

.NET Beginners



Datenbankgrundlagen – Programmieren mit gespeicherten Prozeduren

von Paul Kimmel

Gespeicherte Prozeduren eignen sich hervorragend, um komplexe Geschäftslösungen für Datenbankanwendungen zu optimieren. Dieser Artikel zeigt, wie Sie gespeicherte Prozeduren in Visual Studio .NET bearbeiten und ausführen und wie Sie sie in Visual-Basic-.NET-Anwendungen einbinden. ADO.NET hat diesen Aspekt der Datenbankprogrammierung für alle erreichbar gemacht.

Auf den folgenden Seiten habe ich bereits existierende gespeicherte Prozeduren herausgegriffen. Diese stammen aus verschiedenen Datenbanken, auf die Sie problemlos zugreifen können. Zum einen ist das die SQL-Server-Version der Datenbank *Northwind*, die bei installierter MSDE auf dem Desktop ausführbar ist. (Alternativ können Sie mit der Access-Version *NWIND.mdb* arbeiten, die zum Lieferumfang von Visual Studio 6 gehört.) Zum anderen verwende ich die SQL-Server-Datenbank *Portal* des IBUYSPY-Portals, die sich kostenlos von Microsoft herunterladen lässt. Das IBUYSPY-Portal ist eine Firmenportalanwendung für eine fiktive Firma IBUYSPY, die Spionageartikel vertreibt. Die gesamte Anwendung können Sie herunterladen und lizenzgebührenfrei als Basis für den Aufbau Ihrer Webanwendungen oder zu Studienzwecken nutzen.

Gespeicherte Prozeduren in VS.NET bearbeiten und ausführen

In Visual Studio .NET können Sie gespeicherte Prozeduren bearbeiten, ausführen und sogar debuggen. Zunächst soll ein kurzer Überblick zeigen, wie man gespeicherte Prozeduren bearbeitet und ausführt – das Debuggen scheint leider auf MS SQL Server beschränkt zu sein und verlangt mehr Aufwand zum Konfigurieren und Einrichten, sodass im Rahmen dieses Artikels nicht darauf eingegangen werden kann.

Visual Studio .NET hat ein so genanntes Server-Explorer-Fenster – hier können Sie nicht nur auf andere Server wie das Ereignisprotokoll, Leistungsindikatoren und andere Windows-Dienste zugreifen, es gibt auch eine Liste von Datenbankverbindungen und SQL-Server-Instanzen (Abbildung 1). Wenn Sie in der SQL-Server-Liste eine einzelne Datenbank markieren, stellt das Kontextmenü Befehle für das Bearbeiten, Ausführen und Debuggen von gespeicherten Prozeduren bereit. Wollen Sie beispielsweise die gespeicherte Prozedur *CustOrderHist* der Datenbank *Northwind* ausführen, klicken Sie mit der rechten Maustaste auf diese gespeicherte Prozedur und wählen AUSFÜHREN VON „GESPEICHERTE PROZEDUR“ – es erscheint ein Dialogfeld, in dem Sie Parameter für die gespeicherte Prozedur eingeben können (Abbildung 2). Das Dialogfeld zeigt den Datentyp des Parameters (*nvarchar*), die Übergaberichtung (*Ein*), den Namen des Parameters (*@CustomerID*) und ein Eingabefeld für den Wert an. [Hinweis: Um eine Tabellenansicht in Visual Studio .NET zu öffnen, doppelklicken Sie auf die Tabelle im Server-Explorer. Daraufhin erscheint ein Fenster mit dem Inhalt der Tabelle.]

Falls Sie die gespeicherte Prozedur nicht selbst geschrieben haben, können Sie sie in Visual Studio den Kontextmenübefehl BEARBEITEN VON „GESPEICHERTE PROZEDUR“ öffnen und feststellen, woher die Daten

kommen. Die als Beispiel dienende gespeicherte Prozedur liest aus den Tabellen *Customers*, *Orders* und *Products* – eine CustomerID (Kundenkennung) zum Testen lässt sich aus der Tabelle *Customers* abrufen. Das Beispiel verwendet die CustomerID *ALFKI* für Alfreds Futterkiste.

Um die gespeicherte Prozedur vom Server-Explorer aus zu testen, geben Sie

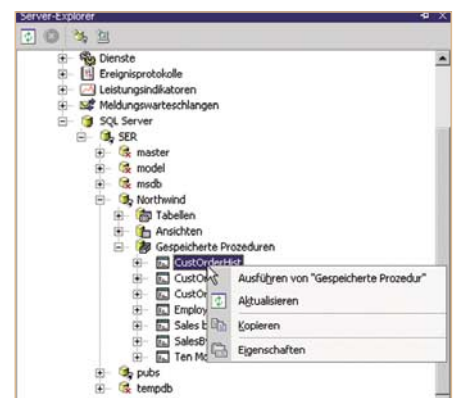


Abb. 1: Die Server-Explorer-Ansicht in Visual Studio .NET mit dem eingblendeten Kontextmenü

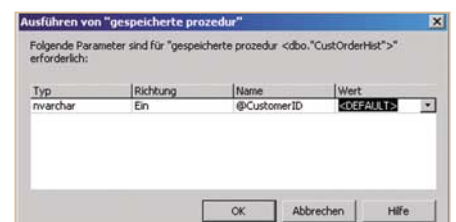


Abb. 2: Im Dialogfeld Ausführen von „gespeicherte Prozedur“ können Sie Eingabeparameter eingeben

Ausgabe	
Datenbankausgabe	
Aniseed Syrup	6
Chartreuse verte	21
Escargots de Bourgogne	40
Flotemysost	20
Grandma's Boysenberry Spread	16
Lakkalikööri	15
Original Frankfurter grüne Soße	2
Raclette Courdavault	15
Rössle Sauerkraut	17
Spegesild	2
Veggie-spread	20
Es gibt keine weiteren Ergebnisse.	
(11 Zeile(n) zurückgegeben)	

Abb. 3: Die Ausgabe einer gespeicherten Prozedur wird an das Ausgabefenster in Visual Studio .NET gesendet

ALFKI in die Spalte *Wert* des Dialogfelds AUSFÜHREN VON „GESPEICHERTE PROZEDUR“ ein und klicken auf OK. Das Ergebnis der gespeicherten Prozedur erscheint im Ausgabefenster (Abbildung 3).

Mit einer Datenbank verbinden

ADO.NET unterstützt herkömmliche Datenbanken als Daten-Provider und auch neuere Daten-Repositories. Die Beispiele in diesem Abschnitt zeigen, wie man die Verbindung zu einer typischen Datenbank herstellt: Dazu sind mehrere Angaben erforderlich – Sie brauchen eine Datenbank, mit der Sie sich verbinden, eine Verbindungszeichenfolge und Visual Basic .NET.

Damit Sie sich mit einer Datenbank verbinden können, müssen Sie wissen, zu welcher Kategorie die Datenbank gehört. Wenn es sich um MS SQL Server ab Version 7.0 aufwärts handelt, müssen Sie die Klassen im Namespace *System.Data.SqlClient* verwenden, für alle anderen Daten-

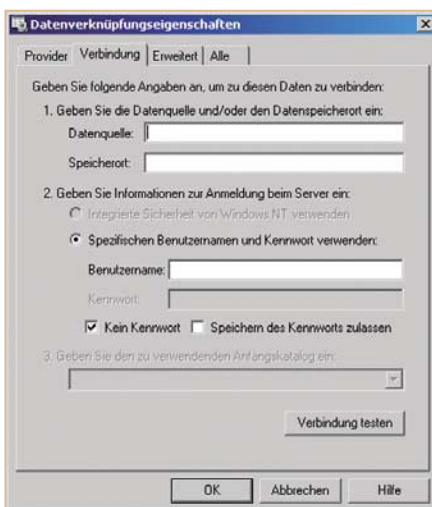
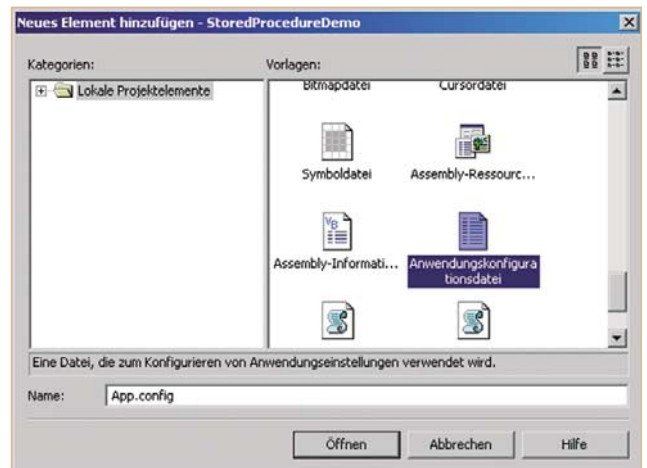


Abb. 5: Mit dem Editor für die Datenverknüpfungseigenschaften lassen sich .udf-Textdateien bearbeiten

Abb. 4: Die Vorlage für die Anwendungskonfigurationsdatei fügt dem Projekt eine *App.config*-Datei hinzu



banken die Klassen im Namespace *System.Data.OleDb*. Die Demonstration arbeitet mit MS SQL Server 2000, sodass die *SqlClient*-Klasse von ADO.NET zum Einsatz kommt.

Die Verbindungszeichenfolge externalisieren

Ich ziehe es vor, die Informationen der Verbindungszeichenfolge in eine *.config*-Datei zu externalisieren. Wenn Sie eine Windows-Anwendung schreiben, können Sie eine Anwendungskonfigurationsdatei in Ihr Projekt einbinden. Bei einer Webanwendung schreiben Sie die Verbindungszeichenfolge in die *Web.config*-Datei, die bei Webanwendungen erstellt wird.

Bequemerweise können Sie die gemeinsam genutzte *AppSettings*-Auflistung der Klasse *System.Configuration.ConfigurationSettings* verwenden, um aus einer Anwendungskonfigurationsdatei zu lesen. Wenn Sie die Informationen der Verbindungszeichenfolge in einem *<appSettings>*-Tag in die *.config*-Datei schreiben, stellt .NET bereits ein Mittel bereit, um diese Informationen zu lesen. Unser Demoprogramm ist eine Windows-Anwendung. Eine *App.config*-Datei können Sie dem Projekt manuell hinzufügen oder DATEI NEUES ELEMENT HINZUFÜGEN wählen und das Anwendungskonfigurationsprogramm im Dialogfeld NEUES ELEMENT HINZUFÜGEN (Abbildung 4) aufrufen. Listing 1 enthält die *App.config*-Datei für unser Beispielprogramm, einschließlich einer initialisierten Verbindungszeichenfolge.

<appSettings>-Elemente werden als Schlüssel- bzw. Wert-Paare hinzugefügt. Der Schlüssel ist der Name, über den Sie auf den Wert zugreifen, und dieser defi-

niert den eigentlichen Wert, den Sie externalisieren wollen. Im Beispiel heißt der Schlüssel *ConnectionString* und der Wert ist eine Verbindungszeichenfolge für eine MS-SQL-Server-Instanz der Northwind-Datenbank. Die Verbindungszeichenfolge kann bei Ihnen auch anders aussehen. Sie lässt sich komfortabel über das Dialogfeld DATENVERKNÜPFUNGSEIGENSCHAFTEN (Abbildung 5) aufbauen, mit dem man *.udf*-Dateien bearbeitet, oder man übernimmt die Verbindungszeichenfolge per Kopieren und Einfügen aus dem Eigenschaftsfenster von Visual Studio .NET.

Nachdem Sie die Verbindungszeichenfolge definiert und in die Datei *App.config* externalisiert haben, können Sie den Wert des *ConnectionString*-Schlüssels mithilfe der Klasse *ConfigurationSettings* lesen. Listing 2 zeigt wie die Verbindungszeichenfolge gelesen und das Verbindungsobjekt initialisiert wird.

Listing 1

Die Verbindungszeichenfolge im *<appSettings>*-Element in einer *.config*-Datei definieren.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="ConnectionString"
         value="data source=SCI\Erewhon;initial catalog=
              Northwind;integrated security=SSPI;persist
              security info= True;workstation id=PTK800;
              packet size=4096" />
  </appSettings>
</configuration>
```

Achtung: Der Parameter *value* ist eine zusammenhängende Zeichenfolge, die in der *.config*-Datei auf ein und derselben Zeile stehen muss.

Die Verbindung öffnen

ADO.NET verwendet ein verbindungsloses Modell. Das heißt nicht, dass Sie ohne Verbindung arbeiten, sondern dass Sie die Verbindung nicht aufrechterhalten müssen. Um mit unserer Datenbank zu kommunizieren, ist es trotzdem erforderlich, ein Verbindungsobjekt zu deklarieren und

Listing 2

Ein Verbindungsobjekt deklarieren und initialisieren

```
Dim Connection As SqlConnection = _
    New SqlConnection(_
        ConfigurationSettings.AppSettings(_
            "ConnectionString"))
```

Listing 3

Ausführlichere Form des Codes von Listing 2

```
Dim ConnectionString As String
ConnectionString = _
    System.Configuration.ConfigurationSettings.
        AppSettings("ConnectionString")
Dim Connection As SqlConnection
Connection = New SqlConnection(ConnectionString)
```

Zum Glück gibt es keinen Grund mehr, so weit-schweifigen Code zu schreiben. Mit der Kurz-version von Listing 1 können Sie viele temporäre Variablen und jede Menge unnötiger Codezeilen einsparen.

Listing 4

Ein Command-Objekt initialisieren

```
1: Dim Connection As SqlConnection = _
2: New SqlConnection(_
3: ConfigurationSettings.AppSettings(_
4: "ConnectionString"))
5:
6: Dim Command As SqlCommand = _
7: New SqlCommand()
8: Command.Connection = Connection
9: Command.CommandText = "Ten Most Expensive Products"
10: Command.CommandType =
    CommandType.StoredProcedure
```

(Die Zeilennummern dienen nur der Orientierung und gehören nicht zum eigentlichen Code.) Die Zeilen 1 bis 5 initialisieren das Verbindungsobjekt, die Zeilen 6 und 7 ein neues *SqlCommand*-Objekt. Zeile 8 weist die Verbindungsinstanz der *Connection*-Eigenschaft des *Command*-Objekts zu. Zeile 9 benennt den auszuführenden Befehl und Zeile 10 gibt an, dass der Text eine gespeicherte Prozedur darstellt. Dieser Code ruft die gespeicherte Prozedur „Ten Most Expensive Products“ auf.

initialisieren. Listing 2 demonstriert, wie man eine externe Verbindungszeichenfolge von einer *App.config*-Datei liest und ein Verbindungsobjekt initialisiert. Die Dekla-

Listing 5

Eine gespeicherte Prozedur aufrufen und die Ergebnisse anzeigen

```
1: Imports System.Data
2: Imports System.Data.SqlClient
3: Imports System.Configuration
4:
5: Public Class Form1
6: Inherits System.Windows.Forms.Form
7:
8: [ Windows Form Designer generated code ]
9:
10: Private Sub Form1_Load(ByVal sender As Object, _
11: ByVal e As System.EventArgs) Handles MyBase.Load
12:
13: NoParameter()
14:
15: End Sub
16:
17: Private Sub NoParameter()
18:
19: Dim Connection As SqlConnection = _
20: New SqlConnection(_
21: ConfigurationSettings.AppSettings(_
22: "ConnectionString"))
23:
24: Dim Command As SqlCommand = _
25: New SqlCommand()
26: Command.Connection = Connection
27: Command.CommandText = "Ten Most Expensive
    Products"
28: Command.CommandType = CommandType.
    StoredProcedure
29:
30: Dim Adapter As SqlDataAdapter = _
31: New SqlDataAdapter(Command)
32:
33: Dim DataSet As DataSet = _
34: New DataSet(Command.CommandText)
35:
36: Adapter.Fill(DataSet)
37: DataGrid1.DataSource = DataSet.Tables(0)
38:
39: End Sub
40:
41: End Class
```

Die Anwendung enthält ein einzelnes Formular mit einem einzelnen *DataGrid*. Die Zeilen 1 bis 3 deklarieren die drei erforderlichen Namespaces. In den Zeilen 30 bis 37 ist der zusätzliche *Adapter*- und *DataSet*-Code zu sehen. Beim Laden des Formulars wird die von der gespeicherten Prozedur zurückgegebene Ergebnismenge im *DataGrid* angezeigt.

Sie suchen die wirklich wichtigen Informationen rund um .NET?

Wir haben Sie!



Ihre Abo-Vorteile:

- Sie erhalten den 128 MB dot.net-USB-Stick*
- Sie sparen rund 10% gegenüber dem Einzelverkauf
- Sie erhalten das *Riesen-Poster* „.NET Framework 2.0“
- Mit der *Jahres-CD* erhalten Sie alle Ausgaben des vergangenen Jahres gratis
- Sie verpassen keine Ausgabe
- Jede Ausgabe inklusive *Heft-CD* mit vielen Tools

Weitere Informationen und Bestellung unter www.dotnet-magazin.de

ration lässt sich auch ausführlicher, wie in Listing 3 gezeigt, formulieren.

Das Command-Objekt erstellen

Command-Objekte (Befehlsobjekte) werden implizit erzeugt oder angelegt. Im Wesentlichen ist das *Command*-Objekt die objektorientierte Darstellung von SQL-Text und Informationen gespeicherter Prozeduren. Um eine gespeicherte Prozedur aufzurufen, müssen Sie ein *Command*-Objekt erzeugen und es mit Informationen über die Datenbank und die Art der Kommunikation mit der Datenbank initialisieren. Listing 4 ist mit Listing 2 kombiniert und zeigt, wie man ein *Command*-Objekt und seine Beziehung zu einer Verbindung initialisiert.

Einen Adapter erstellen

Wenn Sie sich für ein *DataSet* entscheiden – Sie könnten auch einen *DataReader* ver-

wenden – müssen Sie die Lücke zwischen der Verbindung und dem *DataSet* mit einem Adapter überbrücken. Ein rationell denkender Mensch mag sich fragen, warum es so viele Klassen gibt, um Daten aus einer

Warum gibt es so viele Klassen?

Datenbank zu lesen. Die Antwort: Gemeinsame Funktionsmerkmale wurden ausgeklammert, um Redundanz zu vermeiden oder zumindest zu verringern. Stopft man beispielsweise den Code des Adapters in das *DataSet*, würde er die Client-Anwendungen aufblähen und sich außerdem in der *DataTable* wiederholen. Die Trennung der Zuständigkeiten zeugt von einem guten objektorientierten Entwurf.

Sie können einen Adapter erzeugen und das *Command*-Objekt implizit erstellen, indem Sie den SQL-Text und die Verbindung an den Adapter übergeben. Es ist auch möglich, einen Befehl zu erzeugen – wie bereits geschehen – und den Adapter mit dem Befehl zu initialisieren. (Stellen Sie sich den Adapter einfach als Vermittler der Datenübertragung zwischen der Datenbank und dem *DataSet* vor.) Listing 5 zeigt, wie man einen Adapter initialisiert, das *DataSet* füllt und die Daten in einem *DataGrid* präsentiert – das Listing zeigt den gesamten Code einschließlich der Import-Anweisungen.

Eine gespeicherte Prozedur mit Parametern aufrufen

Gespeicherte Prozeduren sind praktisch Funktionen, die in der Datenbank abgelegt sind. Sie müssen in der Lage sein, an die gespeicherten Prozeduren Argumente zu senden, genau wie Sie Argumente an Funktionen und Unterprogramme in Visual Basic .NET übergeben. Außerdem brauchen Sie ein Instrument, das anzeigt, wie diese Parameter verwendet werden. In VB.NET-Code geschieht das mit den Modifizierern *ByVal* und *ByRef*. Bei gespeicherten Prozeduren sind analoge Aufgaben in einer für das Datenbankmodul verständlichen Form zu realisieren – und zwar mit Parameterobjekten.

Eingabeparameter übergeben

Das Beispiel in Listing 6 demonstriert wie ein Parameter an eine gespeicherte Prozedur gesendet wird. Das Listing ist hier nur der Vollständigkeit halber komplett angegeben. Auf den Code gehe ich nicht weiter ein und weise nur darauf hin, dass alles außer den Befehls- und Parameterobjekten mit dem Code identisch ist, der in Listing 5 angegeben und erläutert wurde.

Ausgabeparameter übergeben und abrufen

Ausgabeparameter verwendet man seltener als Eingabeparameter. So haben Sie immer die Möglichkeit, bei einer umfangreichen Ergebnismenge einen Cursor zurückzugeben. In manchen Fällen ist man aber lediglich an einem einzelnen Datenelement interessiert. Nehmen Sie zum Beispiel an, Sie fügen eine neue Zeile ein, für die der Primärschlüssel generiert wird. Möglicherweise möchten Sie diese Information

Listing 6

Eingabeparameter an eine gespeicherte Prozedur senden

```

1: Imports System.Data
2: Imports System.Data.SqlClient
3: Imports System.Configuration
4:
5: Public Class Form1
6: Inherits System.Windows.Forms.Form
7:
8: [ Windows Form Designer generated code ]
9:
10: Private Sub Form1_Load(ByVal sender As Object, _
11: ByVal e As System.EventArgs) Handles MyBase.Load
12:
13: InputParameter()
14:
15: End Sub
16:
17: Public Sub InputParameter()
18:
19: Dim Connection As SqlConnection = _
20: New SqlConnection(_
21: ConfigurationSettings.AppSettings(_
22: "ConnectionString"))
23:
24: Dim Command As SqlCommand = _
25: New SqlCommand()
26: Command.Connection = Connection
27: Command.CommandText = "CustOrderHist"
28: Command.CommandType = CommandType.
29: StoredProcedure
30: Dim Parameter As SqlParameter = _
31: New SqlParameter("@CustomerID", "ALFKI")
32: Parameter.Direction = ParameterDirection.Input
33: Parameter.DbType = DbType.String
34:
35: Command.Parameters.Add(Parameter)
36:
37: Dim Adapter As SqlDataAdapter = _
38: New SqlDataAdapter(Command)
39:
40: Dim DataSet As DataSet = _
41: New DataSet("Order History")
42:
43: Adapter.Fill(DataSet)
44: DataGrid1.DataSource = DataSet.Tables(0)
45:
46: End Sub
47: End Class

```

Die für das Senden des Eingabeparameters geänderten Codezeilen sind fett gedruckt. Zeile 27 zeigt an, dass Sie die gespeicherte Prozedur *CustOrderHist* aufrufen. Das *Parameter*-Objekt wird in Zeile 30 erzeugt und in Zeile 35 der *Parameters*-Auflistung des *Command*-Objekts hinzugefügt. Zeile 31 spezifiziert den Namen und den Wert des Parameters als Argu-

mente an den *SqlParameter*-Konstruktor und in den Zeilen 32 und 33 sind Übergaberichtung und Parametertyp angegeben. Falls die gespeicherte Prozedur mehr als einen Parameter verlangt, wiederholen Sie den Code in den Zeilen 30 bis 35 für jeden Parameter, wobei jeweils Name, Wert, Richtung und Typ entsprechend zu ersetzen sind.

aus der Datenbank abrufen und in Ihre Anwendung übernehmen. Hier bietet sich ein Ausgabeparameter an.

Ausgabeparameter sind mit *ByRef*-Argumenten vergleichbar. Der Wert eines Ausgabeparameters lässt sich nach zwei Verfahren lesen: Man deklariert das Parameterobjekt als separate Variable und liest den Wert nach Ausführung der gespeicherten Prozedur oder man liest den Wert über die *Parameters*-Auflistung des *Command*-Objekts. Der Auszug aus dem Code des IBUYSPY-Portals in Listing 7 zeigt, wie ei-

ne gespeicherte Prozedur mit einem Ausgabeparameter aufgerufen wird und wie die Daten nach dem Aufruf der Prozedur übernommen werden.

Zusammenfassung

Gespeicherte Prozeduren sind ein leistungsfähiger Aspekt der Datenbankprogrammierung für alle Plattformen. Damit lässt sich der datenbankintensive Teil des Codes auf den Server verlagern und man kann die Zuständigkeiten trennen, indem man die Datenbankadministratoren da-

mit beauftragt, den Code der gespeicherten Prozedur zu verwalten.

Wenn es Ihnen wie mir geht, müssen Sie wohl auch viele gespeicherte Prozeduren selbst schreiben und zudem den Code, der sie aufruft. Die Beispiele in diesem Artikel sollen Ihnen dabei ein Wegweiser sein. ●

Paul Kimmel ist freiberuflicher Autor für Developer.com und CodeGuru.com. Er bietet Ihnen Hilfe beim Entwurf und beim Erstellen Ihrer .NET-Lösungen an. Sie können Paul Kimmel unter pkimmel@softconcepts.com erreichen.

Listing 7

Eine gespeicherte Prozedur mit einem Ausgabeparameter aufrufen

```

1: public int AddAnnouncement(int moduleId, int itemId, String userName, _
2: String title, DateTime expireDate, String description, _
3: String moreLink, String mobileMoreLink)
4: {
5:
6: if (userName.Length < 1) {
7:     userName = "unknown";
8: }
9:
10: // Create Instance of Connection and Command Object
11: SqlConnection myConnection = _
12: new SqlConnection(ConfigurationSettings.AppSettings["connectionString"]);
13:
14: SqlCommand myCommand = _
15: new SqlCommand("AddAnnouncement", myConnection);
16:
17: // Mark the Command as a SPROC
18: myCommand.CommandType = CommandType.StoredProcedure;
19:
20: // Add Parameters to SPROC
21: SqlParameter parameterItemID = _
22: new SqlParameter("@ItemID", SqlDbType.Int, 4);
23: parameterItemID.Direction = ParameterDirection.Output;
24: myCommand.Parameters.Add(parameterItemID);
25:
26: SqlParameter parameterModuleID = _
27: new SqlParameter("@ModuleID", SqlDbType.Int, 4);
28: parameterModuleID.Value = moduleId;
29: myCommand.Parameters.Add(parameterModuleID);
30:
31: SqlParameter parameterUserName = _
32: new SqlParameter("@UserName", SqlDbType.NVarChar, 100);
33: parameterUserName.Value = userName;
34: myCommand.Parameters.Add(parameterUserName);
35:
36: SqlParameter parameterTitle = _
37: new SqlParameter("@Title", SqlDbType.NVarChar, 150);
38: parameterTitle.Value = title;
39: myCommand.Parameters.Add(parameterTitle);
40:
41: SqlParameter parameterMoreLink = _
42: new SqlParameter("@MoreLink", SqlDbType.NVarChar, 150);
43: parameterMoreLink.Value = moreLink;
44: myCommand.Parameters.Add(parameterMoreLink);
45:
46: SqlParameter parameterMobileMoreLink = _
47: new SqlParameter("@MobileMoreLink", SqlDbType.NVarChar, 150);
48: parameterMobileMoreLink.Value = mobileMoreLink;
49: myCommand.Parameters.Add(parameterMobileMoreLink);
50:
51: SqlParameter parameterExpireDate = _
52: new SqlParameter("@ExpireDate", SqlDbType.DateTime, 8);
53: parameterExpireDate.Value = expireDate;
54: myCommand.Parameters.Add(parameterExpireDate);
55:
56: SqlParameter parameterDescription = _
57: new SqlParameter("@Description", SqlDbType.NVarChar, 2000);
58: parameterDescription.Value = description;
59: myCommand.Parameters.Add(parameterDescription);
60:
61: myConnection.Open();
62: myCommand.ExecuteNonQuery();
63: myConnection.Close();
64:
65: return (int)parameterItemID.Value;
66: }
    
```

AddAnnouncement fügt eine Ankündigung in die Datenbank des IBUYSPY-Portals ein. Mittlerweile sollte Ihnen der größte Teil des Codes vertraut sein. Er übernimmt alle Eingabeparameter als Argumente an *AddAnnouncement* und verwendet diese als Parameter, die an die gespeicherte Prozedur zu übergeben sind. Die Elemente für den Aufruf der gespeicherten Prozedur sind immer noch die gleichen – das Listing ist nur länger, weil Sie mehrere Parameter erzeugen und hinzufügen. Uns interessieren hier vor allem die Zeilen 21 bis 24 und die Zeile 65. Die

Zeilen 21 bis 24 erzeugen einen Ausgabeparameter für die gespeicherte Prozedur. Dazu wird das Argument *ParameterDirection.Output* an den *SqlParameter*-Konstruktor übergeben. (Denken Sie daran, dass Visual Basic .NET das Überladen von Methoden und Konstruktoren unterstützt. Der Aufruf des *SqlParameter*-Konstruktors ist ein Beispiel für einen überladenen Konstruktor.) Nach dem Aufruf der gespeicherten Prozedur können Sie den von der Prozedur zurückgegebenen Wert lesen, wie es in Zeile 65 geschieht.