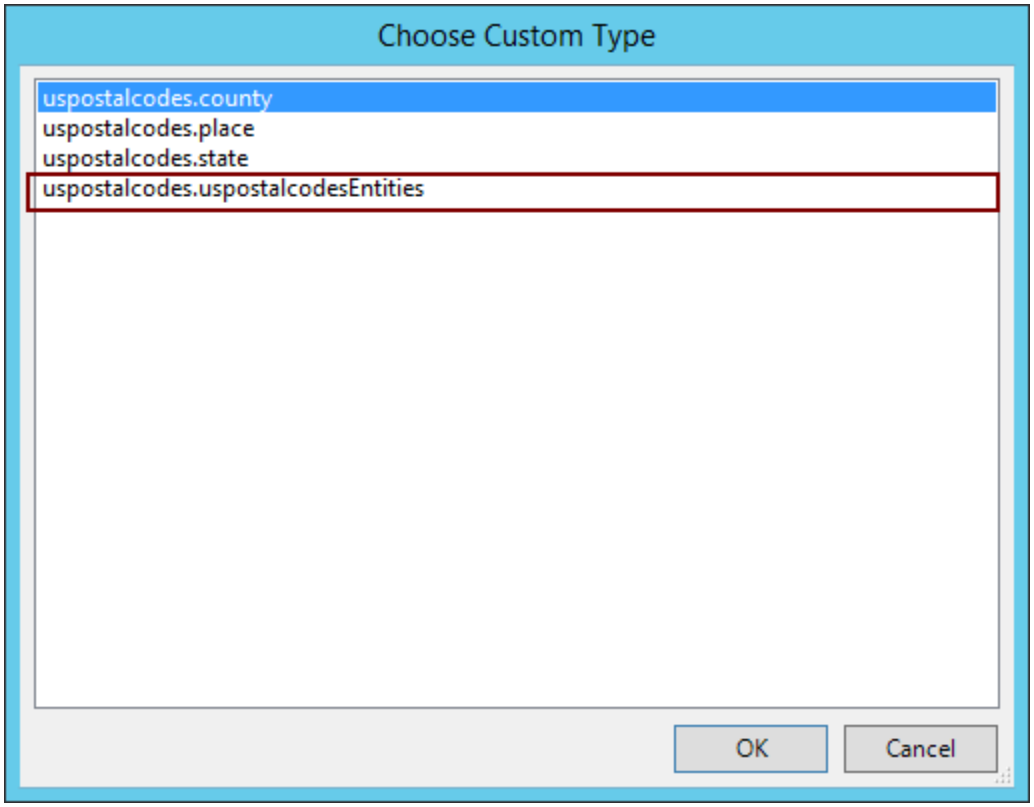


Figure 52: The Choose Custom Type Dialog

After you complete the previous steps, the connection dialog will look like this.

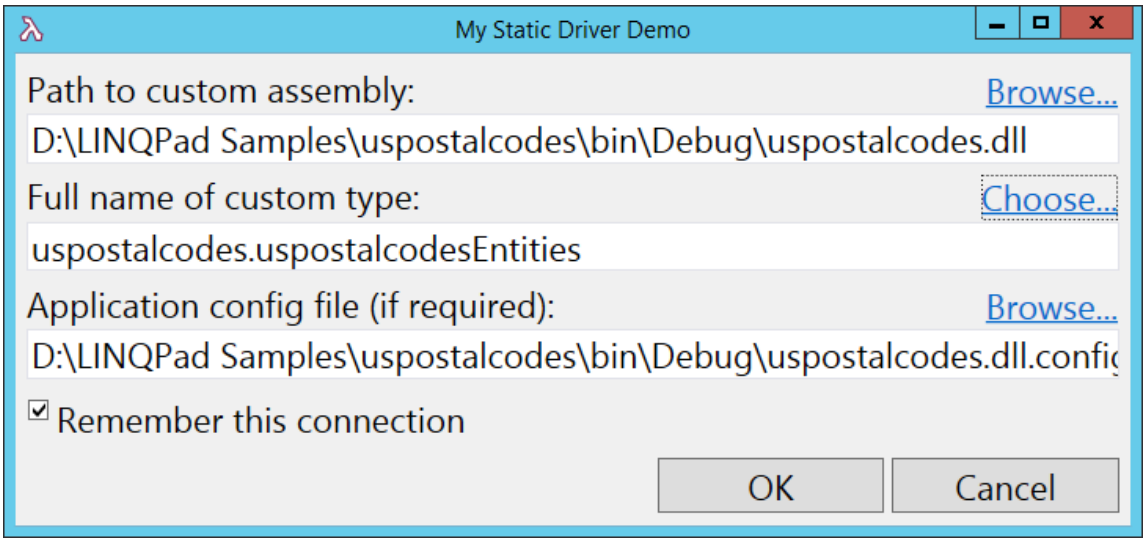


Figure 53: The Connection Dialog Filled In

Creating the connection and viewing the entities

After you click **OK**, LINQPad will create the connection and display the entities in the user interface.

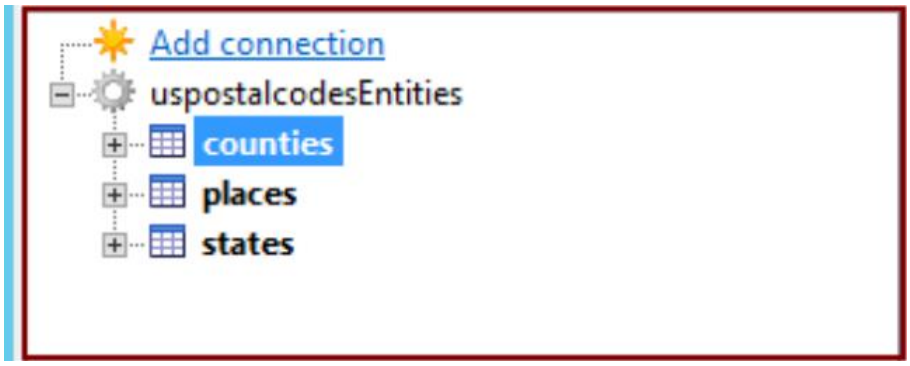


Figure 54: The uspostalcodesEntities Displayed Using Our Custom Driver

Displaying the custom driver description

To display the description of our custom driver, right-click the **uspostalcodesEntities** node. A context menu will be displayed, and the description for our driver will be shown at the top of this menu. The following figure shows an example.

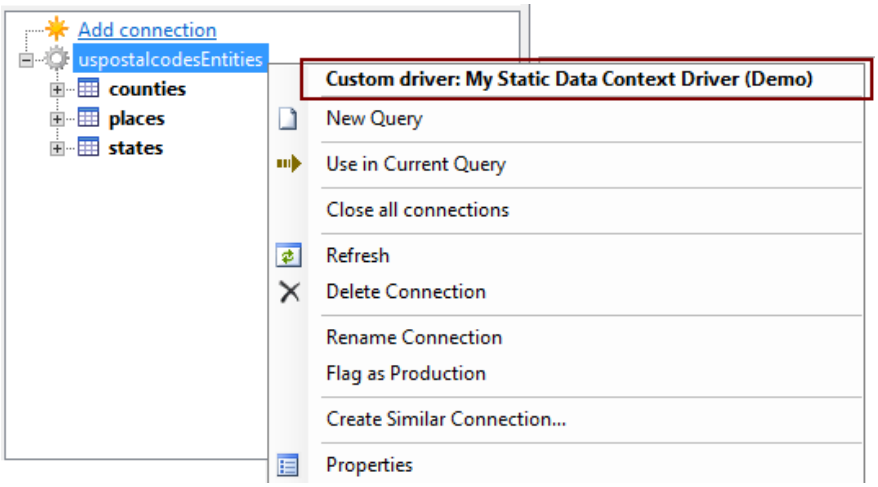


Figure 55: LINQPad Using Our Custom Driver

Executing a query

We will execute a query of one of the displayed entities to finish our testing process. In this case, we'll show the elements stored in the `states` entity using a data grid. To accomplish this, right-click the **states** node in the user interface, and then select the **View top 100 rows in grid** option.

www.dbooks.org

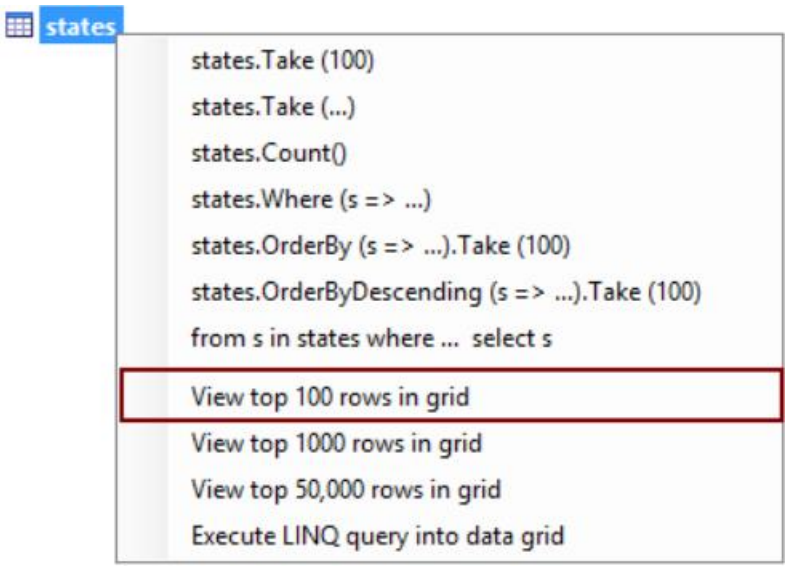


Figure 56: Executing a Query with The Driver

LINQPad will automatically generate the appropriate code for executing the query, and the results will be displayed in the output panel.

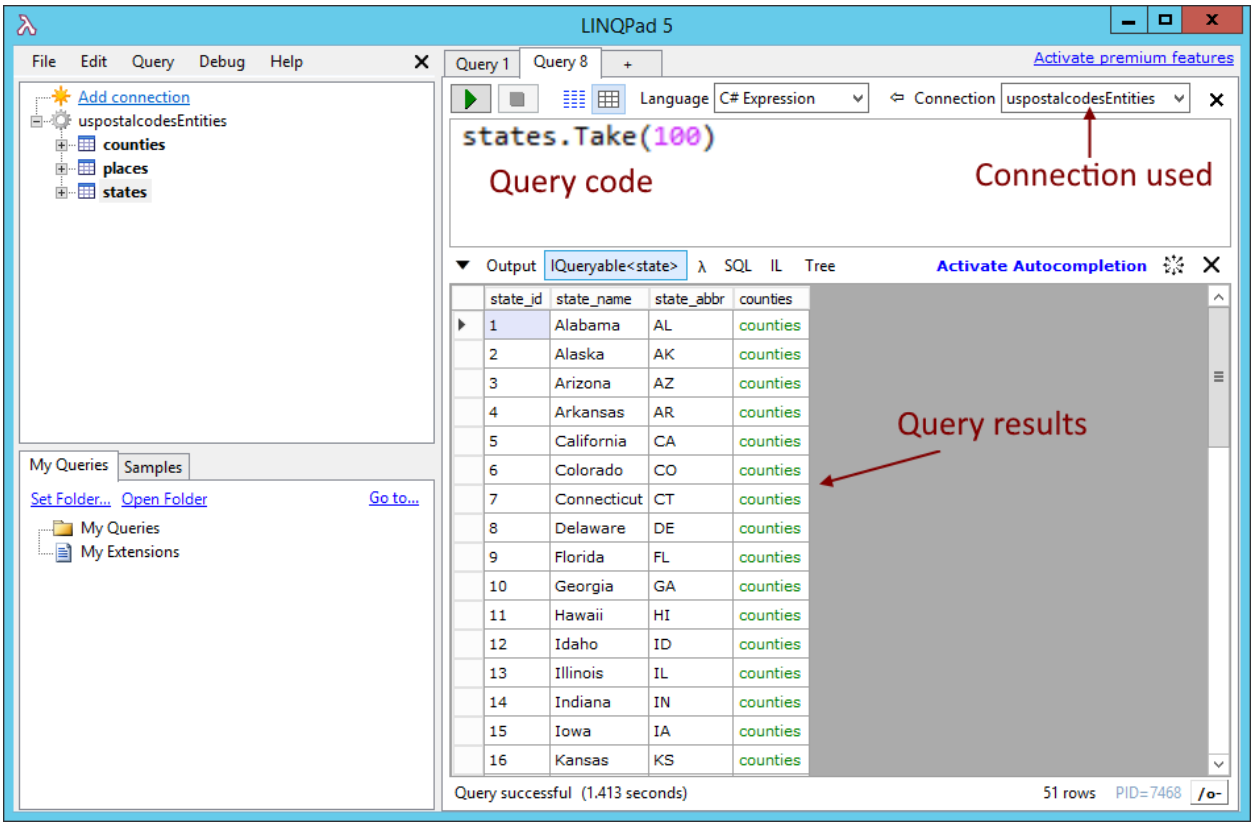


Figure 57: Displaying the Results of the Query

Debugging the driver

We can debug our custom driver by following these steps:

1. Start LINQPad.
2. Open the **MyDataContextDriver** solution in Visual Studio.
3. Once the solution is opened, go to **Debug** > **Attach to Process**. Then, locate LINQPad.exe in the dialog. After selecting **LINQPad.exe** from the list, click the **Attach** button.
4. Set the desired breakpoints in the solution.

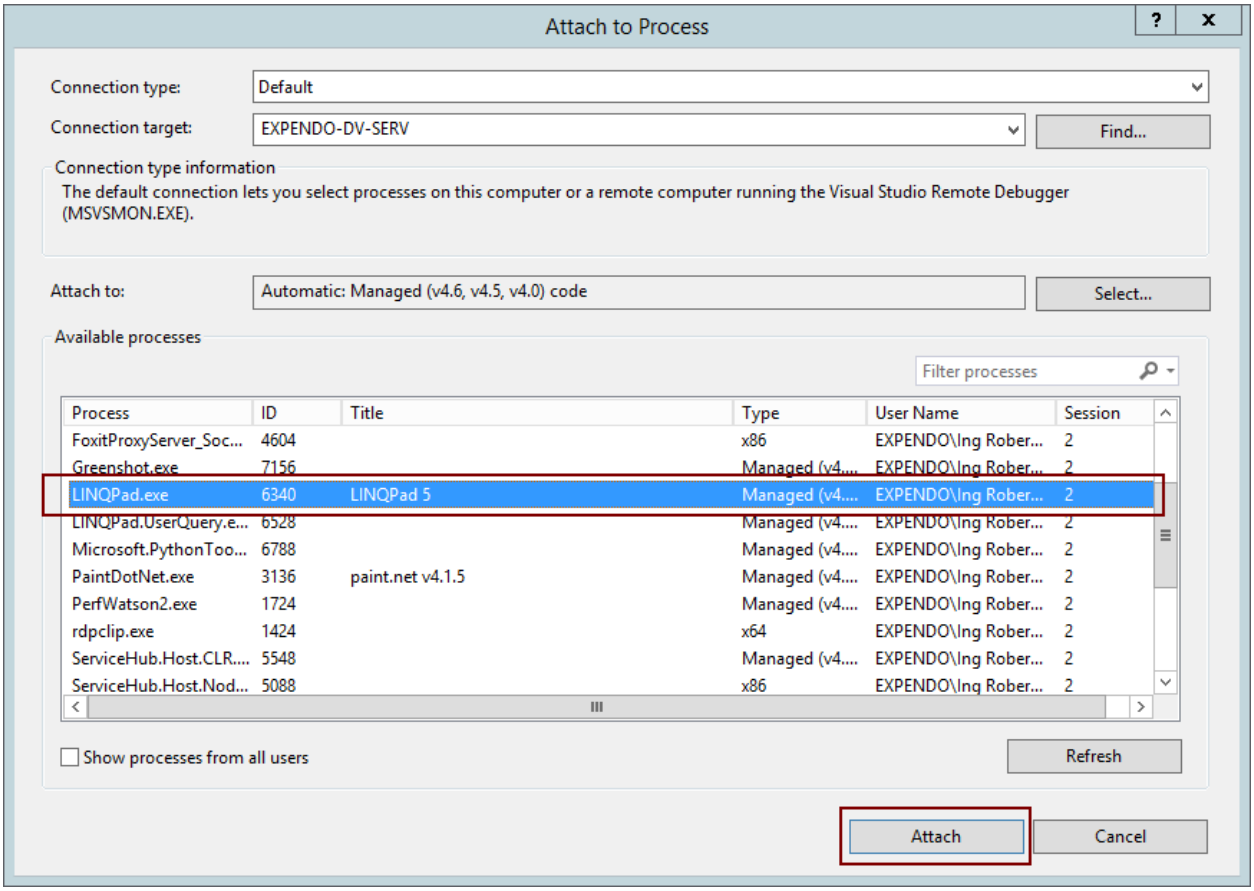


Figure 58: The Attach to Process Dialog

Exception logging

When a custom driver throws an exception, LINQPad writes the exception stack trace and details in a log file. This file resides in the %localappdata%\linqpad\logs\ folder, which is usually located at C:\Users\UserName\AppData\Local\LINQPad\logs.

www.dbooks.org

Code Listing 57: LINQPad log.txt File Example

```
5.36.03 2018-12-19T09:54:55.8803299-07:00 NotImplementedException: The
method or operation is not implemented.
   at
MyDataContextDriver.mystaticdriver.MyStaticDataContextDriver.GetConnectionD
escription(IConnectionInfo cxInfo)
   at
LINQPad.Extensibility.DataContext.DataContextDriver.GetConnectionDescriptio
n(IConnectionInfo cxInfo)
   at LINQPad.Repository.GetFriendlyName(FriendlyNameMode mode)
First chance data:
LINQPad.Extensibility.DataContext.DataContextDriver.GetConnectionDescriptio
n(cxInfo) offset=0xFFFFFFFF
   -LINQPad.Repository.GetFriendlyName(mode) offset=0x7C
   -LINQPad.Repository.GetFriendlyName() offset=0x1
   -LINQPad.UI.SchemaTreeInternal.RepositoryNode..ctor(r) offset=0x22
   -LINQPad.UI.SchemaTreeInternal.StaticSchemaNode..ctor(r) offset=0x7
   -
LINQPad.UI.SchemaTree.AddCx(repos,child,selectNode,expandNode,massPopulatio
n,updateAllNodeText) offset=0x35
   -LINQPad.UI.SchemaTree.AddCx(r,selectNode,expandNode) offset=0xD
   -LINQPad.UI.SchemaTree.AddCx() offset=0x2C
   -LINQPad.UI.SchemaTree.ProcessLeftMouseDown(node) offset=0x250
   -LINQPad.UI.SchemaTree.WndProc(m) offset=0x27
   -
LINQPad.Program.Run(queryToLoad,runQuery,activationCode,activateAll,deactiv
ate,noForward,noUpdate,caller) offset=0x27C
   -LINQPad.Program.Go(args) offset=0x763
   -LINQPad.Program.Start(args) offset=0xA3
   -LINQPad.ProgramStarter.Run(args) offset=0x12
   -LINQPad.Loader.Main(args) offset=0x2B4

5.36.03 2018-12-19T09:54:55.9595444-07:00 NotImplementedException: The
method or operation is not implemented.
   at
MyDataContextDriver.mystaticdriver.MyStaticDataContextDriver.GetConnectionD
escription(IConnectionInfo cxInfo)
   at
LINQPad.Extensibility.DataContext.DataContextDriver.GetConnectionDescriptio
n(IConnectionInfo cxInfo)
   at LINQPad.Repository.GetFriendlyName(FriendlyNameMode mode)
First chance data:
LINQPad.Extensibility.DataContext.DataContextDriver.GetConnectionDescriptio
n(cxInfo) offset=0xFFFFFFFF
   -LINQPad.Repository.GetFriendlyName(mode) offset=0x7C
   -LINQPad.UI.QueryControl.UpdateFocusedRepository() offset=0x3E
   -LINQPad.UI.QueryControl._schemaTree_AfterSelect(sender,e) offset=0x7
   -LINQPad.UI.SchemaTree.WndProc(m) offset=0x3C
```

```
   -LINQPad.UI.SchemaTree.WndProc(m) offset=0x3C
   -
LINQPad.UI.SchemaTree.AddCx(repos,child,selectNode,expandNode,massPopulatio
n,updateAllNodeText) offset=0xED
   -LINQPad.UI.SchemaTree.AddCx(r,selectNode,expandNode) offset=0xD
   -LINQPad.UI.SchemaTree.AddCx() offset=0x2C
   -LINQPad.UI.SchemaTree.ProcessLeftMouseDown(node) offset=0x250
   -LINQPad.UI.SchemaTree.WndProc(m) offset=0x27
   -
LINQPad.Program.Run(queryToLoad,runQuery,activationCode,activateAll,deactiv
ate,noForward,noUpdate,caller) offset=0x27C
   -LINQPad.Program.Go(args) offset=0x763
   -LINQPad.Program.Start(args) offset=0xA3
   -LINQPad.ProgramStarter.Run(args) offset=0x12
   -LINQPad.Loader.Main(args) offset=0x2B4

5.36.03 2018-12-19T09:54:55.9675620-07:00 NotImplementedException: The
method or operation is not implemented.
   at
MyDataContextDriver.mystaticdriver.MyStaticDataContextDriver.GetConnectionD
escription(IConnectionInfo cxInfo)
   at
LINQPad.Extensibility.DataContext.DataContextDriver.GetConnectionDescriptio
n(IConnectionInfo cxInfo)
   at LINQPad.Repository.GetFriendlyName(FriendlyNameMode mode)
First chance data:
LINQPad.Extensibility.DataContext.DataContextDriver.GetConnectionDescriptio
n(cxInfo) offset=0xFFFFFFFF
   -LINQPad.Repository.GetFriendlyName(mode) offset=0x7C
   -LINQPad.UI.QueryControl.UpdateFocusedRepository() offset=0x3E
   -LINQPad.UI.QueryControl._schemaTree_AfterSelect(sender,e) offset=0x7
   -LINQPad.UI.SchemaTree.WndProc(m) offset=0x3C
   -LINQPad.UI.SchemaTree.WndProc(m) offset=0x3C
   -
LINQPad.UI.SchemaTree.AddCx(repos,child,selectNode,expandNode,massPopulatio
n,updateAllNodeText) offset=0xED
   -LINQPad.UI.SchemaTree.AddCx(r,selectNode,expandNode) offset=0xD
   -LINQPad.UI.SchemaTree.AddCx() offset=0x2C
   -LINQPad.UI.SchemaTree.ProcessLeftMouseDown(node) offset=0x250
   -LINQPad.UI.SchemaTree.WndProc(m) offset=0x27
   -
LINQPad.Program.Run(queryToLoad,runQuery,activationCode,activateAll,deactiv
ate,noForward,noUpdate,caller) offset=0x27C
   -LINQPad.Program.Go(args) offset=0x763
   -LINQPad.Program.Start(args) offset=0xA3
   -LINQPad.ProgramStarter.Run(args) offset=0x12
   -LINQPad.Loader.Main(args) offset=0x2B4
```


www.dbooks.org

```
5.36.03 2018-12-19T09:54:56.0106750-07:00 NotImplementedException: The
method or operation is not implemented.
   at
MyDataContextDriver.mystaticdriver.MyStaticDataContextDriver.GetConnectionD
escription(IConnectionInfo cxInfo)
   at
LINQPad.Extensibility.DataContext.DataContextDriver.GetConnectionDescriptio
n(IConnectionInfo cxInfo)
   at LINQPad.Repository.GetFriendlyName(FriendlyNameMode mode)
First chance data:
LINQPad.Extensibility.DataContext.DataContextDriver.GetConnectionDescriptio
n(cxInfo) offset=0xFFFFFFFF
   -LINQPad.Repository.GetFriendlyName(mode) offset=0x7C
   -LINQPad.UI.QueryControl.UpdateFocusedRepository() offset=0x3E
   -LINQPad.UI.QueryControl._schemaTree_AfterSelect(sender,e) offset=0x7
   -LINQPad.UI.SchemaTree.WndProc(m) offset=0x3C
   -LINQPad.UI.SchemaTree.WndProc(m) offset=0x3C
   -
LINQPad.UI.SchemaTree.AddCx(repos,child,selectNode,expandNode,massPopulatio
n,updateAllNodeText) offset=0xED
   -LINQPad.UI.SchemaTree.AddCx(r,selectNode,expandNode) offset=0xD
   -LINQPad.UI.SchemaTree.AddCx() offset=0x2C
   -LINQPad.UI.SchemaTree.ProcessLeftMouseDown(node) offset=0x250
   -LINQPad.UI.SchemaTree.WndProc(m) offset=0x27
   -
LINQPad.Program.Run(queryToLoad,runQuery,activationCode,activateAll,deactiv
ate,noForward,noUpdate,caller) offset=0x27C
   -LINQPad.Program.Go(args) offset=0x763
   -LINQPad.Program.Start(args) offset=0xA3
   -LINQPad.ProgramStarter.Run(args) offset=0x12
   -LINQPad.Loader.Main(args) offset=0x2B4

5.36.03 2018-12-19T09:54:56.0157221-07:00 NotImplementedException: The
method or operation is not implemented.
   at
MyDataContextDriver.mystaticdriver.MyStaticDataContextDriver.GetConnectionD
escription(IConnectionInfo cxInfo)
   at
LINQPad.Extensibility.DataContext.DataContextDriver.GetConnectionDescriptio
n(IConnectionInfo cxInfo)
   at LINQPad.Repository.GetFriendlyName(FriendlyNameMode mode)
First chance data:
LINQPad.Extensibility.DataContext.DataContextDriver.GetConnectionDescriptio
n(cxInfo) offset=0xFFFFFFFF
   -LINQPad.Repository.GetFriendlyName(mode) offset=0x7C
   -LINQPad.UI.QueryControl.UpdateFocusedRepository() offset=0x3E
   -LINQPad.UI.QueryControl._schemaTree_AfterSelect(sender,e) offset=0x7
   -LINQPad.UI.SchemaTree.WndProc(m) offset=0x3C
   -LINQPad.UI.SchemaTree.WndProc(m) offset=0x3C
```

```
   -
LINQPad.UI.SchemaTree.AddCx(repos,child,selectNode,expandNode,massPopulatio
n,updateAllNodeText) offset=0xED
   -LINQPad.UI.SchemaTree.AddCx(r,selectNode,expandNode) offset=0xD
   -LINQPad.UI.SchemaTree.AddCx() offset=0x2C
   -LINQPad.UI.SchemaTree.ProcessLeftMouseDown(node) offset=0x250
   -LINQPad.UI.SchemaTree.WndProc(m) offset=0x27
   -
LINQPad.Program.Run(queryToLoad,runQuery,activationCode,activateAll,deactiv
ate,noForward,noUpdate,caller) offset=0x27C
   -LINQPad.Program.Go(args) offset=0x763
   -LINQPad.Program.Start(args) offset=0xA3
   -LINQPad.ProgramStarter.Run(args) offset=0x12
   -LINQPad.Loader.Main(args) offset=0x2B4

5.36.03 2018-12-19T09:54:56.0307559-07:00 NotImplementedException: The
method or operation is not implemented.
   at
MyDataContextDriver.mystaticdriver.MyStaticDataContextDriver.GetConnectionD
escription(IConnectionInfo cxInfo)
   at
LINQPad.Extensibility.DataContext.DataContextDriver.GetConnectionDescriptio
n(IConnectionInfo cxInfo)
   at LINQPad.Repository.GetFriendlyName(FriendlyNameMode mode)
First chance data:
LINQPad.Extensibility.DataContext.DataContextDriver.GetConnectionDescriptio
n(cxInfo) offset=0xFFFFFFFF
   -LINQPad.Repository.GetFriendlyName(mode) offset=0x7C
   -LINQPad.Repository.GetFriendlyName() offset=0x1
   -LINQPad.UI.SchemaTreeInternal.RepositoryNode.UpdateText() offset=0x2
   -LINQPad.UI.SchemaTreeInternal.StaticSchemaNode.UpdateText() offset=0x7
   -LINQPad.UI.SchemaTree.UpdateAllNodeText() offset=0x31
   -
LINQPad.UI.SchemaTree.AddCx(repos,child,selectNode,expandNode,massPopulatio
n,updateAllNodeText) offset=0xFC
   -LINQPad.UI.SchemaTree.AddCx(r,selectNode,expandNode) offset=0xD
   -LINQPad.UI.SchemaTree.AddCx() offset=0x2C
   -LINQPad.UI.SchemaTree.ProcessLeftMouseDown(node) offset=0x250
   -LINQPad.UI.SchemaTree.WndProc(m) offset=0x27
   -
LINQPad.Program.Run(queryToLoad,runQuery,activationCode,activateAll,deactiv
ate,noForward,noUpdate,caller) offset=0x27C
   -LINQPad.Program.Go(args) offset=0x763
   -LINQPad.Program.Start(args) offset=0xA3
   -LINQPad.ProgramStarter.Run(args) offset=0x12
   -LINQPad.Loader.Main(args) offset=0x2B4
```


www.dbooks.org

Credits

Part of the code of this chapter is based on information and examples provided by Joseph Albahari. This information and the code associated with it are public and can be downloaded from https://www.linqpad.net/DataContextDrivers.docx and https://www.linqpad.net/DataContextDriverDemo.zip.

Chapter summary

A data context driver is a mechanism that allows programmers to extend LINQPad in order to support several data sources. LINQPad can query any data source without a custom data context driver, but in this case, the user must manually reference libraries, import custom namespaces, and formulate all queries.

A data context driver is employed every time a user adds a connection to LINQPad. Those connections appear on the Connection's tree view area of the user interface. To add a connection, click the **Add Connection** hyperlink located at the top of the Connection's tree view area.

The basic steps for building a driver are:

1. Choose between writing a dynamic or static driver.
2. Build a class project in Visual Studio that references LINQPad.
3. Create classes derived from **`DynamicDataContextDriver`** or **`StaticDataContextDriver`**.
4. Implement a few abstract methods (and some virtual methods, optionally).
5. Zip the files generated by the class project (and all their dependencies) and change the extension from .**zip** to .**lpx**.
6. Save the file in any place in the computer, so it can be found by using the **Browse** button.

There are no special terms or conditions for writing a data context driver, unless you plan to submit the driver to LINQPad's Driver Gallery, which is beyond the scope of this book.

The following are basic concepts about data context drivers:

- **Connection**: Relates to what you enter when you click **Add Connection**. This is broader than the classic concept of a database connection because a LINQPad connection can refer to other types of data sources, such as a web service URI.
- **Typed data contexts**: A typed data context is a class with properties, fields, and methods that can be queried by the user. A typical example is a typed LINQ to SQL **`DataContext`** or a typed **`ObjectContext`** in Entity Framework.
- **Dynamic driver**: A driver that builds the data context on the fly, either by generating code and compiling it after, or by using **`Reflection.Emit`**.
- **Static driver**: A driver that requires a data context supplied by the user. The connection dialog will prompt for a path where a custom assembly containing the typed data context is located, and for the name of the type.

For the purposes of this book, writing a data context driver requires you to set up a Visual Studio project with the following properties:

- Project type: Class library (.NET Framework)
- Framework: .NET Framework 4.6.1
- Name: MyDataContextDriver
- Location: D:\LINQPad Samples
- Create directory for solution: Unchecked

Every time LINQPad tries to install a context data driver, it looks for a file named header.xml. If this file is not present in the deployment package, the installation process will fail. So, the first thing to do is to add the corresponding **header.xml** file to the project. This file can be created in a text editor (like Notepad).

There are two base classes for context drivers: **`DynamicDataContextDriver`** and **`StaticDataContextDriver`**. The name of each type tells us the kind of driver we can build with each one of them. Both types are derived from the base class **`DataContextDriver`**, which is defined in the **`LINQPad.Extensibility.DataContext`** namespace. This class defines some abstract members that will be implemented during project creation. The files named **MyDynamicDataContextDriver.cs** and **MyStaticDataContextDriver.cs** will define the **`MyDynamicDataContextDriver`** and **`MyStaticDataContextDriver`** classes, to implement both kind of drivers. Since both classes are derived from the **`DataContextDriver`** class, Visual Studio automatically asks us to implement the abstract members defined in it. We simply let Visual Studio generate the implementation code automatically.

We focused on building the static data context driver defined in **MyStaticDataContextDriver.cs** by implementing the **`GetSchema`** abstract method. This method is employed to return a hierarchy of objects that will be displayed in the Schema Explorer (which is in the Connection's tree view area of the user interface). Then, we implemented the **`ShowConnectionDialog`** method, which displays a modal WPF dialog to prompt the user for connection information.

To deploy the data context driver, we complete the following steps:

1. Build the assembly by selecting **Build** > **Build Solution** or **Rebuild Solution** from the Visual Studio menu bar.
2. Zip the MyDataContextDriver.dll and header.xml files into a single file named mycontextdatadriver.zip.
3. Rename mycontextdatadriver.zip to mycontextdatadriver.lpx.
4. Copy **mycontextdatadriver.lpx** to any desired folder in the file system.

To install the custom driver in LINQPad, it's mandatory that you try adding a new connection by clicking the **Add Connection** hyperlink in the user interface. Then, LINQPad will show the Choose Data Context dialog. Next, click the **View More Drivers** button. This will bring up the Choose a Driver dialog. Once the dialog is displayed, click **Browse**, and the Browse LINQPad Data Context Driver dialog will be shown. After that, locate the file for our custom driver and double-click its name. When installation is finished, LINQPad will show the Installation Succeeded dialog.

Finally, we tested our custom driver using the Entity Data Model assembly created in Chapter 2.

www.dbooks.org

Conclusion

Since LINQPad was added to our toolbox at my company, the development process for any of our products has improved tremendously. LINQPad has helped us in several stages of the development lifecycle, such as testing and product prototyping. LINQPad has also become a valuable tool for LINQ training, allowing our developers to take advantage of LINQ's features in order to create smaller and more efficient pieces of code.

Before LINQPad, almost 20 percent of our main repository space was consumed by single-use Visual Studio projects, most of which were created for testing purposes. Now all those projects have been transformed into LINQPad queries, which can be used more than once. Therefore, repository management has become easier, and the space consumed in our storage devices has greatly decreased.

Another useful thing we've taken advantage of is LINQPad's ability to import data sources and interact with them easily. Since the program allows you to manage several connections with the power of a click, many data models can be taken from the diagrams tool to the real world, and they can be validated in less time than is needed with traditional tools. Our database designers have the power to retrieve data using LINQ to SQL queries, and turn these queries into SQL views or stored procedures by simply selecting the SQL commands automatically generated by LINQPad.

LINQPad's extensibility has helped us get the most from this tool. The ability to create custom methods, which can be taken as if they were included in the shipping product, gives us the ability to customize LINQPad according to our particular needs, increasing our skills with the product and with .NET programming in the process.

LINQPad is quite an awesome product, according to the general opinion among our development team—myself included. That's the reason it's so difficult for us to remove it from our company's toolbox.