Entwicklung eines Datenkontexttreibers für LINQPad

Kurs auf SharePoint

LINQ-Abfragen sind nicht nur mit Visual Studio möglich. Auch Tools stehen dafür zur Verfügung, beispielsweise das Werkzeug LINQPad. Seine Stärke und Flexibilität beweist es, indem es eigene Treiber für den Zugriff auf Datenquellen ermöglicht und damit für Spezialfälle offen ist. So einen Treiber zu schreiben ist gar nicht schwer.

Auf einen Blick

Marcus Peters kennt .NET seit der ersten Stunde und ist bekennender SharePoint-Fan. Er verantwortet im „Competence Center Portal Applications" bei Adesso aus Dortmund die Umsetzung von Geschäftsanwendungen auf der SharePoint-Plattform. Sie erreichen ihn unter marcus.peters@adesso.de.

Inhalt

- Voraussetzungen für LINQPad und SharePoint.
- Die DataContext-Klasse von SharePoint als Grundlage eines LINQPad-Treibers.
- Mit SPMetal Entitäten für den Zugriff auf SharePoint erzeugen.

dnpCode
A1103LINQPad

LINQPad [1] ist ein spannendes und kostenloses Werkzeug für Entwickler, die einfach „mal eben" Daten via LINQ abfragen wollen, ohne dazu gleich Visual Studio starten zu müssen. Joseph Albahari, der Autor von LINQPad – er hat auch einige Bücher geschrieben –, beschreibt dies auf der LINQPad-Homepage plakativ mit: „Kiss goodbye to SQL Management Studio!" Die Benutzung des Tools ist in der Tat so einfach, dass auch hier das typische explorative Verhalten gilt: Einfach runterladen, starten, Datenquelle angeben und loslegen. Abbildung 1 zeigt einen Screenshot des Werkzeugs, während es gerade auf die Northwind-Datenbank zugreift.

LINQPad kann nicht nur mit Datenbanken arbeiten, sondern im Prinzip mit allen Datenquellen, die sich mit LINQ verarbeiten lassen. „Out of the box" unterstützt Version 4 folgende Quelltypen:

- LINQ to SQL,
- WCF-Datendienste (OData),
- „Dallas"-Dienste von Microsofts Daten-Metaplattform,
- das Entity Framework.

Zusätzlich ist LINQPad in der Lage, mit Code der .NET-Sprachen C#, Visual Basic und F# umzugehen, und dient somit als Mini-Entwicklungsumgebung. Damit lassen sich Codeschnipsel schnell mal nebenbei ausprobieren, wie zum Beispiel dieses:

```
void Main()
{
  DateTime tomorrow = DateTime.Now.AddDays(1);
  Console.WriteLine(tomorrow.ToString(
  "dd-MMM-yyyy"));
}
```

Zudem ist es möglich, LINQPad für weitere Datenquellen zu erweitern. So sind zum Beispiel Komponenten für den Umgang mit SQLite- und MySQL-Datenbanken verfügbar, die sich über LINQPad herunterladen und installieren lassen.

Doch darauf beschränkt sich das Tool nicht. Albahari hat ihm eine Schnittstelle mitgegeben, sodass es möglich ist, selbst Erweiterungen für im Prinzip beliebige Datenquellen zu implementieren. Wie Sie so eine Erweiterung selbst programmieren können, zeigt dieser Artikel am Beispiel eines Treibers für den Zugriff auf SharePoint – nennen wir ihn einfach „LINQ to SharePoint". Und für alle, die nicht ohne das Visual-Studio-IntelliSense und seine Codevervollständigung leben können, gibt es die Möglichkeit, diese Funktionen gegen Gebühr für LINQPad zu lizensieren.

Wie LINQPad arbeitet

Jede Abfrage in LINQPad läuft in ihrer eigenen Anwendungsdomäne; das isoliert sie von anderen Abfragen und der Oberfläche des Programms. Abbildung 2 illustriert die Architektur des LINQPad-Tools [2].

Abfrage-Statements des Nutzers kompiliert LINQPad über die entsprechenden Compiler-Klassen, wie zum Beispiel *CSharpCodeProvider*, und führt sie anschließend in einem Datenkontext in Form einer typisierten *DataContext*-Instanz aus. Dieser Datenkontext aus dem Namensraum *Microsoft.SharePoint.Linq* fungiert hier, ähnlich wie bei LINQ to SQL, als Basiskomponente, über welche die Kommunikation mit der Datenquelle und der Zugriff auf sie erfolgt.

LINQPad erweitern

Um die Funktionen von LINQPad zu erweitern, gibt es im Wesentlichen zwei Möglichkeiten. Die erste gestattet es, für jede Abfrage weitere Assemblies und Namensräume einzubinden; die zweite Variante besteht darin, einen *DataContext*-Treiber zu entwickeln, der den Zugriff auf weitere Datenquellen erlaubt und weitere Assemblies automatisch einbindet. So ein Treiber ermöglicht es außerdem noch, dass das Schema der entsprechenden Datenquelle im Schema-Explorer von LINQPad angezeigt wird, wie es Abbildung 3 illustriert.

Für das Entwickeln eines derartigen Treibers sind folgende Schritte nötig:

- Entscheiden, ob es ein dynamischer oder ein statischer Kontexttreiber sein soll; dazu später mehr.
- Erstellen eines Klassenbibliothek-Projekts in Visual Studio und eines Verweises darin auf *LinqPad.exe*.
- Ableiten der Treiberklasse von den LINQPad-Klassen *DynamicDataContextDriver* oder *StaticDataContextDriver*.
- Implementieren einer Handvoll abstrakter Methoden.

- Gegebenenfalls müssen einige virtuelle Methoden für spezielles Verhalten überschrieben werden.
- Verpacken der fertigen Klassenbibliothek und eventueller Abhängigkeiten sowie einer *Header.xml*-Datei mit Support-Informationen in einem Zip-Archiv.
- Umbenennen des Archivs von **.zip* nach **.lpx*.

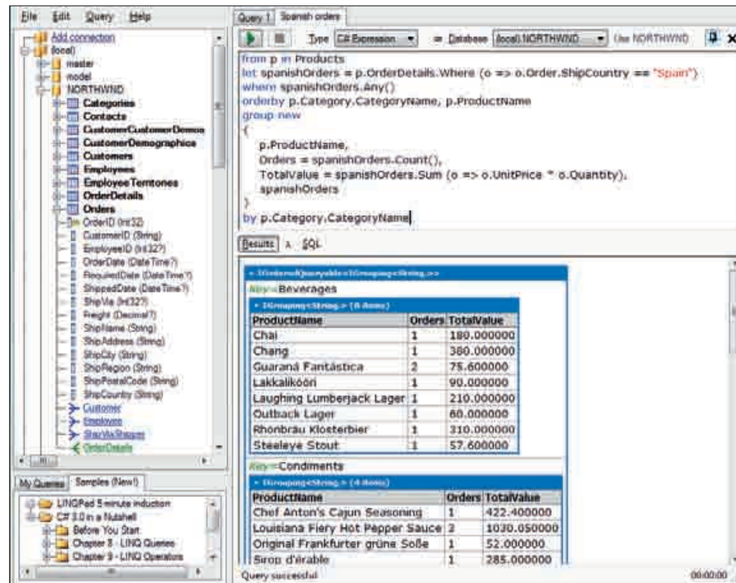
Zur Unterstützung beim Entwickeln hat LINQPad-Autor Albahari im Internet eine Dokumentation sowie zwei Beispieltreiber als Visual-Studio-Lösungsmappen zum Download bereitgestellt [3].

Nach der Installation eines Treibers ist dieser im LINQPad-Datenverzeichnis zu finden unter:

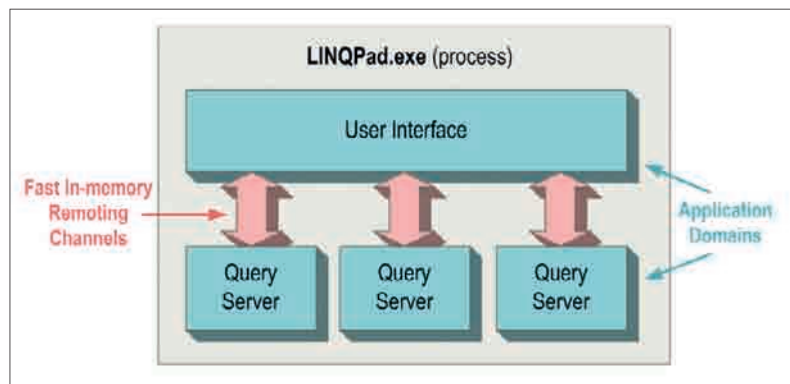
```
Path.Combine (
  Environment.GetFolderPath(Environment.
  SpecialFolder.CommonApplicationData),
  @"LINQPad\Drivers\DataContext\3.5\"
  )\TREIBER_NAME(PublicKeyToken)
```

Das Demoprojekt zu diesem Artikel enthält ein Skript, das sich als Post-Build-Aktion einsetzen lässt und die neu kompilierten Dateien direkt in den entsprechenden Ordner kopiert.

Dies erspart lästiges Hin- und Herkopieren von Hand.



[Abb. 1] LINQPad zeigt Daten aus der Northwind-Datenbank.



[Abb. 2] Die Architektur von LINQPad isoliert Oberfläche und Abfragen voneinander.

[Abb. 3] Der Schema-Explorer von LINQPad erhält die Daten über die SharePoint-Entitäten von einem Datenkontexttreiber.

Voraussetzungen für LINQPad und SharePoint

Bevor es nun mit der Entwicklung eines SharePoint-Treibers für LINQPad losgeht, noch zwei kurze Anmerkungen:

- Da SharePoint .NET 4.0 nicht unterstützt, ist nur die Version für .NET 3.5 von LINQPad verwendbar. Das vorliegende Beispiel arbeitet mit der Version 2.26.2.
- LINQ to SharePoint wird auf SharePoint Server ausgeführt und erfordert somit für die Abfragen ein „echtes" SharePoint im Hintergrund.

LINQ to SharePoint – Wie war das mit dem Datenkontext?

Ähnlich wie LINQ to SQL verwendet LINQ to SharePoint einen *DataContext* als Haupteinstiegspunkt für den Provider. Im Falle von SharePoint bietet dieser den Zugriff auf SharePoint-Listen und -Dokumentenbibliotheken und überwacht auch etwaige Änderungen. Mithilfe der Funktion *GetList(T)* gibt *DataContext* eine Liste der gewünschten SharePoint-Entitäten zurück:

```
DataContext siteContext =
  new DataContext("http://myspsite");
```

```
EntityList<Page> pages =
  siteContext.GetList<Page>("Pages");
```

Das Mapping auf SharePoint-Entitäten kann entweder händisch oder – in vielen Fällen einfacher und umfassender – durch den Einsatz des Tools SPMetal [4] erfolgen, das diesen Vorgang automatisiert erledigt.

Allerdings gibt es Fälle, in denen der von diesem Utility generierte Code nicht ausreicht und erweitert werden muss. Das Microsoft Developer Network bietet gute Erklärungen für die manuelle Erstellung der Entitäten sowie die Erweiterung des OR-Mappings von SharePoint-Entitäten unter Berücksichtigung von Change-Tracking und Concurrency, siehe [5], [6], [7]. Im vorliegenden Fall wurde SPMetal als „On the fly"-Generator eingesetzt, um den erstellten LINQPad-Treiber mit Entitäten zu versorgen. Da es sich hier aber lediglich um einen Überblick handelt, sei noch der Hinweis erlaubt, dass der SharePoint-LINQ-Provider nicht alles unterstützt, was mit LINQ möglich ist; entsprechende Details hierzu finden sich unter [8].

Dynamisch oder statisch – das ist hier die Frage

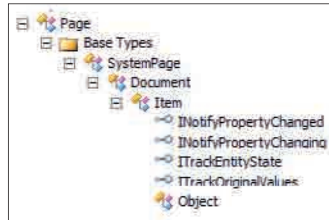
Nun wird es aber höchste Zeit, den Datenkontexttreiber für LINQPad und SharePoint 2010 in den oben dargestellten Schritten zu

entwickeln. Zunächst gilt es also zu entscheiden, ob ein dynamischer oder statischer Treiber entwickelt werden soll.

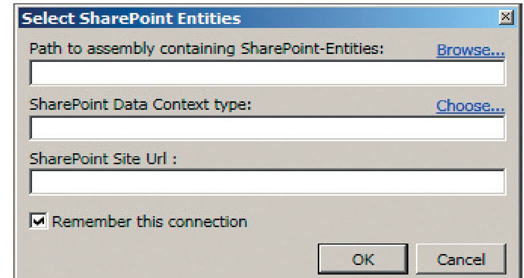
Ein dynamischer Treiber tut im Prinzip genau das, was ein Nutzer erwarten würde: Er bekommt Verbindungsinformationen vom Nutzer, liest anschließend das Schema der Datenquelle aus und erstellt den typisierten Datenkontext daraus dynamisch in einer von LINQPad generierten Assembly. Im Gegensatz dazu erfordert ein statischer Treiber den typisierten Datenkontext in einer vom Benutzer anzugebenden Assembly.

Die Entscheidung für die Entwicklung eines statischen Treibers fällt aus folgenden Gründen: Wie erwähnt, kann die Generierung von SharePoint-Entitäten und des LINQ-Datenkontexts für SharePoint manuell oder durch SPMetal erfolgen. Eine automatische dynamische Generierung ist zwar denkbar, jedoch nicht immer sinnvoll. Denn sowohl manuell erstellte SharePoint-Entitäten als auch Erweiterungen an dem von SPMetal erstellten Code sind doch eher statischer Natur.

Die nächsten Schritte wie das Anlegen eines Klassenbibliothek-Projekts, das Referenzieren von *LinqPad.exe* sowie das Ableiten der Klasse *StaticDataContextDriver* bedürfen keiner näheren Beschreibung; somit kann es gleich mit dem nächsten Punkt auf der Liste weitergehen.



[Abb. 4] Die Klassenhierarchie der von SPMetal erzeugten SharePoint-Entitäten.



[Abb. 5] Der LINQPad-Verbindungsdialog für SharePoint-Sites.

Implementieren einer „Handvoll abstrakter Methoden"

„Klar, gerne, aber welche – und was tun die denn?" Das dürfte wohl die erste Frage sein, die sich jeder Entwickler bei der ersten Begegnung mit der LINQPad-Dokumentation sofort stellt. Wer darin etwas ähnlich Systematisches wie Aktivitätsdiagramme oder dergleichen sucht, wird nicht fündig werden. Als Hilfsmittel dienen neben der Dokumentation die Beispielprojekte mit einem statischen und einem dynamischen Datenkontexttreiber, der Visual-Studio-Debugger sowie ein guter alter Bekannter, der .NET Reflector [9].

Beim Laden eines Kontexttreibers geschieht im Wesentlichen Folgendes:

- Abfragen von Verbindungsinformationen in einem entsprechenden Dialog.
- Extraktion der Entitäten für die Darstellung im Schema-Explorer.

Dazu müssen mindestens die beiden folgenden Methoden implementiert werden:

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

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

ShowConnectionDialog() ruft ein Eingabefenster für den Anwender auf, in dem dieser die Angaben zur Verbindung festlegen kann. Der Parameter *isNewConnection* definiert, ob der Nutzer eine neue Verbindung anlegt oder eine bestehende verändert. Beendet der Anwender den Dialog mit *Ok* – bestätigt er also seine Eingaben –, soll die Methode *true* zurückgeben; andernfalls ist ihr Ergebnis *false* und die Änderungen in *IConnectionInfo* werden zurückgenommen. *GetSchema()* gibt eine hierarchische Liste von Objekten zurück, die im Schema-Explorer angezeigt werden.

Die LINQPad-Dokumentation beschreibt, dass 90 Prozent der Implementierungen für die Methoden von LINQPad-Treibern in den Beispielprojekten enthalten sind.

Dies hat der Selbstversuch des Autors durchaus bestätigt: Der Code des statischen Demotreibers für diese Methoden ließ sich daraus fast vollständig für das hier gezeigte Beispiel übernehmen. Lediglich die folgenden Zeilen waren zu ändern, damit nur SharePoint-Entitäten im Schema-Explorer angezeigt werden:

```
from prop in customType.GetProperties ()
  where prop.PropertyType != typeof (string)
```

Einen typisierten SharePoint-Datenkontext erstellen

Bevor der Verbindungsdialog angepasst wird, soll zunächst die Schema-Assembly vorgestellt werden, die den SharePoint-Datenkontext und die entsprechenden SharePoint-Entitäten enthält.

Hierzu ist wiederum ein Klassenbibliothek-Projekt zu erstellen und die SharePoint-Bibliotheken *Microsoft.SharePoint* und *Microsoft.SharePoint.Linq* sind zu referenzieren. Wie schon angedeutet, kommt hierbei SPMetal zum Einsatz. SPMetal ist derart konfigurierbar, dass unter anderem Namensräume, Modifizierer sowie die Auswahl der zu erstellenden Entitätsklassen parametrisiert werden können.

Eine Konfigurationsdatei für die Entität *Page* zeigt das folgende Listing:

```
<?xml version="1.0" encoding="utf-8"?>
<Web AccessModifier="Public"
     xmlns="http://schemas.microsoft.com/
         SharePoint/2009/spmetal">
    <List Name="Pages">
        <ContentType Name="Pages"
            Class="GeneratedClassName" />
```

Listing 1

SPMetal aktualisiert C#-Code im Projekt via Batch.

```
ECHO This is taken from MSDN http://msdn.microsoft.com/en-us/library/ee538587.aspx

Echo Off
SET SPLANGEXT=cs
SET SPMETALGENDIR=.\SPMetalGeneratedCode

Echo Backing up previous version of generated code ...
IF NOT EXIST %SPMETALGENDIR% MkDir %SPMETALGENDIR%
IF EXIST SiteEntities.%SPLANGEXT% xcopy /Y/V SiteName.%SPLANGEXT% %SPMETALGENDIR%

Echo Generating code ...
SPMetal /web:http://win2k8 /code:SiteEntities.%SPLANGEXT% /parameters:SPMetal.xml
  /namespace:MyNamespace
```

```
    </List>
    <ExcludeOtherLists></ExcludeOtherLists>
</Web>
```

Hierbei entspricht der generierte Code der Vererbungshierarchie der beteiligten SharePoint-Inhaltstypen (Abbildung 4).

Für den vorliegenden Artikel enthält das Projekt neben der Referenz nur eine C#-Datei, die durch ein Pre-Build-Skript wie in Listing 1 via SPMetal aktualisiert wird. Der Aufruf dieser Batch-Datei erfolgt über diese Kommandozeilenbefehle:

```
cd $(ProjectDir)
runspmetal.bat
```

Anpassen des Verbindungsdialogs

Der von SPMetal erzeugte Code enthält als Einstiegspunkt den schon erwähnten Datenkontext von LINQ to SharePoint. Dieser erwartet im Konstruktor einen URL für den Zugriff auf die Entitäten der gewünschten Site. Bei der Instanzierung durch LINQPad muss dieser URL ebenfalls an den Kontext übergeben werden.

Hierzu lassen sich wieder weite Teile des Beispielprojekts verwenden. Abbildung 5 zeigt den angepassten Dialog, der nur SharePoint-Datenkontexttypen aus der ausgewählten Assembly zur Auswahl anzeigt und den URL der gewünschten SharePoint-Site entgegennimmt.

Interessant in dem erzeugten Code ist vor allem diese Zeile:

```
this._cxInfo.DriverData.SetElementValue(
  "SPUrl", u.AbsoluteUri);
```

Listing 2

Verbindungsoptionen an den SharePoint-LINQ-Kontext übergeben.

```
…
public override object[] GetContextConstructorArguments(IConnectionInfo cxInfo)
{
    return new object[] {cxInfo.DriverData.Element("SPUrl").Value};
}

public override ParameterDescriptor[] GetContextConstructorParameters(
    IConnectionInfo cxInfo)
{
    return new ParameterDescriptor[] { new ParameterDescriptor("requestUrl",
        "System.String") };
}
...
```

Listing 3

Das Laden der SharePoint-Assemblies.

```
public override IEnumerable<string> GetAssembliesToAdd()
{
    string spdllHive = Path.Combine(Environment.GetFolderPath(Environment.
        SpecialFolder.CommonProgramFiles), @"Microsoft Shared\Web Server
        Extensions\14\ISAPI");
    return new[] {Path.Combine(spdllHive, "Microsoft.SharePoint.dll"),
        Path.Combine(spdllHive, "Microsoft.SharePoint.Linq.dll")};
}
```

Hier wird durch das Setzen des Werts von *DriverData* von *IConnectionInfo. DriverData* der vom Benutzer eingegebene URL an das Objekt mit der Verbindungsinformation übergeben.

Fertigstellen des Kontexts

Da die Verbindungsinformation in Form eines URL an LINQPad übergeben wird, muss der Kontexttreiber diese auslesen und dem Konstruktor des SharePoint-*Da-*

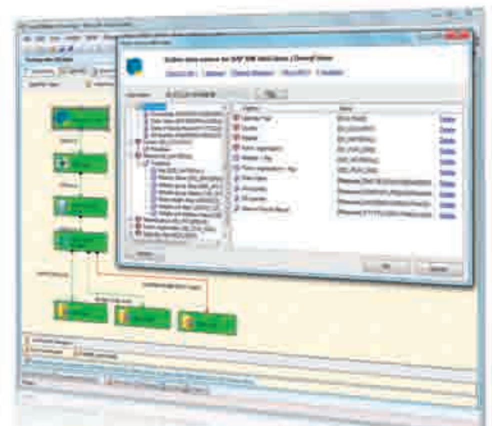
Theobald Software GmbH
Telefon: +49 (0) 711 46 05 99-0
info@theobald-software.com
www.theobald-software.com

simply sophisticated®

Nahtlose SAP® Integration

- .NET – SQL Server* – SharePoint – PowerPivot
- SAP Connectivity
  - Mühelose Datenextraktion für BI
  - synchron und asynchron
  - unterschiedlichste Extraktionstypen
- Schnell, typsicher, stabil, unkompliziert

*Integration Services & Reporting Services



taContext-Objekts übergeben. Dazu stellt LINQPad zwei Methoden bereit, um Argumente und Parameter des Konstruktors wie in Listing 2 zu benennen und zu füllen: *GetConstructorParameters()* und *GetConstructorArguments()*.

Da der Kontexttreiber externe Bibliotheken verwendet, müssen diese ebenfalls eingebunden werden.

Hierzu bietet LINQPad zwei Möglichkeiten an: Einerseits können derartige Abhängigkeiten gemeinsam mit dem Treiber in das LPX-Archiv verpackt, andererseits im Treiber beim Laden eingebunden werden.

Im vorliegenden Fall ist die Weitergabe von SharePoint-Bibliotheken keine Option. Vielmehr geht das Beispiel davon aus, dass sie auf dem Zielsystem vorhanden sein müssen. Die zwei kleinen Zeilen in Listing 3 sorgen dafür, dass LINQPad sie dort auch laden kann.

Damit ist der Datenkontexttreiber in der Lage, Abfragen an SharePoint zu schicken. Allerdings gibt es noch drei kleinere Aufgaben zu lösen: den Kontext „aufräumen“, die Abfrage in CAML-Form anzeigen und den URL der SharePoint-Site anzeigen.

[Abb. 6] LINQPad zeigt CAML-Code einer Abfrage.

Der erste Punkt, das Aufräumen, gehört bekanntermaßen zu einem guten Programmierstil und ist gemäß den bewährten Verfahren der SharePoint Foundation [10] auf einfache Weise durch den Aufruf der Methode *spContext.Dispose()* zu erledigen. Um in Initialisierung und Zerstörung von LINQPad-Kontexttreibern eingreifen zu können, stehen entsprechende Methoden bereit. Listing 4 zeigt, wie ein Datenkontext entfernt wird.

Die zweite Aufgabe fällt weniger in den Bereich der Pflicht als in den der Kür. LINQPad bietet unter anderem die Möglichkeit, Abfragen anzuzeigen. Auch hier ist es möglich, einzugreifen und anstelle von SQL das SharePoint-typische CAML der Abfrage wie in Abbildung 6 anzuzeigen. Hierzu ist bei der Initialisierung des Kontexttreibers die entsprechende Methode wie in Listing 5 zu überschreiben.

Zu guter Letzt wäre es noch schön, wenn der URL zur SharePoint-Site als Orientierung im Schema-Explorer von LINQPad zu sehen wäre. Auch dies ist durch nur eine Zeile Code zu erreichen (Listing 6).

Damit ist der Treiber fertig und kann nun im letzten Schritt als LPX-Archiv eingebunden werden.

Listing 4

SharePoint-Objekte entsorgen.

```
public override void TearDownContext(IConnectionInfo cxInfo, object context,
    QueryExecutionManager executionManager, object[] constructorArguments)
{
    ((IDisposable)context).Dispose();
}
```

Listing 5

Durchreichen des CAML-Codes an LINQPad.

```
public override void InitializeContext(IConnectionInfo cxInfo, object context,
QueryExecutionManager executionManager)
{
    ((Microsoft.SharePoint.Linq.DataContext)context).Log =
        executionManager.SqlTranslationWriter;
}
```

Listing 6

Anzeige des SharePoint-URLs im Schema-Explorer.

```
public override object[] GetContextConstructorArguments(IConnectionInfo cxInfo)
{
    return new object[] {cxInfo.DriverData.Element("SPUrl").Value};
}
```

Fazit

Einen eigenen Datenkontexttreiber für LINQPad zu schreiben ist trotz der zunächst spärlich anmutenden Dokumentation kein Hexenwerk. Der hier entwickelte Treiber für SharePoint basiert in großen Teilen auf den beiden Demoprojekten, die der LINQPad-Autor zur Verfügung stellt und aus denen sich das entsprechende Verhalten mit ein wenig Debugging gut herauslesen lässt. Auf dieser Basis lassen sich sehr einfach Ideen zu eigenen Treibern entwickeln und umsetzen. **[jp]**

[1] LINQPad, www.linqpad.net
[2] How LINQPad Works, www.dotnetpro.de/SL1103LINQPad1
[3] LINQPad Data Context Extensibility Model, www.dotnetpro.de/SL1103LINQPad2
[4] SPMetal, www.dotnetpro.de/SL1103LINQPad3
[5] Object Change Tracking and Optimistic Concurrency, www.dotnetpro.de/SL1103LINQPad4
[6] Extending the Object-Relational Mapping, www.dotnetpro.de/SL1103LINQPad5
[7] How to: Write to Content Databases Using LINQ to SharePoint, www.dotnetpro.de/SL1103LINQPad6
[8] Unsupported LINQ Queries and Two-stage Queries, www.dotnetpro.de/SL1103LINQPad7
[9] .NET Reflector, www.dotnetpro.de/SL1103LINQPad8
[10] Best Practices with SharePoint Foundation – Disposing Objects, www.dotnetpro.de/SL1103LINQPad9