Contents

SQL Server and ADO.NET

DataSets, DataTables, and DataViews

ADO.NET

2/8/2020 • 2 minutes to read • Edit Online

ADO.NET is a set of classes that expose data access services for .NET Framework programmers. ADO.NET provides a rich set of components for creating distributed, data-sharing applications. It is an integral part of the .NET Framework, providing access to relational, XML, and application data. ADO.NET supports a variety of development needs, including the creation of front-end database clients and middle-tier business objects used by applications, tools, languages, or Internet browsers.

In This Section

What's New in ADO.NET
Introduces features that are new in ADO.NET.

ADO.NET Overview
Provides an introduction to the design and components of ADO.NET.

Entity Framework
Describes how to create applications using the Entity Framework.

Securing ADO.NET Applications
Describes secure coding practices when using ADO.NET.

Data Type Mappings in ADO.NET
Describes data type mappings between .NET Framework data types and the .NET Framework data providers.

DataSets, DataTables, and DataViews
Describes how to create and use `DataSets`, typed `DataSets`, `DataTables`, and `DataViews`.

LINQ to DataSet
Provides information about LINQ to DataSet, including programming examples.

Retrieving and Modifying Data in ADO.NET
Describes how to connect to a data source and how to retrieve and modify data using `Commands`, `DataReaders`, and `DataAdapters`.

SQL Server and ADO.NET
Describes how to work with features and functionality that are specific to SQL Server.

Oracle and ADO.NET
Describes features and behaviors that are specific to the .NET Framework Data Provider for Oracle.

Related Sections

Language-Integrated Query (LINQ) - C#
Provides links to LINQ topics and samples using C#.

Language-Integrated Query (LINQ) - Visual Basic
Provides links to LINQ topics and samples using Visual Basic.

WCF Data Services 4.5
Describes how to use WCF Data Services to deploy data services on the Web or an intranet that implement the Open Data Protocol (OData).

.NET Framework Development Guide

Provides links to information about standard development tasks in the .NET Framework.

Samples and tutorials

Provides a list of .NET samples and tutorials.

See also

- Accessing data in Visual Studio
- ADO.NET Overview

What's New in ADO.NET

2/4/2020 • 2 minutes to read • Edit Online

The following features are new in ADO.NET in the .NET Framework 4.5.

SqlClient Data Provider

The following features are new in the .NET Framework Data Provider for SQL Server in .NET Framework 4.5:

- The ConnectRetryCount and ConnectRetryInterval connection string keywords (ConnectionString) let you control the idle connection resiliency feature.
- Streaming support from SQL Server to an application supports scenarios where data on the server is unstructured. See SqlClient Streaming Support for more information.
- Support has been added for asynchronous programming. See Asynchronous Programming for more information.
- Connection failures will now be logged in the extended events log. For more information, see Data Tracing in ADO.NET.
- SqlClient now has support for SQL Server's high availability, disaster recovery feature, AlwaysOn. For more information, see SqlClient Support for High Availability, Disaster Recovery.
- A password can be passed as a SecureString when using SQL Server Authentication. See SqlCredential for more information.
- When `TrustServerCertificate` is false and `Encrypt` is true, the server name (or IP address) in a SQL Server SSL certificate must exactly match the server name (or IP address) specified in the connection string. Otherwise, the connection attempt will fail. For more information, see the description of the `Encrypt` connection option in ConnectionString.

  If this change causes an existing application to no longer connect, you can fix the application using one of the following:

  - Issue a certificate that specifies the short name in the Common Name (CN) or Subject Alternative Name (SAN) field. This solution will work for database mirroring.
  - Add an alias that maps the short name to the fully-qualified domain name.
  - Use the fully-qualified domain name in the connection string.
- SqlClient supports Extended Protection. For more information about Extended Protection, see Connecting to the Database Engine Using Extended Protection.
- SqlClient supports connections to LocalDB databases. For more information, see SqlClient Support for LocalDB.
- `Type System Version=SQL Server 2012;` is new value to pass to the `Type System Version` connection property. The `Type System Version=Latest;` value is now obsolete and has been made equivalent to `Type System Version=SQL Server 2008;`. For more information, see ConnectionString.
- SqlClient provides additional support for sparse columns, a feature that was added in SQL Server 2008. If your application already accesses data in a table that uses sparse columns, you should see an increase in performance. The IsColumnSet column of GetSchemaTable indicates if a column is a sparse column that is a

member of a column set. GetSchema indicates if a column is a sparse column (see SQL Server Schema Collections for more information). For more information about sparse columns, see Use Sparse Columns.

- The assembly Microsoft.SqlServer.Types.dll, which contains the spatial data types, has been upgraded from version 10.0 to version 11.0. Applications that reference this assembly may fail. For more information, see Breaking Changes to Database Engine Features.

ADO.NET Entity Framework

The .NET Framework 4.5 adds APIs that enable new scenarios when working with the Entity Framework 5.0. For more information about improvements and features that were added to the Entity Framework 5.0, see the following topics: What's New and Entity Framework Releases and Versioning.

See also

- ADO.NET
- ADO.NET Overview
- SQL Server and ADO.NET
- What's New in WCF Data Services 5.0

ADO.NET Overview

2/4/2020 • 2 minutes to read • Edit Online

ADO.NET provides consistent access to data sources such as SQL Server and XML, and to data sources exposed through OLE DB and ODBC. Data-sharing consumer applications can use ADO.NET to connect to these data sources and retrieve, handle, and update the data that they contain.

ADO.NET separates data access from data manipulation into discrete components that can be used separately or in tandem. ADO.NET includes .NET Framework data providers for connecting to a database, executing commands, and retrieving results. Those results are either processed directly, placed in an ADO.NET DataSet object in order to be exposed to the user in an ad hoc manner, combined with data from multiple sources, or passed between tiers. The `DataSet` object can also be used independently of a .NET Framework data provider to manage data local to the application or sourced from XML.

The ADO.NET classes are found in System.Data.dll, and are integrated with the XML classes found in System.Xml.dll. For sample code that connects to a database, retrieves data from it, and then displays that data in a console window, see ADO.NET Code Examples.

ADO.NET provides functionality to developers who write managed code similar to the functionality provided to native component object model (COM) developers by ActiveX Data Objects (ADO). We recommend that you use ADO.NET, not ADO, for accessing data in your .NET applications.

ADO.NET provides the most direct method of data access within the .NET Framework. For a higher-level abstraction that allows applications to work against a conceptual model instead of the underlying storage model, see the ADO.NET Entity Framework.

Privacy Statement: The System.Data.dll, System.Data.Design.dll, System.Data.OracleClient.dll, System.Data.SqlXml.dll, System.Data.Linq.dll, System.Data.SqlServerCe.dll, and System.Data.DataSetExtensions.dll assemblies do not distinguish between a user's private data and non-private data. These assemblies do not collect, store, or transport any user's private data. However, third-party applications might collect, store, or transport a user's private data using these assemblies.

In This Section

ADO.NET Architecture
Provides an overview of the architecture and components of ADO.NET.

ADO.NET Technology Options and Guidelines
Describes the products and technologies included with the Entity Data Platform.

LINQ and ADO.NET
Describes how Language-Integrated Query (LINQ) is implemented in ADO.NET and provides links to relevant topics.

.NET Framework Data Providers
Provides an overview of the design of the .NET Framework data provider and of the .NET Framework data providers that are included with ADO.NET.

ADO.NET DataSets
Provides an overview of the `DataSet` design and components.

Side-by-Side Execution in ADO.NET

Discusses differences in ADO.NET versions and their effect on side-by-side execution and application compatibility.

ADO.NET Code Examples
Provides code samples that retrieve data using the ADO.NET data providers.

Related Sections

What's New in ADO.NET
Introduces features that are new in ADO.NET.

Securing ADO.NET Applications
Describes secure coding practices when using ADO.NET.

Data Type Mappings in ADO.NET
Describes data type mappings between .NET Framework data types and the .NET Framework data providers.

Retrieving and Modifying Data in ADO.NET
Describes how to connect to a data source, retrieve data, and modify data. This includes `DataReaders` and `DataAdapters`.

See also

- ADO.NET
- Accessing data in Visual Studio

ADO.NET Architecture

2/4/2020 • 4 minutes to read • Edit Online

Data processing has traditionally relied primarily on a connection-based, two-tier model. As data processing increasingly uses multi-tier architectures, programmers are switching to a disconnected approach to provide better scalability for their applications.

ADO.NET Components

The two main components of ADO.NET for accessing and manipulating data are the .NET Framework data providers and the DataSet.

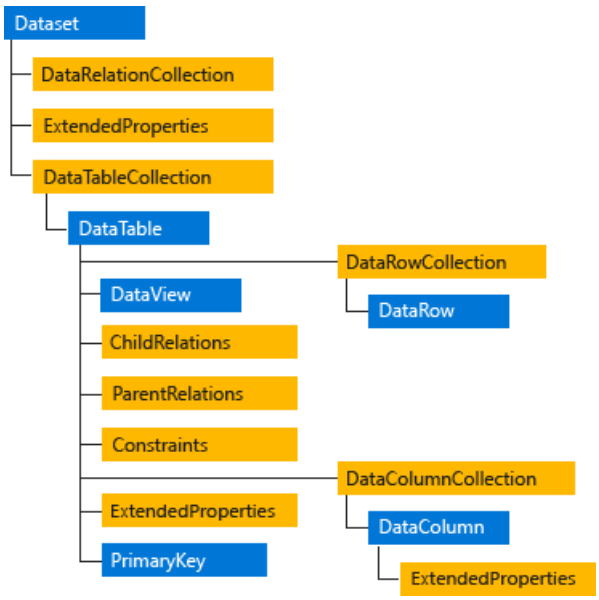
.NET Framework Data Providers

The .NET Framework Data Providers are components that have been explicitly designed for data manipulation and fast, forward-only, read-only access to data. The `Connection` object provides connectivity to a data source. The `Command` object enables access to database commands to return data, modify data, run stored procedures, and send or retrieve parameter information. The `DataReader` provides a high-performance stream of data from the data source. Finally, the `DataAdapter` provides the bridge between the `DataSet` object and the data source. The `DataAdapter` uses `Command` objects to execute SQL commands at the data source to both load the `DataSet` with data and reconcile changes that were made to the data in the `DataSet` back to the data source. For more information, see .NET Framework Data Providers and Retrieving and Modifying Data in ADO.NET.

The DataSet

The ADO.NET `DataSet` is explicitly designed for data access independent of any data source. As a result, it can be used with multiple and differing data sources, used with XML data, or used to manage data local to the application. The `DataSet` contains a collection of one or more DataTable objects consisting of rows and columns of data, and also primary key, foreign key, constraint, and relation information about the data in the `DataTable` objects. For more information, see DataSets, DataTables, and DataViews.

The following diagram illustrates the relationship between a .NET Framework data provider and a `DataSet`.



ADO.NET architecture

Choosing a DataReader or a DataSet

When you decide whether your application should use a `DataReader` (see Retrieving Data Using a DataReader) or a

`DataSet` (see DataSets, DataTables, and DataViews), consider the type of functionality that your application requires. Use a `DataSet` to do the following:

- Cache data locally in your application so that you can manipulate it. If you only need to read the results of a query, the `DataReader` is the better choice.
- Remote data between tiers or from an XML Web service.
- Interact with data dynamically such as binding to a Windows Forms control or combining and relating data from multiple sources.
- Perform extensive processing on data without requiring an open connection to the data source, which frees the connection to be used by other clients.

If you do not require the functionality provided by the `DataSet`, you can improve the performance of your application by using the `DataReader` to return your data in a forward-only, read-only manner. Although the `DataAdapter` uses the `DataReader` to fill the contents of a `DataSet` (see Populating a DataSet from a DataAdapter), by using the `DataReader`, you can boost performance because you will save memory that would be consumed by the `DataSet`, and avoid the processing that is required to create and fill the contents of the `DataSet`.

LINQ to DataSet

LINQ to DataSet provides query capabilities and compile-time type checking over data cached in a DataSet object. It allows you to write queries in one of the .NET Framework development language, such as C# or Visual Basic. For more information, see LINQ to DataSet.

LINQ to SQL

LINQ to SQL supports queries against an object model that is mapped to the data structures of a relational database without using an intermediate conceptual model. Each table is represented by a separate class, tightly coupling the object model to the relational database schema. LINQ to SQL translates language-integrated queries in the object model into Transact-SQL and sends them to the database for execution. When the database returns the results, LINQ to SQL translates the results back into objects. For more information, see LINQ to SQL.

ADO.NET Entity Framework

The ADO.NET Entity Framework is designed to enable developers to create data access applications by programming against a conceptual application model instead of programming directly against a relational storage schema. The goal is to decrease the amount of code and maintenance required for data-oriented applications. For more information, see ADO.NET Entity Framework.

WCF Data Services

WCF Data Services is used to deploy data services on the Web or an intranet. The data is structured as entities and relationships according to the specifications of the Entity Data Model. Data deployed on this model is addressable by standard HTTP protocol. For more information, see WCF Data Services 4.5.

XML and ADO.NET

ADO.NET leverages the power of XML to provide disconnected access to data. ADO.NET was designed hand-in-hand with the XML classes in the .NET Framework; both are components of a single architecture.

ADO.NET and the XML classes in the .NET Framework converge in the `DataSet` object. The `DataSet` can be populated with data from an XML source, whether it is a file or an XML stream. The `DataSet` can be written as World-Wide Web Consortium (W3C) compliant XML that includes its schema as XML schema definition language

(XSD) schema, regardless of the source of the data in the `DataSet`. Because of the native serialization format of the `DataSet` is XML, it is an excellent medium for moving data between tiers, making the `DataSet` an optimal choice for remoting data and schema context to and from an XML Web service. For more information, see XML Documents and Data.

See also

- ADO.NET Overview

ADO.NET Technology Options and Guidelines

2/8/2020 • 3 minutes to read • Edit Online

The ADO.NET Data Platform is a multi-release strategy to decrease the amount of coding and maintenance required for developers by enabling them to program against conceptual entity data models. This platform includes the ADO.NET Entity Framework and related technologies.

Entity Framework

The ADO.NET Entity Framework is designed to enable developers to create data access applications by programming against a conceptual application model instead of programming directly against a relational storage schema. The goal is to decrease the amount of code and maintenance required for data-oriented applications. For more information, see ADO.NET Entity Framework.

Entity Data Model (EDM)

An Entity Data Model (EDM) is a design specification that defines application data as sets of entities and relationships. Data in this model supports object-relational mapping and data programmability across application boundaries.

Object Services

Object Services allows programmers to interact with the conceptual model through a set of common language runtime (CLR) classes. These classes can be automatically generated from the conceptual model or can be developed independently to reflect the structure of the conceptual model. Object Services also provides infrastructure support for the Entity Framework, including services such as state management, change tracking, identity resolution, loading and navigating relationships, propagating object changes to database modifications, and query building support for Entity SQL. For more information, see Object Services Overview (Entity Framework).

LINQ to Entities

LINQ to Entities is a language-integrated query (LINQ) implementation that allows developers to create strongly-typed queries against the Entity Framework object context by using LINQ expressions and LINQ standard query operators. LINQ to Entities allows developers to work against a conceptual model with a flexible object-relational mapping across Microsoft SQL Server and third-party databases. For more information, see LINQ to Entities.

Entity SQL

Entity SQL is a text-based query language designed to interact with an Entity Data Model. Entity SQL is an SQL dialect that contains constructs for querying in terms of higher-level modeling concepts, such as inheritance, complex types, and explicit relationships. Developers can also use Entity SQL directly with Object Services. For more information, see Entity SQL Language.

EntityClient

EntityClient is a new .NET Framework data provider used for interacting with an Entity Data Model. EntityClient follows the .NET Framework data provider pattern of exposing EntityConnection and EntityCommand objects that return an EntityDataReader. EntityClient works with the Entity SQL language, providing flexible mapping to storage-specific data providers. For more information, see EntityClient Provider for the Entity Framework.

Entity Data Model Tools

The Entity Framework provides command-line tools, wizards, and designers to facilitate building EDM applications. The EntityDataSource control supports data binding scenarios based on the EDM. The programming surface of the EntityDataSource control is similar to other data source controls in Visual Studio. For more information, see

ADO.NET Entity Data Model Tools.

LINQ to SQL

LINQ to SQL is an object relational mapping (OR/M) implementation that allows you to model a SQL Server database by using .NET Framework classes. LINQ to SQL allows you to query your database by using LINQ, as well as update, insert, and delete data from it. LINQ to SQL supports transactions, views, and stored procedures, providing an easy way to integrate data validation and business logic rules into your data model. You can use the Object Relational Designer (O/R Designer) to model the entity classes and associations that are based on objects in a database. For more information, see LINQ to SQL Tools in Visual Studio.

WCF Data Services

WCF Data Services deploys data services on the Web or on an intranet. The data is structured as entities and relationships according to the specifications of the Entity Data Model. Data deployed on this model is addressable by standard HTTP protocol. For more information, see WCF Data Services 4.5.

See also

- ADO.NET Overview
- What's New in ADO.NET

LINQ and ADO.NET

2/8/2020 • 3 minutes to read • Edit Online

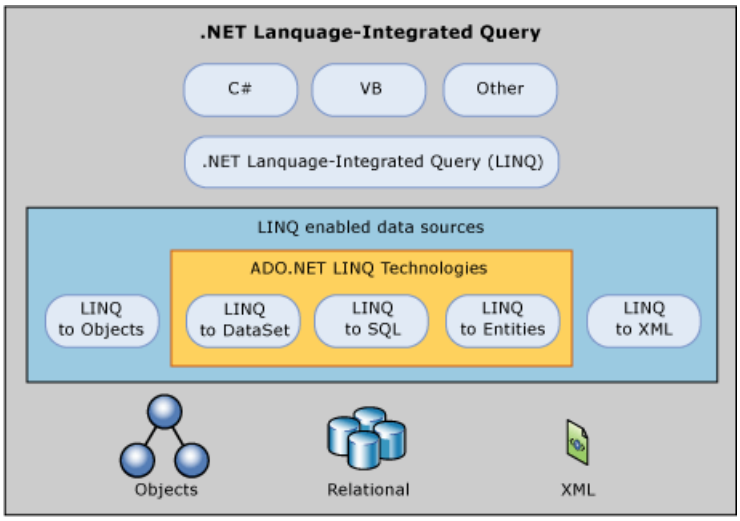
Today, many business developers must use two (or more) programming languages: a high-level language for the business logic and presentation layers (such as Visual C# or Visual Basic), and a query language to interact with the database (such as Transact-SQL). This requires the developer to be proficient in several languages to be effective, and also causes language mismatches in the development environment. For example, an application that uses a data access API to execute a query against a database specifies the query as a string literal by using quotation marks. This query string is unreadable to the compiler and is not checked for errors, such as invalid syntax or whether the columns or rows it references actually exist. There is no type checking of the query parameters and no `IntelliSense` support, either.

Language-Integrated Query (LINQ) enables developers to form set-based queries in their application code, without having to use a separate query language. You can write LINQ queries against various enumerable data sources (that is, a data source that implements the IEnumerable interface), such as in-memory data structures, XML documents, SQL databases, and DataSet objects. Although these enumerable data sources are implemented in various ways, they all expose the same syntax and language constructs. Because queries can be formed in the programming language itself, you do not have to use another query language that is embedded as string literals that cannot be understood or verified by the compiler. Integrating queries into the programming language also enables Visual Studio programmers to be more productive by providing compile-time type and syntax checking, and `IntelliSense`. These features reduce the need for query debugging and error fixing.

Transferring data from SQL tables into objects in memory is often tedious and error-prone. The LINQ provider implemented by LINQ to DataSet and LINQ to SQL converts the source data into IEnumerable-based object collections. The programmer always views the data as an IEnumerable collection, both when you query and when you update. Full `IntelliSense` support is provided for writing queries against those collections.

There are three separate ADO.NET Language-Integrated Query (LINQ) technologies: LINQ to DataSet, LINQ to SQL, and LINQ to Entities. LINQ to DataSet provides richer, optimized querying over the DataSet and LINQ to SQL enables you to directly query SQL Server database schemas, and LINQ to Entities allows you to query an Entity Data Model.

The following diagram provides an overview of how the ADO.NET LINQ technologies relate to high-level programming languages and LINQ-enabled data sources.



For more information about LINQ, see Language Integrated Query (LINQ).

The following sections provide more information about LINQ to DataSet, LINQ to SQL, and LINQ to Entities.

LINQ to DataSet

The DataSet is a key element of the disconnected programming model that ADO.NET is built on, and is widely used. LINQ to DataSet enables developers to build richer query capabilities into DataSet by using the same query formulation mechanism that is available for many other data sources. For more information, see LINQ to DataSet.

LINQ to SQL

LINQ to SQL is a useful tool for developers who do not require mapping to a conceptual model. By using LINQ to SQL, you can use the LINQ programming model directly over existing database schema. LINQ to SQL enables developers to generate .NET Framework classes that represent data. Rather than mapping to a conceptual data model, these generated classes map directly to database tables, views, stored procedures, and user-defined functions.

With LINQ to SQL, developers can write code directly against the storage schema using the same LINQ programming pattern as in-memory collections and the DataSet, in addition to other data sources such as XML. For more information, see LINQ to SQL.

LINQ to Entities

Most applications are currently written on top of relational databases. At some point, these applications will need to interact with the data represented in a relational form. Database schemas are not always ideal for building applications, and the conceptual models of application are not the same as the logical models of databases. The Entity Data Model is a conceptual data model that can be used to model the data of a particular domain so that applications can interact with data as objects. For more information, see ADO.NET Entity Framework.

Through the Entity Data Model, relational data is exposed as objects in the .NET environment. This makes the object layer an ideal target for LINQ support, allowing developers to formulate queries against the database from the language used to build the business logic. This capability is known as LINQ to Entities. For more information, see LINQ to Entities.

See also

- LINQ to DataSet
- LINQ to SQL
- LINQ to Entities
- Language Integrated Query (LINQ)
- ADO.NET Overview

.NET Framework Data Providers

12/24/2019 • 8 minutes to read • Edit Online

A .NET Framework data provider is used for connecting to a database, executing commands, and retrieving results. Those results are either processed directly, placed in a DataSet in order to be exposed to the user as needed, combined with data from multiple sources, or remoted between tiers. .NET Framework data providers are lightweight, creating a minimal layer between the data source and code, increasing performance without sacrificing functionality.

The following table lists the data providers that are included in the .NET Framework.

.NET FRAMEWORK DATA PROVIDER	DESCRIPTION
.NET Framework Data Provider for SQL Server	Provides data access for Microsoft SQL Server. Uses the System.Data.SqlClient namespace.
.NET Framework Data Provider for OLE DB	For data sources exposed by using OLE DB. Uses the System.Data.OleDb namespace.
.NET Framework Data Provider for ODBC	For data sources exposed by using ODBC. Uses the System.Data.Odbc namespace.
.NET Framework Data Provider for Oracle	For Oracle data sources. The .NET Framework Data Provider for Oracle supports Oracle client software version 8.1.7 and later, and uses the System.Data.OracleClient namespace.
EntityClient Provider	Provides data access for Entity Data Model (EDM) applications. Uses the System.Data.EntityClient namespace.
.NET Framework Data Provider for SQL Server Compact 4.0.	Provides data access for Microsoft SQL Server Compact 4.0. Uses the System.Data.SqlServerCe namespace.

Core Objects of .NET Framework Data Providers

The following table outlines the four core objects that make up a .NET Framework data provider.

OBJECT	DESCRIPTION
`Connection`	Establishes a connection to a specific data source. The base class for all `Connection` objects is the DbConnection class.
`Command`	Executes a command against a data source. Exposes `Parameters` and can execute in the scope of a `Transaction` from a `Connection`. The base class for all `Command` objects is the DbCommand class.
`DataReader`	Reads a forward-only, read-only stream of data from a data source. The base class for all `DataReader` objects is the DbDataReader class.

OBJECT	DESCRIPTION
`DataAdapter`	Populates a `DataSet` and resolves updates with the data source. The base class for all `DataAdapter` objects is the DbDataAdapter class.

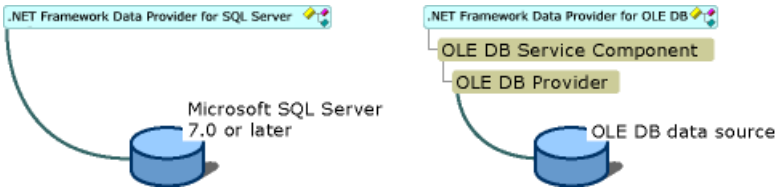
In addition to the core classes listed in the table earlier in this document, a .NET Framework data provider also contains the classes listed in the following table.

OBJECT	DESCRIPTION
`Transaction`	Enlists commands in transactions at the data source. The base class for all `Transaction` objects is the DbTransaction class. ADO.NET also provides support for transactions using classes in the System.Transactions namespace.
`CommandBuilder`	A helper object that automatically generates command properties of a `DataAdapter` or derives parameter information from a stored procedure and populates the `Parameters` collection of a `Command` object. The base class for all `CommandBuilder` objects is the DbCommandBuilder class.
`ConnectionStringBuilder`	A helper object that provides a simple way to create and manage the contents of connection strings used by the `Connection` objects. The base class for all `ConnectionStringBuilder` objects is the DbConnectionStringBuilder class.
`Parameter`	Defines input, output, and return value parameters for commands and stored procedures. The base class for all `Parameter` objects is the DbParameter class.
`Exception`	Returned when an error is encountered at the data source. For an error encountered at the client, .NET Framework data providers throw a .NET Framework exception. The base class for all `Exception` objects is the DbException class.
`Error`	Exposes the information from a warning or error returned by a data source.
`ClientPermission`	Provided for .NET Framework data provider code access security attributes. The base class for all `ClientPermission` objects is the DBDataPermission class.

.NET Framework Data Provider for SQL Server (SqlClient)

The .NET Framework Data Provider for SQL Server (SqlClient) uses its own protocol to communicate with SQL Server. It is lightweight and performs well because it is optimized to access a SQL Server directly without adding an OLE DB or Open Database Connectivity (ODBC) layer. The following illustration contrasts the .NET Framework Data Provider for SQL Server with the .NET Framework Data Provider for OLE DB. The .NET Framework Data Provider for OLE DB communicates to an OLE DB data source through both the OLE DB Service component, which provides connection pooling and transaction services, and the OLE DB provider for the data source.

> **NOTE**
>
> The .NET Framework Data Provider for ODBC has a similar architecture to the .NET Framework Data Provider for OLE DB; for example, it calls into an ODBC Service Component.



Comparison of the .NET Framework Data Provider for SQL Server and the .NET Framework Data Provider for OLE DB

The .NET Framework Data Provider for SQL Server classes are located in the System.Data.SqlClient namespace.

The .NET Framework Data Provider for SQL Server supports both local and distributed transactions. For distributed transactions, the .NET Framework Data Provider for SQL Server, by default, automatically enlists in a transaction and obtains transaction details from Windows Component Services or System.Transactions. For more information, see Transactions and Concurrency.

The following code example shows how to include the `System.Data.SqlClient` namespace in your applications.

```
Imports System.Data.SqlClient
```

```
using System.Data.SqlClient;
```

.NET Framework Data Provider for OLE DB

The .NET Framework Data Provider for OLE DB (OleDb) uses native OLE DB through COM interop to enable data access. The .NET Framework Data Provider for OLE DB supports both local and distributed transactions. For distributed transactions, the .NET Framework Data Provider for OLE DB, by default, automatically enlists in a transaction and obtains transaction details from Windows Component Services. For more information, see Transactions and Concurrency.

The following table shows the providers that have been tested with ADO.NET.

DRIVER	PROVIDER
SQLOLEDB	Microsoft OLE DB provider for SQL Server
MSDAORA	Microsoft OLE DB provider for Oracle
Microsoft.Jet.OLEDB.4.0	OLE DB provider for Microsoft Jet

> **NOTE**
>
> Using an Access (Jet) database as a data source for multithreaded applications, such as ASP.NET applications, is not recommended. If you must use Jet as a data source for an ASP.NET application, realize that ASP.NET applications connecting to an Access database can encounter connection problems.

The .NET Framework Data Provider for OLE DB does not support OLE DB version 2.5 interfaces. OLE DB Providers that require support for OLE DB 2.5 interfaces will not function correctly with the .NET Framework Data Provider

for OLE DB. This includes the Microsoft OLE DB provider for Exchange and the Microsoft OLE DB provider for Internet Publishing.

The .NET Framework Data Provider for OLE DB does not work with the OLE DB provider for ODBC (MSDASQL). To access an ODBC data source using ADO.NET, use the .NET Framework Data Provider for ODBC.

.NET Framework Data Provider for OLE DB classes are located in the System.Data.OleDb namespace. The following code example shows how to include the `System.Data.OleDb` namespace in your applications.

```
Imports System.Data.OleDb
```

```
using System.Data.OleDb;
```

.NET Framework Data Provider for ODBC

The .NET Framework Data Provider for ODBC (Odbc) uses the native ODBC Driver Manager (DM) to enable data access. The ODBC data provider supports both local and distributed transactions. For distributed transactions, the ODBC data provider, by default, automatically enlists in a transaction and obtains transaction details from Windows Component Services. For more information, see Transactions and Concurrency.

The following table shows the ODBC drivers tested with ADO.NET.

DRIVER
SQL Server
Microsoft ODBC for Oracle
Microsoft Access Driver (*.mdb)

.NET Framework Data Provider for ODBC classes are located in the System.Data.Odbc namespace.

The following code example shows how to include the `System.Data.Odbc` namespace in your applications.

```
Imports System.Data.Odbc
```

```
using System.Data.Odbc;
```

> **NOTE**
> The .NET Framework Data Provider for ODBC requires MDAC 2.6 or a later version, and MDAC 2.8 SP1 is recommended. You can download MDAC 2.8 SP1 from the Microsoft Download Center.

.NET Framework Data Provider for Oracle

The .NET Framework Data Provider for Oracle (OracleClient) enables data access to Oracle data sources through Oracle client connectivity software. The data provider supports Oracle client software version 8.1.7 or a later version. The data provider supports both local and distributed transactions. For more information, see Transactions and Concurrency.

The .NET Framework Data Provider for Oracle requires Oracle client software (version 8.1.7 or a later version) on

the system before you can connect to an Oracle data source.

.NET Framework Data Provider for Oracle classes are located in the System.Data.OracleClient namespace and are contained in the `System.Data.OracleClient.dll` assembly. You must reference both the `System.Data.dll` and the `System.Data.OracleClient.dll` when you compile an application that uses the data provider.

The following code example shows how to include the `System.Data.OracleClient` namespace in your applications.

```
Imports System.Data
Imports System.Data.OracleClient
```

```
using System.Data;
using System.Data.OracleClient;
```

Choosing a .NET Framework Data Provider

Depending on the design and data source for your application, your choice of .NET Framework data provider can improve the performance, capability, and integrity of your application. The following table discusses the advantages and limitations of each .NET Framework data provider.

PROVIDER	NOTES
.NET Framework Data Provider for SQL Server	Recommended for middle-tier applications that use Microsoft SQL Server. Recommended for single-tier applications that use Microsoft Database Engine (MSDE) or SQL Server. Recommended over use of the OLE DB provider for SQL Server (SQLOLEDB) with the .NET Framework Data Provider for OLE DB.
.NET Framework Data Provider for OLE DB	For SQL Server, the .NET Framework Data Provider for SQL Server is recommended instead of this provider. Recommended for single-tier applications that use Microsoft Access databases. Use of an Access database for a middle-tier application is not recommended.
.NET Framework Data Provider for ODBC	Recommended for middle and single-tier applications that use ODBC data sources.
.NET Framework Data Provider for Oracle	Recommended for middle and single-tier applications that use Oracle data sources.

EntityClient Provider

The EntityClient provider is used for accessing data based on an Entity Data Model (EDM). Unlike the other .NET Framework data providers, it does not interact directly with a data source. Instead, it uses Entity SQL to communicate with the underlying data provider. For more information, see EntityClient Provider for the Entity Framework.

See also

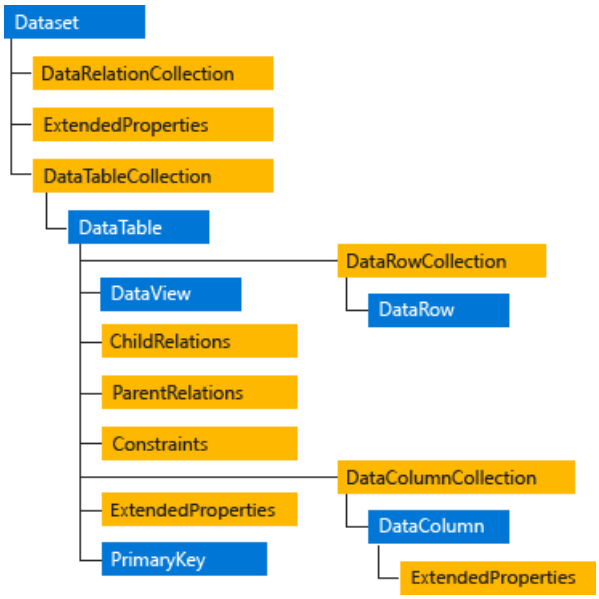
- ADO.NET Overview

- Retrieving and Modifying Data in ADO.NET

ADO.NET DataSets

2/4/2020 • 3 minutes to read • Edit Online

The DataSet object is central to supporting disconnected, distributed data scenarios with ADO.NET. The **DataSet** is a memory-resident representation of data that provides a consistent relational programming model regardless of the data source. It can be used with multiple and differing data sources, with XML data, or to manage data local to the application. The **DataSet** represents a complete set of data, including related tables, constraints, and relationships among the tables. The following illustration shows the **DataSet** object model.



DataSet Object Model

The methods and objects in a **DataSet** are consistent with those in the relational database model.

The **DataSet** can also persist and reload its contents as XML, and its schema as XML schema definition language (XSD) schema. For more information, see Using XML in a DataSet.

The DataTableCollection

An ADO.NET **DataSet** contains a collection of zero or more tables represented by DataTable objects. The DataTableCollection contains all the **DataTable** objects in a **DataSet**.

A **DataTable** is defined in the System.Data namespace and represents a single table of memory-resident data. It contains a collection of columns represented by a DataColumnCollection, and constraints represented by a ConstraintCollection, which together define the schema of the table. A **DataTable** also contains a collection of rows represented by the DataRowCollection, which contains the data in the table. Along with its current state, a DataRow retains both its current and original versions to identify changes to the values stored in the row.

The DataView Class

A DataView enables you to create different views of the data stored in a DataTable, a capability that is often used in data-binding applications. Using a DataView, you can expose the data in a table with different sort orders, and you can filter the data by row state or based on a filter expression. For more information, see DataViews.

The DataRelationCollection

A **DataSet** contains relationships in its DataRelationCollection object. A relationship, represented by the

DataRelation object, associates rows in one **DataTable** with rows in another **DataTable**. A relationship is analogous to a join path that might exist between primary and foreign key columns in a relational database. A **DataRelation** identifies matching columns in two tables of a **DataSet**.

Relationships enable navigation from one table to another in a **DataSet**. The essential elements of a **DataRelation** are the name of the relationship, the name of the tables being related, and the related columns in each table. Relationships can be built with more than one column per table by specifying an array of DataColumn objects as the key columns. When you add a relationship to the DataRelationCollection, you can optionally add a **UniqueKeyConstraint** and a **ForeignKeyConstraint** to enforce integrity constraints when changes are made to related column values.

For more information, see Adding DataRelations.

XML

You can fill a **DataSet** from an XML stream or document. You can use the XML stream or document to supply to the **DataSet** either data, schema information, or both. The information supplied from the XML stream or document can be combined with existing data or schema information already present in the **DataSet**. For more information, see Using XML in a DataSet.

ExtendedProperties

The **DataSet**, **DataTable**, and **DataColumn** all have an **ExtendedProperties** property. **ExtendedProperties** is a **PropertyCollection** where you can place custom information, such as the SELECT statement that was used to generate the result set, or the time when the data was generated. The **ExtendedProperties** collection is persisted with the schema information for the **DataSet**.

LINQ to DataSet

LINQ to DataSet provides language-integrated querying capabilities for disconnected data stored in a DataSet. LINQ to DataSet uses standard LINQ syntax and provides compile-time syntax checking, static typing, and IntelliSense support when you are using the Visual Studio IDE.

For more information, see LINQ to DataSet.

See also

- ADO.NET Overview
- DataSets, DataTables, and DataViews
- Retrieving and Modifying Data in ADO.NET

Side-by-Side Execution in ADO.NET

2/4/2020 • 4 minutes to read • Edit Online

Side-by-side execution in the .NET Framework is the ability to execute an application on a computer that has multiple versions of the .NET Framework installed, exclusively using the version for which the application was compiled. For detailed information about configuring side-by-side execution, see Side-by-Side Execution.

An application compiled by using one version of the .NET Framework can run on a different version of the .NET Framework. However, we recommend that you compile a version of the application for each installed version of the .NET Framework, and run them separately. In either scenario, you should be aware of changes in ADO.NET between releases that can affect the forward compatibility or backward compatibility of your application.

Forward Compatibility and Backward Compatibility

Forward compatibility means that an application can be compiled with an earlier version of the .NET Framework, but will still run successfully on a later version of the .NET Framework. ADO.NET code written for the .NET Framework version 1.1 is forward compatible with later versions.

Backward compatibility means that an application is compiled for a newer version of the .NET Framework, but continues to run on earlier versions of the .NET Framework without any loss of functionality. Of course, this will not be the case for features introduced in a new version of the .NET Framework.

The .NET Framework Data Provider for ODBC

Starting with version 1.1, the .NET Framework Data Provider for ODBC (System.Data.Odbc) is included as a part of the .NET Framework.

If you have an application developed for the .NET Framework version 1.0 that uses the ODBC data provider to connect to your data source, and you want to run that application on the .NET Framework version 1.1 or a later version, you must update the namespace for the ODBC data provider to **System.Data.Odbc**. You then must recompile it for the newer version of the .NET Framework.

If you have an application developed for the .NET Framework version 2.0 or later that uses the ODBC data provider to connect to your data source, and you want to run that application on the .NET Framework version 1.0, you must download the ODBC data provider and install it on the .NET Framework version 1.0 system. You then must change the namespace for the ODBC data provider to **Microsoft.Data.Odbc**, and recompile the application for the .NET Framework version 1.0.

The .NET Framework Data Provider for Oracle

Starting with version 1.1, the .NET Framework Data Provider for Oracle (System.Data.OracleClient) is included as a part of the .NET Framework.

If you have an application developed for the .NET Framework version 2.0 or later that uses the data provider to connect to your data source, and you want to run that application on the .NET Framework version 1.0, you must download the data provider and install it on the .NET Framework version 1.0 system.

Code Access Security

The .NET Framework data providers in the .NET Framework version 1.0 (System.Data.SqlClient, System.Data.OleDb) are required to run with FullTrust permission. Any attempt to use the .NET Framework k data providers from the

.NET Framework version 1.0 in a zone with less than FullTrust permission causes a SecurityException.

However, starting with the .NET Framework version 2.0, all of the .NET Framework data providers can be used in partially trusted zones. In addition, a new security feature was added to the .NET Framework data providers in the .NET Framework version 1.1. This feature enables you to restrict what connection strings can be used in a particular security zone. You can also disable the use of blank passwords for a particular security zone. For more information, see Code Access Security and ADO.NET.

Because each installation of the .NET Framework has a separate Security.config file, there are no compatibility issues with security settings. However, if your application depends on the additional security capabilities of ADO.NET included in the .NET Framework version 1.1 and later, you will not be able to distribute it to a version 1.0 system.

SqlCommand Execution

Starting with the .NET Framework version 1.1, the way that ExecuteReader executes commands at the data source was changed.

In the .NET Framework version 1.0, ExecuteReader executed all commands in the context of the **sp_executesql** stored procedure. As a result, commands that affect the state of the connection (for example, SET NOCOUNT ON), only apply to the execution of the current command. The state of the connection is not modified for any subsequent commands executed while the connection is open.

In the .NET Framework version 1.1 and later, ExecuteReader only executes a command in the context of the **sp_executesql** stored procedure if the command contains parameters, which provides a performance benefit. As a result, if a command affecting the state of the connection is included in a non-parameterized command, it modifies the state of the connection for all subsequent commands executed while the connection is open.

Consider the following batch of commands executed in a call to ExecuteReader.

```
SET NOCOUNT ON;
SELECT * FROM dbo.Customers;
```

In the .NET Framework version 1.1 and later, NOCOUNT will remain ON for any subsequent commands executed while the connection is open. In the .NET Framework version 1.0, NOCOUNT is only ON for the current command execution.

This change can affect both the forward and backward compatibility of your application if you depend on the behavior of ExecuteReader for either version of the .NET Framework.

For applications that run on both earlier and later versions of the .NET Framework, you can write your code to make sure that the behavior is the same regardless of the version you are running on. If you want to make sure that a command modifies the state of the connection for all subsequent commands, we recommend that you execute your command using ExecuteNonQuery. If you want to make sure that a command does not modify the connection for all subsequent commands, we recommend that you include the commands to reset the state of the connection in your command. For example:

```
SET NOCOUNT ON;
SELECT * FROM dbo.Customers;
SET NOCOUNT OFF;
```

See also

- ADO.NET Overview
- Retrieving and Modifying Data in ADO.NET

ADO.NET code examples

3/20/2020 • 12 minutes to read • Edit Online

The code listings on this page demonstrate how to retrieve data from a database by using the following ADO.NET technologies:

- ADO.NET data providers:
  - SqlClient (`System.Data.SqlClient`)
  - OleDb (`System.Data.OleDb`)
  - Odbc (`System.Data.Odbc`)
  - OracleClient (`System.Data.OracleClient`)
- ADO.NET Entity Framework:
  - LINQ to Entities
  - Typed ObjectQuery
  - EntityClient (`System.Data.EntityClient`)
- LINQ to SQL

ADO.NET data provider examples

The following code listings demonstrate how to retrieve data from a database using ADO.NET data providers. The data is returned in a `DataReader` . For more information, see Retrieving Data Using a DataReader.

SqlClient

The code in this example assumes that you can connect to the `Northwind` sample database on Microsoft SQL Server. The code creates a SqlCommand to select rows from the Products table, adding a SqlParameter to restrict the results to rows with a UnitPrice greater than the specified parameter value, in this case 5. The SqlConnection is opened inside a `using` block, which ensures that resources are closed and disposed when the code exits. The code executes the command by using a SqlDataReader, and displays the results in the console window.

```
using System;
using System.Data;
using System.Data.SqlClient;

class Program
{
    static void Main()
    {
        string connectionString =
            "Data Source=(local);Initial Catalog=Northwind;"
            + "Integrated Security=true";

        // Provide the query string with a parameter placeholder.
        string queryString =
            "SELECT ProductID, UnitPrice, ProductName from dbo.products "
                + "WHERE UnitPrice > @pricePoint "
                + "ORDER BY UnitPrice DESC;";

        // Specify the parameter value.
        int paramValue = 5;

        // Create and open the connection in a using block. This
        // ensures that all resources will be closed and disposed
        // when the code exits.
        using (SqlConnection connection =
            new SqlConnection(connectionString))
        {
            // Create the Command and Parameter objects.
            SqlCommand command = new SqlCommand(queryString, connection);
            command.Parameters.AddWithValue("@pricePoint", paramValue);

            // Open the connection in a try/catch block.
            // Create and execute the DataReader, writing the result
            // set to the console window.
            try
            {
                connection.Open();
                SqlDataReader reader = command.ExecuteReader();
                while (reader.Read())
                {
                    Console.WriteLine("\t{0}\t{1}\t{2}",
                        reader[0], reader[1], reader[2]);
                }
                reader.Close();
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
            Console.ReadLine();
        }
    }
```

```
Option Explicit On
Option Strict On

Imports System.Data
Imports System.Data.SqlClient

Public Class Program
    Public Shared Sub Main()

        Dim connectionString As String = _
            "Data Source=(local);Initial Catalog=Northwind;" _
            & "Integrated Security=true"

        ' Provide the query string with a parameter placeholder.
        Dim queryString As String = _
            "SELECT ProductID, UnitPrice, ProductName from dbo.Products " _
            & "WHERE UnitPrice > @pricePoint " _
            & "ORDER BY UnitPrice DESC;"

        ' Specify the parameter value.
        Dim paramValue As Integer = 5

        ' Create and open the connection in a using block. This
        ' ensures that all resources will be closed and disposed
        ' when the code exits.
        Using connection As New SqlConnection(connectionString)

            ' Create the Command and Parameter objects.
            Dim command As New SqlCommand(queryString, connection)
            command.Parameters.AddWithValue("@pricePoint", paramValue)

            ' Open the connection in a try/catch block.
            ' Create and execute the DataReader, writing the result
            ' set to the console window.
            Try
                connection.Open()
                Dim dataReader As SqlDataReader = _
                 command.ExecuteReader()
                Do While dataReader.Read()
                    Console.WriteLine( _
                        vbTab & "{0}" & vbTab & "{1}" & vbTab & "{2}", _
                     dataReader(0), dataReader(1), dataReader(2))
                Loop
                dataReader.Close()

            Catch ex As Exception
                Console.WriteLine(ex.Message)
            End Try
            Console.ReadLine()
        End Using
    End Sub
End Class
```

OleDb

The code in this example assumes that you can connect to the Microsoft Access Northwind sample database. The code creates a OleDbCommand to select rows from the Products table, adding a OleDbParameter to restrict the results to rows with a UnitPrice greater than the specified parameter value, in this case 5. The OleDbConnection is opened inside of a `using` block, which ensures that resources are closed and disposed when the code exits. The code executes the command by using a OleDbDataReader, and displays the results in the console window.

```
using System;
using System.Data;
using System.Data.OleDb;

class Program
{
    static void Main()
    {
        // The connection string assumes that the Access
        // Northwind.mdb is located in the c:\Data folder.
        string connectionString =
        "Provider=Microsoft.Jet.OLEDB.4.0;Data Source="
            + "c:\\Data\\Northwind.mdb;User Id=admin;Password=;";

        // Provide the query string with a parameter placeholder.
        string queryString =
            "SELECT ProductID, UnitPrice, ProductName from products "
                + "WHERE UnitPrice > ? "
                + "ORDER BY UnitPrice DESC;";

        // Specify the parameter value.
        int paramValue = 5;

        // Create and open the connection in a using block. This
        // ensures that all resources will be closed and disposed
        // when the code exits.
        using (OleDbConnection connection =
            new OleDbConnection(connectionString))
        {
            // Create the Command and Parameter objects.
            OleDbCommand command = new OleDbCommand(queryString, connection);
            command.Parameters.AddWithValue("@pricePoint", paramValue);

            // Open the connection in a try/catch block.
            // Create and execute the DataReader, writing the result
            // set to the console window.
            try
            {
                connection.Open();
                OleDbDataReader reader = command.ExecuteReader();
                while (reader.Read())
                {
                    Console.WriteLine("\t{0}\t{1}\t{2}",
                        reader[0], reader[1], reader[2]);
                }
                reader.Close();
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
            Console.ReadLine();
        }
    }
```

```
Option Explicit On
Option Strict On

Imports System.Data
Imports System.Data.OleDb

Public Class Program
    Public Shared Sub Main()

        ' The connection string assumes that the Access
        ' Northwind.mdb is located in the c:\Data folder.
        Dim connectionString As String = _
             "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=" _
            & "c:\Data\Northwind.mdb;User Id=admin;Password=;"

        ' Provide the query string with a parameter placeholder.
        Dim queryString As String = _
            "SELECT ProductID, UnitPrice, ProductName from Products " _
            & "WHERE UnitPrice > ? " _
            & "ORDER BY UnitPrice DESC;"

        ' Specify the parameter value.
        Dim paramValue As Integer = 5

        ' Create and open the connection in a using block. This
        ' ensures that all resources will be closed and disposed
        ' when the code exits.
        Using connection As New OleDbConnection(connectionString)

            ' Create the Command and Parameter objects.
            Dim command As New OleDbCommand(queryString, connection)
            command.Parameters.AddWithValue("@pricePoint", paramValue)

            ' Open the connection in a try/catch block.
            ' Create and execute the DataReader, writing the result
            ' set to the console window.
            Try
                connection.Open()
                Dim dataReader As OleDbDataReader = _
                 command.ExecuteReader()
                Do While dataReader.Read()
                    Console.WriteLine( _
                        vbTab & "{0}" & vbTab & "{1}" & vbTab & "{2}", _
                     dataReader(0), dataReader(1), dataReader(2))
                Loop
                dataReader.Close()

            Catch ex As Exception
                Console.WriteLine(ex.Message)
            End Try
            Console.ReadLine()
        End Using
    End Sub
End Class
```

Odbc

The code in this example assumes that you can connect to the Microsoft Access Northwind sample database. The code creates a OdbcCommand to select rows from the Products table, adding a OdbcParameter to restrict the results to rows with a UnitPrice greater than the specified parameter value, in this case 5. The OdbcConnection is opened inside a `using` block, which ensures that resources are closed and disposed when the code exits. The code executes the command by using a OdbcDataReader, and displays the results in the console window.

```
using System;
using System.Data;
using System.Data.Odbc;

class Program
{
    static void Main()
    {
        // The connection string assumes that the Access
        // Northwind.mdb is located in the c:\Data folder.
        string connectionString =
            "Driver={Microsoft Access Driver (*.mdb)};"
            + "Dbq=c:\\Data\\Northwind.mdb;Uid=Admin;Pwd=;";

        // Provide the query string with a parameter placeholder.
        string queryString =
            "SELECT ProductID, UnitPrice, ProductName from products "
                + "WHERE UnitPrice > ? "
                + "ORDER BY UnitPrice DESC;";

        // Specify the parameter value.
        int paramValue = 5;

        // Create and open the connection in a using block. This
        // ensures that all resources will be closed and disposed
        // when the code exits.
        using (OdbcConnection connection =
            new OdbcConnection(connectionString))
        {
            // Create the Command and Parameter objects.
            OdbcCommand command = new OdbcCommand(queryString, connection);
            command.Parameters.AddWithValue("@pricePoint", paramValue);

            // Open the connection in a try/catch block.
            // Create and execute the DataReader, writing the result
            // set to the console window.
            try
            {
                connection.Open();
                OdbcDataReader reader = command.ExecuteReader();
                while (reader.Read())
                {
                    Console.WriteLine("\t{0}\t{1}\t{2}",
                        reader[0], reader[1], reader[2]);
                }
                reader.Close();
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
            Console.ReadLine();
        }
    }
}
```

```
Option Explicit On
Option Strict On

Imports System.Data
Imports System.Data.Odbc

Public Class Program
    Public Shared Sub Main()

        ' The connection string assumes that the Access
        ' Northwind.mdb is located in the c:\Data folder.
        Dim connectionString As String = _
            "Driver={Microsoft Access Driver (*.mdb)};" _
           & "Dbq=c:\Data\Northwind.mdb;Uid=Admin;Pwd=;"

        ' Provide the query string with a parameter placeholder.
        Dim queryString As String = _
            "SELECT ProductID, UnitPrice, ProductName from Products " _
            & "WHERE UnitPrice > ? " _
            & "ORDER BY UnitPrice DESC;"

        ' Specify the parameter value.
        Dim paramValue As Integer = 5

        ' Create and open the connection in a using block. This
        ' ensures that all resources will be closed and disposed
        ' when the code exits.
        Using connection As New OdbcConnection(connectionString)

            ' Create the Command and Parameter objects.
            Dim command As New OdbcCommand(queryString, connection)
            command.Parameters.AddWithValue("@pricePoint", paramValue)

            ' Open the connection in a try/catch block.
            ' Create and execute the DataReader, writing the result
            ' set to the console window.
            Try
                connection.Open()
                Dim dataReader As OdbcDataReader = _
                 command.ExecuteReader()
                Do While dataReader.Read()
                    Console.WriteLine( _
                        vbTab & "{0}" & vbTab & "{1}" & vbTab & "{2}", _
                     dataReader(0), dataReader(1), dataReader(2))
                Loop
                dataReader.Close()

            Catch ex As Exception
                Console.WriteLine(ex.Message)
            End Try
            Console.ReadLine()
        End Using
    End Sub
End Class
```

OracleClient

The code in this example assumes a connection to DEMO.CUSTOMER on an Oracle server. You must also add a reference to the System.Data.OracleClient.dll. The code returns the data in an OracleDataReader.

```
using System;
using System.Data;
using System.Data.OracleClient;

class Program
{
    static void Main()
    {
        string connectionString =
            "Data Source=ThisOracleServer;Integrated Security=yes;";
        string queryString =
            "SELECT CUSTOMER_ID, NAME FROM DEMO.CUSTOMER";
        using (OracleConnection connection =
                    new OracleConnection(connectionString))
        {
            OracleCommand command = connection.CreateCommand();
            command.CommandText = queryString;

            try
            {
                connection.Open();

                OracleDataReader reader = command.ExecuteReader();

                while (reader.Read())
                {
                    Console.WriteLine("\t{0}\t{1}",
                        reader[0], reader[1]);
                }
                reader.Close();
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
        }
    }
}
```

```
Option Explicit On
Option Strict On

Imports System.Data
Imports System.Data.OracleClient

Public Class Program
    Public Shared Sub Main()

        Dim connectionString As String = _
            "Data Source=ThisOracleServer;Integrated Security=yes;"

        Dim queryString As String = _
            "SELECT CUSTOMER_ID, NAME FROM DEMO.CUSTOMER"

        Using connection As New OracleConnection(connectionString)
            Dim command As OracleCommand = connection.CreateCommand()
            command.CommandText = queryString
            Try
                connection.Open()
                Dim dataReader As OracleDataReader = _
                 command.ExecuteReader()
                Do While dataReader.Read()
                    Console.WriteLine(vbTab & "{0}" & vbTab & "{1}", _
                     dataReader(0), dataReader(1))
                Loop
                dataReader.Close()

            Catch ex As Exception
                Console.WriteLine(ex.Message)
            End Try
        End Using
    End Sub
End Class
```

Entity Framework examples

The following code listings demonstrate how to retrieve data from a data source by querying entities in an Entity Data Model (EDM). These examples use a model based on the Northwind sample database. For more information about Entity Framework, see Entity Framework Overview.

LINQ to Entities

The code in this example uses a LINQ query to return data as Categories objects, which are projected as an anonymous type that contains only the CategoryID and CategoryName properties. For more information, see LINQ to Entities Overview.

```
using System;
using System.Linq;
using System.Data.Objects;
using NorthwindModel;

class LinqSample
{
    public static void ExecuteQuery()
    {
        using (NorthwindEntities context = new NorthwindEntities())
        {
            try
            {
                var query = from category in context.Categories
                            select new
                            {
                                categoryID = category.CategoryID,
                                categoryName = category.CategoryName
                            };

                foreach (var categoryInfo in query)
                {
                    Console.WriteLine("\t{0}\t{1}",
                        categoryInfo.categoryID, categoryInfo.categoryName);
                }
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
        }
    }
}
```

```
Option Explicit On
Option Strict On

Imports System.Linq
Imports System.Data.Objects
Imports NorthwindModel

Class LinqSample
    Public Shared Sub ExecuteQuery()
        Using context As NorthwindEntities = New NorthwindEntities()
            Try
                Dim query = From category In context.Categories _
                            Select New With _
                            { _
                                .categoryID = category.CategoryID, _
                                .categoryName = category.CategoryName _
                            }

                For Each categoryInfo In query
                    Console.WriteLine(vbTab & "{0}" & vbTab & "{1}", _
                        categoryInfo.categoryID, categoryInfo.categoryName)
                Next
            Catch ex As Exception
                Console.WriteLine(ex.Message)
            End Try
        End Using
    End Sub
End Class
```

Typed ObjectQuery

The code in this example uses an ObjectQuery<T> to return data as Categories objects. For more information, see Object Queries.

```
using System;
using System.Data.Objects;
using NorthwindModel;

class ObjectQuerySample
{
    public static void ExecuteQuery()
    {
        using (NorthwindEntities context = new NorthwindEntities())
        {
            ObjectQuery<Categories> categoryQuery = context.Categories;

            foreach (Categories category in
                categoryQuery.Execute(MergeOption.AppendOnly))
            {
                Console.WriteLine("\t{0}\t{1}",
                    category.CategoryID, category.CategoryName);
            }
        }
    }
}
```

```
Option Explicit On
Option Strict On

Imports System.Data.Objects
Imports NorthwindModel

Class ObjectQuerySample
    Public Shared Sub ExecuteQuery()
        Using context As NorthwindEntities = New NorthwindEntities()
            Dim categoryQuery As ObjectQuery(Of Categories) = context.Categories

            For Each category As Categories In _
                categoryQuery.Execute(MergeOption.AppendOnly)
                Console.WriteLine(vbTab & "{0}" & vbTab & "{1}", _
                    category.CategoryID, category.CategoryName)
            Next
        End Using
    End Sub
End Class
```

EntityClient

The code in this example uses an EntityCommand to execute an Entity SQL query. This query returns a list of records that represent instances of the Categories entity type. An EntityDataReader is used to access data records in the result set. For more information, see EntityClient Provider for the Entity Framework.

```
using System;
using System.Data;
using System.Data.Common;
using System.Data.EntityClient;
using NorthwindModel;

class EntityClientSample
{
    public static void ExecuteQuery()
    {
        string queryString =
            @"SELECT c.CategoryID, c.CategoryName
                FROM NorthwindEntities.Categories AS c";

        using (EntityConnection conn =
            new EntityConnection("name=NorthwindEntities"))
        {
            try
            {
                conn.Open();
                using (EntityCommand query = new EntityCommand(queryString, conn))
                {
                    using (DbDataReader rdr =
                        query.ExecuteReader(CommandBehavior.SequentialAccess))
                    {
                        while (rdr.Read())
                        {
                            Console.WriteLine("\t{0}\t{1}", rdr[0], rdr[1]);
                        }
                    }
                }
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
        }
    }
}
```

```
Option Explicit On
Option Strict On

Imports System.Data
Imports System.Data.Common
Imports System.Data.EntityClient
Imports NorthwindModel

Class EntityClientSample
    Public Shared Sub ExecuteQuery()
        Dim queryString As String = _
            "SELECT c.CategoryID, c.CategoryName " & _
                "FROM NorthwindEntities.Categories AS c"

        Using conn As EntityConnection = _
            New EntityConnection("name=NorthwindEntities")

            Try
                conn.Open()
                Using query As EntityCommand = _
                New EntityCommand(queryString, conn)
                    Using rdr As DbDataReader = _
                        query.ExecuteReader(CommandBehavior.SequentialAccess)
                        While rdr.Read()
                            Console.WriteLine(vbTab & "{0}" & vbTab & "{1}", _
                                              rdr(0), rdr(1))
                        End While
                    End Using
                End Using
            Catch ex As Exception
                Console.WriteLine(ex.Message)
            End Try
        End Using
    End Sub
End Class
```

LINQ to SQL

The code in this example uses a LINQ query to return data as Categories objects, which are projected as an anonymous type that contains only the CategoryID and CategoryName properties. This example is based on the Northwind data context. For more information, see Getting Started.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Northwind;

    class LinqSqlSample
    {
        public static void ExecuteQuery()
        {
            using (NorthwindDataContext db = new NorthwindDataContext())
            {
                try
                {
                    var query = from category in db.Categories
                                select new
                                {
                                    categoryID = category.CategoryID,
                                    categoryName = category.CategoryName
                                };

                    foreach (var categoryInfo in query)
                    {
                        Console.WriteLine("vbTab {0} vbTab {1}",
                            categoryInfo.categoryID, categoryInfo.categoryName);
                    }
                }
                catch (Exception ex)
                {
                    Console.WriteLine(ex.Message);
                }
            }
        }
    }
```

```
Option Explicit On
Option Strict On

Imports System.Collections.Generic
Imports System.Linq
Imports System.Text
Imports Northwind

Class LinqSqlSample
    Public Shared Sub ExecuteQuery()
        Using db As NorthwindDataContext = New NorthwindDataContext()
            Try
                Dim query = From category In db.Categories _
                            Select New With _
                            { _
                                .categoryID = category.CategoryID, _
                                .categoryName = category.CategoryName _
                            }

                For Each categoryInfo In query
                    Console.WriteLine(vbTab & "{0}" & vbTab & "{1}", _
                            categoryInfo.categoryID, categoryInfo.categoryName)
                Next
            Catch ex As Exception
                Console.WriteLine(ex.Message)
            End Try
            End Using
    End Sub
End Class
```

See also

- ADO.NET Overview
- Retrieving and Modifying Data in ADO.NET
- Creating Data Applications
- Querying an Entity Data Model (Entity Framework Tasks)
- How to: Execute a Query that Returns Anonymous Type Objects

Securing ADO.NET Applications

2/4/2020 • 2 minutes to read • Edit Online

Writing a secure ADO.NET application involves more than avoiding common coding pitfalls such as not validating user input. An application that accesses data has many potential points of failure that an attacker can exploit to retrieve, manipulate, or destroy sensitive data. It is therefore important to understand all aspects of security, from the process of threat modeling during the design phase of your application, to its eventual deployment and ongoing maintenance.

The .NET Framework provides many useful classes, services, and tools for securing and administering database applications. The common language runtime (CLR) provides a type-safe environment for code to run in, with code access security (CAS) to restrict further the permissions of managed code. Following secure data access coding practices limits the damage that can be inflicted by a potential attacker.

Writing secure code does not guard against self-inflicted security holes when working with unmanaged resources such as databases. Most server databases, such as SQL Server, have their own security systems, which enhance security when implemented correctly. However, even a data source with a robust security system can be victimized in an attack if it is not configured appropriately.

In This Section

Security Overview
Provides recommendations for designing secure ADO.NET applications.

Secure Data Access
Describes how to work with data from a secured data source.

Secure Client Applications
Describes security considerations for client applications.

Code Access Security and ADO.NET
Describes how CAS can help protect ADO.NET code. Also discusses how to work with partial trust.

Privacy and Data Security
Describes encryption options for ADO.NET applications.

Related Sections

SQL Server Security
Describes SQL Server security features from a developer's perspective.

Security Considerations
Describes security for Entity Framework applications.

Security
Contains links to topics describing all aspects of security in .NET.

Security Tools
.NET Framework tools for securing and administering security policy.

Resources for Creating Secure Applications
Provides links to topics for creating secure applications.

Security Bibliography

Provides links to external resources available online and in print.

See also

- ADO.NET
- ADO.NET Overview

Security overview

12/29/2019 • 5 minutes to read • Edit Online

Securing an application is an ongoing process. There will never be a point where a developer can guarantee that an application is safe from all attacks, because it is impossible to predict what kinds of future attacks new technologies will bring about. Conversely, just because nobody has yet discovered (or published) security flaws in a system does not mean that none exist or could exist. You need to plan for security during the design phase of the project, as well as plan how security will be maintained over the lifetime of the application.

Design for Security

One of the biggest problems in developing secure applications is that security is often an afterthought, something to implement after a project is code-complete. Not building security into an application at the outset leads to insecure applications because little thought has been given to what makes an application secure.

Last-minute security implementation leads to more bugs, as software breaks under the new restrictions or has to be rewritten to accommodate unanticipated functionality. Every line of revised code contains the possibility of introducing a new bug. For this reason, you should consider security early in the development process so that it can proceed in tandem with the development of new features.

Threat Modeling

You cannot protect a system against attack unless you understand all the potential attacks that it is exposed to. The process of evaluating security threats, called *threat modeling*, is necessary to determine the likelihood and ramifications of security breaches in your ADO.NET application.

Threat modeling is composed of three high-level steps: understanding the adversary's view, characterizing the security of the system, and determining threats.

Threat modeling is an iterative approach to assessing vulnerabilities in your application to find those that are the most dangerous because they expose the most sensitive data. Once you identify the vulnerabilities, you rank them in order of severity and create a prioritized set of countermeasures to counter the threats.

For more information, see the following resources:

RESOURCE	DESCRIPTION
The Threat Modeling site on the Security Engineering Portal	The resources on this page will help you understand the threat modeling process and build threat models that you can use to secure your own applications

The Principle of Least Privilege

When you design, build, and deploy your application, you must assume that your application will be attacked. Often these attacks come from malicious code that executes with the permissions of the user running the code. Others can originate with well-intentioned code that has been exploited by an attacker. When planning security, always assume the worst-case scenario will occur.

One counter-measure you can employ is to try to erect as many walls around your code as possible by running with least privilege. The principle of least privilege says that any given privilege should be granted to the least amount of code necessary for the shortest duration of time that is required to get the job done.

The best practice for creating secure applications is to start with no permissions at all and then add the narrowest

permissions for the particular task being performed. By contrast, starting with all permissions and then denying individual ones leads to insecure applications that are difficult to test and maintain because security holes may exist from unintentionally granting more permissions than required.

For more information on securing your applications, see the following resources:

RESOURCE	DESCRIPTION
Securing Applications	Contains links to general security topics. Also contains links to topics for securing distributed applications, Web applications, mobile applications, and desktop applications.

Code Access Security (CAS)

Code access security (CAS) is a mechanism that helps limit the access that code has to protected resources and operations. In the .NET Framework, CAS performs the following functions:

- Defines permissions and permission sets that represent the right to access various system resources.
- Enables administrators to configure security policy by associating sets of permissions with groups of code (code groups).
- Enables code to request the permissions it requires in order to run, as well as the permissions that would be useful to have, and specifies which permissions the code must never have.
- Grants permissions to each assembly that is loaded, based on the permissions requested by the code and on the operations permitted by security policy.
- Enables code to demand that its callers have specific permissions.
- Enables code to demand that its callers possess a digital signature, thus allowing only callers from a particular organization or site to call the protected code.
- Enforces restrictions on code at run time by comparing the granted permissions of every caller on the call stack to the permissions that callers must have.

To minimize the amount of damage that can occur if an attack succeeds, choose a security context for your code that grants access only to the resources it needs to get its work done and no more.

For more information, see the following resources:

RESOURCE	DESCRIPTION
Code Access Security and ADO.NET	Describes the interactions between code access security, role-based security, and partially trusted environments from the perspective of an ADO.NET application.
Code Access Security	Contains links to additional topics describing CAS in the .NET Framework.

Database Security

The principle of least privilege also applies to your data source. Some general guidelines for database security include:

- Create accounts with the lowest possible privileges.
- Do not allow users access to administrative accounts just to get code working.

- Do not return server-side error messages to client applications.
- Validate all input at both the client and the server.
- Use parameterized commands and avoid dynamic SQL statements.
- Enable security auditing and logging for the database you are using so that you are alerted to any security breaches.

For more information, see the following resources:

RESOURCE	DESCRIPTION
SQL Server Security	Provides an overview of SQL Server security with application scenarios that provide guidance for creating secure ADO.NET applications that target SQL Server.
Recommendations for Data Access Strategies	Provides recommendations for accessing data and performing database operations.

Security Policy and Administration

Improperly administering code access security (CAS) policy can potentially create security weaknesses. Once an application is deployed, techniques for monitoring security should be used and risks evaluated as new threats emerge.

For more information, see the following resources:

RESOURCE	DESCRIPTION
Security Policy Management	Provides information on creating and administering security policy.
Security Policy Best Practices	Provides links describing how to administer security policy.

See also

- Securing ADO.NET Applications
- Security in .NET
- SQL Server Security
- ADO.NET Overview

Secure Data Access

12/24/2019 • 4 minutes to read • Edit Online

To write secure ADO.NET code, you have to understand the security mechanisms available in the underlying data store, or database. You also need to consider the security implications of other features or components that your application may contain.

Authentication, Authorization and Permissions

When connecting to Microsoft SQL Server, you can use Windows Authentication, also known as Integrated Security, which uses the identity of the current active Windows user rather than passing a user ID and password. Using Windows Authentication is highly recommended because user credentials are not exposed in the connection string. If you cannot use Windows Authentication to connect to SQL Server, then consider creating connection strings at run time using the SqlConnectionStringBuilder.

The credentials used for authentication need to be handled differently based on the type of application. For example, in a Windows Forms application, the user can be prompted to supply authentication information, or the user's Windows credentials can be used. However, a Web application often accesses data using credentials supplied by the application itself rather than by the user.

Once users have been authenticated, the scope of their actions depends on the permissions that have been granted to them. Always follow the principle of least privilege and grant only permissions that are absolutely necessary.

For more information, see the following resources.

RESOURCE	DESCRIPTION
Protecting Connection Information	Describes security best practices and techniques for protecting connection information, such as using protected configuration to encrypt connection strings.
Recommendations for Data Access Strategies	Provides recommendations for accessing data and performing database operations.
Connection String Builders	Describes how to build connection strings from user input at run time.
Overview of SQL Server Security	Describes the SQL Server security architecture.

Parameterized Commands and SQL Injection

Using parameterized commands helps guard against SQL injection attacks, in which an attacker "injects" a command into a SQL statement that compromises security on the server. Parameterized commands guard against a SQL injection attack by ensuring that values received from an external source are passed as values only, and not part of the Transact-SQL statement. As a result, Transact-SQL commands inserted into a value are not executed at the data source. Rather, they are evaluated solely as a parameter value. In addition to the security benefits, parameterized commands provide a convenient method for organizing values passed with a Transact-SQL statement or to a stored procedure.

For more information on using parameterized commands, see the following resources.

RESOURCE	DESCRIPTION
DataAdapter Parameters	Describes how to use parameters with a `DataAdapter`.
Modifying Data with Stored Procedures	Describes how to specify parameters and obtain a return value.
Managing Permissions with Stored Procedures in SQL Server	Describes how to use SQL Server stored procedures to encapsulate data access.

Script Exploits

A script exploit is another form of injection that uses malicious characters inserted into a Web page. The browser does not validate the inserted characters and will process them as part of the page.

For more information, see the following resources.

RESOURCE	DESCRIPTION
Script Exploits Overview	Describes how to guard against scripting and SQL statement exploits.

Probing Attacks

Attackers often use information from an exception, such as the name of your server, database, or table, to mount an attack on your system. Because exceptions can contain specific information about your application or data source, you can help keep your application and data source better protected by only exposing essential information to the client.

For more information, see the following resources.

RESOURCE	DESCRIPTION
Handling and throwing exceptions in .NET	Describes the basic forms of try/catch/finally structured exception handling.
Best Practices for Exceptions	Describes best practices for handling exceptions.

Protecting Microsoft Access and Excel Data Sources

Microsoft Access and Microsoft Excel can act as a data store for an ADO.NET application when security requirements are minimal or nonexistent. Their security features are effective for deterrence, but should not be relied upon to do more than discourage meddling by uninformed users. The physical data files for Access and Excel exist on the file system, and must be accessible to all users. This makes them vulnerable to attacks that could result in theft or data loss since the files can be easily copied or altered. When robust security is required, use SQL Server or another server-based database where the physical data files are not readable from the file system.

For more information on protecting Access and Excel data, see the following resources.

RESOURCE	DESCRIPTION

RESOURCE	DESCRIPTION
Security Considerations and Guidance for Access 2007	Describes security techniques for Access 2007 such encrypting files, administering passwords, converting databases to the new ACCDB and ACCDE formats, and using other security options.
Introduction to Access 2010 security	Provides an overview of the security features offered by Access 2010.

Enterprise Services

COM+ contains its own security model that relies on Windows NT accounts and process/thread impersonation. The System.EnterpriseServices namespace provides wrappers that allow .NET applications to integrate managed code with COM+ security services through the ServicedComponent class.

For more information, see the following resource.

RESOURCE	DESCRIPTION
Role-Based Security	Discusses how to integrate managed code with COM+ security services.

Interoperating with Unmanaged Code

The .NET Framework provides for interaction with unmanaged code, including COM components, COM+ services, external type libraries, and many operating system services. Working with unmanaged code involves going outside the security perimeter for managed code. Both your code and any code that calls it must have unmanaged code permission (SecurityPermission with the UnmanagedCode flag specified). Unmanaged code can introduce unintended security vulnerabilities into your application. Therefore, you should avoid interoperating with unmanaged code unless it is absolutely necessary.

For more information, see the following resources.

RESOURCE	DESCRIPTION
Interoperating with Unmanaged Code	Contains topics describing how to expose COM components to the .NET Framework and how to expose .NET Framework components to COM.
Advanced COM Interoperability	Contains advanced topics such as primary interop assemblies, threading and custom marshaling.

See also

- Securing ADO.NET Applications
- SQL Server Security
- Recommendations for Data Access Strategies
- Protecting Connection Information
- Connection String Builders
- ADO.NET Overview

Secure Client Applications

9/7/2019 • 3 minutes to read • Edit Online

Applications typically consist of many parts that must all be protected from vulnerabilities that could result in data loss or otherwise compromise the system. Creating secure user interfaces can prevent many problems by blocking attackers before they can access data or system resources.

Validate User Input

When constructing an application that accesses data, you should assume that all user input is malicious until proven otherwise. Failure to do so can leave your application vulnerable to attack. The .NET Framework contains classes to help you enforce a domain of values for input controls, such as limiting the number of characters that can be entered. Event hooks allow you to write procedures to check the validity of values. User input data can be validated and strongly typed, limiting an application's exposure to script and SQL injection exploits.

> **IMPORTANT**
> You must also validate user input at the data source as well as in the client application. An attacker may choose to circumvent your application and attack the data source directly.

Security and User Input
Describes how to handle subtle and potentially dangerous bugs involving user input.

Validating User Input in ASP.NET Web Pages
Overview of validating user input using ASP.NET validation controls.

User Input in Windows Forms
Provides links and information for validating mouse and keyboard input in a Windows Forms application.

.NET Framework Regular Expressions
Describes how to use the Regex class to check the validity of user input.

Windows Applications

In the past, Windows applications generally ran with full permissions. The .NET Framework provides the infrastructure to restrict code executing in a Windows application by using code access security (CAS). However, CAS alone is not enough to protect your application.

Windows Forms Security
Discusses how to secure Windows Forms applications and provides links to related topics.

Windows Forms and Unmanaged Applications
Describes how to interact with unmanaged applications in a Windows Forms application.

ClickOnce Deployment for Windows Forms
Describes how to use `ClickOnce` deployment in a Windows Forms application and discusses the security implications.

ASP.NET and XML Web Services

ASP.NET applications generally need to restrict access to some portions of the Web site and provide other mechanisms for data protection and site security. These links provide useful information for securing your ASP.NET

application.

An XML Web service provides data that can be consumed by an ASP.NET application, a Windows Forms application, or another Web service. You need to manage security for the Web service itself as well as security for the client application.

For more information, see the following resources.

RESOURCE	DESCRIPTION
Securing ASP.NET Web Sites	Discusses how to secure ASP.NET applications.
Securing XML Web Services Created Using ASP.NET	Discusses how to implement security for an ASP.NET Web Service.
Script Exploits Overview	Discusses how to guard against a script exploit attack, which attempts to insert malicious characters into a Web page.
Basic Security Practices for Web Applications	General security information and links to further discussion,

Remoting

.NET remoting enables you to build widely distributed applications easily, whether the application components are all on one computer or spread out across the entire world. You can build client applications that use objects in other processes on the same computer or on any other computer that is reachable over its network. You can also use .NET remoting to communicate with other application domains in the same process.

RESOURCE	DESCRIPTION
Configuration of Remote Applications	Discusses how to configure remoting applications in order to avoid common problems.
Security in Remoting	Describes authentication and encryption as well as additional security topics relevant to remoting.
Security and Remoting Considerations	Describes security issues with protected objects and application domain crossing.

See also

- Securing ADO.NET Applications
- Recommendations for Data Access Strategies
- Securing Applications
- Protecting Connection Information
- ADO.NET Overview

Code Access Security and ADO.NET

4/20/2020 • 13 minutes to read • Edit Online

The .NET Framework offers role-based security as well as code access security (CAS), both of which are implemented using a common infrastructure supplied by the common language runtime (CLR). In the world of unmanaged code, most applications execute with the permissions of the user or principal. As a result, computer systems can be damaged and private data compromised when malicious or error-filled software is run by a user with elevated privileges.

By contrast, managed code executing in the .NET Framework includes code access security, which applies to code alone. Whether the code is allowed to run or not depends on the code's origin or other aspects of the code's identity, not solely the identity of the principal. This reduces the likelihood that managed code can be misused.

Code Access Permissions

When code is executed, it presents evidence that is evaluated by the CLR security system. Typically, this evidence comprises the origin of the code including URL, site, and zone, and digital signatures that ensure the identity of the assembly.

The CLR allows code to perform only those operations that the code has permission to perform. Code can request permissions, and those requests are honored based on the security policy set by an administrator.

> **NOTE**
>
> Code executing in the CLR cannot grant permissions to itself. For example, code can request and be granted fewer permissions than a security policy allows, but it will never be granted more permissions. When granting permissions, start with no permissions at all and then add the narrowest permissions for the particular task being performed. Starting with all permissions and then denying individual ones leads to insecure applications that may contain unintentional security holes from granting more permissions than required. For more information, see Configuring Security Policy and Security Policy Management.

There are three types of code access permissions:

- `Code access permissions` derive from the CodeAccessPermission class. Permissions are required in order to access protected resources, such as files and environment variables, and to perform protected operations, such as accessing unmanaged code.
- `Identity permissions` represent characteristics that identify an assembly. Permissions are granted to an assembly based on evidence, which can include items such as a digital signature or where the code originated. Identity permissions also derive from the CodeAccessPermission base class.
- `Role-based security permissions` are based on whether a principal has a specified identity or is a member of a specified role. The PrincipalPermission class allows both declarative and imperative permission checks against the active principal.

To determine whether code is authorized to access a resource or perform an operation, the runtime's security system traverses the call stack, comparing the granted permissions of each caller to the permission being demanded. If any caller in the call stack does not have the demanded permission, a SecurityException is thrown and access is refused.

Requesting Permissions

The purpose of requesting permissions is to inform the runtime which permissions your application requires in

order to run, and to ensure that it receives only the permissions that it actually needs. For example, if your application needs to write data to the local disk, it requires FileIOPermission. If that permission hasn't been granted, the application will fail when it attempts to write to the disk. However, if the application requests `FileIOPermission` and that permission has not been granted, the application will generate the exception at the outset and will not load.

In a scenario where the application only needs to read data from the disk, you can request that it never be granted any write permissions. In the event of a bug or a malicious attack, your code cannot damage the data on which it operates. For more information, see Requesting Permissions.

Role-Based Security and CAS

Implementing both role-based security and code-accessed security (CAS) enhances overall security for your application. Role-based security can be based on a Windows account or a custom identity, making information about the security principal available to the current thread. In addition, applications are often required to provide access to data or resources based on credentials supplied by the user. Typically, such applications check the role of a user and provide access to resources based on those roles.

Role-based security enables a component to identify current users and their associated roles at run time. This information is then mapped using a CAS policy to determine the set of permissions granted at run time. For a specified application domain, the host can change the default role-based security policy and set a default security principal that represents a user and the roles associated with that user.

The CLR uses permissions to implement its mechanism for enforcing restrictions on managed code. Role-based security permissions provide a mechanism for discovering whether a user (or the agent acting on the user's behalf) has a particular identity or is a member of a specified role. For more information, see Security Permissions.

Depending on the type of application you are building, you should also consider implementing role-based permissions in the database. For more information on role-based security in SQL Server, see SQL Server Security.

Assemblies

Assemblies form the fundamental unit of deployment, version control, reuse, activation scoping, and security permissions for a .NET Framework application. An assembly provides a collection of types and resources that are built to work together and form a logical unit of functionality. To the CLR, a type does not exist outside the context of an assembly. For more information on creating and deploying assemblies, see Programming with Assemblies.

Strong-naming Assemblies

A strong name, or digital signature, consists of the assembly's identity, which includes its simple text name, version number, and culture information (if provided), plus a public key and a digital signature. The digital signature is generated from an assembly file using the corresponding private key. The assembly file contains the assembly manifest, which contains the names and hashes of all the files that make up the assembly.

Strong naming an assembly gives an application or component a unique identity that other software can use to refer explicitly to it. Strong naming guards assemblies against being spoofed by an assembly that contains hostile code. Strong-naming also ensures versioning consistency among different versions of a component. You must strong name assemblies that will be deployed to the Global Assembly Cache (GAC). For more information, see Creating and Using Strong-Named Assemblies.

Partial Trust in ADO.NET 2.0

In ADO.NET 2.0, the .NET Framework Data Provider for SQL Server, the .NET Framework Data Provider for OLE DB, the .NET Framework Data Provider for ODBC, and the .NET Framework Data Provider for Oracle can now all run in partially trusted environments. In previous releases of the .NET Framework, only System.Data.SqlClient was supported in less than full-trust applications.

At minimum, a partially trusted application using the SQL Server provider must have execution and SqlClientPermission permissions.

Permission Attribute Properties for Partial Trust

For partial trust scenarios, you can use SqlClientPermissionAttribute members to further restrict the capabilities available for the .NET Framework Data Provider for SQL Server.

The following table lists the available SqlClientPermissionAttribute properties and their descriptions:

PERMISSION ATTRIBUTE PROPERTY	DESCRIPTION
`Action`	Gets or sets a security action. Inherited from SecurityAttribute.
`AllowBlankPassword`	Enables or disables the use of a blank password in a connection string. Valid values are `true` (to enable the use of blank passwords) and `false` (to disable the use of blank passwords). Inherited from DBDataPermissionAttribute.
`ConnectionString`	Identifies a permitted connection string. Multiple connection strings can be identified. **Note:** Do not include a user ID or password in your connection string. In this release, you cannot change connection string restrictions using the .NET Framework Configuration Tool. Inherited from DBDataPermissionAttribute.
`KeyRestrictions`	Identifies connection string parameters that are allowed or disallowed. Connection string parameters are identified in the form *<parameter name>=*. Multiple parameters can be specified, delimited using a semicolon (;). **Note:** If you do not specify `KeyRestrictions`, but you set `KeyRestrictionBehavior` property to `AllowOnly` or `PreventUsage`, no additional connection string parameters are allowed. Inherited from DBDataPermissionAttribute.
`KeyRestrictionBehavior`	Identifies the connection string parameters as the only additional parameters allowed (`AllowOnly`), or identifies the additional parameters that are not allowed (`PreventUsage`). `AllowOnly` is the default. Inherited from DBDataPermissionAttribute.
`TypeID`	Gets a unique identifier for this attribute when implemented in a derived class. Inherited from Attribute.
`Unrestricted`	Indicates whether unrestricted permission to the resource is declared. Inherited from SecurityAttribute.

ConnectionString Syntax

The following example demonstrates how to use the `connectionStrings` element of a configuration file to allow only a specific connection string to be used. See Connection Strings for more information on storing and retrieving connection strings from configuration files.

```
<connectionStrings>
  <add name="DatabaseConnection"
    connectionString="Data Source=(local);Initial
    Catalog=Northwind;Integrated Security=true;" />
</connectionStrings>
```

KeyRestrictions Syntax

The following example enables the same connection string, enables the use of the `Encrypt` and `Packet Size` connection string options, but restricts the use of any other connection string options.

```
<connectionStrings>
  <add name="DatabaseConnection"
    connectionString="Data Source=(local);Initial
    Catalog=Northwind;Integrated Security=true;"
    KeyRestrictions="Encrypt=;Packet Size=;"
    KeyRestrictionBehavior="AllowOnly" />
</connectionStrings>
```

KeyRestrictionBehavior with PreventUsage Syntax

The following example enables the same connection string and allows all other connection parameters except for `User Id`, `Password` and `Persist Security Info`.

```
<connectionStrings>
  <add name="DatabaseConnection"
    connectionString="Data Source=(local);Initial
    Catalog=Northwind;Integrated Security=true;"
    KeyRestrictions="User Id=;Password=;Persist Security Info=;"
    KeyRestrictionBehavior="PreventUsage" />
</connectionStrings>
```

KeyRestrictionBehavior with AllowOnly Syntax

The following example enables two connection strings that also contain `Initial Catalog`, `Connection Timeout`, `Encrypt`, and `Packet Size` parameters. All other connection string parameters are restricted.

```
<connectionStrings>
  <add name="DatabaseConnection"
    connectionString="Data Source=(local);Initial
    Catalog=Northwind;Integrated Security=true;"
    KeyRestrictions="Initial Catalog;Connection Timeout=;
       Encrypt=;Packet Size=;"
    KeyRestrictionBehavior="AllowOnly" />

  <add name="DatabaseConnection2"
    connectionString="Data Source=SqlServer2;Initial
    Catalog=Northwind2;Integrated Security=true;"
    KeyRestrictions="Initial Catalog;Connection Timeout=;
       Encrypt=;Packet Size=;"
    KeyRestrictionBehavior="AllowOnly" />
</connectionStrings>
```

Enabling Partial Trust with a Custom Permission Set

To enable the use of System.Data.SqlClient permissions for a particular zone, a system administrator must create a custom permission set and set it as the permission set for a particular zone. Default permission sets, such as `LocalIntranet`, cannot be modified. For example, to include System.Data.SqlClient permissions for code that has a Zone of `LocalIntranet`, a system administrator could copy the permission set for `LocalIntranet`, rename it to "CustomLocalIntranet", add the System.Data.SqlClient permissions, import the CustomLocalIntranet permission set using the Caspol.exe (Code Access Security Policy Tool), and set the permission set of `LocalIntranet_Zone` to

CustomLocalIntranet.

Sample Permission Set

The following is a sample permission set for the .NET Framework Data Provider for SQL Server in a partially trusted scenario. For information on creating custom permission sets, see Configuring Permission Sets Using Caspol.exe.

```
<PermissionSet class="System.Security.NamedPermissionSet"
  version="1"
  Name="CustomLocalIntranet"
  Description="Custom permission set given to applications on
    the local intranet">

<IPermission class="System.Data.SqlClient.SqlClientPermission, System.Data, Version=2.0.0000.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089"
version="1"
AllowBlankPassword="False">
<add ConnectionString="Data Source=(local);Integrated Security=true;"
 KeyRestrictions="Initial Catalog=;Connection Timeout=;
   Encrypt=;Packet Size=;"
 KeyRestrictionBehavior="AllowOnly" />
 </IPermission>
</PermissionSet>
```

Verifying ADO.NET Code Access Using Security Permissions

For partial-trust scenarios, you can require CAS privileges for particular methods in your code by specifying a SqlClientPermissionAttribute. If that privilege is not allowed by the restricted security policy in effect, an exception is thrown before your code is run. For more information on security policy, see Security Policy Management and Security Policy Best Practices.

Example

The following example demonstrates how to write code that requires a particular connection string. It simulates denying unrestricted permissions to System.Data.SqlClient, which a system administrator would implement using a CAS policy in the real world.

> **IMPORTANT**
>
> When designing CAS permissions for ADO.NET, the correct pattern is to start with the most restrictive case (no permissions at all) and then add the specific permissions that are needed for the particular task that the code needs to perform. The opposite pattern, starting with all permissions and then denying a specific permission, is not secure because there are many ways of expressing the same connection string. For example, if you start with all permissions and then attempt to deny the use of the connection string "server=someserver", the string "server=someserver.mycompany.com" would still be allowed. By always starting by granting no permissions at all, you reduce the chances that there are holes in the permission set.

The following code demonstrates how `SqlClient` performs the security demand, which throws a SecurityException if the appropriate CAS permissions are not in place. The SecurityException output is displayed in the console window.

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Security;
using System.Security.Permissions;

namespace PartialTrustTopic {
   public class PartialTrustHelper : MarshalByRefObject {
      public void TestConnectionOpen(string connectionString) {
```

```
      public void TestConnectionOpen(string connectionString) {
         // Try to open a connection.
         using (SqlConnection connection = new SqlConnection(connectionString)) {
            connection.Open();
         }
      }
   }

   class Program {
      static void Main(string[] args) {
         TestCAS("Data Source=(local);Integrated Security=true", "Data Source=(local);Integrated
Security=true;Initial Catalog=Test");
      }

      static void TestCAS(string connectString1, string connectString2) {
         // Create permission set for sandbox AppDomain.
         // This example only allows execution.
         PermissionSet permissions = new PermissionSet(PermissionState.None);
         permissions.AddPermission(new SecurityPermission(SecurityPermissionFlag.Execution));

         // Create sandbox AppDomain with permission set that only allows execution,
         // and has no SqlClientPermissions.
         AppDomainSetup appDomainSetup = new AppDomainSetup();
         appDomainSetup.ApplicationBase = AppDomain.CurrentDomain.SetupInformation.ApplicationBase;
         AppDomain firstDomain = AppDomain.CreateDomain("NoSqlPermissions", null, appDomainSetup,
permissions);

         // Create helper object in sandbox AppDomain so that code can be executed in that AppDomain.
         Type helperType = typeof(PartialTrustHelper);
         PartialTrustHelper firstHelper =
(PartialTrustHelper)firstDomain.CreateInstanceAndUnwrap(helperType.Assembly.FullName, helperType.FullName);

         try {
            // Attempt to open a connection in the sandbox AppDomain.
            // This is expected to fail.
            firstHelper.TestConnectionOpen(connectString1);
            Console.WriteLine("Connection opened, unexpected.");
         }
         catch (System.Security.SecurityException ex) {
            Console.WriteLine("Failed, as expected: {0}",
                ex.FirstPermissionThatFailed);

            // Uncomment the following line to see Exception details.
            // Console.WriteLine("BaseException: " + ex.GetBaseException());
         }

         // Add permission for a specific connection string.
         SqlClientPermission sqlPermission = new SqlClientPermission(PermissionState.None);
         sqlPermission.Add(connectString1, "", KeyRestrictionBehavior.AllowOnly);

         permissions.AddPermission(sqlPermission);

         AppDomain secondDomain = AppDomain.CreateDomain("OneSqlPermission", null, appDomainSetup,
permissions);
         PartialTrustHelper secondHelper =
(PartialTrustHelper)secondDomain.CreateInstanceAndUnwrap(helperType.Assembly.FullName, helperType.FullName);

         // Try connection open again, it should succeed now.
         try {
            secondHelper.TestConnectionOpen(connectString1);
            Console.WriteLine("Connection opened, as expected.");
         }
         catch (System.Security.SecurityException ex) {
            Console.WriteLine("Unexpected failure: {0}", ex.Message);
         }

         // Try a different connection string. This should fail.
         try {
            secondHelper.TestConnectionOpen(connectString2);
            Console.WriteLine("Connection opened, unexpected.");
```

```
            Console.WriteLine("Connection opened, unexpected.");
         }
         catch (System.Security.SecurityException ex) {
            Console.WriteLine("Failed, as expected: {0}", ex.Message);
         }
      }
   }
}
```

```
Imports System.Data
Imports System.Data.SqlClient
Imports System.Security
Imports System.Security.Permissions

Namespace PartialTrustTopic
   Public Class PartialTrustHelper
      Inherits MarshalByRefObject
      Public Sub TestConnectionOpen(ByVal connectionString As String)
         ' Try to open a connection.
         Using connection As New SqlConnection(connectionString)
            connection.Open()
         End Using
      End Sub
   End Class

   Class Program
      Public Shared Sub Main(ByVal args As String())
         TestCAS("Data Source=(local);Integrated Security=true", "Data Source=(local);Integrated
Security=true;Initial Catalog=Test")
      End Sub

      Public Shared Sub TestCAS(ByVal connectString1 As String, ByVal connectString2 As String)
         ' Create permission set for sandbox AppDomain.
         ' This example only allows execution.
         Dim permissions As New PermissionSet(PermissionState.None)
         permissions.AddPermission(New SecurityPermission(SecurityPermissionFlag.Execution))

         ' Create sandbox AppDomain with permission set that only allows execution,
         ' and has no SqlClientPermissions.
         Dim appDomainSetup As New AppDomainSetup()
         appDomainSetup.ApplicationBase = AppDomain.CurrentDomain.SetupInformation.ApplicationBase
         Dim firstDomain As AppDomain = AppDomain.CreateDomain("NoSqlPermissions", Nothing, appDomainSetup,
permissions)

         ' Create helper object in sandbox AppDomain so that code can be executed in that AppDomain.
         Dim helperType As Type = GetType(PartialTrustHelper)
         Dim firstHelper As PartialTrustHelper =
DirectCast(firstDomain.CreateInstanceAndUnwrap(helperType.Assembly.FullName, helperType.FullName),
PartialTrustHelper)

         Try
            ' Attempt to open a connection in the sandbox AppDomain.
            ' This is expected to fail.
            firstHelper.TestConnectionOpen(connectString1)
            Console.WriteLine("Connection opened, unexpected.")
         Catch ex As System.Security.SecurityException

            ' Uncomment the following line to see Exception details.
            ' Console.WriteLine("BaseException: " + ex.GetBaseException());
            Console.WriteLine("Failed, as expected: {0}", ex.FirstPermissionThatFailed)
         End Try

         ' Add permission for a specific connection string.
         Dim sqlPermission As New SqlClientPermission(PermissionState.None)
         sqlPermission.Add(connectString1, "", KeyRestrictionBehavior.AllowOnly)

         permissions.AddPermission(sqlPermission)
```

```
        Dim secondDomain As AppDomain = AppDomain.CreateDomain("OneSqlPermission", Nothing, appDomainSetup,
permissions)
        Dim secondHelper As PartialTrustHelper =
DirectCast(secondDomain.CreateInstanceAndUnwrap(helperType.Assembly.FullName, helperType.FullName),
PartialTrustHelper)

        ' Try connection open again, it should succeed now.
        Try
           secondHelper.TestConnectionOpen(connectString1)
           Console.WriteLine("Connection opened, as expected.")
        Catch ex As System.Security.SecurityException
           Console.WriteLine("Unexpected failure: {0}", ex.Message)
        End Try

        ' Try a different connection string. This should fail.
        Try
           secondHelper.TestConnectionOpen(connectString2)
           Console.WriteLine("Connection opened, unexpected.")
        Catch ex As System.Security.SecurityException
           Console.WriteLine("Failed, as expected: {0}", ex.Message)
        End Try
     End Sub
  End Class
End Namespace
```

You should see this output in the Console window:

```
Failed, as expected: <IPermission class="System.Data.SqlClient.
SqlClientPermission, System.Data, Version=2.0.0.0,
  Culture=neutral, PublicKeyToken=b77a5c561934e089" version="1"
  AllowBlankPassword="False">
<add ConnectionString="Data Source=(local);Initial Catalog=
  Northwind;Integrated Security=SSPI" KeyRestrictions=""
KeyRestrictionBehavior="AllowOnly"/>
</IPermission>

Connection opened, as expected.
Failed, as expected: Request failed.
```

Interoperability with Unmanaged Code

Code that runs outside the CLR is called unmanaged code. Therefore, security mechanisms such as CAS cannot be applied to unmanaged code. COM components, ActiveX interfaces, and Windows API functions are examples of unmanaged code. Special security considerations apply when executing unmanaged code so that you do not jeopardize overall application security. For more information, see Interoperating with Unmanaged Code.

The .NET Framework also supports backward compatibility to existing COM components by providing access through COM interop. You can incorporate COM components into a .NET Framework application by using COM interop tools to import the relevant COM types. Once imported, the COM types are ready to use. COM interop also enables COM clients to access managed code by exporting assembly metadata to a type library and registering the managed component as a COM component. For more information, see Advanced COM Interoperability.

See also

- Securing ADO.NET Applications
- Security in Native and .NET Framework Code
- Role-Based Security
- ADO.NET Overview

Privacy and Data Security

12/24/2019 • 2 minutes to read • Edit Online

Safeguarding and managing sensitive information in an ADO.NET application is dependent upon the underlying products and technologies used to create it. ADO.NET does not directly provide services for securing or encrypting data.

Cryptography and Hash Codes

The classes in the .NET Framework System.Security.Cryptography namespace can be used from your ADO.NET applications to prevent data from being read or modified by unauthorized third parties. Some classes are wrappers for the unmanaged Microsoft CryptoAPI, while others are managed implementations. The Cryptographic Services topic provides an overview of cryptography in the .NET Framework, describes how cryptograph is implemented, and how you can perform specific cryptographic tasks.

Unlike cryptography, which allows data to be encrypted and then decrypted, hashing data is a one-way process. Hashing data is useful when you want to prevent tampering by checking that data has not been altered: given identical input strings, hashing algorithms always produce identical short output values that can easily be compared. Ensuring Data Integrity with Hash Codes describes how you can generate and verify hash values.

Encrypting Configuration Files

Protecting access to your data source is one of the most important goals when securing an application. A connection string presents a potential vulnerability if it is not secured. Connection strings saved in configuration files are stored in standard XML files for which the .NET Framework has defined a common set of elements. Protected configuration enables you to encrypt sensitive information in a configuration file. Although primarily designed for ASP.NET applications, protected configuration can also be used to encrypt configuration file sections in Windows applications. For more information, see Protecting Connection Information.

Securing String Values in Memory

If a String object contains sensitive information, such as a password, credit card number, or personal data, there is a risk that the information could be revealed after it is used because the application cannot delete the data from computer memory.

A String is immutable; its value cannot be modified once it has been created. Changes that appear to modify the string value actually create a new instance of a String object in memory, storing the data as plain text. In addition, it is not possible to predict when the string instances will be deleted from memory. Memory reclamation with strings is not deterministic with .NET garbage collection. You should avoid using the String and StringBuilder classes if your data is truly sensitive.

The SecureString class provides methods for encrypting text using the Data Protection API (DPAPI) in memory. The string is then deleted from memory when it is no longer needed. There is no `ToString` method to quickly read the contents of a SecureString. You can initialize a new instance of `SecureString` with no value or by passing it a pointer to an array of Char objects. You can then use the various methods of the class to work with the string.

See also

- Securing ADO.NET Applications
- SQL Server Security

- ADO.NET Overview

Data Type Mappings in ADO.NET

2/4/2020 • 2 minutes to read • Edit Online

The .NET Framework is based on the common type system, which defines how types are declared, used, and managed in the runtime. It consists of both value types and reference types, which all derive from the Object base type. When working with a data source, the data type is inferred from the data provider if it is not explicitly specified. For example, a DataSet object is independent of any specific data source. Data in a `DataSet` is retrieved from a data source, and changes are persisted back to the data source by using a `DataAdapter`. This means that when a `DataAdapter` fills a DataTable in a `DataSet` with values from a data source, the resulting data types of the columns in the `DataTable` are .NET Framework types, instead of types specific to the .NET Framework data provider that is used to connect to the data source.

Likewise, when a `DataReader` returns a value from a data source, the resulting value is stored in a local variable that has a .NET Framework type. For both the `Fill` operations of the `DataAdapter` and the `Get` methods of the `DataReader`, the .NET Framework type is inferred from the value returned from the .NET Framework data provider.

Instead of relying on the inferred data type, you can use the typed accessor methods of the `DataReader` when you know the specific type of the value being returned. Typed accessor methods give you better performance by returning a value as a specific .NET Framework type, which eliminates the need for additional type conversion.

> **NOTE**
> Null values for .NET Framework data provider data types are represented by `DBNull.Value`.

In This Section

SQL Server Data Type Mappings
Lists inferred data type mappings and data accessor methods for System.Data.SqlClient.

OLE DB Data Type Mappings
Lists inferred data type mappings and data accessor methods for System.Data.OleDb.

ODBC Data Type Mappings
Lists inferred data type mappings and data accessor methods for System.Data.Odbc.

Oracle Data Type Mappings
Lists inferred data type mappings and data accessor methods for System.Data.OracleClient.

Floating-Point Numbers
Describes issues that developers frequently encounter when working with floating-point numbers.

See also

- SQL Server Data Types and ADO.NET
- Configuring Parameters and Parameter Data Types
- Retrieving Database Schema Information
- Common Type System
- Converting Types
- ADO.NET Overview

SQL Server Data Type Mappings

9/7/2019 • 2 minutes to read • Edit Online

SQL Server and the .NET Framework are based on different type systems. For example, the .NET Framework Decimal structure has a maximum scale of 28, whereas the SQL Server decimal and numeric data types have a maximum scale of 38. To maintain data integrity when reading and writing data, the SqlDataReader exposes SQL Server–specific typed accessor methods that return objects of System.Data.SqlTypes as well as accessor methods that return .NET Framework types. Both SQL Server types and .NET Framework types are also represented by enumerations in the DbType and SqlDbType classes, which you can use when specifying SqlParameter data types.

The following table shows the inferred .NET Framework type, the DbType and SqlDbType enumerations, and the accessor methods for the SqlDataReader.

SQL SERVER DATABASE ENGINE TYPE	.NET FRAMEWORK TYPE	SQLDBTYPE ENUMERATION	SQLDATAREADER SQLTYPES TYPED ACCESSOR	DBTYPE ENUMERATION	SQLDATAREADER DBTYPE TYPED ACCESSOR
bigint	Int64	BigInt	GetSqlInt64	Int64	GetInt64
binary	Byte[]	VarBinary	GetSqlBinary	Binary	GetBytes
bit	Boolean	Bit	GetSqlBoolean	Boolean	GetBoolean
char	String Char[]	Char	GetSqlString	AnsiStringFixedLength, String	GetString GetChars
date [1] (SQL Server 2008 and later)	DateTime	Date [1]	GetSqlDateTime	Date [1]	GetDateTime
datetime	DateTime	DateTime	GetSqlDateTime	DateTime	GetDateTime
datetime2 (SQL Server 2008 and later)	DateTime	DateTime2	None	DateTime2	GetDateTime
datetimeoffset (SQL Server 2008 and later)	DateTimeOffset	DateTimeOffset	none	DateTimeOffset	GetDateTimeOffset
decimal	Decimal	Decimal	GetSqlDecimal	Decimal	GetDecimal
FILESTREAM attribute (varbinary(max))	Byte[]	VarBinary	GetSqlBytes	Binary	GetBytes
float	Double	Float	GetSqlDouble	Double	GetDouble

SQL SERVER DATABASE ENGINE TYPE	.NET FRAMEWORK TYPE	SQLDBTYPE ENUMERATION	SQLDATAREADER SQLTYPES TYPED ACCESSOR	DBTYPE ENUMERATION	SQLDATAREADER DBTYPE TYPED ACCESSOR
image	Byte[]	Binary	GetSqlBinary	Binary	GetBytes
int	Int32	Int	GetSqlInt32	Int32	GetInt32
money	Decimal	Money	GetSqlMoney	Decimal	GetDecimal
nchar	String Char[]	NChar	GetSqlString	StringFixedLength	GetString GetChars
ntext	String Char[]	NText	GetSqlString	String	GetString GetChars
numeric	Decimal	Decimal	GetSqlDecimal	Decimal	GetDecimal
nvarchar	String Char[]	NVarChar	GetSqlString	String	GetString GetChars
real	Single	Real	GetSqlSingle	Single	GetFloat
rowversion	Byte[]	Timestamp	GetSqlBinary	Binary	GetBytes
smalldatetime	DateTime	DateTime	GetSqlDateTime	DateTime	GetDateTime
smallint	Int16	SmallInt	GetSqlInt16	Int16	GetInt16
smallmoney	Decimal	SmallMoney	GetSqlMoney	Decimal	GetDecimal
sql_variant	Object [2]	Variant	GetSqlValue [2]	Object	GetValue [2]
text	String Char[]	Text	GetSqlString	String	GetString GetChars
time (SQL Server 2008 and later)	TimeSpan	Time	none	Time	GetDateTime
timestamp	Byte[]	Timestamp	GetSqlBinary	Binary	GetBytes
tinyint	Byte	TinyInt	GetSqlByte	Byte	GetByte
uniqueidentifier	Guid	UniqueIdentifier	GetSqlGuid	Guid	GetGuid
varbinary	Byte[]	VarBinary	GetSqlBinary	Binary	GetBytes

SQL SERVER DATABASE ENGINE TYPE	.NET FRAMEWORK TYPE	SQLDBTYPE ENUMERATION	SQLDATAREADER SQLTYPES TYPED ACCESSOR	DBTYPE ENUMERATION	SQLDATAREADER DBTYPE TYPED ACCESSOR
varchar	String Char[]	VarChar	GetSqlString	AnsiString, String	GetString GetChars
xml	Xml	Xml	GetSqlXml	Xml	none

[1] You cannot set the `DbType` property of a `SqlParameter` to `SqlDbType.Date`.

[2] Use a specific typed accessor if you know the underlying type of the `sql_variant`.

SQL Server documentation

For more information about SQL Server data types, see Data types (Transact-SQL).

See also

- SQL Server Data Types and ADO.NET
- SQL Server Binary and Large-Value Data
- Data Type Mappings in ADO.NET
- Configuring Parameters and Parameter Data Types
- ADO.NET Overview

OLE DB Data Type Mappings

9/7/2019 • 2 minutes to read • Edit Online

The following table shows the inferred .NET Framework type for data types from the .NET Framework Data Provider for ADO and OLE DB (System.Data.OleDb). The typed accessor methods for the OleDbDataReader are also listed.

ADO TYPE	OLE DB TYPE	.NET FRAMEWORK TYPE	.NET FRAMEWORK TYPED ACCESSOR
adBigInt	DBTYPE_I8	Int64	GetInt64()
adBinary	DBTYPE_BYTES	Byte[]	GetBytes()
adBoolean	DBTYPE_BOOL	Boolean	GetBoolean()
adBSTR	DBTYPE_BSTR	String	GetString()
adChapter	DBTYPE_HCHAPTER	Supported through the `DataReader`. See Retrieving Data Using a DataReader.	GetValue()
adChar	DBTYPE_STR	String	GetString()
adCurrency	DBTYPE_CY	Decimal	GetDecimal()
adDate	DBTYPE_DATE	DateTime	GetDateTime()
adDBDate	DBTYPE_DBDATE	DateTime	GetDateTime()
adDBTime	DBTYPE_DBTIME	DateTime	GetDateTime()
adDBTimeStamp	DBTYPE_DBTIMESTAMP	DateTime	GetDateTime()
adDecimal	DBTYPE_DECIMAL	Decimal	GetDecimal()
adDouble	DBTYPE_R8	Double	GetDouble()
adError	DBTYPE_ERROR	ExternalException	GetValue()
adFileTime	DBTYPE_FILETIME	DateTime	GetDateTime()
adGUID	DBTYPE_GUID	Guid	GetGuid()
adIDispatch	DBTYPE_IDISPATCH *	Object	GetValue()
adInteger	DBTYPE_I4	Int32	GetInt32()
adIUnknown	DBTYPE_IUNKNOWN *	Object	GetValue()

ADO TYPE	OLE DB TYPE	.NET FRAMEWORK TYPE	.NET FRAMEWORK TYPED ACCESSOR
adNumeric	DBTYPE_NUMERIC	Decimal	GetDecimal()
adPropVariant	DBTYPE_PROPVARIANT	Object	GetValue()
adSingle	DBTYPE_R4	Single	GetFloat()
adSmallInt	DBTYPE_I2	Int16	GetInt16()
adTinyInt	DBTYPE_I1	Byte	GetByte()
adUnsignedBigInt	DBTYPE_UI8	UInt64	GetValue()
adUnsignedInt	DBTYPE_UI4	UInt32	GetValue()
adUnsignedSmallInt	DBTYPE_UI2	UInt16	GetValue()
adUnsignedTinyInt	DBTYPE_UI1	Byte	GetByte()
adVariant	DBTYPE_VARIANT	Object	GetValue()
adWChar	DBTYPE_WSTR	String	GetString()
adUserDefined	DBTYPE_UDT	not supported	
adVarNumeric	DBTYPE_VARNUMERIC	not supported	

* For the OLE DB types `DBTYPE_IUNKNOWN` and `DBTYPE_IDISPATCH`, the object reference is a marshaled representation of the pointer.

See also

- Retrieving and Modifying Data in ADO.NET
- ADO.NET Overview

ODBC Data Type Mappings

9/7/2019 • 2 minutes to read • Edit Online

The following table shows the inferred .NET Framework type for data types from the .NET Framework Data Provider for ODBC (System.Data.Odbc). The typed accessor methods for the OdbcDataReader are also listed.

ODBC TYPE	.NET FRAMEWORK TYPE	.NET FRAMEWORK TYPED ACCESSOR
SQL_BIGINT	Int64	GetInt64()
SQL_BINARY	Byte[]	GetBytes()
SQL_BIT	Boolean	GetBoolean()
SQL_CHAR	String Char[]	GetString() GetChars()
SQL_DECIMAL	Decimal	GetDecimal()
SQL_DOUBLE	Double	GetDouble()
SQL_GUID	Guid	GetGuid()
SQL_INTEGER	Int32	GetInt32()
SQL_LONG_VARCHAR	String Char[]	GetString() GetChars()
SQL_LONGVARBINARY	Byte[]	GetBytes()
SQL_NUMERIC	Decimal	GetDecimal()
SQL_REAL	Single	GetFloat()
SQL_SMALLINT	Int16	GetInt16()
SQL_TINYINT	Byte	GetByte()
SQL_TYPE_TIMES	DateTime	GetDateTime()
SQL_TYPE_TIMESTAMP	DateTime	GetDateTime()
SQL_VARBINARY	Byte[]	GetBytes()
SQL_WCHAR	String Char[]	GetString() GetChars()

ODBC TYPE	.NET FRAMEWORK TYPE	.NET FRAMEWORK TYPED ACCESSOR
SQL_WLONGVARCHAR	String Char[]	GetString() GetChars()
SQL_WVARCHAR	String Char[]	GetString() GetChars()

See also

- Retrieving and Modifying Data in ADO.NET
- ADO.NET Overview

Oracle Data Type Mappings

9/7/2019 • 3 minutes to read • Edit Online

The following table lists Oracle data types and their mappings to the OracleDataReader.

ORACLE DATA TYPE	.NET FRAMEWORK DATA TYPE RETURNED BY ORACLEDATAREADER.GETVALUE	ORACLECLIENT DATA TYPE RETURNED BY ORACLEDATAREADER.GETORACLEVALUE	REMARKS
BFILE	Byte[]	OracleBFile	
BLOB	Byte[]	OracleLob	
CHAR	String	OracleString	
CLOB	String	OracleLob	
DATE	DateTime	OracleDateTime	
FLOAT	Decimal	OracleNumber	This data type is an alias for the **NUMBER** data type, and is designed so that the OracleDataReader returns a **System.Decimal** or OracleNumber instead of a floating-point value. Using the .NET Framework data type can cause an overflow.
INTEGER	Decimal	OracleNumber	This data type is an alias for the **NUMBER(38)** data type, and is designed so that the OracleDataReader returns a **System.Decimal** or OracleNumber instead of an integer value. Using the .NET Framework data type can cause an overflow.
INTERVAL YEAR TO MONTH	Int32	OracleMonthSpan	
INTERVAL DAY TO SECOND	TimeSpan	OracleTimeSpan	
LONG	String	OracleString	
LONG RAW	Byte[]	OracleBinary	
NCHAR	String	OracleString	
NCLOB	String	OracleLob	

ORACLE DATA TYPE	.NET FRAMEWORK DATA TYPE RETURNED BY ORACLEDATAREADER.GETVALUE	ORACLECLIENT DATA TYPE RETURNED BY ORACLEDATAREADER.GETORACLEVALUE	REMARKS
NUMBER	Decimal	OracleNumber	Using the .NET Framework data type can cause an overflow.
NVARCHAR2	String	OracleString	
RAW	Byte[]	OracleBinary	
REF CURSOR			The Oracle **REF CURSOR** data type is not supported by the OracleDataReader object.
ROWID	String	OracleString	
TIMESTAMP	DateTime	OracleDateTime	
TIMESTAMP WITH LOCAL TIME ZONE	DateTime	OracleDateTime	
TIMESTAMP WITH TIME ZONE	DateTime	OracleDateTime	
UNSIGNED INTEGER	Number	OracleNumber	This data type is an alias for the **NUMBER(38)** data type, and is designed so that the OracleDataReader returns a **System.Decimal** or OracleNumber instead of an unsigned integer value. Using the .NET Framework data type can cause an overflow.
VARCHAR2	String	OracleString	

The following table lists Oracle data types and the .NET Framework data types (**System.Data.DbType** and OracleType) to use when binding them as parameters.

ORACLE DATA TYPE	DBTYPE ENUMERATION TO BIND AS A PARAMETER	ORACLETYPE ENUMERATION TO BIND AS A PARAMETER	REMARKS
BFILE		BFile	Oracle only allows binding a **BFILE** as a **BFILE** parameter. The .NET Data Provider for Oracle does not automatically construct one for you if you attempt to bind a non-**BFILE** value, such as **byte[]** or OracleBinary.

ORACLE DATA TYPE	DBTYPE ENUMERATION TO BIND AS A PARAMETER	ORACLETYPE ENUMERATION TO BIND AS A PARAMETER	REMARKS
BLOB		Blob	Oracle only allows binding a **BLOB** as a **BLOB** parameter. The .NET Data Provider for Oracle does not automatically construct one for you if you attempt to bind a non-**BLOB** value, such as **byte[]** or OracleBinary.
CHAR	AnsiStringFixedLength	Char	
CLOB		Clob	Oracle only allows binding a **CLOB** as a **CLOB** parameter. The .NET Data Provider for Oracle does not automatically construct one for you if you attempt to bind a non-**CLOB** value, such as **System.String** or OracleString.
DATE	DateTime	DateTime	
FLOAT	Single, Double, Decimal	Float, Double, Number	Size determines the **System.Data.DBType** and OracleType.
INTEGER	SByte, Int16, Int32, Int64, Decimal	SByte, Int16, Int32, Number	Size determines the **System.Data.DBType** and OracleType.
INTERVAL YEAR TO MONTH	Int32	IntervalYearToMonth	OracleType is only available when using both Oracle 9i client and server software.
INTERVAL DAY TO SECOND	Object	IntervalDayToSecond	OracleType is only available when using both Oracle 9i client and server software.
LONG	AnsiString	LongVarChar	
LONG RAW	Binary	LongRaw	
NCHAR	StringFixedLength	NChar	
NCLOB		NClob	Oracle only allows binding a **NCLOB** as a **NCLOB** parameter. The .NET Data Provider for Oracle does not automatically construct one for you if you attempt to bind a non-**NCLOB** value, such as **System.String** or OracleString.

ORACLE DATA TYPE	DBTYPE ENUMERATION TO BIND AS A PARAMETER	ORACLETYPE ENUMERATION TO BIND AS A PARAMETER	REMARKS
NUMBER	VarNumeric	Number	
NVARCHAR2	String	NVarChar	
RAW	Binary	Raw	
REF CURSOR		Cursor	For more information, see Oracle REF CURSORs.
ROWID	AnsiString	Rowid	
TIMESTAMP	DateTime	Timestamp	OracleType is only available when using both Oracle 9i client and server software.
TIMESTAMP WITH LOCAL TIME ZONE	DateTime	TimestampLocal	OracleType is only available when using both Oracle 9i client and server software.
TIMESTAMP WITH TIME ZONE	DateTime	TimestampWithTz	OracleType is only available when using both Oracle 9i client and server software.
UNSIGNED INTEGER	Byte, UInt16, UInt32, UInt64, Decimal	Byte, UInt16, Uint32, Number	Size determines the **System.Data.DBType** and OracleType.
VARCHAR2	AnsiString	VarChar	

The **InputOutput**, **Output**, and **ReturnValue ParameterDirection** values used by the Value property of the OracleParameter object are .NET Framework data types, unless the input value is an Oracle data type (for example, OracleNumber or OracleString). This does not apply to **REF CURSOR**, **BFILE**, or **LOB** data types.

See also

- Oracle and ADO.NET
- ADO.NET Overview

Floating-Point Numbers

9/5/2019 • 2 minutes to read • Edit Online

This topic describes some of the issues that developers frequently encounter when they work with floating-point numbers in ADO.NET. These issues are caused by the way that computers store floating-point numbers, and are not specific to a particular provider such as System.Data.SqlClient or System.Data.OracleClient.

Floating-point numbers generally do not have an exact binary representation. Instead, the computer stores an approximation of the number. At different times, different numbers of binary digits may be used to represent the number. When a floating point number is converted from one representation to another representation, the least significant digits of that number may vary slightly. Conversion typically occurs when the number is cast from one type to another type. The variation occurs whether the conversion occurs within a database, between types that represent database values, or between types. Because of these changes, numbers that would logically be equal may have changes in their least-significant digits that cause them to have different values. The number of digits of precision in the number may be larger or smaller than expected. When formatted as a string, the number may not show the expected value.

To minimize these effects, you should use the closest match between numeric types that is available to you. For example, if you are working with SQL Server, the exact numeric value may change if you convert a Transact-SQL value of real type to a value of float type. In the .NET Framework, converting a Single to a Double may also produce unexpected results. In both of these cases, a good strategy is to make all the values in the application use the same numeric type. You can also use a fixed-precision decimal type, or cast floating-point numbers to a fixed-precision decimal type before you work with them.

To work around problems with equality comparison, consider coding your application so that variations in the least significant digits are ignored. For example, instead of comparing to see whether two numbers are equal, subtract one number from the other number. If the difference is within an acceptable margin of rounding, your application can treat the numbers as if they are the same.

See also

- Why Floating-Point Numbers May Lose Precision
- ADO.NET Overview

Retrieving and Modifying Data in ADO.NET

2/4/2020 • 2 minutes to read • Edit Online

A primary function of any database application is connecting to a data source and retrieving the data that it contains. The .NET Framework data providers of ADO.NET serve as a bridge between an application and a data source, allowing you to execute commands as well as to retrieve data by using a **DataReader** or a **DataAdapter**. A key function of any database application is the ability to update the data that is stored in the database. In ADO.NET, updating data involves using the **DataAdapter** and DataSet, and **Command** objects; and it may also involve using transactions.

In This Section

Connecting to a Data Source
Describes how to establish a connection to a data source and how to work with connection events.

Connection Strings
Contains topics describing various aspects of using connection strings, including connection string keywords, security info, and storing and retrieving them.

Connection Pooling
Describes connection pooling for the .NET Framework data providers.

Commands and Parameters
Contains topics describing how to create commands and command builders, configure parameters, and how to execute commands to retrieve and modify data.

DataAdapters and DataReaders
Contains topics describing DataReaders, DataAdapters, parameters, handling DataAdapter events and performing batch operations.

Transactions and Concurrency
Contains topics describing how to perform local transactions, distributed transactions, and work with optimistic concurrency.

Retrieving Identity or Autonumber Values
Provides an example of mapping the values generated for an **identity** column in a SQL Server table or for an **Autonumber** field in a Microsoft Access table, to a column of an inserted row in a table. Discusses merging identity values in a `DataTable`.

Retrieving Binary Data
Describes how to retrieve binary data or large data structures using `CommandBehavior`. `SequentialAccess` to modify the default behavior of a `DataReader`.

Modifying Data with Stored Procedures
Describes how to use stored procedure input parameters and output parameters to insert a row in a database, returning a new identity value.

Retrieving Database Schema Information
Describes how to obtain available databases or catalogs, tables and views in a database, constraints that exist for tables, and other schema information from a data source.

DbProviderFactories
Describes the provider factory model and demonstrates how to use the base classes in the `System.Data.Common`

namespace.

Data Tracing in ADO.NET
Describes how ADO.NET provides built-in data tracing functionality.

Performance Counters
Describes performance counters available for `SqlClient` and `OracleClient`.

Asynchronous Programming
Describes ADO.NET support for asynchronous programming.

SqlClient Streaming Support
Discusses how to write applications that stream data from SQL Server without having it fully loaded in memory.

See also

- Data Type Mappings in ADO.NET
- DataSets, DataTables, and DataViews
- Securing ADO.NET Applications
- SQL Server and ADO.NET
- ADO.NET Overview

Connecting to a Data Source in ADO.NET

2/8/2020 • 2 minutes to read • Edit Online

In ADO.NET, you use a **Connection** object to connect to a specific data source by supplying necessary authentication information in a connection string. The **Connection** object you use depends on the type of data source.

Each .NET Framework data provider included with the .NET Framework has a DbConnection object: the .NET Framework Data Provider for OLE DB includes an OleDbConnection object, the .NET Framework Data Provider for SQL Server includes a SqlConnection object, the .NET Framework Data Provider for ODBC includes an OdbcConnection object, and the .NET Framework Data Provider for Oracle includes an OracleConnection object.

In This Section

Establishing the Connection
Describes how to use a **Connection** object to establish a connection to a data source.

Connection Events
Describes how to use an **InfoMessage** event to retrieve informational messages from a data source.

See also

- Connection Strings
- Connection Pooling
- Commands and Parameters
- DataAdapters and DataReaders
- Transactions and Concurrency
- ADO.NET Overview

Establishing the Connection

3/12/2020 • 5 minutes to read • Edit Online

To connect to Microsoft SQL Server, use the SqlConnection object of the .NET Framework Data Provider for SQL Server. To connect to an OLE DB data source, use the OleDbConnection object of the .NET Framework Data Provider for OLE DB. To connect to an ODBC data source, use the OdbcConnection object of the .NET Framework Data Provider for ODBC. To connect to an Oracle data source, use the OracleConnection object of the .NET Framework Data Provider for Oracle. For securely storing and retrieving connection strings, see Protecting Connection Information.

Closing Connections

We recommend that you always close the connection when you are finished using it, so that the connection can be returned to the pool. The `Using` block in Visual Basic or C# automatically disposes of the connection when the code exits the block, even in the case of an unhandled exception. See using Statement and Using Statement for more information.

You can also use the `Close` or `Dispose` methods of the connection object for the provider that you are using. Connections that are not explicitly closed might not be added or returned to the pool. For example, a connection that has gone out of scope but that has not been explicitly closed will only be returned to the connection pool if the maximum pool size has been reached and the connection is still valid. For more information, see OLE DB, ODBC, and Oracle Connection Pooling.

> **NOTE**
>
> Do not call `Close` or `Dispose` on a **Connection**, a **DataReader**, or any other managed object in the `Finalize` method of your class. In a finalizer, only release unmanaged resources that your class owns directly. If your class does not own any unmanaged resources, do not include a `Finalize` method in your class definition. For more information, see Garbage Collection.

> **NOTE**
>
> Login and logout events will not be raised on the server when a connection is fetched from or returned to the connection pool, because the connection is not actually closed when it is returned to the connection pool. For more information, see SQL Server Connection Pooling (ADO.NET).

Connecting to SQL Server

The .NET Framework Data Provider for SQL Server supports a connection string format that is similar to the OLE DB (ADO) connection string format. For valid string format names and values, see the ConnectionString property of the SqlConnection object. You can also use the SqlConnectionStringBuilder class to create syntactically valid connection strings at run time. For more information, see Connection String Builders.

The following code example demonstrates how to create and open a connection to a SQL Server database.

```
' Assumes connectionString is a valid connection string.
Using connection As New SqlConnection(connectionString)
    connection.Open()
    ' Do work here.
End Using
```

```
// Assumes connectionString is a valid connection string.
using (SqlConnection connection = new SqlConnection(connectionString))
{
    connection.Open();
    // Do work here.
}
```

Integrated Security and ASP.NET

SQL Server integrated security (also known as trusted connections) helps to provide protection when connecting to SQL Server as it does not expose a user ID and password in the connection string and is the recommended method for authenticating a connection. Integrated security uses the current security identity, or token, of the executing process. For desktop applications, this is typically the identity of the currently logged-on user.

The security identity for ASP.NET applications can be set to one of several different options. To better understand the security identity that an ASP.NET application uses when connecting to SQL Server, see ASP.NET Impersonation, ASP.NET Authentication, and How to: Access SQL Server Using Windows Integrated Security.

Connecting to an OLE DB Data Source

The .NET Framework Data Provider for OLE DB provides connectivity to data sources exposed using OLE DB (through SQLOLEDB, the OLE DB Provider for SQL Server), using the **OleDbConnection** object.

For the .NET Framework Data Provider for OLE DB, the connection string format is identical to the connection string format used in ADO, with the following exceptions:

- The **Provider** keyword is required.
- The **URL**, **Remote Provider**, and **Remote Server** keywords are not supported.

For more information about OLE DB connection strings, see the ConnectionString topic. You can also use the OleDbConnectionStringBuilder to create connection strings at run time.

> **NOTE**
> The **OleDbConnection** object does not support setting or retrieving dynamic properties specific to an OLE DB provider. Only properties that can be passed in the connection string for the OLE DB provider are supported.

The following code example demonstrates how to create and open a connection to an OLE DB data source.

```
' Assumes connectionString is a valid connection string.
Using connection As New OleDbConnection(connectionString)
    connection.Open()
    ' Do work here.
End Using
```

```
// Assumes connectionString is a valid connection string.
using (OleDbConnection connection =
  new OleDbConnection(connectionString))
{
    connection.Open();
    // Do work here.
}
```

Do Not Use Universal Data Link Files

It is possible to supply connection information for an **OleDbConnection** in a Universal Data Link (UDL) file; however you should avoid doing so. UDL files are not encrypted, and expose connection string information in clear text. Because a UDL file is an external file-based resource to your application, it cannot be secured using the .NET Framework.

Connecting to an ODBC Data Source

The .NET Framework Data Provider for ODBC provides connectivity to data sources exposed using ODBC using the **OdbcConnection** object.

For the .NET Framework Data Provider for ODBC, the connection string format is designed to match the ODBC connection string format as closely as possible. You may also supply an ODBC data source name (DSN). For more detail on the **OdbcConnection** , see the OdbcConnection.

The following code example demonstrates how to create and open a connection to an ODBC data source.

```
' Assumes connectionString is a valid connection string.
Using connection As New OdbcConnection(connectionString)
    connection.Open()
    ' Do work here.
End Using
```

```
// Assumes connectionString is a valid connection string.
using (OdbcConnection connection =
  new OdbcConnection(connectionString))
{
    connection.Open();
    // Do work here.
}
```

Connecting to an Oracle Data Source

The .NET Framework Data Provider for Oracle provides connectivity to Oracle data sources using the **OracleConnection** object.

For the .NET Framework Data Provider for Oracle, the connection string format is designed to match the OLE DB Provider for Oracle (MSDAORA) connection string format as closely as possible. For more detail on the **OracleConnection**, see the OracleConnection.

The following code example demonstrates how to create and open a connection to an Oracle data source.

```
' Assumes connectionString is a valid connection string.
Using connection As New OracleConnection(connectionString)
    connection.Open()
    ' Do work here.
End Using
```

```
// Assumes connectionString is a valid connection string.
using (OracleConnection connection =
  new OracleConnection(connectionString))
{
    connection.Open();
    // Do work here.
}
OracleConnection nwindConn = new OracleConnection("Data Source=MyOracleServer;Integrated Security=yes;");
nwindConn.Open();
```

See also

- Connecting to a Data Source
- Connection Strings
- OLE DB, ODBC, and Oracle Connection Pooling
- ADO.NET Overview

Connection Events

3/12/2020 • 3 minutes to read • Edit Online

All of the .NET Framework data providers have **Connection** objects with two events that you can use to retrieve informational messages from a data source or to determine if the state of a **Connection** has changed. The following table describes the events of the **Connection** object.

EVENT	DESCRIPTION
InfoMessage	Occurs when an informational message is returned from a data source. Informational messages are messages from a data source that do not result in an exception being thrown.
StateChange	Occurs when the state of the **Connection** changes.

Working with the InfoMessage Event

You can retrieve warnings and informational messages from a SQL Server data source using the InfoMessage event of the SqlConnection object. Errors returned from the data source with a severity level of 11 through 16 cause an exception to be thrown. However, the InfoMessage event can be used to obtain messages from the data source that are not associated with an error. In the case of Microsoft SQL Server, any error with a severity of 10 or less is considered to be an informational message, and can be captured by using the InfoMessage event. For more information, see the Database Engine Error Severities article.

The InfoMessage event receives an SqlInfoMessageEventArgs object containing, in its **Errors** property, a collection of the messages from the data source. You can query the **Error** objects in this collection for the error number and message text, as well as the source of the error. The .NET Framework Data Provider for SQL Server also includes detail about the database, stored procedure, and line number that the message came from.

Example

The following code example shows how to add an event handler for the InfoMessage event.

```
' Assumes that connection represents a SqlConnection object.
  AddHandler connection.InfoMessage, _
    New SqlInfoMessageEventHandler(AddressOf OnInfoMessage)

Private Shared Sub OnInfoMessage(sender As Object, _
  args As SqlInfoMessageEventArgs)
  Dim err As SqlError
  For Each err In args.Errors
    Console.WriteLine("The {0} has received a severity {1}, _
       state {2} error number {3}\n" & _
      "on line {4} of procedure {5} on server {6}:\n{7}", _
      err.Source, err.Class, err.State, err.Number, err.LineNumber, _
    err.Procedure, err.Server, err.Message)
  Next
End Sub
```

```
// Assumes that connection represents a SqlConnection object.
  connection.InfoMessage +=
    new SqlInfoMessageEventHandler(OnInfoMessage);

protected static void OnInfoMessage(
  object sender, SqlInfoMessageEventArgs args)
{
  foreach (SqlError err in args.Errors)
  {
    Console.WriteLine(
  "The {0} has received a severity {1}, state {2} error number {3}\n" +
  "on line {4} of procedure {5} on server {6}:\n{7}",
   err.Source, err.Class, err.State, err.Number, err.LineNumber,
   err.Procedure, err.Server, err.Message);
  }
}
```

Handling Errors as InfoMessages

The InfoMessage event will normally fire only for informational and warning messages that are sent from the server. However, when an actual error occurs, the execution of the **ExecuteNonQuery** or **ExecuteReader** method that initiated the server operation is halted and an exception is thrown.

If you want to continue processing the rest of the statements in a command regardless of any errors produced by the server, set the FireInfoMessageEventOnUserErrors property of the SqlConnection to `true`. Doing this causes the connection to fire the InfoMessage event for errors instead of throwing an exception and interrupting processing. The client application can then handle this event and respond to error conditions.

> **NOTE**
> An error with a severity level of 17 or above that causes the server to stop processing the command must be handled as an exception. In this case, an exception is thrown regardless of how the error is handled in the InfoMessage event.

Working with the StateChange Event

The **StateChange** event occurs when the state of a **Connection** changes. The **StateChange** event receives StateChangeEventArgs that enable you to determine the change in state of the **Connection** by using the **OriginalState** and **CurrentState** properties. The **OriginalState** property is a ConnectionState enumeration that indicates the state of the **Connection** before it changed. **CurrentState** is a ConnectionState enumeration that indicates the state of the **Connection** after it changed.

The following code example uses the **StateChange** event to write a message to the console when the state of the **Connection** changes.

```
' Assumes connection represents a SqlConnection object.
  AddHandler connection.StateChange, _
    New StateChangeEventHandler(AddressOf OnStateChange)

Protected Shared Sub OnStateChange( _
  sender As Object, args As StateChangeEventArgs)

  Console.WriteLine( _
  "The current Connection state has changed from {0} to {1}.", _
  args.OriginalState, args.CurrentState)
End Sub
```

```
// Assumes connection represents a SqlConnection object.
  connection.StateChange  += new StateChangeEventHandler(OnStateChange);

protected static void OnStateChange(object sender,
  StateChangeEventArgs args)
{
  Console.WriteLine(
    "The current Connection state has changed from {0} to {1}.",
      args.OriginalState, args.CurrentState);
}
```

See also

- Connecting to a Data Source
- ADO.NET Overview

Connection Strings in ADO.NET

2/4/2020 • 2 minutes to read • Edit Online

A connection string contains initialization information that is passed as a parameter from a data provider to a data source. The data provider receives the connection string as the value of the DbConnection.ConnectionString property. The provider parses the connection string and ensures that the syntax is correct and that the keywords are supported. Then the DbConnection.Open() method passes the parsed connection parameters to the data source. The data source performs further validation and establishes a connection.

Connection string syntax

A connection string is a semicolon-delimited list of key/value parameter pairs:

```
keyword1=value; keyword2=value;
```

Keywords are not case-sensitive. Values, however, may be case-sensitive, depending on the data source. Both keywords and values may contain whitespace characters. Leading and trailing white space is ignored in keywords and unquoted values.

If a value contains the semicolon, Unicode control characters, or leading or trailing white space, it must be enclosed in single or double quotation marks. For example:

```
Keyword=" whitespace  ";
Keyword='special;character';
```

The enclosing character may not occur within the value it encloses. Therefore, a value containing single quotation marks can be enclosed only in double quotation marks, and vice versa:

```
Keyword='double"quotation;mark';
Keyword="single'quotation;mark";
```

You can also escape the enclosing character by using two of them together:

```
Keyword="double""quotation";
Keyword='single''quotation';
```

The quotation marks themselves, as well as the equals sign, do not require escaping, so the following connection strings are valid:

```
Keyword=no "escaping" 'required';
Keyword=a=b=c
```

Since each value is read till the next semicolon or the end of string, the value in the latter example is `a=b=c`, and the final semicolon is optional.

All connection strings share the same basic syntax described above. The set of recognized keywords depends on the provider, however, and has evolved over the years from earlier APIs such as *ODBC*. The *.NET Framework* data provider for *SQL Server* (`SqlClient`) supports many keywords from older APIs, but is generally more flexible and

accepts synonyms for many of the common connection string keywords.

Typing mistakes can cause errors. For example, `Integrated Security=true` is valid, but `IntegratedSecurity=true` causes an error.

Connection strings constructed manually at run time from unvalidated user input are vulnerable to string-injection attacks and jeopardize security at the data source. To address these problems, *ADO.NET* 2.0 introduced connection string builders for each *.NET Framework* data provider. These connection string builders expose parameters as strongly-typed properties, and make it possible to validate the connection string before it's sent to the data source.

In This Section

Connection String Builders
Demonstrates how to use the `ConnectionStringBuilder` classes to construct valid connection strings at run time.

Connection Strings and Configuration Files
Demonstrates how to store and retrieve connection strings in configuration files.

Connection String Syntax
Describes how to configure provider-specific connection strings for `SqlClient`, `OracleClient`, `OleDb`, and `Odbc`.

Protecting Connection Information
Demonstrates techniques for protecting information used to connect to a data source.

See also

- Connecting to a Data Source
- ADO.NET Overview

Connection String Builders

3/12/2020 • 3 minutes to read • Edit Online

In earlier versions of ADO.NET, compile-time checking of connection strings with concatenated string values did not occur, so that at run time, an incorrect keyword generated an ArgumentException. Each of the .NET Framework data providers supported different syntax for connection string keywords, which made constructing valid connection strings difficult if done manually. To address this problem, ADO.NET 2.0 introduced new connection string builders for each .NET Framework data provider. Each data provider includes a strongly typed connection string builder class that inherits from DbConnectionStringBuilder. The following table lists the .NET Framework data providers and their associated connection string builder classes.

PROVIDER	CONNECTIONSTRINGBUILDER CLASS
System.Data.SqlClient	System.Data.SqlClient.SqlConnectionStringBuilder
System.Data.OleDb	System.Data.OleDb.OleDbConnectionStringBuilder
System.Data.Odbc	System.Data.Odbc.OdbcConnectionStringBuilder
System.Data.OracleClient	System.Data.OracleClient.OracleConnectionStringBuilder

Connection String Injection Attacks

A connection string injection attack can occur when dynamic string concatenation is used to build connection strings that are based on user input. If the string is not validated and malicious text or characters not escaped, an attacker can potentially access sensitive data or other resources on the server. For example, an attacker could mount an attack by supplying a semicolon and appending an additional value. The connection string is parsed by using a "last one wins" algorithm, and the hostile input is substituted for a legitimate value.

The connection string builder classes are designed to eliminate guesswork and protect against syntax errors and security vulnerabilities. They provide methods and properties corresponding to the known key/value pairs permitted by each data provider. Each class maintains a fixed collection of synonyms and can translate from a synonym to the corresponding well-known key name. Checks are performed for valid key/value pairs and an invalid pair throws an exception. In addition, injected values are handled in a safe manner.

The following example demonstrates how the SqlConnectionStringBuilder handles an inserted extra value for the `Initial Catalog` setting.

```
Dim builder As New System.Data.SqlClient.SqlConnectionStringBuilder
builder("Data Source") = "(local)"
builder("Integrated Security") = True
builder("Initial Catalog") = "AdventureWorks;NewValue=Bad"
Console.WriteLine(builder.ConnectionString)
```

```
System.Data.SqlClient.SqlConnectionStringBuilder builder =
  new System.Data.SqlClient.SqlConnectionStringBuilder();
builder["Data Source"] = "(local)";
builder["integrated Security"] = true;
builder["Initial Catalog"] = "AdventureWorks;NewValue=Bad";
Console.WriteLine(builder.ConnectionString);
```

The output shows that the SqlConnectionStringBuilder handled this correctly by escaping the extra value in double quotation marks instead of appending it to the connection string as a new key/value pair.

```
data source=(local);Integrated Security=True;
initial catalog="AdventureWorks;NewValue=Bad"
```

Building Connection Strings from Configuration Files

If certain elements of a connection string are known beforehand, they can be stored in a configuration file and retrieved at run time to construct a complete connection string. For example, the name of the database might be known in advance, but not the name of the server. Or you might want a user to supply a name and password at run time without being able to inject other values into the connection string.

One of the overloaded constructors for a connection string builder takes a String as an argument, which enables you to supply a partial connection string that can then be completed from user input. The partial connection string can be stored in a configuration file and retrieved at run time.

> **NOTE**
>
> The System.Configuration namespace allows programmatic access to configuration files that use the WebConfigurationManager for Web applications and the ConfigurationManager for Windows applications. For more information about working with connection strings and configuration files, see Connection Strings and Configuration Files.

Example

This example demonstrates retrieving a partial connection string from a configuration file and completing it by setting the DataSource, UserID, and Password properties of the SqlConnectionStringBuilder. The configuration file is defined as follows.

```
<connectionStrings>
  <clear/>
  <add name="partialConnectString"
    connectionString="Initial Catalog=Northwind;"
    providerName="System.Data.SqlClient" />
</connectionStrings>
```

> **NOTE**
>
> You must set a reference to the `System.Configuration.dll` in your project for the code to run.

```
private static void BuildConnectionString(string dataSource,
    string userName, string userPassword)
{
    // Retrieve the partial connection string named databaseConnection
    // from the application's app.config or web.config file.
    ConnectionStringSettings settings =
        ConfigurationManager.ConnectionStrings["partialConnectString"];

    if (null != settings)
    {
        // Retrieve the partial connection string.
        string connectString = settings.ConnectionString;
        Console.WriteLine("Original: {0}", connectString);

        // Create a new SqlConnectionStringBuilder based on the
        // partial connection string retrieved from the config file.
        SqlConnectionStringBuilder builder =
            new SqlConnectionStringBuilder(connectString);

        // Supply the additional values.
        builder.DataSource = dataSource;
        builder.UserID = userName;
        builder.Password = userPassword;
        Console.WriteLine("Modified: {0}", builder.ConnectionString);
    }
}
```

```
Private Sub BuildConnectionString(ByVal dataSource As String, _
    ByVal userName As String, ByVal userPassword As String)

    ' Retrieve the partial connection string named databaseConnection
    ' from the application's app.config or web.config file.
    Dim settings As ConnectionStringSettings = _
       ConfigurationManager.ConnectionStrings("partialConnectString")

    If Not settings Is Nothing Then
        ' Retrieve the partial connection string.
        Dim connectString As String = settings.ConnectionString
        Console.WriteLine("Original: {0}", connectString)

        ' Create a new SqlConnectionStringBuilder based on the
        ' partial connection string retrieved from the config file.
        Dim builder As New SqlConnectionStringBuilder(connectString)

        ' Supply the additional values.
        builder.DataSource = dataSource
        builder.UserID = userName
        builder.Password = userPassword

        Console.WriteLine("Modified: {0}", builder.ConnectionString)
    End If
End Sub
```

See also

- Connection Strings
- Privacy and Data Security
- ADO.NET Overview

Connection Strings and Configuration Files

3/27/2020 • 12 minutes to read • Edit Online

Embedding connection strings in your application's code can lead to security vulnerabilities and maintenance problems. Unencrypted connection strings compiled into an application's source code can be viewed using the Ildasm.exe (IL Disassembler) tool. Moreover, if the connection string ever changes, your application must be recompiled. For these reasons, we recommend storing connection strings in an application configuration file.

Working with Application Configuration Files

Application configuration files contain settings that are specific to a particular application. For example, an ASP.NET application can have one or more **web.config** files, and a Windows application can have an optional **app.config** file. Configuration files share common elements, although the name and location of a configuration file vary depending on the application's host.

The connectionStrings Section

Connection strings can be stored as key/value pairs in the **connectionStrings** section of the **configuration** element of an application configuration file. Child elements include **add**, **clear**, and **remove**.

The following configuration file fragment demonstrates the schema and syntax for storing a connection string. The **name** attribute is a name that you provide to uniquely identify a connection string so that it can be retrieved at run time. The **providerName** is the invariant name of the .NET Framework data provider, which is registered in the machine.config file.

```
<?xml version='1.0' encoding='utf-8'?>
  <configuration>
    <connectionStrings>
      <clear />
      <add name="Name"
       providerName="System.Data.ProviderName"
       connectionString="Valid Connection String;" />
    </connectionStrings>
  </configuration>
```

NOTE

You can save part of a connection string in a configuration file and use the DbConnectionStringBuilder class to complete it at run time. This is useful in scenarios where you do not know elements of the connection string ahead of time, or when you do not want to save sensitive information in a configuration file. For more information, see Connection String Builders.

Using External Configuration Files

External configuration files are separate files that contain a fragment of a configuration file consisting of a single section. The external configuration file is then referenced by the main configuration file. Storing the **connectionStrings** section in a physically separate file is useful in situations where connection strings may be edited after the application is deployed. For example, the standard ASP.NET behavior is to restart an application domain when configuration files are modified, which results in state information being lost. However, modifying an external configuration file does not cause an application restart. External configuration files are not limited to ASP.NET; they can also be used by Windows applications. In addition, file access security and permissions can be used to restrict access to external configuration files. Working with external configuration files at run time is transparent, and requires no special coding.

To store connection strings in an external configuration file, create a separate file that contains only the **connectionStrings** section. Do not include any additional elements, sections, or attributes. This example shows the syntax for an external configuration file.

```
<connectionStrings>
  <add name="Name"
   providerName="System.Data.ProviderName"
   connectionString="Valid Connection String;" />
</connectionStrings>
```

In the main application configuration file, you use the **configSource** attribute to specify the fully qualified name and location of the external file. This example refers to an external configuration file named `connections.config`.

```
<?xml version='1.0' encoding='utf-8'?>
<configuration>
    <connectionStrings configSource="connections.config"/>
</configuration>
```

Retrieving Connection Strings at Run Time

The .NET Framework 2.0 introduced new classes in the System.Configuration namespace to simplify retrieving connection strings from configuration files at run time. You can programmatically retrieve a connection string by name or by provider name.

> **NOTE**
>
> The **machine.config** file also contains a **connectionStrings** section, which contains connection strings used by Visual Studio. When retrieving connection strings by provider name from the **app.config** file in a Windows application, the connection strings in **machine.config** get loaded first, and then the entries from **app.config**. Adding **clear** immediately after the **connectionStrings** element removes all inherited references from the data structure in memory, so that only the connection strings defined in the local **app.config** file are considered.

Working with the Configuration Classes

Starting with the .NET Framework 2.0, ConfigurationManager is used when working with configuration files on the local computer, replacing the deprecated ConfigurationSettings. WebConfigurationManager is used to work with ASP.NET configuration files. It is designed to work with configuration files on a Web server, and allows programmatic access to configuration file sections such as **system.web**.

> **NOTE**
>
> Accessing configuration files at run time requires granting permissions to the caller; the required permissions depend on the type of application, configuration file, and location. For more information, see Using the Configuration Classes and WebConfigurationManager for ASP.NET applications, and ConfigurationManager for Windows applications.

You can use the ConnectionStringSettingsCollection to retrieve connection strings from application configuration files. It contains a collection of ConnectionStringSettings objects, each of which represents a single entry in the **connectionStrings** section. Its properties map to connection string attributes, allowing you to retrieve a connection string by specifying the name or the provider name.

PROPERTY	DESCRIPTION

PROPERTY	DESCRIPTION
Name	The name of the connection string. Maps to the **name** attribute.
ProviderName	The fully qualified provider name. Maps to the **providerName** attribute.
ConnectionString	The connection string. Maps to the **connectionString** attribute.

Example: Listing All Connection Strings

This example iterates through the ConnectionStringSettingsCollection and displays the ConnectionStringSettings.Name, ConnectionStringSettings.ProviderName, and ConnectionStringSettings.ConnectionString properties in the console window.

> **NOTE**
>
> System.Configuration.dll is not included in all project types, and you may need to set a reference to it in order to use the configuration classes. The name and location of a particular application configuration file varies by the type of application and the hosting process.

```
using System.Configuration;

class Program
{
    static void Main()
    {
        GetConnectionStrings();
        Console.ReadLine();
    }

    static void GetConnectionStrings()
    {
        ConnectionStringSettingsCollection settings =
            ConfigurationManager.ConnectionStrings;

        if (settings != null)
        {
            foreach(ConnectionStringSettings cs in settings)
            {
                Console.WriteLine(cs.Name);
                Console.WriteLine(cs.ProviderName);
                Console.WriteLine(cs.ConnectionString);
            }
        }
    }
}
```

```
Imports System.Configuration

Class Program
    Shared Sub Main()
        GetConnectionStrings()
        Console.ReadLine()
    End Sub

    Private Shared Sub GetConnectionStrings()

        Dim settings As ConnectionStringSettingsCollection = _
            ConfigurationManager.ConnectionStrings

        If Not settings Is Nothing Then
            For Each cs As ConnectionStringSettings In settings
                Console.WriteLine(cs.Name)
                Console.WriteLine(cs.ProviderName)
                Console.WriteLine(cs.ConnectionString)
            Next
        End If
    End Sub
End Class
```

Example: Retrieving a Connection String by Name

This example demonstrates how to retrieve a connection string from a configuration file by specifying its name. The code creates a ConnectionStringSettings object, matching the supplied input parameter to the ConnectionStrings name. If no matching name is found, the function returns `null` (`Nothing` in Visual Basic).

```
// Retrieves a connection string by name.
// Returns null if the name is not found.
static string GetConnectionStringByName(string name)
{
    // Assume failure.
    string returnValue = null;

    // Look for the name in the connectionStrings section.
    ConnectionStringSettings settings =
        ConfigurationManager.ConnectionStrings[name];

    // If found, return the connection string.
    if (settings != null)
        returnValue = settings.ConnectionString;

    return returnValue;
}
```

```
' Retrieves a connection string by name.
' Returns Nothing if the name is not found.
Private Shared Function GetConnectionStringByName( _
    ByVal name As String) As String

    ' Assume failure
    Dim returnValue As String = Nothing

    ' Look for the name in the connectionStrings section.
    Dim settings As ConnectionStringSettings = _
       ConfigurationManager.ConnectionStrings(name)

    ' If found, return the connection string.
    If Not settings Is Nothing Then
        returnValue = settings.ConnectionString
    End If

    Return returnValue
End Function
```

Example: Retrieving a Connection String by Provider Name

This example demonstrates how to retrieve a connection string by specifying the provider-invariant name in the format *System.Data.ProviderName*. The code iterates through the ConnectionStringSettingsCollection and returns the connection string for the first ProviderName found. If the provider name is not found, the function returns `null` (`Nothing` in Visual Basic).

```
// Retrieve a connection string by specifying the providerName.
// Assumes one connection string per provider in the config file.
static string GetConnectionStringByProvider(string providerName)
{
    // Return null on failure.
    string returnValue = null;

    // Get the collection of connection strings.
    ConnectionStringSettingsCollection settings =
        ConfigurationManager.ConnectionStrings;

    // Walk through the collection and return the first
    // connection string matching the providerName.
    if (settings != null)
    {
        foreach (ConnectionStringSettings cs in settings)
        {
            if (cs.ProviderName == providerName)
                returnValue = cs.ConnectionString;
            break;
        }
    }
    return returnValue;
}
```

```
' Retrieve a connection string by specifying the providerName.
' Assumes one connection string per provider in the config file.
Private Shared Function GetConnectionStringByProvider( _
    ByVal providerName As String) As String

    'Return Nothing on failure.
    Dim returnValue As String = Nothing

    ' Get the collection of connection strings.
    Dim settings As ConnectionStringSettingsCollection = _
        ConfigurationManager.ConnectionStrings

    ' Walk through the collection and return the first
    ' connection string matching the providerName.
    If Not settings Is Nothing Then
        For Each cs As ConnectionStringSettings In settings
            If cs.ProviderName = providerName Then
                returnValue = cs.ConnectionString
                Exit For
            End If
        Next
    End If

    Return returnValue
End Function
```

Encrypting Configuration File Sections Using Protected Configuration

ASP.NET 2.0 introduced a new feature, called *protected configuration*, that enables you to encrypt sensitive information in a configuration file. Although primarily designed for ASP.NET, protected configuration can also be used to encrypt configuration file sections in Windows applications. For a detailed description of the protected configuration capabilities, see Encrypting Configuration Information Using Protected Configuration.

The following configuration file fragment shows the **connectionStrings** section after it has been encrypted. The **configProtectionProvider** specifies the protected configuration provider used to encrypt and decrypt the connection strings. The **EncryptedData** section contains the cipher text.

```
<connectionStrings configProtectionProvider="DataProtectionConfigurationProvider">
  <EncryptedData>
    <CipherData>
      <CipherValue>AQAAANCMnd8BFdERjHoAwE/Cl+sBAAAAH2... </CipherValue>
    </CipherData>
  </EncryptedData>
</connectionStrings>
```

When the encrypted connection string is retrieved at run time, the .NET Framework uses the specified provider to decrypt the **CipherValue** and make it available to your application. You do not need to write any additional code to manage the decryption process.

Protected Configuration Providers

Protected configuration providers are registered in the **configProtectedData** section of the **machine.config** file on the local computer, as shown in the following fragment, which shows the two protected configuration providers supplied with the .NET Framework. The values shown here have been truncated for readability.

```
<configProtectedData defaultProvider="RsaProtectedConfigurationProvider">
  <providers>
    <add name="RsaProtectedConfigurationProvider"
      type="System.Configuration.RsaProtectedConfigurationProvider" />
    <add name="DataProtectionConfigurationProvider"
      type="System.Configuration.DpapiProtectedConfigurationProvider" />
  </providers>
</configProtectedData>
```

You can configure additional protected configuration providers by adding them to the **machine.config** file. You can also create your own protected configuration provider by inheriting from the ProtectedConfigurationProvider abstract base class. The following table describes the two configuration files included with the .NET Framework.

PROVIDER	DESCRIPTION
RsaProtectedConfigurationProvider	Uses the RSA encryption algorithm to encrypt and decrypt data. The RSA algorithm can be used for both public key encryption and digital signatures. It is also known as "public key" or asymmetrical encryption because it employs two different keys. You can use the ASP.NET IIS Registration Tool (Aspnet_regiis.exe) to encrypt sections in a Web.config file and manage the encryption keys. ASP.NET decrypts the configuration file when it processes the file. The identity of the ASP.NET application must have read access to the encryption key that is used to encrypt and decrypt the encrypted sections.
DpapiProtectedConfigurationProvider	Uses the Windows Data Protection API (DPAPI) to encrypt configuration sections. It uses the Windows built-in cryptographic services and can be configured for either machine-specific or user-account-specific protection. Machine-specific protection is useful for multiple applications on the same server that need to share information. User-account-specific protection can be used with services that run with a specific user identity, such as a shared hosting environment. Each application runs under a separate identity which restricts access to resources such as files and databases.

Both providers offer strong encryption of data. However, if you are planning to use the same encrypted configuration file on multiple servers, such as a Web farm, only the RsaProtectedConfigurationProvider enables you to export the encryption keys used to encrypt the data and import them on another server. For more information, see Importing and Exporting Protected Configuration RSA Key Containers.

Using the Configuration Classes

The System.Configuration namespace provides classes to work with configuration settings programmatically. The ConfigurationManager class provides access to machine, application, and user configuration files. If you are creating an ASP.NET application, you can use the WebConfigurationManager class, which provides the same functionality while also allowing you to access settings that are unique to ASP.NET applications, such as those found in **<system.web>**.

> **NOTE**
>
> The System.Security.Cryptography namespace contains classes that provide additional options for encrypting and decrypting data. Use these classes if you require cryptographic services that are not available using protected configuration. Some of these classes are wrappers for the unmanaged Microsoft CryptoAPI, while others are purely managed implementations. For more information, see Cryptographic Services.

App.config Example

This example demonstrates how to toggle encrypting the **connectionStrings** section in an **app.config** file for a Windows application. In this example, the procedure takes the name of the application as an argument, for example, "MyApplication.exe". The **app.config** file will then be encrypted and copied to the folder that contains the executable under the name of "MyApplication.exe.config".

> **NOTE**
> The connection string can only be decrypted on the computer on which it was encrypted.

The code uses the OpenExeConfiguration method to open the **app.config** file for editing, and the GetSection method returns the **connectionStrings** section. The code then checks the IsProtected property, calling the ProtectSection to encrypt the section if it is not encrypted. The UnprotectSection method is invoked to decrypt the section. The Save method completes the operation and saves the changes.

> **NOTE**
> You must set a reference to `System.Configuration.dll` in your project for the code to run.

```
static void ToggleConfigEncryption(string exeFile)
{
    // Takes the executable file name without the
    // .config extension.
    try
    {
        // Open the configuration file and retrieve
        // the connectionStrings section.
        Configuration config = ConfigurationManager.
            OpenExeConfiguration(exeConfigName);

        ConnectionStringsSection section =
            config.GetSection("connectionStrings")
            as ConnectionStringsSection;

        if (section.SectionInformation.IsProtected)
        {
            // Remove encryption.
            section.SectionInformation.UnprotectSection();
        }
        else
        {
            // Encrypt the section.
            section.SectionInformation.ProtectSection(
                "DataProtectionConfigurationProvider");
        }
        // Save the current configuration.
        config.Save();

        Console.WriteLine("Protected={0}",
            section.SectionInformation.IsProtected);
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
}
```

```
Shared Sub ToggleConfigEncryption(ByVal exeConfigName As String)
    ' Takes the executable file name without the
    ' .config extension.
    Try
        ' Open the configuration file and retrieve
        ' the connectionStrings section.
        Dim config As Configuration = ConfigurationManager. _
            OpenExeConfiguration(exeConfigName)

        Dim section As ConnectionStringsSection = DirectCast( _
            config.GetSection("connectionStrings"), _
            ConnectionStringsSection)

        If section.SectionInformation.IsProtected Then
            ' Remove encryption.
            section.SectionInformation.UnprotectSection()
        Else
            ' Encrypt the section.
            section.SectionInformation.ProtectSection( _
              "DataProtectionConfigurationProvider")
        End If

        ' Save the current configuration.
        config.Save()

        Console.WriteLine("Protected={0}", _
        section.SectionInformation.IsProtected)

    Catch ex As Exception
        Console.WriteLine(ex.Message)
    End Try
End Sub
```

Web.config Example

This example uses the OpenWebConfiguration method of the `WebConfigurationManager`. Note that in this case you can supply the relative path to the **Web.config** file by using a tilde. The code requires a reference to the `System.Web.Configuration` class.

```
static void ToggleWebEncrypt()
{
    // Open the Web.config file.
    Configuration config = WebConfigurationManager.
        OpenWebConfiguration("~");

    // Get the connectionStrings section.
    ConnectionStringsSection section =
        config.GetSection("connectionStrings")
        as ConnectionStringsSection;

    // Toggle encryption.
    if (section.SectionInformation.IsProtected)
    {
        section.SectionInformation.UnprotectSection();
    }
    else
    {
        section.SectionInformation.ProtectSection(
            "DataProtectionConfigurationProvider");
    }

    // Save changes to the Web.config file.
    config.Save();
}
```

```
Shared Sub ToggleWebEncrypt()
    ' Open the Web.config file.
    Dim config As Configuration = WebConfigurationManager. _
      OpenWebConfiguration("~")

    ' Get the connectionStrings section.
    Dim section As ConnectionStringsSection = DirectCast( _
        config.GetSection("connectionStrings"), _
        ConnectionStringsSection)

    ' Toggle encryption.
    If section.SectionInformation.IsProtected Then
        section.SectionInformation.UnprotectSection()
    Else
        section.SectionInformation.ProtectSection( _
          "DataProtectionConfigurationProvider")
    End If

    ' Save changes to the Web.config file.
    config.Save()
End Sub
```

For more information about securing ASP.NET applications, see Securing ASP.NET web sites.

See also

- Connection String Builders
- Protecting Connection Information
- Using the Configuration Classes
- Configuring Apps
- ASP.NET Web Site Administration
- ADO.NET Overview

Connection String Syntax

3/12/2020 • 8 minutes to read • Edit Online

Each .NET Framework data provider has a `Connection` object that inherits from DbConnection as well as a provider-specific ConnectionString property. The specific connection string syntax for each provider is documented in its `ConnectionString` property. The following table lists the four data providers that are included in the .NET Framework.

.NET FRAMEWORK DATA PROVIDER	DESCRIPTION
System.Data.SqlClient	Provides data access for Microsoft SQL Server. For more information on connection string syntax, see ConnectionString.
System.Data.OleDb	Provides data access for data sources exposed using OLE DB. For more information on connection string syntax, see ConnectionString.
System.Data.Odbc	Provides data access for data sources exposed using ODBC. For more information on connection string syntax, see ConnectionString.
System.Data.OracleClient	Provides data access for Oracle version 8.1.7 or later. For more information on connection string syntax, see ConnectionString.

Connection String Builders

ADO.NET 2.0 introduced the following connection string builders for the .NET Framework data providers.

- SqlConnectionStringBuilder
- OleDbConnectionStringBuilder
- OdbcConnectionStringBuilder
- OracleConnectionStringBuilder

The connection string builders allow you to construct syntactically valid connection strings at run time, so you do not have to manually concatenate connection string values in your code. For more information, see Connection String Builders.

Windows Authentication

We recommend using Windows Authentication (sometimes referred to as *integrated security*) to connect to data sources that support it. The syntax employed in the connection string varies by provider. The following table shows the Windows Authentication syntax used with the .NET Framework data providers.

PROVIDER	SYNTAX

PROVIDER	SYNTAX

`SqlClient`	`Integrated Security=true;`
	`-- or --`
	`Integrated Security=SSPI;`

`OleDb`	`Integrated Security=SSPI;`
`Odbc`	`Trusted_Connection=yes;`
`OracleClient`	`Integrated Security=yes;`

NOTE

`Integrated Security=true` throws an exception when used with the `OleDb` provider.

SqlClient Connection Strings

The syntax for a SqlConnection connection string is documented in the SqlConnection.ConnectionString property. You can use the ConnectionString property to get or set a connection string for a SQL Server database. If you need to connect to an earlier version of SQL Server, you must use the .NET Framework Data Provider for OleDb (System.Data.OleDb). Most connection string keywords also map to properties in the SqlConnectionStringBuilder.

IMPORTANT

The default setting for the `Persist Security Info` keyword is `false`. Setting it to `true` or `yes` allows security-sensitive information, including the user ID and password, to be obtained from the connection after the connection has been opened. Keep `Persist Security Info` set to `false` to ensure that an untrusted source does not have access to sensitive connection string information.

Windows authentication with SqlClient

Each of the following forms of syntax uses Windows Authentication to connect to the **AdventureWorks** database on a local server.

```
"Persist Security Info=False;Integrated Security=true;
    Initial Catalog=AdventureWorks;Server=MSSQL1"
"Persist Security Info=False;Integrated Security=SSPI;
    database=AdventureWorks;server=(local)"
"Persist Security Info=False;Trusted_Connection=True;
    database=AdventureWorks;server=(local)"
```

SQL Server authentication with SqlClient

Windows Authentication is preferred for connecting to SQL Server. However, if SQL Server Authentication is required, use the following syntax to specify a user name and password. In this example, asterisks are used to represent a valid user name and password.

```
"Persist Security Info=False;User ID=*****;Password=*****;Initial Catalog=AdventureWorks;Server=MySqlServer"
```

When you connect to Azure SQL Database or to Azure SQL Data Warehouse and provide a login in the format

`user@servername`, make sure that the `servername` value in the login matches the value provided for `Server=`.

> **NOTE**
> Windows authentication takes precedence over SQL Server logins. If you specify both Integrated Security=true as well as a user name and password, the user name and password will be ignored and Windows authentication will be used.

Connect to a named instance of SQL Server

To connect to a named instance of SQL Server, use the *server name**instance name* syntax.

```
"Data Source=MySqlServer\MSSQL1;"
```

You can also set the DataSource property of the `SqlConnectionStringBuilder` to the instance name when building a connection string. The DataSource property of a SqlConnection object is read-only.

Type System Version Changes

The `Type System Version` keyword in a SqlConnection.ConnectionString specifies the client-side representation of SQL Server types. See SqlConnection.ConnectionString for more information about the `Type System Version` keyword.

Connecting and Attaching to SQL Server Express User Instances

User instances are a feature in SQL Server Express. They allow a user running on a least-privileged local Windows account to attach and run a SQL Server database without requiring administrative privileges. A user instance executes with the user's Windows credentials, not as a service.

For more information on working with user instances, see SQL Server Express User Instances.

Using TrustServerCertificate

The `TrustServerCertificate` keyword is valid only when connecting to a SQL Server instance with a valid certificate. When `TrustServerCertificate` is set to `true`, the transport layer will use SSL to encrypt the channel and bypass walking the certificate chain to validate trust.

```
"TrustServerCertificate=true;"
```

> **NOTE**
> If `TrustServerCertificate` is set to `true` and encryption is turned on, the encryption level specified on the server will be used even if `Encrypt` is set to `false` in the connection string. The connection will fail otherwise.

Enabling Encryption

To enable encryption when a certificate has not been provisioned on the server, the **Force Protocol Encryption** and the **Trust Server Certificate** options must be set in SQL Server Configuration Manager. In this case, encryption will use a self-signed server certificate without validation if no verifiable certificate has been provisioned on the server.

Application settings cannot reduce the level of security configured in SQL Server, but can optionally strengthen it. An application can request encryption by setting the `TrustServerCertificate` and `Encrypt` keywords to `true`, guaranteeing that encryption takes place even when a server certificate has not been provisioned and **Force Protocol Encryption** has not been configured for the client. However, if `TrustServerCertificate` is not enabled in

the client configuration, a provisioned server certificate is still required.

The following table describes all cases.

FORCE PROTOCOL ENCRYPTION CLIENT SETTING	TRUST SERVER CERTIFICATE CLIENT SETTING	ENCRYPT/USE ENCRYPTION FOR DATA CONNECTION STRING/ATTRIBUTE	TRUST SERVER CERTIFICATE CONNECTION STRING/ATTRIBUTE	RESULT
No	N/A	No (default)	Ignored	No encryption occurs.
No	N/A	Yes	No (default)	Encryption occurs only if there is a verifiable server certificate, otherwise the connection attempt fails.
No	N/A	Yes	Yes	Encryption always occurs, but may use a self-signed server certificate.
Yes	No	Ignored	Ignored	Encryption occurs only if there is a verifiable server certificate; otherwise, the connection attempt fails.
Yes	Yes	No (default)	Ignored	Encryption always occurs, but may use a self-signed server certificate.
Yes	Yes	Yes	No (default)	Encryption occurs only if there is a verifiable server certificate; otherwise, the connection attempt fails.
Yes	Yes	Yes	Yes	Encryption always occurs, but may use a self-signed server certificate.

For more information, see Using Encryption Without Validation.

OleDb Connection Strings

The ConnectionString property of a OleDbConnection allows you to get or set a connection string for an OLE DB data source, such as Microsoft Access. You can also create an `OleDb` connection string at run time by using the OleDbConnectionStringBuilder class.

OleDb Connection String Syntax

You must specify a provider name for an OleDbConnection connection string. The following connection string connects to a Microsoft Access database using the Jet provider. Note that the `User ID` and `Password` keywords are optional if the database is unsecured (the default).

```
Provider=Microsoft.Jet.OLEDB.4.0; Data Source=d:\Northwind.mdb;User ID=Admin;Password=;
```

If the Jet database is secured using user-level security, you must provide the location of the workgroup information file (.mdw). The workgroup information file is used to validate the credentials presented in the connection string.

```
Provider=Microsoft.Jet.OLEDB.4.0;Data Source=d:\Northwind.mdb;Jet OLEDB:System
Database=d:\NorthwindSystem.mdw;User ID=*****;Password=*****;
```

> **IMPORTANT**
> It is possible to supply connection information for an **OleDbConnection** in a Universal Data Link (UDL) file; however you should avoid doing so. UDL files are not encrypted, and expose connection string information in clear text. Because a UDL file is an external file-based resource to your application, it cannot be secured using the .NET Framework. UDL files are not supported for **SqlClient**.

Using DataDirectory to Connect to Access/Jet

`DataDirectory` is not exclusive to `SqlClient`. It can also be used with the System.Data.OleDb and System.Data.Odbc .NET data providers. The following sample OleDbConnection string demonstrates the syntax required to connect to the Northwind.mdb located in the application's app_data folder. The system database (System.mdw) is also stored in that location.

```
"Provider=Microsoft.Jet.OLEDB.4.0;
Data Source=|DataDirectory|\Northwind.mdb;
Jet OLEDB:System Database=|DataDirectory|\System.mdw;"
```

> **IMPORTANT**
> Specifying the location of the system database in the connection string is not required if the Access/Jet database is unsecured. Security is off by default, with all users connecting as the built-in Admin user with a blank password. Even when user-level security is correctly implemented, a Jet database remains vulnerable to attack. Therefore, storing sensitive information in an Access/Jet database is not recommended because of the inherent weakness of its file-based security scheme.

Connecting to Excel

The Microsoft Jet provider is used to connect to an Excel workbook. In the following connection string, the `Extended Properties` keyword sets properties that are specific to Excel. "HDR=Yes;" indicates that the first row contains column names, not data, and "IMEX=1;" tells the driver to always read "intermixed" data columns as text.

```
Provider=Microsoft.Jet.OLEDB.4.0;Data Source=D:\MyExcel.xls;Extended Properties=""Excel 8.0;HDR=Yes;IMEX=1""
```

Note that the double quotation character required for the `Extended Properties` must also be enclosed in double quotation marks.

Data Shape Provider Connection String Syntax

Use both the `Provider` and the `Data Provider` keywords when using the Microsoft Data Shape provider. The following example uses the Shape provider to connect to a local instance of SQL Server.

```
"Provider=MSDataShape;Data Provider=SQLOLEDB;Data Source=(local);Initial Catalog=pubs;Integrated
Security=SSPI;"
```

Odbc Connection Strings

The ConnectionString property of a OdbcConnection allows you to get or set a connection string for an OLE DB data source. Odbc connection strings are also supported by the OdbcConnectionStringBuilder.

The following connection string uses the Microsoft Text Driver.

```
Driver={Microsoft Text Driver (*.txt; *.csv)};DBQ=d:\bin
```

Using DataDirectory to Connect to Visual FoxPro

The following OdbcConnection connection string sample demonstrates using `DataDirectory` to connect to a Microsoft Visual FoxPro file.

```
"Driver={Microsoft Visual FoxPro Driver};
SourceDB=|DataDirectory|\MyData.DBC;SourceType=DBC;"
```

Oracle Connection Strings

The ConnectionString property of a OracleConnection allows you to get or set a connection string for an OLE DB data source. Oracle connection strings are also supported by the OracleConnectionStringBuilder .

```
Data Source=Oracle9i;User ID=*****;Password=*****;
```

For more information on ODBC connection string syntax, see ConnectionString.

See also

- Connection Strings
- Connecting to a Data Source
- ADO.NET Overview

Protecting Connection Information

3/12/2020 • 3 minutes to read • Edit Online

Protecting access to your data source is one of the most important goals when securing an application. A connection string presents a potential vulnerability if it is not secured. Storing connection information in plain text or persisting it in memory risks compromising your entire system. Connection strings embedded in your source code can be read using the Ildasm.exe (IL Disassembler) to view Microsoft intermediate language (MSIL) in a compiled assembly.

Security vulnerabilities involving connection strings can arise based on the type of authentication used, how connection strings are persisted in memory and on disk, and the techniques used to construct them at run time.

Use Windows Authentication

To help limit access to your data source, you must secure connection information such as user ID, password, and data source name. In order to avoid exposing user information, we recommend using Windows authentication (sometimes referred to as *integrated security*) wherever possible. Windows authentication is specified in a connection string by using the `Integrated Security` or `Trusted_Connection` keywords, eliminating the need to use a user ID and password. When using Windows authentication, users are authenticated by Windows, and access to server and database resources is determined by granting permissions to Windows users and groups.

For situations where it is not possible to use Windows authentication, you must use extra care because user credentials are exposed in the connection string. In an ASP.NET application, you can configure a Windows account as a fixed identity that is used to connect to databases and other network resources. You enable impersonation in the identity element in the **web.config** file and specify a user name and password.

```
<identity impersonate="true"
        userName="MyDomain\UserAccount"
        password="*****" />
```

The fixed identity account should be a low-privilege account that has been granted only necessary permissions in the database. In addition, you should encrypt the configuration file so that the user name and password are not exposed in clear text.

Do Not Use Universal Data Link (UDL) files

Avoid storing connection strings for an OleDbConnection in a Universal Data Link (UDL) file. UDLs are stored in clear text and cannot be encrypted. A UDL file is an external file-based resource to your application, and it cannot be secured or encrypted using the .NET Framework.

Avoid Injection Attacks with Connection String Builders

A connection string injection attack can occur when dynamic string concatenation is used to build connection strings based on user input. If the user input is not validated and malicious text or characters not escaped, an attacker can potentially access sensitive data or other resources on the server. To address this problem, ADO.NET 2.0 introduced new connection string builder classes to validate connection string syntax and ensure that additional parameters are not introduced. For more information, see Connection String Builders.

Use Persist Security Info=False

The default value for `Persist Security Info` is false; we recommend using this default in all connection strings. Setting `Persist Security Info` to `true` or `yes` allows security-sensitive information, including the user ID and password, to be obtained from a connection after it has been opened. When `Persist Security Info` is set to `false` or `no`, security information is discarded after it is used to open the connection, ensuring that an untrusted source does not have access to security-sensitive information.

Encrypt Configuration Files

You can also store connection strings in configuration files, which eliminates the need to embed them in your application's code. Configuration files are standard XML files for which the .NET Framework has defined a common set of elements. Connection strings in configuration files are typically stored inside the **<connectionStrings>** element in the **app.config** for a Windows application, or the **web.config** file for an ASP.NET application. For more information on the basics of storing, retrieving and encrypting connection strings from configuration files, see Connection Strings and Configuration Files.

See also

- Securing ADO.NET Applications
- Encrypting Configuration Information Using Protected Configuration
- Security in .NET
- ADO.NET Overview

Connection Pooling

9/7/2019 • 2 minutes to read • Edit Online

Connecting to a data source can be time consuming. To minimize the cost of opening connections, ADO.NET uses an optimization technique called *connection pooling*, which minimizes the cost of repeatedly opening and closing connections. Connection pooling is handled differently for the .NET Framework data providers.

In This Section

SQL Server Connection Pooling (ADO.NET)
Provides an overview of connection pooling and describes how connection pooling works in SQL Server.

OLE DB, ODBC, and Oracle Connection Pooling
Describes connection pooling for the .NET Framework Data Provider for OLE DB, the .NET Framework Data Provider for ODBC, and the .NET Framework Data Provider for Oracle.

See also

- Retrieving and Modifying Data in ADO.NET
- ADO.NET Overview

SQL Server Connection Pooling (ADO.NET)

3/12/2020 • 10 minutes to read • Edit Online

Connecting to a database server typically consists of several time-consuming steps. A physical channel such as a socket or a named pipe must be established, the initial handshake with the server must occur, the connection string information must be parsed, the connection must be authenticated by the server, checks must be run for enlisting in the current transaction, and so on.

In practice, most applications use only one or a few different configurations for connections. This means that during application execution, many identical connections will be repeatedly opened and closed. To minimize the cost of opening connections, ADO.NET uses an optimization technique called *connection pooling*.

Connection pooling reduces the number of times that new connections must be opened. The *pooler* maintains ownership of the physical connection. It manages connections by keeping alive a set of active connections for each given connection configuration. Whenever a user calls `Open` on a connection, the pooler looks for an available connection in the pool. If a pooled connection is available, it returns it to the caller instead of opening a new connection. When the application calls `Close` on the connection, the pooler returns it to the pooled set of active connections instead of closing it. Once the connection is returned to the pool, it is ready to be reused on the next `Open` call.

Only connections with the same configuration can be pooled. ADO.NET keeps several pools at the same time, one for each configuration. Connections are separated into pools by connection string, and by Windows identity when integrated security is used. Connections are also pooled based on whether they are enlisted in a transaction. When using ChangePassword, the SqlCredential instance affects the connection pool. Different instances of SqlCredential will use different connection pools, even if the user ID and password are the same.

Pooling connections can significantly enhance the performance and scalability of your application. By default, connection pooling is enabled in ADO.NET. Unless you explicitly disable it, the pooler optimizes the connections as they are opened and closed in your application. You can also supply several connection string modifiers to control connection pooling behavior. For more information, see "Controlling Connection Pooling with Connection String Keywords" later in this topic.

> **NOTE**
>
> When connection pooling is enabled, and if a timeout error or other login error occurs, an exception will be thrown and subsequent connection attempts will fail for the next five seconds, the "blocking period". If the application attempts to connect within the blocking period, the first exception will be thrown again. Subsequent failures after a blocking period ends will result in a new blocking periods that is twice as long as the previous blocking period, up to a maximum of one minute.

Pool Creation and Assignment

When a connection is first opened, a connection pool is created based on an exact matching algorithm that associates the pool with the connection string in the connection. Each connection pool is associated with a distinct connection string. When a new connection is opened, if the connection string is not an exact match to an existing pool, a new pool is created. Connections are pooled per process, per application domain, per connection string and when integrated security is used, per Windows identity. Connection strings must also be an exact match; keywords supplied in a different order for the same connection will be pooled separately.

In the following C# example, three new SqlConnection objects are created, but only two connection pools are required to manage them. Note that the first and second connection strings differ by the value assigned for

`Initial Catalog`.

```
using (SqlConnection connection = new SqlConnection(
  "Integrated Security=SSPI;Initial Catalog=Northwind"))
    {
        connection.Open();
        // Pool A is created.
    }

using (SqlConnection connection = new SqlConnection(
  "Integrated Security=SSPI;Initial Catalog=pubs"))
    {
        connection.Open();
        // Pool B is created because the connection strings differ.
    }

using (SqlConnection connection = new SqlConnection(
  "Integrated Security=SSPI;Initial Catalog=Northwind"))
    {
        connection.Open();
        // The connection string matches pool A.
    }
```

If `MinPoolSize` is either not specified in the connection string or is specified as zero, the connections in the pool will be closed after a period of inactivity. However, if the specified `MinPoolSize` is greater than zero, the connection pool is not destroyed until the `AppDomain` is unloaded and the process ends. Maintenance of inactive or empty pools involves minimal system overhead.

> **NOTE**
> The pool is automatically cleared when a fatal error occurs, such as a failover.

Adding Connections

A connection pool is created for each unique connection string. When a pool is created, multiple connection objects are created and added to the pool so that the minimum pool size requirement is satisfied. Connections are added to the pool as needed, up to the maximum pool size specified (100 is the default). Connections are released back into the pool when they are closed or disposed.

When a SqlConnection object is requested, it is obtained from the pool if a usable connection is available. To be usable, a connection must be unused, have a matching transaction context or be unassociated with any transaction context, and have a valid link to the server.

The connection pooler satisfies requests for connections by reallocating connections as they are released back into the pool. If the maximum pool size has been reached and no usable connection is available, the request is queued. The pooler then tries to reclaim any connections until the time-out is reached (the default is 15 seconds). If the pooler cannot satisfy the request before the connection times out, an exception is thrown.

Caution

We strongly recommend that you always close the connection when you are finished using it so that the connection will be returned to the pool. You can do this using either the `Close` or `Dispose` methods of the `Connection` object, or by opening all connections inside a `using` statement in C#, or a `Using` statement in Visual Basic. Connections that are not explicitly closed might not be added or returned to the pool. For more information, see using Statement or How to: Dispose of a System Resource for Visual Basic.

> **NOTE**
>
> Do not call `Close` or `Dispose` on a `Connection`, a `DataReader`, or any other managed object in the `Finalize` method of your class. In a finalizer, only release unmanaged resources that your class owns directly. If your class does not own any unmanaged resources, do not include a `Finalize` method in your class definition. For more information, see Garbage Collection.

For more info about the events associated with opening and closing connections, see Audit Login Event Class and Audit Logout Event Class in the SQL Server documentation.

Removing Connections

The connection pooler removes a connection from the pool after it has been idle for approximately 4-8 minutes, or if the pooler detects that the connection with the server has been severed. Note that a severed connection can be detected only after attempting to communicate with the server. If a connection is found that is no longer connected to the server, it is marked as invalid. Invalid connections are removed from the connection pool only when they are closed or reclaimed.

If a connection exists to a server that has disappeared, this connection can be drawn from the pool even if the connection pooler has not detected the severed connection and marked it as invalid. This is the case because the overhead of checking that the connection is still valid would eliminate the benefits of having a pooler by causing another round trip to the server to occur. When this occurs, the first attempt to use the connection will detect that the connection has been severed, and an exception is thrown.

Clearing the Pool

ADO.NET 2.0 introduced two new methods to clear the pool: ClearAllPools and ClearPool. `ClearAllPools` clears the connection pools for a given provider, and `ClearPool` clears the connection pool that is associated with a specific connection. If there are connections being used at the time of the call, they are marked appropriately. When they are closed, they are discarded instead of being returned to the pool.

Transaction Support

Connections are drawn from the pool and assigned based on transaction context. Unless `Enlist=false` is specified in the connection string, the connection pool makes sure that the connection is enlisted in the Current context. When a connection is closed and returned to the pool with an enlisted `System.Transactions` transaction, it is set aside in such a way that the next request for that connection pool with the same `System.Transactions` transaction will return the same connection if it is available. If such a request is issued, and there are no pooled connections available, a connection is drawn from the non-transacted part of the pool and enlisted. If no connections are available in either area of the pool, a new connection is created and enlisted.

When a connection is closed, it is released back into the pool and into the appropriate subdivision based on its transaction context. Therefore, you can close the connection without generating an error, even though a distributed transaction is still pending. This allows you to commit or abort the distributed transaction later.

Controlling Connection Pooling with Connection String Keywords

The `ConnectionString` property of the SqlConnection object supports connection string key/value pairs that can be used to adjust the behavior of the connection pooling logic. For more information, see ConnectionString.

Pool Fragmentation

Pool fragmentation is a common problem in many Web applications where the application can create a large

number of pools that are not freed until the process exits. This leaves a large number of connections open and consuming memory, which results in poor performance.

Pool Fragmentation Due to Integrated Security

Connections are pooled according to the connection string plus the user identity. Therefore, if you use Basic authentication or Windows Authentication on the Web site and an integrated security login, you get one pool per user. Although this improves the performance of subsequent database requests for a single user, that user cannot take advantage of connections made by other users. It also results in at least one connection per user to the database server. This is a side effect of a particular Web application architecture that developers must weigh against security and auditing requirements.

Pool Fragmentation Due to Many Databases

Many Internet service providers host several Web sites on a single server. They may use a single database to confirm a Forms authentication login and then open a connection to a specific database for that user or group of users. The connection to the authentication database is pooled and used by everyone. However, there is a separate pool of connections to each database, which increase the number of connections to the server.

This is also a side-effect of the application design. There is a relatively simple way to avoid this side effect without compromising security when you connect to SQL Server. Instead of connecting to a separate database for each user or group, connect to the same database on the server and then execute the Transact-SQL USE statement to change to the desired database. The following code fragment demonstrates creating an initial connection to the `master` database and then switching to the desired database specified in the `databaseName` string variable.

```
' Assumes that command is a valid SqlCommand object and that
' connectionString connects to master.
    command.Text = "USE DatabaseName"
Using connection As New SqlConnection(connectionString)
    connection.Open()
    command.ExecuteNonQuery()
End Using
```

```
// Assumes that command is a SqlCommand object and that
// connectionString connects to master.
command.Text = "USE DatabaseName";
using (SqlConnection connection = new SqlConnection(
  connectionString))
  {
    connection.Open();
    command.ExecuteNonQuery();
  }
```

Application Roles and Connection Pooling

After a SQL Server application role has been activated by calling the `sp_setapprole` system stored procedure, the security context of that connection cannot be reset. However, if pooling is enabled, the connection is returned to the pool, and an error occurs when the pooled connection is reused. For more information, see the Knowledge Base article, "SQL application role errors with OLE DB resource pooling."

Application Role Alternatives

We recommend that you take advantage of security mechanisms that you can use instead of application roles. For more information, see Creating Application Roles in SQL Server.

See also

- Connection Pooling

- SQL Server and ADO.NET
- Performance Counters
- ADO.NET Overview

OLE DB, ODBC, and Oracle connection pooling

12/29/2019 • 5 minutes to read • Edit Online

Pooling connections can significantly enhance the performance and scalability of your application. This section discusses connection pooling for the .NET Framework data providers for OLE DB, ODBC, and Oracle.

OleDb

The .NET Framework Data Provider for OLE DB automatically pools connections using OLE DB session pooling. Connection string arguments can be used to enable or disable OLE DB services including pooling. For example, the following connection string disables OLE DB session pooling and automatic transaction enlistment.

```
Provider=SQLOLEDB;OLE DB Services=-4;Data Source=localhost;Integrated Security=SSPI;
```

We recommend that you always close or dispose of a connection when you are finished using it in order to return the connection to the pool. Connections that are not explicitly closed may not get returned to the pool. For example, a connection that has gone out of scope but that has not been explicitly closed will only be returned to the connection pool if the maximum pool size has been reached and the connection is still valid.

For more information about OLE DB session or resource pooling, as well as how to disable pooling by overriding OLE DB provider service defaults, see the OLE DB Programmer's Guide.

ODBC

Connection pooling for the .NET Framework Data Provider for ODBC is managed by the ODBC Driver Manager that is used for the connection, and is not affected by the .NET Framework Data Provider for ODBC.

To enable or disable connection pooling, open **ODBC Data Source Administrator** in the Administrative Tools folder of Control Panel. The **Connection Pooling** tab allows you to specify connection pooling parameters for each ODBC driver installed. Connection pooling changes for a specific ODBC driver affect all applications that use that ODBC driver.

OracleClient

The .NET Framework Data Provider for Oracle provides connection pooling automatically for your ADO.NET client application. You can also supply several connection string modifiers to control connection pooling behavior (see "Controlling Connection Pooling with Connection String Keywords," later in this topic).

Create and assign pools

When a connection is opened, a connection pool is created based on an exact matching algorithm that associates the pool with the connection string in the connection. Each connection pool is associated with a distinct connection string. When a new connection is opened, if the connection string is not an exact match to an existing pool, a new pool is created.

Once created, connection pools are not destroyed until the active process ends. Maintaining inactive or empty pools uses very few system resources.

Connection Addition

A connection pool is created for each unique connection string. When a pool is created, multiple connection objects are created and added to the pool so that the minimum pool size requirement is satisfied. Connections are

added to the pool as needed, up to the maximum pool size.

When an OracleConnection object is requested, it is obtained from the pool if a usable connection is available. To be usable, the connection must currently be unused, have a matching transaction context or not be associated with any transaction context, and have a valid link to the server.

If the maximum pool size has been reached and no usable connection is available, the request is queued. The connection pooler satisfies these requests by reallocating connections as they are released back into the pool. Connections are released back into the pool when they are closed or disposed.

Connection Removal

The connection pooler removes a connection from the pool after it has been idle for an extended period of time or if the pooler detects that the connection with the server has been severed. This can be detected only after attempting to communicate with the server. If a connection is found that is no longer connected to the server, it is marked as invalid. The connection pooler periodically scans connection pools looking for objects that have been released to the pool and are marked as invalid. These connections are then permanently removed.

If a connection exists to a server that has disappeared, this connection can be drawn from the pool if the connection pooler has not detected the severed connection and marked it as invalid. When this occurs, an exception is generated. However, you must still close the connection in order to release it back into the pool.

Do not call `Close` or `Dispose` on a `Connection`, a `DataReader`, or any other managed object in the `Finalize` method of your class. In a finalizer, only release unmanaged resources that your class owns directly. If your class does not own any unmanaged resources, do not include a `Finalize` method in your class definition. For more information, see Garbage Collection.

Transaction Support

Connections are drawn from the pool and assigned based on transaction context. The context of the requesting thread and the assigned connection must match. Therefore, each connection pool is subdivided into connections with no associated transaction context and into *N* subdivisions that each contain connections with a particular transaction context.

When a connection is closed, it is released back into the pool and into the appropriate subdivision based on its transaction context. Therefore, you can close the connection without generating an error, even though a distributed transaction is still pending. This allows you to commit or abort the distributed transaction at a later time.

Control Connection Pooling with Connection String Keywords

The ConnectionString property of the OracleConnection object supports connection string key/value pairs that can be used to adjust the behavior of the connection pooling logic.

The following table describes the ConnectionString values you can use to adjust connection pooling behavior.

NAME	DEFAULT	DESCRIPTION
`Connection Lifetime`	0	When a connection is returned to the pool, its creation time is compared with the current time, and the connection is destroyed if that time span (in seconds) exceeds the value specified by `Connection Lifetime`. This is useful in clustered configurations to force load balancing between a running server and a server just brought online. A value of zero (0) will cause pooled connections to have the maximum time-out.

NAME	DEFAULT	DESCRIPTION
`Enlist`	'true'	When `true`, the pooler automatically enlists the connection in the current transaction context of the creation thread if a transaction context exists.
`Max Pool Size`	100	The maximum number of connections allowed in the pool.
`Min Pool Size`	0	The minimum number of connections maintained in the pool.
`Pooling`	'true'	When `true`, the connection is drawn from the appropriate pool, or if necessary, created and added to the appropriate pool.

See also

- Connection Pooling
- Performance Counters
- ADO.NET Overview

Commands and Parameters

9/7/2019 • 2 minutes to read • Edit Online

After establishing a connection to a data source, you can execute commands and return results from the data source using a DbCommand object. You can create a command using one of the command constructors for the .NET Framework data provider you are working with. Constructors can take optional arguments, such as an SQL statement to execute at the data source, a DbConnection object, or a DbTransaction object. You can also configure those objects as properties of the command. You can also create a command for a particular connection using the CreateCommand method of a `DbConnection` object. The SQL statement being executed by the command can be configured using the CommandText property.

Each .NET Framework data provider included with the .NET Framework has a `Command` object. The .NET Framework Data Provider for OLE DB includes an OleDbCommand object, the .NET Framework Data Provider for SQL Server includes a SqlCommand object, the .NET Framework Data Provider for ODBC includes an OdbcCommand object, and the .NET Framework Data Provider for Oracle includes an OracleCommand object.

In This Section

Executing a Command
Describes the ADO.NET `Command` object and how to use it to execute queries and commands against a data source.

Configuring Parameters and Parameter Data Types
Describes working with `Command` parameters, including direction, data types, and parameter syntax.

Generating Commands with CommandBuilders
Describes how to use command builders to automatically generate INSERT, UPDATE, and DELETE commands for a `DataAdapter` that has a single-table SELECT command.

Obtaining a Single Value from a Database
Describes how to use the `ExecuteScalar` method of a `Command` object to return a single value from a database query.

Using Commands to Modify Data
Describes how to use a data provider to execute stored procedures or data definition language (DDL) statements.

See also

- DataAdapters and DataReaders
- DataSets, DataTables, and DataViews
- Connecting to a Data Source
- ADO.NET Overview

Executing a Command

9/7/2019 • 2 minutes to read • Edit Online

Each .NET Framework data provider included with the .NET Framework has its own command object that inherits from DbCommand. The .NET Framework Data Provider for OLE DB includes an OleDbCommand object, the .NET Framework Data Provider for SQL Server includes a SqlCommand object, the .NET Framework Data Provider for ODBC includes an OdbcCommand object, and the .NET Framework Data Provider for Oracle includes an OracleCommand object. Each of these objects exposes methods for executing commands based on the type of command and desired return value, as described in the following table.

COMMAND	RETURN VALUE
`ExecuteReader`	Returns a `DataReader` object.
`ExecuteScalar`	Returns a single scalar value.
`ExecuteNonQuery`	Executes a command that does not return any rows.
`ExecuteXMLReader`	Returns an XmlReader. Available for a `SqlCommand` object only.

Each strongly typed command object also supports a CommandType enumeration that specifies how a command string is interpreted, as described in the following table.

COMMANDTYPE	DESCRIPTION
`Text`	An SQL command defining the statements to be executed at the data source.
`StoredProcedure`	The name of the stored procedure. You can use the `Parameters` property of a command to access input and output parameters and return values, regardless of which `Execute` method is called. When using `ExecuteReader`, return values and output parameters will not be accessible until the `DataReader` is closed.
`TableDirect`	The name of a table.

Example

The following code example demonstrates how to create a SqlCommand object to execute a stored procedure by setting its properties. A SqlParameter object is used to specify the input parameter to the stored procedure. The command is executed using the ExecuteReader method, and the output from the SqlDataReader is displayed in the console window.

```
static void GetSalesByCategory(string connectionString,
    string categoryName)
{
    using (SqlConnection connection = new SqlConnection(connectionString))
    {
        // Create the command and set its properties.
        SqlCommand command = new SqlCommand();
        command.Connection = connection;
        command.CommandText = "SalesByCategory";
        command.CommandType = CommandType.StoredProcedure;

        // Add the input parameter and set its properties.
        SqlParameter parameter = new SqlParameter();
        parameter.ParameterName = "@CategoryName";
        parameter.SqlDbType = SqlDbType.NVarChar;
        parameter.Direction = ParameterDirection.Input;
        parameter.Value = categoryName;

        // Add the parameter to the Parameters collection.
        command.Parameters.Add(parameter);

        // Open the connection and execute the reader.
        connection.Open();
        using (SqlDataReader reader = command.ExecuteReader())
        {
            if (reader.HasRows)
            {
                while (reader.Read())
                {
                    Console.WriteLine("{0}: {1:C}", reader[0], reader[1]);
                }
            }
            else
            {
                Console.WriteLine("No rows found.");
            }
            reader.Close();
        }
    }
}
```

```
Shared Sub GetSalesByCategory(ByVal connectionString As String, _
    ByVal categoryName As String)

    Using connection As New SqlConnection(connectionString)

        ' Create the command and set its properties.
        Dim command As SqlCommand = New SqlCommand()
        command.Connection = connection
        command.CommandText = "SalesByCategory"
        command.CommandType = CommandType.StoredProcedure

        ' Add the input parameter and set its properties.
        Dim parameter As New SqlParameter()
        parameter.ParameterName = "@CategoryName"
        parameter.SqlDbType = SqlDbType.NVarChar
        parameter.Direction = ParameterDirection.Input
        parameter.Value = categoryName

        ' Add the parameter to the Parameters collection.
        command.Parameters.Add(parameter)

        ' Open the connection and execute the reader.
        connection.Open()
        Using reader As SqlDataReader = command.ExecuteReader()

            If reader.HasRows Then
                Do While reader.Read()
                    Console.WriteLine("{0}: {1:C}", _
                      reader(0), reader(1))
                Loop
            Else
                Console.WriteLine("No rows returned.")
            End If
        End Using
    End Using
End Sub
```

Troubleshooting Commands

The .NET Framework Data Provider for SQL Server adds performance counters to enable you to detect intermittent problems related to failed command executions. For more information see Performance Counters.

See also

- Commands and Parameters
- DataAdapters and DataReaders
- ADO.NET Overview

Configuring parameters and parameter data types

5/18/2019 • 9 minutes to read • Edit Online

Command objects use parameters to pass values to SQL statements or stored procedures, providing type checking and validation. Unlike command text, parameter input is treated as a literal value, not as executable code. This helps guard against "SQL injection" attacks, in which an attacker inserts a command that compromises security on the server into an SQL statement.

Parameterized commands can also improve query execution performance, because they help the database server accurately match the incoming command with a proper cached query plan. For more information, see Execution Plan Caching and Reuse and Parameters and Execution Plan Reuse. In addition to the security and performance benefits, parameterized commands provide a convenient method for organizing values passed to a data source.

A DbParameter object can be created by using its constructor, or by adding it to the DbParameterCollection by calling the `Add` method of the DbParameterCollection collection. The `Add` method will take as input either constructor arguments or an existing parameter object, depending on the data provider.

Supplying the ParameterDirection property

When adding parameters, you must supply a ParameterDirection property for parameters other than input parameters. The following table shows the `ParameterDirection` values that you can use with the ParameterDirection enumeration.

MEMBER NAME	DESCRIPTION
Input	The parameter is an input parameter. This is the default.
InputOutput	The parameter can perform both input and output.
Output	The parameter is an output parameter.
ReturnValue	The parameter represents a return value from an operation such as a stored procedure, built-in function, or user-defined function.

Working with parameter placeholders

The syntax for parameter placeholders depends on the data source. The .NET Framework data providers handle naming and specifying parameters and parameter placeholders differently. This syntax is customized to a specific data source, as described in the following table.

DATA PROVIDER	PARAMETER NAMING SYNTAX
System.Data.SqlClient	Uses named parameters in the format `@`*parametername*.
System.Data.OleDb	Uses positional parameter markers indicated by a question mark (`?`).
System.Data.Odbc	Uses positional parameter markers indicated by a question mark (`?`).

DATA PROVIDER	PARAMETER NAMING SYNTAX
System.Data.OracleClient	Uses named parameters in the format `:` *parmname* (or *parmname*).

Specifying parameter data types

The data type of a parameter is specific to the .NET Framework data provider. Specifying the type converts the value of the `Parameter` to the .NET Framework data provider type before passing the value to the data source. You may also specify the type of a `Parameter` in a generic manner by setting the `DbType` property of the `Parameter` object to a particular DbType.

The .NET Framework data provider type of a `Parameter` object is inferred from the .NET Framework type of the `Value` of the `Parameter` object, or from the `DbType` of the `Parameter` object. The following table shows the inferred `Parameter` type based on the object passed as the `Parameter` value or the specified `DbType`.

.NET FRAMEWORK TYPE	DBTYPE	SQLDBTYPE	OLEDBTYPE	ODBCTYPE	ORACLETYPE
Boolean	Boolean	Bit	Boolean	Bit	Byte
Byte	Byte	TinyInt	UnsignedTinyInt	TinyInt	Byte
byte[]	Binary	VarBinary. This implicit conversion will fail if the byte array is larger than the maximum size of a VarBinary, which is 8000 bytes.For byte arrays larger than 8000 bytes, explicitly set the SqlDbType.	VarBinary	Binary	Raw
Char		Inferring a SqlDbType from char is not supported.	Char	Char	Byte
DateTime	DateTime	DateTime	DBTimeStamp	DateTime	DateTime
DateTimeOffset	DateTimeOffset	DateTimeOffset in SQL Server 2008. Inferring a SqlDbType from DateTimeOffset is not supported in versions of SQL Server earlier than SQL Server 2008.			DateTime

.NET FRAMEWORK TYPE	DBTYPE	SQLDBTYPE	OLEDBTYPE	ODBCTYPE	ORACLETYPE
Decimal	Decimal	Decimal	Decimal	Numeric	Number
Double	Double	Float	Double	Double	Double
Single	Single	Real	Single	Real	Float
Guid	Guid	UniqueIdentifier	Guid	UniqueIdentifier	Raw
Int16	Int16	SmallInt	SmallInt	SmallInt	Int16
Int32	Int32	Int	Int	Int	Int32
Int64	Int64	BigInt	BigInt	BigInt	Number
Object	Object	Variant	Variant	Inferring an OdbcType from Object is not supported.	Blob
String	String	NVarChar. This implicit conversion will fail if the string is larger than the maximum size of an NVarChar, which is 4000 characters. For strings larger than 4000 characters, explicitly set the SqlDbType.	VarWChar	NVarChar	NVarChar
TimeSpan	Time	Time in SQL Server 2008. Inferring a SqlDbType from TimeSpan is not supported in versions of SQL Server earlier than SQL Server 2008.	DBTime	Time	DateTime
UInt16	UInt16	Inferring a SqlDbType from UInt16 is not supported.	UnsignedSmallInt	Int	UInt16
UInt32	UInt32	Inferring a SqlDbType from UInt32 is not supported.	UnsignedInt	BigInt	UInt32

.NET FRAMEWORK TYPE	DBTYPE	SQLDBTYPE	OLEDBTYPE	ODBCTYPE	ORACLETYPE
UInt64	UInt64	Inferring a SqlDbType from UInt64 is not supported.	UnsignedBigInt	Numeric	Number
	AnsiString	VarChar	VarChar	VarChar	VarChar
	AnsiStringFixedLength	Char	Char	Char	Char
	Currency	Money	Currency	Inferring an `OdbcType` from `Currency` is not supported.	Number
	Date	Date in SQL Server 2008. Inferring a SqlDbType from Date is not supported in versions of SQL Server earlier than SQL Server 2008.	DBDate	Date	DateTime
	SByte	Inferring a SqlDbType from SByte is not supported.	TinyInt	Inferring an `OdbcType` from SByte is not supported.	SByte
	StringFixedLength	NChar	WChar	NChar	NChar
	Time	Time in SQL Server 2008. Inferring a SqlDbType from Time is not supported in versions of SQL Server earlier than SQL Server 2008.	DBTime	Time	DateTime
	VarNumeric	Inferring a SqlDbType from VarNumeric is not supported.	VarNumeric	Inferring an `OdbcType` from VarNumeric is not supported.	Number

.NET FRAMEWORK TYPE	DBTYPE	SQLDBTYPE	OLEDBTYPE	ODBCTYPE	ORACLETYPE
user-defined type (an object with SqlUserDefinedAggregateAttribute	Object or String, depending the provider (SqlClient always returns an Object, Odbc always returns a String, and the OleDb managed data provider can see either	SqlDbType.Udt if SqlUserDefinedTypeAttribute is present, otherwise Variant	OleDbType.VarWChar (if value is null) otherwise OleDbType.Variant.	OdbcType.NVarChar	not supported

> **NOTE**
>
> Conversions from decimal to other types are narrowing conversions that round the decimal value to the nearest integer value toward zero. If the result of the conversion is not representable in the destination type, an OverflowException is thrown.

> **NOTE**
>
> When you send a null parameter value to the server, you must specify DBNull, not `null` (`Nothing` in Visual Basic). The null value in the system is an empty object that has no value. DBNull is used to represent null values. For more information about database nulls, see Handling Null Values.

Deriving parameter information

Parameters can also be derived from a stored procedure using the `DbCommandBuilder` class. Both the `SqlCommandBuilder` and `OleDbCommandBuilder` classes provide a static method, `DeriveParameters`, which automatically populates the parameters collection of a command object that uses parameter information from a stored procedure. Note that `DeriveParameters` overwrites any existing parameter information for the command.

> **NOTE**
>
> Deriving parameter information incurs a performance penalty because it requires an additional round trip to the data source to retrieve the information. If parameter information is known at design time, you can improve the performance of your application by setting the parameters explicitly.

For more information, see Generating Commands with CommandBuilders.

Using parameters with a SqlCommand and a stored procedure

Stored procedures offer many advantages in data-driven applications. By using stored procedures, database operations can be encapsulated in a single command, optimized for best performance, and enhanced with additional security. Although a stored procedure can be called by passing the stored procedure name followed by parameter arguments as an SQL statement, by using the Parameters collection of the ADO.NET DbCommand object enables you to more explicitly define stored procedure parameters, and to access output parameters and return values.

NOTE

Parameterized statements are executed on the server by using `sp_executesql,` which allows for query plan reuse. Local cursors or variables in the `sp_executesql` batch are not visible to the batch that calls `sp_executesql`. Changes in database context last only to the end of the `sp_executesql` statement. For more information, see sp_executesql (Transact-SQL).

When using parameters with a SqlCommand to execute a SQL Server stored procedure, the names of the parameters added to the Parameters collection must match the names of the parameter markers in the stored procedure. The .NET Framework Data Provider for SQL Server does not support the question mark (?) placeholder for passing parameters to an SQL statement or a stored procedure. It treats parameters in the stored procedure as named parameters and searches for matching parameter markers. For example, the `CustOrderHist` stored procedure is defined by using a parameter named `@CustomerID`. When your code executes the stored procedure, it must also use a parameter named `@CustomerID`.

```
CREATE PROCEDURE dbo.CustOrderHist @CustomerID varchar(5)
```

Example

This example demonstrates how to call a SQL Server stored procedure in the `Northwind` sample database. The name of the stored procedure is `dbo.SalesByCategory` and it has an input parameter named `@CategoryName` with a data type of `nvarchar(15)`. The code creates a new SqlConnection inside a using block so that the connection is disposed when the procedure ends. The SqlCommand and SqlParameter objects are created, and their properties set. A SqlDataReader executes the `SqlCommand` and returns the result set from the stored procedure, displaying the output in the console window.

NOTE

Instead of creating `SqlCommand` and `SqlParameter` objects and then setting properties in separate statements, you can instead elect to use one of the overloaded constructors to set multiple properties in a single statement.

```
static void GetSalesByCategory(string connectionString,
    string categoryName)
{
    using (SqlConnection connection = new SqlConnection(connectionString))
    {
        // Create the command and set its properties.
        SqlCommand command = new SqlCommand();
        command.Connection = connection;
        command.CommandText = "SalesByCategory";
        command.CommandType = CommandType.StoredProcedure;

        // Add the input parameter and set its properties.
        SqlParameter parameter = new SqlParameter();
        parameter.ParameterName = "@CategoryName";
        parameter.SqlDbType = SqlDbType.NVarChar;
        parameter.Direction = ParameterDirection.Input;
        parameter.Value = categoryName;

        // Add the parameter to the Parameters collection.
        command.Parameters.Add(parameter);

        // Open the connection and execute the reader.
        connection.Open();
        using (SqlDataReader reader = command.ExecuteReader())
        {
            if (reader.HasRows)
            {
                while (reader.Read())
                {
                    Console.WriteLine("{0}: {1:C}", reader[0], reader[1]);
                }
            }
            else
            {
                Console.WriteLine("No rows found.");
            }
            reader.Close();
        }
    }
}
```

```
Shared Sub GetSalesByCategory(ByVal connectionString As String, _
    ByVal categoryName As String)

    Using connection As New SqlConnection(connectionString)

        ' Create the command and set its properties.
        Dim command As SqlCommand = New SqlCommand()
        command.Connection = connection
        command.CommandText = "SalesByCategory"
        command.CommandType = CommandType.StoredProcedure

        ' Add the input parameter and set its properties.
        Dim parameter As New SqlParameter()
        parameter.ParameterName = "@CategoryName"
        parameter.SqlDbType = SqlDbType.NVarChar
        parameter.Direction = ParameterDirection.Input
        parameter.Value = categoryName

        ' Add the parameter to the Parameters collection.
        command.Parameters.Add(parameter)

        ' Open the connection and execute the reader.
        connection.Open()
        Using reader As SqlDataReader = command.ExecuteReader()

            If reader.HasRows Then
                Do While reader.Read()
                    Console.WriteLine("{0}: {1:C}", _
                      reader(0), reader(1))
                Loop
            Else
                Console.WriteLine("No rows returned.")
            End If
        End Using
    End Using
End Sub
```

Using parameters with an OleDbCommand or OdbcCommand

When using parameters with an OleDbCommand or OdbcCommand, the order of the parameters added to the `Parameters` collection must match the order of the parameters defined in your stored procedure. The .NET Framework Data Provider for OLE DB and .NET Framework Data Provider for ODBC treat parameters in a stored procedure as placeholders and apply parameter values in order. In addition, return value parameters must be the first parameters added to the `Parameters` collection.

The .NET Framework Data Provider for OLE DB and .NET Framework Data Provider for ODBC do not support named parameters for passing parameters to an SQL statement or a stored procedure. In this case, you must use the question mark (?) placeholder, as in the following example.

```
SELECT * FROM Customers WHERE CustomerID = ?
```

As a result, the order in which `Parameter` objects are added to the `Parameters` collection must directly correspond to the position of the ? placeholder for the parameter.

OleDb Example

```
Dim command As OleDbCommand = New OleDbCommand( _
  "SampleProc", connection)
command.CommandType = CommandType.StoredProcedure

Dim parameter As OleDbParameter = command.Parameters.Add( _
  "RETURN_VALUE", OleDbType.Integer)
parameter.Direction = ParameterDirection.ReturnValue

parameter = command.Parameters.Add( _
  "@InputParm", OleDbType.VarChar, 12)
parameter.Value = "Sample Value"

parameter = command.Parameters.Add( _
  "@OutputParm", OleDbType.VarChar, 28)
parameter.Direction = ParameterDirection.Output
```

```
OleDbCommand command = new OleDbCommand("SampleProc", connection);
command.CommandType = CommandType.StoredProcedure;

OleDbParameter parameter = command.Parameters.Add(
  "RETURN_VALUE", OleDbType.Integer);
parameter.Direction = ParameterDirection.ReturnValue;

parameter = command.Parameters.Add(
  "@InputParm", OleDbType.VarChar, 12);
parameter.Value = "Sample Value";

parameter = command.Parameters.Add(
  "@OutputParm", OleDbType.VarChar, 28);
parameter.Direction = ParameterDirection.Output;
```

Odbc Example

```
Dim command As OdbcCommand = New OdbcCommand( _
  "{ ? = CALL SampleProc(?, ?) }", connection)
command.CommandType = CommandType.StoredProcedure

Dim parameter As OdbcParameter = command.Parameters.Add("RETURN_VALUE", OdbcType.Int)
parameter.Direction = ParameterDirection.ReturnValue

parameter = command.Parameters.Add( _
  "@InputParm", OdbcType.VarChar, 12)
parameter.Value = "Sample Value"

parameter = command.Parameters.Add( _
  "@OutputParm", OdbcType.VarChar, 28)
parameter.Direction = ParameterDirection.Output
```

```
OdbcCommand command = new OdbcCommand( _
  "{ ? = CALL SampleProc(?, ?) }", connection);
command.CommandType = CommandType.StoredProcedure;

OdbcParameter parameter = command.Parameters.Add( _
  "RETURN_VALUE", OdbcType.Int);
parameter.Direction = ParameterDirection.ReturnValue;

parameter = command.Parameters.Add( _
  "@InputParm", OdbcType.VarChar, 12);
parameter.Value = "Sample Value";

parameter = command.Parameters.Add( _
  "@OutputParm", OdbcType.VarChar, 28);
parameter.Direction = ParameterDirection.Output;
```

See also

- Commands and Parameters
- DataAdapter Parameters
- Data Type Mappings in ADO.NET
- ADO.NET Overview

Generating Commands with CommandBuilders

3/12/2020 • 6 minutes to read • Edit Online

When the `SelectCommand` property is dynamically specified at run time, such as through a query tool that takes a textual command from the user, you may not be able to specify the appropriate `InsertCommand`, `UpdateCommand`, or `DeleteCommand` at design time. If your DataTable maps to or is generated from a single database table, you can take advantage of the DbCommandBuilder object to automatically generate the `DeleteCommand`, `InsertCommand`, and `UpdateCommand` of the DbDataAdapter.

As a minimum requirement, you must set the `SelectCommand` property in order for automatic command generation to work. The table schema retrieved by the `SelectCommand` property determines the syntax of the automatically generated INSERT, UPDATE, and DELETE statements.

The DbCommandBuilder must execute the `SelectCommand` in order to return the metadata necessary to construct the INSERT, UPDATE, and DELETE SQL commands. As a result, an extra trip to the data source is necessary, and this can hinder performance. To achieve optimal performance, specify your commands explicitly rather than using the DbCommandBuilder.

The `SelectCommand` must also return at least one primary key or unique column. If none are present, an `InvalidOperation` exception is generated, and the commands are not generated.

When associated with a `DataAdapter`, the DbCommandBuilder automatically generates the `InsertCommand`, `UpdateCommand`, and `DeleteCommand` properties of the `DataAdapter` if they are null references. If a `Command` already exists for a property, the existing `Command` is used.

Database views that are created by joining two or more tables together are not considered a single database table. In this instance you cannot use the DbCommandBuilder to automatically generate commands; you must specify your commands explicitly. For information about explicitly setting commands to resolve updates to a `DataSet` back to the data source, see Updating Data Sources with DataAdapters.

You might want to map output parameters back to the updated row of a `DataSet`. One common task would be retrieving the value of an automatically generated identity field or time stamp from the data source. The DbCommandBuilder will not map output parameters to columns in an updated row by default. In this instance you must specify your command explicitly. For an example of mapping an automatically generated identity field back to a column of an inserted row, see Retrieving Identity or Autonumber Values.

Rules for Automatically Generated Commands

The following table shows the rules for how automatically generated commands are generated.

COMMAND	RULE
`InsertCommand`	Inserts a row at the data source for all rows in the table with a RowState of Added. Inserts values for all columns that are updateable (but not columns such as identities, expressions, or timestamps).

COMMAND	RULE
`UpdateCommand`	Updates rows at the data source for all rows in the table with a `RowState` of Modified. Updates the values of all columns except for columns that are not updateable, such as identities or expressions. Updates all rows where the column values at the data source match the primary key column values of the row, and where the remaining columns at the data source match the original values of the row. For more information, see "Optimistic Concurrency Model for Updates and Deletes," later in this topic.
`DeleteCommand`	Deletes rows at the data source for all rows in the table with a `RowState` of Deleted. Deletes all rows where the column values match the primary key column values of the row, and where the remaining columns at the data source match the original values of the row. For more information, see "Optimistic Concurrency Model for Updates and Deletes," later in this topic.

Optimistic Concurrency Model for Updates and Deletes

The logic for generating commands automatically for UPDATE and DELETE statements is based on *optimistic concurrency*--that is, records are not locked for editing and can be modified by other users or processes at any time. Because a record could have been modified after it was returned from the SELECT statement, but before the UPDATE or DELETE statement is issued, the automatically generated UPDATE or DELETE statement contains a WHERE clause, specifying that a row is only updated if it contains all original values and has not been deleted from the data source. This is done to avoid overwriting new data. Where an automatically generated update attempts to update a row that has been deleted or that does not contain the original values found in the DataSet, the command does not affect any records, and a DBConcurrencyException is thrown.

If you want the UPDATE or DELETE to complete regardless of original values, you must explicitly set the `UpdateCommand` for the `DataAdapter` and not rely on automatic command generation.

Limitations of Automatic Command Generation Logic

The following limitations apply to automatic command generation.

Unrelated Tables Only

The automatic command generation logic generates INSERT, UPDATE, or DELETE statements for stand-alone tables without taking into account any relationships to other tables at the data source. As a result, you may encounter a failure when calling `Update` to submit changes for a column that participates in a foreign key constraint in the database. To avoid this exception, do not use the DbCommandBuilder for updating columns involved in a foreign key constraint; instead, explicitly specify the statements used to perform the operation.

Table and Column Names

Automatic command generation logic may fail if column names or table names contain any special characters, such as spaces, periods, quotation marks, or other nonalphanumeric characters, even if delimited by brackets. Depending on the provider, setting the QuotePrefix and QuoteSuffix parameters may allow the generation logic to process spaces, but it cannot escape special characters. Fully qualified table names in the form of *catalog.schema.table* are supported.

Using the CommandBuilder to Automatically Generate an SQL Statement

To automatically generate SQL statements for a `DataAdapter`, first set the `SelectCommand` property of the `DataAdapter`, then create a `CommandBuilder` object, and specify as an argument the `DataAdapter` for which the `CommandBuilder` will automatically generate SQL statements.

```
' Assumes that connection is a valid SqlConnection object
' inside of a Using block.
Dim adapter As SqlDataAdapter = New SqlDataAdapter( _
  "SELECT * FROM dbo.Customers", connection)
Dim builder As SqlCommandBuilder = New SqlCommandBuilder(adapter)
builder.QuotePrefix = "["
builder.QuoteSuffix = "]"
```

```
// Assumes that connection is a valid SqlConnection object
// inside of a using block.
SqlDataAdapter adapter = new SqlDataAdapter(
  "SELECT * FROM dbo.Customers", connection);
SqlCommandBuilder builder = new SqlCommandBuilder(adapter);
builder.QuotePrefix = "[";
builder.QuoteSuffix = "]";
```

Modifying the SelectCommand

If you modify the `CommandText` of the `SelectCommand` after the INSERT, UPDATE, or DELETE commands have been automatically generated, an exception may occur. If the modified `SelectCommand.CommandText` contains schema information that is inconsistent with the `SelectCommand.CommandText` used when the insert, update, or delete commands were automatically generated, future calls to the `DataAdapter.Update` method may attempt to access columns that no longer exist in the current table referenced by the `SelectCommand`, and an exception will be thrown.

You can refresh the schema information used by the `CommandBuilder` to automatically generate commands by calling the `RefreshSchema` method of the `CommandBuilder`.

If you want to know what command was automatically generated, you can obtain a reference to the automatically generated commands by using the `GetInsertCommand`, `GetUpdateCommand`, and `GetDeleteCommand` methods of the `CommandBuilder` object and checking the `CommandText` property of the associated command.

The following code example writes to the console the update command that was automatically generated.

```
Console.WriteLine(builder.GetUpdateCommand().CommandText)
```

```
Console.WriteLine(builder.GetUpdateCommand().CommandText);
```

The following example recreates the `Customers` table in the `custDS` dataset. The **RefreshSchema** method is called to refresh the automatically generated commands with this new column information.

```
' Assumes an open SqlConnection and SqlDataAdapter inside of a Using block.
adapter.SelectCommand.CommandText = _
  "SELECT CustomerID, ContactName FROM dbo.Customers"
builder.RefreshSchema()

custDS.Tables.Remove(custDS.Tables("Customers"))
adapter.Fill(custDS, "Customers")
```

```
// Assumes an open SqlConnection and SqlDataAdapter inside of a using block.
adapter.SelectCommand.CommandText =
  "SELECT CustomerID, ContactName FROM dbo.Customers";
builder.RefreshSchema();

custDS.Tables.Remove(custDS.Tables["Customers"]);
adapter.Fill(custDS, "Customers");
```

See also

- Commands and Parameters
- Executing a Command
- DbConnection, DbCommand and DbException
- ADO.NET Overview

Obtaining a Single Value from a Database

9/7/2019 • 2 minutes to read • Edit Online

You may need to return database information that is simply a single value rather than in the form of a table or data stream. For example, you may want to return the result of an aggregate function such as COUNT(*), SUM(Price), or AVG(Quantity). The **Command** object provides the capability to return single values using the **ExecuteScalar** method. The **ExecuteScalar** method returns, as a scalar value, the value of the first column of the first row of the result set.

The following code example inserts a new value in the database using a SqlCommand. The ExecuteScalar method is used to return the identity column value for the inserted record.

```
static public int AddProductCategory(string newName, string connString)
{
    Int32 newProdID = 0;
    string sql =
        "INSERT INTO Production.ProductCategory (Name) VALUES (@Name); "
        + "SELECT CAST(scope_identity() AS int)";
    using (SqlConnection conn = new SqlConnection(connString))
    {
        SqlCommand cmd = new SqlCommand(sql, conn);
        cmd.Parameters.Add("@Name", SqlDbType.VarChar);
        cmd.Parameters["@name"].Value = newName;
        try
        {
            conn.Open();
            newProdID = (Int32)cmd.ExecuteScalar();
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
        }
    }
    return (int)newProdID;
}
```

```
Public Function AddProductCategory( _
  ByVal newName As String, ByVal connString As String) As Integer
    Dim newProdID As Int32 = 0
    Dim sql As String = _
     "INSERT INTO Production.ProductCategory (Name) VALUES (@Name); " _
       & "SELECT CAST(scope_identity() AS int);"

    Using conn As New SqlConnection(connString)
        Dim cmd As New SqlCommand(sql, conn)
        cmd.Parameters.Add("@Name", SqlDbType.VarChar)
        cmd.Parameters("@Name").Value = newName
        Try
            conn.Open()
            newProdID = Convert.ToInt32(cmd.ExecuteScalar())
        Catch ex As Exception
            Console.WriteLine(ex.Message)
        End Try
    End Using

    Return newProdID
End Function
```

See also

- Commands and Parameters
- Executing a Command
- DbConnection, DbCommand and DbException
- ADO.NET Overview

Using Commands to Modify Data

9/7/2019 • 2 minutes to read • Edit Online

Using a .NET Framework data provider, you can execute stored procedures or data definition language statements (for example, CREATE TABLE and ALTER COLUMN) to perform schema manipulation on a database or catalog. These commands do not return rows as a query would, so the **Command** object provides an **ExecuteNonQuery** to process them.

In addition to using **ExecuteNonQuery** to modify schema, you can also use this method to process SQL statements that modify data but that do not return rows, such as INSERT, UPDATE, and DELETE.

Although rows are not returned by the **ExecuteNonQuery** method, input and output parameters and return values can be passed and returned via the **Parameters** collection of the **Command** object.

In This Section

Updating Data in a Data Source
Describes how to execute commands or stored procedures that modify data in a database.

Performing Catalog Operations
Describes how to execute commands that modify database schema.

See also

- Retrieving and Modifying Data in ADO.NET
- Commands and Parameters
- ADO.NET Overview

Updating Data in a Data Source

3/12/2020 • 2 minutes to read • Edit Online

SQL statements that modify data (such as INSERT, UPDATE, or DELETE) do not return rows. Similarly, many stored procedures perform an action but do not return rows. To execute commands that do not return rows, create a **Command** object with the appropriate SQL command and a **Connection**, including any required **Parameters**. Execute the command with the **ExecuteNonQuery** method of the **Command** object.

The **ExecuteNonQuery** method returns an integer that represents the number of rows affected by the statement or stored procedure that was executed. If multiple statements are executed, the value returned is the sum of the records affected by all of the statements executed.

Example

The following code example executes an INSERT statement to insert a record into a database using **ExecuteNonQuery**.

```
' Assumes connection is a valid SqlConnection.
connection.Open()

Dim queryString As String = "INSERT INTO Customers " & _
  "(CustomerID, CompanyName) Values('NWIND', 'Northwind Traders')"

Dim command As SqlCommand = New SqlCommand(queryString, connection)
Dim recordsAffected As Int32 = command.ExecuteNonQuery()
```

```
// Assumes connection is a valid SqlConnection.
connection.Open();

string queryString = "INSERT INTO Customers " +
  "(CustomerID, CompanyName) Values('NWIND', 'Northwind Traders')";

SqlCommand command = new SqlCommand(queryString, connection);
Int32 recordsAffected = command.ExecuteNonQuery();
```

The following code example executes the stored procedure created by the sample code in Performing Catalog Operations. No rows are returned by the stored procedure, so the **ExecuteNonQuery** method is used, but the stored procedure does receive an input parameter and returns an output parameter and a return value.

For the OleDbCommand object, the **ReturnValue** parameter must be added to the **Parameters** collection first.

```
' Assumes connection is a valid SqlConnection.
Dim command As SqlCommand = _
   New SqlCommand("InsertCategory" , connection)
command.CommandType = CommandType.StoredProcedure

Dim parameter As SqlParameter = _
 command.Parameters.Add("@RowCount", SqlDbType.Int)
parameter.Direction = ParameterDirection.ReturnValue

parameter = command.Parameters.Add( _
  "@CategoryName", SqlDbType.NChar, 15)

parameter = command.Parameters.Add("@Identity", SqlDbType.Int)
parameter.Direction = ParameterDirection.Output

command.Parameters("@CategoryName").Value = "New Category"
command.ExecuteNonQuery()

Dim categoryID As Int32 = CInt(command.Parameters("@Identity").Value)
Dim rowCount As Int32 = CInt(command.Parameters("@RowCount").Value)
```

```
// Assumes connection is a valid SqlConnection.
SqlCommand command = new SqlCommand("InsertCategory" , connection);
command.CommandType = CommandType.StoredProcedure;

SqlParameter parameter = command.Parameters.Add(
  "@RowCount", SqlDbType.Int);
parameter.Direction = ParameterDirection.ReturnValue;

parameter = command.Parameters.Add(
  "@CategoryName", SqlDbType.NChar, 15);

parameter = command.Parameters.Add("@Identity", SqlDbType.Int);
parameter.Direction = ParameterDirection.Output;

command.Parameters["@CategoryName"].Value = "New Category";
command.ExecuteNonQuery();

Int32 categoryID = (Int32) command.Parameters["@Identity"].Value;
Int32 rowCount = (Int32) command.Parameters["@RowCount"].Value;
```

See also

- Using Commands to Modify Data
- Updating Data Sources with DataAdapters
- Commands and Parameters
- ADO.NET Overview

Performing Catalog Operations

3/12/2020 • 2 minutes to read • Edit Online

To execute a command to modify a database or catalog, such as the CREATE TABLE or CREATE PROCEDURE statement, create a **Command** object using the appropriate SQL statements and a **Connection** object. Execute the command with the **ExecuteNonQuery** method of the **Command** object.

The following code example creates a stored procedure in a Microsoft SQL Server database.

```
' Assumes connection is a valid SqlConnection.
Dim queryString As String = "CREATE PROCEDURE InsertCategory " & _
    "@CategoryName nchar(15), " & _
    "@Identity int OUT " & _
    "AS " & _
    "INSERT INTO Categories (CategoryName) VALUES(@CategoryName) " & _
    "SET @Identity = @@Identity " & _
    "RETURN @@ROWCOUNT"

Dim command As SqlCommand = New SqlCommand(queryString, connection)
command.ExecuteNonQuery()
```

```
// Assumes connection is a valid SqlConnection.
string queryString = "CREATE PROCEDURE InsertCategory  " +
    "@CategoryName nchar(15), " +
    "@Identity int OUT " +
    "AS " +
    "INSERT INTO Categories (CategoryName) VALUES(@CategoryName) " +
    "SET @Identity = @@Identity " +
    "RETURN @@ROWCOUNT";

SqlCommand command = new SqlCommand(queryString, connection);
command.ExecuteNonQuery();
```

See also

- Using Commands to Modify Data
- Commands and Parameters
- ADO.NET Overview

DataAdapters and DataReaders

9/7/2019 • 2 minutes to read • Edit Online

You can use the ADO.NET **DataReader** to retrieve a read-only, forward-only stream of data from a database. Results are returned as the query executes, and are stored in the network buffer on the client until you request them using the **Read** method of the **DataReader**. Using the **DataReader** can increase application performance both by retrieving data as soon as it is available, and (by default) storing only one row at a time in memory, reducing system overhead.

A DataAdapter is used to retrieve data from a data source and populate tables within a DataSet. The `DataAdapter` also resolves changes made to the `DataSet` back to the data source. The `DataAdapter` uses the `Connection` object of the .NET Framework data provider to connect to a data source, and it uses `Command` objects to retrieve data from and resolve changes to the data source.

Each .NET Framework data provider included with the .NET Framework has a DbDataReader and a DbDataAdapter object: the .NET Framework Data Provider for OLE DB includes an OleDbDataReader and an OleDbDataAdapter object, the .NET Framework Data Provider for SQL Server includes a SqlDataReader and a SqlDataAdapter object, the .NET Framework Data Provider for ODBC includes an OdbcDataReader and an OdbcDataAdapter object, and the .NET Framework Data Provider for Oracle includes an OracleDataReader and an OracleDataAdapter object.

In This Section

Retrieving Data Using a DataReader
Describes the ADO.NET **DataReader** object and how to use it to return a stream of results from a data source.

Populating a DataSet from a DataAdapter
Describes how to fill a `DataSet` with tables, columns, and rows by using a `DataAdapter`.

DataAdapter Parameters
Describes how to use parameters with the command properties of a `DataAdapter` including how to map the contents of a column in a `DataSet` to a command parameter.

Adding Existing Constraints to a DataSet
Describes how to add existing constraints to a `DataSet`.

DataAdapter DataTable and DataColumn Mappings
Describes how to set up `DataTableMappings` and `ColumnMappings` for a `DataAdapter`.

Paging Through a Query Result
Provides an example of viewing the results of a query as pages of data.

Updating Data Sources with DataAdapters
Describes how to use a `DataAdapter` to resolve changes in a `DataSet` back to the database.

Handling DataAdapter Events
Describes `DataAdapter` events and how to use them.

Performing Batch Operations Using DataAdapters
Describes enhancing application performance by reducing the number of round trips to SQL Server when applying updates from the `DataSet`.

See also

- Connecting to a Data Source
- Commands and Parameters
- Transactions and Concurrency
- DataSets, DataTables, and DataViews
- ADO.NET Overview

Retrieve data using a DataReader

3/12/2020 • 8 minutes to read • Edit Online

To retrieve data using a **DataReader**, create an instance of the **Command** object, and then create a **DataReader** by calling **Command.ExecuteReader** to retrieve rows from a data source. The **DataReader** provides an unbuffered stream of data that allows procedural logic to efficiently process results from a data source sequentially. The **DataReader** is a good choice when you're retrieving large amounts of data because the data is not cached in memory.

The following example illustrates using a **DataReader**, where `reader` represents a valid DataReader and `command` represents a valid Command object.

```
reader = command.ExecuteReader();
```

```
reader = command.ExecuteReader()
```

Use the **DataReader.Read** method to obtain a row from the query results. You can access each column of the returned row by passing the name or ordinal number of the column to the **DataReader**. However, for best performance, the **DataReader** provides a series of methods that allow you to access column values in their native data types (**GetDateTime**, **GetDouble**, **GetGuid**, **GetInt32**, and so on). For a list of typed accessor methods for data provider-specific **DataReaders**, see OleDbDataReader and SqlDataReader. Using the typed accessor methods when you know the underlying data type reduces the amount of type conversion required when retrieving the column value.

The following example iterates through a **DataReader** object and returns two columns from each row.

```
static void HasRows(SqlConnection connection)
{
    using (connection)
    {
        SqlCommand command = new SqlCommand(
          "SELECT CategoryID, CategoryName FROM Categories;",
          connection);
        connection.Open();

        SqlDataReader reader = command.ExecuteReader();

        if (reader.HasRows)
        {
            while (reader.Read())
            {
                Console.WriteLine("{0}\t{1}", reader.GetInt32(0),
                    reader.GetString(1));
            }
        }
        else
        {
            Console.WriteLine("No rows found.");
        }
        reader.Close();
    }
}
```

```
Private Sub HasRows(ByVal connection As SqlConnection)
    Using connection
        Dim command As SqlCommand = New SqlCommand( _
          "SELECT CategoryID, CategoryName FROM Categories;", _
          connection)
        connection.Open()

        Dim reader As SqlDataReader = command.ExecuteReader()

        If reader.HasRows Then
            Do While reader.Read()
                Console.WriteLine(reader.GetInt32(0) _
                  & vbTab & reader.GetString(1))
            Loop
        Else
            Console.WriteLine("No rows found.")
        End If

        reader.Close()
    End Using
End Sub
```

Closing the DataReader

Always call the **Close** method when you have finished using the **DataReader** object.

If your **Command** contains output parameters or return values, those values are not available until the **DataReader** is closed.

While a **DataReader** is open, the **Connection** is in use exclusively by that **DataReader**. You cannot execute any commands for the **Connection**, including creating another **DataReader**, until the original **DataReader** is closed.

> **NOTE**
>
> Do not call **Close** or **Dispose** on a **Connection**, a **DataReader**, or any other managed object in the **Finalize** method of your class. In a finalizer, only release unmanaged resources that your class owns directly. If your class does not own any unmanaged resources, do not include a **Finalize** method in your class definition. For more information, see Garbage Collection.

Retrieving multiple result sets using NextResult

If the **DataReader** returns multiple result sets, call the **NextResult** method to iterate through the result sets sequentially. The following example shows the SqlDataReader processing the results of two SELECT statements using the ExecuteReader method.

```
static void RetrieveMultipleResults(SqlConnection connection)
{
    using (connection)
    {
        SqlCommand command = new SqlCommand(
          "SELECT CategoryID, CategoryName FROM dbo.Categories;" +
          "SELECT EmployeeID, LastName FROM dbo.Employees",
          connection);
        connection.Open();

        SqlDataReader reader = command.ExecuteReader();

        while (reader.HasRows)
        {
            Console.WriteLine("\t{0}\t{1}", reader.GetName(0),
                reader.GetName(1));

            while (reader.Read())
            {
                Console.WriteLine("\t{0}\t{1}", reader.GetInt32(0),
                    reader.GetString(1));
            }
            reader.NextResult();
        }
    }
}
```

```
Private Sub RetrieveMultipleResults(ByVal connection As SqlConnection)
    Using connection
        Dim command As SqlCommand = New SqlCommand( _
          "SELECT CategoryID, CategoryName FROM Categories;" & _
          "SELECT EmployeeID, LastName FROM Employees", connection)
        connection.Open()

        Dim reader As SqlDataReader = command.ExecuteReader()

        Do While reader.HasRows
            Console.WriteLine(vbTab & reader.GetName(0) _
              & vbTab & reader.GetName(1))

            Do While reader.Read()
                Console.WriteLine(vbTab & reader.GetInt32(0) _
                  & vbTab & reader.GetString(1))
            Loop

            reader.NextResult()
        Loop
    End Using
End Sub
```

Getting schema information from the DataReader

While a **DataReader** is open, you can retrieve schema information about the current result set using the **GetSchemaTable** method. **GetSchemaTable** returns a DataTable object populated with rows and columns that contain the schema information for the current result set. The **DataTable** contains one row for each column of the result set. Each column of the schema table maps to a property of the columns returned in the rows of the result set, where the **ColumnName** is the name of the property and the value of the column is the value of the property. The following example writes out the schema information for **DataReader**.

```
static void GetSchemaInfo(SqlConnection connection)
{
    using (connection)
    {
        SqlCommand command = new SqlCommand(
          "SELECT CategoryID, CategoryName FROM Categories;",
          connection);
        connection.Open();

        SqlDataReader reader = command.ExecuteReader();
        DataTable schemaTable = reader.GetSchemaTable();

        foreach (DataRow row in schemaTable.Rows)
        {
            foreach (DataColumn column in schemaTable.Columns)
            {
                Console.WriteLine(String.Format("{0} = {1}",
                   column.ColumnName, row[column]));
            }
        }
    }
}
```

```
Private Sub GetSchemaInfo(ByVal connection As SqlConnection)
    Using connection
        Dim command As SqlCommand = New SqlCommand( _
          "SELECT CategoryID, CategoryName FROM Categories;", _
          connection)
        connection.Open()

        Dim reader As SqlDataReader = command.ExecuteReader()
        Dim schemaTable As DataTable = reader.GetSchemaTable()

        Dim row As DataRow
        Dim column As DataColumn

        For Each row In schemaTable.Rows
            For Each column In schemaTable.Columns
                Console.WriteLine(String.Format("{0} = {1}", _
                  column.ColumnName, row(column)))
            Next
            Console.WriteLine()
        Next
        reader.Close()
    End Using
End Sub
```

Working with OLE DB chapters

Hierarchical rowsets, or chapters (OLE DB type **DBTYPE_HCHAPTER**, ADO type **adChapter**), can be retrieved using the OleDbDataReader. When a query that includes a chapter is returned as a **DataReader**, the chapter is returned as a column in that **DataReader** and is exposed as a **DataReader** object.

The ADO.NET **DataSet** can also be used to represent hierarchical rowsets by using parent-child relationships between tables. For more information, see DataSets, DataTables, and DataViews.

The following code example uses the MSDataShape Provider to generate a chapter column of orders for each customer in a list of customers.

```
Using connection As OleDbConnection = New OleDbConnection(
    "Provider=MSDataShape;Data Provider=SQLOLEDB;" &
    "Data Source=localhost;Integrated Security=SSPI;Initial Catalog=northwind")

    Using custCMD As OleDbCommand = New OleDbCommand(
        "SHAPE {SELECT CustomerID, CompanyName FROM Customers} " &
        "APPEND ({SELECT CustomerID, OrderID FROM Orders} AS CustomerOrders " &
        "RELATE CustomerID TO CustomerID)", connection)

        connection.Open()

        Using custReader As OleDbDataReader = custCMD.ExecuteReader()

            Do While custReader.Read()
                Console.WriteLine("Orders for " & custReader.GetString(1))
                ' custReader.GetString(1) = CompanyName

                Using orderReader As OleDbDataReader = custReader.GetValue(2)
                    ' custReader.GetValue(2) = Orders chapter as DataReader

                    Do While orderReader.Read()
                        Console.WriteLine(vbTab & orderReader.GetInt32(1))
                        ' orderReader.GetInt32(1) = OrderID
                    Loop
                    orderReader.Close()
                End Using
            Loop
            ' Make sure to always close readers and connections.
            custReader.Close()
        End Using
    End Using
End Using
```

```
using (OleDbConnection connection = new OleDbConnection(
    "Provider=MSDataShape;Data Provider=SQLOLEDB;" +
    "Data Source=localhost;Integrated Security=SSPI;Initial Catalog=northwind"))
{
    using (OleDbCommand custCMD = new OleDbCommand(
        "SHAPE {SELECT CustomerID, CompanyName FROM Customers} " +
        "APPEND ({SELECT CustomerID, OrderID FROM Orders} AS CustomerOrders " +
        "RELATE CustomerID TO CustomerID)", connection))
    {
        connection.Open();

        using (OleDbDataReader custReader = custCMD.ExecuteReader())
        {

            while (custReader.Read())
            {
                Console.WriteLine("Orders for " + custReader.GetString(1));
                // custReader.GetString(1) = CompanyName

                using (OleDbDataReader orderReader = (OleDbDataReader)custReader.GetValue(2))
                {
                    // custReader.GetValue(2) = Orders chapter as DataReader

                    while (orderReader.Read())
                        Console.WriteLine("\t" + orderReader.GetInt32(1));
                    // orderReader.GetInt32(1) = OrderID
                    orderReader.Close();
                }
            }
            // Make sure to always close readers and connections.
            custReader.Close();
        }
    }
}
```

Returning results with Oracle REF CURSORs

The .NET Framework Data Provider for Oracle supports the use of Oracle REF CURSORs to return a query result. An Oracle REF CURSOR is returned as an OracleDataReader.

You can retrieve an OracleDataReader object that represents an Oracle REF CURSOR by using the ExecuteReader method. You can also specify an OracleCommand that returns one or more Oracle REF CURSORs as the **SelectCommand** for an OracleDataAdapter used to fill a DataSet.

To access a REF CURSOR returned from an Oracle data source, create an OracleCommand for your query and add an output parameter that references the REF CURSOR to the Parameters collection of your OracleCommand. The name of the parameter must match the name of the REF CURSOR parameter in your query. Set the type of the parameter to OracleType.Cursor. The OracleCommand.ExecuteReader() method of your OracleCommand returns an OracleDataReader for the REF CURSOR.

If your OracleCommand returns multiple REF CURSORS, add multiple output parameters. You can access the different REF CURSORs by calling the OracleCommand.ExecuteReader() method. The call to ExecuteReader() returns an OracleDataReader referencing the first REF CURSOR. You can then call the OracleDataReader.NextResult() method to access subsequent REF CURSORs. Although the parameters in your OracleCommand.Parameters collection match the REF CURSOR output parameters by name, the OracleDataReader accesses them in the order in which they were added to the Parameters collection.

For example, consider the following Oracle package and package body.

```
CREATE OR REPLACE PACKAGE CURSPKG AS
  TYPE T_CURSOR IS REF CURSOR;
  PROCEDURE OPEN_TWO_CURSORS (EMPCURSOR OUT T_CURSOR,
    DEPTCURSOR OUT T_CURSOR);
END CURSPKG;

CREATE OR REPLACE PACKAGE BODY CURSPKG AS
  PROCEDURE OPEN_TWO_CURSORS (EMPCURSOR OUT T_CURSOR,
    DEPTCURSOR OUT T_CURSOR)
  IS
  BEGIN
    OPEN EMPCURSOR FOR SELECT * FROM DEMO.EMPLOYEE;
    OPEN DEPTCURSOR FOR SELECT * FROM DEMO.DEPARTMENT;
  END OPEN_TWO_CURSORS;
END CURSPKG;
```

The following code creates an OracleCommand that returns the REF CURSORs from the previous Oracle package by adding two parameters of type OracleType.Cursor to the OracleCommand.Parameters collection.

```
Dim cursCmd As OracleCommand = New OracleCommand("CURSPKG.OPEN_TWO_CURSORS", oraConn)
cursCmd.Parameters.Add("EMPCURSOR", OracleType.Cursor).Direction = ParameterDirection.Output
cursCmd.Parameters.Add("DEPTCURSOR", OracleType.Cursor).Direction = ParameterDirection.Output
```

```
OracleCommand cursCmd = new OracleCommand("CURSPKG.OPEN_TWO_CURSORS", oraConn);
cursCmd.Parameters.Add("EMPCURSOR", OracleType.Cursor).Direction = ParameterDirection.Output;
cursCmd.Parameters.Add("DEPTCURSOR", OracleType.Cursor).Direction = ParameterDirection.Output;
```

The following code returns the results of the previous command using the Read() and NextResult() methods of the OracleDataReader. The REF CURSOR parameters are returned in order.

```
oraConn.Open()

Dim cursCmd As OracleCommand = New OracleCommand("CURSPKG.OPEN_TWO_CURSORS", oraConn)
cursCmd.CommandType = CommandType.StoredProcedure
cursCmd.Parameters.Add("EMPCURSOR", OracleType.Cursor).Direction = ParameterDirection.Output
cursCmd.Parameters.Add("DEPTCURSOR", OracleType.Cursor).Direction = ParameterDirection.Output

Dim reader As OracleDataReader = cursCmd.ExecuteReader()

Console.WriteLine(vbCrLf & "Emp ID" & vbTab & "Name")

Do While reader.Read()
  Console.WriteLine("{0}" & vbTab & "{1}, {2}", reader.GetOracleNumber(0), reader.GetString(1),
reader.GetString(2))
Loop

reader.NextResult()

Console.WriteLine(vbCrLf & "Dept ID" & vbTab & "Name")

Do While reader.Read()
  Console.WriteLine("{0}" & vbTab & "{1}", reader.GetOracleNumber(0), reader.GetString(1))
Loop
' Make sure to always close readers and connections.
reader.Close()
oraConn.Close()
```

```
oraConn.Open();

OracleCommand cursCmd = new OracleCommand("CURSPKG.OPEN_TWO_CURSORS", oraConn);
cursCmd.CommandType = CommandType.StoredProcedure;
cursCmd.Parameters.Add("EMPCURSOR", OracleType.Cursor).Direction = ParameterDirection.Output;
cursCmd.Parameters.Add("DEPTCURSOR", OracleType.Cursor).Direction = ParameterDirection.Output;

OracleDataReader reader = cursCmd.ExecuteReader();

Console.WriteLine("\nEmp ID\tName");

while (reader.Read())
  Console.WriteLine("{0}\t{1}, {2}", reader.GetOracleNumber(0), reader.GetString(1), reader.GetString(2));

reader.NextResult();

Console.WriteLine("\nDept ID\tName");

while (reader.Read())
  Console.WriteLine("{0}\t{1}", reader.GetOracleNumber(0), reader.GetString(1));
// Make sure to always close readers and connections.
reader.Close();
oraConn.Close();
```

The following example uses the previous command to populate a DataSet with the results of the Oracle package.

```
Dim ds As DataSet = New DataSet()

Dim adapter As OracleDataAdapter = New OracleDataAdapter(cursCmd)
adapter.TableMappings.Add("Table", "Employees")
adapter.TableMappings.Add("Table1", "Departments")

adapter.Fill(ds)
```

```
DataSet ds = new DataSet();

OracleDataAdapter adapter = new OracleDataAdapter(cursCmd);
adapter.TableMappings.Add("Table", "Employees");
adapter.TableMappings.Add("Table1", "Departments");

adapter.Fill(ds);
```

NOTE

To avoid an **OverflowException**, we recommend that you also handle any conversion from the Oracle NUMBER type to a valid .NET Framework type before storing the value in a DataRow. You can use the FillError event to determine if an **OverflowException** has occurred. For more information on the FillError event, see Handling DataAdapter Events.

See also

- DataAdapters and DataReaders
- Commands and Parameters
- Retrieving Database Schema Information
- ADO.NET Overview

Populating a DataSet from a DataAdapter

3/12/2020 • 9 minutes to read • Edit Online

The ADO.NET DataSet is a memory-resident representation of data that provides a consistent relational programming model independent of the data source. The `DataSet` represents a complete set of data that includes tables, constraints, and relationships among the tables. Because the `DataSet` is independent of the data source, a `DataSet` can include data local to the application, and data from multiple data sources. Interaction with existing data sources is controlled through the `DataAdapter`.

The `SelectCommand` property of the `DataAdapter` is a `Command` object that retrieves data from the data source. The `InsertCommand`, `UpdateCommand`, and `DeleteCommand` properties of the `DataAdapter` are `Command` objects that manage updates to the data in the data source according to modifications made to the data in the `DataSet`. These properties are covered in more detail in Updating Data Sources with DataAdapters.

The `Fill` method of the `DataAdapter` is used to populate a `DataSet` with the results of the `SelectCommand` of the `DataAdapter`. `Fill` takes as its arguments a `DataSet` to be populated, and a `DataTable` object, or the name of the `DataTable` to be filled with the rows returned from the `SelectCommand`.

> **NOTE**
>
> Using the `DataAdapter` to retrieve all of a table takes time, especially if there are many rows in the table. This is because accessing the database, locating and processing the data, and then transferring the data to the client is time-consuming. Pulling all of the table to the client also locks all of the rows on the server. To improve performance, you can use the `WHERE` clause to greatly reduce the number of rows returned to the client. You can also reduce the amount of data returned to the client by only explicitly listing required columns in the `SELECT` statement. Another good workaround is to retrieve the rows in batches (such as several hundred rows at a time) and only retrieve the next batch when the client is finished with the current batch.

The `Fill` method uses the `DataReader` object implicitly to return the column names and types that are used to create the tables in the `DataSet`, and the data to populate the rows of the tables in the `DataSet`. Tables and columns are only created if they do not already exist; otherwise `Fill` uses the existing `DataSet` schema. Column types are created as .NET Framework types according to the tables in Data Type Mappings in ADO.NET. Primary keys are not created unless they exist in the data source and `DataAdapter.MissingSchemaAction` is set to `MissingSchemaAction.AddWithKey`. If `Fill` finds that a primary key exists for a table, it will overwrite data in the `DataSet` with data from the data source for rows where the primary key column values match those of the row returned from the data source. If no primary key is found, the data is appended to the tables in the `DataSet`. `Fill` uses any mappings that may exist when you populate the `DataSet` (see DataAdapter DataTable and DataColumn Mappings).

> **NOTE**
>
> If the `SelectCommand` returns the results of an OUTER JOIN, the `DataAdapter` does not set a `PrimaryKey` value for the resulting `DataTable`. You must define the `PrimaryKey` yourself to make sure that duplicate rows are resolved correctly. For more information, see Defining Primary Keys.

The following code example creates an instance of a SqlDataAdapter that uses a SqlConnection to the Microsoft SQL Server `Northwind` database and populates a DataTable in a `DataSet` with the list of customers. The SQL statement and SqlConnection arguments passed to the SqlDataAdapter constructor are used to create the SelectCommand property of the SqlDataAdapter.

Example

```
' Assumes that connection is a valid SqlConnection object.
Dim queryString As String = _
  "SELECT CustomerID, CompanyName FROM dbo.Customers"
Dim adapter As SqlDataAdapter = New SqlDataAdapter( _
  queryString, connection)

Dim customers As DataSet = New DataSet
adapter.Fill(customers, "Customers")
```

```
// Assumes that connection is a valid SqlConnection object.
string queryString =
  "SELECT CustomerID, CompanyName FROM dbo.Customers";
SqlDataAdapter adapter = new SqlDataAdapter(queryString, connection);

DataSet customers = new DataSet();
adapter.Fill(customers, "Customers");
```

NOTE

The code shown in this example does not explicitly open and close the `Connection`. The `Fill` method implicitly opens the `Connection` that the `DataAdapter` is using if it finds that the connection is not already open. If `Fill` opened the connection, it also closes the connection when `Fill` is finished. This can simplify your code when you deal with a single operation such as a `Fill` or an `Update`. However, if you are performing multiple operations that require an open connection, you can improve the performance of your application by explicitly calling the `Open` method of the `Connection`, performing the operations against the data source, and then calling the `Close` method of the `Connection`. You should try to keep connections to the data source open as briefly as possible to free resources for use by other client applications.

Multiple Result Sets

If the `DataAdapter` encounters multiple result sets, it creates multiple tables in the `DataSet`. The tables are given an incremental default name of Table*N*, starting with "Table" for Table0. If a table name is passed as an argument to the `Fill` method, the tables are given an incremental default name of TableName*N*, starting with "TableName" for TableName0.

Populating a DataSet from Multiple DataAdapters

Any number of `DataAdapter` objects can be used with a `DataSet`. Each `DataAdapter` can be used to fill one or more `DataTable` objects and resolve updates back to the relevant data source. `DataRelation` and `Constraint` objects can be added to the `DataSet` locally, which enables you to relate data from dissimilar data sources. For example, a `DataSet` can contain data from a Microsoft SQL Server database, an IBM DB2 database exposed through OLE DB, and a data source that streams XML. One or more `DataAdapter` objects can handle communication to each data source.

Example

The following code example populates a list of customers from the `Northwind` database on Microsoft SQL Server, and a list of orders from the `Northwind` database stored in Microsoft Access 2000. The filled tables are related with a `DataRelation`, and the list of customers is then displayed with the orders for that customer. For more information about `DataRelation` objects, see Adding DataRelations and Navigating DataRelations.

```
' Assumes that customerConnection is a valid SqlConnection object.
' Assumes that orderConnection is a valid OleDbConnection object.
Dim custAdapter As SqlDataAdapter = New SqlDataAdapter( _
  "SELECT * FROM dbo.Customers", customerConnection)

Dim ordAdapter As OleDbDataAdapter = New OleDbDataAdapter( _
  "SELECT * FROM Orders", orderConnection)

Dim customerOrders As DataSet = New DataSet()
custAdapter.Fill(customerOrders, "Customers")
ordAdapter.Fill(customerOrders, "Orders")

Dim relation As DataRelation = _
  customerOrders.Relations.Add("CustOrders", _
  customerOrders.Tables("Customers").Columns("CustomerID"), _
  customerOrders.Tables("Orders").Columns("CustomerID"))

Dim pRow, cRow As DataRow
For Each pRow In customerOrders.Tables("Customers").Rows
  Console.WriteLine(pRow("CustomerID").ToString())

  For Each cRow In pRow.GetChildRows(relation)
    Console.WriteLine(vbTab & cRow("OrderID").ToString())
  Next
Next
```

```
// Assumes that customerConnection is a valid SqlConnection object.
// Assumes that orderConnection is a valid OleDbConnection object.
SqlDataAdapter custAdapter = new SqlDataAdapter(
  "SELECT * FROM dbo.Customers", customerConnection);
OleDbDataAdapter ordAdapter = new OleDbDataAdapter(
  "SELECT * FROM Orders", orderConnection);

DataSet customerOrders = new DataSet();

custAdapter.Fill(customerOrders, "Customers");
ordAdapter.Fill(customerOrders, "Orders");

DataRelation relation = customerOrders.Relations.Add("CustOrders",
  customerOrders.Tables["Customers"].Columns["CustomerID"],
  customerOrders.Tables["Orders"].Columns["CustomerID"]);

foreach (DataRow pRow in customerOrders.Tables["Customers"].Rows)
{
  Console.WriteLine(pRow["CustomerID"]);
   foreach (DataRow cRow in pRow.GetChildRows(relation))
    Console.WriteLine("\t" + cRow["OrderID"]);
}
```

SQL Server Decimal Type

By default, the `DataSet` stores data by using .NET Framework data types. For most applications, these provide a convenient representation of data source information. However, this representation may cause a problem when the data type in the data source is a SQL Server decimal or numeric data type. The .NET Framework `decimal` data type allows a maximum of 28 significant digits, whereas the SQL Server `decimal` data type allows 38 significant digits. If the `SqlDataAdapter` determines during a `Fill` operation that the precision of a SQL Server `decimal` field is larger than 28 characters, the current row is not added to the `DataTable`. Instead the `FillError` event occurs, which enables you to determine whether a loss of precision will occur, and respond appropriately. For more information about the `FillError` event, see Handling DataAdapter Events. To get the SQL Server `decimal` value, you can also use a SqlDataReader object and call the GetSqlDecimal method.

ADO.NET 2.0 introduced enhanced support for System.Data.SqlTypes in the `DataSet`. For more information, see SqlTypes and the DataSet.

OLE DB Chapters

Hierarchical rowsets, or chapters (OLE DB type `DBTYPE_HCHAPTER`, ADO type `adChapter`) can be used to fill the contents of a `DataSet`. When the OleDbDataAdapter encounters a chaptered column during a `Fill` operation, a `DataTable` is created for the chaptered column, and that table is filled with the columns and rows from the chapter. The table created for the chaptered column is named by using both the parent table name and the chaptered column name in the form "*ParentTableNameChapteredColumnName*". If a table already exists in the `DataSet` that matches the name of the chaptered column, the current table is filled with the chapter data. If there is no column in an existing table that matches a column found in the chapter, a new column is added.

Before the tables in the `DataSet` are filled with the data in the chaptered columns, a relation is created between the parent and child tables of the hierarchical rowset by adding an integer column to both the parent and child table, setting the parent column to auto-increment, and creating a `DataRelation` using the added columns from both tables. The added relation is named by using the parent table and chapter column names in the form "*ParentTableNameChapterColumnName*".

Note that the related column only exists in the `DataSet`. Subsequent fills from the data source can cause new rows to be added to the tables instead of changes being merged into existing rows.

Note also that, if you use the `DataAdapter.Fill` overload that takes a `DataTable`, only that table will be filled. An auto-incrementing integer column will still be added to the table, but no child table will be created or filled, and no relation will be created.

The following example uses the MSDataShape Provider to generate a chapter column of orders for each customer in a list of customers. A `DataSet` is then filled with the data.

```
Using connection As OleDbConnection = New OleDbConnection( _
  "Provider=MSDataShape;Data Provider=SQLOLEDB;" & _
  "Data Source=(local);Integrated " & _
  "Security=SSPI;Initial Catalog=northwind")

Dim adapter As OleDbDataAdapter = New OleDbDataAdapter( _
  "SHAPE {SELECT CustomerID, CompanyName FROM Customers} " & _
  "APPEND ({SELECT CustomerID, OrderID FROM Orders} AS Orders " & _
  "RELATE CustomerID TO CustomerID)", connection)

Dim customers As DataSet = New DataSet()

adapter.Fill(customers, "Customers")
End Using
```

```
using (OleDbConnection connection = new OleDbConnection("Provider=MSDataShape;Data Provider=SQLOLEDB;" +
  "Data Source=(local);Integrated Security=SSPI;Initial Catalog=northwind"))
{
OleDbDataAdapter adapter = new OleDbDataAdapter("SHAPE {SELECT CustomerID, CompanyName FROM Customers} " +
  "APPEND ({SELECT CustomerID, OrderID FROM Orders} AS Orders " +
  "RELATE CustomerID TO CustomerID)", connection);

DataSet customers = new DataSet();
adapter.Fill(customers, "Customers");
}
```

When the `Fill` operation is complete, the `DataSet` contains two tables: `Customers` and `CustomersOrders`, where `CustomersOrders` represents the chaptered column. An additional column named `Orders` is added to the

`Customers` table, and an additional column named `CustomersOrders` is added to the `CustomersOrders` table. The `Orders` column in the `Customers` table is set to auto-increment. A `DataRelation`, `CustomersOrders`, is created by using the columns that were added to the tables with `Customers` as the parent table. The following tables show some sample results.

TableName: Customers

CUSTOMERID	COMPANYNAME	ORDERS
ALFKI	Alfreds Futterkiste	0
ANATR	Ana Trujillo Emparedados y helados	1

TableName: CustomersOrders

CUSTOMERID	ORDERID	CUSTOMERSORDERS
ALFKI	10643	0
ALFKI	10692	0
ANATR	10308	1
ANATR	10625	1

See also

- DataAdapters and DataReaders
- Data Type Mappings in ADO.NET
- Modifying Data with a DbDataAdapter
- Multiple Active Result Sets (MARS)
- ADO.NET Overview

DataAdapter Parameters

3/12/2020 • 7 minutes to read • Edit Online

The DbDataAdapter has four properties that are used to retrieve data from and update data to the data source: the SelectCommand property returns data from the data source; and the InsertCommand , UpdateCommand, and DeleteCommand properties are used to manage changes at the data source. The `SelectCommand` property must be set before you call the `Fill` method of the `DataAdapter`. The `InsertCommand`, `UpdateCommand`, or `DeleteCommand` properties must be set before the `Update` method of the `DataAdapter` is called, depending on what changes were made to the data in the DataTable. For example, if rows have been added, the `InsertCommand` must be set before you call `Update`. When `Update` is processing an inserted, updated, or deleted row, the `DataAdapter` uses the respective `Command` property to process the action. Current information about the modified row is passed to the `Command` object through the `Parameters` collection.

When you update a row at the data source, you call the UPDATE statement, which uses a unique identifier to identify the row in the table to be updated. The unique identifier is typically the value of a primary key field. The UPDATE statement uses parameters that contain both the unique identifier and the columns and values to be updated, as shown in the following Transact-SQL statement.

```
UPDATE Customers SET CompanyName = @CompanyName
  WHERE CustomerID = @CustomerID
```

> **NOTE**
>
> The syntax for parameter placeholders depends on the data source. This example shows placeholders for a SQL Server data source. Use question mark (?) placeholders for System.Data.OleDb and System.Data.Odbc parameters.

In this Visual Basic example, the `CompanyName` field is updated with the value of the `@CompanyName` parameter for the row where `CustomerID` equals the value of the `@CustomerID` parameter. The parameters retrieve information from the modified row using the SourceColumn property of the SqlParameter object. The following are the parameters for the previous sample UPDATE statement. The code assumes that the variable `adapter` represents a valid SqlDataAdapter object.

```
adapter.Parameters.Add( _
  "@CompanyName", SqlDbType.NChar, 15, "CompanyName")
Dim parameter As SqlParameter = _
  adapter.UpdateCommand.Parameters.Add("@CustomerID", _
  SqlDbType.NChar, 5, "CustomerID")
parameter.SourceVersion = DataRowVersion.Original
```

The `Add` method of the `Parameters` collection takes the name of the parameter, the data type, the size (if applicable to the type), and the name of the SourceColumn from the `DataTable`. Notice that the SourceVersion of the `@CustomerID` parameter is set to `Original`. This guarantees that the existing row in the data source is updated if the value of the identifying column or columns has been changed in the modified DataRow. In that case, the `Original` row value would match the current value at the data source, and the `Current` row value would contain the updated value. The `SourceVersion` for the `@CompanyName` parameter is not set and uses the default, `Current` row value.

NOTE

For both the `Fill` operations of the `DataAdapter` and the `Get` methods of the `DataReader`, the .NET Framework type is inferred from the type returned from the .NET Framework data provider. The inferred .NET Framework types and accessor methods for Microsoft SQL Server, OLE DB, and ODBC data types are described in Data Type Mappings in ADO.NET.

Parameter.SourceColumn, Parameter.SourceVersion

The `SourceColumn` and `SourceVersion` may be passed as arguments to the `Parameter` constructor, or set as properties of an existing `Parameter`. The `SourceColumn` is the name of the DataColumn from the DataRow where the value of the `Parameter` will be retrieved. The `SourceVersion` specifies the `DataRow` version that the `DataAdapter` uses to retrieve the value.

The following table shows the DataRowVersion enumeration values available for use with `SourceVersion`.

DATAROWVERSION ENUMERATION	DESCRIPTION
`Current`	The parameter uses the current value of the column. This is the default.
`Default`	The parameter uses the `DefaultValue` of the column.
`Original`	The parameter uses the original value of the column.
`Proposed`	The parameter uses a proposed value.

The `SqlClient` code example in the next section defines a parameter for an UpdateCommand in which the `CustomerID` column is used as a `SourceColumn` for two parameters: `@CustomerID` (`SET CustomerID = @CustomerID`), and `@OldCustomerID` (`WHERE CustomerID = @OldCustomerID`). The `@CustomerID` parameter is used to update the **CustomerID** column to the current value in the `DataRow`. As a result, the `CustomerID` `SourceColumn` with a `SourceVersion` of `Current` is used. The `@OldCustomerID` parameter is used to identify the current row in the data source. Because the matching column value is found in the `Original` version of the row, the same `SourceColumn` (`CustomerID`) with a `SourceVersion` of `Original` is used.

Working with SqlClient Parameters

The following example demonstrates how to create a SqlDataAdapter and set the MissingSchemaAction to AddWithKey in order to retrieve additional schema information from the database. The SelectCommand, InsertCommand, UpdateCommand, and DeleteCommand properties set and their corresponding SqlParameter objects added to the Parameters collection. The method returns a `SqlDataAdapter` object.

```
public static SqlDataAdapter CreateSqlDataAdapter(SqlConnection connection)
{
    SqlDataAdapter adapter = new SqlDataAdapter();
    adapter.MissingSchemaAction = MissingSchemaAction.AddWithKey;

    // Create the commands.
    adapter.SelectCommand = new SqlCommand(
        "SELECT CustomerID, CompanyName FROM CUSTOMERS", connection);
    adapter.InsertCommand = new SqlCommand(
        "INSERT INTO Customers (CustomerID, CompanyName) " +
        "VALUES (@CustomerID, @CompanyName)", connection);
    adapter.UpdateCommand = new SqlCommand(
        "UPDATE Customers SET CustomerID = @CustomerID, CompanyName = @CompanyName " +
        "WHERE CustomerID = @oldCustomerID", connection);
    adapter.DeleteCommand = new SqlCommand(
        "DELETE FROM Customers WHERE CustomerID = @CustomerID", connection);

    // Create the parameters.
    adapter.InsertCommand.Parameters.Add("@CustomerID",
        SqlDbType.Char, 5, "CustomerID");
    adapter.InsertCommand.Parameters.Add("@CompanyName",
        SqlDbType.VarChar, 40, "CompanyName");

    adapter.UpdateCommand.Parameters.Add("@CustomerID",
        SqlDbType.Char, 5, "CustomerID");
    adapter.UpdateCommand.Parameters.Add("@CompanyName",
        SqlDbType.VarChar, 40, "CompanyName");
    adapter.UpdateCommand.Parameters.Add("@oldCustomerID",
        SqlDbType.Char, 5, "CustomerID").SourceVersion =
        DataRowVersion.Original;

    adapter.DeleteCommand.Parameters.Add("@CustomerID",
        SqlDbType.Char, 5, "CustomerID").SourceVersion =
        DataRowVersion.Original;

    return adapter;
}
```

```
Public Function CreateSqlDataAdapter( _
    ByVal connection As SqlConnection) As SqlDataAdapter

    Dim adapter As New SqlDataAdapter()
    adapter.MissingSchemaAction = MissingSchemaAction.AddWithKey

    ' Create the commands.
    adapter.SelectCommand = New SqlCommand( _
        "SELECT CustomerID, CompanyName FROM CUSTOMERS", connection)
    adapter.InsertCommand = New SqlCommand( _
        "INSERT INTO Customers (CustomerID, CompanyName) " & _
         "VALUES (@CustomerID, @CompanyName)", connection)
    adapter.UpdateCommand = New SqlCommand( _
        "UPDATE Customers SET CustomerID = @CustomerID, CompanyName = " & _
        "@CompanyName WHERE CustomerID = @oldCustomerID", connection)
    adapter.DeleteCommand = New SqlCommand( _
        "DELETE FROM Customers WHERE CustomerID = @CustomerID", connection)

    ' Create the parameters.
    adapter.InsertCommand.Parameters.Add("@CustomerID", _
        SqlDbType.Char, 5, "CustomerID")
    adapter.InsertCommand.Parameters.Add("@CompanyName", _
        SqlDbType.VarChar, 40, "CompanyName")

    adapter.UpdateCommand.Parameters.Add("@CustomerID", _
        SqlDbType.Char, 5, "CustomerID")
    adapter.UpdateCommand.Parameters.Add("@CompanyName", _
        SqlDbType.VarChar, 40, "CompanyName")
    adapter.UpdateCommand.Parameters.Add("@oldCustomerID", _
        SqlDbType.Char, 5, "CustomerID").SourceVersion = _
        DataRowVersion.Original

    adapter.DeleteCommand.Parameters.Add("@CustomerID", _
        SqlDbType.Char, 5, "CustomerID").SourceVersion = _
        DataRowVersion.Original

    Return adapter
End Function
```

OleDb Parameter Placeholders

For the OleDbDataAdapter and OdbcDataAdapter objects, you must use question mark (?) placeholders to identify the parameters.

```
Dim selectSQL As String = _
  "SELECT CustomerID, CompanyName FROM Customers " & _
  "WHERE CountryRegion = ? AND City = ?"
Dim insertSQL AS String = _
  "INSERT INTO Customers (CustomerID, CompanyName) VALUES (?, ?)"
Dim updateSQL AS String = _
  "UPDATE Customers SET CustomerID = ?, CompanyName = ? " & _
  WHERE CustomerID = ?"
Dim deleteSQL As String = "DELETE FROM Customers WHERE CustomerID = ?"
```

```
string selectSQL =
  "SELECT CustomerID, CompanyName FROM Customers " +
  "WHERE CountryRegion = ? AND City = ?";
string insertSQL =
  "INSERT INTO Customers (CustomerID, CompanyName) " +
  "VALUES (?, ?)";
string updateSQL =
  "UPDATE Customers SET CustomerID = ?, CompanyName = ? " +
  "WHERE CustomerID = ? ";
string deleteSQL = "DELETE FROM Customers WHERE CustomerID = ?";
```

The parameterized query statements define which input and output parameters must be created. To create a parameter, use the `Parameters.Add` method or the `Parameter` constructor to specify the column name, data type, and size. For intrinsic data types, such as `Integer`, you do not have to include the size, or you can specify the default size.

The following code example creates the parameters for a SQL statement and then fills a `DataSet`.

OleDb Example

```
' Assumes that connection is a valid OleDbConnection object.
Dim adapter As OleDbDataAdapter = New OleDbDataAdapter

Dim selectCMD AS OleDbCommand = New OleDbCommand(selectSQL, connection)
adapter.SelectCommand = selectCMD

' Add parameters and set values.
selectCMD.Parameters.Add( _
  "@CountryRegion", OleDbType.VarChar, 15).Value = "UK"
selectCMD.Parameters.Add( _
  "@City", OleDbType.VarChar, 15).Value = "London"

Dim customers As DataSet = New DataSet
adapter.Fill(customers, "Customers")
```

```
// Assumes that connection is a valid OleDbConnection object.
OleDbDataAdapter adapter = new OleDbDataAdapter();

OleDbCommand selectCMD = new OleDbCommand(selectSQL, connection);
adapter.SelectCommand = selectCMD;

// Add parameters and set values.
selectCMD.Parameters.Add(
  "@CountryRegion", OleDbType.VarChar, 15).Value = "UK";
selectCMD.Parameters.Add(
  "@City", OleDbType.VarChar, 15).Value = "London";

DataSet customers = new DataSet();
adapter.Fill(customers, "Customers");
```

Odbc Parameters

```
' Assumes that connection is a valid OdbcConnection object.
Dim adapter As OdbcDataAdapter = New OdbcDataAdapter

Dim selectCMD AS OdbcCommand = New OdbcCommand(selectSQL, connection)
adapter.SelectCommand = selectCMD

' Add Parameters and set values.
selectCMD.Parameters.Add("@CountryRegion", OdbcType.VarChar, 15).Value = "UK"
selectCMD.Parameters.Add("@City", OdbcType.VarChar, 15).Value = "London"

Dim customers As DataSet = New DataSet
adapter.Fill(customers, "Customers")
```

```
// Assumes that connection is a valid OdbcConnection object.
OdbcDataAdapter adapter = new OdbcDataAdapter();

OdbcCommand selectCMD = new OdbcCommand(selectSQL, connection);
adapter.SelectCommand = selectCMD;

//Add Parameters and set values.
selectCMD.Parameters.Add("@CountryRegion", OdbcType.VarChar, 15).Value = "UK";
selectCMD.Parameters.Add("@City", OdbcType.VarChar, 15).Value = "London";

DataSet customers = new DataSet();
adapter.Fill(customers, "Customers");
```

NOTE

If a parameter name is not supplied for a parameter, the parameter is given an incremental default name of Parameter*N*, starting with "Parameter1". We recommend that you avoid the Parameter*N* naming convention when you supply a parameter name, because the name that you supply might conflict with an existing default parameter name in the `ParameterCollection`. If the supplied name already exists, an exception is thrown.

See also

- DataAdapters and DataReaders
- Commands and Parameters
- Updating Data Sources with DataAdapters
- Modifying Data with Stored Procedures
- Data Type Mappings in ADO.NET
- ADO.NET Overview

Adding Existing Constraints to a DataSet

10/17/2019 • 2 minutes to read • Edit Online

The **Fill** method of the **DataAdapter** fills a DataSet only with table columns and rows from a data source; though constraints are commonly set by the data source, the **Fill** method does not add this schema information to the **DataSet** by default. To populate a **DataSet** with existing primary key constraint information from a data source, you can either call the **FillSchema** method of the **DataAdapter**, or set the **MissingSchemaAction** property of the **DataAdapter** to **AddWithKey** before calling **Fill**. This will ensure that primary key constraints in the **DataSet** reflect those at the data source. Foreign key constraint information is not included and must be created explicitly, as shown in DataTable Constraints.

Adding schema information to a **DataSet** before filling it with data ensures that primary key constraints are included with the DataTable objects in the **DataSet**. As a result, when additional calls to fill the **DataSet** are made, the primary key column information is used to match new rows from the data source with current rows in each **DataTable**, and current data in the tables is overwritten with data from the data source. Without the schema information, the new rows from the data source are appended to the **DataSet**, resulting in duplicate rows.

> **NOTE**
>
> If a column in a data source is identified as auto-incrementing, the **FillSchema** method, or the **Fill** method with a **MissingSchemaAction** of **AddWithKey**, creates a **DataColumn** with an **AutoIncrement** property set to `true`. However, you will need to set the **AutoIncrementStep** and **AutoIncrementSeed** values yourself. For more information about auto-incrementing columns, see Creating AutoIncrement Columns.

Using **FillSchema** or setting the **MissingSchemaAction** to **AddWithKey** requires extra processing at the data source to determine primary key column information. This additional processing can hinder performance. If you know the primary key information at design time, we recommend that you explicitly specify the primary key column or columns in order to achieve optimal performance. For information about explicitly setting primary key information for a table, see Defining Primary Keys.

The following code example shows how to add schema information to a **DataSet** using **FillSchema**:

```
Dim custDataSet As New DataSet()

custAdapter.FillSchema(custDataSet, SchemaType.Source, "Customers")
custAdapter.Fill(custDataSet, "Customers")
```

```
var custDataSet = new DataSet();

custAdapter.FillSchema(custDataSet, SchemaType.Source, "Customers");
custAdapter.Fill(custDataSet, "Customers");
```

The following code example shows how to add schema information to a **DataSet** using the **MissingSchemaAction.AddWithKey** property of the **Fill** method:

```
Dim custDataSet As New DataSet()

custAdapter.MissingSchemaAction = MissingSchemaAction.AddWithKey
custAdapter.Fill(custDataSet, "Customers")
```

```
var custDataSet = new DataSet();

custAdapter.MissingSchemaAction = MissingSchemaAction.AddWithKey;
custAdapter.Fill(custDataSet, "Customers");
```

Handling multiple result sets

If the **DataAdapter** encounters multiple result sets returned from the **SelectCommand**, it will create multiple tables in the **DataSet**. The tables will be given a zero-based incremental default name of **Table** *N*, starting with **Table** instead of "Table0". If a table name is passed as an argument to the **FillSchema** method, the tables will be given a zero-based incremental name of **TableName** *N*, starting with **TableName** instead of "TableName0".

> **NOTE**
>
> If the **FillSchema** method of the **OleDbDataAdapter** object is called for a command that returns multiple result sets, only the schema information from the first result set is returned. When returning schema information for multiple result sets using the **OleDbDataAdapter**, it is recommended that you specify a **MissingSchemaAction** of **AddWithKey** and obtain the schema information when calling the **Fill** method.

See also

- DataAdapters and DataReaders
- DataSets, DataTables, and DataViews
- Retrieving and Modifying Data in ADO.NET
- ADO.NET Overview

DataAdapter DataTable and DataColumn Mappings

3/12/2020 • 3 minutes to read • Edit Online

A **DataAdapter** contains a collection of zero or more DataTableMapping objects in its **TableMappings** property. A **DataTableMapping** provides a master mapping between the data returned from a query against a data source, and a DataTable. The **DataTableMapping** name can be passed in place of the **DataTable** name to the **Fill** method of the **DataAdapter**. The following example creates a **DataTableMapping** named **AuthorsMapping** for the **Authors** table.

```
workAdapter.TableMappings.Add("AuthorsMapping", "Authors")
```

```
workAdapter.TableMappings.Add("AuthorsMapping", "Authors");
```

A **DataTableMapping** enables you to use column names in a **DataTable** that are different from those in the database. The **DataAdapter** uses the mapping to match the columns when the table is updated.

If you do not specify a **TableName** or a **DataTableMapping** name when calling the **Fill** or **Update** method of the **DataAdapter**, the **DataAdapter** looks for a **DataTableMapping** named "Table". If that **DataTableMapping** does not exist, the **TableName** of the **DataTable** is "Table". You can specify a default **DataTableMapping** by creating a **DataTableMapping** with the name of "Table".

The following code example creates a **DataTableMapping** (from the System.Data.Common namespace) and makes it the default mapping for the specified **DataAdapter** by naming it "Table". The example then maps the columns from the first table in the query result (the **Customers** table of the **Northwind** database) to a set of more user-friendly names in the **Northwind Customers** table in the DataSet. For columns that are not mapped, the name of the column from the data source is used.

```
Dim mapping As DataTableMapping = _
  adapter.TableMappings.Add("Table", "NorthwindCustomers")
mapping.ColumnMappings.Add("CompanyName", "Company")
mapping.ColumnMappings.Add("ContactName", "Contact")
mapping.ColumnMappings.Add("PostalCode", "ZIPCode")

adapter.Fill(custDS)
```

```
DataTableMapping mapping =
  adapter.TableMappings.Add("Table", "NorthwindCustomers");
mapping.ColumnMappings.Add("CompanyName", "Company");
mapping.ColumnMappings.Add("ContactName", "Contact");
mapping.ColumnMappings.Add("PostalCode", "ZIPCode");

adapter.Fill(custDS);
```

In more advanced situations, you may decide that you want the same **DataAdapter** to support loading different tables with different mappings. To do this, simply add additional **DataTableMapping** objects.

When the **Fill** method is passed an instance of a **DataSet** and a **DataTableMapping** name, if a mapping with that name exists it is used; otherwise, a **DataTable** with that name is used.

The following examples create a **DataTableMapping** with a name of **Customers** and a **DataTable** name of **BizTalkSchema**. The example then maps the rows returned by the SELECT statement to the **BizTalkSchema**

DataTable.

```
Dim mapping As ITableMapping = _
  adapter.TableMappings.Add("Customers", "BizTalkSchema")
mapping.ColumnMappings.Add("CustomerID", "ClientID")
mapping.ColumnMappings.Add("CompanyName", "ClientName")
mapping.ColumnMappings.Add("ContactName", "Contact")
mapping.ColumnMappings.Add("PostalCode", "ZIP")

adapter.Fill(custDS, "Customers")
```

```
ITableMapping mapping =
  adapter.TableMappings.Add("Customers", "BizTalkSchema");
mapping.ColumnMappings.Add("CustomerID", "ClientID");
mapping.ColumnMappings.Add("CompanyName", "ClientName");
mapping.ColumnMappings.Add("ContactName", "Contact");
mapping.ColumnMappings.Add("PostalCode", "ZIP");

adapter.Fill(custDS, "Customers");
```

NOTE

If a source column name is not supplied for a column mapping or a source table name is not supplied for a table mapping, default names will be automatically generated. If no source column is supplied for a column mapping, the column mapping is given an incremental default name of **SourceColumn** *N*, starting with **SourceColumn1**. If no source table name is supplied for a table mapping, the table mapping is given an incremental default name of **SourceTable** *N*, starting with **SourceTable1**.

NOTE

We recommend that you avoid the naming convention of **SourceColumn** *N* for a column mapping, or **SourceTable** *N* for a table mapping, because the name you supply may conflict with an existing default column mapping name in the **ColumnMappingCollection** or table mapping name in the **DataTableMappingCollection**. If the supplied name already exists, an exception will be thrown.

Handling Multiple Result Sets

If your **SelectCommand** returns multiple tables, **Fill** automatically generates table names with incremental values for the tables in the **DataSet**, starting with the specified table name and continuing on in the form **TableName** *N*, starting with **TableName1**. You can use table mappings to map the automatically generated table name to a name you want specified for the table in the **DataSet**. For example, for a **SelectCommand** that returns two tables, **Customers** and **Orders**, issue the following call to **Fill**.

```
adapter.Fill(customersDataSet, "Customers")
```

```
adapter.Fill(customersDataSet, "Customers");
```

Two tables are created in the **DataSet**: **Customers** and **Customers1**. You can use table mappings to ensure that the second table is named **Orders** instead of **Customers1**. To do this, map the source table of **Customers1** to the **DataSet** table **Orders**, as shown in the following example.

```
adapter.TableMappings.Add("Customers1", "Orders")
adapter.Fill(customersDataSet, "Customers")
```

```
adapter.TableMappings.Add("Customers1", "Orders");
adapter.Fill(customersDataSet, "Customers");
```

See also

- DataAdapters and DataReaders
- Retrieving and Modifying Data in ADO.NET
- ADO.NET Overview

Paging Through a Query Result

3/12/2020 • 3 minutes to read • Edit Online

Paging through a query result is the process of returning the results of a query in smaller subsets of data, or pages. This is a common practice for displaying results to a user in small, easy-to-manage chunks.

The **DataAdapter** provides a facility for returning only a page of data, through overloads of the **Fill** method. However, this might not be the best choice for paging through large query results because, although the **DataAdapter** fills the target DataTable or DataSet with only the requested records, the resources to return the entire query are still used. To return a page of data from a data source without using the resources to return the entire query, specify additional criteria for your query that reduce the rows returned to only those required.

To use the **Fill** method to return a page of data, specify a **startRecord** parameter, for the first record in the page of data, and a **maxRecords** parameter, for the number of records in the page of data.

The following code example shows how to use the **Fill** method to return the first page of a query result where the page size is five records.

```
Dim currentIndex As Integer = 0
Dim pageSize As Integer = 5

Dim orderSQL As String = "SELECT * FROM dbo.Orders ORDER BY OrderID"
' Assumes that connection is a valid SqlConnection object.
Dim adapter As SqlDataAdapter = _
  New SqlDataAdapter(orderSQL, connection)

Dim dataSet As DataSet = New DataSet()
adapter.Fill(dataSet, currentIndex, pageSize, "Orders")
```

```
int currentIndex = 0;
int pageSize = 5;

string orderSQL = "SELECT * FROM Orders ORDER BY OrderID";
// Assumes that connection is a valid SqlConnection object.
SqlDataAdapter adapter = new SqlDataAdapter(orderSQL, connection);

DataSet dataSet = new DataSet();
adapter.Fill(dataSet, currentIndex, pageSize, "Orders");
```

In the previous example, the **DataSet** is only filled with five records, but the entire **Orders** table is returned. To fill the **DataSet** with those same five records, but only return five records, use the TOP and WHERE clauses in your SQL statement, as in the following code example.

```
Dim pageSize As Integer = 5

Dim orderSQL As String = "SELECT TOP " & pageSize & _
  " * FROM Orders ORDER BY OrderID"
Dim adapter As SqlDataAdapter = _
  New SqlDataAdapter(orderSQL, connection)

Dim dataSet As DataSet = New DataSet()
adapter.Fill(dataSet, "Orders")
```

```
int pageSize = 5;

string orderSQL = "SELECT TOP " + pageSize +
  " * FROM Orders ORDER BY OrderID";
SqlDataAdapter adapter = new SqlDataAdapter(orderSQL, connection);

DataSet dataSet = new DataSet();
adapter.Fill(dataSet, "Orders");
```

Note that, when paging through the query results in this way, you must preserve the unique identifier that orders the rows, in order to pass the unique ID to the command to return the next page of records, as shown in the following code example.

```
Dim lastRecord As String = _
  dataSet.Tables("Orders").Rows(pageSize - 1)("OrderID").ToString()
```

```
string lastRecord =
  dataSet.Tables["Orders"].Rows[pageSize - 1]["OrderID"].ToString();
```

To return the next page of records using the overload of the **Fill** method that takes the **startRecord** and **maxRecords** parameters, increment the current record index by the page size and fill the table. Remember that the database server returns the entire query results even though only one page of records is added to the **DataSet**. In the following code example, the table rows are cleared before they are filled with the next page of data. You might want to preserve a certain number of returned rows in a local cache to reduce trips to the database server.

```
currentIndex = currentIndex + pageSize

dataSet.Tables("Orders").Rows.Clear()

adapter.Fill(dataSet, currentIndex, pageSize, "Orders")
```

```
currentIndex += pageSize;

dataSet.Tables["Orders"].Rows.Clear();

adapter.Fill(dataSet, currentIndex, pageSize, "Orders");
```

To return the next page of records without having the database server return the entire query, specify restrictive criteria to the SELECT statement. Because the preceding example preserved the last record returned, you can use it in the WHERE clause to specify a starting point for the query, as shown in the following code example.

```
orderSQL = "SELECT TOP " & pageSize & _
  " * FROM Orders WHERE OrderID > " & lastRecord & " ORDER BY OrderID"
adapter.SelectCommand.CommandText = orderSQL

dataSet.Tables("Orders").Rows.Clear()

adapter.Fill(dataSet, "Orders")
```

```
orderSQL = "SELECT TOP " + pageSize +
  " * FROM Orders WHERE OrderID > " + lastRecord + " ORDER BY OrderID";
adapter.SelectCommand.CommandText = orderSQL;

dataSet.Tables["Orders"].Rows.Clear();

adapter.Fill(dataSet, "Orders");
```

See also

- DataAdapters and DataReaders
- ADO.NET Overview

Updating Data Sources with DataAdapters

10/2/2019 • 13 minutes to read • Edit Online

The `Update` method of the DataAdapter is called to resolve changes from a DataSet back to the data source. The `Update` method, like the `Fill` method, takes as arguments an instance of a `DataSet`, and an optional DataTable object or `DataTable` name. The `DataSet` instance is the `DataSet` that contains the changes that have been made, and the `DataTable` identifies the table from which to retrieve the changes. If no `DataTable` is specified, the first `DataTable` in the `DataSet` is used.

When you call the `Update` method, the `DataAdapter` analyzes the changes that have been made and executes the appropriate command (INSERT, UPDATE, or DELETE). When the `DataAdapter` encounters a change to a DataRow, it uses the InsertCommand, UpdateCommand, or DeleteCommand to process the change. This allows you to maximize the performance of your ADO.NET application by specifying command syntax at design time and, where possible, through the use of stored procedures. You must explicitly set the commands before calling `Update`. If `Update` is called and the appropriate command does not exist for a particular update (for example, no `DeleteCommand` for deleted rows), an exception is thrown.

> **NOTE**
> If you are using SQL Server stored procedures to edit or delete data using a `DataAdapter`, make sure that you do not use SET NOCOUNT ON in the stored procedure definition. This causes the rows affected count returned to be zero, which the `DataAdapter` interprets as a concurrency conflict. In this event, a DBConcurrencyException will be thrown.

Command parameters can be used to specify input and output values for an SQL statement or stored procedure for each modified row in a `DataSet`. For more information, see DataAdapter Parameters.

> **NOTE**
> It is important to understand the difference between deleting a row in a DataTable and removing the row. When you call the `Remove` or `RemoveAt` method, the row is removed immediately. Any corresponding rows in the back end data source will not be affected if you then pass the `DataTable` or `DataSet` to a `DataAdapter` and call `Update`. When you use the `Delete` method, the row remains in the `DataTable` and is marked for deletion. If you then pass the `DataTable` or `DataSet` to a `DataAdapter` and call `Update`, the corresponding row in the back end data source is deleted.

If your `DataTable` maps to or is generated from a single database table, you can take advantage of the DbCommandBuilder object to automatically generate the `DeleteCommand`, `InsertCommand`, and `UpdateCommand` objects for the `DataAdapter`. For more information, see Generating Commands with CommandBuilders.

Using UpdatedRowSource to Map Values to a DataSet

You can control how the values returned from the data source are mapped back to the `DataTable` following a call to the Update method of a `DataAdapter`, by using the UpdatedRowSource property of a DbCommand object. By setting the `UpdatedRowSource` property to one of the UpdateRowSource enumeration values, you can control whether output parameters returned by the `DataAdapter` commands are ignored or applied to the changed row in the `DataSet`. You can also specify whether the first returned row (if it exists) is applied to the changed row in the `DataTable`.

The following table describes the different values of the `UpdateRowSource` enumeration and how they affect the

behavior of a command used with a `DataAdapter`.

UPDATEDROWSOURCE ENUMERATION	DESCRIPTION
Both	Both the output parameters and the first row of a returned result set may be mapped to the changed row in the `DataSet`.
FirstReturnedRecord	Only the data in the first row of a returned result set may be mapped to the changed row in the `DataSet`.
None	Any output parameters or rows of a returned result set are ignored.
OutputParameters	Only output parameters may be mapped to the changed row in the `DataSet`.

The `Update` method resolves your changes back to the data source; however other clients may have modified data at the data source since the last time you filled the `DataSet`. To refresh your `DataSet` with current data, use the `DataAdapter` and `Fill` method. New rows will be added to the table, and updated information will be incorporated into existing rows. The `Fill` method determines whether a new row will be added or an existing row will be updated by examining the primary key values of the rows in the `DataSet` and the rows returned by the `SelectCommand`. If the `Fill` method encounters a primary key value for a row in the `DataSet` that matches a primary key value from a row in the results returned by the `SelectCommand`, it updates the existing row with the information from the row returned by the `SelectCommand` and sets the RowState of the existing row to `Unchanged`. If a row returned by the `SelectCommand` has a primary key value that does not match any of the primary key values of the rows in the `DataSet`, the `Fill` method adds a new row with a `RowState` of `Unchanged`.

> **NOTE**
> If the `SelectCommand` returns the results of an OUTER JOIN, the `DataAdapter` will not set a `PrimaryKey` value for the resulting `DataTable`. You must define the `PrimaryKey` yourself to ensure that duplicate rows are resolved correctly. For more information, see Defining Primary Keys.

To handle exceptions that may occur when calling the `Update` method, you can use the `RowUpdated` event to respond to row update errors as they occur (see Handling DataAdapter Events), or you can set `DataAdapter.ContinueUpdateOnError` to `true` before calling `Update`, and respond to the error information stored in the `RowError` property of a particular row when the update is complete (see Row Error Information).

> **NOTE**
> Calling `AcceptChanges` on the `DataSet`, `DataTable`, or `DataRow` will cause all `Original` values for a `DataRow` to be overwritten with the `Current` values for the `DataRow`. If the field values that identify the row as unique have been modified, after calling `AcceptChanges` the `Original` values will no longer match the values in the data source. `AcceptChanges` is called automatically for each row during a call to the Update method of a `DataAdapter`. You can preserve the original values during a call to the Update method by first setting the `AcceptChangesDuringUpdate` property of the `DataAdapter` to false, or by creating an event handler for the `RowUpdated` event and setting the Status to SkipCurrentRow. For more information, see Merging DataSet Contents and Handling DataAdapter Events.

Example

The following examples demonstrate how to perform updates to modified rows by explicitly setting the

`UpdateCommand` of a `DataAdapter` and calling its `Update` method. Notice that the parameter specified in the WHERE clause of the UPDATE statement is set to use the `Original` value of the `SourceColumn`. This is important, because the `Current` value may have been modified and may not match the value in the data source. The `Original` value is the value that was used to populate the `DataTable` from the data source.

```
private static void AdapterUpdate(string connectionString)
{
    using (SqlConnection connection =
                new SqlConnection(connectionString))
    {
        SqlDataAdapter dataAdpater = new SqlDataAdapter(
          "SELECT CategoryID, CategoryName FROM Categories",
          connection);

        dataAdpater.UpdateCommand = new SqlCommand(
           "UPDATE Categories SET CategoryName = @CategoryName " +
           "WHERE CategoryID = @CategoryID", connection);

        dataAdpater.UpdateCommand.Parameters.Add(
           "@CategoryName", SqlDbType.NVarChar, 15, "CategoryName");

        SqlParameter parameter = dataAdpater.UpdateCommand.Parameters.Add(
          "@CategoryID", SqlDbType.Int);
        parameter.SourceColumn = "CategoryID";
        parameter.SourceVersion = DataRowVersion.Original;

        DataTable categoryTable = new DataTable();
        dataAdpater.Fill(categoryTable);

        DataRow categoryRow = categoryTable.Rows[0];
        categoryRow["CategoryName"] = "New Beverages";

        dataAdpater.Update(categoryTable);

        Console.WriteLine("Rows after update.");
        foreach (DataRow row in categoryTable.Rows)
        {
            {
                Console.WriteLine("{0}: {1}", row[0], row[1]);
            }
        }
    }
}
```

```
Private Sub AdapterUpdate(ByVal connectionString As String)

    Using connection As SqlConnection = New SqlConnection( _
       connectionString)

        Dim adapter As SqlDataAdapter = New SqlDataAdapter( _
          "SELECT CategoryID, CategoryName FROM dbo.Categories", _
          connection)

        adapter.UpdateCommand = New SqlCommand( _
          "UPDATE Categories SET CategoryName = @CategoryName " & _
           "WHERE CategoryID = @CategoryID", connection)

        adapter.UpdateCommand.Parameters.Add( _
           "@CategoryName", SqlDbType.NVarChar, 15, "CategoryName")

        Dim parameter As SqlParameter = _
           adapter.UpdateCommand.Parameters.Add( _
           "@CategoryID", SqlDbType.Int)
        parameter.SourceColumn = "CategoryID"
        parameter.SourceVersion = DataRowVersion.Original

        Dim categoryTable As New DataTable
        adapter.Fill(categoryTable)

        Dim categoryRow As DataRow = categoryTable.Rows(0)
        categoryRow("CategoryName") = "New Beverages"

        adapter.Update(categoryTable)

        Console.WriteLine("Rows after update.")
        Dim row As DataRow
        For Each row In categoryTable.Rows
            Console.WriteLine("{0}: {1}", row(0), row(1))
        Next
    End Using
End Sub
```

AutoIncrement Columns

If the tables from your data source have auto-incrementing columns, you can fill the columns in your `DataSet` either by returning the auto-increment value as an output parameter of a stored procedure and mapping that to a column in a table, by returning the auto-increment value in the first row of a result set returned by a stored procedure or SQL statement, or by using the `RowUpdated` event of the `DataAdapter` to execute an additional SELECT statement. For more information and an example, see Retrieving Identity or Autonumber Values.

Ordering of Inserts, Updates, and Deletes

In many circumstances, the order in which changes made through the `DataSet` are sent to the data source is important. For example, if a primary key value for an existing row is updated, and a new row has been added with the new primary key value as a foreign key, it is important to process the update before the insert.

You can use the `Select` method of the `DataTable` to return a `DataRow` array that only references rows with a particular `RowState`. You can then pass the returned `DataRow` array to the `Update` method of the `DataAdapter` to process the modified rows. By specifying a subset of rows to be updated, you can control the order in which inserts, updates, and deletes are processed.

Example

For example, the following code ensures that the deleted rows of the table are processed first, then the updated

rows, and then the inserted rows.

```
Dim table As DataTable = dataSet.Tables("Customers")

' First process deletes.
dataSet.Update(table.Select(Nothing, Nothing, _
  DataViewRowState.Deleted))

' Next process updates.
adapter.Update(table.Select(Nothing, Nothing, _
  DataViewRowState.ModifiedCurrent))

' Finally, process inserts.
adapter.Update(table.Select(Nothing, Nothing, _
  DataViewRowState.Added))
```

```
DataTable table = dataSet.Tables["Customers"];

// First process deletes.
adapter.Update(table.Select(null, null, DataViewRowState.Deleted));

// Next process updates.
adapter.Update(table.Select(null, null,
  DataViewRowState.ModifiedCurrent));

// Finally, process inserts.
adapter.Update(table.Select(null, null, DataViewRowState.Added));
```

Use a DataAdapter to Retrieve and Update Data

You can use a DataAdapter to retrieve and update the data.

- The sample uses DataAdapter.AcceptChangesDuringFill to clone the data in the database. If the property is set as false, AcceptChanges is not called when filling the table, and the newly added rows are treated as inserted rows. So, the sample uses these rows to insert the new rows into the database.
- The samples uses DataAdapter.TableMappings to define the mapping between the source table and DataTable.
- The sample uses DataAdapter.FillLoadOption to determine how the adapter fills the DataTable from the DbDataReader. When you create a DataTable, you can only write the data from database to the current version or the original version by setting the property as the LoadOption.Upsert or the LoadOption.PreserveChanges.
- The sample will also update the table by using DbDataAdapter.UpdateBatchSize to perform batch operations.

Before you compile and run the sample, you need to create the sample database:

```
USE [master]
GO

CREATE DATABASE [MySchool]

GO

USE [MySchool]
GO

SET ANSI_NULLS ON
GO
```

```
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[Course]([CourseID] [nvarchar](10) NOT NULL,
[Year] [smallint] NOT NULL,
[Title] [nvarchar](100) NOT NULL,
[Credits] [int] NOT NULL,
[DepartmentID] [int] NOT NULL,
 CONSTRAINT [PK_Course] PRIMARY KEY CLUSTERED
(
[CourseID] ASC,
[Year] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS = ON,
ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]) ON [PRIMARY]

GO

SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[Department]([DepartmentID] [int] IDENTITY(1,1) NOT NULL,
[Name] [nvarchar](50) NOT NULL,
[Budget] [money] NOT NULL,
[StartDate] [datetime] NOT NULL,
[Administrator] [int] NULL,
 CONSTRAINT [PK_Department] PRIMARY KEY CLUSTERED
(
[DepartmentID] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS = ON,
ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]) ON [PRIMARY]

GO

INSERT [dbo].[Course] ([CourseID], [Year], [Title], [Credits], [DepartmentID]) VALUES (N'C1045', 2012,
N'Calculus', 4, 7)
INSERT [dbo].[Course] ([CourseID], [Year], [Title], [Credits], [DepartmentID]) VALUES (N'C1061', 2012,
N'Physics', 4, 1)
INSERT [dbo].[Course] ([CourseID], [Year], [Title], [Credits], [DepartmentID]) VALUES (N'C2021', 2012,
N'Composition', 3, 2)
INSERT [dbo].[Course] ([CourseID], [Year], [Title], [Credits], [DepartmentID]) VALUES (N'C2042', 2012,
N'Literature', 4, 2)

SET IDENTITY_INSERT [dbo].[Department] ON

INSERT [dbo].[Department] ([DepartmentID], [Name], [Budget], [StartDate], [Administrator]) VALUES (1,
N'Engineering', 350000.0000, CAST(0x0000999C00000000 AS DateTime), 2)
INSERT [dbo].[Department] ([DepartmentID], [Name], [Budget], [StartDate], [Administrator]) VALUES (2,
N'English', 120000.0000, CAST(0x0000999C00000000 AS DateTime), 6)
INSERT [dbo].[Department] ([DepartmentID], [Name], [Budget], [StartDate], [Administrator]) VALUES (4,
N'Economics', 200000.0000, CAST(0x0000999C00000000 AS DateTime), 4)
INSERT [dbo].[Department] ([DepartmentID], [Name], [Budget], [StartDate], [Administrator]) VALUES (7,
N'Mathematics', 250024.0000, CAST(0x0000999C00000000 AS DateTime), 3)
SET IDENTITY_INSERT [dbo].[Department] OFF

ALTER TABLE [dbo].[Course]  WITH CHECK ADD  CONSTRAINT [FK_Course_Department] FOREIGN KEY([DepartmentID])
REFERENCES [dbo].[Department] ([DepartmentID])
GO
ALTER TABLE [dbo].[Course] CHECK CONSTRAINT [FK_Course_Department]
GO
```

C# and Visual Basic projects with this code sample can be found on Developer Code Samples.

```
using System;
using System.Data;
using System.Data.Common;
using System.Data.SqlClient;
using System.Linq;
```

```
using CSDataAdapterOperations.Properties;

namespace CSDataAdapterOperations.Properties {
   internal sealed partial class Settings : global::System.Configuration.ApplicationSettingsBase {

      private static Settings defaultInstance = ((Settings)
(global::System.Configuration.ApplicationSettingsBase.Synchronized(new Settings())));

      public static Settings Default {
         get {
            return defaultInstance;
         }
      }

      [global::System.Configuration.ApplicationScopedSettingAttribute()]
      [global::System.Configuration.DefaultSettingValueAttribute("Data Source=(local);Initial
Catalog=MySchool;Integrated Security=True")]
      public string MySchoolConnectionString {
         get {
            return ((string)(this["MySchoolConnectionString"]));
         }
      }
   }
}

class Program {
   static void Main(string[] args) {
      Settings settings = new Settings();

      // Copy the data from the database.  Get the table Department and Course from the database.
      String selectString = @"SELECT [DepartmentID],[Name],[Budget],[StartDate],[Administrator]
                                    FROM [MySchool].[dbo].[Department];

                                  SELECT [CourseID],@Year as [Year],Max([Title]) as [Title],
                                  Max([Credits]) as [Credits],Max([DepartmentID]) as [DepartmentID]
                                  FROM [MySchool].[dbo].[Course]
                                  Group by [CourseID]";

      DataSet mySchool = new DataSet();

      SqlCommand selectCommand = new SqlCommand(selectString);
      SqlParameter parameter = selectCommand.Parameters.Add("@Year", SqlDbType.SmallInt, 2);
      parameter.Value = new Random(DateTime.Now.Millisecond).Next(9999);

      // Use DataTableMapping to map the source tables and the destination tables.
      DataTableMapping[] tableMappings = {new DataTableMapping("Table", "Department"), new
DataTableMapping("Table1", "Course")};
      CopyData(mySchool, settings.MySchoolConnectionString, selectCommand, tableMappings);

      Console.WriteLine("The following tables are from the database.");
      foreach (DataTable table in mySchool.Tables) {
         Console.WriteLine(table.TableName);
         ShowDataTable(table);
      }

      // Roll back the changes
      DataTable department = mySchool.Tables["Department"];
      DataTable course = mySchool.Tables["Course"];

      department.Rows[0]["Name"] = "New" + department.Rows[0][1];
      course.Rows[0]["Title"] = "New" + course.Rows[0]["Title"];
      course.Rows[0]["Credits"] = 10;

      Console.WriteLine("After we changed the tables:");
      foreach (DataTable table in mySchool.Tables) {
         Console.WriteLine(table.TableName);
         ShowDataTable(table);
      }
```

```
      department.RejectChanges();
      Console.WriteLine("After use the RejectChanges method in Department table to roll back the changes:");
      ShowDataTable(department);

      DataColumn[] primaryColumns = { course.Columns["CourseID"] };
      DataColumn[] resetColumns = { course.Columns["Title"] };
      ResetCourse(course, settings.MySchoolConnectionString, primaryColumns, resetColumns);
      Console.WriteLine("After use the ResetCourse method in Course table to roll back the changes:");
      ShowDataTable(course);

      // Batch update the table.
      String insertString = @"Insert into [MySchool].[dbo].[Course]([CourseID],[Year],[Title],
                                [Credits],[DepartmentID])
            values (@CourseID,@Year,@Title,@Credits,@DepartmentID)";
      SqlCommand insertCommand = new SqlCommand(insertString);
      insertCommand.Parameters.Add("@CourseID", SqlDbType.NVarChar, 10, "CourseID");
      insertCommand.Parameters.Add("@Year", SqlDbType.SmallInt, 2, "Year");
      insertCommand.Parameters.Add("@Title", SqlDbType.NVarChar, 100, "Title");
      insertCommand.Parameters.Add("@Credits", SqlDbType.Int, 4, "Credits");
      insertCommand.Parameters.Add("@DepartmentID", SqlDbType.Int, 4, "DepartmentID");

      const Int32 batchSize = 10;
      BatchInsertUpdate(course, settings.MySchoolConnectionString, insertCommand, batchSize);
   }

   private static void CopyData(DataSet dataSet, String connectionString, SqlCommand selectCommand,
DataTableMapping[] tableMappings) {
      using (SqlConnection connection = new SqlConnection(connectionString)) {
         selectCommand.Connection = connection;

         connection.Open();

         using (SqlDataAdapter adapter = new SqlDataAdapter(selectCommand))
{adapter.TableMappings.AddRange(tableMappings);
            // If set the AcceptChangesDuringFill as the false, AcceptChanges will not be called on a
            // DataRow after it is added to the DataTable during any of the Fill operations.
            adapter.AcceptChangesDuringFill = false;

            adapter.Fill(dataSet);
         }
      }
   }

   // Roll back only one column or several columns data of the Course table by call ResetDataTable method.
   private static void ResetCourse(DataTable table, String connectionString,
       DataColumn[] primaryColumns, DataColumn[] resetColumns) {
      table.PrimaryKey = primaryColumns;

      // Build the query string
      String primaryCols = String.Join(",", primaryColumns.Select(col => col.ColumnName));
      String resetCols = String.Join(",", resetColumns.Select(col => $"Max({col.ColumnName}) as
{col.ColumnName}"));

      String selectString = $"Select {primaryCols},{resetCols} from Course Group by {primaryCols}";

      SqlCommand selectCommand = new SqlCommand(selectString);

      ResetDataTable(table, connectionString, selectCommand);
   }

   // RejectChanges will roll back all changes made to the table since it was loaded, or the last time
AcceptChanges
   // was called. When you copy from the database, you can lose all the data after calling RejectChanges
   // The ResetDataTable method rolls back one or more columns of data.
   private static void ResetDataTable(DataTable table, String connectionString,
       SqlCommand selectCommand) {
      using (SqlConnection connection = new SqlConnection(connectionString)) {
         selectCommand.Connection = connection;
```

```
            connection.Open();

            using (SqlDataAdapter adapter = new SqlDataAdapter(selectCommand)) {
               // The incoming values for this row will be written to the current version of each
               // column. The original version of each column's data will not be changed.
               adapter.FillLoadOption = LoadOption.Upsert;

               adapter.Fill(table);
            }
         }
      }

      private static void BatchInsertUpdate(DataTable table, String connectionString,
          SqlCommand insertCommand, Int32 batchSize) {
         using (SqlConnection connection = new SqlConnection(connectionString)) {
            insertCommand.Connection = connection;
            // When setting UpdateBatchSize to a value other than 1, all the commands
            // associated with the SqlDataAdapter have to have their UpdatedRowSource
            // property set to None or OutputParameters. An exception is thrown otherwise.
            insertCommand.UpdatedRowSource = UpdateRowSource.None;

            connection.Open();

            using (SqlDataAdapter adapter = new SqlDataAdapter()) {
               adapter.InsertCommand = insertCommand;
               // Gets or sets the number of rows that are processed in each round-trip to the server.
               // Setting it to 1 disables batch updates, as rows are sent one at a time.
               adapter.UpdateBatchSize = batchSize;

               adapter.Update(table);

               Console.WriteLine("Successfully to update the table.");
            }
         }
      }

      private static void ShowDataTable(DataTable table) {
         foreach (DataColumn col in table.Columns) {
            Console.Write("{0,-14}", col.ColumnName);
         }
         Console.WriteLine("{0,-14}", "RowState");

         foreach (DataRow row in table.Rows) {
            foreach (DataColumn col in table.Columns) {
               if (col.DataType.Equals(typeof(DateTime)))
                  Console.Write("{0,-14:d}", row[col]);
               else if (col.DataType.Equals(typeof(Decimal)))
                  Console.Write("{0,-14:C}", row[col]);
               else
                  Console.Write("{0,-14}", row[col]);
            }
            Console.WriteLine("{0,-14}", row.RowState);
         }
      }
   }
```

See also

- DataAdapters and DataReaders
- Row States and Row Versions
- AcceptChanges and RejectChanges
- Merging DataSet Contents
- Retrieving Identity or Autonumber Values

- ADO.NET Overview

Handling DataAdapter Events

3/12/2020 • 6 minutes to read • Edit Online

The ADO.NET DataAdapter exposes three events that you can use to respond to changes made to data at the data source. The following table shows the `DataAdapter` events.

EVENT	DESCRIPTION
`RowUpdating`	An UPDATE, INSERT, or DELETE operation on a row (by a call to one of the `Update` methods) is about to begin.
`RowUpdated`	An UPDATE, INSERT, or DELETE operation on a row (by a call to one of the `Update` methods) is complete.
`FillError`	An error has occurred during a `Fill` operation.

RowUpdating and RowUpdated

`RowUpdating` is raised before any update to a row from the DataSet has been processed at the data source. `RowUpdated` is raised after any update to a row from the `DataSet` has been processed at the data source. As a result, you can use `RowUpdating` to modify update behavior before it happens, to provide additional handling when an update will occur, to retain a reference to an updated row, to cancel the current update and schedule it for a batch process to be processed later, and so on. `RowUpdated` is useful for responding to errors and exceptions that occur during the update. You can add error information to the `DataSet`, as well as retry logic, and so on.

The RowUpdatingEventArgs and RowUpdatedEventArgs arguments passed to the `RowUpdating` and `RowUpdated` events include the following: a `Command` property that references the `Command` object being used to perform the update; a `Row` property that references the `DataRow` object containing the updated information; a `StatementType` property for what type of update is being performed; the `TableMapping`, if applicable; and the `Status` of the operation.

You can use the `Status` property to determine if an error has occurred during the operation and, if desired, to control the actions against the current and resulting rows. When the event occurs, the `Status` property equals either `Continue` or `ErrorsOccurred`. The following table shows the values to which you can set the `Status` property in order to control later actions during the update.

STATUS	DESCRIPTION
`Continue`	Continue the update operation.
`ErrorsOccurred`	Abort the update operation and throw an exception.
`SkipCurrentRow`	Ignore the current row and continue the update operation.
`SkipAllRemainingRows`	Abort the update operation but do not throw an exception.

Setting the `Status` property to `ErrorsOccurred` causes an exception to be thrown. You can control which exception is thrown by setting the `Errors` property to the desired exception. Using one of the other values for `Status` prevents an exception from being thrown.

You can also use the `ContinueUpdateOnError` property to handle errors for updated rows. If `DataAdapter.ContinueUpdateOnError` is `true`, when an update to a row results in an exception being thrown, the text of the exception is placed into the `RowError` information of the particular row, and processing continues without throwing an exception. This enables you to respond to errors when the `Update` is complete, in contrast to the `RowUpdated` event, which enables you to respond to errors when the error is encountered.

The following code sample shows how to both add and remove event handlers. The `RowUpdating` event handler writes a log of all deleted records with a time stamp. The `RowUpdated` event handler adds error information to the `RowError` property of the row in the `DataSet`, suppresses the exception, and continues processing (mirroring the behavior of `ContinueUpdateOnError` = `true`).

```
' Assumes that connection is a valid SqlConnection object.
Dim custAdapter As SqlDataAdapter = New SqlDataAdapter( _
  "SELECT CustomerID, CompanyName FROM Customers", connection)

' Add handlers.
AddHandler custAdapter.RowUpdating, New SqlRowUpdatingEventHandler( _
  AddressOf OnRowUpdating)
AddHandler custAdapter.RowUpdated, New SqlRowUpdatedEventHandler(
  AddressOf OnRowUpdated)

' Set DataAdapter command properties, fill DataSet, and modify DataSet.

custAdapter.Update(custDS, "Customers")

' Remove handlers.
RemoveHandler custAdapter.RowUpdating, _
  New SqlRowUpdatingEventHandler(AddressOf OnRowUpdating)
RemoveHandler custAdapter.RowUpdated, _
  New SqlRowUpdatedEventHandler(AddressOf OnRowUpdated)

Private Shared Sub OnRowUpdating(sender As Object, _
  args As SqlRowUpdatingEventArgs)
  If args.StatementType = StatementType.Delete Then
    Dim tw As System.IO.TextWriter = _
  System.IO.File.AppendText("Deletes.log")
    tw.WriteLine( _
      "{0}: Customer {1} Deleted.", DateTime.Now, args.Row(_
      "CustomerID", DataRowVersion.Original))
    tw.Close()
  End If
End Sub

Private Shared Sub OnRowUpdated( _
  sender As Object, args As SqlRowUpdatedEventArgs)
  If args.Status = UpdateStatus.ErrorsOccurred
    args.Status = UpdateStatus.SkipCurrentRow
    args.Row.RowError = args.Errors.Message
  End If
End Sub
```

```
// Assumes that connection is a valid SqlConnection object.
SqlDataAdapter custAdapter = new SqlDataAdapter(
  "SELECT CustomerID, CompanyName FROM Customers", connection);

// Add handlers.
custAdapter.RowUpdating += new SqlRowUpdatingEventHandler(OnRowUpdating);
custAdapter.RowUpdated += new SqlRowUpdatedEventHandler(OnRowUpdated);

// Set DataAdapter command properties, fill DataSet, modify DataSet.

custAdapter.Update(custDS, "Customers");

// Remove handlers.
custAdapter.RowUpdating -= new SqlRowUpdatingEventHandler(OnRowUpdating);
custAdapter.RowUpdated -= new SqlRowUpdatedEventHandler(OnRowUpdated);

protected static void OnRowUpdating(
  object sender, SqlRowUpdatingEventArgs args)
{
  if (args.StatementType == StatementType.Delete)
  {
    System.IO.TextWriter tw = System.IO.File.AppendText("Deletes.log");
    tw.WriteLine(
      "{0}: Customer {1} Deleted.", DateTime.Now,
       args.Row["CustomerID", DataRowVersion.Original]);
    tw.Close();
  }
}

protected static void OnRowUpdated(
  object sender, SqlRowUpdatedEventArgs args)
{
  if (args.Status == UpdateStatus.ErrorsOccurred)
  {
    args.Row.RowError = args.Errors.Message;
    args.Status = UpdateStatus.SkipCurrentRow;
  }
}
```

FillError

The `DataAdapter` issues the `FillError` event when an error occurs during a `Fill` operation. This type of error commonly occurs when the data in the row being added could not be converted to a .NET Framework type without some loss of precision.

If an error occurs during a `Fill` operation, the current row is not added to the `DataTable`. The `FillError` event enables you to resolve the error and add the row, or to ignore the excluded row and continue the `Fill` operation.

The `FillErrorEventArgs` passed to the `FillError` event can contain several properties that enable you to respond to and resolve errors. The following table shows the properties of the `FillErrorEventArgs` object.

PROPERTY	DESCRIPTION
`Errors`	The `Exception` that occurred.
`DataTable`	The `DataTable` object being filled when the error occurred.

PROPERTY	DESCRIPTION
`Values`	An array of objects that contains the values of the row being added when the error occurred. The ordinal references of the `Values` array correspond to the ordinal references of the columns of the row being added. For example, `Values[0]` is the value that was being added as the first column of the row.
`Continue`	Allows you to choose whether or not to throw an exception. Setting the `Continue` property to `false` will halt the current `Fill` operation, and an exception will be thrown. Setting `Continue` to `true` continues the `Fill` operation despite the error.

The following code example adds an event handler for the `FillError` event of the `DataAdapter`. In the `FillError` event code, the example determines if there is the potential for precision loss, providing the opportunity to respond to the exception.

```
AddHandler adapter.FillError, New FillErrorEventHandler( _
  AddressOf FillError)

Dim dataSet As DataSet = New DataSet
adapter.Fill(dataSet, "ThisTable")

Private Shared Sub FillError(sender As Object, _
  args As FillErrorEventArgs)
  If args.Errors.GetType() Is Type.GetType("System.OverflowException") Then
    ' Code to handle precision loss.
    ' Add a row to table using the values from the first two columns.
    DataRow myRow = args.DataTable.Rows.Add(New Object() _
      {args.Values(0), args.Values(1), DBNull.Value})
    ' Set the RowError containing the value for the third column.
    myRow.RowError = _
      "OverflowException encountered. Value from data source: " & _
      args.Values(2)
    args.Continue = True
  End If
End Sub
```

```
adapter.FillError += new FillErrorEventHandler(FillError);

DataSet dataSet = new DataSet();
adapter.Fill(dataSet, "ThisTable");

protected static void FillError(object sender, FillErrorEventArgs args)
{
  if (args.Errors.GetType() == typeof(System.OverflowException))
  {
    // Code to handle precision loss.
    //Add a row to table using the values from the first two columns.
    DataRow myRow = args.DataTable.Rows.Add(new object[]
       {args.Values[0], args.Values[1], DBNull.Value});
    //Set the RowError containing the value for the third column.
    myRow.RowError =
       "OverflowException Encountered. Value from data source: " +
       args.Values[2];
    args.Continue = true;
  }
}
```

See also

- DataAdapters and DataReaders
- Handling DataSet Events
- Handling DataTable Events
- Events
- ADO.NET Overview

Performing Batch Operations Using DataAdapters

3/12/2020 • 5 minutes to read • Edit Online

Batch support in ADO.NET allows a DataAdapter to group INSERT, UPDATE, and DELETE operations from a DataSet or DataTable to the server, instead of sending one operation at a time. The reduction in the number of round trips to the server typically results in significant performance gains. Batch updates are supported for the .NET data providers for SQL Server (System.Data.SqlClient) and Oracle (System.Data.OracleClient).

When updating a database with changes from a DataSet in previous versions of ADO.NET, the `Update` method of a `DataAdapter` performed updates to the database one row at a time. As it iterated through the rows in the specified DataTable, it examined each DataRow to see if it had been modified. If the row had been modified, it called the appropriate `UpdateCommand`, `InsertCommand`, or `DeleteCommand`, depending on the value of the RowState property for that row. Every row update involved a network round-trip to the database.

Starting with ADO.NET 2.0, the DbDataAdapter exposes an UpdateBatchSize property. Setting the `UpdateBatchSize` to a positive integer value causes updates to the database to be sent as batches of the specified size. For example, setting the `UpdateBatchSize` to 10 will group 10 separate statements and submit them as single batch. Setting the `UpdateBatchSize` to 0 will cause the DataAdapter to use the largest batch size that the server can handle. Setting it to 1 disables batch updates, as rows are sent one at a time.

Executing an extremely large batch could decrease performance. Therefore, you should test for the optimum batch size setting before implementing your application.

Using the UpdateBatchSize Property

When batch updates are enabled, the UpdatedRowSource property value of the DataAdapter's `UpdateCommand`, `InsertCommand`, and `DeleteCommand` should be set to None or OutputParameters. When performing a batch update, the command's UpdatedRowSource property value of FirstReturnedRecord or Both is invalid.

The following procedure demonstrates the use of the `UpdateBatchSize` property. The procedure takes two arguments, a DataSet object that has columns representing the **ProductCategoryID** and **Name** fields in the **Production.ProductCategory** table, and an integer representing the batch size (the number of rows in the batch). The code creates a new SqlDataAdapter object, setting its UpdateCommand, InsertCommand, and DeleteCommand properties. The code assumes that the DataSet object has modified rows. It sets the `UpdateBatchSize` property and executes the update.

```
Public Sub BatchUpdate( _
  ByVal dataTable As DataTable, ByVal batchSize As Int32)
    ' Assumes GetConnectionString() returns a valid connection string.
    Dim connectionString As String = GetConnectionString()

    ' Connect to the AdventureWorks database.
    Using connection As New SqlConnection(connectionString)
        ' Create a SqlDataAdapter.
        Dim adapter As New SqlDataAdapter()

        'Set the UPDATE command and parameters.
        adapter.UpdateCommand = New SqlCommand( _
          "UPDATE Production.ProductCategory SET " _
          & "Name=@Name WHERE ProductCategoryID=@ProdCatID;", _
          connection)
        adapter.UpdateCommand.Parameters.Add("@Name", _
          SqlDbType.NVarChar, 50, "Name")
        adapter.UpdateCommand.Parameters.Add("@ProdCatID",  _
          SqlDbType.Int, 4, " ProductCategoryID ")
        adapter.UpdateCommand.UpdatedRowSource = _
          UpdateRowSource.None

        'Set the INSERT command and parameter.
        adapter.InsertCommand = New SqlCommand( _
          "INSERT INTO Production.ProductCategory (Name) VALUES (@Name);", _
  connection)
        adapter.InsertCommand.Parameters.Add("@Name", _
          SqlDbType.NVarChar, 50, "Name")
        adapter.InsertCommand.UpdatedRowSource = _
          UpdateRowSource.None

        'Set the DELETE command and parameter.
        adapter.DeleteCommand = New SqlCommand( _
          "DELETE FROM Production.ProductCategory " _
          & "WHERE ProductCategoryID=@ProdCatID;", connection)
        adapter.DeleteCommand.Parameters.Add("@ProdCatID", _
           SqlDbType.Int, 4, " ProductCategoryID ")
        adapter.DeleteCommand.UpdatedRowSource = UpdateRowSource.None

        ' Set the batch size.
        adapter.UpdateBatchSize = batchSize

        ' Execute the update.
        adapter.Update(dataTable)
    End Using
End Sub
```

```
public static void BatchUpdate(DataTable dataTable,Int32 batchSize)
{
    // Assumes GetConnectionString() returns a valid connection string.
    string connectionString = GetConnectionString();

    // Connect to the AdventureWorks database.
    using (SqlConnection connection = new
      SqlConnection(connectionString))
    {

        // Create a SqlDataAdapter.
        SqlDataAdapter adapter = new SqlDataAdapter();

        // Set the UPDATE command and parameters.
        adapter.UpdateCommand = new SqlCommand(
            "UPDATE Production.ProductCategory SET "
            + "Name=@Name WHERE ProductCategoryID=@ProdCatID;",
            connection);
        adapter.UpdateCommand.Parameters.Add("@Name",
           SqlDbType.NVarChar, 50, "Name");
        adapter.UpdateCommand.Parameters.Add("@ProdCatID",
           SqlDbType.Int, 4, "ProductCategoryID");
         adapter.UpdateCommand.UpdatedRowSource = UpdateRowSource.None;

        // Set the INSERT command and parameter.
        adapter.InsertCommand = new SqlCommand(
            "INSERT INTO Production.ProductCategory (Name) VALUES (@Name);",
            connection);
        adapter.InsertCommand.Parameters.Add("@Name",
          SqlDbType.NVarChar, 50, "Name");
        adapter.InsertCommand.UpdatedRowSource = UpdateRowSource.None;

        // Set the DELETE command and parameter.
        adapter.DeleteCommand = new SqlCommand(
            "DELETE FROM Production.ProductCategory "
            + "WHERE ProductCategoryID=@ProdCatID;", connection);
        adapter.DeleteCommand.Parameters.Add("@ProdCatID",
          SqlDbType.Int, 4, "ProductCategoryID");
        adapter.DeleteCommand.UpdatedRowSource = UpdateRowSource.None;

        // Set the batch size.
        adapter.UpdateBatchSize = batchSize;

        // Execute the update.
        adapter.Update(dataTable);
    }
}
```

Handling Batch Update-Related Events and Errors

The **DataAdapter** has two update-related events: **RowUpdating** and **RowUpdated**. In previous versions of ADO.NET, when batch processing is disabled, each of these events is generated once for each row processed. **RowUpdating** is generated before the update occurs, and **RowUpdated** is generated after the database update has been completed.

Event Behavior Changes with Batch Updates

When batch processing is enabled, multiple rows are updated in a single database operation. Therefore, only one `RowUpdated` event occurs for each batch, whereas the `RowUpdating` event occurs for each row processed. When batch processing is disabled, the two events are fired with one-to-one interleaving, where one `RowUpdating` event and one `RowUpdated` event fire for a row, and then one `RowUpdating` and one `RowUpdated` event fire for the next row, until all of the rows are processed.

Accessing Updated Rows

When batch processing is disabled, the row being updated can be accessed using the Row property of the RowUpdatedEventArgs class.

When batch processing is enabled, a single `RowUpdated` event is generated for multiple rows. Therefore, the value of the `Row` property for each row is null. `RowUpdating` events are still generated for each row. The CopyToRows method of the RowUpdatedEventArgs class allows you to access the processed rows by copying references to the rows into an array. If no rows are being processed, `CopyToRows` throws an ArgumentNullException. Use the RowCount property to return the number of rows processed before calling the CopyToRows method.

Handling Data Errors

Batch execution has the same effect as the execution of each individual statement. Statements are executed in the order that the statements were added to the batch. Errors are handled the same way in batch mode as they are when batch mode is disabled. Each row is processed separately. Only rows that have been successfully processed in the database will be updated in the corresponding DataRow within the DataTable.

The data provider and the back-end database server determine which SQL constructs are supported for batch execution. An exception may be thrown if a non-supported statement is submitted for execution.

See also

- DataAdapters and DataReaders
- Updating Data Sources with DataAdapters
- Handling DataAdapter Events
- ADO.NET Overview

Transactions and Concurrency

9/7/2019 • 2 minutes to read • Edit Online

A transaction consists of a single command or a group of commands that execute as a package. Transactions allow you to combine multiple operations into a single unit of work. If a failure occurs at one point in the transaction, all of the updates can be rolled back to their pre-transaction state.

A transaction must conform to the ACID properties—atomicity, consistency, isolation, and durability—in order to guarantee data consistency. Most relational database systems, such as Microsoft SQL Server, support transactions by providing locking, logging, and transaction management facilities whenever a client application performs an update, insert, or delete operation.

> **NOTE**
> Transactions that involve multiple resources can lower concurrency if locks are held too long. Therefore, keep transactions as short as possible.

If a transaction involves multiple tables in the same database or server, then explicit transactions in stored procedures often perform better. You can create transactions in SQL Server stored procedures by using the Transact-SQL `BEGIN TRANSACTION`, `COMMIT TRANSACTION`, and `ROLLBACK TRANSACTION` statements. For more information, see SQL Server Books Online.

Transactions involving different resource managers, such as a transaction between SQL Server and Oracle, require a distributed transaction.

In This Section

Local Transactions
Demonstrates how to perform transactions against a database.

Distributed Transactions
Describes how to perform distributed transactions in ADO.NET.

System.Transactions Integration with SQL Server
Describes System.Transactions integration with SQL Server for working with distributed transactions.

Optimistic Concurrency
Describes optimistic and pessimistic concurrency, and how you can test for concurrency violations.

See also

- Transaction Fundamentals
- Connecting to a Data Source
- Commands and Parameters
- DataAdapters and DataReaders
- DbProviderFactories
- ADO.NET Overview

Local Transactions

9/7/2019 • 3 minutes to read • Edit Online

Transactions in ADO.NET are used when you want to bind multiple tasks together so that they execute as a single unit of work. For example, imagine that an application performs two tasks. First, it updates a table with order information. Second, it updates a table that contains inventory information, debiting the items ordered. If either task fails, then both updates are rolled back.

Determining the Transaction Type

A transaction is considered to be a local transaction when it is a single-phase transaction and is handled by the database directly. A transaction is considered to be a distributed transaction when it is coordinated by a transaction monitor and uses fail-safe mechanisms (such as two-phase commit) for transaction resolution.

Each of the .NET Framework data providers has its own `Transaction` object for performing local transactions. If you require a transaction to be performed in a SQL Server database, select a System.Data.SqlClient transaction. For an Oracle transaction, use the System.Data.OracleClient provider. In addition, there is a DbTransaction class that is available for writing provider-independent code that requires transactions.

> **NOTE**
>
> Transactions are most efficient when they are performed on the server. If you are working with a SQL Server database that makes extensive use of explicit transactions, consider writing them as stored procedures using the Transact-SQL BEGIN TRANSACTION statement.

Performing a Transaction Using a Single Connection

In ADO.NET, you control transactions with the `Connection` object. You can initiate a local transaction with the `BeginTransaction` method. Once you have begun a transaction, you can enlist a command in that transaction with the `Transaction` property of a `Command` object. You can then commit or roll back modifications made at the data source based on the success or failure of the components of the transaction.

> **NOTE**
>
> The `EnlistDistributedTransaction` method should not be used for a local transaction.

The scope of the transaction is limited to the connection. The following example performs an explicit transaction that consists of two separate commands in the `try` block. The commands execute INSERT statements against the Production.ScrapReason table in the AdventureWorks SQL Server sample database, which are committed if no exceptions are thrown. The code in the `catch` block rolls back the transaction if an exception is thrown. If the transaction is aborted or the connection is closed before the transaction has completed, it is automatically rolled back.

Example

Follow these steps to perform a transaction.

1. Call the BeginTransaction method of the SqlConnection object to mark the start of the transaction. The BeginTransaction method returns a reference to the transaction. This reference is assigned to the

SqlCommand objects that are enlisted in the transaction.

2. Assign the `Transaction` object to the Transaction property of the SqlCommand to be executed. If a command is executed on a connection with an active transaction, and the `Transaction` object has not been assigned to the `Transaction` property of the `Command` object, an exception is thrown.

3. Execute the required commands.

4. Call the Commit method of the SqlTransaction object to complete the transaction, or call the Rollback method to end the transaction. If the connection is closed or disposed before either the Commit or Rollback methods have been executed, the transaction is rolled back.

The following code example demonstrates transactional logic using ADO.NET with Microsoft SQL Server.

```
using (SqlConnection connection = new SqlConnection(connectionString))
{
    connection.Open();

    // Start a local transaction.
    SqlTransaction sqlTran = connection.BeginTransaction();

    // Enlist a command in the current transaction.
    SqlCommand command = connection.CreateCommand();
    command.Transaction = sqlTran;

    try
    {
        // Execute two separate commands.
        command.CommandText =
          "INSERT INTO Production.ScrapReason(Name) VALUES('Wrong size')";
        command.ExecuteNonQuery();
        command.CommandText =
          "INSERT INTO Production.ScrapReason(Name) VALUES('Wrong color')";
        command.ExecuteNonQuery();

        // Commit the transaction.
        sqlTran.Commit();
        Console.WriteLine("Both records were written to database.");
    }
    catch (Exception ex)
    {
        // Handle the exception if the transaction fails to commit.
        Console.WriteLine(ex.Message);

        try
        {
            // Attempt to roll back the transaction.
            sqlTran.Rollback();
        }
        catch (Exception exRollback)
        {
            // Throws an InvalidOperationException if the connection
            // is closed or the transaction has already been rolled
            // back on the server.
            Console.WriteLine(exRollback.Message);
        }
    }
}
```

```
Using connection As New SqlConnection(connectionString)
    connection.Open()

    ' Start a local transaction.
    Dim sqlTran As SqlTransaction = connection.BeginTransaction()

    ' Enlist a command in the current transaction.
    Dim command As SqlCommand = connection.CreateCommand()
    command.Transaction = sqlTran

    Try
        ' Execute two separate commands.
        command.CommandText = _
          "INSERT INTO Production.ScrapReason(Name) VALUES('Wrong size')"
        command.ExecuteNonQuery()
        command.CommandText = _
          "INSERT INTO Production.ScrapReason(Name) VALUES('Wrong color')"
        command.ExecuteNonQuery()

        ' Commit the transaction
        sqlTran.Commit()
        Console.WriteLine("Both records were written to database.")

    Catch ex As Exception
        ' Handle the exception if the transaction fails to commit.
        Console.WriteLine(ex.Message)

        Try
            ' Attempt to roll back the transaction.
            sqlTran.Rollback()

        Catch exRollback As Exception
            ' Throws an InvalidOperationException if the connection
            ' is closed or the transaction has already been rolled
            ' back on the server.
            Console.WriteLine(exRollback.Message)
        End Try
    End Try
End Using
```

See also

- Transactions and Concurrency
- Distributed Transactions
- System.Transactions Integration with SQL Server
- ADO.NET Overview

Distributed Transactions

9/7/2019 • 4 minutes to read • Edit Online

A transaction is a set of related tasks that either succeeds (commit) or fails (abort) as a unit, among other things. A *distributed transaction* is a transaction that affects several resources. For a distributed transaction to commit, all participants must guarantee that any change to data will be permanent. Changes must persist despite system crashes or other unforeseen events. If even a single participant fails to make this guarantee, the entire transaction fails, and any changes to data within the scope of the transaction are rolled back.

> **NOTE**
>
> An exception will be thrown if you attempt to commit or roll back a transaction if a `DataReader` is started while the transaction is active.

Working with System.Transactions

In the .NET Framework, distributed transactions are managed through the API in the System.Transactions namespace. The System.Transactions API will delegate distributed transaction handling to a transaction monitor such as the Microsoft Distributed Transaction Coordinator (MS DTC) when multiple persistent resource managers are involved. For more information, see Transaction Fundamentals.

ADO.NET 2.0 introduced support for enlisting in a distributed transaction using the `EnlistTransaction` method, which enlists a connection in a Transaction instance. In previous versions of ADO.NET, explicit enlistment in distributed transactions was performed using the `EnlistDistributedTransaction` method of a connection to enlist a connection in a ITransaction instance, which is supported for backwards compatibility. For more information on Enterprise Services transactions, see Interoperability with Enterprise Services and COM+ Transactions.

When using a System.Transactions transaction with the .NET Framework Provider for SQL Server against a SQL Server database, a lightweight Transaction will automatically be used. The transaction can then be promoted to a full distributed transaction on an as-needed basis. For more information, see System.Transactions Integration with SQL Server.

> **NOTE**
>
> The maximum number of distributed transactions that an Oracle database can participate in at one time is set to 10 by default. After the 10th transaction when connected to an Oracle database, an exception is thrown. Oracle does not support `DDL` inside of a distributed transaction.

Automatically Enlisting in a Distributed Transaction

Automatic enlistment is the default (and preferred) way of integrating ADO.NET connections with `System.Transactions`. A connection object will automatically enlist in an existing distributed transaction if it determines that a transaction is active, which, in `System.Transaction` terms, means that `Transaction.Current` is not null. Automatic transaction enlistment occurs when the connection is opened. It will not happen after that even if a command is executed inside of a transaction scope. You can disable auto-enlistment in existing transactions by specifying `Enlist=false` as a connection string parameter for a SqlConnection.ConnectionString, or `OLE DB Services=-7` as a connection string parameter for an OleDbConnection.ConnectionString. For more information on Oracle and ODBC connection string parameters, see OracleConnection.ConnectionString and OdbcConnection.ConnectionString.

Manually Enlisting in a Distributed Transaction

If auto-enlistment is disabled or you need to enlist a transaction that was started after the connection was opened, you can enlist in an existing distributed transaction using the `EnlistTransaction` method of the DbConnection object for the provider you are working with. Enlisting in an existing distributed transaction ensures that, if the transaction is committed or rolled back, modifications made by the code at the data source will be committed or rolled back as well.

Enlisting in distributed transactions is particularly applicable when pooling business objects. If a business object is pooled with an open connection, automatic transaction enlistment only occurs when that connection is opened. If multiple transactions are performed using the pooled business object, the open connection for that object will not automatically enlist in newly initiated transactions. In this case, you can disable automatic transaction enlistment for the connection and enlist the connection in transactions using `EnlistTransaction`.

`EnlistTransaction` takes a single argument of type Transaction that is a reference to the existing transaction. After calling the connection's `EnlistTransaction` method, all modifications made at the data source using the connection are included in the transaction. Passing a null value unenlists the connection from its current distributed transaction enlistment. Note that the connection must be opened before calling `EnlistTransaction`.

> **NOTE**
> Once a connection is explicitly enlisted on a transaction, it cannot be un-enlisted or enlisted in another transaction until the first transaction finishes.

Caution

`EnlistTransaction` throws an exception if the connection has already begun a transaction using the connection's BeginTransaction method. However, if the transaction is a local transaction started at the data source (for example, executing the BEGIN TRANSACTION statement explicitly using a SqlCommand), `EnlistTransaction` will roll back the local transaction and enlist in the existing distributed transaction as requested. You will not receive notice that the local transaction was rolled back, and must manage any local transactions not started using BeginTransaction. If you are using the .NET Framework Data Provider for SQL Server (`SqlClient`) with SQL Server, an attempt to enlist will throw an exception. All other cases will go undetected.

Promotable Transactions in SQL Server

SQL Server supports promotable transactions in which a local lightweight transaction can be automatically promoted to a distributed transaction only if it is required. A promotable transaction does not invoke the added overhead of a distributed transaction unless the added overhead is required. For more information and a code sample, see System.Transactions Integration with SQL Server.

Configuring Distributed Transactions

You may need to enable the MS DTC over the network in order to use distributed transactions. If have the Windows Firewall enabled, you must allow the MS DTC service to use the network or open the MS DTC port.

See also

- Transactions and Concurrency
- System.Transactions Integration with SQL Server
- ADO.NET Overview

System.Transactions Integration with SQL Server

3/12/2020 • 9 minutes to read • Edit Online

The .NET Framework version 2.0 introduced a transaction framework that can be accessed through the System.Transactions namespace. This framework exposes transactions in a way that is fully integrated in the .NET Framework, including ADO.NET.

In addition to the programmability enhancements, System.Transactions and ADO.NET can work together to coordinate optimizations when you work with transactions. A promotable transaction is a lightweight (local) transaction that can be automatically promoted to a fully distributed transaction on an as-needed basis.

Starting with ADO.NET 2.0, System.Data.SqlClient supports promotable transactions when you work with SQL Server. A promotable transaction does not invoke the added overhead of a distributed transaction unless the added overhead is required. Promotable transactions are automatic and require no intervention from the developer.

Promotable transactions are only available when you use the .NET Framework Data Provider for SQL Server (`SqlClient`) with SQL Server.

Creating Promotable Transactions

The .NET Framework Provider for SQL Server provides support for promotable transactions, which are handled through the classes in the .NET Framework System.Transactions namespace. Promotable transactions optimize distributed transactions by deferring creating a distributed transaction until it is needed. If only one resource manager is required, no distributed transaction occurs.

> **NOTE**
>
> In a partially trusted scenario, the DistributedTransactionPermission is required when a transaction is promoted to a distributed transaction.

Promotable Transaction Scenarios

Distributed transactions typically consume significant system resources, being managed by Microsoft Distributed Transaction Coordinator (MS DTC), which integrates all the resource managers accessed in the transaction. A promotable transaction is a special form of a System.Transactions transaction that effectively delegates the work to a simple SQL Server transaction. System.Transactions, System.Data.SqlClient, and SQL Server coordinate the work involved in handling the transaction, promoting it to a full distributed transaction as needed.

The benefit of using promotable transactions is that when a connection is opened by using an active TransactionScope transaction, and no other connections are opened, the transaction commits as a lightweight transaction, instead of incurring the additional overhead of a full distributed transaction.

Connection String Keywords

The ConnectionString property supports a keyword, `Enlist` , which indicates whether System.Data.SqlClient will detect transactional contexts and automatically enlist the connection in a distributed transaction. If `Enlist=true` , the connection is automatically enlisted in the opening thread's current transaction context. If `Enlist=false` , the `SqlClient` connection does not interact with a distributed transaction. The default value for `Enlist` is true. If `Enlist` is not specified in the connection string, the connection is automatically enlisted in a distributed transaction if one is detected when the connection is opened.

The `Transaction Binding` keywords in a SqlConnection connection string control the connection's association with an enlisted `System.Transactions` transaction. It is also available through the TransactionBinding property of a SqlConnectionStringBuilder.

The following table describes the possible values.

KEYWORD	DESCRIPTION
Implicit Unbind	The default. The connection detaches from the transaction when it ends, switching back to autocommit mode.
Explicit Unbind	The connection remains attached to the transaction until the transaction is closed. The connection will fail if the associated transaction is not active or does not match Current.

Using TransactionScope

The TransactionScope class makes a code block transactional by implicitly enlisting connections in a distributed transaction. You must call the Complete method at the end of the TransactionScope block before leaving it. Leaving the block invokes the Dispose method. If an exception has been thrown that causes the code to leave scope, the transaction is considered aborted.

We recommend that you use a `using` block to make sure that Dispose is called on the TransactionScope object when the using block is exited. Failure to commit or roll back pending transactions can significantly damage performance because the default time-out for the TransactionScope is one minute. If you do not use a `using` statement, you must perform all work in a `Try` block and explicitly call the Dispose method in the `Finally` block.

If an exception occurs in the TransactionScope, the transaction is marked as inconsistent and is abandoned. It will be rolled back when the TransactionScope is disposed. If no exception occurs, participating transactions commit.

> **NOTE**
> The `TransactionScope` class creates a transaction with a IsolationLevel of `Serializable` by default. Depending on your application, you might want to consider lowering the isolation level to avoid high contention in your application.

> **NOTE**
> We recommend that you perform only updates, inserts, and deletes within distributed transactions because they consume significant database resources. Select statements may lock database resources unnecessarily, and in some scenarios, you may have to use transactions for selects. Any non-database work should be done outside the scope of the transaction, unless it involves other transacted resource managers. Although an exception in the scope of the transaction prevents the transaction from committing, the TransactionScope class has no provision for rolling back any changes your code has made outside the scope of the transaction itself. If you have to take some action when the transaction is rolled back, you must write your own implementation of the IEnlistmentNotification interface and explicitly enlist in the transaction.

Example

Working with System.Transactions requires that you have a reference to System.Transactions.dll.

The following function demonstrates how to create a promotable transaction against two different SQL Server instances, represented by two different SqlConnection objects, which are wrapped in a TransactionScope block. The code creates the TransactionScope block with a `using` statement and opens the first connection, which automatically enlists it in the TransactionScope. The transaction is initially enlisted as a lightweight transaction, not a full distributed transaction. The second connection is enlisted in the TransactionScope only if the command in

the first connection does not throw an exception. When the second connection is opened, the transaction is automatically promoted to a full distributed transaction. The Complete method is invoked, which commits the transaction only if no exceptions have been thrown. If an exception has been thrown at any point in the TransactionScope block, `Complete` will not be called, and the distributed transaction will roll back when the TransactionScope is disposed at the end of its `using` block.

```
// This function takes arguments for the 2 connection strings and commands in order
// to create a transaction involving two SQL Servers. It returns a value > 0 if the
// transaction committed, 0 if the transaction rolled back. To test this code, you can
// connect to two different databases on the same server by altering the connection string,
// or to another RDBMS such as Oracle by altering the code in the connection2 code block.
static public int CreateTransactionScope(
    string connectString1, string connectString2,
    string commandText1, string commandText2)
{
    // Initialize the return value to zero and create a StringWriter to display results.
    int returnValue = 0;
    System.IO.StringWriter writer = new System.IO.StringWriter();

    // Create the TransactionScope in which to execute the commands, guaranteeing
    // that both commands will commit or roll back as a single unit of work.
    using (TransactionScope scope = new TransactionScope())
    {
        using (SqlConnection connection1 = new SqlConnection(connectString1))
        {
            try
            {
                // Opening the connection automatically enlists it in the
                // TransactionScope as a lightweight transaction.
                connection1.Open();

                // Create the SqlCommand object and execute the first command.
                SqlCommand command1 = new SqlCommand(commandText1, connection1);
                returnValue = command1.ExecuteNonQuery();
                writer.WriteLine("Rows to be affected by command1: {0}", returnValue);

                // if you get here, this means that command1 succeeded. By nesting
                // the using block for connection2 inside that of connection1, you
                // conserve server and network resources by opening connection2
                // only when there is a chance that the transaction can commit.
                using (SqlConnection connection2 = new SqlConnection(connectString2))
                    try
                    {
                        // The transaction is promoted to a full distributed
                        // transaction when connection2 is opened.
                        connection2.Open();

                        // Execute the second command in the second database.
                        returnValue = 0;
                        SqlCommand command2 = new SqlCommand(commandText2, connection2);
                        returnValue = command2.ExecuteNonQuery();
                        writer.WriteLine("Rows to be affected by command2: {0}", returnValue);
                    }
                    catch (Exception ex)
                    {
                        // Display information that command2 failed.
                        writer.WriteLine("returnValue for command2: {0}", returnValue);
                        writer.WriteLine("Exception Message2: {0}", ex.Message);
                    }
            }
            catch (Exception ex)
            {
                // Display information that command1 failed.
                writer.WriteLine("returnValue for command1: {0}", returnValue);
                writer.WriteLine("Exception Message1: {0}", ex.Message);
            }
        }
```

```
        }

        // If an exception has been thrown, Complete will not
        // be called and the transaction is rolled back.
        scope.Complete();
    }

    // The returnValue is greater than 0 if the transaction committed.
    if (returnValue > 0)
    {
        writer.WriteLine("Transaction was committed.");
    }
    else
    {
        // You could write additional business logic here, notify the caller by
        // throwing a TransactionAbortedException, or log the failure.
        writer.WriteLine("Transaction rolled back.");
    }

    // Display messages.
    Console.WriteLine(writer.ToString());

    return returnValue;
}
```

```
' This function takes arguments for the 2 connection strings and commands in order
' to create a transaction involving two SQL Servers. It returns a value > 0 if the
' transaction committed, 0 if the transaction rolled back. To test this code, you can
' connect to two different databases on the same server by altering the connection string,
' or to another RDBMS such as Oracle by altering the code in the connection2 code block.
Public Function CreateTransactionScope( _
  ByVal connectString1 As String, ByVal connectString2 As String, _
  ByVal commandText1 As String, ByVal commandText2 As String) As Integer

    ' Initialize the return value to zero and create a StringWriter to display results.
    Dim returnValue As Integer = 0
    Dim writer As System.IO.StringWriter = New System.IO.StringWriter

    ' Create the TransactionScope in which to execute the commands, guaranteeing
    ' that both commands will commit or roll back as a single unit of work.
    Using scope As New TransactionScope()
        Using connection1 As New SqlConnection(connectString1)
            Try
                ' Opening the connection automatically enlists it in the
                ' TransactionScope as a lightweight transaction.
                connection1.Open()

                ' Create the SqlCommand object and execute the first command.
                Dim command1 As SqlCommand = New SqlCommand(commandText1, connection1)
                returnValue = command1.ExecuteNonQuery()
                writer.WriteLine("Rows to be affected by command1: {0}", returnValue)

                ' If you get here, this means that command1 succeeded. By nesting
                ' the Using block for connection2 inside that of connection1, you
                ' conserve server and network resources by opening connection2
                ' only when there is a chance that the transaction can commit.
                Using connection2 As New SqlConnection(connectString2)
                    Try
                        ' The transaction is promoted to a full distributed
                        ' transaction when connection2 is opened.
                        connection2.Open()

                        ' Execute the second command in the second database.
                        returnValue = 0
                        Dim command2 As SqlCommand = New SqlCommand(commandText2, connection2)
                        returnValue = command2.ExecuteNonQuery()
                        writer.WriteLine("Rows to be affected by command2: {0}", returnValue)
```

```
                    Catch ex As Exception
                        ' Display information that command2 failed.
                        writer.WriteLine("returnValue for command2: {0}", returnValue)
                        writer.WriteLine("Exception Message2: {0}", ex.Message)
                    End Try
                End Using

            Catch ex As Exception
                ' Display information that command1 failed.
                writer.WriteLine("returnValue for command1: {0}", returnValue)
                writer.WriteLine("Exception Message1: {0}", ex.Message)
            End Try
        End Using

        ' If an exception has been thrown, Complete will
        ' not be called and the transaction is rolled back.
        scope.Complete()
    End Using

    ' The returnValue is greater than 0 if the transaction committed.
    If returnValue > 0 Then
        writer.WriteLine("Transaction was committed.")
    Else
        ' You could write additional business logic here, notify the caller by
        ' throwing a TransactionAbortedException, or log the failure.
       writer.WriteLine("Transaction rolled back.")
     End If

    ' Display messages.
    Console.WriteLine(writer.ToString())

    Return returnValue
End Function
```

See also

- Transactions and Concurrency
- ADO.NET Overview

Optimistic Concurrency

3/12/2020 • 8 minutes to read • Edit Online

In a multiuser environment, there are two models for updating data in a database: optimistic concurrency and pessimistic concurrency. The DataSet object is designed to encourage the use of optimistic concurrency for long-running activities, such as remoting data and interacting with data.

Pessimistic concurrency involves locking rows at the data source to prevent other users from modifying data in a way that affects the current user. In a pessimistic model, when a user performs an action that causes a lock to be applied, other users cannot perform actions that would conflict with the lock until the lock owner releases it. This model is primarily used in environments where there is heavy contention for data, so that the cost of protecting data with locks is less than the cost of rolling back transactions if concurrency conflicts occur.

Therefore, in a pessimistic concurrency model, a user who updates a row establishes a lock. Until the user has finished the update and released the lock, no one else can change that row. For this reason, pessimistic concurrency is best implemented when lock times will be short, as in programmatic processing of records. Pessimistic concurrency is not a scalable option when users are interacting with data and causing records to be locked for relatively large periods of time.

> **NOTE**
>
> If you need to update multiple rows in the same operation, then creating a transaction is a more scalable option than using pessimistic locking.

By contrast, users who use optimistic concurrency do not lock a row when reading it. When a user wants to update a row, the application must determine whether another user has changed the row since it was read. Optimistic concurrency is generally used in environments with a low contention for data. Optimistic concurrency improves performance because no locking of records is required, and locking of records requires additional server resources. Also, in order to maintain record locks, a persistent connection to the database server is required. Because this is not the case in an optimistic concurrency model, connections to the server are free to serve a larger number of clients in less time.

In an optimistic concurrency model, a violation is considered to have occurred if, after a user receives a value from the database, another user modifies the value before the first user has attempted to modify it. How the server resolves a concurrency violation is best shown by first describing the following example.

The following tables follow an example of optimistic concurrency.

At 1:00 p.m., User1 reads a row from the database with the following values:

CustID LastName FirstName

101 Smith Bob

COLUMN NAME	ORIGINAL VALUE	CURRENT VALUE	VALUE IN DATABASE
CustID	101	101	101
LastName	Smith	Smith	Smith
FirstName	Bob	Bob	Bob

At 1:01 p.m., User2 reads the same row.

At 1:03 p.m., User2 changes **FirstName** from "Bob" to "Robert" and updates the database.

COLUMN NAME	ORIGINAL VALUE	CURRENT VALUE	VALUE IN DATABASE
CustID	101	101	101
LastName	Smith	Smith	Smith
FirstName	Bob	Robert	Bob

The update succeeds because the values in the database at the time of update match the original values that User2 has.

At 1:05 p.m., User1 changes "Bob"'s first name to "James" and tries to update the row.

COLUMN NAME	ORIGINAL VALUE	CURRENT VALUE	VALUE IN DATABASE
CustID	101	101	101
LastName	Smith	Smith	Smith
FirstName	Bob	James	Robert

At this point, User1 encounters an optimistic concurrency violation because the value in the database ("Robert") no longer matches the original value that User1 was expecting ("Bob"). The concurrency violation simply lets you know that the update failed. The decision now needs to be made whether to overwrite the changes supplied by User2 with the changes supplied by User1, or to cancel the changes by User1.

Testing for Optimistic Concurrency Violations

There are several techniques for testing for an optimistic concurrency violation. One involves including a timestamp column in the table. Databases commonly provide timestamp functionality that can be used to identify the date and time when the record was last updated. Using this technique, a timestamp column is included in the table definition. Whenever the record is updated, the timestamp is updated to reflect the current date and time. In a test for optimistic concurrency violations, the timestamp column is returned with any query of the contents of the table. When an update is attempted, the timestamp value in the database is compared to the original timestamp value contained in the modified row. If they match, the update is performed and the timestamp column is updated with the current time to reflect the update. If they do not match, an optimistic concurrency violation has occurred.

Another technique for testing for an optimistic concurrency violation is to verify that all the original column values in a row still match those found in the database. For example, consider the following query:

```
SELECT Col1, Col2, Col3 FROM Table1
```

To test for an optimistic concurrency violation when updating a row in **Table1**, you would issue the following UPDATE statement:

```
UPDATE Table1 Set Col1 = @NewCol1Value,
              Set Col2 = @NewCol2Value,
              Set Col3 = @NewCol3Value
WHERE Col1 = @OldCol1Value AND
      Col2 = @OldCol2Value AND
      Col3 = @OldCol3Value
```

As long as the original values match the values in the database, the update is performed. If a value has been modified, the update will not modify the row because the WHERE clause will not find a match.

Note that it is recommended to always return a unique primary key value in your query. Otherwise, the preceding UPDATE statement may update more than one row, which might not be your intent.

If a column at your data source allows nulls, you may need to extend your WHERE clause to check for a matching null reference in your local table and at the data source. For example, the following UPDATE statement verifies that a null reference in the local row still matches a null reference at the data source, or that the value in the local row still matches the value at the data source.

```
UPDATE Table1 Set Col1 = @NewVal1
  WHERE (@OldVal1 IS NULL AND Col1 IS NULL) OR Col1 = @OldVal1
```

You may also choose to apply less restrictive criteria when using an optimistic concurrency model. For example, using only the primary key columns in the WHERE clause causes the data to be overwritten regardless of whether the other columns have been updated since the last query. You can also apply a WHERE clause only to specific columns, resulting in data being overwritten unless particular fields have been updated since they were last queried.

The DataAdapter.RowUpdated Event

The **RowUpdated** event of the DataAdapter object can be used in conjunction with the techniques described earlier, to provide notification to your application of optimistic concurrency violations. **RowUpdated** occurs after each attempt to update a **Modified** row from a **DataSet**. This enables you to add special handling code, including processing when an exception occurs, adding custom error information, adding retry logic, and so on. The RowUpdatedEventArgs object returns a **RecordsAffected** property containing the number of rows affected by a particular update command for a modified row in a table. By setting the update command to test for optimistic concurrency, the **RecordsAffected** property will, as a result, return a value of 0 when an optimistic concurrency violation has occurred, because no records were updated. If this is the case, an exception is thrown. The **RowUpdated** event enables you to handle this occurrence and avoid the exception by setting an appropriate **RowUpdatedEventArgs.Status** value, such as **UpdateStatus.SkipCurrentRow**. For more information about the **RowUpdated** event, see Handling DataAdapter Events.

Optionally, you can set **DataAdapter.ContinueUpdateOnError** to **true**, before calling **Update**, and respond to the error information stored in the **RowError** property of a particular row when the **Update** is completed. For more information, see Row Error Information.

Optimistic Concurrency Example

The following is a simple example that sets the **UpdateCommand** of a **DataAdapter** to test for optimistic concurrency, and then uses the **RowUpdated** event to test for optimistic concurrency violations. When an optimistic concurrency violation is encountered, the application sets the **RowError** of the row that the update was issued for to reflect an optimistic concurrency violation.

Note that the parameter values passed to the WHERE clause of the UPDATE command are mapped to the **Original** values of their respective columns.

```
' Assumes connection is a valid SqlConnection.
Dim adapter As SqlDataAdapter = New SqlDataAdapter( _
  "SELECT CustomerID, CompanyName FROM Customers ORDER BY CustomerID", _
  connection)

' The Update command checks for optimistic concurrency violations
' in the WHERE clause.
adapter.UpdateCommand = New SqlCommand("UPDATE Customers " &
  "(CustomerID, CompanyName) VALUES(@CustomerID, @CompanyName) " & _
  "WHERE CustomerID = @oldCustomerID AND CompanyName = " &
  "@oldCompanyName", connection)
adapter.UpdateCommand.Parameters.Add( _
  "@CustomerID", SqlDbType.NChar, 5, "CustomerID")
adapter.UpdateCommand.Parameters.Add( _
  "@CompanyName", SqlDbType.NVarChar, 30, "CompanyName")

' Pass the original values to the WHERE clause parameters.
Dim parameter As SqlParameter = adapter.UpdateCommand.Parameters.Add( _
  "@oldCustomerID", SqlDbType.NChar, 5, "CustomerID")
parameter.SourceVersion = DataRowVersion.Original
parameter = adapter.UpdateCommand.Parameters.Add( _
  "@oldCompanyName", SqlDbType.NVarChar, 30, "CompanyName")
parameter.SourceVersion = DataRowVersion.Original

' Add the RowUpdated event handler.
AddHandler adapter.RowUpdated, New SqlRowUpdatedEventHandler( _
  AddressOf OnRowUpdated)

Dim dataSet As DataSet = New DataSet()
adapter.Fill(dataSet, "Customers")

' Modify the DataSet contents.
adapter.Update(dataSet, "Customers")

Dim dataRow As DataRow

For Each dataRow In dataSet.Tables("Customers").Rows
    If dataRow.HasErrors Then
       Console.WriteLine(dataRow (0) & vbCrLf & dataRow.RowError)
    End If
Next

Private Shared Sub OnRowUpdated( _
  sender As object, args As SqlRowUpdatedEventArgs)
   If args.RecordsAffected = 0
      args.Row.RowError = "Optimistic Concurrency Violation!"
      args.Status = UpdateStatus.SkipCurrentRow
   End If
End Sub
```

```
// Assumes connection is a valid SqlConnection.
SqlDataAdapter adapter = new SqlDataAdapter(
  "SELECT CustomerID, CompanyName FROM Customers ORDER BY CustomerID",
  connection);

// The Update command checks for optimistic concurrency violations
// in the WHERE clause.
adapter.UpdateCommand = new SqlCommand("UPDATE Customers Set CustomerID = @CustomerID, CompanyName =
@CompanyName " +
   "WHERE CustomerID = @oldCustomerID AND CompanyName = @oldCompanyName", connection);
adapter.UpdateCommand.Parameters.Add(
  "@CustomerID", SqlDbType.NChar, 5, "CustomerID");
adapter.UpdateCommand.Parameters.Add(
  "@CompanyName", SqlDbType.NVarChar, 30, "CompanyName");

// Pass the original values to the WHERE clause parameters.
SqlParameter parameter = adapter.UpdateCommand.Parameters.Add(
  "@oldCustomerID", SqlDbType.NChar, 5, "CustomerID");
parameter.SourceVersion = DataRowVersion.Original;
parameter = adapter.UpdateCommand.Parameters.Add(
  "@oldCompanyName", SqlDbType.NVarChar, 30, "CompanyName");
parameter.SourceVersion = DataRowVersion.Original;

// Add the RowUpdated event handler.
adapter.RowUpdated += new SqlRowUpdatedEventHandler(OnRowUpdated);

DataSet dataSet = new DataSet();
adapter.Fill(dataSet, "Customers");

// Modify the DataSet contents.

adapter.Update(dataSet, "Customers");

foreach (DataRow dataRow in dataSet.Tables["Customers"].Rows)
{
    if (dataRow.HasErrors)
       Console.WriteLine(dataRow [0] + "\n" + dataRow.RowError);
}

protected static void OnRowUpdated(object sender, SqlRowUpdatedEventArgs args)
{
  if (args.RecordsAffected == 0)
  {
    args.Row.RowError = "Optimistic Concurrency Violation Encountered";
    args.Status = UpdateStatus.SkipCurrentRow;
  }
}
```

See also

- Retrieving and Modifying Data in ADO.NET
- Updating Data Sources with DataAdapters
- Row Error Information
- Transactions and Concurrency
- ADO.NET Overview

Retrieving Identity or Autonumber Values

9/7/2019 • 24 minutes to read • Edit Online

A primary key in a relational database is a column or combination of columns that always contain unique values. Knowing the primary key value allows you to locate the row that contains it. Relational database engines, such as SQL Server, Oracle, and Microsoft Access/Jet support the creation of automatically incrementing columns that can be designated as primary keys. These values are generated by the server as rows are added to a table. In SQL Server, you set the identity property of a column, in Oracle you create a Sequence, and in Microsoft Access you create an AutoNumber column.

A DataColumn can also be used to generate automatically incrementing values by setting the AutoIncrement property to true. However, you might end up with duplicate values in separate instances of a DataTable, if multiple client applications are independently generating automatically incrementing values. Having the server generate automatically incrementing values eliminates potential conflicts by allowing each user to retrieve the generated value for each inserted row.

During a call to the `Update` method of a `DataAdapter`, the database can send data back to your ADO.NET application as output parameters or as the first returned record of the result set of a SELECT statement executed in the same batch as the INSERT statement. ADO.NET can retrieve these values and update the corresponding columns in the DataRow being updated.

Some database engines, such as the Microsoft Access Jet database engine, do not support output parameters and cannot process multiple statements in a single batch. When working with the Jet database engine, you can retrieve the new AutoNumber value generated for an inserted row by executing a separate SELECT command in an event handler for the `RowUpdated` event of the `DataAdapter`.

> **NOTE**
>
> An alternative to using an auto incrementing value is to use the NewGuid method of a Guid object to generate a GUID, or globally unique identifier, on the client computer that can be copied to the server as each new row is inserted. The `NewGuid` method generates a 16-byte binary value that is created using an algorithm that provides a high probability that no value will be duplicated. In a SQL Server database, a GUID is stored in a `uniqueidentifier` column which SQL Server can automatically generate using the Transact-SQL `NEWID()` function. Using a GUID as a primary key can adversely affect performance. SQL Server provides support for the `NEWSEQUENTIALID()` function, which generates a sequential GUID that is not guaranteed to be globally unique but that can be indexed more efficiently.

Retrieving SQL Server Identity Column Values

When working with Microsoft SQL Server, you can create a stored procedure with an output parameter to return the identity value for an inserted row. The following table describes the three Transact-SQL functions in SQL Server that can be used to retrieve identity column values.

FUNCTION	DESCRIPTION
SCOPE_IDENTITY	Returns the last identity value within the current execution scope. SCOPE_IDENTITY is recommended for most scenarios.
@@IDENTITY	Contains the last identity value generated in any table in the current session. @@IDENTITY can be affected by triggers and may not return the identity value that you expect.

FUNCTION	DESCRIPTION
IDENT_CURRENT	Returns the last identity value generated for a specific table in any session and any scope.

The following stored procedure demonstrates how to insert a row into the **Categories** table and use an output parameter to return the new identity value generated by the Transact-SQL SCOPE_IDENTITY() function.

```
CREATE PROCEDURE dbo.InsertCategory
  @CategoryName nvarchar(15),
  @Identity int OUT
AS
INSERT INTO Categories (CategoryName) VALUES(@CategoryName)
SET @Identity = SCOPE_IDENTITY()
```

The stored procedure can then be specified as the source of the InsertCommand of a SqlDataAdapter object. The CommandType property of the InsertCommand must be set to StoredProcedure. The identity output is retrieved by creating a SqlParameter that has a ParameterDirection of Output. When the `InsertCommand` is processed, the auto-incremented identity value is returned and placed in the **CategoryID** column of the current row if you set the UpdatedRowSource property of the insert command to `UpdateRowSource.OutputParameters` or to `UpdateRowSource.Both`.

If your insert command executes a batch that includes both an INSERT statement and a SELECT statement that returns the new identity value, then you can retrieve the new value by setting the `UpdatedRowSource` property of the insert command to `UpdateRowSource.FirstReturnedRecord`.

```
private static void RetrieveIdentity(string connectionString)
{
    using (SqlConnection connection =
               new SqlConnection(connectionString))
    {
        // Create a SqlDataAdapter based on a SELECT query.
        SqlDataAdapter adapter =
            new SqlDataAdapter(
            "SELECT CategoryID, CategoryName FROM dbo.Categories",
            connection);

        //Create the SqlCommand to execute the stored procedure.
        adapter.InsertCommand = new SqlCommand("dbo.InsertCategory",
            connection);
        adapter.InsertCommand.CommandType = CommandType.StoredProcedure;

        // Add the parameter for the CategoryName. Specifying the
        // ParameterDirection for an input parameter is not required.
        adapter.InsertCommand.Parameters.Add(
           new SqlParameter("@CategoryName", SqlDbType.NVarChar, 15,
           "CategoryName"));

        // Add the SqlParameter to retrieve the new identity value.
        // Specify the ParameterDirection as Output.
        SqlParameter parameter =
            adapter.InsertCommand.Parameters.Add(
            "@Identity", SqlDbType.Int, 0, "CategoryID");
        parameter.Direction = ParameterDirection.Output;

        // Create a DataTable and fill it.
        DataTable categories = new DataTable();
        adapter.Fill(categories);

        // Add a new row.
        DataRow newRow = categories.NewRow();
        newRow["CategoryName"] = "New Category";
        categories.Rows.Add(newRow);

        adapter.Update(categories);

        Console.WriteLine("List All Rows:");
        foreach (DataRow row in categories.Rows)
        {
            {
                Console.WriteLine("{0}: {1}", row[0], row[1]);
            }
        }
    }
}
```

```
Private Sub RetrieveIdentity(ByVal connectionString As String)
    Using connection As SqlConnection = New SqlConnection( _
       connectionString)

        ' Create a SqlDataAdapter based on a SELECT query.
        Dim adapter As SqlDataAdapter = New SqlDataAdapter( _
           "SELECT CategoryID, CategoryName FROM dbo.Categories", _
           connection)

        ' Create the SqlCommand to execute the stored procedure.
        adapter.InsertCommand = New SqlCommand("dbo.InsertCategory", _
           connection)
        adapter.InsertCommand.CommandType = CommandType.StoredProcedure

        ' Add the parameter for the CategoryName. Specifying the
        ' ParameterDirection for an input parameter is not required.
        adapter.InsertCommand.Parameters.Add( _
          "@CategoryName", SqlDbType.NVarChar, 15, "CategoryName")

        ' Add the SqlParameter to retrieve the new identity value.
        ' Specify the ParameterDirection as Output.
        Dim parameter As SqlParameter = _
           adapter.InsertCommand.Parameters.Add( _
          "@Identity", SqlDbType.Int, 0, "CategoryID")
        parameter.Direction = ParameterDirection.Output

        ' Create a DataTable and fill it.
        Dim categories As DataTable = New DataTable
        adapter.Fill(categories)

        ' Add a new row.
        Dim newRow As DataRow = categories.NewRow()
        newRow("CategoryName") = "New Category"
        categories.Rows.Add(newRow)

        ' Update the database.
        adapter.Update(categories)

        Console.WriteLine("List All Rows:")
        Dim row As DataRow
        For Each row In categories.Rows
            Console.WriteLine("{0}: {1}", row(0), row(1))
        Next
    End Using
End Sub
```

Merging New Identity Values

A common scenario is to call the `GetChanges` method of a `DataTable` to create a copy that contains only changed rows, and to use the new copy when calling the `Update` method of a `DataAdapter`. This is especially useful when you need to marshal the changed rows to a separate component that performs the update. Following the update, the copy can contain new identity values that must then be merged back into the original `DataTable`. The new identity values are likely to be different from the original values in the `DataTable`. To accomplish the merge, the original values of the **AutoIncrement** columns in the copy must be preserved, in order to be able to locate and update existing rows in the original `DataTable`, rather than appending new rows containing the new identity values. However, by default those original values are lost after a call to the `Update` method of a `DataAdapter`, because `AcceptChanges` is implicitly called for each updated `DataRow`.

There are two ways to preserve the original values of a `DataColumn` in a `DataRow` during a `DataAdapter` update:

- The first method of preserving the original values is to set the `AcceptChangesDuringUpdate` property of the `DataAdapter` to `false`. This affects every `DataRow` in the `DataTable` being updated. For more information

and a code example, see AcceptChangesDuringUpdate.

- The second method is to write code in the `RowUpdated` event handler of the `DataAdapter` to set the Status to SkipCurrentRow. The `DataRow` is updated but the original value of each `DataColumn` is preserved. This method enables you to preserve the original values for some rows and not for others. For example, your code can preserve the original values for added rows and not for edited or deleted rows by first checking the StatementType and then setting Status to SkipCurrentRow only for rows with a `StatementType` of `Insert`.

When either of these methods is used to preserve original values in a `DataRow` during a `DataAdapter` update, ADO.NET performs a series of actions to set the current values of the `DataRow` to new values returned by output parameters or by the first returned row of a result set, while still preserving the original value in each `DataColumn`. First, the `AcceptChanges` method of the `DataRow` is called to preserve the current values as original values, and then the new values are assigned. Following these actions, `DataRows` that had their RowState property set to Added will have their `RowState` property set to Modified, which may be unexpected.

How the command results are applied to each DataRow being updated is determined by the UpdatedRowSource property of each DbCommand. This property is set to a value from the `UpdateRowSource` enumeration.

The following table describes how the `UpdateRowSource` enumeration values affect the RowState property of updated rows.

MEMBER NAME	DESCRIPTION
Both	`AcceptChanges` is called and both output parameter values and/or the values in the first row of any returned result set are placed in the `DataRow` being updated. If there are no values to apply, the `RowState` will be Unchanged.
FirstReturnedRecord	If a row was returned, `AcceptChanges` is called and the row is mapped to the changed row in the `DataTable`, setting the `RowState` to `Modified`. If no row is returned, then `AcceptChanges` is not called and the `RowState` remains `Added`.
None	Any returned parameters or rows are ignored. There is no call to `AcceptChanges` and the `RowState` remains `Added`.
OutputParameters	`AcceptChanges` is called and any output parameters are mapped to the changed row in the `DataTable`, setting the `RowState` to `Modified`. If there are no output parameters, the `RowState` will be `Unchanged`.

Example

This example demonstrates extracting changed rows from a `DataTable` and using a SqlDataAdapter to update the data source and retrieve a new identity column value. The InsertCommand executes two Transact-SQL statements; the first one is the INSERT statement, and the second one is a SELECT statement that uses the SCOPE_IDENTITY function to retrieve the identity value.

```
INSERT INTO dbo.Shippers (CompanyName)
VALUES (@CompanyName);
SELECT ShipperID, CompanyName FROM dbo.Shippers
WHERE ShipperID = SCOPE_IDENTITY();
```

The `UpdatedRowSource` property of the insert command is set to `UpdateRowSource.FirstReturnedRow` and the

MissingSchemaAction property of the `DataAdapter` is set to `MissingSchemaAction.AddWithKey`. The `DataTable` is filled and the code adds a new row to the `DataTable`. The changed rows are then extracted into a new `DataTable`, which is passed to the `DataAdapter`, which then updates the server.

```
private static void MergeIdentityColumns(string connectionString)
{
    using (SqlConnection connection =
                new SqlConnection(connectionString))
    {
        // Create the DataAdapter
        SqlDataAdapter adapter =
            new SqlDataAdapter(
            "SELECT ShipperID, CompanyName FROM dbo.Shippers",
            connection);

        //Add the InsertCommand to retrieve new identity value.
        adapter.InsertCommand = new SqlCommand(
            "INSERT INTO dbo.Shippers (CompanyName) " +
            "VALUES (@CompanyName); " +
            "SELECT ShipperID, CompanyName FROM dbo.Shippers " +
            "WHERE ShipperID = SCOPE_IDENTITY();", connection);

        // Add the parameter for the inserted value.
        adapter.InsertCommand.Parameters.Add(
           new SqlParameter("@CompanyName", SqlDbType.NVarChar, 40,
           "CompanyName"));
        adapter.InsertCommand.UpdatedRowSource = UpdateRowSource.Both;

        // MissingSchemaAction adds any missing schema to
        // the DataTable, including identity columns
        adapter.MissingSchemaAction = MissingSchemaAction.AddWithKey;

        // Fill the DataTable.
        DataTable shipper = new DataTable();
        adapter.Fill(shipper);

        // Add a new shipper.
        DataRow newRow = shipper.NewRow();
        newRow["CompanyName"] = "New Shipper";
        shipper.Rows.Add(newRow);

        // Add changed rows to a new DataTable. This
        // DataTable will be used by the DataAdapter.
        DataTable dataChanges = shipper.GetChanges();

        // Add the event handler.
        adapter.RowUpdated +=
            new SqlRowUpdatedEventHandler(OnRowUpdated);

        adapter.Update(dataChanges);
        connection.Close();

        // Merge the updates.
        shipper.Merge(dataChanges);

        // Commit the changes.
        shipper.AcceptChanges();

        Console.WriteLine("Rows after merge.");
        foreach (DataRow row in shipper.Rows)
        {
            {
                Console.WriteLine("{0}: {1}", row[0], row[1]);
            }
        }
    }
}
```

```
Private Sub MergeIdentityColumns(ByVal connectionString As String)

    Using connection As SqlConnection = New SqlConnection( _
       connectionString)

        ' Create the DataAdapter
        Dim adapter As SqlDataAdapter = New SqlDataAdapter( _
          "SELECT ShipperID, CompanyName FROM dbo.Shippers", connection)

        ' Add the InsertCommand to retrieve new identity value.
        adapter.InsertCommand = New SqlCommand( _
            "INSERT INTO dbo.Shippers (CompanyName) " & _
            "VALUES (@CompanyName); " & _
            "SELECT ShipperID, CompanyName FROM dbo.Shippers " & _
            "WHERE ShipperID = SCOPE_IDENTITY();", _
            connection)

        ' Add the parameter for the inserted value.
        adapter.InsertCommand.Parameters.Add( _
           New SqlParameter("@CompanyName", SqlDbType.NVarChar, 40, _
           "CompanyName"))
        adapter.InsertCommand.UpdatedRowSource = UpdateRowSource.Both

        ' MissingSchemaAction adds any missing schema to
        ' the DataTable, including identity columns
        adapter.MissingSchemaAction = MissingSchemaAction.AddWithKey

        ' Fill the DataTable.
        Dim shipper As New DataTable
        adapter.Fill(shipper)

        ' Add a new shipper.
        Dim newRow As DataRow = shipper.NewRow()
        newRow("CompanyName") = "New Shipper"
        shipper.Rows.Add(newRow)

        ' Add changed rows to a new DataTable. This
        ' DataTable will be used by the DataAdapter.
        Dim dataChanges As DataTable = shipper.GetChanges()

        ' Add the event handler.
        AddHandler adapter.RowUpdated, New _
           SqlRowUpdatedEventHandler(AddressOf OnRowUpdated)

        ' Update the datasource with the modified records.
        adapter.Update(dataChanges)

        ' Merge the two DataTables.
        shipper.Merge(dataChanges)

        ' Commit the changes.
        shipper.AcceptChanges()

        Console.WriteLine("Rows after merge.")
        Dim row As DataRow
        For Each row In shipper.Rows
            Console.WriteLine("{0}: {1}", row(0), row(1))
        Next
    End Using
End Sub
```

The `OnRowUpdated` event handler checks the StatementType of the SqlRowUpdatedEventArgs to determine if the row is an insert. If it is, then the Status property is set to SkipCurrentRow. The row is updated, but the original values in the row are preserved. In the main body of the procedure, the Merge method is called to merge the new identity value into the original `DataTable`, and finally `AcceptChanges` is called.

```
protected static void OnRowUpdated(
    object sender, SqlRowUpdatedEventArgs e)
{
    // If this is an insert, then skip this row.
    if (e.StatementType == StatementType.Insert)
    {
        e.Status = UpdateStatus.SkipCurrentRow;
    }
}
```

```
Private Sub OnRowUpdated( _
    ByVal sender As Object, ByVal e As SqlRowUpdatedEventArgs)
    ' If this is an insert, then skip this row.
    If e.StatementType = StatementType.Insert Then
        e.Status = UpdateStatus.SkipCurrentRow
    End If
End Sub
```

Retrieving Microsoft Access Autonumber Values

This section includes a sample that shows how to retrieve `Autonumber` values from a Jet 4.0 database. The Jet database engine does not support the execution of multiple statements in a batch or the use of output parameters, so it is not possible to use either of these techniques to return the new `Autonumber` value assigned to an inserted row. However, you can add code to the `RowUpdated` event handler that executes a separate SELECT @@IDENTITY statement to retrieve the new `Autonumber` value.

Example

Instead of adding schema information using `MissingSchemaAction.AddWithKey`, this example configures a `DataTable` with the correct schema prior to calling the OleDbDataAdapter to fill the `DataTable`. In this case, the **CategoryID** column is configured to decrement the value assigned each inserted row starting from zero, by setting AutoIncrement to `true`, AutoIncrementSeed to 0, and AutoIncrementStep to -1. The code then adds two new rows and uses `GetChanges` to add the changed rows to a new `DataTable` that is passed to the `Update` method.

```
private static OleDbConnection connection = null;

private static void MergeIdentityColumns(OleDbConnection connection)
{
    using (connection)
    {
        // Create a DataAdapter based on a SELECT query.
        OleDbDataAdapter adapter = new OleDbDataAdapter(
         "SELECT CategoryID, CategoryName FROM Categories",
         connection);

        // Create the INSERT command for the new category.
        adapter.InsertCommand = new OleDbCommand(
          "INSERT INTO Categories (CategoryName) Values(?)", connection);
        adapter.InsertCommand.CommandType = CommandType.Text;

        // Add the parameter for the CategoryName.
        adapter.InsertCommand.Parameters.Add(
          "@CategoryName", OleDbType.VarWChar, 15, "CategoryName");
        adapter.InsertCommand.UpdatedRowSource = UpdateRowSource.Both;

        // Create a DataTable
        DataTable categories = new DataTable();

        // Create the CategoryID column and set its auto
```

```
        // incrementing properties to decrement from zero.
        DataColumn column = new DataColumn();
        column.DataType = System.Type.GetType("System.Int32");
        column.ColumnName = "CategoryID";
        column.AutoIncrement = true;
        column.AutoIncrementSeed = 0;
        column.AutoIncrementStep = -1;
        categories.Columns.Add(column);

        // Create the CategoryName column.
        column = new DataColumn();
        column.DataType = System.Type.GetType("System.String");
        column.ColumnName = "CategoryName";
        categories.Columns.Add(column);

        // Set the primary key on CategoryID.
        DataColumn[] pKey = new DataColumn[1];
        pKey[0] = categories.Columns["CategoryID"];
        categories.PrimaryKey = pKey;

        // Fetch the data and fill the DataTable
        adapter.Fill(categories);

        // Add a new row.
        DataRow newRow = categories.NewRow();
        newRow["CategoryName"] = "New Category";
        categories.Rows.Add(newRow);

        // Add another new row.
        DataRow newRow2 = categories.NewRow();
        newRow2["CategoryName"] = "Another New Category";
        categories.Rows.Add(newRow2);

        // Add changed rows to a new DataTable that will be
        // used to post the inserts to the database.
        DataTable dataChanges = categories.GetChanges();

        // Include an event to fill in the Autonumber value.
        adapter.RowUpdated +=
            new OleDbRowUpdatedEventHandler(OnRowUpdated);

        // Update the database, inserting the new rows.
        adapter.Update(dataChanges);

        Console.WriteLine("Rows before merge:");
        foreach (DataRow row in categories.Rows)
        {
            {
                Console.WriteLine("  {0}: {1}", row[0], row[1]);
            }
        }

        // Merge the two DataTables.
        categories.Merge(dataChanges);

        // Commit the changes.
        categories.AcceptChanges();

        Console.WriteLine("Rows after merge:");
        foreach (DataRow row in categories.Rows)
        {
            {
                Console.WriteLine("  {0}: {1}", row[0], row[1]);
            }
        }
    }
}
```

```
Shared connection As OleDbConnection = Nothing

Private Shared Sub MergeIdentityColumns(ByVal connection As OleDbConnection)
    Using connection

        ' Create a DataAdapter based on a SELECT query.
        Dim adapter As OleDbDataAdapter = New OleDbDataAdapter( _
          "SELECT CategoryID, CategoryName FROM Categories", _
          connection)

        ' Create the INSERT command for the new category.
        adapter.InsertCommand = New OleDbCommand( _
          "INSERT INTO Categories (CategoryName) Values(?)", connection)
        adapter.InsertCommand.CommandType = CommandType.Text

        ' Add the parameter for the CategoryName.
        adapter.InsertCommand.Parameters.Add( _
          "@CategoryName", OleDbType.VarWChar, 15, "CategoryName")
        adapter.InsertCommand.UpdatedRowSource = UpdateRowSource.Both

        ' Create a DataTable.
        Dim categories As DataTable = New DataTable

        ' Create the CategoryID column and set its auto
        ' incrementing properties to decrement from zero.
        Dim column As New DataColumn()
        column.DataType = System.Type.GetType("System.Int32")
        column.ColumnName = "CategoryID"
        column.AutoIncrement = True
        column.AutoIncrementSeed = 0
        column.AutoIncrementStep = -1
        categories.Columns.Add(column)

        ' Create the CategoryName column.
        column = New DataColumn()
        column.DataType = System.Type.GetType("System.String")
        column.ColumnName = "CategoryName"
        categories.Columns.Add(column)

        ' Set the primary key on CategoryID.
        Dim pKey(0) As DataColumn
        pKey(0) = categories.Columns("CategoryID")
        categories.PrimaryKey = pKey

        ' Fetch the data and fill the DataTable.
        adapter.Fill(categories)

        ' Add a new row.
        Dim newRow As DataRow = categories.NewRow()
        newRow("CategoryName") = "New Category"
        categories.Rows.Add(newRow)

        ' Add another new row.
        Dim newRow2 As DataRow = categories.NewRow()
        newRow2("CategoryName") = "Another New Category"
        categories.Rows.Add(newRow2)

        ' Add changed rows to a new DataTable that will be
        ' used to post the inserts to the database.
        Dim dataChanges As DataTable = categories.GetChanges()

        ' Include an event to fill in the Autonumber value.
        AddHandler adapter.RowUpdated, _
          New OleDbRowUpdatedEventHandler(AddressOf OnRowUpdated)

        ' Update the database, inserting the new rows.
        adapter.Update(dataChanges)
```

```
        Console.WriteLine("Rows before merge:")
        Dim row1 As DataRow
        For Each row1 In categories.Rows
            Console.WriteLine("  {0}: {1}", row1(0), row1(1))
        Next

        ' Merge the two DataTables.
        categories.Merge(dataChanges)

        ' Commit the changes.
        categories.AcceptChanges()

        Console.WriteLine("Rows after merge:")
        Dim row As DataRow
        For Each row In categories.Rows
            Console.WriteLine("  {0}: {1}", row(0), row(1))
        Next
    End Using
End Sub
```

The `RowUpdated` event handler uses the same open OleDbConnection as the `Update` statement of the `OleDbDataAdapter`. It checks the `StatementType` of the OleDbRowUpdatedEventArgs for inserted rows. For each inserted row a new OleDbCommand is created to execute the SELECT @@IDENTITY statement on the connection, returning the new `Autonumber` value, which is placed in the **CategoryID** column of the `DataRow`. The `Status` property is then set to `UpdateStatus.SkipCurrentRow` to suppress the hidden call to `AcceptChanges`. In the main body of the procedure, the `Merge` method is called to merge the two `DataTable` objects, and finally `AcceptChanges` is called.

```
private static void OnRowUpdated(
  object sender, OleDbRowUpdatedEventArgs e)
{
    // Conditionally execute this code block on inserts only.
    if (e.StatementType == StatementType.Insert)
    {
        OleDbCommand cmdNewID = new OleDbCommand("SELECT @@IDENTITY",
            connection);
        // Retrieve the Autonumber and store it in the CategoryID column.
        e.Row["CategoryID"] = (int)cmdNewID.ExecuteScalar();
        e.Status = UpdateStatus.SkipCurrentRow;
    }
}
```

```
Private Shared Sub OnRowUpdated( _
    ByVal sender As Object, ByVal e As OleDbRowUpdatedEventArgs)
    ' Conditionally execute this code block on inserts only.
    If e.StatementType = StatementType.Insert Then
        ' Retrieve the Autonumber and store it in the CategoryID column.
        Dim cmdNewID As New OleDbCommand("SELECT @@IDENTITY", _
           connection)
        e.Row("CategoryID") = CInt(cmdNewID.ExecuteScalar)
        e.Status = UpdateStatus.SkipCurrentRow
    End If
End Sub
```

Retrieving Identity Values

We often set the column as identity when the values in the column must be unique. And sometimes we need the identity value of new data. This sample demonstrates how to retrieve identity values:

- Creates a stored procedure to insert data and return an identity value.
- Executes a command to insert the new data and display the result.

- Uses SqlDataAdapter to insert new data and display the result.

Before you compile and run the sample, you must create the sample database, using the following script:

```
USE [master]
GO

CREATE DATABASE [MySchool]
GO

USE [MySchool]
GO

SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE procedure [dbo].[CourseExtInfo] @CourseId int
as
select c.CourseID,c.Title,c.Credits,d.Name as DepartmentName
from Course as c left outer join Department as d on c.DepartmentID=d.DepartmentID
where c.CourseID=@CourseId

GO

SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
create procedure [dbo].[DepartmentInfo] @DepartmentId int,@CourseCount int output
as
select @CourseCount=Count(c.CourseID)
from course as c
where c.DepartmentID=@DepartmentId

select d.DepartmentID,d.Name,d.Budget,d.StartDate,d.Administrator
from Department as d
where d.DepartmentID=@DepartmentId

GO

SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
Create PROCEDURE [dbo].[GetDepartmentsOfSpecifiedYear]
@Year int,@BudgetSum money output
AS
BEGIN
        SELECT @BudgetSum=SUM([Budget])
  FROM [MySchool].[dbo].[Department]
  Where YEAR([StartDate])=@Year

SELECT [DepartmentID]
      ,[Name]
      ,[Budget]
      ,[StartDate]
      ,[Administrator]
  FROM [MySchool].[dbo].[Department]
  Where YEAR([StartDate])=@Year

END
GO

SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
```

```
GO
CREATE PROCEDURE [dbo].[GradeOfStudent]
-- Add the parameters for the stored procedure here
@CourseTitle nvarchar(100),@FirstName nvarchar(50),
@LastName nvarchar(50),@Grade decimal(3,2) output
AS
BEGIN
select @Grade=Max(Grade)
from [dbo].[StudentGrade] as s join [dbo].[Course] as c on
s.CourseID=c.CourseID join [dbo].[Person] as p on s.StudentID=p.PersonID
where c.Title=@CourseTitle and p.FirstName=@FirstName
and p.LastName= @LastName
END
GO

SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE PROCEDURE [dbo].[InsertPerson]
-- Add the parameters for the stored procedure here
@FirstName nvarchar(50),@LastName nvarchar(50),
@PersonID int output
AS
BEGIN
    insert [dbo].[Person](LastName,FirstName) Values(@LastName,@FirstName)

    set @PersonID=SCOPE_IDENTITY()
END
Go

SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[Course]([CourseID] [nvarchar](10) NOT NULL,
[Year] [smallint] NOT NULL,
[Title] [nvarchar](100) NOT NULL,
[Credits] [int] NOT NULL,
[DepartmentID] [int] NOT NULL,
 CONSTRAINT [PK_Course] PRIMARY KEY CLUSTERED
(
[CourseID] ASC,
[Year] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS = ON,
ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]) ON [PRIMARY]

GO

SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[Department]([DepartmentID] [int] IDENTITY(1,1) NOT NULL,
[Name] [nvarchar](50) NOT NULL,
[Budget] [money] NOT NULL,
[StartDate] [datetime] NOT NULL,
[Administrator] [int] NULL,
 CONSTRAINT [PK_Department] PRIMARY KEY CLUSTERED
(
[DepartmentID] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS = ON,
ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]) ON [PRIMARY]

GO

SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
```

```
GO
SET ANSI_PADDING ON
GO
CREATE TABLE [dbo].[Person]([PersonID] [int] IDENTITY(1,1) NOT NULL,
[LastName] [nvarchar](50) NOT NULL,
[FirstName] [nvarchar](50) NOT NULL,
[HireDate] [datetime] NULL,
[EnrollmentDate] [datetime] NULL,
[Picture] [varbinary](max) NULL,
 CONSTRAINT [PK_School.Student] PRIMARY KEY CLUSTERED
(
[PersonID] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS = ON,
ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]) ON [PRIMARY] TEXTIMAGE_ON [PRIMARY]

GO

SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[StudentGrade]([EnrollmentID] [int] IDENTITY(1,1) NOT NULL,
[CourseID] [nvarchar](10) NOT NULL,
[StudentID] [int] NOT NULL,
[Grade] [decimal](3, 2) NOT NULL,
 CONSTRAINT [PK_StudentGrade] PRIMARY KEY CLUSTERED
(
[EnrollmentID] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS = ON,
ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]) ON [PRIMARY]

GO

SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
create view [dbo].[EnglishCourse]
as
select c.CourseID,c.Title,c.Credits,c.DepartmentID
from Course as c join Department as d on c.DepartmentID=d.DepartmentID
where d.Name=N'English'

GO
INSERT [dbo].[Course] ([CourseID], [Year], [Title], [Credits], [DepartmentID]) VALUES (N'C1045', 2012,
N'Calculus', 4, 7)
INSERT [dbo].[Course] ([CourseID], [Year], [Title], [Credits], [DepartmentID]) VALUES (N'C1061', 2012,
N'Physics', 4, 1)
INSERT [dbo].[Course] ([CourseID], [Year], [Title], [Credits], [DepartmentID]) VALUES (N'C2021', 2012,
N'Composition', 3, 2)
INSERT [dbo].[Course] ([CourseID], [Year], [Title], [Credits], [DepartmentID]) VALUES (N'C2042', 2012,
N'Literature', 4, 2)
SET IDENTITY_INSERT [dbo].[Department] ON

INSERT [dbo].[Department] ([DepartmentID], [Name], [Budget], [StartDate], [Administrator]) VALUES (1,
N'Engineering', 350000.0000, CAST(0x0000999C00000000 AS DateTime), 2)
INSERT [dbo].[Department] ([DepartmentID], [Name], [Budget], [StartDate], [Administrator]) VALUES (2,
N'English', 120000.0000, CAST(0x0000999C00000000 AS DateTime), 6)
INSERT [dbo].[Department] ([DepartmentID], [Name], [Budget], [StartDate], [Administrator]) VALUES (4,
N'Economics', 200000.0000, CAST(0x0000999C00000000 AS DateTime), 4)
INSERT [dbo].[Department] ([DepartmentID], [Name], [Budget], [StartDate], [Administrator]) VALUES (7,
N'Mathematics', 250024.0000, CAST(0x0000999C00000000 AS DateTime), 3)
SET IDENTITY_INSERT [dbo].[Department] OFF
SET IDENTITY_INSERT [dbo].[Person] ON

INSERT [dbo].[Person] ([PersonID], [LastName], [FirstName], [HireDate], [EnrollmentDate]) VALUES (1, N'Hu',
N'Nan', NULL, CAST(0x0000A0BF00000000 AS DateTime))
INSERT [dbo].[Person] ([PersonID], [LastName], [FirstName], [HireDate], [EnrollmentDate]) VALUES (2,
N'Norman', N'Laura', NULL, CAST(0x0000A0BF00000000 AS DateTime))
```

```
INSERT [dbo].[Person] ([PersonID], [LastName], [FirstName], [HireDate], [EnrollmentDate]) VALUES (3,
N'Olivotto', N'Nino', NULL, CAST(0x0000A0BF00000000 AS DateTime))
INSERT [dbo].[Person] ([PersonID], [LastName], [FirstName], [HireDate], [EnrollmentDate]) VALUES (4, N'Anand',
N'Arturo', NULL, CAST(0x0000A0BF00000000 AS DateTime))
INSERT [dbo].[Person] ([PersonID], [LastName], [FirstName], [HireDate], [EnrollmentDate]) VALUES (5, N'Jai',
N'Damien', NULL, CAST(0x0000A0BF00000000 AS DateTime))
INSERT [dbo].[Person] ([PersonID], [LastName], [FirstName], [HireDate], [EnrollmentDate]) VALUES (6, N'Holt',
N'Roger', CAST(0x000097F100000000 AS DateTime), NULL)
INSERT [dbo].[Person] ([PersonID], [LastName], [FirstName], [HireDate], [EnrollmentDate]) VALUES (7,
N'Martin', N'Randall', CAST(0x00008B1A00000000 AS DateTime), NULL)
SET IDENTITY_INSERT [dbo].[Person] OFF
SET IDENTITY_INSERT [dbo].[StudentGrade] ON

INSERT [dbo].[StudentGrade] ([EnrollmentID], [CourseID], [StudentID], [Grade]) VALUES (1, N'C1045', 1,
CAST(3.50 AS Decimal(3, 2)))
INSERT [dbo].[StudentGrade] ([EnrollmentID], [CourseID], [StudentID], [Grade]) VALUES (2, N'C1045', 2,
CAST(3.00 AS Decimal(3, 2)))
INSERT [dbo].[StudentGrade] ([EnrollmentID], [CourseID], [StudentID], [Grade]) VALUES (3, N'C1045', 3,
CAST(2.50 AS Decimal(3, 2)))
INSERT [dbo].[StudentGrade] ([EnrollmentID], [CourseID], [StudentID], [Grade]) VALUES (4, N'C1045', 4,
CAST(4.00 AS Decimal(3, 2)))
INSERT [dbo].[StudentGrade] ([EnrollmentID], [CourseID], [StudentID], [Grade]) VALUES (5, N'C1045', 5,
CAST(3.50 AS Decimal(3, 2)))
INSERT [dbo].[StudentGrade] ([EnrollmentID], [CourseID], [StudentID], [Grade]) VALUES (6, N'C1061', 1,
CAST(4.00 AS Decimal(3, 2)))
INSERT [dbo].[StudentGrade] ([EnrollmentID], [CourseID], [StudentID], [Grade]) VALUES (7, N'C1061', 3,
CAST(3.50 AS Decimal(3, 2)))
INSERT [dbo].[StudentGrade] ([EnrollmentID], [CourseID], [StudentID], [Grade]) VALUES (8, N'C1061', 4,
CAST(2.50 AS Decimal(3, 2)))
INSERT [dbo].[StudentGrade] ([EnrollmentID], [CourseID], [StudentID], [Grade]) VALUES (9, N'C1061', 5,
CAST(1.50 AS Decimal(3, 2)))
INSERT [dbo].[StudentGrade] ([EnrollmentID], [CourseID], [StudentID], [Grade]) VALUES (10, N'C2021', 1,
CAST(2.50 AS Decimal(3, 2)))
INSERT [dbo].[StudentGrade] ([EnrollmentID], [CourseID], [StudentID], [Grade]) VALUES (11, N'C2021', 2,
CAST(3.50 AS Decimal(3, 2)))
INSERT [dbo].[StudentGrade] ([EnrollmentID], [CourseID], [StudentID], [Grade]) VALUES (12, N'C2021', 4,
CAST(3.00 AS Decimal(3, 2)))
INSERT [dbo].[StudentGrade] ([EnrollmentID], [CourseID], [StudentID], [Grade]) VALUES (13, N'C2021', 5,
CAST(3.00 AS Decimal(3, 2)))
INSERT [dbo].[StudentGrade] ([EnrollmentID], [CourseID], [StudentID], [Grade]) VALUES (14, N'C2042', 1,
CAST(2.00 AS Decimal(3, 2)))
INSERT [dbo].[StudentGrade] ([EnrollmentID], [CourseID], [StudentID], [Grade]) VALUES (15, N'C2042', 2,
CAST(3.50 AS Decimal(3, 2)))
INSERT [dbo].[StudentGrade] ([EnrollmentID], [CourseID], [StudentID], [Grade]) VALUES (16, N'C2042', 3,
CAST(4.00 AS Decimal(3, 2)))
INSERT [dbo].[StudentGrade] ([EnrollmentID], [CourseID], [StudentID], [Grade]) VALUES (17, N'C2042', 5,
CAST(3.00 AS Decimal(3, 2)))
SET IDENTITY_INSERT [dbo].[StudentGrade] OFF
ALTER TABLE [dbo].[Course]  WITH CHECK ADD  CONSTRAINT [FK_Course_Department] FOREIGN KEY([DepartmentID])
REFERENCES [dbo].[Department] ([DepartmentID])
GO
ALTER TABLE [dbo].[Course] CHECK CONSTRAINT [FK_Course_Department]
GO
ALTER TABLE [dbo].[StudentGrade]  WITH CHECK ADD  CONSTRAINT [FK_StudentGrade_Student] FOREIGN
KEY([StudentID])
REFERENCES [dbo].[Person] ([PersonID])
GO
ALTER TABLE [dbo].[StudentGrade] CHECK CONSTRAINT [FK_StudentGrade_Student]
GO
```

The code listing follows:

TIP

The code listing refers to an Access database file called MySchool.mdb. You can download MySchool.mdb (as part of the full C# or Visual Basic sample project) from code.msdn.microsoft.com.

```
using System;
using System.Data;
using System.Data.OleDb;
using System.Data.SqlClient;

class Program {
   static void Main(string[] args) {
      String SqlDbConnectionString = "Data Source=(local);Initial Catalog=MySchool;Integrated
Security=True;Asynchronous Processing=true;";

      InsertPerson(SqlDbConnectionString, "Janice", "Galvin");
      Console.WriteLine();

      InsertPersonInAdapter(SqlDbConnectionString, "Peter", "Krebs");
      Console.WriteLine();

      String oledbConnectionString = "Provider=Microsoft.Jet.OLEDB.4.0; Data Source=Database\\MySchool.mdb";
      InsertPersonInJet4Database(oledbConnectionString, "Janice", "Galvin");
      Console.WriteLine();

      Console.WriteLine("Please press any key to exit.....");
      Console.ReadKey();
   }

   // Using stored procedure to insert a new row and retrieve the identity value
   static void InsertPerson(String connectionString, String firstName, String lastName) {
      String commandText = "dbo.InsertPerson";

      using (SqlConnection conn = new SqlConnection(connectionString)) {
         using (SqlCommand cmd = new SqlCommand(commandText, conn)) {
            cmd.CommandType = CommandType.StoredProcedure;

            cmd.Parameters.Add(new SqlParameter("@FirstName", firstName));
            cmd.Parameters.Add(new SqlParameter("@LastName", lastName));
            SqlParameter personId = new SqlParameter("@PersonID", SqlDbType.Int);
            personId.Direction = ParameterDirection.Output;
            cmd.Parameters.Add(personId);

            conn.Open();
            cmd.ExecuteNonQuery();

            Console.WriteLine("Person Id of new person:{0}", personId.Value);
         }
      }
   }

   // Using stored procedure in adapter to insert new rows and update the identity value.
   static void InsertPersonInAdapter(String connectionString, String firstName, String lastName) {
      String commandText = "dbo.InsertPerson";
      using (SqlConnection conn = new SqlConnection(connectionString)) {
         SqlDataAdapter mySchool = new SqlDataAdapter("Select PersonID,FirstName,LastName from [dbo].
[Person]", conn);

         mySchool.InsertCommand = new SqlCommand(commandText, conn);
         mySchool.InsertCommand.CommandType = CommandType.StoredProcedure;

         mySchool.InsertCommand.Parameters.Add(
             new SqlParameter("@FirstName", SqlDbType.NVarChar, 50, "FirstName"));
         mySchool.InsertCommand.Parameters.Add(
             new SqlParameter("@LastName", SqlDbType.NVarChar, 50, "LastName"));

         SqlParameter personId = mySchool.InsertCommand.Parameters.Add(new SqlParameter("@PersonID",
SqlDbType.Int, 0, "PersonID"));
         personId.Direction = ParameterDirection.Output;

         DataTable persons = new DataTable();
         mySchool.Fill(persons);
```

```
         DataRow newPerson = persons.NewRow();
         newPerson["FirstName"] = firstName;
         newPerson["LastName"] = lastName;
         persons.Rows.Add(newPerson);

         mySchool.Update(persons);
         Console.WriteLine("Show all persons:");
         ShowDataTable(persons, 14);
      }
   }

   /// For a Jet 4.0 database, we need use the single statement and event handler to insert new rows and
retrieve the identity value.
   static void InsertPersonInJet4Database(String connectionString, String firstName, String lastName) {
      String commandText = "Insert into Person(FirstName,LastName) Values(?,?)";
      using (OleDbConnection conn = new OleDbConnection(connectionString)) {
         OleDbDataAdapter mySchool = new OleDbDataAdapter("Select PersonID,FirstName,LastName from Person",
conn);

         // Create Insert Command
         mySchool.InsertCommand = new OleDbCommand(commandText, conn);
         mySchool.InsertCommand.CommandType = CommandType.Text;

         mySchool.InsertCommand.Parameters.Add(new OleDbParameter("@FirstName", OleDbType.VarChar, 50,
"FirstName"));
         mySchool.InsertCommand.Parameters.Add(new OleDbParameter("@LastName", OleDbType.VarChar, 50,
"LastName"));
         mySchool.InsertCommand.UpdatedRowSource = UpdateRowSource.Both;

         DataTable persons = CreatePersonsTable();

         mySchool.Fill(persons);

         DataRow newPerson = persons.NewRow();
         newPerson["FirstName"] = firstName;
         newPerson["LastName"] = lastName;
         persons.Rows.Add(newPerson);

         DataTable dataChanges = persons.GetChanges();

         mySchool.RowUpdated += OnRowUpdated;

         mySchool.Update(dataChanges);

         Console.WriteLine("Data before merging:");
         ShowDataTable(persons, 14);
         Console.WriteLine();

         persons.Merge(dataChanges);
         persons.AcceptChanges();

         Console.WriteLine("Data after merging");
         ShowDataTable(persons, 14);
      }
   }

   static void OnRowUpdated(object sender, OleDbRowUpdatedEventArgs e) {
      if (e.StatementType == StatementType.Insert) {
         // Retrieve the identity value
         OleDbCommand cmdNewId = new OleDbCommand("Select @@IDENTITY", e.Command.Connection);
         e.Row["PersonID"] = (Int32)cmdNewId.ExecuteScalar();

         // After the status is changed, the original values in the row are preserved. And the
         // Merge method will be called to merge the new identity value into the original DataTable.
         e.Status = UpdateStatus.SkipCurrentRow;
      }
   }

   // Create the Persons table before filling.
```

```
    // Create the Persons table before filling.
    private static DataTable CreatePersonsTable() {
       DataTable persons = new DataTable();

       DataColumn personId = new DataColumn();
       personId.DataType = Type.GetType("System.Int32");
       personId.ColumnName = "PersonID";
       personId.AutoIncrement = true;
       personId.AutoIncrementSeed = 0;
       personId.AutoIncrementStep = -1;
       persons.Columns.Add(personId);

       DataColumn firstName = new DataColumn();
       firstName.DataType = Type.GetType("System.String");
       firstName.ColumnName = "FirstName";
       persons.Columns.Add(firstName);

       DataColumn lastName = new DataColumn();
       lastName.DataType = Type.GetType("System.String");
       lastName.ColumnName = "LastName";
       persons.Columns.Add(lastName);

       DataColumn[] pkey = { personId };
       persons.PrimaryKey = pkey;

       return persons;
    }

    private static void ShowDataTable(DataTable table, Int32 length) {
       foreach (DataColumn col in table.Columns) {
          Console.Write("{0,-" + length + "}", col.ColumnName);
       }
       Console.WriteLine();

       foreach (DataRow row in table.Rows) {
          foreach (DataColumn col in table.Columns) {
             if (col.DataType.Equals(typeof(DateTime)))
                Console.Write("{0,-" + length + ":d}", row[col]);
             else if (col.DataType.Equals(typeof(Decimal)))
                Console.Write("{0,-" + length + ":C}", row[col]);
             else
                Console.Write("{0,-" + length + "}", row[col]);
          }

          Console.WriteLine();
       }
    }
}
```

See also

- Retrieving and Modifying Data in ADO.NET
- DataAdapters and DataReaders
- Row States and Row Versions
- AcceptChanges and RejectChanges
- Merging DataSet Contents
- Updating Data Sources with DataAdapters
- ADO.NET Overview

Retrieving Binary Data

3/12/2020 • 5 minutes to read • Edit Online

By default, the **DataReader** loads incoming data as a row as soon as an entire row of data is available. Binary large objects (BLOBs) need different treatment, however, because they can contain gigabytes of data that cannot be contained in a single row. The **Command.ExecuteReader** method has an overload that will take a CommandBehavior argument to modify the default behavior of the **DataReader**. You can pass SequentialAccess to the **ExecuteReader** method to modify the default behavior of the **DataReader** so that instead of loading rows of data, it will load data sequentially as it is received. This is ideal for loading BLOBs or other large data structures. Note that this behavior may depend on your data source. For example, returning a BLOB from Microsoft Access will load the entire BLOB being loaded into memory, rather than sequentially as it is received.

When setting the **DataReader** to use **SequentialAccess**, it is important to note the sequence in which you access the fields returned. The default behavior of the **DataReader**, which loads an entire row as soon as it is available, allows you to access the fields returned in any order until the next row is read. When using **SequentialAccess** however, you must access the fields returned by the **DataReader** in order. For example, if your query returns three columns, the third of which is a BLOB, you must return the values of the first and second fields before accessing the BLOB data in the third field. If you access the third field before the first or second fields, the first and second field values are no longer available. This is because **SequentialAccess** has modified the **DataReader** to return data in sequence and the data is not available after the **DataReader** has read past it.

When accessing the data in the BLOB field, use the **GetBytes** or **GetChars** typed accessors of the **DataReader**, which fill an array with data. You can also use **GetString** for character data; however. to conserve system resources you might not want to load an entire BLOB value into a single string variable. You can instead specify a specific buffer size of data to be returned, and a starting location for the first byte or character to be read from the returned data. **GetBytes** and **GetChars** will return a `long` value, which represents the number of bytes or characters returned. If you pass a null array to **GetBytes** or **GetChars**, the long value returned will be the total number of bytes or characters in the BLOB. You can optionally specify an index in the array as a starting position for the data being read.

Example

The following example returns the publisher ID and logo from the **pubs** sample database in Microsoft SQL Server. The publisher ID (`pub_id`) is a character field, and the logo is an image, which is a BLOB. Because the **logo** field is a bitmap, the example returns binary data using **GetBytes**. Notice that the publisher ID is accessed for the current row of data before the logo, because the fields must be accessed sequentially.

```
' Assumes that connection is a valid SqlConnection object.
Dim command As SqlCommand = New SqlCommand( _
  "SELECT pub_id, logo FROM pub_info", connection)

' Writes the BLOB to a file (*.bmp).
Dim stream As FileStream
' Streams the binary data to the FileStream object.
Dim writer As BinaryWriter
' The size of the BLOB buffer.
Dim bufferSize As Integer = 100
' The BLOB byte() buffer to be filled by GetBytes.
Dim outByte(bufferSize - 1) As Byte
' The bytes returned from GetBytes.
Dim retval As Long
' The starting position in the BLOB output.
Dim startIndex As Long = 0

' The publisher id to use in the file name.
Dim pubID As String = ""

' Open the connection and read data into the DataReader.
connection.Open()
Dim reader As SqlDataReader = command.ExecuteReader(CommandBehavior.SequentialAccess)

Do While reader.Read()
  ' Get the publisher id, which must occur before getting the logo.
  pubID = reader.GetString(0)

  ' Create a file to hold the output.
  stream = New FileStream( _
    "logo" & pubID & ".bmp", FileMode.OpenOrCreate, FileAccess.Write)
  writer = New BinaryWriter(stream)

  ' Reset the starting byte for a new BLOB.
  startIndex = 0

  ' Read bytes into outByte() and retain the number of bytes returned.
  retval = reader.GetBytes(1, startIndex, outByte, 0, bufferSize)

  ' Continue while there are bytes beyond the size of the buffer.
  Do While retval = bufferSize
    writer.Write(outByte)
    writer.Flush()

    ' Reposition start index to end of the last buffer and fill buffer.
    startIndex += bufferSize
    retval = reader.GetBytes(1, startIndex, outByte, 0, bufferSize)
  Loop

  ' Write the remaining buffer.
  writer.Write(outByte, 0 , retval - 1)
  writer.Flush()

  ' Close the output file.
  writer.Close()
  stream.Close()
Loop

' Close the reader and the connection.
reader.Close()
connection.Close()
```

```
// Assumes that connection is a valid SqlConnection object.
SqlCommand command = new SqlCommand(
  "SELECT pub_id, logo FROM pub_info", connection);

// Writes the BLOB to a file (*.bmp).
FileStream stream;
// Streams the BLOB to the FileStream object.
BinaryWriter writer;

// Size of the BLOB buffer.
int bufferSize = 100;
// The BLOB byte[] buffer to be filled by GetBytes.
byte[] outByte = new byte[bufferSize];
// The bytes returned from GetBytes.
long retval;
// The starting position in the BLOB output.
long startIndex = 0;

// The publisher id to use in the file name.
string pubID = "";

// Open the connection and read data into the DataReader.
connection.Open();
SqlDataReader reader = command.ExecuteReader(CommandBehavior.SequentialAccess);

while (reader.Read())
{
  // Get the publisher id, which must occur before getting the logo.
  pubID = reader.GetString(0);

  // Create a file to hold the output.
  stream = new FileStream(
    "logo" + pubID + ".bmp", FileMode.OpenOrCreate, FileAccess.Write);
  writer = new BinaryWriter(stream);

  // Reset the starting byte for the new BLOB.
  startIndex = 0;

  // Read bytes into outByte[] and retain the number of bytes returned.
  retval = reader.GetBytes(1, startIndex, outByte, 0, bufferSize);

  // Continue while there are bytes beyond the size of the buffer.
  while (retval == bufferSize)
  {
    writer.Write(outByte);
    writer.Flush();

    // Reposition start index to end of last buffer and fill buffer.
    startIndex += bufferSize;
    retval = reader.GetBytes(1, startIndex, outByte, 0, bufferSize);
  }

  // Write the remaining buffer.
  writer.Write(outByte, 0, (int)retval);
  writer.Flush();

  // Close the output file.
  writer.Close();
  stream.Close();
}

// Close the reader and the connection.
reader.Close();
connection.Close();
```

See also

- SQL Server Binary and Large-Value Data
- ADO.NET Overview

Modifying Data with Stored Procedures

9/7/2019 • 3 minutes to read • Edit Online

Stored procedures can accept data as input parameters and can return data as output parameters, result sets, or return values. The sample below illustrates how ADO.NET sends and receives input parameters, output parameters, and return values. The example inserts a new record into a table where the primary key column is an identity column in a SQL Server database.

> **NOTE**
>
> If you are using SQL Server stored procedures to edit or delete data using a SqlDataAdapter, make sure that you do not use SET NOCOUNT ON in the stored procedure definition. This causes the rows affected count returned to be zero, which the `DataAdapter` interprets as a concurrency conflict. In this event, a DBConcurrencyException will be thrown.

Example

The sample uses the following stored procedure to insert a new category into the **Northwind Categories** table. The stored procedure takes the value in the **CategoryName** column as an input parameter and uses the SCOPE_IDENTITY() function to retrieve the new value of the identity field, **CategoryID**, and return it in an output parameter. The RETURN statement uses the @@ROWCOUNT function to return the number of rows inserted.

```
CREATE PROCEDURE dbo.InsertCategory
  @CategoryName nvarchar(15),
  @Identity int OUT
AS
INSERT INTO Categories (CategoryName) VALUES(@CategoryName)
SET @Identity = SCOPE_IDENTITY()
RETURN @@ROWCOUNT
```

The following code example uses the `InsertCategory` stored procedure shown above as the source for the InsertCommand of the SqlDataAdapter. The `@Identity` output parameter will be reflected in the DataSet after the record has been inserted into the database when the `Update` method of the SqlDataAdapter is called. The code also retrieves the return value.

> **NOTE**
>
> When using the OleDbDataAdapter, you must specify parameters with a ParameterDirection of **ReturnValue** before the other parameters.

```
using System;
using System.Data;
using System.Data.SqlClient;

class Program
{
    static void Main()
    {
        string connectionString = GetConnectionString();
        ReturnIdentity(connectionString);
        // Console.ReadLine();
    }
```

```
    private static void ReturnIdentity(string connectionString)
    {
        using (SqlConnection connection = new SqlConnection(connectionString))
        {
            // Create a SqlDataAdapter based on a SELECT query.
            SqlDataAdapter adapter = new SqlDataAdapter("SELECT CategoryID, CategoryName FROM dbo.Categories",
connection);

            // Create a SqlCommand to execute the stored procedure.
            adapter.InsertCommand = new SqlCommand("InsertCategory", connection);
            adapter.InsertCommand.CommandType = CommandType.StoredProcedure;

            // Create a parameter for the ReturnValue.
            SqlParameter parameter = adapter.InsertCommand.Parameters.Add("@RowCount", SqlDbType.Int);
            parameter.Direction = ParameterDirection.ReturnValue;

            // Create an input parameter for the CategoryName.
            // You do not need to specify direction for input parameters.
            adapter.InsertCommand.Parameters.Add("@CategoryName", SqlDbType.NChar, 15, "CategoryName");

            // Create an output parameter for the new identity value.
            parameter = adapter.InsertCommand.Parameters.Add("@Identity", SqlDbType.Int, 0, "CategoryID");
            parameter.Direction = ParameterDirection.Output;

            // Create a DataTable and fill it.
            DataTable categories = new DataTable();
            adapter.Fill(categories);

            // Add a new row.
            DataRow categoryRow = categories.NewRow();
            categoryRow["CategoryName"] = "New Beverages";
            categories.Rows.Add(categoryRow);

            // Update the database.
            adapter.Update(categories);

            // Retrieve the ReturnValue.
            Int32 rowCount = (Int32)adapter.InsertCommand.Parameters["@RowCount"].Value;

            Console.WriteLine("ReturnValue: {0}", rowCount.ToString());
            Console.WriteLine("All Rows:");
            foreach (DataRow row in categories.Rows)
            {
                    Console.WriteLine("  {0}: {1}", row[0], row[1]);
            }
        }
    }

    static private string GetConnectionString()
    {
        // To avoid storing the connection string in your code,
        // you can retrieve it from a configuration file.
        return "Data Source=(local);Initial Catalog=Northwind;Integrated Security=true";
    }
}
```

```
Option Explicit On
Option Strict On

Imports System.Data
Imports System.Data.SqlClient

Module Class1

    Sub Main()
        Dim connectionString As String = _
            GetConnectionString()
```

```
        ReturnIdentity(connectionString)
        ' Console.ReadLine()
    End Sub


    Private Sub ReturnIdentity(ByVal connectionString As String)
        Using connection As SqlConnection = New SqlConnection( _
           connectionString)

            ' Create a SqlDataAdapter based on a SELECT query.
            Dim adapter As SqlDataAdapter = New SqlDataAdapter( _
               "SELECT CategoryID, CategoryName FROM dbo.Categories", _
               connection)

            ' Create a SqlCommand to execute the stored procedure.
            adapter.InsertCommand = New SqlCommand("dbo.InsertCategory", _
               connection)
            adapter.InsertCommand.CommandType = CommandType.StoredProcedure

            ' Create a parameter for the ReturnValue.
            Dim parameter As SqlParameter = _
               adapter.InsertCommand.Parameters.Add( _
              "@RowCount", SqlDbType.Int)
            parameter.Direction = ParameterDirection.ReturnValue

            ' Create an input parameter for the CategoryName.
            ' You do not need to specify direction for input parameters.
            adapter.InsertCommand.Parameters.Add( _
              "@CategoryName", SqlDbType.NChar, 15, "CategoryName")

            ' Create an output parameter for the new identity value.
            parameter = adapter.InsertCommand.Parameters.Add( _
              "@Identity", SqlDbType.Int, 0, "CategoryID")
            parameter.Direction = ParameterDirection.Output

            ' Create a DataTable and fill it.
            Dim categories As DataTable = New DataTable
            adapter.Fill(categories)

            ' Add a new row.
            Dim newRow As DataRow = categories.NewRow()
            newRow("CategoryName") = "New Category"
            categories.Rows.Add(newRow)

            ' Update the database.
            adapter.Update(categories)

            ' Retrieve the ReturnValue.
            Dim rowCount As Int32 = _
               CInt(adapter.InsertCommand.Parameters("@RowCount").Value)

            Console.WriteLine("ReturnValue: {0}", rowCount.ToString())
            Console.WriteLine("All Rows:")
            Dim row As DataRow
            For Each row In categories.Rows
                Console.WriteLine("  {0}: {1}", row(0), row(1))
            Next
        End Using
    End Sub

    Private Function GetConnectionString() As String
        ' To avoid storing the connection string in your code,
        ' you can retrieve it from a configuration file.
        Return "Data Source=(local);Initial Catalog=Northwind;" _
           & "Integrated Security=true;"
    End Function

End Module
```

See also

- Retrieving and Modifying Data in ADO.NET
- DataAdapters and DataReaders
- Executing a Command
- ADO.NET Overview

Retrieving Database Schema Information

9/7/2019 • 2 minutes to read • Edit Online

Obtaining schema information from a database is accomplished with the process of schema discovery. Schema discovery allows applications to request that managed providers find and return information about the database schema, also known as *metadata*, of a given database. Different database schema elements such as tables, columns, and stored-procedures are exposed through schema collections. Each schema collection contains a variety of schema information specific to the provider being used.

Each of the .NET Framework managed providers implement the **GetSchema** method in the **Connection** class, and the schema information that is returned from the **GetSchema** method comes in the form of a DataTable. The **GetSchema** method is an overloaded method that provides optional parameters for specifying the schema collection to return, and restricting the amount of information returned.

The .NET Framework Data Providers for OLE DB, ODBC, Oracle, and SqlClient provide a **GetSchemaTable** method that returns a DataTable describing the column metadata of the **DataReader**.

The .NET Framework Data Provider for OLE DB also exposes schema information by using the GetOleDbSchemaTable method of the OleDbConnection object. As arguments, **GetOleDbSchemaTable** takes an OleDbSchemaGuid that identifies the schema information to return, and an array of restrictions on those returned columns. **GetOleDbSchemaTable** returns a DataTable populated with the requested schema information.

In This Section

GetSchema and Schema Collections
Describes the **GetSchema** method and how it can be used to retrieve and restrict schema information from a database.

Schema Restrictions
Describes schema restrictions that can be used with **GetSchema**.

Common Schema Collections
Describes all of the common schema collections supported by all of the .NET Framework managed providers.

SQL Server Schema Collections
Describes the schema collection supported by the .NET Framework provider for SQL Server.

Oracle Schema Collections
Describes the schema collection supported by the .NET Framework provider for Oracle.

ODBC Schema Collections
Describes the schema collections for ODBC drivers.

OLE DB Schema Collections
Describes the schema collections for OLE DB providers.

Reference

GetSchema
Describes the **GetSchema** method of the DbConnection class.

GetSchema
Describes the **GetSchema** method of the OdbcConnection class.

GetSchema

Describes the **GetSchema** method of the OleDbConnection class.

GetSchema

Describes the **GetSchema** method of the OracleConnection class.

GetSchema

Describes the **GetSchema** method of the SqlConnection class.

GetSchemaTable

Describes the **GetSchemaTable** method of the DbDataReader class.

GetSchemaTable

Describes the **GetSchemaTable** method of the OdbcDataReader class.

GetSchemaTable

Describes the **GetSchemaTable** method of the OleDbDataReader class.

GetSchemaTable

Describes the **GetSchemaTable** method of the OracleDataReader class.

GetSchemaTable

Describes the **GetSchemaTable** method of the SqlDataReader class.

See also

- Retrieving and Modifying Data in ADO.NET
- ADO.NET Overview

GetSchema and Schema Collections

3/12/2020 • 2 minutes to read • Edit Online

The **Connection** classes in each of the .NET Framework managed providers implement a **GetSchema** method which is used to retrieve schema information about the database that is currently connected, and the schema information returned from the **GetSchema** method comes in the form of a DataTable. The **GetSchema** method is an overloaded method that provides optional parameters for specifying the schema collection to return, and restricting the amount of information returned.

Specifying the Schema Collections

The first optional parameter of the **GetSchema** method is the collection name which is specified as a string. There are two types of schema collections: common schema collections that are common to all providers, and specific schema collections which are specific to each provider.

You can query a .NET Framework managed provider to determine the list of supported schema collections by calling the **GetSchema** method with no arguments, or with the schema collection name "MetaDataCollections". This will return a DataTable with a list of the supported schema collections, the number of restrictions that they each support, and the number of identifier parts that they use.

Retrieving Schema Collections Example

The following examples demonstrate how to use the GetSchema method of the .NET Framework Data Provider for the SQL Server SqlConnection class to retrieve schema information about all of the tables contained in the **AdventureWorks** sample database:

```
Imports System.Data.SqlClient

Module Module1
   Sub Main()
      Dim connectionString As String = GetConnectionString()
      Using connection As New SqlConnection(connectionString)
         'Connect to the database then retrieve the schema information.
         connection.Open()
         Dim table As DataTable = connection.GetSchema("Tables")

         ' Display the contents of the table.
         DisplayData(table)
         Console.WriteLine("Press any key to continue.")
         Console.ReadKey()
      End Using
   End Sub

   Private Function GetConnectionString() As String
      ' To avoid storing the connection string in your code,
      ' you can retrieve it from a configuration file.
      Return "Data Source=(local);Database=AdventureWorks;" _
         & "Integrated Security=true;"
   End Function

   Private Sub DisplayData(ByVal table As DataTable)
      For Each row As DataRow In table.Rows
         For Each col As DataColumn In table.Columns
            Console.WriteLine("{0} = {1}", col.ColumnName, row(col))
         Next
         Console.WriteLine("============================")
      Next
   End Sub
End Module
```

```
using System;
using System.Data;
using System.Data.SqlClient;

class Program
{
  static void Main()
  {
  string connectionString = GetConnectionString();
  using (SqlConnection connection = new SqlConnection(connectionString))
  {
   // Connect to the database then retrieve the schema information.
   connection.Open();
   DataTable table = connection.GetSchema("Tables");

   // Display the contents of the table.
   DisplayData(table);
   Console.WriteLine("Press any key to continue.");
   Console.ReadKey();
   }
 }

  private static string GetConnectionString()
  {
   // To avoid storing the connection string in your code,
   // you can retrieve it from a configuration file.
   return "Data Source=(local);Database=AdventureWorks;" +
      "Integrated Security=true;";
  }

  private static void DisplayData(System.Data.DataTable table)
  {
     foreach (System.Data.DataRow row in table.Rows)
     {
        foreach (System.Data.DataColumn col in table.Columns)
        {
           Console.WriteLine("{0} = {1}", col.ColumnName, row[col]);
        }
     Console.WriteLine("============================");
     }
  }
}
```

See also

- Retrieving Database Schema Information
- ADO.NET Overview

Schema Restrictions

3/12/2020 • 4 minutes to read • Edit Online

The second optional parameter of the **GetSchema** method is the restrictions that are used to limit the amount of schema information returned, and it is passed to the **GetSchema** method as an array of strings. The position in the array determines the values that you can pass, and this is equivalent to the restriction number.

For example, the following table describes the restrictions supported by the "Tables" schema collection using the .NET Framework Data Provider for SQL Server. Additional restrictions for SQL Server schema collections are listed at the end of this topic.

RESTRICTION NAME	PARAMETER NAME	RESTRICTION DEFAULT	RESTRICTION NUMBER
Catalog	@Catalog	TABLE_CATALOG	1
Owner	@Owner	TABLE_SCHEMA	2
Table	@Name	TABLE_NAME	3
TableType	@TableType	TABLE_TYPE	4

Specifying Restriction Values

To use one of the restrictions of the "Tables" schema collection, simply create an array of strings with four elements, then place a value in the element that matches the restriction number. For example, to restrict the tables returned by the **GetSchema** method to only those tables in the "Sales" schema, set the second element of the array to "Sales" before passing it to the **GetSchema** method.

> **NOTE**
>
> The restrictions collections for `SqlClient` and `OracleClient` have an additional `ParameterName` column. The restriction default column is still there for backwards compatibility, but is currently ignored. Parameterized queries rather than string replacement should be used to minimize the risk of an SQL injection attack when specifying restriction values.

> **NOTE**
>
> The number of elements in the array must be less than or equal to the number of restrictions supported for the specified schema collection else an ArgumentException will be thrown. There can be fewer than the maximum number of restrictions. The missing restrictions are assumed to be null (unrestricted).

You can query a .NET Framework managed provider to determine the list of supported restrictions by calling the **GetSchema** method with the name of the restrictions schema collection, which is "Restrictions". This will return a DataTable with a list of the collection names, the restriction names, the default restriction values, and the restriction numbers.

Example

The following examples demonstrate how to use the GetSchema method of the .NET Framework Data Provider for the SQL Server SqlConnection class to retrieve schema information about all of the tables contained in the **AdventureWorks** sample database, and to restrict the information returned to only those tables in the "Sales"

schema:

```
Imports System.Data.SqlClient

Module Module1
Sub Main()
  Dim connectionString As String = _
    "Data Source=(local);Database=AdventureWorks;" & _
       "Integrated Security=true;";

  Dim restrictions(3) As String
  Using connection As New SqlConnection(connectionString)
    connection.Open()

    'Specify the restrictions.
    restrictions(1) = "Sales"
    Dim table As DataTable = connection.GetSchema("Tables", _
       restrictions)

    ' Display the contents of the table.
      For Each row As DataRow In table.Rows
         For Each col As DataColumn In table.Columns
            Console.WriteLine("{0} = {1}", col.ColumnName, row(col))
         Next
         Console.WriteLine("============================")
      Next
    Console.WriteLine("Press any key to continue.")
    Console.ReadKey()
  End Using
End Sub
End Module
```

```
using System;
using System.Data;
using System.Data.SqlClient;

class Program
{
  static void Main()
  {
    string connectionString =
       "Data Source=(local);Database=AdventureWorks;" +
       "Integrated Security=true;";
    using (SqlConnection connection =
       new SqlConnection(connectionString))
    {
        connection.Open();

        // Specify the restrictions.
        string[] restrictions = new string[4];
        restrictions[1] = "Sales";
        System.Data.DataTable table = connection.GetSchema(
          "Tables", restrictions);

        // Display the contents of the table.
        foreach (System.Data.DataRow row in table.Rows)
        {
            foreach (System.Data.DataColumn col in table.Columns)
            {
                Console.WriteLine("{0} = {1}",
                  col.ColumnName, row[col]);
            }
            Console.WriteLine("============================");
        }
        Console.WriteLine("Press any key to continue.");
        Console.ReadKey();
    }
  }

  private static string GetConnectionString()
  {
     // To avoid storing the connection string in your code,
     // you can retrieve it from a configuration file.
     return "Data Source=(local);Database=AdventureWorks;" +
        "Integrated Security=true;";
  }

  private static void DisplayData(System.Data.DataTable table)
  {
     foreach (System.Data.DataRow row in table.Rows)
     {
        foreach (System.Data.DataColumn col in table.Columns)
        {
           Console.WriteLine("{0} = {1}", col.ColumnName, row[col]);
        }
     Console.WriteLine("============================");
     }
  }
}
```

SQL Server Schema Restrictions

The following tables list the restrictions for SQL Server schema collections.

Users

RESTRICTION NAME	PARAMETER NAME	RESTRICTION DEFAULT	RESTRICTION NUMBER
User_Name	@Name	name	1

Databases

RESTRICTION NAME	PARAMETER NAME	RESTRICTION DEFAULT	RESTRICTION NUMBER
Name	@Name	Name	1

Tables

RESTRICTION NAME	PARAMETER NAME	RESTRICTION DEFAULT	RESTRICTION NUMBER
Catalog	@Catalog	TABLE_CATALOG	1
Owner	@Owner	TABLE_SCHEMA	2
Table	@Name	TABLE_NAME	3
TableType	@TableType	TABLE_TYPE	4

Columns

RESTRICTION NAME	PARAMETER NAME	RESTRICTION DEFAULT	RESTRICTION NUMBER
Catalog	@Catalog	TABLE_CATALOG	1
Owner	@Owner	TABLE_SCHEMA	2
Table	@Table	TABLE_NAME	3
Column	@Column	COLUMN_NAME	4

StructuredTypeMembers

RESTRICTION NAME	PARAMETER NAME	RESTRICTION DEFAULT	RESTRICTION NUMBER
Catalog	@Catalog	TABLE_CATALOG	1
Owner	@Owner	TABLE_SCHEMA	2
Table	@Table	TABLE_NAME	3
Column	@Column	COLUMN_NAME	4

Views

RESTRICTION NAME	PARAMETER NAME	RESTRICTION DEFAULT	RESTRICTION NUMBER
Catalog	@Catalog	TABLE_CATALOG	1
Owner	@Owner	TABLE_SCHEMA	2

RESTRICTION NAME	PARAMETER NAME	RESTRICTION DEFAULT	RESTRICTION NUMBER
Table	@Table	TABLE_NAME	3

ViewColumns

RESTRICTION NAME	PARAMETER NAME	RESTRICTION DEFAULT	RESTRICTION NUMBER
Catalog	@Catalog	VIEW_CATALOG	1
Owner	@Owner	VIEW_SCHEMA	2
Table	@Table	VIEW_NAME	3
Column	@Column	COLUMN_NAME	4

ProcedureParameters

RESTRICTION NAME	PARAMETER NAME	RESTRICTION DEFAULT	RESTRICTION NUMBER
Catalog	@Catalog	SPECIFIC_CATALOG	1
Owner	@Owner	SPECIFIC_SCHEMA	2
Name	@Name	SPECIFIC_NAME	3
Parameter	@Parameter	PARAMETER_NAME	4

Procedures

RESTRICTION NAME	PARAMETER NAME	RESTRICTION DEFAULT	RESTRICTION NUMBER
Catalog	@Catalog	SPECIFIC_CATALOG	1
Owner	@Owner	SPECIFIC_SCHEMA	2
Name	@Name	SPECIFIC_NAME	3
Type	@Type	ROUTINE_TYPE	4

IndexColumns

RESTRICTION NAME	PARAMETER NAME	RESTRICTION DEFAULT	RESTRICTION NUMBER
Catalog	@Catalog	db_name()	1
Owner	@Owner	user_name()	2
Table	@Table	o.name	3
ConstraintName	@ConstraintName	x.name	4
Column	@Column	c.name	5

Indexes

RESTRICTION NAME	PARAMETER NAME	RESTRICTION DEFAULT	RESTRICTION NUMBER
Catalog	@Catalog	db_name()	1
Owner	@Owner	user_name()	2
Table	@Table	o.name	3

UserDefinedTypes

RESTRICTION NAME	PARAMETER NAME	RESTRICTION DEFAULT	RESTRICTION NUMBER
assembly_name	@AssemblyName	assemblies.name	1
udt_name	@UDTName	types.assembly_class	2

ForeignKeys

RESTRICTION NAME	PARAMETER NAME	RESTRICTION DEFAULT	RESTRICTION NUMBER
Catalog	@Catalog	CONSTRAINT_CATALOG	1
Owner	@Owner	CONSTRAINT_SCHEMA	2
Table	@Table	TABLE_NAME	3
Name	@Name	CONSTRAINT_NAME	4

SQL Server 2008 Schema Restrictions

The following tables list the restrictions for SQL Server 2008 schema collections. These restrictions are valid beginning with version 3.5 SP1 of the .NET Framework and SQL Server 2008. They are not supported in earlier versions of the .NET Framework and SQL Server.

ColumnSetColumns

RESTRICTION NAME	PARAMETER NAME	RESTRICTION DEFAULT	RESTRICTION NUMBER
Catalog	@Catalog	TABLE_CATALOG	1
Owner	@Owner	TABLE_SCHEMA	2
Table	@Table	TABLE_NAME	3

AllColumns

RESTRICTION NAME	PARAMETER NAME	RESTRICTION DEFAULT	RESTRICTION NUMBER
Catalog	@Catalog	TABLE_CATALOG	1
Owner	@Owner	TABLE_SCHEMA	2

RESTRICTION NAME	PARAMETER NAME	RESTRICTION DEFAULT	RESTRICTION NUMBER
Table	@Table	TABLE_NAME	3
Column	@Column	COLUMN_NAME	4

See also

- ADO.NET Overview

Common Schema Collections

3/25/2020 • 10 minutes to read • Edit Online

The common schema collections are the schema collections that are implemented by each of the .NET Framework managed providers. You can query a .NET Framework managed provider to determine the list of supported schema collections by calling the **GetSchema** method with no arguments, or with the schema collection name "MetaDataCollections". This will return a DataTable with a list of the supported schema collections, the number of restrictions that they each support, and the number of identifier parts that they use. These collections describe all of the required columns. Providers are free to add additional columns if they wish. For example, `SqlClient` and `OracleClient` add ParameterName to the restrictions collection.

If a provider is unable to determine the value of a required column, it will return null.

For more information about using the **GetSchema** methods, see GetSchema and Schema Collections.

MetaDataCollections

This schema collection exposes information about all of the schema collections supported by the .NET Framework managed provider that is currently used to connect to the database.

COLUMNNAME	DATATYPE	DESCRIPTION
CollectionName	string	The name of the collection to pass to the **GetSchema** method to return the collection.
NumberOfRestrictions	int	The number of restrictions that may be specified for the collection.
NumberOfIdentifierParts	int	The number of parts in the composite identifier/database object name. For example, in SQL Server, this would be 3 for tables and 4 for columns. In Oracle, it would be 2 for tables and 3 for columns.

DataSourceInformation

This schema collection exposes information about data source that the .NET Framework managed provider is currently connect to.

COLUMNNAME	DATATYPE	DESCRIPTION

COLUMNNAME	DATATYPE	DESCRIPTION
CompositeIdentifierSeparatorPattern	string	The regular expression to match the composite separators in a composite identifier. For example, "\." (for SQL Server) or "@|\." (for Oracle). A composite identifier is typically what is used for a database object name, for example: pubs.dbo.authors or pubs@dbo.authors. For SQL Server, use the regular expression "\.". For OracleClient, use "@|\.". For ODBC use the Catalog_name_seperator. For OLE DB use DBLITERAL_CATALOG_SEPARATOR or DBLITERAL_SCHEMA_SEPARATOR.
DataSourceProductName	string	The name of the product accessed by the provider, such as "Oracle" or "SQLServer".
DataSourceProductVersion	string	Indicates the version of the product accessed by the provider, in the data sources native format and not in Microsoft format. In some cases DataSourceProductVersion and DataSourceProductVersionNormalized will be the same value. In the case of OLE DB and ODBC, these will always be the same as they are mapped to the same function call in the underlying native API.

COLUMNNAME	DATATYPE	DESCRIPTION
DataSourceProductVersionNormalized	string	A normalized version for the data source, such that it can be compared with `String.Compare()`. The format of this is consistent for all versions of the provider to prevent version 10 from sorting between version 1 and version 2. For example, the Oracle provider uses a format of "nn.nn.nn.nn.nn" for its normalized version, which causes an Oracle 8i data source to return "08.01.07.04.01". SQL Server uses the typical Microsoft "nn.nn.nnnn" format. In some cases, DataSourceProductVersion and DataSourceProductVersionNormalized will be the same value. In the case of OLE DB and ODBC these will always be the same as they are mapped to the same function call in the underlying native API.
GroupByBehavior	GroupByBehavior	Specifies the relationship between the columns in a GROUP BY clause and the non-aggregated columns in the select list.
IdentifierPattern	string	A regular expression that matches an identifier and has a match value of the identifier. For example "[A-Za-z0-9_#$]".
IdentifierCase	IdentifierCase	Indicates whether non-quoted identifiers are treated as case sensitive or not.
OrderByColumnsInSelect	bool	Specifies whether columns in an ORDER BY clause must be in the select list. A value of true indicates that they are required to be in the select list, a value of false indicates that they are not required to be in the select list.

COLUMNNAME	DATATYPE	DESCRIPTION
ParameterMarkerFormat	string	A format string that represents how to format a parameter. If named parameters are supported by the data source, the first placeholder in this string should be where the parameter name should be formatted. For example, if the data source expects parameters to be named and prefixed with an ':' this would be ":{0}". When formatting this with a parameter name of "p1" the resulting string is ":p1". If the data source expects parameters to be prefixed with the '@', but the names already include them, this would be '{0}', and the result of formatting a parameter named "@p1" would simply be "@p1". For data sources that do not expect named parameters and expect the use of the '?' character, the format string can be specified as simply '?', which would ignore the parameter name. For OLE DB we return '?'.
ParameterMarkerPattern	string	A regular expression that matches a parameter marker. It will have a match value of the parameter name, if any. For example, if named parameters are supported with an '@' lead-in character that will be included in the parameter name, this would be: "(@[A-Za-z0-9_$#]*)". However, if named parameters are supported with a ':' as the lead-in character and it is not part of the parameter name, this would be: ":([A-Za-z0-9_$#]*)". Of course, if the data source doesn't support named parameters, this would simply be "?".
ParameterNameMaxLength	int	The maximum length of a parameter name in characters. Visual Studio expects that if parameter names are supported, the minimum value for the maximum length is 30 characters. If the data source does not support named parameters, this property returns zero.

COLUMNNAME	DATATYPE	DESCRIPTION
ParameterNamePattern	string	A regular expression that matches the valid parameter names. Different data sources have different rules regarding the characters that may be used for parameter names. Visual Studio expects that if parameter names are supported, the characters "\p{Lu}\p{Ll}\p{Lt}\p{Lm}\p{Lo}\p{Nl}\p{Nd}" are the minimum supported set of characters that are valid for parameter names.
QuotedIdentifierPattern	string	A regular expression that matches a quoted identifier and has a match value of the identifier itself without the quotes. For example, if the data source used double-quotes to identify quoted identifiers, this would be: "(([^\"]|\"\")*)".
QuotedIdentifierCase	IdentifierCase	Indicates whether quoted identifiers are treated as case sensitive or not.
StatementSeparatorPattern	string	A regular expression that matches the statement separator.
StringLiteralPattern	string	A regular expression that matches a string literal and has a match value of the literal itself. For example, if the data source used single-quotes to identify strings, this would be: "('([^']|'')*)'"
SupportedJoinOperators	SupportedJoinOperators	Specifies what types of SQL join statements are supported by the data source.

DataTypes

This schema collection exposes information about the data types that are supported by the database that the .NET Framework managed provider is currently connected to.

COLUMNNAME	DATATYPE	DESCRIPTION
TypeName	string	The provider-specific data type name.
ProviderDbType	int	The provider-specific type value that should be used when specifying a parameter's type. For example, SqlDbType.Money or OracleType.Blob.

COLUMNNAME	DATATYPE	DESCRIPTION
ColumnSize	long	The length of a non-numeric column or parameter refers to either the maximum or the length defined for this type by the provider. For character data, this is the maximum or defined length in units, defined by the data source. Oracle has the concept of specifying a length and then specifying the actual storage size for some character data types. This defines only the length in units for Oracle. For date-time data types, this is the length of the string representation (assuming the maximum allowed precision of the fractional seconds component). If the data type is numeric, this is the upper bound on the maximum precision of the data type.
CreateFormat	string	Format string that represents how to add this column to a data definition statement, such as CREATE TABLE. Each element in the CreateParameter array should be represented by a "parameter marker" in the format string. For example, the SQL data type DECIMAL needs a precision and a scale. In this case, the format string would be "DECIMAL({0},{1})".
CreateParameters	string	The creation parameters that must be specified when creating a column of this data type. Each creation parameter is listed in the string, separated by a comma in the order they are to be supplied. For example, the SQL data type DECIMAL needs a precision and a scale. In this case, the creation parameters should contain the string "precision, scale". In a text command to create a DECIMAL column with a precision of 10 and a scale of 2, the value of the CreateFormat column might be DECIMAL({0},{1})" and the complete type specification would be DECIMAL(10,2).
DataType	string	The name of the .NET Framework type of the data type.

COLUMNNAME	DATATYPE	DESCRIPTION
IsAutoincrementable	bool	true—Values of this data type may be auto-incrementing. false—Values of this data type may not be auto-incrementing. Note that this merely indicates whether a column of this data type may be auto-incrementing, not that all columns of this type are auto-incrementing.
IsBestMatch	bool	true—The data type is the best match between all data types in the data store and the .NET Framework data type indicated by the value in the DataType column. false—The data type is not the best match. For each set of rows in which the value of the DataType column is the same, the IsBestMatch column is set to true in only one row.
IsCaseSensitive	bool	true—The data type is a character type and is case-sensitive. false—The data type is not a character type or is not case-sensitive.
IsFixedLength	bool	true—Columns of this data type created by the data definition language (DDL) will be of fixed length. false—Columns of this data type created by the DDL will be of variable length. DBNull.Value—It is not known whether the provider will map this field with a fixed-length or variable-length column.
IsFixedPrecisionScale	bool	true—The data type has a fixed precision and scale. false—The data type does not have a fixed precision and scale.
IsLong	bool	true—The data type contains very long data; the definition of very long data is provider-specific. false—The data type does not contain very long data.

COLUMNNAME	DATATYPE	DESCRIPTION
IsNullable	bool	true—The data type is nullable. false—The data type is not nullable. DBNull.Value—It is not known whether the data type is nullable.
IsSearchable	bool	true—The data type can be used in a WHERE clause with any operator except the LIKE predicate. false—The data type cannot be used in a WHERE clause with any operator except the LIKE predicate.
IsSearchableWithLike	bool	true—The data type can be used with the LIKE predicate false—The data type cannot be used with the LIKE predicate.
IsUnsigned	bool	true—The data type is unsigned. false—The data type is signed. DBNull.Value—Not applicable to data type.
MaximumScale	short	If the type indicator is a numeric type, this is the maximum number of digits allowed to the right of the decimal point. Otherwise, this is DBNull.Value.
MinimumScale	short	If the type indicator is a numeric type, this is the minimum number of digits allowed to the right of the decimal point. Otherwise, this is DBNull.Value.
IsConcurrencyType	bool	true – the data type is updated by the database every time the row is changed and the value of the column is different from all previous values false – the data type is note updated by the database every time the row is changed DBNull.Value – the database does not support this type of data type
IsLiteralSupported	bool	true – the data type can be expressed as a literal false – the data type can not be expressed as a literal
LiteralPrefix	string	The prefix applied to a given literal.

COLUMNNAME	DATATYPE	DESCRIPTION
LiteralSuffix	string	The suffix applied to a given literal.
NativeDataType	String	NativeDataType is an OLE DB specific column for exposing the OLE DB type of the data type .

Restrictions

This schema collection exposed information about the restrictions that are supported by the .NET Framework managed provider that is currently used to connect to the database.

COLUMNNAME	DATATYPE	DESCRIPTION
CollectionName	string	The name of the collection that these restrictions apply to.
RestrictionName	string	The name of the restriction in the collection.
RestrictionDefault	string	Ignored.
RestrictionNumber	int	The actual location in the collections restrictions that this particular restriction falls in.

ReservedWords

This schema collection exposes information about the words that are reserved by the database that the .NET Framework managed provider that is currently connected to.

COLUMNNAME	DATATYPE	DESCRIPTION
ReservedWord	string	Provider specific reserved word.

See also

- Retrieving Database Schema Information
- GetSchema and Schema Collections
- ADO.NET Overview

SQL Server Schema Collections

9/7/2019 • 11 minutes to read • Edit Online

The Microsoft .NET Framework Data Provider for SQL Server supports additional schema collections in addition to the common schema collections. The schema collections vary slightly by the version of SQL Server you are using. To determine the list of supported schema collections, call the **GetSchema** method with no arguments, or with the schema collection name "MetaDataCollections". This will return a DataTable with a list of the supported schema collections, the number of restrictions that they each support, and the number of identifier parts that they use.

Databases

COLUMNNAME	DATATYPE	DESCRIPTION
database_name	String	Name of the database.
dbid	Int16	Database ID.
create_date	DateTime	Creation Date of the database.

Foreign Keys

COLUMNNAME	DATATYPE	DESCRIPTION
CONSTRAINT_CATALOG	String	Catalog the constraint belongs to.
CONSTRAINT_SCHEMA	String	Schema that contains the constraint.
CONSTRAINT_NAME	String	Name.
TABLE_CATALOG	String	Table Name constraint is part of.
TABLE_SCHEMA	String	Schema that contains the table.
TABLE_NAME	String	Table Name
CONSTRAINT_TYPE	String	Type of constraint. Only "FOREIGN KEY" is allowed.
IS_DEFERRABLE	String	Specifies whether the constraint is deferrable. Returns NO.
INITIALLY_DEFERRED	String	Specifies whether the constraint is initially deferrable. Returns NO.

Indexes

COLUMNNAME	DATATYPE	DESCRIPTION
constraint_catalog	String	Catalog that index belongs to.
constraint_schema	String	Schema that contains the index.
constraint_name	String	Name of the index.
table_catalog	String	Table name the index is associated with.
table_schema	String	Schema that contains the table the index is associated with.
table_name	String	Table Name.
index_name	String	Index Name.

Indexes (SQL Server 2008)

Beginning with the .NET Framework version 3.5 SP1 and SQL Server 2008, the following columns have been added to the Indexes schema collection to support new spatial types, filestream and sparse columns. These columns are not supported in earlier versions of the .NET Framework and SQL Server.

COLUMNNAME	DATATYPE	DESCRIPTION
type_desc	String	The type of the index will be one of the following: - HEAP - CLUSTERED - NONCLUSTERED - XML - SPATIAL

IndexColumns

COLUMNNAME	DATATYPE	DESCRIPTION
constraint_catalog	String	Catalog that index belongs to.
constraint_schema	String	Schema that contains the index.
constraint_name	String	Name of the index.
table_catalog	String	Table name the index is associated with.
table_schema	String	Schema that contains the table the index is associated with.
table_name	String	Table Name.
column_name	String	Column name the index is associated with.

COLUMNNAME	DATATYPE	DESCRIPTION
ordinal_position	Int32	Column ordinal position.
KeyType	Byte	The type of object.
index_name	String	Index Name.

Procedures

COLUMNNAME	DATATYPE	DESCRIPTION
SPECIFIC_CATALOG	String	Specific name for the catalog.
SPECIFIC_SCHEMA	String	Specific name of the schema.
SPECIFIC_NAME	String	Specific name of the catalog.
ROUTINE_CATALOG	String	Catalog the stored procedure belongs to.
ROUTINE_SCHEMA	String	Schema that contains the stored procedure.
ROUTINE_NAME	String	Name of the stored procedure.
ROUTINE_TYPE	String	Returns PROCEDURE for stored procedures and FUNCTION for functions.
CREATED	DateTime	Time the procedure was created.
LAST_ALTERED	DateTime	The last time the procedure was modified.

Procedure Parameters

COLUMNNAME	DATATYPE	DESCRIPTION
SPECIFIC_CATALOG	String	Catalog name of the procedure for which this is a parameter.
SPECIFIC_SCHEMA	String	Schema that contains the procedure for which this parameter is part of.
SPECIFIC_NAME	String	Name of the procedure for which this parameter is a part of.
ORDINAL_POSITION	Int32	Ordinal position of the parameter starting at 1. For the return value of a procedure, this is a 0.

COLUMNNAME	DATATYPE	DESCRIPTION
PARAMETER_MODE	String	Returns IN if an input parameter, OUT if an output parameter, and INOUT if an input/output parameter.
IS_RESULT	String	Returns YES if indicates result of the procedure that is a function. Otherwise, returns NO.
AS_LOCATOR	String	Returns YES if declared as locator. Otherwise, returns NO.
PARAMETER_NAME	String	Name of the parameter. NULL if this corresponds to the return value of a function.
DATA_TYPE	String	System-supplied data type.
CHARACTER_MAXIMUM_LENGTH	Int32	Maximum length in characters for binary or character data types. Otherwise, returns NULL.
CHARACTER_OCTET_LENGTH	Int32	Maximum length, in bytes, for binary or character data types. Otherwise, returns NULL.
COLLATION_CATALOG	String	Catalog name of the collation of the parameter. If not one of the character types, returns NULL.
COLLATION_SCHEMA	String	Always returns NULL.
COLLATION_NAME	String	Name of the collation of the parameter. If not one of the character types, returns NULL.
CHARACTER_SET_CATALOG	String	Catalog name of the character set of the parameter. If not one of the character types, returns NULL.
CHARACTER_SET_SCHEMA	String	Always returns NULL.
CHARACTER_SET_NAME	String	Name of the character set of the parameter. If not one of the character types, returns NULL.
NUMERIC_PRECISION	Byte	Precision of approximate numeric data, exact numeric data, integer data, or monetary data. Otherwise, returns NULL.
NUMERIC_PRECISION_RADIX	Int16	Precision radix of approximate numeric data, exact numeric data, integer data, or monetary data. Otherwise, returns NULL.

COLUMNNAME	DATATYPE	DESCRIPTION
NUMERIC_SCALE	Int32	Scale of approximate numeric data, exact numeric data, integer data, or monetary data. Otherwise, returns NULL.
DATETIME_PRECISION	Int16	Precision in fractional seconds if the parameter type is datetime or smalldatetime. Otherwise, returns NULL.
INTERVAL_TYPE	String	NULL. Reserved for future use by SQL Server.
INTERVAL_PRECISION	Int16	NULL. Reserved for future use by SQL Server.

Tables

COLUMNNAME	DATATYPE	DESCRIPTION
TABLE_CATALOG	String	Catalog of the table.
TABLE_SCHEMA	String	Schema that contains the table.
TABLE_NAME	String	Table name.
TABLE_TYPE	String	Type of table. Can be VIEW or BASE TABLE.

Columns

COLUMNNAME	DATATYPE	DESCRIPTION
TABLE_CATALOG	String	Catalog of the table.
TABLE_SCHEMA	String	Schema that contains the table.
TABLE_NAME	String	Table name.
COLUMN_NAME	String	Column name.
ORDINAL_POSITION	Int32	Column identification number.
COLUMN_DEFAULT	String	Default value of the column
IS_NULLABLE	String	Nullability of the column. If this column allows NULL, this column returns YES. Otherwise, No is returned.
DATA_TYPE	String	System-supplied data type.

COLUMNNAME	DATATYPE	DESCRIPTION
CHARACTER_MAXIMUM_LENGTH	Int32 – Sql8, Int16 – Sql7	Maximum length, in characters, for binary data, character data, or text and image data. Otherwise, NULL is returned.
CHARACTER_OCTET_LENGTH	Int32 – SQL8, Int16 – Sql7	Maximum length, in bytes, for binary data, character data, or text and image data. Otherwise, NULL is returned.
NUMERIC_PRECISION	Unsigned Byte	Precision of approximate numeric data, exact numeric data, integer data, or monetary data. Otherwise, NULL is returned.
NUMERIC_PRECISION_RADIX	Int16	Precision radix of approximate numeric data, exact numeric data, integer data, or monetary data. Otherwise, NULL is returned.
NUMERIC_SCALE	Int32	Scale of approximate numeric data, exact numeric data, integer data, or monetary data. Otherwise, NULL is returned.
DATETIME_PRECISION	Int16	Subtype code for datetime and SQL-92 interval data types. For other data types, NULL is returned.
CHARACTER_SET_CATALOG	String	Returns master, indicating the database in which the character set is located, if the column is character data or text data type. Otherwise, NULL is returned.
CHARACTER_SET_SCHEMA	String	Always returns NULL.
CHARACTER_SET_NAME	String	Returns the unique name for the character set if this column is character data or text data type. Otherwise, NULL is returned.
COLLATION_CATALOG	String	Returns master, indicating the database in which the collation is defined, if the column is character data or text data type. Otherwise, this column is NULL.

Columns (SQL Server 2008)

Beginning with the .NET Framework version 3.5 SP1 and SQL Server 2008, the following columns have been added to the Columns schema collection to support new spatial types, filestream and sparse columns. These columns are not supported in earlier versions of the .NET Framework and SQL Server.

COLUMNNAME	DATATYPE	DESCRIPTION

COLUMNNAME	DATATYPE	DESCRIPTION
IS_FILESTREAM	String	YES if the column has FILESTREAM attribute. NO if the column does not have FILESTREAM attribute.
IS_SPARSE	String	YES if the column is a sparse column. NO if the column is not a sparse column.
IS_COLUMN_SET	String	YES if the column is a column set column. NO if the column is not a column set column.

AllColumns (SQL Server 2008)

Beginning with the .NET Framework version 3.5 SP1 and SQL Server 2008, the AllColumns schema collection has been added to support sparse columns. AllColumns is not supported in earlier versions of the .NET Framework and SQL Server.

AllColumns has the same restrictions and resulting DataTable schema as the Columns schema collection. The only difference is that AllColumns includes column set columns that are not included in the Columns schema collection. The following table describes these columns.

COLUMNNAME	DATATYPE	DESCRIPTION
TABLE_CATALOG	String	Catalog of the table.
TABLE_SCHEMA	String	Schema that contains the table.
TABLE_NAME	String	Table name.
COLUMN_NAME	String	Column name.
ORDINAL_POSITION	Int32	Column identification number.
COLUMN_DEFAULT	String	Default value of the column
IS_NULLABLE	String	Nullability of the column. If this column allows NULL, this column returns YES. Otherwise, NO is returned.
DATA_TYPE	String	System-supplied data type.
CHARACTER_MAXIMUM_LENGTH	Int32	Maximum length, in characters, for binary data, character data, or text and image data. Otherwise, NULL is returned.
CHARACTER_OCTET_LENGTH	Int32	Maximum length, in bytes, for binary data, character data, or text and image data. Otherwise, NULL is returned.

COLUMNNAME	DATATYPE	DESCRIPTION
NUMERIC_PRECISION	Unsigned Byte	Precision of approximate numeric data, exact numeric data, integer data, or monetary data. Otherwise, NULL is returned.
NUMERIC_PRECISION_RADIX	Int16	Precision radix of approximate numeric data, exact numeric data, integer data, or monetary data. Otherwise, NULL is returned.
NUMERIC_SCALE	Int32	Scale of approximate numeric data, exact numeric data, integer data, or monetary data. Otherwise, NULL is returned.
DATETIME_PRECISION	Int16	Subtype code for datetime and SQL-92 interval data types. For other data types, NULL is returned.
CHARACTER_SET_CATALOG	String	Returns master, indicating the database in which the character set is located, if the column is character data or text data type. Otherwise, NULL is returned.
CHARACTER_SET_SCHEMA	String	Always returns NULL.
CHARACTER_SET_NAME	String	Returns the unique name for the character set if this column is character data or text data type. Otherwise, NULL is returned.
COLLATION_CATALOG	String	Returns master, indicating the database in which the collation is defined, if the column is character data or text data type. Otherwise, this column is NULL.
IS_FILESTREAM	String	YES if the column has FILESTREAM attribute. NO if the column does not have FILESTREAM attribute.
IS_SPARSE	String	YES if the column is a sparse column. NO if the column is not a sparse column.
IS_COLUMN_SET	String	YES if the column is a column set column. NO if the column is not a column set column.

ColumnSetColumns (SQL Server 2008)

Beginning with the .NET Framework version 3.5 SP1 and SQL Server 2008, the ColumnSetColumns schema collection has been added to support sparse columns. ColumnSetColumns is not supported in earlier versions of

the .NET Framework and SQL Server. The ColumnSetColumns schema collection returns the schema for all of the columns in a column set. The following table describes these columns.

COLUMNNAME	DATATYPE	DESCRIPTION
TABLE_CATALOG	String	Catalog of the table.
TABLE_SCHEMA	String	Schema that contains the table.
TABLE_NAME	String	Table name.
COLUMN_NAME	String	Column name.
ORDINAL_POSITION	Int32	Column identification number.
COLUMN_DEFAULT	String	Default value of the column
IS_NULLABLE	String	Nullability of the column. If this column allows NULL, this column returns YES. Otherwise, NO is returned.
DATA_TYPE	String	System-supplied data type.
CHARACTER_MAXIMUM_LENGTH	Int32	Maximum length, in characters, for binary data, character data, or text and image data. Otherwise, NULL is returned.
CHARACTER_OCTET_LENGTH	Int32	Maximum length, in bytes, for binary data, character data, or text and image data. Otherwise, NULL is returned.
NUMERIC_PRECISION	Unsigned Byte	Precision of approximate numeric data, exact numeric data, integer data, or monetary data. Otherwise, NULL is returned.
NUMERIC_PRECISION_RADIX	Int16	Precision radix of approximate numeric data, exact numeric data, integer data, or monetary data. Otherwise, NULL is returned.
NUMERIC_SCALE	Int32	Scale of approximate numeric data, exact numeric data, integer data, or monetary data. Otherwise, NULL is returned.
DATETIME_PRECISION	Int16	Subtype code for datetime and SQL-92 interval data types. For other data types, NULL is returned.
CHARACTER_SET_CATALOG	String	Returns master, indicating the database in which the character set is located, if the column is character data or text data type. Otherwise, NULL is returned.

COLUMNNAME	DATATYPE	DESCRIPTION
CHARACTER_SET_SCHEMA	String	Always returns NULL.
CHARACTER_SET_NAME	String	Returns the unique name for the character set if this column is character data or text data type. Otherwise, NULL is returned.
COLLATION_CATALOG	String	Returns master, indicating the database in which the collation is defined, if the column is character data or text data type. Otherwise, this column is NULL.
IS_FILESTREAM	String	YES if the column has FILESTREAM attribute. NO if the column does not have FILESTREAM attribute.
IS_SPARSE	String	YES if the column is a sparse column. NO if the column is not a sparse column.
IS_COLUMN_SET	String	YES if the column is a column set column. NO if the column is not a column set column.

Users

COLUMNNAME	DATATYPE	DESCRIPTION
uid	Int16	User ID, unique in this database. 1 is the database owner.
user_name	String	Username or group name, unique in this database.
createdate	DateTime	Date the account was added.
updatedate	DateTime	Date the account was last changed.

Views

COLUMNNAME	DATATYPE	DESCRIPTION
TABLE_CATALOG	String	Catalog of the view.
TABLE_SCHEMA	String	Schema that contains the view.
TABLE_NAME	String	View name.

COLUMNNAME	DATATYPE	DESCRIPTION
CHECK_OPTION	String	Type of WITH CHECK OPTION. Is CASCADE if the original view was created using the WITH CHECK OPTION. Otherwise, NONE is returned.
IS_UPDATABLE	String	Specifies whether the view is updatable. Always returns NO.

ViewColumns

COLUMNNAME	DATATYPE	DESCRIPTION
VIEW_CATALOG	String	Catalog of the view.
VIEW_SCHEMA	String	Schema that contains the view.
VIEW_NAME	String	View name.
TABLE_CATALOG	String	Catalog of the table that is associated with this view.
TABLE_SCHEMA	String	Schema that contains the table that is associated with this view.
TABLE_NAME	String	Name of the table that is associated with the view. Base Table.
COLUMN_NAME	String	Column name.

UserDefinedTypes

COLUMNNAME	DATATYPE	DESCRIPTION
assembly_name	String	The name of the file for the assembly.
udt_name	String	The class name for the assembly.
version_major	Object	Major Version Number.
version_minor	Object	Minor Version Number.
version_build	Object	Build Number.
version_revision	Object	Revision Number.
culture_info	Object	The culture information associated with this UDT.
public_key	Object	The public key used by this Assembly.

COLUMNNAME	DATATYPE	DESCRIPTION
is_fixed_length	Boolean	Specifies whether length of type is always same as max_length.
max_length	Int16	Maximum length of type in bytes.
Create_Date	DateTime	The date the assembly was created/registered.
Permission_set_desc	String	The friendly name for the permission-set/security-level for the assembly.

See also

- Retrieving Database Schema Information
- ADO.NET Overview

Oracle Schema Collections

9/7/2019 • 14 minutes to read • Edit Online

The Microsoft .NET Framework Data Provider for Oracle supports the following specific schema collections in addition to the common schema collections:

- Columns
- Indexes
- IndexColumns
- Procedures
- Sequences
- Synonyms
- Tables
- Users
- Views
- Functions
- Packages
- PackageBodies
- Arguments
- UniqueKeys
- PrimaryKeys
- ForeignKeys
- ForeignKeyColumns
- ProcedureParameters

Columns

COLUMNNAME	DATATYPE	DESCRIPTION
OWNER	String	Owner of the table, view or cluster.
TABLE_NAME	String	Table, view, or cluster name.
COLUMN_NAME	String	Column name.
ID	Decimal	Sequence number of the column as created.
DATATYPE	String	Datatype of the column.

COLUMNNAME	DATATYPE	DESCRIPTION
LENGTH	Decimal	Length of the column in bytes.
PRECISION	Decimal	Decimal precision for NUMBER datatype; binary precision for FLOAT datatype, null for all other datatypes.
SCALE	Decimal	Digits to right of decimal point in a number.
NULLABLE	String	Specifies whether a column allows NULLs. Value is N if there is a NOT NULL constraint on the column or if the column is part of a PRIMARY KEY.

Indexes

COLUMNNAME	DATATYPE	DESCRIPTION
OWNER	String	Owner of the index
INDEX_NAME	String	Name of the index.
INDEX_TYPE	String	Type of index (NORMAL, BITMAP, FUNCTION-BASED NORMAL, FUNCTION-BASED BITMAP, or DOMAIN).
TABLE_OWNER	String	Owner of the indexed object.
TABLE_NAME	String	Name of the indexed object.
TABLE_TYPE	String	Type of the indexed object (for example, TABLE, CLUSTER).
UNIQUENESS	String	Whether the index is UNIQUE or NONUNIQUE.
COMPRESSION	String	Whether the index is ENABLED or DISABLED.
PREFIX_LENGTH	Decimal	Number of columns in the prefix of the compression key.
TABLESPACE_NAME	String	Name of the tablespace containing the index.
INI_TRANS	Decimal	Initial number of transactions.
MAX_TRANS	Decimal	Maximum number of transactions.
INITIAL_EXTENT	Decimal	Size of the initial extent.

COLUMNNAME	DATATYPE	DESCRIPTION
NEXT_EXTENT	Decimal	Size of secondary extents.
MIN_EXTENTS	Decimal	Minimum number of extents allowed in the segment.
MAX_EXTENTS	Decimal	Maximum number of extents allowed in the segment.
PCT_INCREASE	Decimal	Percentage increase in extent size.
PCT_THRESHOLD	Decimal	Threshold percentage of block space allowed per index entry.
INCLUDE_COLUMN	Decimal	Column ID of the last column to be included in index-organized table primary key (non-overflow) index. This column maps to the COLUMN_ID column of the *_TAB_COLUMNS data dictionary views.
FREELISTS	Decimal	Number of process freelists allocated to this segment.
FREELIST_GROUPS	Decimal	Number of freelist groups allocated to this segment.
PCT_FREE	Decimal	Minimum percentage of free space in a block.
LOGGING	String	Logging information.
BLEVEL	Decimal	B*-Tree level: depth of the index from its root block to its leaf blocks. A depth of 0 indicates that the root block and leaf block are the same.
LEAF_BLOCKS	Decimal	Number of leaf blocks in the index
DISTINCT_KEYS	Decimal	Number of distinct indexed values. For indexes that enforce UNIQUE and PRIMARY KEY constraints, this value is the same as the number of rows in the table (USER_TABLES.NUM_ROWS).
AVG_LEAF_BLOCKS_PER_KEY	Decimal	Average number of leaf blocks in which each distinct value in the index appears rounded to the nearest integer. For indexes that enforce UNIQUE and PRIMARY KEY constraints, this value is always 1.

COLUMNNAME	DATATYPE	DESCRIPTION
AVG_DATA_BLOCKS_PER_KEY	Decimal	Average number of data blocks in the table that are pointed to by a distinct value in the index rounded to the nearest integer. This statistic is the average number of data blocks that contain rows that contain a given value for the indexed columns.
CLUSTERING_FACTOR	Decimal	Indicates the amount of order of the rows in the table based on the values of the index.
STATUS	String	Whether a nonpartitioned index is VALID or UNUSABLE.
NUM_ROWS	Decimal	Number of rows in the index.
SAMPLE_SIZE	Decimal	Size of the sample used to analyze the index.
LAST_ANALYZED	DateTime	Date on which this index was most recently analyzed.
DEGREE	String	Number of threads per instance for scanning the index.
INSTANCES	String	Number of instances across which the indexes to be scanned.
PARTITIONED	String	Whether this index is partitioned (YES | NO).
TEMPORARY	String	Whether the index is on a temporary table.
GENERATED	String	Whether the name of the index is system generated (Y|N).
SECONDARY	String	Whether the index is a secondary object created by the ODCIIndexCreate method of the Oracle9i Data Cartridge (Y|N).
BUFFER_POOL	String	Name of the default buffer pool to be used for the index blocks.
USER_STATS	String	Whether the statistics were entered directly by the user.
DURATION	String	Indicates the duration of a temporary table: 1)SYS$SESSION: the rows are preserved for the duration of the session, 2) SYS$TRANSACTION: the rows are deleted after COMMIT, 3) Null for permanent Table.

COLUMNNAME	DATATYPE	DESCRIPTION
PCT_DIRECT_ACCESS	Decimal	For a secondary index on an index-organized table, the percentage of rows with VALID guess
ITYP_OWNER	String	For a domain index, the owner of the indextype.
ITYP_NAME	String	For a domain index, the name of the indextype.
PARAMETERS	String	For a domain index, the parameter string.
GLOBAL_STATS	String	For partitioned indexes, indicates whether statistics were collected by analyzing index as a whole (YES) or were estimated from statistics on underlying index partitions and subpartitions (NO).
DOMIDX_STATUS	String	Reflects the status of the domain index. NULL: the specified index is not a domain index. VALID: the index is a valid domain index. IDXTYP_INVLD: the index type of this domain index is invalid.
DOMIDX_OPSTATUS	String	Reflects the status of an operation that was performed on a domain index: NULL: the specified index is not a domain index. VALID: the operation performed without errors. FAILED: the operation failed with an error.
FUNCIDX_STATUS	String	Indicates the status of a function-based index: NULL: this is not a function-based index, ENABLED: the function-based index is enabled, DISABLED: the function-based index is disabled.
JOIN_INDEX	String	Indicates whether this is a join index or not.

IndexColumns

COLUMNNAME	DATATYPE	DESCRIPTION
INDEX_OWNER	String	Owner of the index.
INDEX_NAME	String	Name of the index.
TABLE_OWNER	String	Owner of the table or cluster.
TABLE_NAME	String	Name of the table or cluster.

COLUMNNAME	DATATYPE	DESCRIPTION
COLUMN_NAME	String	Column name or attribute of object type column.
COLUMN_POSITION	Decimal	Position of column or attribute within the index.
COLUMN_LENGTH	Decimal	Indexed length of the column.
CHAR_LENGTH	Decimal	Maximum codepoint length of the column.
DESCEND	String	Whether the column is sorted in descending order.

Procedures

COLUMNNAME	DATATYPE	DESCRIPTION
OWNER	String	Owner of the object.
OBJECT_NAME	String	Name of the object.
SUBOBJECT_NAME	String	Name of the subobject (for example, partition).
OBJECT_ID	Decimal	Dictionary object number of the object.
DATA_OBJECT_ID	Decimal	Dictionary object number of the segment that contains the object.
LAST_DDL_TIME	DateTime	Timestamp for the last modification of the object resulting from a DDL command (including grants and revokes).
TIMESTAMP	String	Timestamp for the specification of the object (character data).
STATUS	String	Status of the object (VALID, INVALID, or N/A).
TEMPORARY	String	Whether the object is temporary (the current session can see only data that it placed in this object itself).
GENERATED	String	Was the name of this object system generated? (Y | N).
SECONDARY	String	Whether this is a secondary object created by the ODCIIndexCreate method of the Oracle9i Data Cartridge (Y | N).

COLUMNNAME	DATATYPE	DESCRIPTION
CREATED	DateTime	The date the object was created.

Sequences

COLUMNNAME	DATATYPE	DESCRIPTION
SEQUENCE_OWNER	String	Name of the owner of the sequence.
SEQUENCE_NAME	String	Sequence name.
MIN_VALUE	Decimal	Minimum value of the sequence.
MAX_VALUE	Decimal	Maximum value of the sequence.
INCREMENT_BY	Decimal	Value by which sequence is incremented.
CYCLE_FLAG	String	Does sequence wrap around on reaching limit.
ORDER_FLAG	String	Are sequence numbers generated in order.
CACHE_SIZE	Decimal	Number of sequence numbers to cache.
LAST_NUMBER	Decimal	Last sequence number written to disk. If a sequence uses caching, the number written to disk is the last number placed in the sequence cache. This number is likely to be greater than the last sequence number that was used.

Synonyms

COLUMNNAME	DATATYPE	DESCRIPTION
OWNER	String	Owner of the synonym.
SYNONYM_NAME	String	Name of the synonym.
TABLE_OWNER	String	Owner of the object referenced by the synonym.
TABLE_NAME	String	Name of the object referenced by the synonym.
DB_LINK	String	Name of the database link referenced, if any.

Tables

COLUMNNAME	DATATYPE	DESCRIPTION
OWNER	String	Owner of the table.
TABLE_NAME	String	Name of the table.
TYPE	String	Type of table.

Users

COLUMNNAME	DATATYPE	DESCRIPTION
NAME	String	Name of the user.
ID	Decimal	ID number of the user.
CREATEDATE	DateTime	User creation date.

Views

COLUMNNAME	DATATYPE	DESCRIPTION
OWNER	String	Owner of the view.
VIEW_NAME	String	Name of the view.
TEXT_LENGTH	Decimal	Length of the view text.
TEXT	String	View text.
TYPE_TEXT_LENGTH	Decimal	Length of the type clause of the typed view.
TYPE_TEXT	String	Type clause of the typed view.
OID_TEXT_LENGTH	Decimal	Length of the WITH OID clause of the typed view.
OID_TEXT	String	WITH OID clause of the typed view.
VIEW_TYPE_OWNER	String	Owner of the type of the view if the view is a typed view.
VIEW_TYPE	String	Type of the view if the view is a typed view.
SUPERVIEW_NAME	String	Name of the superview.

Functions

COLUMNNAME	DATATYPE	DESCRIPTION
OWNER	String	Owner of the object.
OBJECT_NAME	String	Name of the object.
SUBOBJECT_NAME	String	Name of the subobject (for example, partition).
OBJECT_ID	Decimal	Dictionary object number of the object.
DATA_OBJECT_ID	Decimal	Dictionary object number of the segment that contains the object.
OBJECT_TYPE	String	Type of the object.
CREATED	DateTime	The date the object was created.
LAST_DDL_TIME	DateTime	Timestamp for the last modification of the object resulting from a DDL command (including grants and revokes).
TIMESTAMP	String	Timestamp for the specification of the object (character data)
STATUS	String	Status of the object (VALID, INVALID, or N/A).
TEMPORARY	String	Whether the object is temporary (the current session can see only data that it placed in this object itself).
GENERATED	String	Was the name of this object system generated? (Y | N).
SECONDARY	String	Whether this is a secondary object created by the ODCIIndexCreate method of the Oracle9i Data Cartridge (Y | N).

Packages

COLUMNNAME	DATATYPE	DESCRIPTION
OWNER	String	Owner of the object.
OBJECT_NAME	String	Name of the object.
SUBOBJECT_NAME	String	Name of the subobject (for example, partition).
OBJECT_ID	Decimal	Dictionary object number of the object.

COLUMNNAME	DATATYPE	DESCRIPTION
DATA_OBJECT_ID	Decimal	Dictionary object number of the segment that contains the object.
LAST_DDL_TIME	DateTime	Timestamp for the last modification of the object resulting from a DDL command (including grants and revokes).
TIMESTAMP	String	Timestamp for the specification of the object (character data).
STATUS	String	Status of the object (VALID, INVALID, or N/A).
TEMPORARY	String	Whether the object is temporary (the current session can see only data that it placed in this object itself).
GENERATED	String	Was the name of this object system generated? (Y | N).
SECONDARY	String	Whether this is a secondary object created by the ODCIIndexCreate method of the Oracle9i Data Cartridge (Y | N).
CREATED	DateTime	The date the object was created.

PackageBodies

COLUMNNAME	DATATYPE	DESCRIPTION
OWNER	String	Owner of the object.
OBJECT_NAME	String	Name of the object.
SUBOBJECT_NAME	String	Name of the subobject (for example, partition).
OBJECT_ID	Decimal	Dictionary object number of the object.
DATA_OBJECT_ID	Decimal	Dictionary object number of the segment that contains the object.
LAST_DDL_TIME	DateTime	Timestamp for the last modification of the object resulting from a DDL command (including grants and revokes).
TIMESTAMP	String	Timestamp for the specification of the object (character data).

COLUMNNAME	DATATYPE	DESCRIPTION
STATUS	String	Status of the object (VALID, INVALID, or N/A).
TEMPORARY	String	Whether the object is temporary (the current session can see only data that it placed in this object itself).
GENERATED	String	Was the name of this object system generated? (Y | N).
SECONDARY	String	Whether this is a secondary object created by the ODCIIndexCreate method of the Oracle9i Data Cartridge (Y | N).
CREATED	DateTime	The date the object was created.

Arguments

COLUMNNAME	DATATYPE	DESCRIPTION
OWNER	String	Name of the owner of the object.
PACKAGE_NAME	String	Package name.
OBJECT_NAME	String	Name of the procedure or function.
ARGUMENT_NAME	String	Name of the argument.
POSITION	Decimal	Position in argument list, or NULL for function return value.
SEQUENCE	Decimal	Argument sequence, including all nesting levels.
DEFAULT_VALUE	String	Default value for the argument.
DEFAULT_LENGTH	Decimal	Length of default value for the argument.
IN_OUT	String	Argument direction (IN, OUT, or IN/OUT).
DATA_LENGTH	Decimal	Length of the column in bytes.
DATA_PRECISION	Decimal	Length in decimal digits (NUMBER) or binary digits (FLOAT).
DATA_SCALE	Decimal	Digits to right of decimal point in a number.
DATA_TYPE	String	Data type of the argument.

UniqueKeys

COLUMNNAME	DATATYPE	DESCRIPTION
OWNER	String	Owner of the constraint definition.
CONSTRAINT_NAME	String	Name of the constraint definition.
TABLE_NAME	String	Name associated with the table (or view) with constraint definition.
SEARCH_CONDITION	String	Text of search condition for a check constraint.
R_OWNER	String	Owner of table referred to in a referential constraint.
R_CONSTRAINT_NAME	String	Name of the unique constraint definition for referenced table.
DELETE_RULE	String	Delete rule for a referential constraint (CASCADE or NO ACTION).
STATUS	String	Enforcement status of constraint (ENABLED or DISABLED).
DEFERRABLE	String	Whether the constraint is deferrable.
VALIDATED	String	Whether all data obeys the constraint (VALIDATED or NOT VALIDATED).
GENERATED	String	Whether the name of the constraint is user or system generated.
BAD	String	A YES value indicates that this constraint specifies a century in an ambiguous manner. To avoid errors resulting from this ambiguity, rewrite the constraint using the TO_DATE function with a four-digit year.
RELY	String	Whether an enabled constraint is enforced or unenforced.
LAST_CHANGE	DateTime	When the constraint was last enabled or disabled
INDEX_OWNER	String	Name of the user owning the index
INDEX_NAME	String	Name of the index

PrimaryKeys

COLUMNNAME	DATATYPE	DESCRIPTION
OWNER	String	Owner of the constraint definition.
CONSTRAINT_NAME	String	Name of the constraint definition.
TABLE_NAME	String	Name associated with the table (or view) with constraint definition.
SEARCH_CONDITION	String	Text of search condition for a check constraint.
R_OWNER	String	Owner of table referred to in a referential constraint.
R_CONSTRAINT_NAME	String	Name of the unique constraint definition for referenced table.
DELETE_RULE	String	Delete rule for a referential constraint (CASCADE or NO ACTION).
STATUS	String	Enforcement status of constraint (ENABLED or DISABLED).
DEFERRABLE	String	Whether the constraint is deferrable.
VALIDATED	String	Whether all data obeys the constraint (VALIDATED or NOT VALIDATED).
GENERATED	String	Whether the name of the constraint is user or system generated.
BAD	String	A YES value indicates that this constraint specifies a century in an ambiguous manner. To avoid errors resulting from this ambiguity, rewrite the constraint using the TO_DATE function with a four-digit year.
RELY	String	Whether an enabled constraint is enforced or unenforced.
LAST_CHANGE	DateTime	When the constraint was last enabled or disabled.
INDEX_OWNER	String	Name of the user owning the index.
INDEX_NAME	String	Name of the index.

ForeignKeys

COLUMNNAME	DATATYPE	DESCRIPTION
PRIMARY_KEY_CONSTRAINT_NAME	String	Name of the constraint definition.

COLUMNNAME	DATATYPE	DESCRIPTION
PRIMARY_KEY_OWNER	String	Owner of the constraint definition.
PRIMARY_KEY_TABLE_NAME	String	Name associated with the table (or view) with constraint definition
FOREIGN_KEY_OWNER	String	Owner of the constraint definition.
FOREIGN_KEY_CONSTRAINT_NAME	String	Name of the constraint definition.
FOREIGN_KEY_TABLE_NAME	String	Name associated with the table (or view) with constraint definition.
SEARCH_CONDITION	String	Text of search condition for a check constraint
R_OWNER	String	Owner of table referred to in a referential constraint.
R_CONSTRAINT_NAME	String	Name of the unique constraint definition for referenced table.
DELETE_RULE	String	Delete rule for a referential constraint (CASCADE or NO ACTION).
STATUS	String	Enforcement status of constraint (ENABLED or DISABLED).
VALIDATED	String	Whether all data obeys the constraint (VALIDATED or NOT VALIDATED).
GENERATED	String	Whether the name of the constraint is user or system generated.
RELY	String	Whether an enabled constraint is enforced or unenforced.
LAST_CHANGE	DateTime	When the constraint was last enabled or disabled.
INDEX_OWNER	String	Name of the user owning the index.
INDEX_NAME	String	Name of the index.

ForeignKeyColumns

COLUMNNAME	DATATYPE	DESCRIPTION
OWNER	String	Owner of the constraint definition.
CONSTRAINT_NAME	String	Name of the constraint definition.

COLUMNNAME	DATATYPE	DESCRIPTION
TABLE_NAME	String	Name of the table with constraint definition.
COLUMN_NAME	String	Name of the column or attribute of the object type column specified in the constraint definition.
POSITION	Decimal	Original position of column or attribute in the definition of the object.

ProcedureParameters

COLUMNNAME	DATATYPE	DESCRIPTION
OWNER	String	Owner of the object.
OBJECT_NAME	String	Name of the procedure or function.
PACKAGE_NAME	String	Name of the procedure or function.
OBJECT_ID	Decimal	Object number of the object.
OVERLOAD	String	Overload unique identifier.
ARGUMENT_NAME	String	Name of the argument.
POSITION	Decimal	Position in the argument list, or null for a function return value.
SEQUENCE	Decimal	Argument sequence, including all nesting levels.
DATA_LEVEL	Decimal	Nesting depth of the argument for composite types.
DATA_TYPE	String	Data type of the argument.
DEFAULT_VALUE	String	Default value for the argument.
DEFAULT_LENGTH	Decimal	Length of the default value for the argument.
IN_OUT	String	Argument Direction (IN, OUT, or IN/OUT).
DATA_LENGTH	Decimal	Length of the column (in bytes).
DATA_PRECISION	Decimal	Length in decimal digits (NUMBER) or binary digits (FLOAT).

COLUMNNAME	DATATYPE	DESCRIPTION
DATA_SCALE	Decimal	Digits to the right of the decimal point in a number.
RADIX	Decimal	Argument radix for a number.
CHARACTER_SET_NAME	String	Character set name for the argument.
TYPE_OWNER	String	Owner of the type of the argument.
TYPE_NAME	String	Name of the type of the argument. If the type is a package local type (that is, it is declared in a package specification), then this column displays the name of the package.
TYPE_SUBNAME	String	Relevant only for package local types. Displays the name of the type declared in the package identified in the TYPE_NAME column.
TYPE_LINK	String	Relevant only for package local types when the package identified in the TYPE_NAME column is a remote package. This column displays the database link used to refer to the remote package.
PLS_TYPE	String	For numeric arguments, the name of the PL/SQL type of the argument. Null otherwise.
CHAR_LENGTH	Decimal	Character limit for string data types.
CHAR_USED	String	Indicates whether the byte limit (B) or char limit (C) is official for the string.

See also

- ADO.NET Overview

ODBC Schema Collections

9/7/2019 • 2 minutes to read • Edit Online

This section discusses schema collection support for the ODBC drivers for Microsoft SQL Server, Oracle, and Microsoft Jet.

Microsoft SQL Server ODBC Driver

The Microsoft SQL Server ODBC Driver supports the following specific schema collections in addition to the common schema collections:

- Tables
- Indexes
- Columns
- Procedures
- ProcedureColumns
- ProcedureParameters
- Views

Tables and Views

COLUMNNAME	DATATYPE
TABLE_CAT	String
TABLE_SCHEM	String
TABLE_NAME	String
TABLE_TYPE	String
REMARKS	String

Indexes

COLUMNNAME	DATATYPE
TABLE_CAT	String
TABLE_SCHEM	String
TABLE_NAME	String
NON_UNIQUE	Int16
INDEX_QUALIFIER	String

COLUMNNAME	DATATYPE
INDEX_NAME	String
TYPE	Int16
ORDINAL_POSITION	Int16
COLUMN_NAME	String
ASC_OR_DESC	String
CARDINALITY	Int32
PAGES	Int32
FILTER_CONDITION	String
SS_TYPE_SCHEMA	String
SS_DATA_TYPE	Byte

Columns

COLUMNNAME	DATATYPE
TABLE_CAT	String
TABLE_SCHEM	String
TABLE_NAME	String
COLUMN_NAME	String
DATA_TYPE	Int16
TYPE_NAME	String
COLUMN_SIZE	Int32
BUFFER_LENGTH	Int32
DECIMAL_DIGITS	Int16
NUM_PREC_RADIX	Int16
NULLABLE	Int16
REMARKS	String
COLUMN_DEF	String

COLUMNNAME	DATATYPE
SQL_DATA_TYPE	Int16
SQL_DATETIME_SUB	Int16
CHAR_OCTET_LENGTH	Int32
ORDINAL_POSITION	Int32
IS_NULLABLE	String
SS_TYPE_CATALOG	String
SS_TYPE_SCHEMA	String
SS_DATA_TYPE	Byte

Procedures

COLUMNNAME	DATATYPE
PROCEDURE_CAT	String
PROCEDURE_SCHEM	String
PROCEDURE_NAME	String
NUM_INPUT_PARAMS	Int32
NUM_OUTPUT_PARAMS	Int32
NUM_RESULT_SETS	Int32
REMARKS	String
PROCEDURE_TYPE	Int16

ProcedureColumns

COLUMNNAME	DATATYPE
PROCEDURE_CAT	String
PROCEDURE_SCHEM	String
PROCEDURE_NAME	String
COLUMN_NAME	String
COLUMN_TYPE	Int16
DATA_TYPE	Int16

COLUMNNAME	DATATYPE
TYPE_NAME	String
COLUMN_SIZE	Int32
BUFFER_LENGTH	Int32
DECIMAL_DIGITS	Int16
NUM_PREC_RADIX	Int16
NULLABLE	Int16
REMARKS	String
COLUMN_DEF	String
SQL_DATA_TYPE	Int16
SQL_DATETIME_SUB	Int16
CHAR_OCTET_LENGTH	Int32
ORDINAL_POSITION	Int32
IS_NULLABLE	String
SS_TYPE_CATALOG	String
SS_TYPE_SCHEMA	String
SS_DATA_TYPE	Byte

ProcedureParameters

COLUMNNAME	DATATYPE
PROCEDURE_CAT	String
PROCEDURE_SCHEM	String
PROCEDURE_NAME	String
COLUMN_NAME	String
COLUMN_TYPE	Int16
DATA_TYPE	Int16
TYPE_NAME	String

COLUMNNAME	DATATYPE
COLUMN_SIZE	Int32
BUFFER_LENGTH	Int32
DECIMAL_DIGITS	Int16
NUM_PREC_RADIX	Int16
NULLABLE	Int16
REMARKS	String
COLUMN_DEF	String
SQL_DATA_TYPE	Int16
SQL_DATETIME_SUB	Int16
CHAR_OCTET_LENGTH	Int32
ORDINAL_POSITION	Int32
IS_NULLABLE	String
SS_TYPE_CATALOG	String
SS_TYPE_SCHEMA	String
SS_DATA_TYPE	Byte

Microsoft Oracle ODBC Driver

The Microsoft SQL Server Oracle ODBC Driver supports the following specific schema collections in addition to the common schema collections:

- Tables
- Columns
- Procedures
- ProcedureColumns
- ProcedureParameters
- Views
- Indexes

Tables and Views

COLUMNNAME	DATATYPE
TABLE_QUALIFIER	String
TABLE_OWNER	String
TABLE_NAME	String
TABLE_TYPE	String
REMARKS	String

Columns

COLUMNNAME	DATATYPE
TABLE_QUALIFIER	String
TABLE_OWNER	String
TABLE_NAME	String
COLUMN_NAME	String
DATA_TYPE	Int16
TYPE_NAME	String
PRECISION	Int32
LENGTH	Int32
SCALE	Int16
RADIX	Int16
NULLABLE	Int16
REMARKS	String
ORDINAL_POSITION	Int32

Procedures

COLUMNNAME	DATATYPE
PROCEDURE_QUALIFIER	String
PROCEDURE_OWNER	String
PROCEDURE_NAME	String
NUM_INPUT_PARAMS	Int16

COLUMNNAME	DATATYPE
NUM_OUTPUT_PARAMS	Int16
NUM_RESULT_SETS	Int16
REMARKS	String
PROCEDURE_TYPE	Int16

ProcedureColumns

COLUMNNAME	DATATYPE
PROCEDURE_QUALIFIER	String
PROCEDURE_OWNER	String
PROCEDURE_NAME	String
COLUMN_NAME	String
COLUMN_TYPE	Int16
DATA_TYPE	Int16
TYPE_NAME	String
PRECISION	Int32
LENGTH	Int32
SCALE	Int16
RADIX	Int16
NULLABLE	Int16
REMARKS	String
OVERLOAD	Int32
ORDINAL_POSITION	Int32

Microsoft Jet ODBC Driver

The Microsoft Jet ODBC Driver supports the following specific schema collections in addition to the common schema collections:

- Tables
- Indexes
- Columns

- Procedures
- ProcedureColumns
- ProcedureParameters
- Views

Tables and Views

COLUMNNAME	DATATYPE
TABLE_QUALIFIER	String
TABLE_OWNER	String
TABLE_NAME	String
TABLE_TYPE	String
REMARKS	String

Columns

COLUMNNAME	DATATYPE
TABLE_QUALIFIER	String
TABLE_OWNER	String
TABLE_NAME	String
COLUMN_NAME	String
DATA_TYPE	Int16
TYPE_NAME	String
PRECISION	Int32
LENGTH	Int32
SCALE	Int16
RADIX	Int16
NULLABLE	Int16
REMARKS	String
ORDINAL_POSITION	Int32

Procedures

COLUMNNAME	DATATYPE
PROCEDURE_QUALIFIER	String
PROCEDURE_OWNER	String
PROCEDURE_NAME	String
NUM_INPUT_PARAMS	Int16
NUM_OUTPUT_PARAMS	Int16
NUM_RESULT_SETS	Int16
REMARKS	String
PROCEDURE_TYPE	Int16

ProcedureColumns

COLUMNNAME	DATATYPE
PROCEDURE_QUALIFIER	String
PROCEDURE_OWNER	String
PROCEDURE_NAME	String
COLUMN_NAME	String
COLUMN_TYPE	Int16
DATA_TYPE	Int16
TYPE_NAME	String
PRECISION	Int32
LENGTH	Int32
SCALE	Int16
RADIX	Int16
NULLABLE	Int16
REMARKS	String
OVERLOAD	Int32
ORDINAL_POSITION	Int32

ProcedureParameters

COLUMNNAME	DATATYPE
PROCEDURE_CAT	String
PROCEDURE_SCHEM	String
PROCEDURE_NAME	String
COLUMN_NAME	String
COLUMN_TYPE	Int16
DATA_TYPE	Int16
TYPE_NAME	String
COLUMN_SIZE	Int32
BUFFER_LENGTH	Int32
DECIMAL_DIGITS	Int16
NUM_PREC_RADIX	Int16
NULLABLE	Int16
REMARKS	String
COLUMN_DEF	String
SQL_DATA_TYPE	Int16
SQL_DATETIME_SUB	Int16
CHAR_OCTET_LENGTH	Int32
ORDINAL_POSITION	Int32
IS_NULLABLE	String

See also

- ADO.NET Overview

OLE DB Schema Collections

9/7/2019 • 3 minutes to read • Edit Online

This section discusses schema collection support for the OLE DB providers for Microsoft SQL Server, Oracle, and Microsoft Jet.

Microsoft SQL Server OLE DB Provider

The Microsoft SQL Server OLE DB Driver supports the following specific schema collections in addition to the common schema collections:

- Tables
- Columns
- Procedures
- ProcedureParameters
- Catalog
- Indexes

Tables

COLUMNNAME	DATATYPE
TABLE_CATALOG	String
TABLE_SCHEMA	String
TABLE_NAME	String
TABLE_TYPE	String
TABLE_GUID	Guid
DESCRIPTION	String
TABLE_PROPID	Int64
DATE_CREATED	DateTime
DATE_MODIFIED	DateTime

Columns

COLUMNNAME	DATATYPE
TABLE_CATALOG	String
TABLE_SCHEMA	String

COLUMNNAME	DATATYPE
TABLE_NAME	String
COLUMN_NAME	String
COLUMN_GUID	Guid
COLUMN_PROPID	Int64
ORDINAL_POSITION	Int64
COLUMN_HASDEFAULT	Boolean
COLUMN_DEFAULT	String
COLUMN_FLAGS	Int64
IS_NULLABLE	Boolean
DATA_TYPE	Int32
TYPE_GUID	Guid
CHARACTER_MAXIMUM_LENGTH	Int64
CHARACTER_OCTET_LENGTH	Int64
NUMERIC_PRECISION	Int32
NUMERIC_SCALE	Int16
DATETIME_PRECISION	Int64
CHARACTER_SET_CATALOG	String
CHARACTER_SET_SCHEMA	String
CHARACTER_SET_NAME	String
COLLATION_CATALOG	String
COLLATION_SCHEMA	String
COLLATION_NAME	String
DOMAIN_CATALOG	String
DOMAIN_SCHEMA	String
DOMAIN_NAME	String

COLUMNNAME	DATATYPE
DESCRIPTION	String
COLUMN_LCID	Int32
COLUMN_COMPFLAGS	Int32
COLUMN_SORTID	Int32
COLUMN_TDSCOLLATION	Byte[]
IS_COMPUTED	Boolean

Procedures

COLUMNNAME	DATATYPE
PROCEDURE_CATALOG	String
PROCEDURE_SCHEMA	String
PROCEDURE_NAME	String
PROCEDURE_TYPE	Int16
PROCEDURE_DEFINITION	String
DESCRIPTION	String
DATE_CREATED	DateTime
DATE_MODIFIED	DateTime

ProcedureParameters

COLUMNNAME	DATATYPE
PROCEDURE_CATALOG	String
PROCEDURE_SCHEMA	String
PROCEDURE_NAME	String
PARAMETER_NAME	String
ORDINAL_POSITION	Int32
PARAMETER_TYPE	Int32
PARAMETER_HASDEFAULT	Boolean
PARAMETER_DEFAULT	String

COLUMNNAME	DATATYPE
IS_NULLABLE	Boolean
DATA_TYPE	Int32
CHARACTER_MAXIMUM_LENGTH	Int64
CHARACTER_OCTET_LENGTH	Int64
NUMERIC_PRECISION	Int32
NUMERIC_SCALE	Int16
DESCRIPTION	String
TYPE_NAME	String
LOCAL_TYPE_NAME	String

Catalog

COLUMNNAME	DATATYPE
CATALOG_NAME	String
DESCRIPTION	String

Indexes

COLUMNNAME	DATATYPE
TABLE_CATALOG	String
TABLE_SCHEMA	String
TABLE_NAME	String
INDEX_CATALOG	String
INDEX_SCHEMA	String
INDEX_NAME	String
PRIMARY_KEY	Boolean
UNIQUE	Boolean
CLUSTERED	Boolean
TYPE	Int32
FILL_FACTOR	Int32

COLUMNNAME	DATATYPE
INITIAL_SIZE	Int32
NULLS	Int32
SORT_BOOKMARKS	Boolean
AUTO_UPDATE	Boolean
NULL_COLLATION	Int32
ORDINAL_POSITION	Int64
COLUMN_NAME	String
COLUMN_GUID	Guid
COLUMN_PROPID	Int64
COLLATION	Int16
CARDINALITY	Decimal
PAGES	Int32
FILTER_CONDITION	String
INTEGRATED	Boolean

Microsoft Oracle OLE DB Provider

The Microsoft Oracle OLE DB Driver supports the following specific schema collections in addition to the common schema collections:

- Tables
- Columns
- Procedures
- ProcedureColumns
- ProcedureParameters
- Views
- Indexes

Tables

COLUMNNAME	DATATYPE
TABLE_CATALOG	String

COLUMNNAME	DATATYPE
TABLE_SCHEMA	String
TABLE_NAME	String
TABLE_TYPE	String
TABLE_GUID	Guid
DESCRIPTION	String
TABLE_PROPID	Int64
DATE_CREATED	DateTime
DATE_MODIFIED	DateTime

Columns

COLUMNNAME	DATATYPE
TABLE_CATALOG	String
TABLE_SCHEMA	String
TABLE_NAME	String
COLUMN_NAME	String
COLUMN_GUID	Guid
COLUMN_PROPID	Int64
ORDINAL_POSITION	Int64
COLUMN_HASDEFAULT	Boolean
COLUMN_DEFAULT	String
COLUMN_FLAGS	Int64
IS_NULLABLE	Boolean
DATA_TYPE	Int32
TYPE_GUID	Guid
CHARACTER_MAXIMUM_LENGTH	Int64
CHARACTER_OCTET_LENGTH	Int64

COLUMNNAME	DATATYPE
NUMERIC_PRECISION	Int32
NUMERIC_SCALE	Int16
DATETIME_PRECISION	Int64
CHARACTER_SET_CATALOG	String
CHARACTER_SET_SCHEMA	String
CHARACTER_SET_NAME	String
COLLATION_CATALOG	String
COLLATION_SCHEMA	String
COLLATION_NAME	String
DOMAIN_CATALOG	String
DOMAIN_SCHEMA	String
DOMAIN_NAME	String
DESCRIPTION	String

Procedures

COLUMNNAME	DATATYPE
PROCEDURE_CATALOG	String
PROCEDURE_SCHEMA	String
PROCEDURE_NAME	String
PROCEDURE_TYPE	Int16
PROCEDURE_DEFINITION	String
DESCRIPTION	String
DATE_CREATED	DateTime
DATE_MODIFIED	DateTime

ProcedureColumns

COLUMNNAME	DATATYPE
PROCEDURE_CATALOG	String

COLUMNNAME	DATATYPE
PROCEDURE_SCHEMA	String
PROCEDURE_NAME	String
COLUMN_NAME	String
COLUMN_GUID	Guid
COLUMN_PROPID	Int64
ROWSET_NUMBER	Int64
ORDINAL_POSITION	Int64
IS_NULLABLE	Boolean
DATA_TYPE	Int32
TYPE_GUID	Guid
CHARACTER_MAXIMUM_LENGTH	Int64
CHARACTER_OCTET_LENGTH	Int64
NUMERIC_PRECISION	Int32
NUMERIC_SCALE	Int16
DESCRIPTION	String
OVERLOAD	Int16

Views

COLUMNNAME	DATATYPE
TABLE_CATALOG	String
TABLE_SCHEMA	String
TABLE_NAME	String
VIEW_DEFINITION	String
CHECK_OPTION	Boolean
IS_UPDATABLE	Boolean
DESCRIPTION	String

COLUMNNAME	DATATYPE
DATE_CREATED	DateTime
DATE_MODIFIED	DateTime

Indexes

COLUMNNAME	DATATYPE
TABLE_CATALOG	String
TABLE_SCHEMA	String
TABLE_NAME	String
INDEX_CATALOG	String
INDEX_SCHEMA	String
INDEX_NAME	String
PRIMARY_KEY	Boolean
UNIQUE	Boolean
CLUSTERED	Boolean
TYPE	Int32
FILL_FACTOR	Int32
INITIAL_SIZE	Int32
NULLS	Int32
SORT_BOOKMARKS	Boolean
AUTO_UPDATE	Boolean
NULL_COLLATION	Int32
ORDINAL_POSITION	Int64
COLUMN_NAME	String
COLUMN_GUID	Guid
COLUMN_PROPID	Int64
COLLATION	Int16

COLUMNNAME	DATATYPE
CARDINALITY	Decimal
PAGES	Int32
FILTER_CONDITION	String
INTEGRATED	Boolean

Microsoft Jet OLE DB Provider

The Microsoft Jet OLE DB Driver supports the following specific schema collections in addition to the common schema collections:

- Tables
- Columns
- Procedures
- Views
- Indexes

Tables

COLUMNNAME	DATATYPE
TABLE_CATALOG	String
TABLE_SCHEMA	String
TABLE_NAME	String
TABLE_TYPE	String
TABLE_GUID	Guid
DESCRIPTION	String
TABLE_PROPID	Int64
DATE_CREATED	DateTime
DATE_MODIFIED	DateTime

Columns

COLUMNNAME	DATATYPE
TABLE_CATALOG	String
TABLE_SCHEMA	String

COLUMNNAME	DATATYPE
TABLE_NAME	String
COLUMN_NAME	String
COLUMN_GUID	Guid
COLUMN_PROPID	Int64
ORDINAL_POSITION	Int64
COLUMN_HASDEFAULT	Boolean
COLUMN_DEFAULT	String
COLUMN_FLAGS	Int64
IS_NULLABLE	Boolean
DATA_TYPE	Int32
TYPE_GUID	Guid
CHARACTER_MAXIMUM_LENGTH	Int64
CHARACTER_OCTET_LENGTH	Int64
NUMERIC_PRECISION	Int32
NUMERIC_SCALE	Int16
DATETIME_PRECISION	Int64
CHARACTER_SET_CATALOG	String
CHARACTER_SET_SCHEMA	String
CHARACTER_SET_NAME	String
COLLATION_CATALOG	String
COLLATION_SCHEMA	String
COLLATION_NAME	String
DOMAIN_CATALOG	String
DOMAIN_SCHEMA	String
DOMAIN_NAME	String

COLUMNNAME	DATATYPE
DESCRIPTION	String

Procedures

COLUMNNAME	DATATYPE
PROCEDURE_CATALOG	String
PROCEDURE_SCHEMA	String
PROCEDURE_NAME	String
PROCEDURE_TYPE	Int16
PROCEDURE_DEFINITION	String
DESCRIPTION	String
DATE_CREATED	DateTime
DATE_MODIFIED	DateTime

Views

COLUMNNAME	DATATYPE
TABLE_CATALOG	String
TABLE_SCHEMA	String
TABLE_NAME	String
VIEW_DEFINITION	String
CHECK_OPTION	Boolean
IS_UPDATABLE	Boolean
DESCRIPTION	String
DATE_CREATED	DateTime
DATE_MODIFIED	DateTime

Indexes

COLUMNNAME	DATATYPE
TABLE_CATALOG	String
TABLE_SCHEMA	String

COLUMNNAME	DATATYPE
TABLE_NAME	String
INDEX_CATALOG	String
INDEX_SCHEMA	String
INDEX_NAME	String
PRIMARY_KEY	Boolean
UNIQUE	Boolean
CLUSTERED	Boolean
TYPE	Int32
FILL_FACTOR	Int32
INITIAL_SIZE	Int32
NULLS	Int32
SORT_BOOKMARKS	Boolean
AUTO_UPDATE	Boolean
NULL_COLLATION	Int32
ORDINAL_POSITION	Int64
COLUMN_NAME	String
COLUMN_GUID	Guid
COLUMN_PROPID	Int64
COLLATION	Int16
CARDINALITY	Decimal
PAGES	Int32
FILTER_CONDITION	String
INTEGRATED	Boolean

See also

- ADO.NET Overview

DbProviderFactories

9/7/2019 • 2 minutes to read • Edit Online

The System.Data.Common namespace provides classes for creating DbProviderFactory instances to work with specific data sources. When you create a DbProviderFactory instance and pass it information about the data provider, the `DbProviderFactory` can determine the correct, strongly typed connection object to return based on the information it has been provided.

Beginning in the .NET Framework version 4, data providers such as System.Data.Odbc, System.Data.OleDb, System.Data.SqlClient, and System.Data.OracleClient are no longer listed in machine.config file, but custom providers will continue to be listed there.

In This Section

Factory Model Overview
Provides an overview of the factory design pattern and programming interface.

Obtaining a DbProviderFactory
Demonstrates how to list the installed data providers and create a DbConnection from a `DbProviderFactory`.

DbConnection, DbCommand and DbException
Demonstrates how to create a DbCommand and DbDataReader, and how to handle data errors using DbException.

Modifying Data with a DbDataAdapter
Demonstrates how to use a DbCommandBuilder with a DbDataAdapter to retrieve and modify data.

See also

- Retrieving and Modifying Data in ADO.NET
- ADO.NET Overview

Factory Model Overview

12/24/2019 • 2 minutes to read • Edit Online

ADO.NET 2.0 introduced new base classes in the System.Data.Common namespace. The base classes are abstract, which means that they can't be directly instantiated. They include DbConnection, DbCommand, and DbDataAdapter and are shared by the .NET Framework data providers, such as System.Data.SqlClient and System.Data.OleDb. The addition of base classes simplifies adding functionality to the .NET Framework data providers without having to create new interfaces.

ADO.NET 2.0 also introduced abstract base classes, which enable a developer to write generic data access code that does not depend on a specific data provider.

The Factory Design Pattern

The programming model for writing provider-independent code is based on the use of the "factory" design pattern, which uses a single API to access databases across multiple providers. This pattern is aptly named, as it calls for the use of a specialized object solely to create other objects, much like a real-world factory. For a more detailed description of the factory design pattern, see Writing Generic Data Access Code in ASP.NET 2.0 and ADO.NET 2.0.

Starting with ADO.NET 2.0, the DbProviderFactories class provides `static` (or `Shared` in Visual Basic) methods for creating a DbProviderFactory instance. The instance then returns a correct strongly typed object based on provider information and the connection string supplied at run time.

See also

- Obtaining a DbProviderFactory
- DbConnection, DbCommand and DbException
- Modifying Data with a DbDataAdapter
- ADO.NET Overview

Obtaining a DbProviderFactory

3/12/2020 • 5 minutes to read • Edit Online

The process of obtaining a DbProviderFactory involves passing information about a data provider to the DbProviderFactories class. Based on this information, the GetFactory method creates a strongly typed provider factory. For example, to create a SqlClientFactory, you can pass `GetFactory` a string with the provider name specified as "System.Data.SqlClient". The other overload of `GetFactory` takes a DataRow. Once you create the provider factory, you can then use its methods to create additional objects. Some of the methods of a `SqlClientFactory` include CreateConnection, CreateCommand, and CreateDataAdapter.

> **NOTE**
>
> The .NET Framework OracleClientFactory, OdbcFactory, and OleDbFactory classes also provide similar functionality.

Registering DbProviderFactories

Each .NET Framework data provider that supports a factory-based class registers configuration information in the **DbProviderFactories** section of the **machine.config** file on the local computer. The following configuration file fragment shows the syntax and format for System.Data.SqlClient.

```
<system.data>
  <DbProviderFactories>
    <add name="SqlClient Data Provider"
     invariant="System.Data.SqlClient"
     description=".Net Framework Data Provider for SqlServer"
     type="System.Data.SqlClient.SqlClientFactory, System.Data,
     Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
    />
  </DbProviderFactories>
</system.data>
```

The **invariant** attribute identifies the underlying data provider. This three-part naming syntax is also used when creating a new factory and for identifying the provider in an application configuration file so that the provider name, along with its associated connection string, can be retrieved at run time.

Retrieving Provider Information

You can retrieve information about all of the data providers installed on the local computer by using the GetFactoryClasses method. It returns a DataTable named **DbProviderFactories** that contains the columns described in the following table.

COLUMN ORDINAL	COLUMN NAME	EXAMPLE OUTPUT	DESCRIPTION
0	**Name**	SqlClient Data Provider	Readable name for the data provider
1	**Description**	.Net Framework Data Provider for SqlServer	Readable description of the data provider

COLUMN ORDINAL	COLUMN NAME	EXAMPLE OUTPUT	DESCRIPTION
2	**InvariantName**	System.Data.SqlClient	Name that can be used programmatically to refer to the data provider
3	**AssemblyQualifiedName**	System.Data.SqlClient.SqlClientFactory, System.Data, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089	Fully qualified name of the factory class, which contains enough information to instantiate the object

This `DataTable` can be used to enable a user to select a DataRow at run time. The selected `DataRow` can then be passed to the GetFactory method to create a strongly typed DbProviderFactory. A selected DataRow can be passed to the `GetFactory` method to create the desired `DbProviderFactory` object.

Listing the Installed Provider Factory Classes

This example demonstrates how to use the GetFactoryClasses method to return a DataTable containing information about the installed providers. The code iterates through each row in the `DataTable`, displaying information for each installed provider in the console window.

```
// This example assumes a reference to System.Data.Common.
static DataTable GetProviderFactoryClasses()
{
    // Retrieve the installed providers and factories.
    DataTable table = DbProviderFactories.GetFactoryClasses();

    // Display each row and column value.
    foreach (DataRow row in table.Rows)
    {
        foreach (DataColumn column in table.Columns)
        {
            Console.WriteLine(row[column]);
        }
    }
    return table;
}
```

```
' This example assumes a reference to System.Data.Common.
Private Shared Function GetProviderFactoryClasses() As DataTable

    ' Retrieve the installed providers and factories.
    Dim table As DataTable = DbProviderFactories.GetFactoryClasses()

    ' Display each row and column value.
    Dim row As DataRow
    Dim column As DataColumn
    For Each row In table.Rows
        For Each column In table.Columns
            Console.WriteLine(row(column))
        Next
    Next

    Return table
End Function
```

Using Application Configuration Files to Store Factory Information

The design pattern used for working with factories entails storing provider and connection string information in an application configuration file, such as **app.config** for a Windows application, and **web.config** for an ASP.NET application.

The following configuration file fragment demonstrates how to save two named connection strings, "NorthwindSQL" for a connection to the Northwind database in SQL Server, and "NorthwindAccess" for a connection to the Northwind database in Access/Jet. The **invariant** name is used for the **providerName** attribute.

```
<configuration>
  <connectionStrings>
    <clear/>
    <add name="NorthwindSQL"
     providerName="System.Data.SqlClient"
     connectionString=
     "Data Source=MSSQL1;Initial Catalog=Northwind;Integrated Security=true"
    />

    <add name="NorthwindAccess"
     providerName="System.Data.OleDb"
     connectionString=
     "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\Data\Northwind.mdb;"
    />
  </connectionStrings>
</configuration>
```

Retrieving a Connection String by Provider Name

In order to create a provider factory, you must supply a connection string as well as the provider name. This example demonstrates how to retrieve a connection string from an application configuration file by passing the provider name in the invariant format "*System.Data.ProviderName*". The code iterates through the ConnectionStringSettingsCollection. It returns the ProviderName on success; otherwise `null` (`Nothing` in Visual Basic). If there are multiple entries for a provider, the first one found is returned. For more information and examples of retrieving connection strings from configuration files, see Connection Strings and Configuration Files.

> **NOTE**
>
> A reference to `System.Configuration.dll` is required in order for the code to run.

```
// Retrieve a connection string by specifying the providerName.
// Assumes one connection string per provider in the config file.
static string GetConnectionStringByProvider(string providerName)
{
    // Return null on failure.
    string returnValue = null;

    // Get the collection of connection strings.
    ConnectionStringSettingsCollection settings =
        ConfigurationManager.ConnectionStrings;

    // Walk through the collection and return the first
    // connection string matching the providerName.
    if (settings != null)
    {
        foreach (ConnectionStringSettings cs in settings)
        {
            if (cs.ProviderName == providerName)
                returnValue = cs.ConnectionString;
            break;
        }
    }
    return returnValue;
}
```

```
' Retrieve a connection string by specifying the providerName.
' Assumes one connection string per provider in the config file.
Private Shared Function GetConnectionStringByProvider( _
    ByVal providerName As String) As String

    'Return Nothing on failure.
    Dim returnValue As String = Nothing

    ' Get the collection of connection strings.
    Dim settings As ConnectionStringSettingsCollection = _
        ConfigurationManager.ConnectionStrings

    ' Walk through the collection and return the first
    ' connection string matching the providerName.
    If Not settings Is Nothing Then
        For Each cs As ConnectionStringSettings In settings
            If cs.ProviderName = providerName Then
                returnValue = cs.ConnectionString
                Exit For
            End If
        Next
    End If

    Return returnValue
End Function
```

Creating the DbProviderFactory and DbConnection

This example demonstrates how to create a DbProviderFactory and DbConnection object by passing it the provider name in the format "*System.Data.ProviderName*" and a connection string. A `DbConnection` object is returned on success; `null` (`Nothing` in Visual Basic) on any error.

The code obtains the `DbProviderFactory` by calling GetFactory. Then the CreateConnection method creates the DbConnection object and the ConnectionString property is set to the connection string.

```
// Given a provider name and connection string,
// create the DbProviderFactory and DbConnection.
// Returns a DbConnection on success; null on failure.
static DbConnection CreateDbConnection(
    string providerName, string connectionString)
{
    // Assume failure.
    DbConnection connection = null;

    // Create the DbProviderFactory and DbConnection.
    if (connectionString != null)
    {
        try
        {
            DbProviderFactory factory =
                DbProviderFactories.GetFactory(providerName);

            connection = factory.CreateConnection();
            connection.ConnectionString = connectionString;
        }
        catch (Exception ex)
        {
            // Set the connection to null if it was created.
            if (connection != null)
            {
                connection = null;
            }
            Console.WriteLine(ex.Message);
        }
    }
    // Return the connection.
    return connection;
}
```

```
' Given a provider, create a DbProviderFactory and DbConnection.
' Returns a DbConnection on success; Nothing on failure.
Private Shared Function CreateDbConnection( _
    ByVal providerName As String, ByVal connectionString As String) _
    As DbConnection

    ' Assume failure.
    Dim connection As DbConnection = Nothing

    ' Create the DbProviderFactory and DbConnection.
    If Not connectionString Is Nothing Then
        Try
            Dim factory As DbProviderFactory = _
               DbProviderFactories.GetFactory(providerName)

            connection = factory.CreateConnection()
            connection.ConnectionString = connectionString

        Catch ex As Exception
            ' Set the connection to Nothing if it was created.
            If Not connection Is Nothing Then
                connection = Nothing
            End If
            Console.WriteLine(ex.Message)
        End Try
    End If

    ' Return the connection.
    Return connection
End Function
```

See also

- DbProviderFactories
- Connection Strings
- Using the Configuration Classes
- ADO.NET Overview

DbConnection, DbCommand and DbException

9/7/2019 • 4 minutes to read • Edit Online

Once you have created a DbProviderFactory and a DbConnection, you can then work with commands and data readers to retrieve data from the data source.

Retrieving Data Example

This example takes a `DbConnection` object as an argument. A DbCommand is created to select data from the Categories table by setting the CommandText to a SQL SELECT statement. The code assumes that the Categories table exists at the data source. The connection is opened and the data is retrieved using a DbDataReader.

```
// Takes a DbConnection and creates a DbCommand to retrieve data
// from the Categories table by executing a DbDataReader.
static void DbCommandSelect(DbConnection connection)
{
    string queryString =
        "SELECT CategoryID, CategoryName FROM Categories";

    // Check for valid DbConnection.
    if (connection != null)
    {
        using (connection)
        {
            try
            {
                // Create the command.
                DbCommand command = connection.CreateCommand();
                command.CommandText = queryString;
                command.CommandType = CommandType.Text;

                // Open the connection.
                connection.Open();

                // Retrieve the data.
                DbDataReader reader = command.ExecuteReader();
                while (reader.Read())
                {
                    Console.WriteLine("{0}. {1}", reader[0], reader[1]);
                }
            }

            catch (Exception ex)
            {
                Console.WriteLine("Exception.Message: {0}", ex.Message);
            }
        }
    }
    else
    {
        Console.WriteLine("Failed: DbConnection is null.");
    }
}
```

```
' Takes a DbConnection and creates a DbCommand to retrieve data
' from the Categories table by executing a DbDataReader.
Private Shared Sub DbCommandSelect(ByVal connection As DbConnection)

    Dim queryString As String = _
       "SELECT CategoryID, CategoryName FROM Categories"

    ' Check for valid DbConnection.
    If Not connection Is Nothing Then
        Using connection
            Try
                ' Create the command.
                Dim command As DbCommand = connection.CreateCommand()
                command.CommandText = queryString
                command.CommandType = CommandType.Text

                ' Open the connection.
                connection.Open()

                ' Retrieve the data.
                Dim reader As DbDataReader = command.ExecuteReader()
                Do While reader.Read()
                    Console.WriteLine("{0}. {1}", reader(0), reader(1))
                Loop

            Catch ex As Exception
                Console.WriteLine("Exception.Message: {0}", ex.Message)
            End Try
        End Using
    Else
        Console.WriteLine("Failed: DbConnection is Nothing.")
    End If
End Sub
```

Executing a Command Example

This example takes a `DbConnection` object as an argument. If the `DbConnection` is valid, the connection is opened and a DbCommand is created and executed. The CommandText is set to a SQL INSERT statement that performs an insert to the Categories table in the Northwind database. The code assumes that the Northwind database exists at the data source, and that the SQL syntax used in the INSERT statement is valid for the specified provider. Errors occurring at the data source are handled by the DbException code block, and all other exceptions are handled in the Exception block.

```
// Takes a DbConnection, creates and executes a DbCommand.
// Assumes SQL INSERT syntax is supported by provider.
static void ExecuteDbCommand(DbConnection connection)
{
    // Check for valid DbConnection object.
    if (connection != null)
    {
        using (connection)
        {
            try
            {
                // Open the connection.
                connection.Open();

                // Create and execute the DbCommand.
                DbCommand command = connection.CreateCommand();
                command.CommandText =
                    "INSERT INTO Categories (CategoryName) VALUES ('Low Carb')";
                int rows = command.ExecuteNonQuery();

                // Display number of rows inserted.
                Console.WriteLine("Inserted {0} rows.", rows);
            }
                // Handle data errors.
            catch (DbException exDb)
            {
                Console.WriteLine("DbException.GetType: {0}", exDb.GetType());
                Console.WriteLine("DbException.Source: {0}", exDb.Source);
                Console.WriteLine("DbException.ErrorCode: {0}", exDb.ErrorCode);
                Console.WriteLine("DbException.Message: {0}", exDb.Message);
            }
                // Handle all other exceptions.
            catch (Exception ex)
            {
                Console.WriteLine("Exception.Message: {0}", ex.Message);
            }
        }
    }
    else
    {
        Console.WriteLine("Failed: DbConnection is null.");
    }
}
```

```
' Takes a DbConnection and executes an INSERT statement.
' Assumes SQL INSERT syntax is supported by provider.
Private Shared Sub ExecuteDbCommand(ByVal connection As DbConnection)

    ' Check for valid DbConnection object.
    If Not connection Is Nothing Then
        Using connection
            Try
                ' Open the connection.
                connection.Open()

                ' Create and execute the DbCommand.
                Dim command As DbCommand = connection.CreateCommand()
                command.CommandText = _
                  "INSERT INTO Categories (CategoryName) VALUES ('Low Carb')"
                Dim rows As Integer = command.ExecuteNonQuery()

                ' Display number of rows inserted.
                Console.WriteLine("Inserted {0} rows.", rows)

            ' Handle data errors.
            Catch exDb As DbException
                Console.WriteLine("DbException.GetType: {0}", exDb.GetType())
                Console.WriteLine("DbException.Source: {0}", exDb.Source)
                Console.WriteLine("DbException.ErrorCode: {0}", exDb.ErrorCode)
                Console.WriteLine("DbException.Message: {0}", exDb.Message)

            ' Handle all other exceptions.
            Catch ex As Exception
                Console.WriteLine("Exception.Message: {0}", ex.Message)
            End Try
        End Using
    Else
        Console.WriteLine("Failed: DbConnection is Nothing.")
    End If
End Sub
```

Handling Data Errors with DbException

The DbException class is the base class for all exceptions thrown on behalf of a data source. You can use it in your exception handling code to handle exceptions thrown by different providers without having to reference a specific exception class. The following code fragment demonstrates how to use DbException to display error information returned by the data source using GetType, Source, ErrorCode, and Message properties. The output will display the type of error, the source indicating the provider name, an error code, and the message associated with the error.

```
Try
    ' Do work here.
Catch ex As DbException
    ' Display information about the exception.
    Console.WriteLine("GetType: {0}", ex.GetType())
    Console.WriteLine("Source: {0}", ex.Source)
    Console.WriteLine("ErrorCode: {0}", ex.ErrorCode)
    Console.WriteLine("Message: {0}", ex.Message)
Finally
    ' Perform cleanup here.
End Try
```

```
try
{
    // Do work here.
}
catch (DbException ex)
{
    // Display information about the exception.
    Console.WriteLine("GetType: {0}", ex.GetType());
    Console.WriteLine("Source: {0}", ex.Source);
    Console.WriteLine("ErrorCode: {0}", ex.ErrorCode);
    Console.WriteLine("Message: {0}", ex.Message);
}
finally
{
    // Perform cleanup here.
}
```

See also

- DbProviderFactories
- Obtaining a DbProviderFactory
- Modifying Data with a DbDataAdapter
- ADO.NET Overview

Modifying Data with a DbDataAdapter

9/7/2019 • 5 minutes to read • Edit Online

The CreateDataAdapter method of a DbProviderFactory object gives you a DbDataAdapter object that is strongly typed to the underlying data provider specified at the time you create the factory. You can then use a DbCommandBuilder to create commands to insert, update, and delete data from a DataSet to a data source.

Retrieving Data with a DbDataAdapter

This example demonstrates how to create a strongly typed `DbDataAdapter` based on a provider name and connection string. The code uses the CreateConnection method of the DbProviderFactory to create a DbConnection. Next, the code uses the CreateCommand method to create a DbCommand to select data by setting its `CommandText` and `Connection` properties. Finally, the code creates a DbDataAdapter object using the CreateDataAdapter method and sets its `SelectCommand` property. The `Fill` method of the `DbDataAdapter` loads the data into a DataTable.

```
static void CreateDataAdapter(string providerName, string connectionString)
{
    try
    {
        // Create the DbProviderFactory and DbConnection.
        DbProviderFactory factory =
            DbProviderFactories.GetFactory(providerName);

        DbConnection connection = factory.CreateConnection();
        connection.ConnectionString = connectionString;

        using (connection)
        {
            // Define the query.
            string queryString =
                "SELECT CategoryName FROM Categories";

            // Create the DbCommand.
            DbCommand command = factory.CreateCommand();
            command.CommandText = queryString;
            command.Connection = connection;

            // Create the DbDataAdapter.
            DbDataAdapter adapter = factory.CreateDataAdapter();
            adapter.SelectCommand = command;

            // Fill the DataTable.
            DataTable table = new DataTable();
            adapter.Fill(table);

            //  Display each row and column value.
            foreach (DataRow row in table.Rows)
            {
                foreach (DataColumn column in table.Columns)
                {
                    Console.WriteLine(row[column]);
                }
            }
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
}
```

```
Shared Sub CreateDataAdapter(ByVal providerName As String, _
    ByVal connectionString As String)

    ' Create the DbProviderFactory and DbConnection.
    Try
        Dim factory As DbProviderFactory = _
           DbProviderFactories.GetFactory(providerName)

        Dim connection As DbConnection = _
            factory.CreateConnection()
        connection.ConnectionString = connectionString
        Using connection

            ' Define the query.
            Dim queryString As String = _
              "SELECT CategoryName FROM Categories"

            'Create the DbCommand.
            Dim command As DbCommand = _
                factory.CreateCommand()
            command.CommandText = queryString
            command.Connection = connection

            ' Create the DbDataAdapter.
            Dim adapter As DbDataAdapter = _
                factory.CreateDataAdapter()
            adapter.SelectCommand = command

            ' Fill the DataTable
            Dim table As New DataTable
            adapter.Fill(table)

            'Display each row and column value.
            Dim row As DataRow
            Dim column As DataColumn
            For Each row In table.Rows
                For Each column In table.Columns
                    Console.WriteLine(row(column))
                Next
            Next
        End Using

    Catch ex As Exception
        Console.WriteLine(ex.Message)
    End Try
End Sub
```

Modifying Data with a DbDataAdapter

This example demonstrates how to modify data in a `DataTable` using a DbDataAdapter by using a DbCommandBuilder to generate the commands required for updating data at the data source. The SelectCommand of the `DbDataAdapter` is set to retrieve the CustomerID and CompanyName from the Customers table. The GetInsertCommand method is used to set the InsertCommand property, the GetUpdateCommand method is used to set the UpdateCommand property, and the GetDeleteCommand method is used to set the DeleteCommand property. The code adds a new row to the Customers table and updates the data source. The code then locates the added row by searching on the CustomerID, which is the primary key defined for the Customers table. It changes the CompanyName and updates the data source. Finally, the code deletes the row.

```
static void CreateDataAdapter(string providerName, string connectionString)
{
    try
    {
        // Create the DbProviderFactory and DbConnection.
```

```
// Create the DbProviderFactory and DbConnection.
DbProviderFactory factory =
    DbProviderFactories.GetFactory(providerName);

DbConnection connection = factory.CreateConnection();
connection.ConnectionString = connectionString;

using (connection)
{
    // Define the query.
    string queryString =
        "SELECT CustomerID, CompanyName FROM Customers";

    // Create the select command.
    DbCommand command = factory.CreateCommand();
    command.CommandText = queryString;
    command.Connection = connection;

    // Create the DbDataAdapter.
    DbDataAdapter adapter = factory.CreateDataAdapter();
    adapter.SelectCommand = command;

    // Create the DbCommandBuilder.
    DbCommandBuilder builder = factory.CreateCommandBuilder();
    builder.DataAdapter = adapter;

    // Get the insert, update and delete commands.
    adapter.InsertCommand = builder.GetInsertCommand();
    adapter.UpdateCommand = builder.GetUpdateCommand();
    adapter.DeleteCommand = builder.GetDeleteCommand();

    // Display the CommandText for each command.
    Console.WriteLine("InsertCommand: {0}",
        adapter.InsertCommand.CommandText);
    Console.WriteLine("UpdateCommand: {0}",
        adapter.UpdateCommand.CommandText);
    Console.WriteLine("DeleteCommand: {0}",
        adapter.DeleteCommand.CommandText);

    // Fill the DataTable.
    DataTable table = new DataTable();
    adapter.Fill(table);

    // Insert a new row.
    DataRow newRow = table.NewRow();
    newRow["CustomerID"] = "XYZZZ";
    newRow["CompanyName"] = "XYZ Company";
    table.Rows.Add(newRow);

    adapter.Update(table);

    // Display rows after insert.
    Console.WriteLine();
    Console.WriteLine("----List All Rows-----");
    foreach (DataRow row in table.Rows)
    {
        Console.WriteLine("{0} {1}", row[0], row[1]);
    }
    Console.WriteLine("----After Insert-----");

    // Edit an existing row.
    DataRow[] editRow = table.Select("CustomerID = 'XYZZZ'");
    editRow[0]["CompanyName"] = "XYZ Corporation";

    adapter.Update(table);

    // Display rows after update.
    Console.WriteLine();
    foreach (DataRow row in table.Rows)
    {
```

```
            {
                Console.WriteLine("{0} {1}", row[0], row[1]);
            }
            Console.WriteLine("----After Update-----");

            // Delete a row.
            DataRow[] deleteRow = table.Select("CustomerID = 'XYZZZ'");
            foreach (DataRow row in deleteRow)
            {
                row.Delete();
            }

            adapter.Update(table);

            // Display rows after delete.
            Console.WriteLine();
            foreach (DataRow row in table.Rows)
            {
                Console.WriteLine("{0} {1}", row[0], row[1]);
            }
            Console.WriteLine("----After Delete-----");
            Console.WriteLine("Customer XYZZZ was deleted.");
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
}
```

```
Shared Sub CreateDataAdapter(ByVal providerName As String, _
    ByVal connectionString As String)

    ' Create the DbProviderFactory and DbConnection.
    Try
        Dim factory As DbProviderFactory = _
           DbProviderFactories.GetFactory(providerName)

        Dim connection As DbConnection = _
            factory.CreateConnection()
        connection.ConnectionString = connectionString

        Using connection
            ' Define the query.
            Dim queryString As String = _
              "SELECT CustomerID, CompanyName FROM Customers"

            'Create the select command.
            Dim command As DbCommand = _
                factory.CreateCommand()
            command.CommandText = queryString
            command.Connection = connection

            ' Create the DbDataAdapter.
            Dim adapter As DbDataAdapter = _
                factory.CreateDataAdapter()
            adapter.SelectCommand = command

            ' Create the DbCommandBuilder.
            Dim builder As DbCommandBuilder = _
              factory.CreateCommandBuilder()
            builder.DataAdapter = adapter

            ' Get the insert, update and delete commands.
            adapter.InsertCommand = builder.GetInsertCommand()
            adapter.UpdateCommand = builder.GetUpdateCommand()
            adapter.DeleteCommand = builder.GetDeleteCommand()
```

```
            ' Display the CommandText for each command.
            Console.WriteLine("InsertCommand: {0}", _
              adapter.InsertCommand.CommandText)
            Console.WriteLine("UpdateCommand: {0}", _
              adapter.UpdateCommand.CommandText)
            Console.WriteLine("DeleteCommand: {0}", _
              adapter.DeleteCommand.CommandText)

            ' Fill the DataTable
            Dim table As New DataTable
            adapter.Fill(table)

            ' Insert a new row.
            Dim newRow As DataRow = table.NewRow
            newRow("CustomerID") = "XYZZZ"
            newRow("CompanyName") = "XYZ Company"
            table.Rows.Add(newRow)

            adapter.Update(table)

            ' Display rows after insert.
            Console.WriteLine()
            Console.WriteLine("----List All Rows-----")
            Dim row As DataRow
            For Each row In table.Rows
                Console.WriteLine("{0} {1}", row(0), row(1))
            Next
            Console.WriteLine("----After Insert-----")

            ' Edit an existing row.
            Dim editRow() As DataRow = _
              table.Select("CustomerID = 'XYZZZ'")
            editRow(0)("CompanyName") = "XYZ Corporation"

            adapter.Update(table)

            ' Display rows after update.
            Console.WriteLine()
            For Each row In table.Rows
                Console.WriteLine("{0} {1}", row(0), row(1))
            Next
            Console.WriteLine("----After Update-----")

            ' Delete a row.
            Dim deleteRow() As DataRow = _
              table.Select("CustomerID = 'XYZZZ'")
            For Each row In deleteRow
                row.Delete()
            Next

            adapter.Update(table)
            table.AcceptChanges()

            ' Display each row and column value after delete.
            Console.WriteLine()
            For Each row In table.Rows
                Console.WriteLine("{0} {1}", row(0), row(1))
            Next
            Console.WriteLine("----After Delete-----")
            Console.WriteLine("Customer XYZZZ was deleted.")
        End Using

    Catch ex As Exception
        Console.WriteLine(ex.Message)
    End Try
End Sub
```

Handling Parameters

The .NET Framework data providers handle naming and specifying parameters and parameter placeholders differently. This syntax is tailored to a specific data source, as described in the following table.

DATA PROVIDER	PARAMETER NAMING SYNTAX
`SqlClient`	Uses named parameters in the format `@` *parametername*.
`OracleClient`	Uses named parameters in the format `:` *parmname* (or *parmname*).
`OleDb`	Uses positional parameter markers indicated by a question mark (`?`).
`Odbc`	Uses positional parameter markers indicated by a question mark (`?`).

The factory model is not helpful for creating parameterized `DbCommand` and `DbDataAdapter` objects. You will need to branch in your code to create parameters that are tailored to your data provider.

> **IMPORTANT**
>
> Avoiding provider-specific parameters altogether by using string concatenation to construct direct SQL statements is not recommended for security reasons. Using string concatenation instead of parameters leaves your application vulnerable to SQL injection attacks.

See also

- DbProviderFactories
- Obtaining a DbProviderFactory
- DbConnection, DbCommand and DbException
- ADO.NET Overview

Data Tracing in ADO.NET

2/4/2020 • 3 minutes to read • Edit Online

ADO.NET features built-in data tracing functionality that is supported by the .NET data providers for SQL Server, Oracle, OLE DB and ODBC, as well as the ADO.NET DataSet, and the SQL Server network protocols.

Tracing data access API calls can help diagnose the following problems:

- Schema mismatch between client program and the database.
- Database unavailability or network library problems.
- Incorrect SQL whether hard coded or generated by an application.
- Incorrect programming logic.
- Issues resulting from the interaction between multiple ADO.NET components or between ADO.NET and your own components.

To support different trace technologies, tracing is extensible, so a developer can trace a problem at any level of the application stack. Although tracing is not an ADO.NET-only feature, Microsoft providers take advantage of generalized tracing and instrumentation APIs.

For more information about setting and configuring managed tracing in ADO.NET, see Tracing Data Access.

Accessing Diagnostic Information in the Extended Events Log

In the .NET Framework Data Provider for SQL Server, data access tracing (Data Access Tracing) has been updated to make it easier to correlate client events with diagnostic information, such as connection failures, from the server's connectivity ring buffer and application performance information in the extended events log. For information about reading the extended events log, see View Event Session Data.

For connection operations, ADO.NET will send a client connection ID. If the connection fails, you can access the connectivity ring buffer (Connectivity troubleshooting in SQL Server 2008 with the Connectivity Ring Buffer) and find the `ClientConnectionID` field and get diagnostic information about the connection failure. Client connection IDs are logged in the ring buffer only if an error occurs. (If a connection fails before sending the prelogin packet, a client connection ID will not be generated.) The client connection ID is a 16-byte GUID. You can also find the client connection ID in the extended events target output, if the `client_connection_id` action is added to events in an extended events session. You can enable data access tracing and rerun the connection command and observe the `ClientConnectionID` field in the data access trace, if you need further client driver diagnostic assistance.

You can get the client connection ID programmatically by using the `SqlConnection.ClientConnectionID` property.

The `ClientConnectionID` is available for a SqlConnection object that successfully establishes a connection. If a connection attempt fails, `ClientConnectionID` may be available via `SqlException.ToString`.

ADO.NET also sends a thread-specific activity ID. The activity ID is captured in the extended events sessions if the sessions are started with the TRACK_CAUSALITY option enabled. For performance issues with an active connection, you can get the activity ID from the client's data access trace (`ActivityID` field) and then locate the activity ID in the extended events output. The activity ID in extended events is a 16-byte GUID (not the same as the GUID for the client connection ID) appended with a four-byte sequence number. The sequence number represents the order of a request within a thread and indicates the relative ordering of batch and RPC statements for the thread. The `ActivityID` is currently optionally sent for SQL batch statements and RPC requests when data access tracing is

enabled on and the 18th bit in the data access tracing configuration word is turned ON.

The following is a sample that uses Transact-SQL to start an extended events session that will be stored in a ring buffer and will record the activity ID sent from a client on RPC and batch operations.

```
create event session MySession on server
add event connectivity_ring_buffer_recorded,
add event sql_statement_starting (action (client_connection_id)),
add event sql_statement_completed (action (client_connection_id)),
add event rpc_starting (action (client_connection_id)),
add event rpc_completed (action (client_connection_id))
add target ring_buffer with (track_causality=on)
```

See also

- Network Tracing in the .NET Framework
- Tracing and Instrumenting Applications
- ADO.NET Overview

Performance Counters in ADO.NET

3/12/2020 • 8 minutes to read • Edit Online

ADO.NET 2.0 introduced expanded support for performance counters that includes support for both System.Data.SqlClient and System.Data.OracleClient. The System.Data.SqlClient performance counters available in previous versions of ADO.NET have been deprecated and replaced with the new performance counters discussed in this topic. You can use ADO.NET performance counters to monitor the status of your application and the connection resources that it uses. Performance counters can be monitored by using Windows Performance Monitor or can be accessed programmatically using the PerformanceCounter class in the System.Diagnostics namespace.

Available Performance Counters

Currently there are 14 different performance counters available for System.Data.SqlClient and System.Data.OracleClient as described in the following table. Note that the names for the individual counters are not localized across regional versions of the Microsoft .NET Framework.

PERFORMANCE COUNTER	DESCRIPTION
`HardConnectsPerSecond`	The number of connections per second that are being made to a database server.
`HardDisconnectsPerSecond`	The number of disconnects per second that are being made to a database server.
`NumberOfActiveConnectionPoolGroups`	The number of unique connection pool groups that are active. This counter is controlled by the number of unique connection strings that are found in the AppDomain.
`NumberOfActiveConnectionPools`	The total number of connection pools.
`NumberOfActiveConnections`	The number of active connections that are currently in use. **Note:** This performance counter is not enabled by default. To enable this performance counter, see Activating Off-By-Default Counters.
`NumberOfFreeConnections`	The number of connections available for use in the connection pools. **Note:** This performance counter is not enabled by default. To enable this performance counter, see Activating Off-By-Default Counters.
`NumberOfInactiveConnectionPoolGroups`	The number of unique connection pool groups that are marked for pruning. This counter is controlled by the number of unique connection strings that are found in the AppDomain.
`NumberOfInactiveConnectionPools`	The number of inactive connection pools that have not had any recent activity and are waiting to be disposed.
`NumberOfNonPooledConnections`	The number of active connections that are not pooled.

PERFORMANCE COUNTER	DESCRIPTION
`NumberOfPooledConnections`	The number of active connections that are being managed by the connection pooling infrastructure.
`NumberOfReclaimedConnections`	The number of connections that have been reclaimed through garbage collection where `Close` or `Dispose` was not called by the application. Not explicitly closing or disposing connections hurts performance.
`NumberOfStasisConnections`	The number of connections currently awaiting completion of an action and which are therefore unavailable for use by your application.
`SoftConnectsPerSecond`	The number of active connections being pulled from the connection pool. **Note:** This performance counter is not enabled by default. To enable this performance counter, see Activating Off-By-Default Counters.
`SoftDisconnectsPerSecond`	The number of active connections that are being returned to the connection pool. **Note:** This performance counter is not enabled by default. To enable this performance counter, see Activating Off-By-Default Counters.

Connection Pool Groups and Connection Pools

When using Windows Authentication (integrated security), you must monitor both the `NumberOfActiveConnectionPoolGroups` and `NumberOfActiveConnectionPools` performance counters. The reason is that connection pool groups map to unique connection strings. When integrated security is used, connection pools map to connection strings and additionally create separate pools for individual Windows identities. For example, if Fred and Julie, each within the same AppDomain, both use the connection string `"Data Source=MySqlServer;Integrated Security=true"`, a connection pool group is created for the connection string, and two additional pools are created, one for Fred and one for Julie. If John and Martha use a connection string with an identical SQL Server login, `"Data Source=MySqlServer;User Id=lowPrivUser;Password=Strong?Password"`, then only a single pool is created for the **lowPrivUser** identity.

Activating Off-By-Default Counters

The performance counters `NumberOfFreeConnections`, `NumberOfActiveConnections`, `SoftDisconnectsPerSecond`, and `SoftConnectsPerSecond` are off by default. Add the following information to the application's configuration file to enable them:

```
<system.diagnostics>
  <switches>
    <add name="ConnectionPoolPerformanceCounterDetail"
         value="4"/>
  </switches>
</system.diagnostics>
```

Retrieving Performance Counter Values

The following console application shows how to retrieve performance counter values in your application. Connections must be open and active for information to be returned for all of the ADO.NET performance counters.

NOTE

This example uses the sample **AdventureWorks** database included with SQL Server. The connection strings provided in the sample code assume that the database is installed and available on the local computer with an instance name of SqlExpress, and that you have created SQL Server logins that match those supplied in the connection strings. You may need to enable SQL Server logins if your server is configured using the default security settings which allow only Windows Authentication. Modify the connection strings as necessary to suit your environment.

Example

```
Option Explicit On
Option Strict On

Imports System.Data.SqlClient
Imports System.Diagnostics
Imports System.Runtime.InteropServices

Class Program

    Private PerfCounters(9) As PerformanceCounter
    Private connection As SqlConnection = New SqlConnection

    Public Shared Sub Main()
        Dim prog As Program = New Program
        ' Open a connection and create the performance counters.
        prog.connection.ConnectionString = _
           GetIntegratedSecurityConnectionString()
        prog.SetUpPerformanceCounters()
        Console.WriteLine("Available Performance Counters:")

        ' Create the connections and display the results.
        prog.CreateConnections()
        Console.WriteLine("Press Enter to finish.")
        Console.ReadLine()
    End Sub

    Private Sub CreateConnections()
        ' List the Performance counters.
        WritePerformanceCounters()

        ' Create 4 connections and display counter information.
        Dim connection1 As SqlConnection = New SqlConnection( _
           GetIntegratedSecurityConnectionString)
        connection1.Open()
        Console.WriteLine("Opened the 1st Connection:")
        WritePerformanceCounters()

        Dim connection2 As SqlConnection = New SqlConnection( _
           GetSqlConnectionStringDifferent)
        connection2.Open()
        Console.WriteLine("Opened the 2nd Connection:")
        WritePerformanceCounters()

        Console.WriteLine("Opened the 3rd Connection:")
        Dim connection3 As SqlConnection = New SqlConnection( _
           GetSqlConnectionString)
        connection3.Open()
        WritePerformanceCounters()

        Dim connection4 As SqlConnection = New SqlConnection( _
           GetSqlConnectionString)
        connection4.Open()
        Console.WriteLine("Opened the 4th Connection:")
        WritePerformanceCounters()
```

```
        connection1.Close()
        Console.WriteLine("Closed the 1st Connection:")
        WritePerformanceCounters()

        connection2.Close()
        Console.WriteLine("Closed the 2nd Connection:")
        WritePerformanceCounters()

        connection3.Close()
        Console.WriteLine("Closed the 3rd Connection:")
        WritePerformanceCounters()

        connection4.Close()
        Console.WriteLine("Closed the 4th Connection:")
        WritePerformanceCounters()
    End Sub

    Private Enum ADO_Net_Performance_Counters
        NumberOfActiveConnectionPools
        NumberOfReclaimedConnections
        HardConnectsPerSecond
        HardDisconnectsPerSecond
        NumberOfActiveConnectionPoolGroups
        NumberOfInactiveConnectionPoolGroups
        NumberOfInactiveConnectionPools
        NumberOfNonPooledConnections
        NumberOfPooledConnections
        NumberOfStasisConnections
        ' The following performance counters are more expensive to track.
        ' Enable ConnectionPoolPerformanceCounterDetail in your config file.
        '     SoftConnectsPerSecond
        '     SoftDisconnectsPerSecond
        '     NumberOfActiveConnections
        '     NumberOfFreeConnections
    End Enum

    Private Sub SetUpPerformanceCounters()
        connection.Close()
        Me.PerfCounters(9) = New PerformanceCounter()

        Dim instanceName As String = GetInstanceName()
        Dim apc As Type = GetType(ADO_Net_Performance_Counters)
        Dim i As Integer = 0
        Dim s As String = ""
        For Each s In [Enum].GetNames(apc)
            Me.PerfCounters(i) = New PerformanceCounter()
            Me.PerfCounters(i).CategoryName = ".NET Data Provider for SqlServer"
            Me.PerfCounters(i).CounterName = s
            Me.PerfCounters(i).InstanceName = instanceName
            i = (i + 1)
        Next
    End Sub

    Private Declare Function GetCurrentProcessId Lib "kernel32.dll" () As Integer

    Private Function GetInstanceName() As String
        'This works for Winforms apps.
        Dim instanceName As String = _
           System.Reflection.Assembly.GetEntryAssembly.GetName.Name

        ' Must replace special characters like (, ), #, /, \\
        Dim instanceName2 As String = _
           AppDomain.CurrentDomain.FriendlyName.ToString.Replace("(", "[") _
           .Replace(")", "]").Replace("#", "_").Replace("/", "_").Replace("\\", "_")

        'For ASP.NET applications your instanceName will be your CurrentDomain's
        'FriendlyName. Replace the line above that sets the instanceName with this:
        'instanceName = AppDomain.CurrentDomain.FriendlyName.ToString.Replace("(", "[") _
        '    .Replace(")", "]").Replace("#", "_").Replace("/", "_").Replace("\\", "_")
```

```
        Dim pid As String = GetCurrentProcessId.ToString
        instanceName = (instanceName + ("[" & (pid & "]")))
        Console.WriteLine("Instance Name: {0}", instanceName)
        Console.WriteLine("---------------------------")
        Return instanceName
    End Function

    Private Sub WritePerformanceCounters()
        Console.WriteLine("---------------------------")
        For Each p As PerformanceCounter In Me.PerfCounters
            Console.WriteLine("{0} = {1}", p.CounterName, p.NextValue)
        Next
        Console.WriteLine("---------------------------")
    End Sub

    Private Shared Function GetIntegratedSecurityConnectionString() As String
        ' To avoid storing the connection string in your code,
        ' you can retrieve it from a configuration file.
        Return ("Data Source=.\SqlExpress;Integrated Security=True;" &
          "Initial Catalog=AdventureWorks")
    End Function

    Private Shared Function GetSqlConnectionString() As String
        ' To avoid storing the connection string in your code,
        ' you can retrieve it from a configuration file.
        Return ("Data Source=.\SqlExpress;User Id=LowPriv;Password=Data!05;" &
          "Initial Catalog=AdventureWorks")
    End Function

    Private Shared Function GetSqlConnectionStringDifferent() As String
        ' To avoid storing the connection string in your code,
        ' you can retrieve it from a configuration file.
        Return ("Initial Catalog=AdventureWorks;Data Source=.\SqlExpress;" & _
          "User Id=LowPriv;Password=Data!05;")
    End Function
End Class
```

```
using System;
using System.Data.SqlClient;
using System.Diagnostics;
using System.Runtime.InteropServices;

class Program
{
    PerformanceCounter[] PerfCounters = new PerformanceCounter[10];
    SqlConnection connection = new SqlConnection();

    static void Main()
    {
        Program prog = new Program();
        // Open a connection and create the performance counters.
        prog.connection.ConnectionString =
           GetIntegratedSecurityConnectionString();
        prog.SetUpPerformanceCounters();
        Console.WriteLine("Available Performance Counters:");

        // Create the connections and display the results.
        prog.CreateConnections();
        Console.WriteLine("Press Enter to finish.");
        Console.ReadLine();
    }

    private void CreateConnections()
    {
        // List the Performance counters.
        WritePerformanceCounters();
```

```
        // Create 4 connections and display counter information.
        SqlConnection connection1 = new SqlConnection(
              GetIntegratedSecurityConnectionString());
        connection1.Open();
        Console.WriteLine("Opened the 1st Connection:");
        WritePerformanceCounters();

        SqlConnection connection2 = new SqlConnection(
              GetSqlConnectionStringDifferent());
        connection2.Open();
        Console.WriteLine("Opened the 2nd Connection:");
        WritePerformanceCounters();

        SqlConnection connection3 = new SqlConnection(
              GetSqlConnectionString());
        connection3.Open();
        Console.WriteLine("Opened the 3rd Connection:");
        WritePerformanceCounters();

        SqlConnection connection4 = new SqlConnection(
              GetSqlConnectionString());
        connection4.Open();
        Console.WriteLine("Opened the 4th Connection:");
        WritePerformanceCounters();

        connection1.Close();
        Console.WriteLine("Closed the 1st Connection:");
        WritePerformanceCounters();

        connection2.Close();
        Console.WriteLine("Closed the 2nd Connection:");
        WritePerformanceCounters();

        connection3.Close();
        Console.WriteLine("Closed the 3rd Connection:");
        WritePerformanceCounters();

        connection4.Close();
        Console.WriteLine("Closed the 4th Connection:");
        WritePerformanceCounters();
    }

    private enum ADO_Net_Performance_Counters
    {
        NumberOfActiveConnectionPools,
        NumberOfReclaimedConnections,
        HardConnectsPerSecond,
        HardDisconnectsPerSecond,
        NumberOfActiveConnectionPoolGroups,
        NumberOfInactiveConnectionPoolGroups,
        NumberOfInactiveConnectionPools,
        NumberOfNonPooledConnections,
        NumberOfPooledConnections,
        NumberOfStasisConnections
        // The following performance counters are more expensive to track.
        // Enable ConnectionPoolPerformanceCounterDetail in your config file.
        //     SoftConnectsPerSecond
        //     SoftDisconnectsPerSecond
        //     NumberOfActiveConnections
        //     NumberOfFreeConnections
    }

    private void SetUpPerformanceCounters()
    {
        connection.Close();
        this.PerfCounters = new PerformanceCounter[10];
        string instanceName = GetInstanceName();
        Type apc = typeof(ADO_Net_Performance_Counters);
```

```
        int i = 0;
        foreach (string s in Enum.GetNames(apc))
        {
            this.PerfCounters[i] = new PerformanceCounter();
            this.PerfCounters[i].CategoryName = ".NET Data Provider for SqlServer";
            this.PerfCounters[i].CounterName = s;
            this.PerfCounters[i].InstanceName = instanceName;
            i++;
        }
    }

    [DllImport("kernel32.dll", SetLastError = true)]
    static extern int GetCurrentProcessId();

    private string GetInstanceName()
    {
        //This works for Winforms apps.
        string instanceName =
            System.Reflection.Assembly.GetEntryAssembly().GetName().Name;

        // Must replace special characters like (, ), #, /, \\
        string instanceName2 =
            AppDomain.CurrentDomain.FriendlyName.ToString().Replace('(', '[')
            .Replace(')', ']').Replace('#', '_').Replace('/', '_').Replace('\\', '_');

        // For ASP.NET applications your instanceName will be your CurrentDomain's
        // FriendlyName. Replace the line above that sets the instanceName with this:
        // instanceName = AppDomain.CurrentDomain.FriendlyName.ToString().Replace('(','[')
        // .Replace(')',']').Replace('#','_').Replace('/','_').Replace('\\','_');

        string pid = GetCurrentProcessId().ToString();
        instanceName = instanceName + "[" + pid + "]";
        Console.WriteLine("Instance Name: {0}", instanceName);
        Console.WriteLine("---------------------------");
        return instanceName;
    }

    private void WritePerformanceCounters()
    {
        Console.WriteLine("---------------------------");
        foreach (PerformanceCounter p in this.PerfCounters)
        {
            Console.WriteLine("{0} = {1}", p.CounterName, p.NextValue());
        }
        Console.WriteLine("---------------------------");
    }

    private static string GetIntegratedSecurityConnectionString()
    {
        // To avoid storing the connection string in your code,
        // you can retrieve it from a configuration file.
        return @"Data Source=.\SqlExpress;Integrated Security=True;" +
          "Initial Catalog=AdventureWorks";
    }
    private static string GetSqlConnectionString()
    {
        // To avoid storing the connection string in your code,
        // you can retrieve it from a configuration file.
        return @"Data Source=.\SqlExpress;User Id=LowPriv;Password=Data!05;" +
          "Initial Catalog=AdventureWorks";
    }

    private static string GetSqlConnectionStringDifferent()
    {
        // To avoid storing the connection string in your code,
        // you can retrieve it from a configuration file.
        return @"Initial Catalog=AdventureWorks;Data Source=.\SqlExpress;" +
          "User Id=LowPriv;Password=Data!05;";
    }
```

```
}
```

See also

- Connecting to a Data Source
- OLE DB, ODBC, and Oracle Connection Pooling
- Performance Counters for ASP.NET
- Runtime Profiling
- Introduction to Monitoring Performance Thresholds
- ADO.NET Overview

Asynchronous Programming

1/14/2020 • 15 minutes to read • Edit Online

This topic discusses support for asynchronous programming in the .NET Framework Data Provider for SQL Server (SqlClient) including enhancements made to support asynchronous programming functionality that was introduced in .NET Framework 4.5.

Legacy Asynchronous Programming

Prior to .NET Framework 4.5, asynchronous programming with SqlClient was done with the following methods and the `Asynchronous Processing=true` connection property:

1. SqlCommand.BeginExecuteNonQuery
2. SqlCommand.BeginExecuteReader
3. SqlCommand.BeginExecuteXmlReader

This functionality remains in SqlClient in .NET Framework 4.5.

> **TIP**
> Beginning in the .NET Framework 4.5, these legacy methods no longer require `Asynchronous Processing=true` in the connection string.

Asynchronous Programming Features Added in .NET Framework 4.5

The new asynchronous programming feature provides a simple technique to make code asynchronous.

For more information about the asynchronous programming feature that was introduced in .NET Framework 4.5, see:

- Asynchronous programming in C#
- Asynchronous Programming with Async and Await (Visual Basic)
- Using SqlDataReader's new async methods in .NET 4.5 (Part 1)
- Using SqlDataReader's new async methods in .NET 4.5 (Part 2)

When your user interface is unresponsive or your server does not scale, it is likely that you need your code to be more asynchronous. Writing asynchronous code has traditionally involved installing a callback (also called continuation) to express the logic that occurs after the asynchronous operation finishes. This complicates the structure of asynchronous code as compared with synchronous code.

You can now call into asynchronous methods without using callbacks, and without splitting your code across multiple methods or lambda expressions.

The `async` modifier specifies that a method is asynchronous. When calling an `async` method, a task is returned. When the `await` operator is applied to a task, the current method exits immediately. When the task finishes, execution resumes in the same method.

WARNING

Asynchronous calls are not supported if an application also uses the `Context Connection` connection string keyword.

Calling an `async` method does not allocate any additional threads. It may use the existing I/O completion thread briefly at the end.

The following methods were added in .NET Framework 4.5 to support asynchronous programming:

- DbConnection.OpenAsync
- DbCommand.ExecuteDbDataReaderAsync
- DbCommand.ExecuteNonQueryAsync
- DbCommand.ExecuteReaderAsync
- DbCommand.ExecuteScalarAsync
- GetFieldValueAsync
- IsDBNullAsync
- DbDataReader.NextResultAsync
- DbDataReader.ReadAsync
- SqlConnection.OpenAsync
- SqlCommand.ExecuteNonQueryAsync
- SqlCommand.ExecuteReaderAsync
- SqlCommand.ExecuteScalarAsync
- SqlCommand.ExecuteXmlReaderAsync
- SqlDataReader.NextResultAsync
- SqlDataReader.ReadAsync
- SqlBulkCopy.WriteToServerAsync

Other asynchronous members were added to support SqlClient Streaming Support.

TIP

The new asynchronous methods don't require `Asynchronous Processing=true` in the connection string.

Synchronous to Asynchronous Connection Open

You can upgrade an existing application to use the new asynchronous feature. For example, assume an application has a synchronous connection algorithm and blocks the UI thread every time it connects to the database and, once connected, the application calls a stored procedure that signals other users of the one who just signed in.

```
using SqlConnection conn = new SqlConnection("…");
{
   conn.Open();
   using (SqlCommand cmd = new SqlCommand("StoredProcedure_Logon", conn))
   {
      cmd.ExecuteNonQuery();
   }
}
```

When converted to use the new asynchronous functionality, the program would look like:

```
using System;
using System.Data.SqlClient;
using System.Threading.Tasks;

class A {

   static async Task<int> Method(SqlConnection conn, SqlCommand cmd) {
      await conn.OpenAsync();
      await cmd.ExecuteNonQueryAsync();
      return 1;
   }

   public static void Main() {
      using (SqlConnection conn = new SqlConnection("Data Source=(local); Initial Catalog=NorthWind;
Integrated Security=SSPI")) {
         SqlCommand command = new SqlCommand("select top 2 * from orders", conn);

         int result = A.Method(conn, command).Result;

         SqlDataReader reader = command.ExecuteReader();
         while (reader.Read())
            Console.WriteLine(reader[0]);
      }
   }
}
```

Adding the New Asynchronous Feature in an Existing Application (Mixing Old and New Patterns)

It is also possible to add new asynchronous capability (SqlConnection::OpenAsync) without changing the existing asynchronous logic. For example, if an application currently uses:

```
AsyncCallback productList = new AsyncCallback(ProductList);
SqlConnection conn = new SqlConnection("…");
conn.Open();
SqlCommand cmd = new SqlCommand("SELECT * FROM [Current Product List]", conn);
IAsyncResult ia = cmd.BeginExecuteReader(productList, cmd);
```

You can begin to use the new asynchronous pattern without substantially changing the existing algorithm.

```
using System;
using System.Data.SqlClient;
using System.Threading.Tasks;

class A {
   static void ProductList(IAsyncResult result) { }

   public static void Main() {
      // AsyncCallback productList = new AsyncCallback(ProductList);
      // SqlConnection conn = new SqlConnection("Data Source=(local); Initial Catalog=NorthWind; Integrated
Security=SSPI");
      // conn.Open();
      // SqlCommand cmd = new SqlCommand("select top 2 * from orders", conn);
      // IAsyncResult ia = cmd.BeginExecuteReader(productList, cmd);

      AsyncCallback productList = new AsyncCallback(ProductList);
      SqlConnection conn = new SqlConnection("Data Source=(local); Initial Catalog=NorthWind; Integrated
Security=SSPI");
      conn.OpenAsync().ContinueWith((task) => {
         SqlCommand cmd = new SqlCommand("select top 2 * from orders", conn);
         IAsyncResult ia = cmd.BeginExecuteReader(productList, cmd);
      }, TaskContinuationOptions.OnlyOnRanToCompletion);
   }
}
```

Using the Base Provider Model and the New Asynchronous Feature

You may need to create a tool that is able to connect to different databases and execute queries. You can use the base provider model and the new asynchronous feature.

The Microsoft Distributed Transaction Controller (MSDTC) must be enabled on the server to use distributed transactions. For information on how to enable MSDTC, see How to Enable MSDTC on a Web Server.

```
using System;
using System.Data.Common;
using System.Data.SqlClient;
using System.Threading.Tasks;

class A {
   static async Task PerformDBOperationsUsingProviderModel(string connectionString, string providerName) {
      DbProviderFactory factory = DbProviderFactories.GetFactory(providerName);
      using (DbConnection connection = factory.CreateConnection()) {
         connection.ConnectionString = connectionString;
         await connection.OpenAsync();

         DbCommand command = connection.CreateCommand();
         command.CommandText = "SELECT * FROM AUTHORS";

         using (DbDataReader reader = await command.ExecuteReaderAsync()) {
            while (await reader.ReadAsync()) {
               for (int i = 0; i < reader.FieldCount; i++) {
                  // Process each column as appropriate
                  object obj = await reader.GetFieldValueAsync<object>(i);
                  Console.WriteLine(obj);
               }
            }
         }
      }
   }

   public static void Main()
   {
       SqlConnectionStringBuilder builder = new SqlConnectionStringBuilder();
       // replace these with your own values
       builder.DataSource = "your_server";
       builder.InitialCatalog = "pubs";
       builder.IntegratedSecurity = true;
       string provider = "System.Data.SqlClient";

       Task task = PerformDBOperationsUsingProviderModel(builder.ConnectionString, provider);
       task.Wait();
   }
}
```

Using SQL Transactions and the New Asynchronous Feature

```
using System;
using System.Data.SqlClient;
using System.Threading.Tasks;

class Program {
   static void Main() {
      string connectionString =
          "Persist Security Info=False;Integrated Security=SSPI;database=Northwind;server=(local)";
      Task task = ExecuteSqlTransaction(connectionString);
      task.Wait();
   }

   static async Task ExecuteSqlTransaction(string connectionString) {
      using (SqlConnection connection = new SqlConnection(connectionString)) {
         await connection.OpenAsync();

         SqlCommand command = connection.CreateCommand();
         SqlTransaction transaction = null;

         // Start a local transaction.
         transaction = await Task.Run<SqlTransaction>(
             () => connection.BeginTransaction("SampleTransaction")
             );

         // Must assign both transaction object and connection
         // to Command object for a pending local transaction
         command.Connection = connection;
         command.Transaction = transaction;

         try {
            command.CommandText =
                "Insert into Region (RegionID, RegionDescription) VALUES (555, 'Description')";
            await command.ExecuteNonQueryAsync();

            command.CommandText =
                "Insert into Region (RegionID, RegionDescription) VALUES (556, 'Description')";
            await command.ExecuteNonQueryAsync();

            // Attempt to commit the transaction.
            await Task.Run(() => transaction.Commit());
            Console.WriteLine("Both records are written to database.");
         }
         catch (Exception ex) {
            Console.WriteLine("Commit Exception Type: {0}", ex.GetType());
            Console.WriteLine("  Message: {0}", ex.Message);

            // Attempt to roll back the transaction.
            try {
               transaction.Rollback();
            }
            catch (Exception ex2) {
               // This catch block will handle any errors that may have occurred
               // on the server that would cause the rollback to fail, such as
               // a closed connection.
               Console.WriteLine("Rollback Exception Type: {0}", ex2.GetType());
               Console.WriteLine("  Message: {0}", ex2.Message);
            }
         }
      }
   }
}
```

Using SQL Transactions and the New Asynchronous Feature

In an enterprise application, you may need to add distributed transactions in some scenarios, to enable transactions between multiple database servers. You can use the System.Transactions namespace and enlist a

distributed transaction, as follows:

```
using System;
using System.Data.SqlClient;
using System.Threading.Tasks;
using System.Transactions;

class Program {
   public static void Main()
   {
       SqlConnectionStringBuilder builder = new SqlConnectionStringBuilder();
       // replace these with your own values
       builder.DataSource = "your_server";
       builder.InitialCatalog = "your_data_source";
       builder.IntegratedSecurity = true;

       Task task = ExecuteDistributedTransaction(builder.ConnectionString, builder.ConnectionString);
       task.Wait();
   }

   static async Task ExecuteDistributedTransaction(string connectionString1, string connectionString2) {
      using (SqlConnection connection1 = new SqlConnection(connectionString1))
      using (SqlConnection connection2 = new SqlConnection(connectionString2)) {
         using (CommittableTransaction transaction = new CommittableTransaction()) {
            await connection1.OpenAsync();
            connection1.EnlistTransaction(transaction);

            await connection2.OpenAsync();
            connection2.EnlistTransaction(transaction);

            try {
               SqlCommand command1 = connection1.CreateCommand();
               command1.CommandText = "Insert into RegionTable1 (RegionID, RegionDescription) VALUES (100,
'Description')";
               await command1.ExecuteNonQueryAsync();

               SqlCommand command2 = connection2.CreateCommand();
               command2.CommandText = "Insert into RegionTable2 (RegionID, RegionDescription) VALUES (100,
'Description')";
               await command2.ExecuteNonQueryAsync();

               transaction.Commit();
            }
            catch (Exception ex) {
               Console.WriteLine("Exception Type: {0}", ex.GetType());
               Console.WriteLine("  Message: {0}", ex.Message);

               try {
                  transaction.Rollback();
               }
               catch (Exception ex2) {
                  Console.WriteLine("Rollback Exception Type: {0}", ex2.GetType());
                  Console.WriteLine("  Message: {0}", ex2.Message);
               }
            }
         }
      }
   }
}
```

Cancelling an Asynchronous Operation

You can cancel an asynchronous request by using the CancellationToken.

```
using System;
using System.Data.SqlClient;
using System.Threading;
using System.Threading.Tasks;

namespace Samples {
   class CancellationSample {
      public static void Main(string[] args) {
         CancellationTokenSource source = new CancellationTokenSource();
         source.CancelAfter(2000); // give up after 2 seconds
         try {
            Task result = CancellingAsynchronousOperations(source.Token);
            result.Wait();
         }
         catch (AggregateException exception) {
            if (exception.InnerException is SqlException) {
               Console.WriteLine("Operation canceled");
            }
            else {
               throw;
            }
         }
      }

      static async Task CancellingAsynchronousOperations(CancellationToken cancellationToken) {
         using (SqlConnection connection = new SqlConnection("Server=(local);Integrated Security=true")) {
            await connection.OpenAsync(cancellationToken);

            SqlCommand command = new SqlCommand("WAITFOR DELAY '00:10:00'", connection);
            await command.ExecuteNonQueryAsync(cancellationToken);
         }
      }
   }
}
```

Asynchronous Operations with SqlBulkCopy

Asynchronous capabilities were also added to System.Data.SqlClient.SqlBulkCopy with SqlBulkCopy.WriteToServerAsync.

```
using System;
using System.Collections.Generic;
using System.Data;
using System.Data.Odbc;
using System.Data.SqlClient;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

namespace SqlBulkCopyAsyncCodeSample {
   class Program {
      static string selectStatement = "SELECT * FROM [pubs].[dbo].[titles]";
      static string createDestTableStatement =
          @"CREATE TABLE {0} (
            [title_id] [varchar](6) NOT NULL,
            [title] [varchar](80) NOT NULL,
            [type] [char](12) NOT NULL,
            [pub_id] [char](4) NULL,
            [price] [money] NULL,
            [advance] [money] NULL,
            [royalty] [int] NULL,
            [ytd_sales] [int] NULL,
            [notes] [varchar](200) NULL,
            [pubdate] [datetime] NOT NULL)";
```

```
      // Replace the connection string if needed, for instance to connect to SQL Express: @"Server=
(local)\SQLEXPRESS;Database=Demo;Integrated Security=true"
      // static string connectionString = @"Server=(localdb)\V11.0;Database=Demo";
      static string connectionString = @"Server=(local);Database=Demo;Integrated Security=true";

      // static string odbcConnectionString = @"Driver={SQL Server};Server=
(localdb)\V11.0;UID=oledb;Pwd=1Password!;Database=Demo";
      static string odbcConnectionString = @"Driver={SQL Server};Server=(local);Database=Demo;Integrated
Security=true";

      // static string marsConnectionString = @"Server=
(localdb)\V11.0;Database=Demo;MultipleActiveResultSets=true;";
      static string marsConnectionString = @"Server=
(local);Database=Demo;MultipleActiveResultSets=true;Integrated Security=true";

      // Replace the Server name with your actual sql azure server name and User ID/Password
      static string azureConnectionString = @"Server=SqlAzure;User
ID=myUserID;Password=myPassword;Database=Demo";

      static void Main(string[] args) {
         SynchronousSqlBulkCopy();
         AsyncSqlBulkCopy().Wait();
         MixSyncAsyncSqlBulkCopy().Wait();
         AsyncSqlBulkCopyNotifyAfter().Wait();
         AsyncSqlBulkCopyDataRows().Wait();
         // AsyncSqlBulkCopySqlServerToSqlAzure().Wait();
         // AsyncSqlBulkCopyCancel().Wait();
         AsyncSqlBulkCopyMARS().Wait();
      }

      // 3.1.1 Synchronous bulk copy in .NET 4.5
      private static void SynchronousSqlBulkCopy() {
         using (SqlConnection conn = new SqlConnection(connectionString)) {
            conn.Open();
            DataTable dt = new DataTable();
            using (SqlCommand cmd = new SqlCommand(selectStatement, conn)) {
               SqlDataAdapter adapter = new SqlDataAdapter(cmd);
               adapter.Fill(dt);

               string temptable = "[#" + Guid.NewGuid().ToString("N") + "]";
               cmd.CommandText = string.Format(createDestTableStatement, temptable);
               cmd.ExecuteNonQuery();

               using (SqlBulkCopy bcp = new SqlBulkCopy(conn)) {
                  bcp.DestinationTableName = temptable;
                  bcp.WriteToServer(dt);
               }
            }
         }

      }

      // 3.1.2 Asynchronous bulk copy in .NET 4.5
      private static async Task AsyncSqlBulkCopy() {
         using (SqlConnection conn = new SqlConnection(connectionString)) {
            await conn.OpenAsync();
            DataTable dt = new DataTable();
            using (SqlCommand cmd = new SqlCommand(selectStatement, conn)) {
               SqlDataAdapter adapter = new SqlDataAdapter(cmd);
               adapter.Fill(dt);

               string temptable = "[#" + Guid.NewGuid().ToString("N") + "]";
               cmd.CommandText = string.Format(createDestTableStatement, temptable);
               await cmd.ExecuteNonQueryAsync();

               using (SqlBulkCopy bcp = new SqlBulkCopy(conn)) {
                  bcp.DestinationTableName = temptable;
                  await bcp.WriteToServerAsync(dt);
               }
```

```
            }
         }
      }

      // 3.2 Add new Async.NET capabilities in an existing application (Mixing synchronous and asynchronous
calls)
      private static async Task MixSyncAsyncSqlBulkCopy() {
         using (OdbcConnection odbcconn = new OdbcConnection(odbcConnectionString)) {
            odbcconn.Open();
            using (OdbcCommand odbccmd = new OdbcCommand(selectStatement, odbcconn)) {
               using (OdbcDataReader odbcreader = odbccmd.ExecuteReader()) {
                  using (SqlConnection conn = new SqlConnection(connectionString)) {
                     await conn.OpenAsync();
                     string temptable = "temptable";//"[#" + Guid.NewGuid().ToString("N") + "]";
                     SqlCommand createCmd = new SqlCommand(string.Format(createDestTableStatement, temptable),
conn);
                     await createCmd.ExecuteNonQueryAsync();
                     using (SqlBulkCopy bcp = new SqlBulkCopy(conn)) {
                        bcp.DestinationTableName = temptable;
                        await bcp.WriteToServerAsync(odbcreader);
                     }
                  }
               }
            }
         }
      }

      // 3.3 Using the NotifyAfter property
      private static async Task AsyncSqlBulkCopyNotifyAfter() {
         using (SqlConnection conn = new SqlConnection(connectionString)) {
            await conn.OpenAsync();
            DataTable dt = new DataTable();
            using (SqlCommand cmd = new SqlCommand(selectStatement, conn)) {
               SqlDataAdapter adapter = new SqlDataAdapter(cmd);
               adapter.Fill(dt);

               string temptable = "[#" + Guid.NewGuid().ToString("N") + "]";
               cmd.CommandText = string.Format(createDestTableStatement, temptable);
               await cmd.ExecuteNonQueryAsync();

               using (SqlBulkCopy bcp = new SqlBulkCopy(conn)) {
                  bcp.DestinationTableName = temptable;
                  bcp.NotifyAfter = 5;
                  bcp.SqlRowsCopied += new SqlRowsCopiedEventHandler(OnSqlRowsCopied);
                  await bcp.WriteToServerAsync(dt);
               }
            }
         }
      }

      private static void OnSqlRowsCopied(object sender, SqlRowsCopiedEventArgs e) {
         Console.WriteLine("Copied {0} so far...", e.RowsCopied);
      }

      // 3.4 Using the new SqlBulkCopy Async.NET capabilities with DataRow[]
      private static async Task AsyncSqlBulkCopyDataRows() {
         using (SqlConnection conn = new SqlConnection(connectionString)) {
            await conn.OpenAsync();
            DataTable dt = new DataTable();
            using (SqlCommand cmd = new SqlCommand(selectStatement, conn)) {
               SqlDataAdapter adapter = new SqlDataAdapter(cmd);
               adapter.Fill(dt);
               DataRow[] rows = dt.Select();

               string temptable = "[#" + Guid.NewGuid().ToString("N") + "]";
               cmd.CommandText = string.Format(createDestTableStatement, temptable);
               await cmd.ExecuteNonQueryAsync();

               using (SqlBulkCopy bcp = new SqlBulkCopy(conn)) {
```

```
                using (SqlBulkCopy bcp = new SqlBulkCopy(conn)) {
                   bcp.DestinationTableName = temptable;
                   await bcp.WriteToServerAsync(rows);
                }
             }
          }
       }

       // 3.5 Copying data from SQL Server to SQL Azure in .NET 4.5
       //private static async Task AsyncSqlBulkCopySqlServerToSqlAzure() {
       //   using (SqlConnection srcConn = new SqlConnection(connectionString))
       //   using (SqlConnection destConn = new SqlConnection(azureConnectionString)) {
       //      await srcConn.OpenAsync();
       //      await destConn.OpenAsync();
       //      using (SqlCommand srcCmd = new SqlCommand(selectStatement, srcConn)) {
       //         using (SqlDataReader reader = await srcCmd.ExecuteReaderAsync()) {
       //            string temptable = "[#" + Guid.NewGuid().ToString("N") + "]";
       //            using (SqlCommand destCmd = new SqlCommand(string.Format(createDestTableStatement,
temptable), destConn)) {
       //               await destCmd.ExecuteNonQueryAsync();
       //               using (SqlBulkCopy bcp = new SqlBulkCopy(destConn)) {
       //                  bcp.DestinationTableName = temptable;
       //                  await bcp.WriteToServerAsync(reader);
       //               }
       //            }
       //         }
       //      }
       //   }
       //}

       // 3.6 Cancelling an Asynchronous Operation to SQL Azure
       //private static async Task AsyncSqlBulkCopyCancel() {
       //   CancellationTokenSource cts = new CancellationTokenSource();
       //   using (SqlConnection srcConn = new SqlConnection(connectionString))
       //   using (SqlConnection destConn = new SqlConnection(azureConnectionString)) {
       //      await srcConn.OpenAsync(cts.Token);
       //      await destConn.OpenAsync(cts.Token);
       //      using (SqlCommand srcCmd = new SqlCommand(selectStatement, srcConn)) {
       //         using (SqlDataReader reader = await srcCmd.ExecuteReaderAsync(cts.Token)) {
       //            string temptable = "[#" + Guid.NewGuid().ToString("N") + "]";
       //            using (SqlCommand destCmd = new SqlCommand(string.Format(createDestTableStatement,
temptable), destConn)) {
       //               await destCmd.ExecuteNonQueryAsync(cts.Token);
       //               using (SqlBulkCopy bcp = new SqlBulkCopy(destConn)) {
       //                  bcp.DestinationTableName = temptable;
       //                  await bcp.WriteToServerAsync(reader, cts.Token);
       //                  //Cancel Async SqlBulCopy Operation after 200 ms
       //                  cts.CancelAfter(200);
       //               }
       //            }
       //         }
       //      }
       //   }
       //}

       // 3.7 Using Async.Net and MARS
       private static async Task AsyncSqlBulkCopyMARS() {
          using (SqlConnection marsConn = new SqlConnection(marsConnectionString)) {
             await marsConn.OpenAsync();

             SqlCommand titlesCmd = new SqlCommand("SELECT * FROM [pubs].[dbo].[titles]", marsConn);
             SqlCommand authorsCmd = new SqlCommand("SELECT * FROM [pubs].[dbo].[authors]", marsConn);
             //With MARS we can have multiple active results sets on the same connection
             using (SqlDataReader titlesReader = await titlesCmd.ExecuteReaderAsync())
             using (SqlDataReader authorsReader = await authorsCmd.ExecuteReaderAsync()) {
                await authorsReader.ReadAsync();

                string temptable = "[#" + Guid.NewGuid().ToString("N") + "]";
                using (SqlConnection destConn = new SqlConnection(connectionString)) {
                   await destConn.OpenAsync();
```

```
                    await destConn.OpenAsync();
                    using (SqlCommand destCmd = new SqlCommand(string.Format(createDestTableStatement,
temptable), destConn)) {
                       await destCmd.ExecuteNonQueryAsync();
                       using (SqlBulkCopy bcp = new SqlBulkCopy(destConn)) {
                          bcp.DestinationTableName = temptable;
                          await bcp.WriteToServerAsync(titlesReader);
                       }
                    }
                 }
              }
           }
        }
     }
  }
}
```

Asynchronously Using Multiple Commands with MARS

The example opens a single connection to the **AdventureWorks** database. Using a SqlCommand object, a SqlDataReader is created. As the reader is used, a second SqlDataReader is opened, using data from the first SqlDataReader as input to the WHERE clause for the second reader.

> **NOTE**
>
> The following example uses the sample **AdventureWorks** database included with SQL Server. The connection string provided in the sample code assumes that the database is installed and available on the local computer. Modify the connection string as necessary for your environment.

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Threading.Tasks;

class Class1 {
   static void Main() {
      Task task = MultipleCommands();
      task.Wait();
   }

   static async Task MultipleCommands() {
      // By default, MARS is disabled when connecting to a MARS-enabled.
      // It must be enabled in the connection string.
      string connectionString = GetConnectionString();

      int vendorID;
      SqlDataReader productReader = null;
      string vendorSQL =
        "SELECT VendorId, Name FROM Purchasing.Vendor";
      string productSQL =
        "SELECT Production.Product.Name FROM Production.Product " +
        "INNER JOIN Purchasing.ProductVendor " +
        "ON Production.Product.ProductID = " +
        "Purchasing.ProductVendor.ProductID " +
        "WHERE Purchasing.ProductVendor.VendorID = @VendorId";

      using (SqlConnection awConnection =
        new SqlConnection(connectionString)) {
         SqlCommand vendorCmd = new SqlCommand(vendorSQL, awConnection);
         SqlCommand productCmd =
           new SqlCommand(productSQL, awConnection);

         productCmd.Parameters.Add("@VendorId", SqlDbType.Int);

         await awConnection.OpenAsync();
         using (SqlDataReader vendorReader = await vendorCmd.ExecuteReaderAsync()) {
            while (await vendorReader.ReadAsync()) {
               Console.WriteLine(vendorReader["Name"]);

               vendorID = (int)vendorReader["VendorId"];

               productCmd.Parameters["@VendorId"].Value = vendorID;
               // The following line of code requires a MARS-enabled connection.
               productReader = await productCmd.ExecuteReaderAsync();
               using (productReader) {
                  while (await productReader.ReadAsync()) {
                     Console.WriteLine("  " +
                       productReader["Name"].ToString());
                  }
               }
            }
         }
      }
   }

   private static string GetConnectionString() {
      // To avoid storing the connection string in your code, you can retrieve it from a configuration file.
      return "Data Source=(local);Integrated Security=SSPI;Initial
Catalog=AdventureWorks;MultipleActiveResultSets=True";
   }
}
```

Asynchronously Reading and Updating Data with MARS

MARS allows a connection to be used for both read operations and data manipulation language (DML) operations with more than one pending operation. This feature eliminates the need for an application to deal with connection-busy errors. In addition, MARS can replace the user of server-side cursors, which generally consume more resources. Finally, because multiple operations can operate on a single connection, they can share the same transaction context, eliminating the need to use **sp_getbindtoken** and **sp_bindsession** system stored procedures.

The following Console application demonstrates how to use two SqlDataReader objects with three SqlCommand objects and a single SqlConnection object with MARS enabled. The first command object retrieves a list of vendors whose credit rating is 5. The second command object uses the vendor ID provided from a SqlDataReader to load the second SqlDataReader with all of the products for the particular vendor. Each product record is visited by the second SqlDataReader. A calculation is performed to determine what the new **OnOrderQty** should be. The third command object is then used to update the **ProductVendor** table with the new value. This entire process takes place within a single transaction, which is rolled back at the end.

> **NOTE**
>
> The following example uses the sample **AdventureWorks** database included with SQL Server. The connection string provided in the sample code assumes that the database is installed and available on the local computer. Modify the connection string as necessary for your environment.

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Data;
using System.Data.SqlClient;
using System.Threading.Tasks;

class Program {
   static void Main() {
      Task task = ReadingAndUpdatingData();
      task.Wait();
   }

   static async Task ReadingAndUpdatingData() {
      // By default, MARS is disabled when connecting to a MARS-enabled host.
      // It must be enabled in the connection string.
      string connectionString = GetConnectionString();

      SqlTransaction updateTx = null;
      SqlCommand vendorCmd = null;
      SqlCommand prodVendCmd = null;
      SqlCommand updateCmd = null;

      SqlDataReader prodVendReader = null;

      int vendorID = 0;
      int productID = 0;
      int minOrderQty = 0;
      int maxOrderQty = 0;
      int onOrderQty = 0;
      int recordsUpdated = 0;
      int totalRecordsUpdated = 0;

      string vendorSQL =
          "SELECT VendorID, Name FROM Purchasing.Vendor " +
          "WHERE CreditRating = 5";
      string prodVendSQL =
          "SELECT ProductID, MaxOrderQty, MinOrderQty, OnOrderQty " +
          "FROM Purchasing.ProductVendor " +
          "WHERE VendorID = @VendorID";
      string updateSQL =
```

```
            "UPDATE Purchasing.ProductVendor " +
            "SET OnOrderQty = @OrderQty " +
            "WHERE ProductID = @ProductID AND VendorID = @VendorID";

        using (SqlConnection awConnection =
          new SqlConnection(connectionString)) {
           await awConnection.OpenAsync();
           updateTx = await Task.Run(() => awConnection.BeginTransaction());

           vendorCmd = new SqlCommand(vendorSQL, awConnection);
           vendorCmd.Transaction = updateTx;

           prodVendCmd = new SqlCommand(prodVendSQL, awConnection);
           prodVendCmd.Transaction = updateTx;
           prodVendCmd.Parameters.Add("@VendorId", SqlDbType.Int);

           updateCmd = new SqlCommand(updateSQL, awConnection);
           updateCmd.Transaction = updateTx;
           updateCmd.Parameters.Add("@OrderQty", SqlDbType.Int);
           updateCmd.Parameters.Add("@ProductID", SqlDbType.Int);
           updateCmd.Parameters.Add("@VendorID", SqlDbType.Int);

           using (SqlDataReader vendorReader = await vendorCmd.ExecuteReaderAsync()) {
              while (await vendorReader.ReadAsync()) {
                 Console.WriteLine(vendorReader["Name"]);

                 vendorID = (int)vendorReader["VendorID"];
                 prodVendCmd.Parameters["@VendorID"].Value = vendorID;
                 prodVendReader = await prodVendCmd.ExecuteReaderAsync();

                 using (prodVendReader) {
                    while (await prodVendReader.ReadAsync()) {
                       productID = (int)prodVendReader["ProductID"];

                       if (prodVendReader["OnOrderQty"] == DBNull.Value) {
                          minOrderQty = (int)prodVendReader["MinOrderQty"];
                          onOrderQty = minOrderQty;
                       }
                       else {
                          maxOrderQty = (int)prodVendReader["MaxOrderQty"];
                          onOrderQty = (int)(maxOrderQty / 2);
                       }

                       updateCmd.Parameters["@OrderQty"].Value = onOrderQty;
                       updateCmd.Parameters["@ProductID"].Value = productID;
                       updateCmd.Parameters["@VendorID"].Value = vendorID;

                       recordsUpdated = await updateCmd.ExecuteNonQueryAsync();
                       totalRecordsUpdated += recordsUpdated;
                    }
                 }
              }
           }
           Console.WriteLine("Total Records Updated: ", totalRecordsUpdated.ToString());
           await Task.Run(() => updateTx.Rollback());
           Console.WriteLine("Transaction Rolled Back");
        }
     }

     private static string GetConnectionString() {
        // To avoid storing the connection string in your code, you can retrieve it from a configuration file.
        return "Data Source=(local);Integrated Security=SSPI;Initial
Catalog=AdventureWorks;MultipleActiveResultSets=True";
     }
}
```

See also

- Retrieving and Modifying Data in ADO.NET

SqlClient Streaming Support

9/7/2019 • 11 minutes to read • Edit Online

Streaming support between SQL Server and an application (new in .NET Framework 4.5) supports unstructured data on the server (documents, images, and media files). A SQL Server database can store binary large objects (BLOBs), but retrieving BLOBS can use a lot of memory.

Streaming support to and from SQL Server simplifies writing applications that stream data, without having to fully load the data into memory, resulting in fewer memory overflow exceptions.

Streaming support will also enable middle-tier applications to scale better, especially in scenarios where business objects connect to SQL Azure in order to send, retrieve, and manipulate large BLOBs.

WARNING

Asynchronous calls are not supported if an application also uses the `Context Connection` connection string keyword.

The members added to support streaming are used to retrieve data from queries and to pass parameters to queries and stored procedures. The streaming feature addresses basic OLTP and data migration scenarios and is applicable to on premise and off premise data migrations.environments.

Streaming Support from SQL Server

Streaming support from SQL Server introduces new functionality in the DbDataReader and in the SqlDataReader classes in order to get Stream, XmlReader, and TextReader objects and react to them. These classes are used to retrieve data from queries. As a result, Streaming support from SQL Server addresses OLTP scenarios and applies to on-premise and off-premise environments.

The following members were added to SqlDataReader to enable streaming support from SQL Server:

1. IsDBNullAsync
2. SqlDataReader.GetFieldValue
3. GetFieldValueAsync
4. GetStream
5. GetTextReader
6. GetXmlReader

The following members were added to DbDataReader to enable streaming support from SQL Server:

1. GetFieldValue
2. GetStream
3. GetTextReader

Streaming Support to SQL Server

Streaming support to SQL Server introduces new functionality in the SqlParameter class so it can accept and react to XmlReader, Stream, and TextReader objects. SqlParameter is used to pass parameters to queries and stored procedures.

Disposing a SqlCommand object or calling Cancel must cancel any streaming operation. If an application sends CancellationToken, cancellation is not guaranteed.

The following SqlDbType types will accept a Value of Stream:

- **Binary**
- **VarBinary**

The following SqlDbType types will accept a Value of TextReader:

- **Char**
- **NChar**
- **NVarChar**
- **Xml**

The **Xml** SqlDbType type will accept a Value of XmlReader.

SqlValue can accept values of type XmlReader, TextReader, and Stream.

The XmlReader, TextReader, and Stream object will be transferred up to the value defined by the Size.

Sample -- Streaming from SQL Server

Use the following Transact-SQL to create the sample database:

```
CREATE DATABASE [Demo]
GO
USE [Demo]
GO
CREATE TABLE [Streams] (
[id] INT PRIMARY KEY IDENTITY(1, 1),
[textdata] NVARCHAR(MAX),
[bindata] VARBINARY(MAX),
[xmldata] XML)
GO
INSERT INTO [Streams] (textdata, bindata, xmldata) VALUES (N'This is a test', 0x48656C6C6F,
N'<test>value</test>')
INSERT INTO [Streams] (textdata, bindata, xmldata) VALUES (N'Hello, World!', 0x54657374696E67,
N'<test>value2</test>')
INSERT INTO [Streams] (textdata, bindata, xmldata) VALUES (N'Another row', 0x666F6F626172, N'<fff>bbb</fff>
<fff>bbc</fff>')
GO
```

The sample shows how to do the following:

- Avoid blocking a user-interface thread by providing an asynchronous way to retrieve large files.
- Transfer a large text file from SQL Server in .NET Framework 4.5.
- Transfer a large XML file from SQL Server in .NET Framework 4.5.
- Retrieve data from SQL Server.
- Transfer large files (BLOBs) from one SQL Server database to another without running out of memory.

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.IO;
using System.Threading.Tasks;
```

```
using System.Threading.Tasks;
using System.Xml;

namespace StreamingFromServer {
   class Program {
      // Replace the connection string if needed, for instance to connect to SQL Express: @"Server=
(local)\SQLEXPRESS;Database=Demo;Integrated Security=true"
      private const string connectionString = @"Server=(localdb)\V11.0;Database=Demo";

      static void Main(string[] args) {
         CopyBinaryValueToFile().Wait();
         PrintTextValues().Wait();
         PrintXmlValues().Wait();
         PrintXmlValuesViaNVarChar().Wait();

         Console.WriteLine("Done");
      }

      // Application retrieving a large BLOB from SQL Server in .NET 4.5 using the new asynchronous capability
      private static async Task CopyBinaryValueToFile() {
         string filePath = Path.Combine(Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments),
"binarydata.bin");

         using (SqlConnection connection = new SqlConnection(connectionString)) {
            await connection.OpenAsync();
            using (SqlCommand command = new SqlCommand("SELECT [bindata] FROM [Streams] WHERE [id]=@id",
connection)) {
               command.Parameters.AddWithValue("id", 1);

               // The reader needs to be executed with the SequentialAccess behavior to enable network
streaming
               // Otherwise ReadAsync will buffer the entire BLOB into memory which can cause scalability
issues or even OutOfMemoryExceptions
               using (SqlDataReader reader = await
command.ExecuteReaderAsync(CommandBehavior.SequentialAccess)) {
                  if (await reader.ReadAsync()) {
                     if (!(await reader.IsDBNullAsync(0))) {
                        using (FileStream file = new FileStream(filePath, FileMode.Create, FileAccess.Write))
{
                           using (Stream data = reader.GetStream(0)) {

                              // Asynchronously copy the stream from the server to the file we just created
                              await data.CopyToAsync(file);
                           }
                        }
                     }
                  }
               }
            }
         }
      }

      // Application transferring a large Text File from SQL Server in .NET 4.5
      private static async Task PrintTextValues() {
         using (SqlConnection connection = new SqlConnection(connectionString)) {
            await connection.OpenAsync();
            using (SqlCommand command = new SqlCommand("SELECT [id], [textdata] FROM [Streams]", connection))
{

               // The reader needs to be executed with the SequentialAccess behavior to enable network
streaming
               // Otherwise ReadAsync will buffer the entire text document into memory which can cause
scalability issues or even OutOfMemoryExceptions
               using (SqlDataReader reader = await
command.ExecuteReaderAsync(CommandBehavior.SequentialAccess)) {
                  while (await reader.ReadAsync()) {
                     Console.Write("{0}: ", reader.GetInt32(0));

                     if (await reader.IsDBNullAsync(1)) {
```

```
                        Console.Write("(NULL)");
                    }
                    else {
                        char[] buffer = new char[4096];
                        int charsRead = 0;
                        using (TextReader data = reader.GetTextReader(1)) {
                            do {
                                // Grab each chunk of text and write it to the console
                                // If you are writing to a TextWriter you should use WriteAsync or
WriteLineAsync
                                charsRead = await data.ReadAsync(buffer, 0, buffer.Length);
                                Console.Write(buffer, 0, charsRead);
                            } while (charsRead > 0);
                        }
                    }

                    Console.WriteLine();
                }
            }
        }
    }

    // Application transferring a large Xml Document from SQL Server in .NET 4.5
    private static async Task PrintXmlValues() {
        using (SqlConnection connection = new SqlConnection(connectionString)) {
            await connection.OpenAsync();
            using (SqlCommand command = new SqlCommand("SELECT [id], [xmldata] FROM [Streams]", connection)) {

                // The reader needs to be executed with the SequentialAccess behavior to enable network
streaming
                // Otherwise ReadAsync will buffer the entire Xml Document into memory which can cause
scalability issues or even OutOfMemoryExceptions
                using (SqlDataReader reader = await
command.ExecuteReaderAsync(CommandBehavior.SequentialAccess)) {
                    while (await reader.ReadAsync()) {
                        Console.WriteLine("{0}: ", reader.GetInt32(0));

                        if (await reader.IsDBNullAsync(1)) {
                            Console.WriteLine("\t(NULL)");
                        }
                        else {
                            using (XmlReader xmlReader = reader.GetXmlReader(1)) {
                                int depth = 1;
                                // NOTE: The XmlReader returned by GetXmlReader does NOT support async operations
                                // See the example below (PrintXmlValuesViaNVarChar) for how to get an XmlReader
with asynchronous capabilities
                                while (xmlReader.Read()) {
                                    switch (xmlReader.NodeType) {
                                        case XmlNodeType.Element:
                                            Console.WriteLine("{0}<{1}>", new string('\t', depth), xmlReader.Name);
                                            depth++;
                                            break;
                                        case XmlNodeType.Text:
                                            Console.WriteLine("{0}{1}", new string('\t', depth), xmlReader.Value);
                                            break;
                                        case XmlNodeType.EndElement:
                                            depth--;
                                            Console.WriteLine("{0}</{1}>", new string('\t', depth), xmlReader.Name);
                                            break;
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    }
```

```
      // Application transferring a large Xml Document from SQL Server in .NET 4.5
      // This goes via NVarChar and TextReader to enable asynchronous reading
      private static async Task PrintXmlValuesViaNVarChar() {
         XmlReaderSettings xmlSettings = new XmlReaderSettings() {
            // Async must be explicitly enabled in the XmlReaderSettings otherwise the XmlReader will throw
exceptions when async methods are called
            Async = true,
            // Since we will immediately wrap the TextReader we are creating in an XmlReader, we will permit
the XmlReader to take care of closing\disposing it
            CloseInput = true,
            // If the Xml you are reading is not a valid document (as per
<https://docs.microsoft.com/previous-versions/dotnet/netframework-4.0/6bts1x50(v=vs.100)>) you will need to
set the conformance level to Fragment
            ConformanceLevel = ConformanceLevel.Fragment
         };

         using (SqlConnection connection = new SqlConnection(connectionString)) {
            await connection.OpenAsync();

            // Cast the XML into NVarChar to enable GetTextReader - trying to use GetTextReader on an XML type
will throw an exception
            using (SqlCommand command = new SqlCommand("SELECT [id], CAST([xmldata] AS NVARCHAR(MAX)) FROM
[Streams]", connection)) {

               // The reader needs to be executed with the SequentialAccess behavior to enable network
streaming
               // Otherwise ReadAsync will buffer the entire Xml Document into memory which can cause
scalability issues or even OutOfMemoryExceptions
               using (SqlDataReader reader = await
command.ExecuteReaderAsync(CommandBehavior.SequentialAccess)) {
                  while (await reader.ReadAsync()) {
                     Console.WriteLine("{0}:", reader.GetInt32(0));

                     if (await reader.IsDBNullAsync(1)) {
                        Console.WriteLine("\t(NULL)");
                     }
                     else {
                        // Grab the row as a TextReader, then create an XmlReader on top of it
                        // We are not keeping a reference to the TextReader since the XmlReader is created
with the "CloseInput" setting (so it will close the TextReader when needed)
                        using (XmlReader xmlReader = XmlReader.Create(reader.GetTextReader(1), xmlSettings)) {
                           int depth = 1;
                           // The XmlReader above now supports asynchronous operations, so we can use
ReadAsync here
                           while (await xmlReader.ReadAsync()) {
                              switch (xmlReader.NodeType) {
                                 case XmlNodeType.Element:
                                    Console.WriteLine("{0}<{1}>", new string('\t', depth), xmlReader.Name);
                                    depth++;
                                    break;
                                 case XmlNodeType.Text:
                                    // Depending on what your data looks like, you should either use Value or
GetValueAsync
                                    // Value has less overhead (since it doesn't create a Task), but it may
also block if additional data is required
                                    Console.WriteLine("{0}{1}", new string('\t', depth), await
xmlReader.GetValueAsync());
                                    break;
                                 case XmlNodeType.EndElement:
                                    depth--;
                                    Console.WriteLine("{0}</{1}>", new string('\t', depth), xmlReader.Name);
                                    break;
                              }
                           }
                        }
                     }
                  }
               }
```

```
            }
         }
      }
   }
}
```

Sample -- Streaming to SQL Server

Use the following Transact-SQL to create the sample database:

```
CREATE DATABASE [Demo2]
GO
USE [Demo2]
GO
CREATE TABLE [BinaryStreams] (
[id] INT PRIMARY KEY IDENTITY(1, 1),
[bindata] VARBINARY(MAX))
GO
CREATE TABLE [TextStreams] (
[id] INT PRIMARY KEY IDENTITY(1, 1),
[textdata] NVARCHAR(MAX))
GO
CREATE TABLE [BinaryStreamsCopy] (
[id] INT PRIMARY KEY IDENTITY(1, 1),
[bindata] VARBINARY(MAX))
GO
```

The sample shows how to do the following:

- Transferring a large BLOB to SQL Server in .NET Framework 4.5.
- Transferring a large text file to SQL Server in .NET Framework 4.5.
- Using the new asynchronous feature to transfer a large BLOB.
- Using the new asynchronous feature and the await keyword to transfer a large BLOB.
- Cancelling the transfer of a large BLOB.
- Streaming from one SQL Server to another using the new asynchronous feature.

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.IO;
using System.Threading;
using System.Threading.Tasks;

namespace StreamingToServer {
   class Program {
      // Replace the connection string if needed, for instance to connect to SQL Express: @"Server=
(local)\SQLEXPRESS;Database=Demo2;Integrated Security=true"
      private const string connectionString = @"Server=(localdb)\V11.0;Database=Demo2";

      static void Main(string[] args) {
         CreateDemoFiles();

         StreamBLOBToServer().Wait();
         StreamTextToServer().Wait();

         // Create a CancellationTokenSource that will be cancelled after 100ms
         // Typically this token source will be cancelled by a user request (e.g. a Cancel button)
         CancellationTokenSource tokenSource = new CancellationTokenSource();
         tokenSource.CancelAfter(100);
```

```
         try {
            CancelBLOBStream(tokenSource.Token).Wait();
         }
         catch (AggregateException ex) {
            // Cancelling an async operation will throw an exception
            // Since we are using the Task's Wait method, this exception will be wrapped in an
AggregateException
            // If you were using the 'await' keyword, the compiler would take care of unwrapping the
AggregateException
            // Depending on when the cancellation occurs, you can either get an error from SQL Server or from
.Net
            if ((ex.InnerException is SqlException) || (ex.InnerException is TaskCanceledException)) {
               // This is an expected exception
               Console.WriteLine("Got expected exception: {0}", ex.InnerException.Message);
            }
            else {
               // Did not expect this exception - re-throw it
               throw;
            }
         }

         Console.WriteLine("Done");
      }

      // This is used to generate the files which are used by the other sample methods
      private static void CreateDemoFiles() {
         Random rand = new Random();
         byte[] data = new byte[1024];
         rand.NextBytes(data);

         using (FileStream file = File.Open("binarydata.bin", FileMode.Create)) {
            file.Write(data, 0, data.Length);
         }

         using (StreamWriter writer = new StreamWriter(File.Open("textdata.txt", FileMode.Create))) {
            writer.Write(Convert.ToBase64String(data));
         }
      }

      // Application transferring a large BLOB to SQL Server in .NET 4.5
      private static async Task StreamBLOBToServer() {
         using (SqlConnection conn = new SqlConnection(connectionString)) {
            await conn.OpenAsync();
            using (SqlCommand cmd = new SqlCommand("INSERT INTO [BinaryStreams] (bindata) VALUES (@bindata)",
conn)) {
               using (FileStream file = File.Open("binarydata.bin", FileMode.Open)) {

                  // Add a parameter which uses the FileStream we just opened
                  // Size is set to -1 to indicate "MAX"
                  cmd.Parameters.Add("@bindata", SqlDbType.Binary, -1).Value = file;

                  // Send the data to the server asynchronously
                  await cmd.ExecuteNonQueryAsync();
               }
            }
         }
      }

      // Application transferring a large Text File to SQL Server in .NET 4.5
      private static async Task StreamTextToServer() {
         using (SqlConnection conn = new SqlConnection(connectionString)) {
            await conn.OpenAsync();
            using (SqlCommand cmd = new SqlCommand("INSERT INTO [TextStreams] (textdata) VALUES (@textdata)",
conn)) {
               using (StreamReader file = File.OpenText("textdata.txt")) {

                  // Add a parameter which uses the StreamReader we just opened
                  // Size is set to -1 to indicate "MAX"
                  cmd.Parameters.Add("@textdata", SqlDbType.NVarChar, -1).Value = file;
```

```
                cmd.Parameters.Add("@bindata", SqlDbType.VarChar, -1).Value = file;

                // Send the data to the server asynchronously
                await cmd.ExecuteNonQueryAsync();
            }
         }
      }

      // Cancelling the transfer of a large BLOB
      private static async Task CancelBLOBStream(CancellationToken cancellationToken) {
         using (SqlConnection conn = new SqlConnection(connectionString)) {
            // We can cancel not only sending the data to the server, but also opening the connection
            await conn.OpenAsync(cancellationToken);

            // Artificially delay the command by 100ms
            using (SqlCommand cmd = new SqlCommand("WAITFOR DELAY '00:00:00:100';INSERT INTO [BinaryStreams]
(bindata) VALUES (@bindata)", conn)) {
               using (FileStream file = File.Open("binarydata.bin", FileMode.Open)) {

                  // Add a parameter which uses the FileStream we just opened
                  // Size is set to -1 to indicate "MAX"
                  cmd.Parameters.Add("@bindata", SqlDbType.Binary, -1).Value = file;

                  // Send the data to the server asynchronously
                  // Pass the cancellation token such that the command will be cancelled if needed
                  await cmd.ExecuteNonQueryAsync(cancellationToken);
               }
            }
         }
      }
   }
}
```

Sample -- Streaming From One SQL Server to Another SQL Server

This sample demonstrates how to asynchronously stream a large BLOB from one SQL Server to another, with support for cancellation.

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.IO;
using System.Threading;
using System.Threading.Tasks;

namespace StreamingFromServerToAnother {
   class Program {
      // Replace the connection string if needed, for instance to connect to SQL Express: @"Server=
(local)\SQLEXPRESS;Database=Demo2;Integrated Security=true"
      private const string connectionString = @"Server=(localdb)\V11.0;Database=Demo2";

      static void Main(string[] args) {
         // For this example, we don't want to cancel
         // So we can pass in a "blank" cancellation token
         E2EStream(CancellationToken.None).Wait();

         Console.WriteLine("Done");
      }

      // Streaming from one SQL Server to Another One using the new Async.NET
      private static async Task E2EStream(CancellationToken cancellationToken) {
         using (SqlConnection readConn = new SqlConnection(connectionString)) {
            using (SqlConnection writeConn = new SqlConnection(connectionString)) {

               // Note that we are using the same cancellation token for calls to both connections\commands
```

```
                // Also we can start both the connection opening asynchronously, and then wait for both to
complete
                Task openReadConn = readConn.OpenAsync(cancellationToken);
                Task openWriteConn = writeConn.OpenAsync(cancellationToken);
                await Task.WhenAll(openReadConn, openWriteConn);

                using (SqlCommand readCmd = new SqlCommand("SELECT [bindata] FROM [BinaryStreams]", readConn))
{
                   using (SqlCommand writeCmd = new SqlCommand("INSERT INTO [BinaryStreamsCopy] (bindata)
VALUES (@bindata)", writeConn)) {

                      // Add an empty parameter to the write command which will be used for the streams we are
copying
                      // Size is set to -1 to indicate "MAX"
                      SqlParameter streamParameter = writeCmd.Parameters.Add("@bindata", SqlDbType.Binary, -1);

                      // The reader needs to be executed with the SequentialAccess behavior to enable network
streaming
                      // Otherwise ReadAsync will buffer the entire BLOB into memory which can cause
scalability issues or even OutOfMemoryExceptions
                      using (SqlDataReader reader = await
readCmd.ExecuteReaderAsync(CommandBehavior.SequentialAccess, cancellationToken)) {
                         while (await reader.ReadAsync(cancellationToken)) {
                            // Grab a stream to the binary data in the source database
                            using (Stream dataStream = reader.GetStream(0)) {

                               // Set the parameter value to the stream source that was opened
                               streamParameter.Value = dataStream;

                               // Asynchronously send data from one database to another
                               await writeCmd.ExecuteNonQueryAsync(cancellationToken);
                            }
                         }
                      }
                   }
                }
             }
          }
       }
    }
}
```

See also

- Retrieving and Modifying Data in ADO.NET

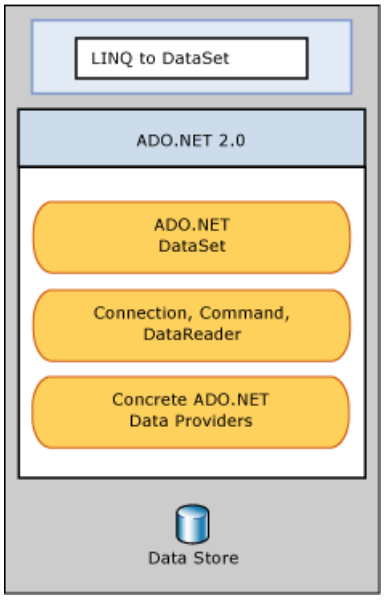
LINQ to DataSet

9/7/2019 • 2 minutes to read • Edit Online

LINQ to DataSet makes it easier and faster to query over data cached in a DataSet object. Specifically, LINQ to DataSet simplifies querying by enabling developers to write queries from the programming language itself, instead of by using a separate query language. This is especially useful for Visual Studio developers, who can now take advantage of the compile-time syntax checking, static typing, and IntelliSense support provided by the Visual Studio in their queries.

LINQ to DataSet can also be used to query over data that has been consolidated from one or more data sources. This enables many scenarios that require flexibility in how data is represented and handled, such as querying locally aggregated data and middle-tier caching in Web applications. In particular, generic reporting, analysis, and business intelligence applications require this method of manipulation.

The LINQ to DataSet functionality is exposed primarily through the extension methods in the DataRowExtensions and DataTableExtensions classes. LINQ to DataSet builds on and uses the existing ADO.NET architecture, and is not meant to replace ADO.NET in application code. Existing ADO.NET code will continue to function in a LINQ to DataSet application. The relationship of LINQ to DataSet to ADO.NET and the data store is illustrated in the following diagram.



In This Section

Getting Started

Programming Guide

Reference

DataTableExtensions

DataRowExtensions

DataRowComparer

See also

- Language-Integrated Query (LINQ) - C#
- Language-Integrated Query (LINQ) - Visual Basic
- LINQ and ADO.NET
- ADO.NET

Getting Started (LINQ to DataSet)

9/7/2019 • 2 minutes to read • Edit Online

This section provides introductory information about programming with LINQ to DataSet.

In This Section

LINQ to DataSet Overview
Provides a conceptual overview of LINQ to DataSet.

Loading Data Into a DataSet
Provides an example of populating a DataSet. This example uses DataAdapter to retrieve data from a database.

Downloading Sample Databases
Provides information about downloading the AdventureWorks sample database, which is used in the samples throughout the LINQ to DataSet section.

How to: Create a LINQ to DataSet Project In Visual Studio
Provides information about creating a LINQ to DataSet project in Visual Studio.

Reference

DataRowComparer

DataRowExtensions

DataTableExtensions

See also

- LINQ and ADO.NET
- Language-Integrated Query (LINQ) - C#
- Language-Integrated Query (LINQ) - Visual Basic

LINQ to DataSet Overview

1/3/2020 • 3 minutes to read • Edit Online

The DataSet is one of the more widely used components of ADO.NET. It is a key element of the disconnected programming model that ADO.NET is based on, and it enables you to explicitly cache data from different data sources. For the presentation tier, the DataSet is tightly integrated with GUI controls for data-binding. For the middle-tier, it provides a cache that preserves the relational shape of data, and includes fast simple query and hierarchy navigation services. A common technique used to lower the number of requests on a database is to use the DataSet for caching in the middle-tier. For example, consider a data-driven ASP.NET Web application. Often, a significant portion of the application data does not change frequently and is common across sessions or users. This data can be kept in memory on the Web server, which reduces the number of requests against the database and speeds up the user's interactions. Another useful aspect of the DataSet is that it allows an application to bring subsets of data from one or more data source into the application space. The application can then manipulate the data in-memory, while retaining its relational shape.

Despite its prominence, the DataSet has limited query capabilities. The Select method can be used for filtering and sorting, and the GetChildRows and GetParentRow methods can be used for hierarchy navigation. For anything more complex, however, the developer must write a custom query. This can result in applications that perform poorly and are difficult to maintain.

LINQ to DataSet makes it easier and faster to query over data cached in a DataSet object. These queries are expressed in the programming language itself, rather than as string literals embedded in the application code. This means that developers do not have to learn a separate query language. Additionally, LINQ to DataSet enables Visual Studio developers to work more productively, because the Visual Studio IDE provides compile-time syntax checking, static typing, and IntelliSense support for LINQ. LINQ to DataSet can also be used to query over data that has been consolidated from one or more data sources. This enables many scenarios that require flexibility in how data is represented and handled. In particular, generic reporting, analysis, and business intelligence applications require this method of manipulation.

Querying DataSets Using LINQ to DataSet

Before you can begin querying a DataSet object using LINQ to DataSet, you must populate the DataSet. There are several ways to load data into a DataSet, such as using the DataAdapter class or LINQ to SQL. After the data has been loaded into a DataSet object, you can begin to query it. Formulating queries using LINQ to DataSet is similar to using Language-Integrated Query (LINQ) against other LINQ-enabled data sources. LINQ queries can be performed against single tables in a DataSet or against more than one table by using the Join and GroupJoin standard query operators.

LINQ queries are supported against both typed and untyped DataSet objects. If the schema of the DataSet is known at application design time, a typed DataSet is recommended. In a typed DataSet, the tables and rows have typed members for each of the columns, which makes queries simpler and more readable.

In addition to the standard query operators implemented in System.Core.dll, LINQ to DataSet adds several DataSet-specific extensions that make it easier to query over a set of DataRow objects. These DataSet-specific extensions include operators for comparing sequences of rows, as well as methods that provide access to the column values of a DataRow.

N-tier Applications and LINQ to DataSet

N-tier data applications are data-centric applications that are separated into multiple logical layers (or tiers). A typical N-tier application includes a presentation tier, a middle tier, and a data tier. Separating application

components into separate tiers increases the maintainability and scalability of the application. For more information about N-tier data applications, see Work with datasets in n-tier applications.

In N-tier applications, the DataSet is often used in the middle-tier to cache information for a Web application. The LINQ to DataSet querying functionality is implemented through extension methods and extends the existing ADO.NET 2.0 DataSet.

See also

- Querying DataSets
- Language-Integrated Query (LINQ) - C#
- Language-Integrated Query (LINQ) - Visual Basic
- LINQ to SQL

Loading Data Into a DataSet

9/7/2019 • 3 minutes to read • Edit Online

A DataSet object must first be populated before you can query over it with LINQ to DataSet. There are several different ways to populate the DataSet. For example, you can use LINQ to SQL to query the database and load the results into the DataSet. For more information, see LINQ to SQL.

Another common way to load data into a DataSet is to use the DataAdapter class to retrieve data from the database. This is illustrated in the following example.

Example

This example uses a DataAdapter to query the AdventureWorks database for sales information from the year 2002, and loads the results into a DataSet. After the DataSet has been populated, you can write queries against it by using LINQ to DataSet. The `FillDataSet` method in this example is used in the example queries in LINQ to DataSet Examples. For more information, see Querying DataSets.

```
try
{
    // Create a new adapter and give it a query to fetch sales order, contact,
    // address, and product information for sales in the year 2002. Point connection
    // information to the configuration setting "AdventureWorks".
    string connectionString = "Data Source=localhost;Initial Catalog=AdventureWorks;"
        + "Integrated Security=true;";

    SqlDataAdapter da = new SqlDataAdapter(
        "SELECT SalesOrderID, ContactID, OrderDate, OnlineOrderFlag, " +
        "TotalDue, SalesOrderNumber, Status, ShipToAddressID, BillToAddressID " +
        "FROM Sales.SalesOrderHeader " +
        "WHERE DATEPART(YEAR, OrderDate) = @year; " +

        "SELECT d.SalesOrderID, d.SalesOrderDetailID, d.OrderQty, " +
        "d.ProductID, d.UnitPrice " +
        "FROM Sales.SalesOrderDetail d " +
        "INNER JOIN Sales.SalesOrderHeader h " +
        "ON d.SalesOrderID = h.SalesOrderID  " +
        "WHERE DATEPART(YEAR, OrderDate) = @year; " +

        "SELECT p.ProductID, p.Name, p.ProductNumber, p.MakeFlag, " +
        "p.Color, p.ListPrice, p.Size, p.Class, p.Style, p.Weight  " +
        "FROM Production.Product p; " +

        "SELECT DISTINCT a.AddressID, a.AddressLine1, a.AddressLine2, " +
        "a.City, a.StateProvinceID, a.PostalCode " +
        "FROM Person.Address a " +
        "INNER JOIN Sales.SalesOrderHeader h " +
        "ON  a.AddressID = h.ShipToAddressID OR a.AddressID = h.BillToAddressID " +
        "WHERE DATEPART(YEAR, OrderDate) = @year; " +

        "SELECT DISTINCT c.ContactID, c.Title, c.FirstName, " +
        "c.LastName, c.EmailAddress, c.Phone " +
        "FROM Person.Contact c " +
        "INNER JOIN Sales.SalesOrderHeader h " +
        "ON c.ContactID = h.ContactID " +
        "WHERE DATEPART(YEAR, OrderDate) = @year;",
    connectionString);

    // Add table mappings.
    da.SelectCommand.Parameters.AddWithValue("@year", 2002);
```

```
    da.TableMappings.Add("Table", "SalesOrderHeader");
    da.TableMappings.Add("Table1", "SalesOrderDetail");
    da.TableMappings.Add("Table2", "Product");
    da.TableMappings.Add("Table3", "Address");
    da.TableMappings.Add("Table4", "Contact");

    // Fill the DataSet.
    da.Fill(ds);

    // Add data relations.
    DataTable orderHeader = ds.Tables["SalesOrderHeader"];
    DataTable orderDetail = ds.Tables["SalesOrderDetail"];
    DataRelation order = new DataRelation("SalesOrderHeaderDetail",
                            orderHeader.Columns["SalesOrderID"],
                            orderDetail.Columns["SalesOrderID"], true);
    ds.Relations.Add(order);

    DataTable contact = ds.Tables["Contact"];
    DataTable orderHeader2 = ds.Tables["SalesOrderHeader"];
    DataRelation orderContact = new DataRelation("SalesOrderContact",
                                    contact.Columns["ContactID"],
                                    orderHeader2.Columns["ContactID"], true);
    ds.Relations.Add(orderContact);
}
catch (SqlException ex)
{
    Console.WriteLine("SQL exception occurred: " + ex.Message);
}
```

```
Try
    Dim connectionString As String

    connectionString = "Data Source=localhost;Initial Catalog=AdventureWorks;" & _
    "Integrated Security=true;"

    ' Create a new adapter and give it a query to fetch sales order, contact,
    ' address, and product information for sales in the year 2002. Point connection
    ' information to the configuration setting "AdventureWorks".
    Dim da = New SqlDataAdapter( _
    "SELECT SalesOrderID, ContactID, OrderDate, OnlineOrderFlag, " & _
        "TotalDue, SalesOrderNumber, Status, ShipToAddressID, BillToAddressID " & _
        "FROM Sales.SalesOrderHeader " & _
        "WHERE DATEPART(YEAR, OrderDate) = @year; " & _
        "SELECT d.SalesOrderID, d.SalesOrderDetailID, d.OrderQty, " & _
        "d.ProductID, d.UnitPrice " & _
        "FROM Sales.SalesOrderDetail d " & _
        "INNER JOIN Sales.SalesOrderHeader h " & _
        "ON d.SalesOrderID = h.SalesOrderID  " & _
        "WHERE DATEPART(YEAR, OrderDate) = @year; " & _
        "SELECT p.ProductID, p.Name, p.ProductNumber, p.MakeFlag, " & _
        "p.Color, p.ListPrice, p.Size, p.Class, p.Style  " & _
        "FROM Production.Product p; " & _
        "SELECT DISTINCT a.AddressID, a.AddressLine1, a.AddressLine2, " & _
        "a.City, a.StateProvinceID, a.PostalCode " & _
        "FROM Person.Address a " & _
        "INNER JOIN Sales.SalesOrderHeader h " & _
        "ON  a.AddressID = h.ShipToAddressID OR a.AddressID = h.BillToAddressID " & _
        "WHERE DATEPART(YEAR, OrderDate) = @year; " & _
        "SELECT DISTINCT c.ContactID, c.Title, c.FirstName, " & _
        "c.LastName, c.EmailAddress, c.Phone " & _
        "FROM Person.Contact c " & _
        "INNER JOIN Sales.SalesOrderHeader h " & _
        "ON c.ContactID = h.ContactID " & _
        "WHERE DATEPART(YEAR, OrderDate) = @year;", _
    connectionString)

    ' Add table mappings.
    da.SelectCommand.Parameters.AddWithValue("@year", 2002)
    da.TableMappings.Add("Table", "SalesOrderHeader")
    da.TableMappings.Add("Table1", "SalesOrderDetail")
    da.TableMappings.Add("Table2", "Product")
    da.TableMappings.Add("Table3", "Address")
    da.TableMappings.Add("Table4", "Contact")

    da.Fill(ds)

    ' Add data relations.
    Dim orderHeader As DataTable = ds.Tables("SalesOrderHeader")
    Dim orderDetail As DataTable = ds.Tables("SalesOrderDetail")
    Dim co As DataRelation = New DataRelation("SalesOrderHeaderDetail", _
                                orderHeader.Columns("SalesOrderID"), _
                                orderDetail.Columns("SalesOrderID"), True)
    ds.Relations.Add(co)

    Dim contact As DataTable = ds.Tables("Contact")
    Dim orderContact As DataRelation = New DataRelation("SalesOrderContact", _
                                contact.Columns("ContactID"), _
                                orderHeader.Columns("ContactID"), True)
    ds.Relations.Add(orderContact)
Catch ex As SqlException
    Console.WriteLine("SQL exception occurred: " & ex.Message)
End Try
```

See also

- LINQ to DataSet Overview
- Querying DataSets
- LINQ to DataSet Examples

Download sample databases (LINQ to DataSet)

4/10/2020 • 2 minutes to read • Edit Online

The samples and walkthroughs in the LINQ to DataSet documentation use the AdventureWorks sample database. You can download this sample database free of charge. Browse to AdventureWorks sample databases to download the database. Then, follow the instructions on that page for attaching or restoring a database.

SQL Server Express

The samples and walkthroughs in the LINQ to DataSet section use SQL Server 2005 as the data store, but you can modify them to use SQL Server Express instead. SQL Server Express is free, and you can redistribute it with applications.

See also

- Getting Started

How to: Create a LINQ to DataSet project in Visual Studio

12/4/2019 • 2 minutes to read • Edit Online

The different types of LINQ projects require certain assembly references and imported namespaces (Visual Basic) or using directives (C#). The minimum requirement for LINQ is a reference to *System.Core.dll* and a `using` directive for System.Linq.

These requirements are supplied by default if you create a new C# console app project in Visual Studio 2017 or a later version. If you're upgrading a project from an earlier version of Visual Studio, you might have to supply these LINQ-related references manually.

LINQ to DataSet requires two additional references to *System.Data.dll* and *System.Data.DataSetExtensions.dll*.

> **NOTE**
> If you're building from a command prompt, you must manually reference the LINQ-related DLLs in *%ProgramFiles%\Reference Assemblies\Microsoft\Framework\v3.5*.

To enable LINQ to DataSet functionality

Follow these steps to enable LINQ to DataSet functionality in an existing project.

1. Add references to **System.Core**, **System.Data**, and **System.Data.DataSetExtensions**.

   In **Solution Explorer**, right-click on the **References** node and select **Add Reference**. In the **Reference Manager** dialog box, select **System.Core**, **System.Data**, and **System.Data.DataSetExtensions**. Select **OK**.

2. Add using directives (or Imports statements in Visual Basic) for **System.Data** and **System.Linq**.

   ```
   using System.Data;
   using System.Linq;
   ```

3. Optionally, add a `using` directive (or `Imports` statement) for **System.Data.Common** or **System.Data.SqlClient**, depending on how you connect to the database.

See also

- Get started with LINQ to DataSet

Programming Guide (LINQ to DataSet)

1/3/2020 • 2 minutes to read • Edit Online

This section provides conceptual information and examples for programming with LINQ to DataSet.

In This Section

Queries in LINQ to DataSet
Provides information about how to write LINQ to DataSet queries.

Querying DataSets
Describes how to query DataSet objects.

Comparing DataRows
Describes how to use the DataRowComparer object to compare data rows.

Creating a DataTable From a Query
Provides information about creating a DataTable from a LINQ to DataSet query by using the CopyToDataTable method.

How to: Implement CopyToDataTable<T> Where the Generic Type T Is Not a DataRow
Describes how to implement a custom `CopyToDataTable<T>` method where the generic parameter T is not of type DataRow.

Generic Field and SetField Methods
Provides information about the generic Field and SetField methods.

Data Binding and LINQ to DataSet
Describes databinding using the DataView object.

Debugging LINQ to DataSet Queries
Provides information about debugging and troubleshooting LINQ to DataSet queries.

Security
Describes security issues in LINQ to DataSet.

LINQ to DataSet Examples
Provides query examples that use the LINQ operators.

Reference

DataRowComparer

DataRowExtensions

DataTableExtensions

DataView

See also

- LINQ and ADO.NET
- Language Integrated Query (LINQ)

Queries in LINQ to DataSet

1/3/2020 • 7 minutes to read • Edit Online

A query is an expression that retrieves data from a data source. Queries are usually expressed in a specialized query language, such as SQL for relational databases and XQuery for XML. Therefore, developers have had to learn a new query language for each type of data source or data format that they query. Language-Integrated Query (LINQ) offers a simpler, consistent model for working with data across various kinds of data sources and formats. In a LINQ query, you always work with programming objects.

A LINQ query operation consists of three actions: obtain the data source or sources, create the query, and execute the query.

Data sources that implement the IEnumerable<T> generic interface can be queried through LINQ. Calling AsEnumerable on a DataTable returns an object which implements the generic IEnumerable<T> interface, which serves as the data source for LINQ to DataSet queries.

In the query, you specify exactly the information that you want to retrieve from the data source. A query can also specify how that information should be sorted, grouped, and shaped before it is returned. In LINQ, a query is stored in a variable. If the query is designed to return a sequence of values, the query variable itself must be a enumerable type. This query variable takes no action and returns no data; it only stores the query information. After you create a query you must execute that query to retrieve any data.

In a query that returns a sequence of values, the query variable itself never holds the query results and only stores the query commands. Execution of the query is deferred until the query variable is iterated over in a `foreach` or `For Each` loop. This is called *deferred execution*; that is, query execution occurs some time after the query is constructed. This means that you can execute a query as often as you want to. This is useful when, for example, you have a database that is being updated by other applications. In your application, you can create a query to retrieve the latest information and repeatedly execute the query, returning the updated information every time.

In contrast to deferred queries, which return a sequence of values, queries that return a singleton value are executed immediately. Some examples of singleton queries are Count, Max, Average, and First. These execute immediately because the query results are required to calculate the singleton result. For example, in order to find the average of the query results the query must be executed so that the averaging function has input data to work with. You can also use the ToList or ToArray methods on a query to force immediate execution of a query that does not produce a singleton value. These techniques to force immediate execution can be useful when you want to cache the results of a query.

Queries

LINQ to DataSet queries can be formulated in two different syntaxes: query expression syntax and method-based query syntax.

Query Expression Syntax

Query expressions are a declarative query syntax. This syntax enables a developer to write queries in C# or Visual Basic in a format similar to SQL. By using query expression syntax, you can perform even complex filtering, ordering, and grouping operations on data sources with minimal code. For more information, see LINQ Query Expressions and Basic Query Operations (Visual Basic).

The .NET Framework common language runtime (CLR) cannot read the query expression syntax itself. Therefore, at compile time, query expressions are translated to something that the CLR does understand: method calls. These methods are referred to as the *standard query operators*. As a developer, you have the option of calling them

directly by using method syntax, instead of using query syntax. For more information, see Query Syntax and Method Syntax in LINQ. For more information about the standard query operators, see Standard Query Operators Overview.

The following example uses Select to return all the rows from `Product` table and display the product names.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable products = ds.Tables["Product"];

IEnumerable<DataRow> query =
    from product in products.AsEnumerable()
    select product;

Console.WriteLine("Product Names:");
foreach (DataRow p in query)
{
    Console.WriteLine(p.Field<string>("Name"));
}
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim products As DataTable = ds.Tables("Product")

Dim query = From product In products.AsEnumerable() _
            Select product
Console.WriteLine("Product Names:")
For Each p In query
    Console.WriteLine(p.Field(Of String)("Name"))
Next
```

Method-Based Query Syntax

The other way to formulate LINQ to DataSet queries is by using method-based queries. The method-based query syntax is a sequence of direct method calls to LINQ operator methods, passing lambda expressions as the parameters. For more information, see Lambda Expressions.

This example uses Select to return all the rows from `Product` and display the product names.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable products = ds.Tables["Product"];

var query = products.AsEnumerable().
    Select(product => new
    {
        ProductName = product.Field<string>("Name"),
        ProductNumber = product.Field<string>("ProductNumber"),
        Price = product.Field<decimal>("ListPrice")
    });

Console.WriteLine("Product Info:");
foreach (var productInfo in query)
{
    Console.WriteLine("Product name: {0} Product number: {1} List price: ${2} ",
        productInfo.ProductName, productInfo.ProductNumber, productInfo.Price);
}
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim products As DataTable = ds.Tables("Product")

Dim query = products.AsEnumerable() _
    .Select(Function(product As DataRow) New With _
    { _
        .ProductName = product.Field(Of String)("Name"), _
        .ProductNumber = product.Field(Of String)("ProductNumber"), _
        .Price = product.Field(Of Decimal)("ListPrice") _
    })

Console.WriteLine("Product Info:")
For Each product In query
    Console.Write("Product name: " & product.ProductName)
    Console.Write("Product number: " & product.ProductNumber)
    Console.WriteLine("List price: $ " & product.Price)
Next
```

Composing Queries

As mentioned earlier in this topic, the query variable itself only stores the query commands when the query is designed to return a sequence of values. If the query does not contain a method that will cause immediate execution, the actual execution of the query is deferred until you iterate over the query variable in a `foreach` or `For Each` loop. Deferred execution enables multiple queries to be combined or a query to be extended. When a query is extended, it is modified to include the new operations, and the eventual execution will reflect the changes. In the following example, the first query returns all the products. The second query extends the first by using `Where` to return all the products of size "L":

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable products = ds.Tables["Product"];

IEnumerable<DataRow> productsQuery =
    from product in products.AsEnumerable()
    select product;

IEnumerable<DataRow> largeProducts =
    productsQuery.Where(p => p.Field<string>("Size") == "L");

Console.WriteLine("Products of size 'L':");
foreach (DataRow product in largeProducts)
{
    Console.WriteLine(product.Field<string>("Name"));
}
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim products As DataTable = ds.Tables("Product")

Dim productsQuery = From product In products.AsEnumerable() _
            Select product

Dim largeProducts = _
    productsQuery.Where(Function(p) p.Field(Of String)("Size") = "L")

Console.WriteLine("Products of size 'L':")
For Each product In largeProducts
    Console.WriteLine(product.Field(Of String)("Name"))
Next
```

After a query has been executed, no additional queries can be composed, and all subsequent queries will use the in-memory LINQ operators. Query execution will occur when you iterate over the query variable in a `foreach` or `For Each` statement, or by a call to one of the LINQ conversion operators that cause immediate execution. These operators include the following: ToList, ToArray, ToLookup, and ToDictionary.

In the following example, the first query returns all the products ordered by list price. The ToArray method is used to force immediate query execution:

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable products = ds.Tables["Product"];

IEnumerable<DataRow> query =
    from product in products.AsEnumerable()
    orderby product.Field<Decimal>("ListPrice") descending
    select product;

// Force immediate execution of the query.
IEnumerable<DataRow> productsArray = query.ToArray();

Console.WriteLine("Every price from highest to lowest:");
foreach (DataRow prod in productsArray)
{
    Console.WriteLine(prod.Field<Decimal>("ListPrice"));
}
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim products As DataTable = ds.Tables("Product")

Dim query = _
        From product In products.AsEnumerable() _
        Order By product.Field(Of Decimal)("ListPrice") Descending _
        Select product

' Force immediate execution of the query.
Dim productsArray = query.ToArray()

Console.WriteLine("Every price From highest to lowest:")
For Each prod In productsArray
    Console.WriteLine(prod.Field(Of Decimal)("ListPrice"))
Next
```

See also

- Programming Guide
- Querying DataSets
- Getting Started with LINQ in C#
- Getting Started with LINQ in Visual Basic

Querying DataSets (LINQ to DataSet)

1/3/2020 • 2 minutes to read • Edit Online

After a DataSet object has been populated with data, you can begin querying it. Formulating queries with LINQ to DataSet is similar to using Language-Integrated Query (LINQ) against other LINQ-enabled data sources. Remember, however, that when you use LINQ queries over a DataSet object, you're querying an enumeration of DataRow objects instead of an enumeration of a custom type. This means that you can use any of the members of the DataRow class in your LINQ queries. This lets you create rich, complex queries.

As with other implementations of LINQ, you can create LINQ to DataSet queries in two different forms: query expression syntax and method-based query syntax. You can use query expression syntax or method-based query syntax to perform queries against single tables in a DataSet, against multiple tables in a DataSet, or against tables in a typed DataSet.

In This Section

Single-Table Queries
Describes how to perform single-table queries.

Cross-Table Queries
Describes how to perform cross-table queries.

Querying Typed DataSets
Describes how to query typed DataSet objects.

See also

- LINQ to DataSet Examples
- Loading Data Into a DataSet

Single-Table Queries (LINQ to DataSet)

3/25/2020 • 3 minutes to read • Edit Online

Language-Integrated Query (LINQ) queries work on data sources that implement the IEnumerable<T> interface or the IQueryable<T> interface. The DataTable class does not implement either interface, so you must call the AsEnumerable method if you want to use the DataTable as a source in the `From` clause of a LINQ query.

The following example gets all the online orders from the SalesOrderHeader table and outputs the order ID, order date, and order number to the console.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable orders = ds.Tables["SalesOrderHeader"];

var query =
    from order in orders.AsEnumerable()
    where order.Field<bool>("OnlineOrderFlag") == true
    select new
    {
        SalesOrderID = order.Field<int>("SalesOrderID"),
        OrderDate = order.Field<DateTime>("OrderDate"),
        SalesOrderNumber = order.Field<string>("SalesOrderNumber")
    };

foreach (var onlineOrder in query)
{
    Console.WriteLine("Order ID: {0} Order date: {1:d} Order number: {2}",
        onlineOrder.SalesOrderID,
        onlineOrder.OrderDate,
        onlineOrder.SalesOrderNumber);
}
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim orders As DataTable = ds.Tables("SalesOrderHeader")

Dim query = _
    From order In orders.AsEnumerable() _
    Where order.Field(Of Boolean)("OnlineOrderFlag") = True _
    Select New With { _
        .SalesOrderID = order.Field(Of Integer)("SalesOrderID"), _
        .OrderDate = order.Field(Of DateTime)("OrderDate"), _
        .SalesOrderNumber = order.Field(Of String)("SalesOrderNumber") _
     }

For Each onlineOrder In query
    Console.Write("Order ID: " & onlineOrder.SalesOrderID)
    Console.Write(" Order date: " & onlineOrder.OrderDate)
    Console.WriteLine(" Order number: " & onlineOrder.SalesOrderNumber)
Next
```

The local variable query is initialized with a query expression, which operates on one or more information sources by applying one or more query operators from either the standard query operators or, in the case of LINQ to DataSet, operators specific to the DataSet class. The query expression in the previous example uses two of the standard query operators: `Where` and `Select`.

The `Where` clause filters the sequence based on a condition, in this case that the `OnlineOrderFlag` is set to `true`. The `Select` operator allocates and returns an enumerable object that captures the arguments passed to the operator. In this above example, an anonymous type is created with three properties: `SalesOrderID`, `OrderDate`, and `SalesOrderNumber`. The values of these three properties are set to the values of the `SalesOrderID`, `OrderDate`, and `SalesOrderNumber` columns from the `SalesOrderHeader` table.

The `foreach` loop then enumerates the enumerable object returned by `Select` and yields the query results. Because query is an Enumerable type, which implements IEnumerable<T>, the evaluation of the query is deferred until the query variable is iterated over using the `foreach` loop. Deferred query evaluation allows queries to be kept as values that can be evaluated multiple times, each time yielding potentially different results.

The Field method provides access to the column values of a DataRow and the SetField (not shown in the previous example) sets column values in a DataRow. Both the Field method and SetField method handle nullable value types, so you do not have to explicitly check for null values. Both methods are generic methods, also, which means you do not have to cast the return type. You could use the pre-existing column accessor in DataRow (for example, `o["OrderDate"]`), but doing so would require you to cast the return object to the appropriate type. If the column is a nullable value type you have to check if the value is null by using the IsNull method. For more information, see Generic Field and SetField Methods.

Note that the data type specified in the generic parameter `T` of the Field method and SetField method must match the type of the underlying value or an InvalidCastException will be thrown. The specified column name must also match the name of a column in the DataSet or an ArgumentException will be thrown. In both cases, the exception is thrown at run time data enumeration when the query is executed.

See also

- Cross-Table Queries
- Querying Typed DataSets
- Generic Field and SetField Methods

Cross-Table Queries (LINQ to DataSet)

1/3/2020 • 2 minutes to read • Edit Online

In addition to querying a single table, you can also perform cross-table queries in LINQ to DataSet. This is done by using a *join*. A join is the association of objects in one data source with objects that share a common attribute in another data source, such as a product or contact ID. In object-oriented programming, relationships between objects are relatively easy to navigate because each object has a member that references another object. In external database tables, however, navigating relationships is not as straightforward. Database tables do not contain built-in relationships. In these cases, the join operation can be used to match elements from each source. For example, given two tables that contain product information and sales information, you could use a join operation to match sales information and products for the same sales order.

The Language-Integrated Query (LINQ) framework provides two join operators, Join and GroupJoin. These operators perform *equi-joins*: that is, joins that match two data sources only when their keys are equal. (By contrast, Transact-SQL supports join operators other than `equals`, such as the `less than` operator.)

In relational database terms, Join implements an inner join. An inner join is a type of join in which only those objects that have a match in the opposite data set are returned.

The GroupJoin operators have no direct equivalent in relational database terms; they implement a superset of inner joins and left outer joins. A left outer join is a join that returns each element of the first (left) collection, even if it has no correlated elements in the second collection.

For more information about joins, see Join Operations.

Example

The following example performs a traditional join of the `SalesOrderHeader` and `SalesOrderDetail` tables from the AdventureWorks sample database to obtain online orders from the month of August.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable orders = ds.Tables["SalesOrderHeader"];
DataTable details = ds.Tables["SalesOrderDetail"];

var query =
    from order in orders.AsEnumerable()
    join detail in details.AsEnumerable()
    on order.Field<int>("SalesOrderID") equals
        detail.Field<int>("SalesOrderID")
    where order.Field<bool>("OnlineOrderFlag") == true
    && order.Field<DateTime>("OrderDate").Month == 8
    select new
    {
        SalesOrderID =
            order.Field<int>("SalesOrderID"),
        SalesOrderDetailID =
            detail.Field<int>("SalesOrderDetailID"),
        OrderDate =
            order.Field<DateTime>("OrderDate"),
        ProductID =
            detail.Field<int>("ProductID")
    };

foreach (var order in query)
{
    Console.WriteLine("{0}\t{1}\t{2:d}\t{3}",
        order.SalesOrderID,
        order.SalesOrderDetailID,
        order.OrderDate,
        order.ProductID);
}
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim orders As DataTable = ds.Tables("SalesOrderHeader")
Dim details As DataTable = ds.Tables("SalesOrderDetail")


Dim query = _
    From order In orders.AsEnumerable() _
    Join detail In details.AsEnumerable() _
    On order.Field(Of Integer)("SalesOrderID") Equals _
            detail.Field(Of Integer)("SalesOrderID") _
    Where order.Field(Of Boolean)("OnlineOrderFlag") = True And _
            order.Field(Of DateTime)("OrderDate").Month = 8 _
    Select New With _
    { _
        .SalesOrderID = order.Field(Of Integer)("SalesOrderID"), _
        .SalesOrderDetailID = detail.Field(Of Integer)("SalesOrderDetailID"), _
        .OrderDate = order.Field(Of DateTime)("OrderDate"), _
        .ProductID = detail.Field(Of Integer)("ProductID") _
    }

For Each order In query
    Console.WriteLine(order.SalesOrderID & vbTab & _
        order.SalesOrderDetailID & vbTab & _
        order.OrderDate & vbTab & _
        order.ProductID)
Next
```

See also

- Querying DataSets
- Single-Table Queries
- Querying Typed DataSets
- Join Operations
- LINQ to DataSet Examples

Query typed DataSets

9/7/2019 • 2 minutes to read • Edit Online

If the schema of the DataSet is known at application design time, we recommend that you use a typed DataSet when using LINQ to DataSet. A typed DataSet is a class that derives from a DataSet. As such, it inherits all the methods, events, and properties of a DataSet. Additionally, a typed DataSet provides strongly typed methods, events, and properties. This means that you can access tables and columns by name, instead of using collection-based methods. This makes queries simpler and more readable. For more information, see Typed DataSets.

LINQ to DataSet also supports querying over a typed DataSet. With a typed DataSet, you do not have to use the generic Field method or SetField method to access column data. Property names are available at compile time because the type information is included in the DataSet. LINQ to DataSet provides access to column values as the correct type, so that type mismatch errors are caught when the code is compiled instead of at run time.

Before you can begin querying a typed DataSet, you must generate the class by using the **DataSet Designer** in Visual Studio. For more information, see Create and configure DataSets.

Example

The following example shows a query over a typed DataSet:

```
var query = from o in orders
            where o.OnlineOrderFlag == true
            select new { o.SalesOrderID,
                         o.OrderDate,
                         o.SalesOrderNumber };

foreach(var order in query)
{
    Console.WriteLine("{0}\t{1:d}\t{2}",
      order.SalesOrderID,
      order.OrderDate,
      order.SalesOrderNumber);
}
```

```
Dim orders = ds.Tables("SalesOrderHeader")

Dim query = _
       From o In orders _
       Where o.OnlineOrderFlag = True _
       Select New {SalesOrderID := o.SalesOrderID, _
                   OrderDate := o.OrderDate, _
                   SalesOrderNumber := o.SalesOrderNumber}

For Each Dim onlineOrder In query
 Console.WriteLine("{0}\t{1:d}\t{2}", _
 onlineOrder.SalesOrderID, _
 onlineOrder.OrderDate, _
 onlineOrder.SalesOrderNumber)
Next
```

See also

- Querying DataSets

- Cross-Table Queries
- Single-Table Queries

Comparing DataRows (LINQ to DataSet)

1/3/2020 • 2 minutes to read • Edit Online

Language-Integrated Query (LINQ) defines various set operators to compare source elements to see if they are equal. LINQ provides the following set operators:

- Distinct
- Union
- Intersect
- Except

These operators compare source elements by calling the GetHashCode and Equals methods on each collection of elements. In the case of a DataRow, these operators perform a reference comparison, which is generally not the ideal behavior for set operations over tabular data. For set operations, you usually want to determine whether the element values are equal and not the element references. Therefore, the DataRowComparer class has been added to LINQ to DataSet. This class can be used to compare row values.

The DataRowComparer class contains a value comparison implementation for DataRow, so this class can be used for set operations such as Distinct. This class cannot be directly instantiated; instead, the Default property must be used to return an instance of the DataRowComparer<TRow>. The Equals method is then called and the two DataRow objects to be compared are passed in as input parameters. The Equals method returns `true` if the ordered set of column values in both DataRow objects are equal; otherwise, `false`.

Example

This example uses `Intersect` to return contacts that appear in both tables.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable contactTable = ds.Tables["Contact"];

// Create two tables.
IEnumerable<DataRow> query1 = from contact in contactTable.AsEnumerable()
                              where contact.Field<string>("Title") == "Ms."
                              select contact;

IEnumerable<DataRow> query2 = from contact in contactTable.AsEnumerable()
                              where contact.Field<string>("FirstName") == "Sandra"
                              select contact;

DataTable contacts1 = query1.CopyToDataTable();
DataTable contacts2 = query2.CopyToDataTable();

// Find the intersection of the two tables.
var contacts = contacts1.AsEnumerable().Intersect(contacts2.AsEnumerable(),
                                                  DataRowComparer.Default);

Console.WriteLine("Intersection of contacts tables");
foreach (DataRow row in contacts)
{
    Console.WriteLine("Id: {0} {1} {2} {3}",
        row["ContactID"], row["Title"], row["FirstName"], row["LastName"]);
}
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim contactTable As DataTable = ds.Tables("Contact")

Dim query1 = _
    From contact In contactTable.AsEnumerable() _
    Where contact.Field(Of String)("Title") = "Ms." _
    Select contact

Dim query2 = _
    From contact In contactTable.AsEnumerable() _
    Where contact.Field(Of String)("FirstName") = "Sandra" _
    Select contact

Dim contacts1 = query1.CopyToDataTable()
Dim contacts2 = query2.CopyToDataTable()

Dim contacts = contacts1.AsEnumerable() _
    .Intersect(contacts2.AsEnumerable(), DataRowComparer.Default)

Console.WriteLine("Intersect of employees tables")

For Each row In contacts
    Console.WriteLine("Id: {0} {1} {2} {3}", _
            row("ContactID"), row("Title"), row("FirstName"), row("LastName"))
Next
```

Example

The following example compares two rows and gets their hash codes.

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

' Get two rows from the SalesOrderHeader table.
Dim table As DataTable = ds.Tables("SalesOrderHeader")
Dim left = table.Rows(0)
Dim right = table.Rows(1)

' Compare the two different rows.
Dim comparer As IEqualityComparer(Of DataRow) = DataRowComparer.Default
Dim bEqual = comparer.Equals(left, right)

If (bEqual = True) Then
    Console.WriteLine("Two rows are equal")
Else
    Console.WriteLine("Two rows are not equal")
End If

' Output the hash codes of the two rows.
Console.WriteLine("The hashcodes for the two rows are {0}, {1}", _
    comparer.GetHashCode(left), _
    comparer.GetHashCode(right))
```

See also

- DataRowComparer
- Loading Data Into a DataSet
- LINQ to DataSet Examples

Creating a DataTable From a Query (LINQ to DataSet)

3/20/2020 • 10 minutes to read • Edit Online

Data binding is a common use of DataTable object. The CopyToDataTable method takes the results of a query and copies the data into a DataTable, which can then be used for data binding. When the data operations have been performed, the new DataTable is merged back into the source DataTable.

The CopyToDataTable method uses the following process to create a DataTable from a query:

1. The CopyToDataTable method clones a DataTable from the source table (a DataTable object that implements the IQueryable<T> interface). The IEnumerable source has generally originated from a LINQ to DataSet expression or method query.
2. The schema of the cloned DataTable is built from the columns of the first enumerated DataRow object in the source table and the name of the cloned table is the name of the source table with the word "query" appended to it.
3. For each row in the source table, the content of the row is copied into a new DataRow object, which is then inserted into the cloned table. The RowState and RowError properties are preserved across the copy operation. An ArgumentException is thrown if the DataRow objects in the source are from different tables.
4. The cloned DataTable is returned after all DataRow objects in the input queryable table have been copied. If the source sequence does not contain any DataRow objects, the method returns an empty DataTable.

Calling the CopyToDataTable method causes the query bound to the source table to execute.

When the CopyToDataTable method encounters either a null reference or nullable value type in a row in the source table, it replaces the value with Value. This way, null values are handled correctly in the returned DataTable.

Note: The CopyToDataTable method accepts as input a query that can return rows from multiple DataTable or DataSet objects. The CopyToDataTable method will copy the data but not the properties from the source DataTable or DataSet objects to the returned DataTable. You will need to explicitly set the properties on the returned DataTable, such as Locale and TableName.

The following example queries the SalesOrderHeader table for orders after August 8, 2001 and uses the CopyToDataTable method to create a DataTable from that query. The DataTable is then bound to a BindingSource, which acts as proxy for a DataGridView.

```
// Bind the System.Windows.Forms.DataGridView object
// to the System.Windows.Forms.BindingSource object.
dataGridView.DataSource = bindingSource;

// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable orders = ds.Tables["SalesOrderHeader"];

// Query the SalesOrderHeader table for orders placed
// after August 8, 2001.
IEnumerable<DataRow> query =
    from order in orders.AsEnumerable()
    where order.Field<DateTime>("OrderDate") > new DateTime(2001, 8, 1)
    select order;

// Create a table from the query.
DataTable boundTable = query.CopyToDataTable<DataRow>();

// Bind the table to a System.Windows.Forms.BindingSource object,
// which acts as a proxy for a System.Windows.Forms.DataGridView object.
bindingSource.DataSource = boundTable;
```

```
' Bind the System.Windows.Forms.DataGridView object
' to the System.Windows.Forms.BindingSource object.
dataGridView.DataSource = bindingSource

' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim orders As DataTable = ds.Tables("SalesOrderHeader")

' Query the SalesOrderHeader table for orders placed
'  after August 8, 2001.
Dim query = _
    From order In orders.AsEnumerable() _
    Where order.Field(Of DateTime)("OrderDate") > New DateTime(2001, 8, 1) _
    Select order

' Create a table from the query.
Dim boundTable As DataTable = query.CopyToDataTable()

' Bind the table to a System.Windows.Forms.BindingSource object,
' which acts as a proxy for a System.Windows.Forms.DataGridView object.
bindingSource.DataSource = boundTable
```

Creating a Custom CopyToDataTable<T> Method

The existing CopyToDataTable methods only operate on an IEnumerable<T> source where the generic parameter `T` is of type DataRow. Although this is useful, it does not allow tables to be created from a sequence of scalar types, from queries that return anonymous types, or from queries that perform table joins. For an example of how to implement two custom `CopyToDataTable` methods that load a table from a sequence of scalar or anonymous types, see How to: Implement CopyToDataTable<T> Where the Generic Type T Is Not a DataRows.

The examples in this section use the following custom types:

```
public class Item
{
    public int Id { get; set; }
    public double Price { get; set; }
    public string Genre { get; set; }
}

public class Book : Item
{
    public string Author { get; set; }
}

public class Movie : Item
{
    public string Director { get; set; }
}
```

```
Public Class Item
    Private _Id As Int32
    Private _Price As Double
    Private _Genre As String

    Public Property Id() As Int32
        Get
            Return Id
        End Get
        Set(ByVal value As Int32)
            _Id = value
        End Set
    End Property

    Public Property Price() As Double
        Get
            Return _Price
        End Get
        Set(ByVal value As Double)
            _Price = value
        End Set
    End Property

    Public Property Genre() As String
        Get
            Return _Genre
        End Get
        Set(ByVal value As String)
            _Genre = value
        End Set
    End Property

End Class
Public Class Book
    Inherits Item
    Private _Author As String
    Public Property Author() As String
        Get
            Return _Author
        End Get
        Set(ByVal value As String)
            _Author = value
        End Set
    End Property
End Class

Public Class Movie
    Inherits Item
    Private _Director As String
    Public Property Director() As String
        Get
            Return _Director
        End Get
        Set(ByVal value As String)
            _Director = value
        End Set
    End Property

End Class
```

Example

This example performs a join over the `SalesOrderHeader` and `SalesOrderDetail` tables to get online orders from the month of August and creates a table from the query.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable orders = ds.Tables["SalesOrderHeader"];
DataTable details = ds.Tables["SalesOrderDetail"];

var query =
    from order in orders.AsEnumerable()
    join detail in details.AsEnumerable()
    on order.Field<int>("SalesOrderID") equals
        detail.Field<int>("SalesOrderID")
    where order.Field<bool>("OnlineOrderFlag") == true
    && order.Field<DateTime>("OrderDate").Month == 8
    select new
    {
        SalesOrderID =
            order.Field<int>("SalesOrderID"),
        SalesOrderDetailID =
            detail.Field<int>("SalesOrderDetailID"),
        OrderDate =
            order.Field<DateTime>("OrderDate"),
        ProductID =
            detail.Field<int>("ProductID")
    };

DataTable orderTable = query.CopyToDataTable();
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim orders As DataTable = ds.Tables("SalesOrderHeader")
Dim details As DataTable = ds.Tables("SalesOrderDetail")


Dim query = _
    From order In orders.AsEnumerable() _
    Join detail In details.AsEnumerable() _
    On order.Field(Of Integer)("SalesOrderID") Equals _
            detail.Field(Of Integer)("SalesOrderID") _
    Where order.Field(Of Boolean)("OnlineOrderFlag") = True And _
            order.Field(Of DateTime)("OrderDate").Month = 8 _
    Select New With _
    { _
        .SalesOrderID = order.Field(Of Integer)("SalesOrderID"), _
        .SalesOrderDetailID = detail.Field(Of Integer)("SalesOrderDetailID"), _
        .OrderDate = order.Field(Of DateTime)("OrderDate"), _
        .ProductID = detail.Field(Of Integer)("ProductID") _
    }

Dim table As DataTable = query.CopyToDataTable()
```

Example

The following example queries a collection for items of price greater than $9.99 and creates a table from the query results.

```
// Create a sequence.
Item[] items = new Item[]
{ new Book{Id = 1, Price = 13.50, Genre = "Comedy", Author = "Gustavo Achong"},
  new Book{Id = 2, Price = 8.50, Genre = "Drama", Author = "Jessie Zeng"},
  new Movie{Id = 1, Price = 22.99, Genre = "Comedy", Director = "Marissa Barnes"},
  new Movie{Id = 1, Price = 13.40, Genre = "Action", Director = "Emmanuel Fernandez"}};

// Query for items with price greater than 9.99.
var query = from i in items
             where i.Price > 9.99
             orderby i.Price
             select i;

// Load the query results into new DataTable.
DataTable table = query.CopyToDataTable();
```

```
Dim book1 As New Book()
book1.Id = 1
book1.Price = 13.5
book1.Genre = "Comedy"
book1.Author = "Gustavo Achong"

Dim book2 As New Book
book2.Id = 2
book2.Price = 8.5
book2.Genre = "Drama"
book2.Author = "Jessie Zeng"

Dim movie1 As New Movie
movie1.Id = 1
movie1.Price = 22.99
movie1.Genre = "Comedy"
movie1.Director = "Marissa Barnes"

Dim movie2 As New Movie
movie2.Id = 1
movie2.Price = 13.4
movie2.Genre = "Action"
movie2.Director = "Emmanuel Fernandez"

Dim items(3) As Item
items(0) = book1
items(1) = book2
items(2) = movie1
items(3) = movie2

' Query for items with price greater than 9.99.
Dim query = From i In items _
            Where i.Price > 9.99 _
            Order By i.Price _
            Select New With {i.Price, i.Genre}

Dim table As DataTable
table = query.CopyToDataTable()
```

Example

The following example queries a collection for items of price greater than 9.99 and projects the results. The returned sequence of anonymous types is loaded into an existing table.

```
// Create a sequence.
Item[] items = new Item[]
{ new Book{Id = 1, Price = 13.50, Genre = "Comedy", Author = "Gustavo Achong"},
  new Book{Id = 2, Price = 8.50, Genre = "Drama", Author = "Jessie Zeng"},
  new Movie{Id = 1, Price = 22.99, Genre = "Comedy", Director = "Marissa Barnes"},
  new Movie{Id = 1, Price = 13.40, Genre = "Action", Director = "Emmanuel Fernandez"}};

// Create a table with a schema that matches that of the query results.
DataTable table = new DataTable();
table.Columns.Add("Price", typeof(int));
table.Columns.Add("Genre", typeof(string));

var query = from i in items
             where i.Price > 9.99
             orderby i.Price
             select new { i.Price, i.Genre };

query.CopyToDataTable(table, LoadOption.PreserveChanges);
```

```
Dim book1 As New Book()
book1.Id = 1
book1.Price = 13.5
book1.Genre = "Comedy"
book1.Author = "Gustavo Achong"

Dim book2 As New Book
book2.Id = 2
book2.Price = 8.5
book2.Genre = "Drama"
book2.Author = "Jessie Zeng"

Dim movie1 As New Movie
movie1.Id = 1
movie1.Price = 22.99
movie1.Genre = "Comedy"
movie1.Director = "Marissa Barnes"

Dim movie2 As New Movie
movie2.Id = 1
movie2.Price = 13.4
movie2.Genre = "Action"
movie2.Director = "Emmanuel Fernandez"

Dim items(3) As Item
items(0) = book1
items(1) = book2
items(2) = movie1
items(3) = movie2

' Create a table with a schema that matches that of the query results.
Dim table As DataTable = New DataTable()
table.Columns.Add("Price", GetType(Integer))
table.Columns.Add("Genre", GetType(String))

' Query for items with price greater than 9.99.
Dim query = From i In items _
            Where i.Price > 9.99 _
            Order By i.Price _
            Select New With {i.Price, i.Genre}

query.CopyToDataTable(table, LoadOption.PreserveChanges)
```

Example

The following example queries a collection for items of price greater than $9.99 and projects the results. The

returned sequence of anonymous types is loaded into an existing table. The table schema is automatically expanded because the `Book` and `Movies` types are derived from the `Item` type.

```
// Create a sequence.
Item[] items = new Item[]
{ new Book{Id = 1, Price = 13.50, Genre = "Comedy", Author = "Gustavo Achong"},
  new Book{Id = 2, Price = 8.50, Genre = "Drama", Author = "Jessie Zeng"},
  new Movie{Id = 1, Price = 22.99, Genre = "Comedy", Director = "Marissa Barnes"},
  new Movie{Id = 1, Price = 13.40, Genre = "Action", Director = "Emmanuel Fernandez"}};

// Load into an existing DataTable, expand the schema and
// autogenerate a new Id.
DataTable table = new DataTable();
DataColumn dc = table.Columns.Add("NewId", typeof(int));
dc.AutoIncrement = true;
table.Columns.Add("ExtraColumn", typeof(string));

var query = from i in items
             where i.Price > 9.99
             orderby i.Price
             select new { i.Price, i.Genre };

query.CopyToDataTable(table, LoadOption.PreserveChanges);
```

```
Dim book1 As New Book()
book1.Id = 1
book1.Price = 13.5
book1.Genre = "Comedy"
book1.Author = "Gustavo Achong"

Dim book2 As New Book
book2.Id = 2
book2.Price = 8.5
book2.Genre = "Drama"
book2.Author = "Jessie Zeng"

Dim movie1 As New Movie
movie1.Id = 1
movie1.Price = 22.99
movie1.Genre = "Comedy"
movie1.Director = "Marissa Barnes"

Dim movie2 As New Movie
movie2.Id = 1
movie2.Price = 13.4
movie2.Genre = "Action"
movie2.Director = "Emmanuel Fernandez"

Dim items(3) As Item
items(0) = book1
items(1) = book2
items(2) = movie1
items(3) = movie2

' Load into an existing DataTable, expand the schema and
' autogenerate a new Id.
Dim table As DataTable = New DataTable()
Dim dc As DataColumn = table.Columns.Add("NewId", GetType(Integer))
dc.AutoIncrement = True
table.Columns.Add("ExtraColumn", GetType(String))

Dim query = From i In items _
            Where i.Price > 9.99 _
            Order By i.Price _
            Select New With {i.Price, i.Genre}

query.CopyToDataTable(table, LoadOption.PreserveChanges)
```

Example

The following example queries a collection for items of price greater than $9.99 and returns a sequence of Double, which is loaded into a new table.

```
// Create a sequence.
Item[] items = new Item[]
{ new Book{Id = 1, Price = 13.50, Genre = "Comedy", Author = "Gustavo Achong"},
  new Book{Id = 2, Price = 8.50, Genre = "Drama", Author = "Jessie Zeng"},
  new Movie{Id = 1, Price = 22.99, Genre = "Comedy", Director = "Marissa Barnes"},
  new Movie{Id = 1, Price = 13.40, Genre = "Action", Director = "Emmanuel Fernandez"}};

// load sequence of scalars.
IEnumerable<double> query = from i in items
             where i.Price > 9.99
             orderby i.Price
             select i.Price;

DataTable table = query.CopyToDataTable();
```

```
Dim book1 As New Book()
book1.Id = 1
book1.Price = 13.5
book1.Genre = "Comedy"
book1.Author = "Gustavo Achong"

Dim book2 As New Book
book2.Id = 2
book2.Price = 8.5
book2.Genre = "Drama"
book2.Author = "Jessie Zeng"

Dim movie1 As New Movie
movie1.Id = 1
movie1.Price = 22.99
movie1.Genre = "Comedy"
movie1.Director = "Marissa Barnes"

Dim movie2 As New Movie
movie2.Id = 1
movie2.Price = 13.4
movie2.Genre = "Action"
movie2.Director = "Emmanuel Fernandez"

Dim items(3) As Item
items(0) = book1
items(1) = book2
items(2) = movie1
items(3) = movie2

Dim query = From i In items _
            Where i.Price > 9.99 _
            Order By i.Price _
            Select i.Price

Dim table As DataTable
table = query.CopyToDataTable()
```

See also

- Programming Guide
- Generic Field and SetField Methods
- LINQ to DataSet Examples

How to: Implement CopyToDataTable<T> Where the Generic Type T Is Not a DataRow

3/25/2020 • 8 minutes to read • Edit Online

The DataTable object is often used for data binding. The CopyToDataTable method takes the results of a query and copies the data into a DataTable, which can then be used for data binding. The CopyToDataTable methods, however, only operate on an IEnumerable<T> source where the generic parameter `T` is of type DataRow. Although this is useful, it does not allow tables to be created from a sequence of scalar types, from queries that project anonymous types, or from queries that perform table joins.

This topic describes how to implement two custom `CopyToDataTable<T>` extension methods that accept a generic parameter `T` of a type other than DataRow. The logic to create a DataTable from an IEnumerable<T> source is contained in the `ObjectShredder<T>` class, which is then wrapped in two overloaded `CopyToDataTable<T>` extension methods. The `Shred` method of the `ObjectShredder<T>` class returns the filled DataTable and accepts three input parameters: an IEnumerable<T> source, a DataTable, and a LoadOption enumeration. The initial schema of the returned DataTable is based on the schema of the type `T`. If an existing table is provided as input, the schema must be consistent with the schema of the type `T`. Each public property and field of the type `T` is converted to a DataColumn in the returned table. If the source sequence contains a type derived from `T`, the returned table schema is expanded for any additional public properties or fields.

For examples of using the custom `CopyToDataTable<T>` methods, see Creating a DataTable From a Query.

To implement the custom CopyToDataTable<T> methods in your application

1. Implement the `ObjectShredder<T>` class to create a DataTable from an IEnumerable<T> source:

```
public class ObjectShredder<T>
{
    private System.Reflection.FieldInfo[] _fi;
    private System.Reflection.PropertyInfo[] _pi;
    private System.Collections.Generic.Dictionary<string, int> _ordinalMap;
    private System.Type _type;

    // ObjectShredder constructor.
    public ObjectShredder()
    {
        _type = typeof(T);
        _fi = _type.GetFields();
        _pi = _type.GetProperties();
        _ordinalMap = new Dictionary<string, int>();
    }

    /// <summary>
    /// Loads a DataTable from a sequence of objects.
    /// </summary>
    /// <param name="source">The sequence of objects to load into the DataTable.</param>
    /// <param name="table">The input table. The schema of the table must match that
    /// the type T.  If the table is null, a new table is created with a schema
    /// created from the public properties and fields of the type T.</param>
    /// <param name="options">Specifies how values from the source sequence will be applied to
    /// existing rows in the table.</param>
    /// <returns>A DataTable created from the source sequence.</returns>
    public DataTable Shred(IEnumerable<T> source, DataTable table, LoadOption? options)
    {
        // Load the table from the scalar sequence if T is a primitive type.
        if (typeof(T).IsPrimitive)
        {
```

```
        {
            return ShredPrimitive(source, table, options);
        }

        // Create a new table if the input table is null.
        table ??= new DataTable(typeof(T).Name);

        // Initialize the ordinal map and extend the table schema based on type T.
        table = ExtendTable(table, typeof(T));

        // Enumerate the source sequence and load the object values into rows.
        table.BeginLoadData();
        using (IEnumerator<T> e = source.GetEnumerator())
        {
            while (e.MoveNext())
            {
                if (options != null)
                {
                    table.LoadDataRow(ShredObject(table, e.Current), (LoadOption)options);
                }
                else
                {
                    table.LoadDataRow(ShredObject(table, e.Current), true);
                }
            }
        }
        table.EndLoadData();

        // Return the table.
        return table;
    }

    public DataTable ShredPrimitive(IEnumerable<T> source, DataTable table, LoadOption? options)
    {
        // Create a new table if the input table is null.
        table ??= new DataTable(typeof(T).Name);

        if (!table.Columns.Contains("Value"))
        {
            table.Columns.Add("Value", typeof(T));
        }

        // Enumerate the source sequence and load the scalar values into rows.
        table.BeginLoadData();
        using (IEnumerator<T> e = source.GetEnumerator())
        {
            Object[] values = new object[table.Columns.Count];
            while (e.MoveNext())
            {
                values[table.Columns["Value"].Ordinal] = e.Current;

                if (options != null)
                {
                    table.LoadDataRow(values, (LoadOption)options);
                }
                else
                {
                    table.LoadDataRow(values, true);
                }
            }
        }
        table.EndLoadData();

        // Return the table.
        return table;
    }

    public object[] ShredObject(DataTable table, T instance)
    {
```

```
        FieldInfo[] fi = _fi;
        PropertyInfo[] pi = _pi;

        if (instance.GetType() != typeof(T))
        {
            // If the instance is derived from T, extend the table schema
            // and get the properties and fields.
            ExtendTable(table, instance.GetType());
            fi = instance.GetType().GetFields();
            pi = instance.GetType().GetProperties();
        }

        // Add the property and field values of the instance to an array.
        Object[] values = new object[table.Columns.Count];
        foreach (FieldInfo f in fi)
        {
            values[_ordinalMap[f.Name]] = f.GetValue(instance);
        }

        foreach (PropertyInfo p in pi)
        {
            values[_ordinalMap[p.Name]] = p.GetValue(instance, null);
        }

        // Return the property and field values of the instance.
        return values;
    }

    public DataTable ExtendTable(DataTable table, Type type)
    {
        // Extend the table schema if the input table was null or if the value
        // in the sequence is derived from type T.
        foreach (FieldInfo f in type.GetFields())
        {
            if (!_ordinalMap.ContainsKey(f.Name))
            {
                // Add the field as a column in the table if it doesn't exist
                // already.
                DataColumn dc = table.Columns.Contains(f.Name) ? table.Columns[f.Name]
                    : table.Columns.Add(f.Name, f.FieldType);

                // Add the field to the ordinal map.
                _ordinalMap.Add(f.Name, dc.Ordinal);
            }
        }
        foreach (PropertyInfo p in type.GetProperties())
        {
            if (!_ordinalMap.ContainsKey(p.Name))
            {
                // Add the property as a column in the table if it doesn't exist
                // already.
                DataColumn dc = table.Columns.Contains(p.Name) ? table.Columns[p.Name]
                    : table.Columns.Add(p.Name, p.PropertyType);

                // Add the property to the ordinal map.
                _ordinalMap.Add(p.Name, dc.Ordinal);
            }
        }

        // Return the table.
        return table;
    }
}
```

```
Public Class ObjectShredder(Of T)
    ' Fields
    Private _fi As FieldInfo()
```

```
    Private _ordinalMap As Dictionary(Of String, Integer)
    Private _pi As PropertyInfo()
    Private _type As Type

    ' Constructor
    Public Sub New()
        Me._type = GetType(T)
        Me._fi = Me._type.GetFields
        Me._pi = Me._type.GetProperties
        Me._ordinalMap = New Dictionary(Of String, Integer)
    End Sub

    Public Function ShredObject(ByVal table As DataTable, ByVal instance As T) As Object()
        Dim fi As FieldInfo() = Me._fi
        Dim pi As PropertyInfo() = Me._pi
        If (Not instance.GetType Is GetType(T)) Then
            ' If the instance is derived from T, extend the table schema
            ' and get the properties and fields.
            Me.ExtendTable(table, instance.GetType)
            fi = instance.GetType.GetFields
            pi = instance.GetType.GetProperties
        End If

        ' Add the property and field values of the instance to an array.
        Dim values As Object() = New Object(table.Columns.Count - 1) {}
        Dim f As FieldInfo
        For Each f In fi
            values(Me._ordinalMap.Item(f.Name)) = f.GetValue(instance)
        Next
        Dim p As PropertyInfo
        For Each p In pi
            values(Me._ordinalMap.Item(p.Name)) = p.GetValue(instance, Nothing)
        Next

        ' Return the property and field values of the instance.
        Return values
    End Function


    ' Summary:           Loads a DataTable from a sequence of objects.
    ' source parameter:  The sequence of objects to load into the DataTable.</param>
    ' table parameter:   The input table. The schema of the table must match that
    '                    the type T.  If the table is null, a new table is created
    '                    with a schema created from the public properties and fields
    '                    of the type T.
    ' options parameter: Specifies how values from the source sequence will be applied to
    '                    existing rows in the table.
    ' Returns:           A DataTable created from the source sequence.

    Public Function Shred(ByVal source As IEnumerable(Of T), ByVal table As DataTable, ByVal options As
LoadOption?) As DataTable

        ' Load the table from the scalar sequence if T is a primitive type.
        If GetType(T).IsPrimitive Then
            Return Me.ShredPrimitive(source, table, options)
        End If

        ' Create a new table if the input table is null.
        If (table Is Nothing) Then
            table = New DataTable(GetType(T).Name)
        End If

        ' Initialize the ordinal map and extend the table schema based on type T.
        table = Me.ExtendTable(table, GetType(T))

        ' Enumerate the source sequence and load the object values into rows.
        table.BeginLoadData()
        Using e As IEnumerator(Of T) = source.GetEnumerator
            Do While e.MoveNext
```

```
            Do While e.MoveNext
                If options.HasValue Then
                    table.LoadDataRow(Me.ShredObject(table, e.Current), options.Value)
                Else
                    table.LoadDataRow(Me.ShredObject(table, e.Current), True)
                End If
            Loop
        End Using
        table.EndLoadData()

        ' Return the table.
        Return table
    End Function


    Public Function ShredPrimitive(ByVal source As IEnumerable(Of T), ByVal table As DataTable, ByVal
options As LoadOption?) As DataTable
        ' Create a new table if the input table is null.
        If (table Is Nothing) Then
            table = New DataTable(GetType(T).Name)
        End If
        If Not table.Columns.Contains("Value") Then
            table.Columns.Add("Value", GetType(T))
        End If

        ' Enumerate the source sequence and load the scalar values into rows.
        table.BeginLoadData()
        Using e As IEnumerator(Of T) = source.GetEnumerator
            Dim values As Object() = New Object(table.Columns.Count - 1) {}
            Do While e.MoveNext
                values(table.Columns.Item("Value").Ordinal) = e.Current
                If options.HasValue Then
                    table.LoadDataRow(values, options.Value)
                Else
                    table.LoadDataRow(values, True)
                End If
            Loop
        End Using
        table.EndLoadData()

        ' Return the table.
        Return table
    End Function

    Public Function ExtendTable(ByVal table As DataTable, ByVal type As Type) As DataTable
        ' Extend the table schema if the input table was null or if the value
        ' in the sequence is derived from type T.
        Dim f As FieldInfo
        Dim p As PropertyInfo

        For Each f In type.GetFields
            If Not Me._ordinalMap.ContainsKey(f.Name) Then
                Dim dc As DataColumn

                ' Add the field as a column in the table if it doesn't exist
                ' already.
                dc = IIf(table.Columns.Contains(f.Name), table.Columns.Item(f.Name),
table.Columns.Add(f.Name, f.FieldType))

                ' Add the field to the ordinal map.
                Me._ordinalMap.Add(f.Name, dc.Ordinal)
            End If

        Next

        For Each p In type.GetProperties
            If Not Me._ordinalMap.ContainsKey(p.Name) Then
                ' Add the property as a column in the table if it doesn't exist
                ' already.
                Dim dc As DataColumn
```

```
                Dim dc As DataColumn
                dc = IIf(table.Columns.Contains(p.Name), table.Columns.Item(p.Name),
table.Columns.Add(p.Name, p.PropertyType))

                ' Add the property to the ordinal map.
                Me._ordinalMap.Add(p.Name, dc.Ordinal)
            End If
        Next

        ' Return the table.
        Return table
    End Function

End Class
```

The preceding example assumes that the properties of the `DataColumn` are not nullable value types. To handle properties with nullable value types, use the following code:

```
DataColumn dc = table.Columns.Contains(p.Name) ? table.Columns[p.Name] : table.Columns.Add(p.Name,
Nullable.GetUnderlyingType(p.PropertyType) ?? p.PropertyType);
```

2. Implement the custom `CopyToDataTable<T>` extension methods in a class:

```
public static class CustomLINQtoDataSetMethods
{
    public static DataTable CopyToDataTable<T>(this IEnumerable<T> source)
    {
        return new ObjectShredder<T>().Shred(source, null, null);
    }

    public static DataTable CopyToDataTable<T>(this IEnumerable<T> source,
                                               DataTable table, LoadOption? options)
    {
        return new ObjectShredder<T>().Shred(source, table, options);
    }
}
```

```
Public Module CustomLINQtoDataSetMethods
    <Extension()> _
    Public Function CopyToDataTable(Of T)(ByVal source As IEnumerable(Of T)) As DataTable
        Return New ObjectShredder(Of T)().Shred(source, Nothing, Nothing)
    End Function

    <Extension()> _
    Public Function CopyToDataTable(Of T)(ByVal source As IEnumerable(Of T), ByVal table As DataTable,
ByVal options As LoadOption?) As DataTable
        Return New ObjectShredder(Of T)().Shred(source, table, options)
    End Function

End Module
```

3. Add the `ObjectShredder<T>` class and `CopyToDataTable<T>` extension methods to your application.

```
Module Module1
    Sub Main()
        ' Your application code using CopyToDataTable<T>.
    End Sub
End Module

Public Module CustomLINQtoDataSetMethods
…
End Module

Public Class ObjectShredder(Of T)
…
End Class
```

```
class Program
{
    static void Main(string[] args)
    {
        // Your application code using CopyToDataTable<T>.
    }
}
public static class CustomLINQtoDataSetMethods
{
…
}
public class ObjectShredder<T>
{
…
}
```

See also

- Creating a DataTable From a Query
- Programming Guide

Generic Field and SetField Methods (LINQ to DataSet)

3/25/2020 • 3 minutes to read • Edit Online

LINQ to DataSet provides extension methods to the DataRow class for accessing column values: the Field method and the SetField method. These methods provide easier access to column values for developers, especially regarding null values. The DataSet uses DBNull.Value to represent null values, whereas LINQ uses the Nullable and Nullable<T> types. Using the pre-existing column accessor in DataRow requires you to cast the return object to the appropriate type. If a particular field in a DataRow can be null, you must explicitly check for a null value because returning DBNull.Value and implicitly casting it to another type throws an InvalidCastException. In the following example, if the DataRow.IsNull method was not used to check for a null value, an exception would be thrown if the indexer returned DBNull.Value and tried to cast it to a String.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable products = ds.Tables["Product"];

var query =
    from product in products.AsEnumerable()
    where !product.IsNull("Color") &&
        (string)product["Color"] == "Red"
    select new
    {
        Name = product["Name"],
        ProductNumber = product["ProductNumber"],
        ListPrice = product["ListPrice"]
    };

foreach (var product in query)
{
    Console.WriteLine("Name: {0}", product.Name);
    Console.WriteLine("Product number: {0}", product.ProductNumber);
    Console.WriteLine("List price: ${0}", product.ListPrice);
    Console.WriteLine("");
}
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim products As DataTable = ds.Tables("Product")

Dim query = _
    From product In products.AsEnumerable() _
    Where product!Color IsNot DBNull.Value AndAlso product!Color = "Red" _
    Select New With _
       { _
           .Name = product!Name, _
           .ProductNumber = product!ProductNumber, _
           .ListPrice = product!ListPrice _
       }

For Each product In query
    Console.WriteLine("Name: " & product.Name)
    Console.WriteLine("Product number: " & product.ProductNumber)
    Console.WriteLine("List price: $" & product.ListPrice & vbNewLine)
Next
```

The Field method provides access to the column values of a DataRow and the SetField sets column values in a DataRow. Both the Field method and SetField method handle nullable value types, so you do not have to explicitly check for null values as in the previous example. Both methods are generic methods, also, so you do not have to cast the return type.

The following example uses the Field method.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable products = ds.Tables["Product"];

var query =
    from product in products.AsEnumerable()
    where product.Field<string>("Color") == "Red"
    select new
    {
        Name = product.Field<string>("Name"),
        ProductNumber = product.Field<string>("ProductNumber"),
        ListPrice = product.Field<Decimal>("ListPrice")
    };

foreach (var product in query)
{
    Console.WriteLine("Name: {0}", product.Name);
    Console.WriteLine("Product number: {0}", product.ProductNumber);
    Console.WriteLine("List price: ${0}", product.ListPrice);
    Console.WriteLine("");
}
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim products As DataTable = ds.Tables("Product")

Dim query = _
    From product In products.AsEnumerable() _
    Where product.Field(Of String)("Color") = "Red" _
    Select New With _
       { _
           .Name = product.Field(Of String)("Name"), _
           .ProductNumber = product.Field(Of String)("ProductNumber"), _
           .ListPrice = product.Field(Of Decimal)("ListPrice") _
       }

For Each product In query
    Console.WriteLine("Name: " & product.Name)
    Console.WriteLine("Product number: " & product.ProductNumber)
    Console.WriteLine("List price: $ " & product.ListPrice & vbNewLine)
Next
```

Note that the data type specified in the generic parameter `T` of the Field method and the SetField method must match the type of the underlying value. Otherwise, an InvalidCastException exception will be thrown. The specified column name must also match the name of a column in the DataSet, or an ArgumentException will be thrown. In both cases, the exception is thrown at run time during the enumeration of the data when the query is executed.

The SetField method itself does not perform any type conversions. This does not mean, however, that a type conversion will not occur. The SetField method exposes the ADO.NET behavior of the DataRow class. A type conversion could be performed by the DataRow object and the converted value would then be saved to the DataRow object.

See also

- DataRowExtensions

Data Binding and LINQ to DataSet

1/3/2020 • 2 minutes to read • Edit Online

Data binding is the process that establishes a connection between the application UI and business logic. If the binding has the correct settings and the data provides the proper notifications, when the data changes its value, the elements that are bound to the data reflect changes automatically. The DataSet is an in- memory representation of data that provides a consistent relational programming model, regardless of the source of the data it contains. The ADO.NET 2.0 DataView enables you to sort and filter the data stored in a DataTable. This functionality is often used in data-binding applications. By using a DataView, you can expose the data in a table with different sort orders, and you can filter the data by row state or based on a filter expression. For more information about the DataView object, see DataViews.

LINQ to DataSet allows developers to create complex, powerful queries over a DataSet by using Language-Integrated Query (LINQ). However, a LINQ to DataSet query returns an enumeration of DataRow objects, which is not easily used in a binding scenario. To make binding easier, you can create a DataView from a LINQ to DataSet query. This DataView uses the filtering and sorting specified in the query, but is better suited for data binding. LINQ to DataSet extends the functionality of the DataView by providing LINQ expression-based filtering and sorting, which allows for much more complex and powerful filtering and sorting operations than string-based filtering and sorting.

Note that the DataView represents the query itself and is not a view on top of the query. The DataView is bound to a UI control, such as a DataGrid or a DataGridView, providing a simple data binding model. A DataView can also be created from a DataTable, providing a default view of that table.

In This Section

Creating a DataView Object
Provides information about creating a DataView.

Filtering with DataView
Describes how to filter with the DataView.

Sorting with DataView
Describes how to sort with the DataView.

Querying the DataRowView Collection in a DataView
Provides information about querying the DataRowView collection exposed by DataView.

DataView Performance
Provides information about DataView and performance.

How to: Bind a DataView Object to a Windows Forms DataGridView Control
Describes how to bind a DataView object to a DataGridView.

See also

- Programming Guide

Creating a DataView Object (LINQ to DataSet)

9/7/2019 • 4 minutes to read • Edit Online

There are two ways to create a DataView in the LINQ to DataSet context. You can create a DataView from a LINQ to DataSet query over a DataTable, or you can create it from a typed or un-typed DataTable. In both cases, you create the DataView by using one of the AsDataView extension methods; DataView is not directly constructible in the LINQ to DataSet context.

After the DataView has been created, you can bind it to a UI control in a Windows forms application or an ASP.NET application, or change the filtering and sorting settings.

DataView constructs an index, which significantly increases the performance of operations that can use the index, such as filtering and sorting. The index for a DataView is built both when the DataView is created and when any of the sorting or filtering information is modified. Creating a DataView and then setting the sorting or filtering information later causes the index to be built at least twice: once when the DataView is created, and again when any of the sort or filter properties are modified.

For more information about filtering and sorting with DataView, see Filtering with DataView and Sorting with DataView.

Creating DataView from a LINQ to DataSet Query

A DataView object can be created from the results of a LINQ to DataSet query, where the results are a projection of DataRow objects. The newly created DataView inherits the filtering and sorting information from the query it is created from.

> **NOTE**
> In most cases, the expressions used for filtering and sorting should not have side effects and must be deterministic. Also, the expressions should not contain any logic that depend on a set number of executions, as the sorting and filtering operations may be executed any number of times.

Creating a DataView from a query that returns anonymous types or queries that perform join operations is not supported.

Only the following query operators are supported in a query used to create DataView:

- Cast
- OrderBy
- OrderByDescending
- Select
- ThenBy
- ThenByDescending
- Where

Note that when a DataView is created from a LINQ to DataSet query the Select method must be the final method called in the query. This is shown in the following example, which creates a DataView of online orders sorted by total due:

```
DataTable orders = dataSet.Tables["SalesOrderHeader"];

EnumerableRowCollection<DataRow> query =
    from order in orders.AsEnumerable()
    where order.Field<bool>("OnlineOrderFlag") == true
    orderby order.Field<decimal>("TotalDue")
    select order;

DataView view = query.AsDataView();

bindingSource1.DataSource = view;
```

```
Dim orders As DataTable = dataSet.Tables("SalesOrderHeader")

Dim query = _
    From order In orders.AsEnumerable() _
    Where order.Field(Of Boolean)("OnlineOrderFlag") = True _
    Order By order.Field(Of Decimal)("TotalDue") _
    Select order

Dim view As DataView = query.AsDataView()
bindingSource1.DataSource = view
```

You can also use the string-based RowFilter and Sort properties to filter and sort a DataView after it has been created from a query. Note that this will clear the sorting and filtering information inherited from the query. The following example creates a DataView from a LINQ to DataSet query that filters by last names that start with 'S'. The string-based Sort property is set to sort on last names in ascending order and then first names in descending order:

```
DataTable contacts = dataSet.Tables["Contact"];

EnumerableRowCollection<DataRow> query = from contact in contacts.AsEnumerable()
                                         where contact.Field<string>("LastName").StartsWith("S")
                                         select contact;

DataView view = query.AsDataView();

bindingSource1.DataSource = view;

view.Sort = "LastName desc, FirstName asc";
```

```
Dim contacts As DataTable = dataSet.Tables("Contact")

Dim query = _
    From contact In contacts.AsEnumerable() _
    Where contact.Field(Of String)("LastName").StartsWith("S") _
    Select contact

Dim view As DataView = query.AsDataView()
bindingSource1.DataSource = view
view.Sort = "LastName desc, FirstName asc"
```

Creating a DataView from a DataTable

In addition to being created from a LINQ to DataSet query, a DataView object can be created from a DataTable by using the AsDataView method.

The following example creates a DataView from the SalesOrderDetail table and sets it as the data source of a

BindingSource object. This object acts as a proxy for a DataGridView control.

```
DataTable orders = dataSet.Tables["SalesOrderDetail"];

DataView view = orders.AsDataView();
bindingSource1.DataSource = view;

dataGridView1.AutoResizeColumns();
```

```
Dim orders As DataTable = dataSet.Tables("SalesOrderDetail")

Dim view As DataView = orders.AsDataView()
bindingSource1.DataSource = view
dataGridView1.AutoResizeColumns()
```

Filtering and sorting can be set on the DataView after it has been created from a DataTable. The following example creates a DataView from the Contact table and sets the Sort property to sort on last names in ascending order and then first names in descending order:

```
DataTable contacts = dataSet.Tables["Contact"];

DataView view = contacts.AsDataView();

view.Sort = "LastName desc, FirstName asc";

bindingSource1.DataSource = view;
dataGridView1.AutoResizeColumns();
```

```
Dim contacts As DataTable = dataSet.Tables("Contact")

Dim view As DataView = contacts.AsDataView()

view.Sort = "LastName desc, FirstName asc"

bindingSource1.DataSource = view
dataGridView1.AutoResizeColumns()
```

However, there is a performance loss that comes with setting the RowFilter or Sort property after the DataView has been created from a query, because DataView constructs an index to support filtering and sorting operations. Setting the RowFilter or Sort property rebuilds the index for the data, adding overhead to your application and decreasing performance. When possible, it is better to specify the filtering and sorting information when you first create the DataView and avoid modifying it afterwards.

See also

- Data Binding and LINQ to DataSet
- Filtering with DataView
- Sorting with DataView

Filtering with DataView (LINQ to DataSet)

1/3/2020 • 9 minutes to read • Edit Online

The ability to filter data using specific criteria and then present the data to a client through a UI control is an important aspect of data binding. DataView provides several ways to filter data and return subsets of data rows meeting specific filter criteria. In addition to the string-based filtering capabilities, DataView also provides the ability to use LINQ expressions for the filtering criteria. LINQ expressions allow for much more complex and powerful filtering operations than the string-based filtering.

There are two ways to filter data using a DataView:

- Create a DataView from a LINQ to DataSet query with a Where clause.
- Use the existing, string-based filtering capabilities of DataView.

Creating DataView from a Query with Filtering Information

A DataView object can be created from a LINQ to DataSet query. If that query contains a `Where` clause, the DataView is created with the filtering information from the query. The expression in the `Where` clause is used to determine which data rows will be included in the DataView, and is the basis for the filter.

Expression-based filters offer more powerful and complex filtering than the simpler string-based filters. The string-based and expression-based filters are mutually exclusive. When the string-based RowFilter is set after a DataView is created from a query, the expression based filter inferred from the query is cleared.

> **NOTE**
> In most cases, the expressions used for filtering should not have side effects and must be deterministic. Also, the expressions should not contain any logic that depends on a set number of executions, because the filtering operations might be executed any number of times.

Example

The following example queries the SalesOrderDetail table for orders with a quantity greater than 2 and less than 6; creates a DataView from that query; and binds the DataView to a BindingSource:

```
DataTable orders = dataSet.Tables["SalesOrderDetail"];

EnumerableRowCollection<DataRow> query = from order in orders.AsEnumerable()
                                         where order.Field<Int16>("OrderQty") > 2 && order.Field<Int16>
("OrderQty") < 6
                                         select order;

DataView view = query.AsDataView();

bindingSource1.DataSource = view;
```

```
Dim orders As DataTable = dataSet.Tables("SalesOrderDetail")

Dim query = _
    From order In orders.AsEnumerable() _
    Where order.Field(Of Int16)("OrderQty") > 2 And _
          order.Field(Of Int16)("OrderQty") < 6 _
    Select order

Dim view As DataView = query.AsDataView()
bindingSource1.DataSource = view
```

Example

The following example creates a DataView from a query for orders placed after June 6, 2001:

```
DataTable orders = dataSet.Tables["SalesOrderHeader"];

EnumerableRowCollection<DataRow> query = from order in orders.AsEnumerable()
                                         where order.Field<DateTime>("OrderDate") > new DateTime(2002, 6, 1)
                                         select order;

DataView view = query.AsDataView();

bindingSource1.DataSource = view;
```

```
Dim orders As DataTable = dataSet.Tables("SalesOrderHeader")

Dim query = _
    From order In orders.AsEnumerable() _
    Where order.Field(Of DateTime)("OrderDate") > New DateTime(2002, 6, 1) _
    Select order

Dim view As DataView = query.AsDataView()
bindingSource1.DataSource = view
```

Example

Filtering can also be combined with sorting. The following example creates a DataView from a query for contacts whose last name start with "S" and sorted by last name, then first name:

```
DataTable contacts = dataSet.Tables["Contact"];

EnumerableRowCollection<DataRow> query = from contact in contacts.AsEnumerable()
                                         where contact.Field<string>("LastName").StartsWith("S")
                                         orderby contact.Field<string>("LastName"), contact.Field<string>
("FirstName")
                                         select contact;

DataView view = query.AsDataView();

bindingSource1.DataSource = view;
dataGridView1.AutoResizeColumns();
```

```
Dim contacts As DataTable = dataSet.Tables("Contact")

Dim query = _
    From contact In contacts.AsEnumerable() _
    Where contact.Field(Of String)("LastName").StartsWith("S") _
    Order By contact.Field(Of String)("LastName"), contact.Field(Of String)("FirstName") _
    Select contact

Dim view As DataView = query.AsDataView()
bindingSource1.DataSource = view
dataGridView1.AutoResizeColumns()
```

Example

The following example uses the SoundEx algorithm to find contacts whose last name is similar to "Zhu". The SoundEx algorithm is implemented in the SoundEx method.

```
DataTable contacts = dataSet.Tables["Contact"];

string soundExCode = SoundEx("Zhu");

EnumerableRowCollection<DataRow> query = from contact in contacts.AsEnumerable()
                                         where SoundEx(contact.Field<string>("LastName")) == soundExCode
                                         select contact;

DataView view = query.AsDataView();

bindingSource1.DataSource = view;
dataGridView1.AutoResizeColumns();
```

```
Dim contacts As DataTable = dataSet.Tables("Contact")
Dim soundExCode As String = SoundEx("Zhu")

Dim query = _
    From contact In contacts.AsEnumerable() _
    Where SoundEx(contact.Field(Of String)("LastName")) = soundExCode _
    Select contact

Dim view As DataView = query.AsDataView()
bindingSource1.DataSource = view
dataGridView1.AutoResizeColumns()
```

SoundEx is a phonetic algorithm used for indexing names by sound, as they are pronounced in English, originally developed by the U.S. Census Bureau. The SoundEx method returns a four character code for a name consisting of an English letter followed by three numbers. The letter is the first letter of the name and the numbers encode the remaining consonants in the name. Similar sounding names share the same SoundEx code. The SoundEx implementation used in the SoundEx method of the previous example is shown here:

```
static private string SoundEx(string word)
{
    // The length of the returned code.
    int length = 4;

    // Value to return.
    string value = "";

    // The size of the word to process.
    int size = word.Length;

    // The word must be at least two characters in length.
    if (size > 1)
```

```
    {
        // Convert the word to uppercase characters.
        word = word.ToUpper(System.Globalization.CultureInfo.InvariantCulture);

        // Convert the word to a character array.
        char[] chars = word.ToCharArray();

        // Buffer to hold the character codes.
        StringBuilder buffer = new StringBuilder();
        buffer.Length = 0;

        // The current and previous character codes.
        int prevCode = 0;
        int currCode = 0;

        // Add the first character to the buffer.
        buffer.Append(chars[0]);

        // Loop through all the characters and convert them to the proper character code.
        for (int i = 1; i < size; i++)
        {
            switch (chars[i])
            {
                case 'A':
                case 'E':
                case 'I':
                case 'O':
                case 'U':
                case 'H':
                case 'W':
                case 'Y':
                    currCode = 0;
                    break;
                case 'B':
                case 'F':
                case 'P':
                case 'V':
                    currCode = 1;
                    break;
                case 'C':
                case 'G':
                case 'J':
                case 'K':
                case 'Q':
                case 'S':
                case 'X':
                case 'Z':
                    currCode = 2;
                    break;
                case 'D':
                case 'T':
                    currCode = 3;
                    break;
                case 'L':
                    currCode = 4;
                    break;
                case 'M':
                case 'N':
                    currCode = 5;
                    break;
                case 'R':
                    currCode = 6;
                    break;
            }

            // Check if the current code is the same as the previous code.
            if (currCode != prevCode)
            {
                // Check to see if the current code is 0 (a vowel); do not process vowels.
```

```
                if (currCode != 0)
                    buffer.Append(currCode);
            }
            // Set the previous character code.
            prevCode = currCode;

            // If the buffer size meets the length limit, exit the loop.
            if (buffer.Length == length)
                break;
        }
        // Pad the buffer, if required.
        size = buffer.Length;
        if (size < length)
            buffer.Append('0', (length - size));

        // Set the value to return.
        value = buffer.ToString();
    }
    // Return the value.
    return value;
}
```

```
Private Function SoundEx(ByVal word As String) As String

    Dim length As Integer = 4
    ' Value to return
    Dim value As String = ""
    ' Size of the word to process
    Dim size As Integer = word.Length
    ' Make sure the word is at least two characters in length
    If (size > 1) Then
        ' Convert the word to all uppercase
        word = word.ToUpper(System.Globalization.CultureInfo.InvariantCulture)
        ' Convert the word to character array for faster processing
        Dim chars As Char() = word.ToCharArray()
        ' Buffer to build up with character codes
        Dim buffer As StringBuilder = New StringBuilder()
        ' The current and previous character codes
        Dim prevCode As Integer = 0
        Dim currCode As Integer = 0
        ' Append the first character to the buffer
        buffer.Append(chars(0))
        ' Loop through all the characters and convert them to the proper character code
        For i As Integer = 1 To size - 1
            Select Case chars(i)

                Case "A", "E", "I", "O", "U", "H", "W", "Y"
                    currCode = 0

                Case "B", "F", "P", "V"
                    currCode = 1

                Case "C", "G", "J", "K", "Q", "S", "X", "Z"
                    currCode = 2

                Case "D", "T"
                    currCode = 3

                Case "L"
                    currCode = 4

                Case "M", "N"
                    currCode = 5

                Case "R"
                    currCode = 6
            End Select
```

```
            ' Check to see if the current code is the same as the last one
            If (currCode <> prevCode) Then

                ' Check to see if the current code is 0 (a vowel); do not process vowels
                If (currCode <> 0) Then
                    buffer.Append(currCode)
                End If
            End If
            ' Set the new previous character code
            prevCode = currCode
            ' If the buffer size meets the length limit, then exit the loop
            If (buffer.Length = length) Then
                Exit For
            End If
        Next
        ' Pad the buffer, if required
        size = buffer.Length
        If (size < length) Then
            buffer.Append("0", (length - size))
        End If
        ' Set the value to return
        value = buffer.ToString()
    End If
    ' Return the value
    Return value
End Function
```

Using the RowFilter Property

The existing string-based filtering functionality of DataView still works in the LINQ to DataSet context. For more information about string-based RowFilter filtering, see Sorting and Filtering Data.

The following example creates a DataView from the Contact table and then sets the RowFilter property to return rows where the contact's last name is "Zhu":

```
DataTable contacts = dataSet.Tables["Contact"];

DataView view = contacts.AsDataView();

view.RowFilter = "LastName='Zhu'";

bindingSource1.DataSource = view;
dataGridView1.AutoResizeColumns();
```

```
Dim contacts As DataTable = dataSet.Tables("Contact")

Dim view As DataView = contacts.AsDataView()
view.RowFilter = "LastName='Zhu'"
bindingSource1.DataSource = view
dataGridView1.AutoResizeColumns()
```

After a DataView has been created from a DataTable or LINQ to DataSet query, you can use the RowFilter property to specify subsets of rows based on their column values. The string-based and expression-based filters are mutually exclusive. Setting the RowFilter property will clear the filter expression inferred from the LINQ to DataSet query, and the filter expression cannot be reset.

```
DataTable contacts = dataSet.Tables["Contact"];

EnumerableRowCollection<DataRow> query = from contact in contacts.AsEnumerable()
                                         where contact.Field<string>("LastName") == "Hernandez"
                                         select contact;

DataView view = query.AsDataView();

bindingSource1.DataSource = view;
dataGridView1.AutoResizeColumns();

view.RowFilter = "LastName='Zhu'";
```

```
Dim contacts As DataTable = dataSet.Tables("Contact")

Dim query = _
    From contact In contacts.AsEnumerable() _
    Where contact.Field(Of String)("LastName") = "Hernandez" _
    Select contact

Dim view As DataView = query.AsDataView()
bindingSource1.DataSource = view

dataGridView1.AutoResizeColumns()
view.RowFilter = "LastName='Zhu'"
```

If you want to return the results of a particular query on the data, as opposed to providing a dynamic view of a subset of the data, you can use the Find or FindRows methods of the DataView, rather than setting the RowFilter property. The RowFilter property is best used in a data-bound application where a bound control displays filtered results. Setting the RowFilter property rebuilds the index for the data, adding overhead to your application and decreasing performance. The Find and FindRows methods use the current index without requiring the index to be rebuilt. If you are going to call Find or FindRows only once, then you should use the existing DataView. If you are going to call Find or FindRows multiple times, you should create a new DataView to rebuild the index on the column you want to search on, and then call the Find or FindRows methods. For more information about the Find and FindRows methods see Finding Rows and DataView Performance.

Clearing the Filter

The filter on a DataView can be cleared after filtering has been set using the RowFilter property. The filter on a DataView can be cleared in two different ways:

- Set the RowFilter property to `null`.
- Set the RowFilter property to an empty string.

Example

The following example creates a DataView from a query and then clears the filter by setting RowFilter property to `null`:

```
DataTable orders = dataSet.Tables["SalesOrderHeader"];

EnumerableRowCollection<DataRow> query = from order in orders.AsEnumerable()
                                         where order.Field<DateTime>("OrderDate") > new DateTime(2002, 11, 20)
                                            && order.Field<Decimal>("TotalDue") < new Decimal(60.00)
                                         select order;

DataView view = query.AsDataView();

bindingSource1.DataSource = view;

view.RowFilter = null;
```

```
Dim orders As DataTable = dataSet.Tables("SalesOrderHeader")

Dim query = _
    From order In orders.AsEnumerable() _
    Where order.Field(Of DateTime)("OrderDate") > New DateTime(2002, 11, 20) _
        And order.Field(Of Decimal)("TotalDue") < New Decimal(60.0) _
    Select order

Dim view As DataView = query.AsDataView()
bindingSource1.DataSource = view
view.RowFilter = Nothing
```

Example

The following example creates a DataView from a table sets the RowFilter property, and then clears the filter by setting the RowFilter property to an empty string:

```
DataTable contacts = dataSet.Tables["Contact"];

DataView view = contacts.AsDataView();

view.RowFilter = "LastName='Zhu'";

bindingSource1.DataSource = view;
dataGridView1.AutoResizeColumns();

// Clear the row filter.
view.RowFilter = "";
```

```
Dim contacts As DataTable = dataSet.Tables("Contact")

Dim view As DataView = contacts.AsDataView()
view.RowFilter = "LastName='Zhu'"
bindingSource1.DataSource = view
dataGridView1.AutoResizeColumns()

' Clear the row filter.
view.RowFilter = ""
```

See also

- Data Binding and LINQ to DataSet
- Sorting with DataView

Sorting with DataView (LINQ to DataSet)

1/3/2020 • 4 minutes to read • Edit Online

The ability to sort data based on specific criteria and then present the data to a client through a UI control is an important aspect of data binding. DataView provides several ways to sort data and return data rows ordered by specific ordering criteria. In addition to its string-based sorting capabilities, DataView also enables you to use Language-Integrated Query (LINQ) expressions for the sorting criteria. LINQ expressions allow for much more complex and powerful sorting operations than string-based sorting. This topic describes both approaches to sorting using DataView.

Creating DataView from a Query with Sorting Information

A DataView object can be created from a LINQ to DataSet query. If that query contains an OrderBy, OrderByDescending, ThenBy, or ThenByDescending clause the expressions in these clauses are used as the basis for sorting the data in the DataView. For example, if the query contains the `Order By...` and `Then By...` clauses, the resulting DataView would order the data by both columns specified.

Expression-based sorting offers more powerful and complex sorting than the simpler string-based sorting. Note that string-based and expression-based sorting are mutually exclusive. If the string-based Sort is set after a DataView is created from a query, the expression-based filter inferred from the query is cleared and cannot be reset.

The index for a DataView is built both when the DataView is created and when any of the sorting or filtering information is modified. You get the best performance by supplying sorting criteria in the LINQ to DataSet query that the DataView is created from and not modifying the sorting information, later. For more information, see DataView Performance.

> **NOTE**
> In most cases, the expressions used for sorting should not have side effects and must be deterministic. Also, the expressions should not contain any logic that depends on a set number of executions, because the sorting operations might be executed any number of times.

Example

The following example queries the SalesOrderHeader table and orders the returned rows by the order date; creates a DataView from that query; and binds the DataView to a BindingSource.

```
DataTable orders = dataSet.Tables["SalesOrderHeader"];

EnumerableRowCollection<DataRow> query = from order in orders.AsEnumerable()
                                         orderby order.Field<DateTime>("OrderDate")
                                         select order;

DataView view = query.AsDataView();

bindingSource1.DataSource = view;
```

```
Dim orders As DataTable = dataSet.Tables("SalesOrderHeader")

Dim query = _
    From order In orders.AsEnumerable() _
    Order By order.Field(Of DateTime)("OrderDate") _
    Select order

Dim view As DataView = query.AsDataView()
bindingSource1.DataSource = view
```

Example

The following example queries the SalesOrderHeader table and orders the returned row by total amount due; creates a DataView from that query; and binds the DataView to a BindingSource.

```
DataTable orders = dataSet.Tables["SalesOrderHeader"];

EnumerableRowCollection<DataRow> query =
    from order in orders.AsEnumerable()
    orderby order.Field<decimal>("TotalDue")
    select order;

DataView view = query.AsDataView();

bindingSource1.DataSource = view;
```

```
Dim orders As DataTable = dataSet.Tables("SalesOrderHeader")

Dim query = _
    From order In orders.AsEnumerable() _
    Order By order.Field(Of Decimal)("TotalDue") _
    Select order

Dim view As DataView = query.AsDataView()
bindingSource1.DataSource = view
```

Example

The following example queries the SalesOrderDetail table and orders the returned rows by order quantity and then by sales order ID; creates a DataView from that query; and binds the DataView to a BindingSource.

```
DataTable orders = dataSet.Tables["SalesOrderDetail"];

EnumerableRowCollection<DataRow> query = from order in orders.AsEnumerable()
                                         orderby order.Field<Int16>("OrderQty"), order.Field<int>
("SalesOrderID")
                                         select order;

DataView view = query.AsDataView();

bindingSource1.DataSource = view;
```

```
Dim orders As DataTable = dataSet.Tables("SalesOrderDetail")

Dim query = _
    From order In orders.AsEnumerable() _
    Order By order.Field(Of Int16)("OrderQty"), order.Field(Of Integer)("SalesOrderID") _
    Select order

Dim view As DataView = query.AsDataView()
bindingSource1.DataSource = view
```

Using the String-Based Sort Property

The string-based sorting functionality of DataView still works with LINQ to DataSet. After a DataView has been created from a LINQ to DataSet query, you can use the Sort property to set the sorting on the DataView.

The string-based and expression-based sorting functionality are mutually exclusive. Setting the Sort property will clear the expression-based sort inherited from the query that the DataView was created from.

For more information about string-based Sort filtering, see Sorting and Filtering Data.

Example

The follow example creates a DataView from the Contact table and sorts the rows by last name in descending order, then first name in ascending order:

```
DataTable contacts = dataSet.Tables["Contact"];

DataView view = contacts.AsDataView();

view.Sort = "LastName desc, FirstName asc";

bindingSource1.DataSource = view;
dataGridView1.AutoResizeColumns();
```

```
Dim contacts As DataTable = dataSet.Tables("Contact")

Dim view As DataView = contacts.AsDataView()

view.Sort = "LastName desc, FirstName asc"

bindingSource1.DataSource = view
dataGridView1.AutoResizeColumns()
```

Example

The following example queries the Contact table for last names that start with the letter "S". A DataView is created from that query and bound to a BindingSource object.

```
DataTable contacts = dataSet.Tables["Contact"];

EnumerableRowCollection<DataRow> query = from contact in contacts.AsEnumerable()
                                         where contact.Field<string>("LastName").StartsWith("S")
                                         select contact;

DataView view = query.AsDataView();

bindingSource1.DataSource = view;

view.Sort = "LastName desc, FirstName asc";
```

```
Dim contacts As DataTable = dataSet.Tables("Contact")

Dim query = _
    From contact In contacts.AsEnumerable() _
    Where contact.Field(Of String)("LastName").StartsWith("S") _
    Select contact

Dim view As DataView = query.AsDataView()
bindingSource1.DataSource = view
view.Sort = "LastName desc, FirstName asc"
```

Clearing the Sort

The sorting information on a DataView can be cleared after it has been set using the Sort property. There are two ways to clear the sorting information in DataView:

- Set the Sort property to `null`.
- Set the Sort property to an empty string.

Example

The following example creates a DataView from a query and clears the sorting by setting the Sort property to an empty string:

```
DataTable orders = dataSet.Tables["SalesOrderHeader"];

EnumerableRowCollection<DataRow> query = from order in orders.AsEnumerable()
                                         orderby order.Field<decimal>("TotalDue")
                                         select order;

DataView view = query.AsDataView();

bindingSource1.DataSource = view;

view.Sort = "";
```

```
Dim orders As DataTable = dataSet.Tables("SalesOrderHeader")

Dim query = _
    From order In orders.AsEnumerable() _
    Order By order.Field(Of Decimal)("TotalDue") _
    Select order

Dim view As DataView = query.AsDataView()
bindingSource1.DataSource = view
view.Sort = ""
```

Example

The following example creates a DataView from the Contact table and sets the Sort property to sort by last name in descending order. The sorting information is then cleared by setting the Sort property to `null`:

```
DataTable contacts = dataSet.Tables["Contact"];

DataView view = contacts.AsDataView();

view.Sort = "LastName desc";

bindingSource1.DataSource = view;
dataGridView1.AutoResizeColumns();

// Clear the sort.
view.Sort = null;
```

```
Dim contacts As DataTable = dataSet.Tables("Contact")

Dim view As DataView = contacts.AsDataView()
view.Sort = "LastName desc"

bindingSource1.DataSource = view
dataGridView1.AutoResizeColumns()

'Clear the sort.
view.Sort = Nothing
```

See also

- Data Binding and LINQ to DataSet
- Filtering with DataView
- Sorting Data

Querying the DataRowView Collection in a DataView

9/7/2019 • 2 minutes to read • Edit Online

The DataView exposes an enumerable collection of DataRowView objects. DataRowView represents a customized view of a DataRow and displays a specific version of that DataRow in a control. Only one version of a DataRow can be displayed through a control, such as a DataGridView. You can access the DataRow that is exposed by the DataRowView through the Row property of the DataRowView. When you view values by using a DataRowView, the RowStateFilter property determines which row version of the underlying DataRow is exposed. For information about accessing different row versions using a DataRow, see Row States and Row Versions. Because the collection of DataRowView objects exposed by the DataView is enumerable, you can use LINQ to DataSet to query over it.

The following example queries the `Product` table for red-colored products and creates a table from that query. A DataView is created from the table and the RowStateFilter property is set to filter on deleted and modified rows. The DataView is then used as a source in a LINQ query, and the DataRowView objects that have been modified and deleted are bound to a DataGridView control.

```
DataTable products = dataSet.Tables["Product"];

// Query for red colored products.
EnumerableRowCollection<DataRow> redProductsQuery =
    from product in products.AsEnumerable()
    where product.Field<string>("Color") == "Red"
    orderby product.Field<decimal>("ListPrice")
    select product;

// Create a table and view from the query.
DataTable redProducts = redProductsQuery.CopyToDataTable<DataRow>();
DataView view = new DataView(redProducts);

// Mark a row as deleted.
redProducts.Rows[0].Delete();

// Modify product price.
redProducts.Rows[1]["ListPrice"] = 20.00;
redProducts.Rows[2]["ListPrice"] = 30.00;

view.RowStateFilter = DataViewRowState.ModifiedCurrent | DataViewRowState.Deleted;

// Query for the modified and deleted rows.
IEnumerable<DataRowView> modifiedDeletedQuery = from DataRowView rowView in view
                                                select rowView;

dataGridView2.DataSource = modifiedDeletedQuery.ToList();
```

```
Dim products As DataTable = dataSet.Tables("Product")

' Query for red colored products.
Dim redProductsQuery = _
From product In products.AsEnumerable() _
        Where product.Field(Of String)("Color") = "Red" _
        Order By product.Field(Of Decimal)("ListPrice") _
        Select product
' Create a table and view from the query.
Dim redProducts As DataTable = redProductsQuery.CopyToDataTable()
Dim view As DataView = New DataView(redProducts)

' Mark a row as deleted.
redProducts.Rows(0).Delete()

' Modify product price.
redProducts.Rows(1)("ListPrice") = 20.0
redProducts.Rows(2)("ListPrice") = 30.0

view.RowStateFilter = DataViewRowState.ModifiedCurrent Or DataViewRowState.Deleted

' Query for the modified and deleted rows.
Dim modifiedDeletedQuery = From rowView As DataRowView In view _
                           Select rowView

dataGridView2.DataSource = modifiedDeletedQuery.ToList()
```

The following example creates a table of products from a view that is bound to a DataGridView control. The DataView is queried for red-colored products and the ordered results are bound to a DataGridView control.

```
// Create a table from the bound view representing a query of
// available products.
DataView view = (DataView)bindingSource1.DataSource;
DataTable productsTable = (DataTable)view.Table;

// Set RowStateFilter to display the current rows.
view.RowStateFilter = DataViewRowState.CurrentRows ;

// Query the DataView for red colored products ordered by list price.
var productQuery = from DataRowView rowView in view
                   where rowView.Row.Field<string>("Color") == "Red"
                   orderby rowView.Row.Field<decimal>("ListPrice")
                   select new { Name = rowView.Row.Field<string>("Name"),
                                Color = rowView.Row.Field<string>("Color"),
                                Price = rowView.Row.Field<decimal>("ListPrice")};

// Bind the query results to another DataGridView.
dataGridView2.DataSource = productQuery.ToList();
```

```
' Create a table from the bound view representing a query of
' available products.
Dim view As DataView = CType(bindingSource1.DataSource, DataView)
Dim productsTable As DataTable = CType(view.Table, DataTable)

' Set RowStateFilter to display the current rows.
view.RowStateFilter = DataViewRowState.CurrentRows

' Query the DataView for red colored products ordered by list price.
Dim productQuery = From rowView As DataRowView In view _
                   Where rowView.Row.Field(Of String)("Color") = "Red" _
                   Order By rowView.Row.Field(Of Decimal)("ListPrice") _
                   Select New With {.Name = rowView.Row.Field(Of String)("Name"), _
                                    .Color = rowView.Row.Field(Of String)("Color"), _
                                    .Price = rowView.Row.Field(Of Decimal)("ListPrice")}

' Bind the query results to another DataGridView.
dataGridView2.DataSource = productQuery.ToList()
```

See also

- Data Binding and LINQ to DataSet

DataView Performance

3/12/2020 • 3 minutes to read • Edit Online

This topic discusses the performance benefits of using the Find and FindRows methods of the DataView class, and of caching a DataView in a Web application.

Find and FindRows

DataView constructs an index. An index contains keys built from one or more columns in the table or view. These keys are stored in a structure that enables the DataView to find the row or rows associated with the key values quickly and efficiently. Operations that use the index, such as filtering and sorting, see significant performance increases. The index for a DataView is built both when the DataView is created and when any of the sorting or filtering information is modified. Creating a DataView and then setting the sorting or filtering information later causes the index to be built at least twice: once when the DataView is created, and again when any of the sort or filter properties are modified. For more information about filtering and sorting with DataView, see Filtering with DataView and Sorting with DataView.

If you want to return the results of a particular query on the data, as opposed to providing a dynamic view of a subset of the data, you can use the Find or FindRows methods of the DataView, rather than setting the RowFilter property. The RowFilter property is best used in a data-bound application where a bound control displays filtered results. Setting the RowFilter property rebuilds the index for the data, adding overhead to your application and decreasing performance. The Find and FindRows methods use the current index without requiring the index to be rebuilt. If you are going to call Find or FindRows only once, then you should use the existing DataView. If you are going to call Find or FindRows multiple times, you should create a new DataView to rebuild the index on the column you want to search on, and then call the Find or FindRows methods. For more information about the Find and FindRows methods, see Finding Rows.

The following example uses the Find method to find a contact with the last name "Zhu".

```
DataTable contacts = dataSet.Tables["Contact"];

EnumerableRowCollection<DataRow> query = from contact in contacts.AsEnumerable()
                                         orderby contact.Field<string>("LastName")
                                         select contact;

DataView view = query.AsDataView();

// Find a contact with the last name of Zhu.
int found = view.Find("Zhu");
```

```
Dim contacts As DataTable = dataSet.Tables("Contact")

Dim query = _
    From contact In contacts.AsEnumerable() _
    Order By contact.Field(Of String)("LastName") _
    Select contact

Dim view As DataView = query.AsDataView()

Dim found As Integer = view.Find("Zhu")
```

The following example uses the FindRows method to find all the red colored products.

```
DataTable products = dataSet.Tables["Product"];

EnumerableRowCollection<DataRow> query = from product in products.AsEnumerable()
                                         orderby product.Field<Decimal>("ListPrice"), product.Field<string>
("Color")
                                         select product;

DataView view = query.AsDataView();

view.Sort = "Color";

object[] criteria = new object[] { "Red"};

DataRowView[] foundRowsView = view.FindRows(criteria);
```

```
Dim products As DataTable = dataSet.Tables("Product")

Dim query = _
From product In products.AsEnumerable() _
Order By product.Field(Of Decimal)("ListPrice"), product.Field(Of String)("Color") _
Select product

Dim view As DataView = query.AsDataView()
view.Sort = "Color"

Dim criteria As Object() = New Object() {"Red"}

Dim foundRowsView As DataRowView() = view.FindRows(criteria)
```

ASP.NET

ASP.NET has a caching mechanism that allows you to store objects that require extensive server resources to create in memory. Caching these types of resources can significantly improve the performance of your application. Caching is implemented by the Cache class, with cache instances that are private to each application. Because creating a new DataView object can be resource intensive, you might want to use this caching functionality in Web applications so that the DataView does not have to be rebuilt every time the Web page is refreshed.

In the following example, the DataView is cached so that the data does not have to be re-sorted when the page is refreshed.

```
If (Cache("ordersView") = Nothing) Then

Dim dataSet As New DataSet()

   FillDataSet(dataSet)

   Dim orders As DataTable = dataSet.Tables("SalesOrderHeader")

   Dim query = _
                   From order In orders.AsEnumerable() _
                   Where order.Field(Of Boolean)("OnlineOrderFlag") = True _
                   Order By order.Field(Of Decimal)("TotalDue") _
                   Select order

   Dim view As DataView = query.AsDataView()

   Cache.Insert("ordersView", view)

End If

Dim ordersView = CType(Cache("ordersView"), DataView)

GridView1.DataSource = ordersView
GridView1.DataBind()
```

```
if (Cache["ordersView"] == null)
{
   // Fill the DataSet.
   DataSet dataSet = FillDataSet();

   DataTable orders = dataSet.Tables["SalesOrderHeader"];

   EnumerableRowCollection<DataRow> query =
                       from order in orders.AsEnumerable()
                       where order.Field<bool>("OnlineOrderFlag") == true
                       orderby order.Field<decimal>("TotalDue")
                       select order;

   DataView view = query.AsDataView();
   Cache.Insert("ordersView", view);
}

DataView ordersView = (DataView)Cache["ordersView"];

GridView1.DataSource = ordersView;
GridView1.DataBind();
```

See also

- Data Binding and LINQ to DataSet

How to: Bind a DataView Object to a Windows Forms DataGridView Control

9/7/2019 • 2 minutes to read • Edit Online

The DataGridView control provides a powerful and flexible way to display data in a tabular format. The DataGridView control supports the standard Windows Forms data binding model, so it will bind to DataView and a variety of other data sources. In most situations, however, you will bind to a BindingSource component that will manage the details of interacting with the data source.

For more information about the DataGridView control, see DataGridView Control Overview.

To connect a DataGridView control to a DataView

1. Implement a method to handle the details of retrieving data from a database. The following code example implements a `GetData` method that initializes a SqlDataAdapter component and uses it to fill a DataSet. Be sure to set the `connectionString` variable to a value that is appropriate for your database. You will need access to a server with the AdventureWorks SQL Server sample database installed.

```
private void GetData()
{
    try
    {
        // Initialize the DataSet.
        dataSet = new DataSet();
        dataSet.Locale = CultureInfo.InvariantCulture;

        // Create the connection string for the AdventureWorks sample database.
        string connectionString = "Data Source=localhost;Initial Catalog=AdventureWorks;"
            + "Integrated Security=true;";

        // Create the command strings for querying the Contact table.
        string contactSelectCommand = "SELECT ContactID, Title, FirstName, LastName, EmailAddress, Phone
FROM Person.Contact";

        // Create the contacts data adapter.
        contactsDataAdapter = new SqlDataAdapter(
            contactSelectCommand,
            connectionString);

        // Create a command builder to generate SQL update, insert, and
        // delete commands based on the contacts select command. These are used to
        // update the database.
        SqlCommandBuilder contactsCommandBuilder = new SqlCommandBuilder(contactsDataAdapter);

        // Fill the data set with the contact information.
        contactsDataAdapter.Fill(dataSet, "Contact");
    }
    catch (SqlException ex)
    {
        MessageBox.Show(ex.Message);
    }
}
```

```
Private Sub GetData()
    Try
        ' Initialize the DataSet.
        dataSet = New DataSet()
        dataSet.Locale = CultureInfo.InvariantCulture

        ' Create the connection string for the AdventureWorks sample database.
        Dim connectionString As String = "Data Source=localhost;Initial Catalog=AdventureWorks;" _
                & "Integrated Security=true;"

        ' Create the command strings for querying the Contact table.
        Dim contactSelectCommand As String = "SELECT ContactID, Title, FirstName, LastName,
EmailAddress, Phone FROM Person.Contact"

        ' Create the contacts data adapter.
        contactsDataAdapter = New SqlDataAdapter( _
            contactSelectCommand, _
            connectionString)

        ' Create a command builder to generate SQL update, insert, and
        ' delete commands based on the contacts select command. These are used to
        ' update the database.
        Dim contactsCommandBuilder As SqlCommandBuilder = New SqlCommandBuilder(contactsDataAdapter)

        ' Fill the data set with the contact information.
        contactsDataAdapter.Fill(dataSet, "Contact")

    Catch ex As SqlException
        MessageBox.Show(ex.Message)
    End Try
End Sub
```

2. In the Load event handler of your form, bind the DataGridView control to the BindingSource component and call the `GetData` method to retrieve the data from the database. The DataView is created from a LINQ to DataSet query over the Contact DataTable and is then bound to the BindingSource component.

```
private void Form1_Load(object sender, EventArgs e)
{
    // Connect to the database and fill the DataSet.
    GetData();

    contactDataGridView.DataSource = contactBindingSource;

    // Create a LinqDataView from a LINQ to DataSet query and bind it
    // to the Windows forms control.
    EnumerableRowCollection<DataRow> contactQuery = from row in dataSet.Tables["Contact"].AsEnumerable()
                                                    where row.Field<string>("EmailAddress") != null
                                                    orderby row.Field<string>("LastName")
                                                    select row;

    contactView = contactQuery.AsDataView();

    // Bind the DataGridView to the BindingSource.
    contactBindingSource.DataSource = contactView;
    contactDataGridView.AutoResizeColumns();
}
```

```
Private Sub Form1_Load(ByVal sender As System.Object, _
                       ByVal e As System.EventArgs) _
                       Handles MyBase.Load
    ' Connect to the database and fill the DataSet.
    GetData()

    contactDataGridView.DataSource = contactBindingSource

    ' Create a LinqDataView from a LINQ to DataSet query and bind it
    ' to the Windows forms control.
    Dim contactQuery = _
        From row In dataSet.Tables("Contact").AsEnumerable() _
        Where row.Field(Of String)("EmailAddress") <> Nothing _
        Order By row.Field(Of String)("LastName") _
        Select row

    contactView = contactQuery.AsDataView()

    ' Bind the DataGridView to the BindingSource.
    contactBindingSource.DataSource = contactView
    contactDataGridView.AutoResizeColumns()
End Sub
```

See also

- Data Binding and LINQ to DataSet

Debugging LINQ to DataSet Queries

9/7/2019 • 2 minutes to read • Edit Online

Visual Studio supports the debugging of LINQ to DataSet code. However, there are some differences between debugging LINQ to DataSet code and non-LINQ to DataSet managed code. Most debugging features work with LINQ to DataSet statements, including stepping, setting breakpoints, and viewing results that are shown in debugger windows. However, deferred query execution in has some side effects that you should consider while debugging LINQ to DataSet code and there are some limitations to using Edit and Continue. This topic discusses aspects of debugging that are unique to LINQ to DataSet compared to non-LINQ to DataSet managed code.

Viewing Results

You can view the result of a LINQ to DataSet statement by using DataTips, the Watch window, and the QuickWatch dialog box. By using a source window, you can pause the pointer on a query in the source window and a DataTip will appear. You can copy a LINQ to DataSet variable and paste it into the Watch window or the QuickWatch dialog box. In LINQ to DataSet, a query is not evaluated when it is created or declared, but only when the query is executed. This is called *deferred execution*. Therefore, the query variable does not have a value until it is evaluated. For more information, see Queries in LINQ to DataSet.

The debugger must evaluate a query to display the query results. This implicit evaluation occurs when you view a LINQ to DataSet query result in the debugger, and it has some effects you should consider. Each evaluation of the query takes time. Expanding the results node takes time. For some queries, repeated evaluation might cause a noticeable performance penalty. Evaluating a query can also cause side effects, which are changes to the value of data or the state of your program. Not all queries have side effects. To determine whether a query can be safely evaluated without side effects, you must understand the code that implements the query. For more information, see Side Effects and Expressions.

Edit and Continue

Edit and Continue does not support changes to LINQ to DataSet queries. If you add, remove, or change a LINQ to DataSet statement during a debugging session, a dialog box appears that tells you the change is not supported by Edit and Continue. At that point, you can either undo the changes or stop the debugging session and restart a new session with the edited code.

In addition, Edit and Continue does not support changing the type or the value of a variable that is used in a LINQ to DataSet statement. Again, you can either undo the changes or stop and restart the debugging session.

In Visual C# in Visual Studio, you cannot use Edit and Continue on any code in a method that contains a LINQ to DataSet query.

In Visual Basic in Visual Studio, you can use Edit and Continue on non-LINQ to DataSet code, even in a method that contains a LINQ to DataSet query. You can add or remove code before the LINQ to DataSet statement, even if the changes affect line number of the LINQ to DataSet query. Your Visual Basic debugging experience for non-LINQ to DataSet code remains the same as it was before LINQ to DataSet was introduced. You cannot change, add, or remove a LINQ to DataSet query, however, unless you stop debugging to apply the changes.

See also

- Debugging Managed Code
- Programming Guide

Security (LINQ to DataSet)

9/7/2019 • 2 minutes to read • Edit Online

This topic discusses security issues in LINQ to DataSet.

Passing a Query to an Untrusted Component

A LINQ to DataSet query can be formulated in one point of a program and executed in a different one. At the point where the query is formulated, the query can reference any element that is visible at that point, such as private members of the class that the calling method belongs to, or symbols representing local variables/arguments. At execution time, the query will effectively be able to access those members that were referenced by the query at formulation, even if the calling code does not have visibility into them. The code that executes the query does not have arbitrary added visibility, in that it cannot choose what to access. It will be able to access strictly what the query accesses, and only through the query itself.

This implies that by passing a reference to a query to another piece of code the component receiving the query is being trusted with access to all public and private members that the query refers to. In general, LINQ to DataSet queries should not be passed to untrusted components, unless the query has been carefully constructed so that it does not expose information that should be kept private.

External Input

Applications often take external input (from a user or another external agent) and perform actions based on that input. In the case of LINQ to DataSet, the application might construct a query in a certain way, based on external input or use external input in the query. LINQ to DataSet queries accept parameters everywhere that literals are accepted. Application developers should use parameterized queries, rather than injecting literals from an external agent directly into the query.

Any input directly or indirectly derived from the user or an external agent might have content that leverages the syntax of the target language in order to perform unauthorized actions. This is known as a SQL injection attack, named after an attack pattern where the target language is Transact-SQL. User input injected directly into the query is used to drop a database table, cause a denial of service, or otherwise change the nature of the operation being performed. Although query composition is possible in LINQ to DataSet, it is performed through the object model API. LINQ to DataSet queries are not composed by using string manipulation or concatenation, as they are in Transact-SQL, and are not susceptible to SQL injection attacks in the traditional sense.

See also

- Programming Guide

LINQ to DataSet Examples

9/7/2019 • 2 minutes to read • Edit Online

This section provides LINQ to DataSet programming examples that use the standard query operators. The DataSet used in these examples is populated by using the `FillDataSet` method, which is specified in Loading Data Into a DataSet. For more information, see Standard Query Operators Overview (C#) or Standard Query Operators Overview (Visual Basic).

In This Section

Query Expression Examples
Contains the following examples:

- Projection
- Restriction
- Partitioning
- Ordering
- Element Operators
- Aggregate Operators
- Join Operators

Method-Based Query Examples
Contains the following examples:

- Projection
- Partitioning
- Ordering
- Set Operators
- Conversion Operators
- Element Operators
- Aggregate Operators
- Join

DataSet-Specific Operator Examples
Contains examples that demonstrate how to use the CopyToDataTable method and the DataRowComparer class.

See also

- Programming Guide
- Loading Data Into a DataSet

Query Expression Examples (LINQ to DataSet)

9/7/2019 • 2 minutes to read • Edit Online

This section provides LINQ to DataSet programming examples in query expression syntax that use the standard query operators. The DataSet used in these examples is populated by using the `FillDataSet` method, which is specified in Loading Data Into a DataSet. For more information, see Standard Query Operators Overview (C#) or Standard Query Operators Overview (Visual Basic).

In This Section

Projection
The examples in this topic demonstrate how to use the Select and SelectMany methods to query a DataSet.

Restriction
The examples in this topic demonstrate how to use the Where method to query a DataSet.

Partitioning
The examples in this topic demonstrate how to use the Skip and Take methods to query a DataSet and partition the results.

Ordering
The examples in this topic demonstrate how to use the OrderBy, OrderByDescending, Reverse, and ThenByDescending methods to query a DataSet and order the results.

Element Operators
The examples in this topic demonstrate how to use the First and ElementAt methods to get DataRow elements from a DataSet.

Aggregate Operators
The examples in this topic demonstrate how to use the Average, Count, Max, Min, and Sum methods to query a DataSet and aggregate data.

Join Operators
The examples in this topic demonstrate how to use the GroupJoin and Join methods to query a DataSet.

See also

- Method-Based Query Examples
- DataSet-Specific Operator Examples
- LINQ to DataSet Examples

Query Expression Syntax Examples: Projection (LINQ to DataSet)

9/7/2019 • 5 minutes to read • Edit Online

The examples in this topic demonstrate how to use the Select and SelectMany methods to query a DataSet using the query expression syntax.

The `FillDataSet` method used in these examples is specified in Loading Data Into a DataSet.

The examples in this topic use the Contact, Address, Product, SalesOrderHeader, and SalesOrderDetail tables in the AdventureWorks sample database.

The examples in this topic use the following `using` / `Imports` statements:

```
using System;
using System.Linq;
using System.Linq.Expressions;
using System.Collections.Generic;
using System.Data;
using System.Data.SqlClient;
using System.Data.Common;
using System.Globalization;
```

```
Option Explicit On

Imports System.Linq
Imports System.Linq.Expressions
Imports System.Collections.Generic
Imports System.Data
Imports System.Data.SqlClient
Imports System.Data.Common
Imports System.Globalization
```

For more information, see How to: Create a LINQ to DataSet Project In Visual Studio.

Select

Example

This example uses the Select method to return all the rows from the `Product` table and display the product names.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable products = ds.Tables["Product"];

IEnumerable<DataRow> query =
    from product in products.AsEnumerable()
    select product;

Console.WriteLine("Product Names:");
foreach (DataRow p in query)
{
    Console.WriteLine(p.Field<string>("Name"));
}
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim products As DataTable = ds.Tables("Product")

Dim query = From product In products.AsEnumerable() _
            Select product
Console.WriteLine("Product Names:")
For Each p In query
    Console.WriteLine(p.Field(Of String)("Name"))
Next
```

Example

This example uses Select to return a sequence of only product names.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable products = ds.Tables["Product"];

IEnumerable<string> query =
    from product in products.AsEnumerable()
    select product.Field<string>("Name");

Console.WriteLine("Product Names:");
foreach (string productName in query)
{
    Console.WriteLine(productName);
}
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim products As DataTable = ds.Tables("Product")

Dim query = From product In products.AsEnumerable() _
            Select product.Field(Of String)("Name")

Console.WriteLine("Product Names:")
For Each productName In query
    Console.WriteLine(productName)
Next
```

SelectMany

Example

This example uses `From …, …` (the equivalent of the SelectMany method) to select all orders where `TotalDue` is less than 500.00.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable contacts = ds.Tables["Contact"];
DataTable orders = ds.Tables["SalesOrderHeader"];

var query =
    from contact in contacts.AsEnumerable()
    from order in orders.AsEnumerable()
    where contact.Field<int>("ContactID") == order.Field<int>("ContactID")
        && order.Field<decimal>("TotalDue") < 500.00M
    select new
    {
        ContactID = contact.Field<int>("ContactID"),
        LastName = contact.Field<string>("LastName"),
        FirstName = contact.Field<string>("FirstName"),
        OrderID = order.Field<int>("SalesOrderID"),
        Total = order.Field<decimal>("TotalDue")
    };

foreach (var smallOrder in query)
{
    Console.WriteLine("Contact ID: {0} Name: {1}, {2} Order ID: {3} Total Due: ${4} ",
        smallOrder.ContactID, smallOrder.LastName, smallOrder.FirstName,
        smallOrder.OrderID, smallOrder.Total);
}
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim contacts As DataTable = ds.Tables("Contact")
Dim orders As DataTable = ds.Tables("SalesOrderHeader")

Dim query = _
    From contact In contacts.AsEnumerable() _
    From order In orders.AsEnumerable() _
        Where (contact.Field(Of Integer)("ContactID") = _
            order.Field(Of Integer)("ContactID")) _
            And (order.Field(Of Decimal)("TotalDue") < 500D) _
        Select New With _
        { _
            .ContactID = contact.Field(Of Integer)("ContactID"), _
            .LastName = contact.Field(Of String)("LastName"), _
            .FirstName = contact.Field(Of String)("FirstName"), _
            .OrderID = order.Field(Of Integer)("SalesOrderID"), _
            .TotalDue = order.Field(Of Decimal)("TotalDue") _
        }

For Each smallOrder In query
    Console.Write("ContactID: " & smallOrder.ContactID)
    Console.Write(" Name: {0}, {1}", smallOrder.LastName, _
                      smallOrder.FirstName)
    Console.Write(" OrderID: " & smallOrder.OrderID)
    Console.WriteLine(" TotalDue: $" & smallOrder.TotalDue)
Next
```

Example

This example uses `From …, …` (the equivalent of the SelectMany method) to select all orders where the order was made on October 1, 2002 or later.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable contacts = ds.Tables["Contact"];
DataTable orders = ds.Tables["SalesOrderHeader"];

var query =
    from contact in contacts.AsEnumerable()
    from order in orders.AsEnumerable()
    where contact.Field<int>("ContactID") == order.Field<int>("ContactID") &&
        order.Field<DateTime>("OrderDate") >= new DateTime(2002, 10, 1)
    select new
    {
        ContactID = contact.Field<int>("ContactID"),
        LastName = contact.Field<string>("LastName"),
        FirstName = contact.Field<string>("FirstName"),
        OrderID = order.Field<int>("SalesOrderID"),
        OrderDate = order.Field<DateTime>("OrderDate")
    };

foreach (var order in query)
{
    Console.WriteLine("Contact ID: {0} Name: {1}, {2} Order ID: {3} Order date: {4:d} ",
        order.ContactID, order.LastName, order.FirstName,
        order.OrderID, order.OrderDate);
}
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim contacts As DataTable = ds.Tables("Contact")
Dim orders As DataTable = ds.Tables("SalesOrderHeader")

Dim query = _
    From contact In contacts.AsEnumerable() _
    From order In orders.AsEnumerable() _
    Where contact.Field(Of Integer)("ContactID") = order.Field(Of Integer)("ContactID") And _
        order.Field(Of DateTime)("OrderDate") >= New DateTime(2002, 10, 1) _
    Select New With _
        { _
           .ContactID = contact.Field(Of Integer)("ContactID"), _
           .LastName = contact.Field(Of String)("LastName"), _
           .FirstName = contact.Field(Of String)("FirstName"), _
           .OrderID = order.Field(Of Integer)("SalesOrderID"), _
           .OrderDate = order.Field(Of DateTime)("OrderDate") _
        }

For Each order In query
    Console.Write("Contact ID: " & order.ContactID)
    Console.Write(" Name: " & order.LastName & ", " & order.FirstName)
    Console.Write(" Order ID: " & order.OrderID)
    Console.WriteLine(" Order date: {0:d} ", order.OrderDate)
Next
```

Example

This example uses a `From …, …` (the equivalent of the SelectMany method) to select all orders where the order total is greater than 10000.00 and uses `From` assignment to avoid requesting the total twice.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable contacts = ds.Tables["Contact"];
DataTable orders = ds.Tables["SalesOrderHeader"];

var query =
    from contact in contacts.AsEnumerable()
    from order in orders.AsEnumerable()
    let total = order.Field<decimal>("TotalDue")
    where contact.Field<int>("ContactID") == order.Field<int>("ContactID") &&
          total >= 10000.0M
    select new
    {
        ContactID = contact.Field<int>("ContactID"),
        LastName = contact.Field<string>("LastName"),
        OrderID = order.Field<int>("SalesOrderID"),
        total
    };
foreach (var order in query)
{
    Console.WriteLine("Contact ID: {0} Last name: {1} Order ID: {2} Total: {3}",
        order.ContactID, order.LastName, order.OrderID, order.total);
}
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim contacts As DataTable = ds.Tables("Contact")
Dim orders As DataTable = ds.Tables("SalesOrderHeader")

Dim query = _
    From contact In contacts.AsEnumerable() _
    From order In orders.AsEnumerable() _
    Let total = order.Field(Of Decimal)("TotalDue") _
    Where contact.Field(Of Integer)("ContactID") = order.Field(Of Integer)("ContactID") And _
        total >= 10000D _
    Select New With _
        { _
           .ContactID = contact.Field(Of Integer)("ContactID"), _
           .LastName = contact.Field(Of String)("LastName"), _
           .OrderID = order.Field(Of Integer)("SalesOrderID"), _
           .OrderDate = order.Field(Of DateTime)("OrderDate"), _
           total _
        }

For Each order In query
    Console.Write("Contact ID: " & order.ContactID)
    Console.Write(" Last Name: " & order.LastName)
    Console.Write(" Order ID: " & order.OrderID)
    Console.WriteLine(" Total: $" & order.total)
Next
```

See also

- Loading Data Into a DataSet
- LINQ to DataSet Examples

- Standard Query Operators Overview (C#)
- Standard Query Operators Overview (Visual Basic)

Query Expression Syntax Examples: Restriction (LINQ to DataSet)

3/12/2020 • 3 minutes to read • Edit Online

The examples in this topic demonstrate how to use the Where method to query a DataSet using the query expression syntax.

The `FillDataSet` method used in these examples is specified in Loading Data Into a DataSet.

The examples in this topic use the Contact, Address, Product, SalesOrderHeader, and SalesOrderDetail tables in the AdventureWorks sample database.

The examples in this topic use the following `using` / `Imports` statements:

```
using System;
using System.Linq;
using System.Linq.Expressions;
using System.Collections.Generic;
using System.Data;
using System.Data.SqlClient;
using System.Data.Common;
using System.Globalization;
```

For more information, see How to: Create a LINQ to DataSet Project In Visual Studio.

Where

Example

This example returns all online orders.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable orders = ds.Tables["SalesOrderHeader"];

var query =
    from order in orders.AsEnumerable()
    where order.Field<bool>("OnlineOrderFlag") == true
    select new
    {
        SalesOrderID = order.Field<int>("SalesOrderID"),
        OrderDate = order.Field<DateTime>("OrderDate"),
        SalesOrderNumber = order.Field<string>("SalesOrderNumber")
    };

foreach (var onlineOrder in query)
{
    Console.WriteLine("Order ID: {0} Order date: {1:d} Order number: {2}",
        onlineOrder.SalesOrderID,
        onlineOrder.OrderDate,
        onlineOrder.SalesOrderNumber);
}
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim orders As DataTable = ds.Tables("SalesOrderHeader")

Dim query = _
    From order In orders.AsEnumerable() _
    Where order.Field(Of Boolean)("OnlineOrderFlag") = True _
    Select New With { _
        .SalesOrderID = order.Field(Of Integer)("SalesOrderID"), _
        .OrderDate = order.Field(Of DateTime)("OrderDate"), _
        .SalesOrderNumber = order.Field(Of String)("SalesOrderNumber") _
     }

For Each onlineOrder In query
    Console.Write("Order ID: " & onlineOrder.SalesOrderID)
    Console.Write(" Order date: " & onlineOrder.OrderDate)
    Console.WriteLine(" Order number: " & onlineOrder.SalesOrderNumber)
Next
```

Example

This example returns the orders where the order quantity is greater than 2 and less than 6.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable orders = ds.Tables["SalesOrderDetail"];

var query =
    from order in orders.AsEnumerable()
    where order.Field<Int16>("OrderQty") > 2 &&
        order.Field<Int16>("OrderQty") < 6
    select new
    {
        SalesOrderID = (int)order.Field<int>("SalesOrderID"),
        OrderQty = order.Field<Int16>("OrderQty")
    };

foreach (var order in query)
{
    Console.WriteLine("Order ID: {0} Order quantity: {1}",
        order.SalesOrderID, order.OrderQty);
}
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim orders As DataTable = ds.Tables("SalesOrderDetail")

Dim query = _
    From order In orders.AsEnumerable() _
    Where order.Field(Of Short)("OrderQty") > 2 And _
            order.Field(Of Short)("OrderQty") < 6 _
    Select New With _
    { _
        .SalesOrderID = order.Field(Of Integer)("SalesOrderID"), _
        .OrderQty = order.Field(Of Short)("OrderQty") _
    }

For Each order In query
    Console.Write("Order ID: " & order.SalesOrderID)
    Console.WriteLine(" Order quantity: " & order.OrderQty)
Next
```

Example

This example returns all red colored products.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable products = ds.Tables["Product"];

var query =
    from product in products.AsEnumerable()
    where product.Field<string>("Color") == "Red"
    select new
    {
        Name = product.Field<string>("Name"),
        ProductNumber = product.Field<string>("ProductNumber"),
        ListPrice = product.Field<Decimal>("ListPrice")
    };

foreach (var product in query)
{
    Console.WriteLine("Name: {0}", product.Name);
    Console.WriteLine("Product number: {0}", product.ProductNumber);
    Console.WriteLine("List price: ${0}", product.ListPrice);
    Console.WriteLine("");
}
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim products As DataTable = ds.Tables("Product")

Dim query = _
    From product In products.AsEnumerable() _
    Where product.Field(Of String)("Color") = "Red" _
    Select New With _
       { _
           .Name = product.Field(Of String)("Name"), _
           .ProductNumber = product.Field(Of String)("ProductNumber"), _
           .ListPrice = product.Field(Of Decimal)("ListPrice") _
       }

For Each product In query
    Console.WriteLine("Name: " & product.Name)
    Console.WriteLine("Product number: " & product.ProductNumber)
    Console.WriteLine("List price: $ " & product.ListPrice & vbNewLine)
Next
```

Example

This example uses the Where method to find orders that were made after December 1, 2002 and then uses the GetChildRows method to get the details for each order.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable orders = ds.Tables["SalesOrderHeader"];

IEnumerable<DataRow> query =
    from order in orders.AsEnumerable()
    where order.Field<DateTime>("OrderDate") >= new DateTime(2002, 12, 1)
    select order;

Console.WriteLine("Orders that were made after 12/1/2002:");
foreach (DataRow order in query)
{
    Console.WriteLine("OrderID {0} Order date: {1:d} ",
        order.Field<int>("SalesOrderID"), order.Field<DateTime>("OrderDate"));
    foreach (DataRow orderDetail in order.GetChildRows("SalesOrderHeaderDetail"))
    {
        Console.WriteLine("  Product ID: {0} Unit Price {1}",
            orderDetail["ProductID"], orderDetail["UnitPrice"]);
    }
}
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim orders As DataTable = ds.Tables("SalesOrderHeader")

Dim query As IEnumerable(Of DataRow) = _
    From order In orders.AsEnumerable() _
    Where order.Field(Of DateTime)("OrderDate") >= New DateTime(2002, 12, 1) _
    Select order

Console.WriteLine("Orders that were made after 12/1/2002:")
For Each order As DataRow In query
    Console.WriteLine("OrderID {0} Order date: {1:d} ", _
        order.Field(Of Integer)("SalesOrderID"), order.Field(Of DateTime)("OrderDate"))
    For Each orderDetail As DataRow In order.GetChildRows("SalesOrderHeaderDetail")
        Console.WriteLine("  Product ID: {0} Unit Price {1}", _
            orderDetail("ProductID"), orderDetail("UnitPrice"))
    Next
Next
```

See also

- Loading Data Into a DataSet
- LINQ to DataSet Examples
- Standard Query Operators Overview (C#)
- Standard Query Operators Overview (Visual Basic)

Query Expression Syntax Examples: Partitioning (LINQ to DataSet)

9/7/2019 • 2 minutes to read • Edit Online

The examples in this topic demonstrate how to use the Skip and Take methods to query a DataSet using the query expression syntax.

The `FillDataSet` method used in these examples is specified in Loading Data Into a DataSet.

The examples in this topic use the Contact, Address, Product, SalesOrderHeader, and SalesOrderDetail tables in the AdventureWorks sample database.

The examples in this topic use the following `using` / `Imports` statements:

```
using System;
using System.Linq;
using System.Linq.Expressions;
using System.Collections.Generic;
using System.Data;
using System.Data.SqlClient;
using System.Data.Common;
using System.Globalization;
```

```
Option Explicit On

Imports System.Linq
Imports System.Linq.Expressions
Imports System.Collections.Generic
Imports System.Data
Imports System.Data.SqlClient
Imports System.Data.Common
Imports System.Globalization
```

For more information, see How to: Create a LINQ to DataSet Project In Visual Studio.

Skip

Example

This example uses the Skip method to get all but the first two addresses in Seattle.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable addresses = ds.Tables["Address"];
DataTable orders = ds.Tables["SalesOrderHeader"];

var query = (
    from address in addresses.AsEnumerable()
    from order in orders.AsEnumerable()
    where address.Field<int>("AddressID") == order.Field<int>("BillToAddressID")
         && address.Field<string>("City") == "Seattle"
    select new
    {
        City = address.Field<string>("City"),
        OrderID = order.Field<int>("SalesOrderID"),
        OrderDate = order.Field<DateTime>("OrderDate")
    }).Skip(2);

Console.WriteLine("All but first 2 orders in Seattle:");
foreach (var order in query)
{
    Console.WriteLine("City: {0} Order ID: {1} Total Due: {2:d}",
        order.City, order.OrderID, order.OrderDate);
}
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim addresses As DataTable = ds.Tables("Address")
Dim orders As DataTable = ds.Tables("SalesOrderHeader")

Dim query = ( _
    From address In addresses.AsEnumerable() _
    From order In orders.AsEnumerable() _
    Where (address.Field(Of Integer)("AddressID") = _
           order.Field(Of Integer)("BillToAddressID")) _
         And address.Field(Of String)("City") = "Seattle" _
    Select New With _
       { _
           .City = address.Field(Of String)("City"), _
           .OrderID = order.Field(Of Integer)("SalesOrderID"), _
           .OrderDate = order.Field(Of DateTime)("OrderDate") _
       }).Skip(2)

Console.WriteLine("All but first 2 orders in Seattle:")
For Each addOrder In query
    Console.Write("City: " & addOrder.City)
    Console.Write(" Order ID: " & addOrder.OrderID)
    Console.WriteLine(" Order date: " & addOrder.OrderDate)
Next
```

Take

Example

This example uses the Take method to get the first three addresses in Seattle.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable addresses = ds.Tables["Address"];
DataTable orders = ds.Tables["SalesOrderHeader"];

var query = (
    from address in addresses.AsEnumerable()
    from order in orders.AsEnumerable()
    where address.Field<int>("AddressID") == order.Field<int>("BillToAddressID")
         && address.Field<string>("City") == "Seattle"
    select new
    {
        City = address.Field<string>("City"),
        OrderID = order.Field<int>("SalesOrderID"),
        OrderDate = order.Field<DateTime>("OrderDate")
    }).Take(3);

Console.WriteLine("First 3 orders in Seattle:");
foreach (var order in query)
{
    Console.WriteLine("City: {0} Order ID: {1} Total Due: {2:d}",
        order.City, order.OrderID, order.OrderDate);
}
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim addresses As DataTable = ds.Tables("Address")
Dim orders As DataTable = ds.Tables("SalesOrderHeader")

Dim query = ( _
    From address In addresses.AsEnumerable() _
    From order In orders.AsEnumerable() _
    Where (address.Field(Of Integer)("AddressID") = _
        order.Field(Of Integer)("BillToAddressID")) _
         And address.Field(Of String)("City") = "Seattle" _
    Select New With _
    { _
        .City = address.Field(Of String)("City"), _
        .OrderID = order.Field(Of Integer)("SalesOrderID"), _
        .OrderDate = order.Field(Of DateTime)("OrderDate") _
    }).Take(3)

Console.WriteLine("First 3 orders in Seattle:")
For Each order In query
    Console.Write("City: " & order.City)
    Console.Write(" Order ID: " & order.OrderID)
    Console.WriteLine(" Order date: " & order.OrderDate)
Next
```

See also

- Loading Data Into a DataSet
- LINQ to DataSet Examples
- Standard Query Operators Overview (C#)
- Standard Query Operators Overview (Visual Basic)

Query Expression Syntax Examples: Ordering (LINQ to DataSet)

9/7/2019 • 4 minutes to read • Edit Online

The examples in this topic demonstrate how to use the OrderBy, OrderByDescending, Reverse, and ThenByDescending methods to query a DataSet and order the results using the query expression syntax.

The `FillDataSet` method used in these examples is specified in Loading Data Into a DataSet.

The examples in this topic use the Contact, Address, Product, SalesOrderHeader, and SalesOrderDetail tables in the AdventureWorks sample database.

The examples in this topic use the following `using` / `Imports` statements:

```
using System;
using System.Linq;
using System.Linq.Expressions;
using System.Collections.Generic;
using System.Data;
using System.Data.SqlClient;
using System.Data.Common;
using System.Globalization;
```

```
Option Explicit On

Imports System.Linq
Imports System.Linq.Expressions
Imports System.Collections.Generic
Imports System.Data
Imports System.Data.SqlClient
Imports System.Data.Common
Imports System.Globalization
```

For more information, see How to: Create a LINQ to DataSet Project In Visual Studio.

OrderBy

Example

This example uses OrderBy to return a list of contacts ordered by last name.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable contacts = ds.Tables["Contact"];

IEnumerable<DataRow> query =
    from contact in contacts.AsEnumerable()
    orderby contact.Field<string>("LastName")
    select contact;

Console.WriteLine("The sorted list of last names:");
foreach (DataRow contact in query)
{
    Console.WriteLine(contact.Field<string>("LastName"));
}
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim contacts As DataTable = ds.Tables("Contact")

Dim query = _
    From contact In contacts.AsEnumerable() _
    Select contact _
    Order By contact.Field(Of String)("LastName")

Console.WriteLine("The sorted list of last names:")
For Each contact In query
    Console.WriteLine(contact.Field(Of String)("LastName"))
Next
```

Example

This example uses OrderBy to sort a list of contacts by length of last name.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable contacts = ds.Tables["Contact"];

IEnumerable<DataRow> query =
    from contact in contacts.AsEnumerable()
    orderby contact.Field<string>("LastName").Length
    select contact;

Console.WriteLine("The sorted list of last names (by length):");
foreach (DataRow contact in query)
{
    Console.WriteLine(contact.Field<string>("LastName"));
}
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim contacts As DataTable = ds.Tables("Contact")

Dim query = _
    From contact In contacts.AsEnumerable() _
    Select contact _
    Order By contact.Field(Of String)("LastName").Length

Console.WriteLine("The sorted list of last names (by length):")
For Each contact In query
    Console.WriteLine(contact.Field(Of String)("LastName"))
Next
```

OrderByDescending

Example

This example uses `orderby… descending` (`Order By … Descending`), which is equivalent to the OrderByDescending method, to sort the price list from highest to lowest.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable products = ds.Tables["Product"];

IEnumerable<Decimal> query =
    from product in products.AsEnumerable()
    orderby product.Field<Decimal>("ListPrice") descending
    select product.Field<Decimal>("ListPrice");

Console.WriteLine("The list price from highest to lowest:");
foreach (Decimal product in query)
{
    Console.WriteLine(product);
}
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim products As DataTable = ds.Tables("Product")

Dim query = _
    From product In products.AsEnumerable() _
    Select product _
    Order By product.Field(Of Decimal)("ListPrice") Descending

Console.WriteLine("The list price From highest to lowest:")

For Each product In query
    Console.WriteLine(product.Field(Of Decimal)("ListPrice"))
Next
```

Reverse

Example

This example uses Reverse to create a list of orders where `OrderDate` is earlier than Feb 20, 2002.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable orders = ds.Tables["SalesOrderHeader"];

IEnumerable<DataRow> query = (
    from order in orders.AsEnumerable()
    where order.Field<DateTime>("OrderDate") < new DateTime(2002, 02, 20)
    select order).Reverse();

Console.WriteLine("A backwards list of orders where OrderDate < Feb 20, 2002");
foreach (DataRow order in query)
{
    Console.WriteLine(order.Field<DateTime>("OrderDate"));
}
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim orders As DataTable = ds.Tables("SalesOrderHeader")

Dim query = ( _
    From order In orders.AsEnumerable() _
    Where order.Field(Of DateTime)("OrderDate") < New DateTime(2002, 2, 20) _
    Select order).Reverse()

Console.WriteLine("A backwards list of orders where OrderDate < Feb 20, 2002")

For Each order In query
    Console.WriteLine(order.Field(Of DateTime)("OrderDate"))
Next
```

ThenByDescending

Example

This example uses `OrderBy… Descending` , which is equivalent to the ThenByDescending method, to sort a list of products, first by name and then by list price, from highest to lowest.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable products = ds.Tables["Product"];

IEnumerable<DataRow> query =
    from product in products.AsEnumerable()
    orderby product.Field<string>("Name"),
        product.Field<Decimal>("ListPrice") descending
    select product;

foreach (DataRow product in query)
{
    Console.WriteLine("Product ID: {0} Product Name: {1} List Price {2}",
        product.Field<int>("ProductID"),
        product.Field<string>("Name"),
        product.Field<Decimal>("ListPrice"));
}
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim products As DataTable = ds.Tables("Product")

Dim query = _
    From product In products.AsEnumerable() _
    Order By product.Field(Of String)("Name"), _
            product.Field(Of Decimal)("ListPrice") Descending _
    Select product

For Each product In query
    Console.Write("Product ID: " & product.Field(Of Integer)("ProductID"))
    Console.Write(" Product Name: " & product.Field(Of String)("Name"))
    Console.WriteLine(" List Price: " & product.Field(Of Decimal)("ListPrice"))
Next
```

See also

- Loading Data Into a DataSet
- LINQ to DataSet Examples
- Standard Query Operators Overview (C#)
- Standard Query Operators Overview (Visual Basic)

Query Expression Syntax Examples: Element Operators (LINQ to DataSet)

9/7/2019 • 2 minutes to read • Edit Online

The examples in this topic demonstrate how to use the First and ElementAt methods to get DataRow elements from a DataSet using the query expression syntax.

The `FillDataSet` method used in these examples is specified in Loading Data Into a DataSet.

The examples in this topic use the Contact, Address, Product, SalesOrderHeader, and SalesOrderDetail tables in the AdventureWorks sample database.

The examples in this topic use the following `using` / `Imports` statements:

```
using System;
using System.Linq;
using System.Linq.Expressions;
using System.Collections.Generic;
using System.Data;
using System.Data.SqlClient;
using System.Data.Common;
using System.Globalization;
```

```
Option Explicit On

Imports System.Linq
Imports System.Linq.Expressions
Imports System.Collections.Generic
Imports System.Data
Imports System.Data.SqlClient
Imports System.Data.Common
Imports System.Globalization
```

For more information, see How to: Create a LINQ to DataSet Project In Visual Studio.

ElementAt

Example

This example uses the ElementAt method to retrieve the fifth address where `PostalCode` == "M4B 1V7".

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable addresses = ds.Tables["Address"];

var fifthAddress = (
    from address in addresses.AsEnumerable()
    where address.Field<string>("PostalCode") == "M4B 1V7"
    select address.Field<string>("AddressLine1"))
.ElementAt(5);

Console.WriteLine("Fifth address where PostalCode = 'M4B 1V7': {0}",
    fifthAddress);
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim addresses As DataTable = ds.Tables("Address")

Dim fifthAddress = ( _
    From address In addresses.AsEnumerable() _
    Where address.Field(Of String)("PostalCode") = "M4B 1V7" _
    Select address.Field(Of String)("AddressLine1")).ElementAt(5)

Console.WriteLine("Fifth address where PostalCode = 'M4B 1V7': " & _
        fifthAddress)
```

First

Example

This example uses the First method to return the first contact whose first name is 'Brooke'.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable contacts = ds.Tables["Contact"];

DataRow query = (
    from contact in contacts.AsEnumerable()
    where (string)contact["FirstName"] == "Brooke"
    select contact)
    .First();

Console.WriteLine("ContactID: " + query.Field<int>("ContactID"));
Console.WriteLine("FirstName: " + query.Field<string>("FirstName"));
Console.WriteLine("LastName: " + query.Field<string>("LastName"));
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim contacts As DataTable = ds.Tables("Contact")

Dim query = ( _
    From contact In contacts.AsEnumerable() _
    Where contact.Field(Of String)("FirstName") = "Brooke" _
    Select contact).First()

Console.WriteLine("ContactID: " & query.Field(Of Integer)("ContactID"))
Console.WriteLine("FirstName: " & query.Field(Of String)("FirstName"))
Console.WriteLine("LastName: " & query.Field(Of String)("LastName"))
```

See also

- Loading Data Into a DataSet
- LINQ to DataSet Examples
- Standard Query Operators Overview (C#)
- Standard Query Operators Overview (Visual Basic)

Query Expression Syntax Examples: Aggregate Operators (LINQ to DataSet)

9/7/2019 • 9 minutes to read • Edit Online

The examples in this topic demonstrate how to use the Average, Count, Max, Min, and Sum methods to query a DataSet and aggregate data using query expression syntax.

The `FillDataSet` method used in these examples is specified in Loading Data Into a DataSet.

The examples in this topic use the Contact, Address, Product, SalesOrderHeader, and SalesOrderDetail tables in the AdventureWorks sample database.

The examples in this topic use the following `using` / `Imports` statements:

```
using System;
using System.Linq;
using System.Linq.Expressions;
using System.Collections.Generic;
using System.Data;
using System.Data.SqlClient;
using System.Data.Common;
using System.Globalization;
```

```
Option Explicit On

Imports System.Linq
Imports System.Linq.Expressions
Imports System.Collections.Generic
Imports System.Data
Imports System.Data.SqlClient
Imports System.Data.Common
Imports System.Globalization
```

For more information, see How to: Create a LINQ to DataSet Project In Visual Studio.

Average

Example

This example uses the Average method to find the average list price of the products of each style.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

var products = ds.Tables["Product"].AsEnumerable();

var query = from product in products
            group product by product.Field<string>("Style") into g
            select new
            {
                Style = g.Key,
                AverageListPrice =
                    g.Average(product => product.Field<Decimal>("ListPrice"))
            };

foreach (var product in query)
{
    Console.WriteLine("Product style: {0} Average list price: {1}",
        product.Style, product.AverageListPrice);
}
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim products As IEnumerable(Of DataRow) = _
    ds.Tables("Product").AsEnumerable()

Dim query = _
    From product In products _
    Group product By style = product.Field(Of String)("Style") Into g = Group _
    Select New With _
        { _
            .Style = style, _
            .AverageListPrice = g.Average(Function(product) _
                    product.Field(Of Decimal)("ListPrice")) _
        }

For Each product In query
    Console.WriteLine("Product style: {0} Average list price: {1}", _
        product.Style, product.AverageListPrice)
Next
```

Example

This example uses Average to get the average total due for each contact ID.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable orders = ds.Tables["SalesOrderHeader"];

var query =
    from order in orders.AsEnumerable()
    group order by order.Field<Int32>("ContactID") into g
    select new
    {
        Category = g.Key,
        averageTotalDue =
            g.Average(order => order.Field<decimal>("TotalDue"))
    };

foreach (var order in query)
{
    Console.WriteLine("ContactID = {0} \t Average TotalDue = {1}",
        order.Category,
        order.averageTotalDue);
}
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim orders As DataTable = ds.Tables("SalesOrderHeader")

Dim query = _
    From order In orders.AsEnumerable() _
    Group order By contactID = order.Field(Of Int32)("ContactID") Into g = Group _
    Select New With _
    { _
        .Category = contactID, _
        .averageTotalDue = g.Average(Function(order) order. _
                Field(Of Decimal)("TotalDue")) _
    }

For Each order In query
    Console.WriteLine("ContactID = {0} " & vbTab & _
        " Average TotalDue = {1}", order.Category, _
        order.averageTotalDue)
Next
```

Example

This example uses Average to get the orders with the average `TotalDue` for each contact.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable orders = ds.Tables["SalesOrderHeader"];

var query =
    from order in orders.AsEnumerable()
    group order by order.Field<Int32>("ContactID") into g
    let averageTotalDue = g.Average(order => order.Field<decimal>("TotalDue"))
    select new
    {
        Category = g.Key,
        CheapestProducts =
            g.Where(order => order.Field<decimal>("TotalDue") ==
                        averageTotalDue)
    };

foreach (var orderGroup in query)
{
    Console.WriteLine("ContactID: {0}", orderGroup.Category);
    foreach (var order in orderGroup.CheapestProducts)
    {
        Console.WriteLine("Average total due for SalesOrderID {1} is: {0}",
            order.Field<decimal>("TotalDue"),
            order.Field<Int32>("SalesOrderID"));
    }
    Console.WriteLine("");
}
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim orders As DataTable = ds.Tables("SalesOrderHeader")

Dim query = _
    From order In orders.AsEnumerable() _
    Group order By contactID = order.Field(Of Int32)("ContactID") Into g = Group _
    Let averageTotalDue = g.Average(Function(order) order.Field(Of Decimal)("TotalDue")) _
    Select New With _
    { _
        .Category = contactID, _
        .CheapestProducts = g.Where(Function(order) order. _
                Field(Of Decimal)("TotalDue") = averageTotalDue) _
    }

For Each orderGroup In query
    Console.WriteLine("ContactID: " & orderGroup.Category)
    For Each order In orderGroup.CheapestProducts
        Console.WriteLine("Average total due for SalesOrderID {1} is: {0}", _
            order.Field(Of Decimal)("TotalDue"), _
            order.Field(Of Int32)("SalesOrderID"))
    Next
    Console.WriteLine("")
Next
```

Count

Example

This example uses Count to return a list of contact IDs and how many orders each has.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable contacts = ds.Tables["Contact"];

var query = from contact in contacts.AsEnumerable()
            select new
            {
                CustomerID = contact.Field<int>("ContactID"),
                OrderCount =
                    contact.GetChildRows("SalesOrderContact").Count()
            };

foreach (var contact in query)
{
    Console.WriteLine("CustomerID = {0} \t OrderCount = {1}",
        contact.CustomerID,
        contact.OrderCount);
}
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim contacts As DataTable = ds.Tables("Contact")

Dim query = _
    From contact In contacts.AsEnumerable() _
    Select New With _
    { _
        .ContactID = contact.Field(Of Integer)("ContactID"), _
        .OrderCount = contact.GetChildRows("SalesOrderContact").Count() _
    }

For Each contact In query
    Console.Write("CustomerID = " & contact.ContactID)
    Console.WriteLine(vbTab & "OrderCount = " & contact.OrderCount)
Next
```

Example

This example groups products by color and uses Count to return the number of products in each color group.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable products = ds.Tables["Product"];

var query =
    from product in products.AsEnumerable()
    group product by product.Field<string>("Color") into g
    select new { Color = g.Key, ProductCount = g.Count() };

foreach (var product in query)
{
    Console.WriteLine("Color = {0} \t ProductCount = {1}",
        product.Color,
        product.ProductCount);
}
```

```
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim products As DataTable = ds.Tables("Product")

Dim query = _
    From product In products.AsEnumerable() _
    Group product By color = product.Field(Of String)("Color") Into g = Group _
    Select New With {.Color = color, .ProductCount = g.Count()}

For Each product In query
    Console.WriteLine("Color = {0} " & vbTab & "ProductCount = {1}", _
        product.Color, _
        product.ProductCount)
Next
```

Max

Example

This example uses the Max method to get the largest total due for each contact ID.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable orders = ds.Tables["SalesOrderHeader"];

var query =
    from order in orders.AsEnumerable()
    group order by order.Field<Int32>("ContactID") into g
    select new
    {
        Category = g.Key,
        maxTotalDue =
            g.Max(order => order.Field<decimal>("TotalDue"))
    };

foreach (var order in query)
{
    Console.WriteLine("ContactID = {0} \t Maximum TotalDue = {1}",
        order.Category, order.maxTotalDue);
}
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim orders As DataTable = ds.Tables("SalesOrderHeader")

Dim query = _
   From order In orders.AsEnumerable() _
   Group order By contactID = order.Field(Of Int32)("ContactID") Into g = Group _
   Select New With _
   { _
       .Category = contactID, _
       .maxTotalDue = _
           g.Max(Function(order) order.Field(Of Decimal)("TotalDue")) _
   }
For Each order In query
    Console.WriteLine("ContactID = {0} " & vbTab & _
        " Maximum TotalDue = {1}", _
        order.Category, order.maxTotalDue)
Next
```

Example

This example uses the Max method to get the orders with the largest `TotalDue` for each contact ID.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable orders = ds.Tables["SalesOrderHeader"];

var query =
    from order in orders.AsEnumerable()
    group order by order.Field<Int32>("ContactID") into g
    let maxTotalDue = g.Max(order => order.Field<decimal>("TotalDue"))
    select new
    {
        Category = g.Key,
        CheapestProducts =
            g.Where(order => order.Field<decimal>("TotalDue") ==
                        maxTotalDue)
    };

foreach (var orderGroup in query)
{
    Console.WriteLine("ContactID: {0}", orderGroup.Category);
    foreach (var order in orderGroup.CheapestProducts)
    {
        Console.WriteLine("MaxTotalDue {0} for SalesOrderID {1}: ",
            order.Field<decimal>("TotalDue"),
            order.Field<Int32>("SalesOrderID"));
    }
}
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim orders As DataTable = ds.Tables("SalesOrderHeader")

Dim query = _
    From order In orders.AsEnumerable() _
    Group order By contactID = order.Field(Of Int32)("ContactID") Into g = Group _
    Let maxTotalDue = g.Max(Function(order) order.Field(Of Decimal)("TotalDue")) _
    Select New With _
    { _
        .Category = contactID, _
        .CheapestProducts = _
            g.Where(Function(order) order. _
                    Field(Of Decimal)("TotalDue") = maxTotalDue) _
    }

For Each orderGroup In query
    Console.WriteLine("ContactID: " & orderGroup.Category)
    For Each order In orderGroup.CheapestProducts
        Console.WriteLine("MaxTotalDue {0} for SalesOrderID {1} ", _
            order.Field(Of Decimal)("TotalDue"), _
            order.Field(Of Int32)("SalesOrderID"))
    Next
Next
```

Min

Example

This example uses the Min method to get the smallest total due for each contact ID.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable orders = ds.Tables["SalesOrderHeader"];

var query =
    from order in orders.AsEnumerable()
    group order by order.Field<Int32>("ContactID") into g
    select new
    {
        Category = g.Key,
        smallestTotalDue =
            g.Min(order => order.Field<decimal>("TotalDue"))
    };

foreach (var order in query)
{
    Console.WriteLine("ContactID = {0} \t Minimum TotalDue = {1}",
        order.Category, order.smallestTotalDue);
}
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim orders As DataTable = ds.Tables("SalesOrderHeader")

Dim query = _
    From order In orders.AsEnumerable() _
    Group order By contactID = order.Field(Of Int32)("ContactID") Into g = Group _
    Select New With _
    { _
       .Category = contactID, _
       .smallestTotalDue = g.Min(Function(order) _
                order.Field(Of Decimal)("TotalDue")) _
    }

For Each order In query
    Console.WriteLine("ContactID = {0} " & vbTab & _
        "Minimum TotalDue = {1}", order.Category, order.smallestTotalDue)
Next
```

Example

This example uses the Min method to get the orders with the smallest total due for each contact.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable orders = ds.Tables["SalesOrderHeader"];

var query =
    from order in orders.AsEnumerable()
    group order by order.Field<Int32>("ContactID") into g
    let minTotalDue = g.Min(order => order.Field<decimal>("TotalDue"))
    select new
    {
        Category = g.Key,
        smallestTotalDue =
            g.Where(order => order.Field<decimal>("TotalDue") ==
                        minTotalDue)
    };

foreach (var orderGroup in query)
{
    Console.WriteLine("ContactID: {0}", orderGroup.Category);
    foreach (var order in orderGroup.smallestTotalDue)
    {
        Console.WriteLine("Mininum TotalDue {0} for SalesOrderID {1}: ",
            order.Field<decimal>("TotalDue"),
            order.Field<Int32>("SalesOrderID"));
    }
    Console.WriteLine("");
}
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim orders As DataTable = ds.Tables("SalesOrderHeader")

Dim query = _
    From order In orders.AsEnumerable() _
    Group order By contactID = order.Field(Of Int32)("ContactID") Into g = Group _
    Let minTotalDue = g.Min(Function(order) order.Field(Of Decimal)("TotalDue")) _
    Select New With _
    { _
       .Category = contactID, _
       .smallestTotalDue = g.Where(Function(order) order. _
            Field(Of Decimal)("TotalDue") = minTotalDue) _
    }

For Each orderGroup In query
    Console.WriteLine("ContactID: " & orderGroup.Category)
    For Each order In orderGroup.smallestTotalDue
        Console.WriteLine("Mininum TotalDue {0} for SalesOrderID {1} ", _
            order.Field(Of Decimal)("TotalDue"), _
            order.Field(Of Int32)("SalesOrderID"))
    Next
    Console.WriteLine("")
Next
```

Sum

Example

This example uses the Sum method to get the total due for each contact ID.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable orders = ds.Tables["SalesOrderHeader"];

var query =
    from order in orders.AsEnumerable()
    group order by order.Field<Int32>("ContactID") into g
    select new
    {
        Category = g.Key,
        TotalDue = g.Sum(order => order.Field<decimal>("TotalDue")),
    };
foreach (var order in query)
{
    Console.WriteLine("ContactID = {0} \t TotalDue sum = {1}",
        order.Category, order.TotalDue);
}
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim orders As DataTable = ds.Tables("SalesOrderHeader")

Dim query = _
    From order In orders.AsEnumerable() _
    Group order By contactID = order.Field(Of Int32)("ContactID") Into g = Group _
    Select New With _
       { _
           .Category = contactID, _
           .TotalDue = g.Sum(Function(order) order. _
                Field(Of Decimal)("TotalDue")) _
       }

For Each order In query
    Console.WriteLine("ContactID = {0} " & vbTab & "TotalDue sum = {1}", _
        order.Category, order.TotalDue)
Next
```

See also

- Loading Data Into a DataSet
- LINQ to DataSet Examples
- Standard Query Operators Overview (C#)
- Standard Query Operators Overview (Visual Basic)

Query Expression Syntax Examples: Join Operators (LINQ to DataSet)

9/7/2019 • 4 minutes to read • Edit Online

Joining is an important operation in queries that target data sources that have no navigable relationships to each other, such as relational database tables. A join of two data sources is the association of objects in one data source with objects that share a common attribute in the other data source. For more information, see Standard Query Operators Overview (C#) or Standard Query Operators Overview (Visual Basic).

The examples in this topic demonstrate how to use the GroupJoin and Join methods to query a DataSet using the query expression syntax.

The `FillDataSet` method used in these examples is specified in Loading Data Into a DataSet.

The examples in this topic use the Contact, Address, Product, SalesOrderHeader, and SalesOrderDetail tables in the AdventureWorks sample database.

The examples in this topic use the following `using` / `Imports` statements:

```
using System;
using System.Linq;
using System.Linq.Expressions;
using System.Collections.Generic;
using System.Data;
using System.Data.SqlClient;
using System.Data.Common;
using System.Globalization;
```

```
Option Explicit On

Imports System.Linq
Imports System.Linq.Expressions
Imports System.Collections.Generic
Imports System.Data
Imports System.Data.SqlClient
Imports System.Data.Common
Imports System.Globalization
```

For more information, see How to: Create a LINQ to DataSet Project In Visual Studio.

GroupJoin

Example

This example performs a GroupJoin over the `SalesOrderHeader` and `SalesOrderDetail` tables to find the number of orders per customer. A group join is the equivalent of a left outer join, which returns each element of the first (left) data source, even if no correlated elements are in the other data source.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

var orders = ds.Tables["SalesOrderHeader"].AsEnumerable();
var details = ds.Tables["SalesOrderDetail"].AsEnumerable();

var query =
    from order in orders
    join detail in details
    on order.Field<int>("SalesOrderID")
    equals detail.Field<int>("SalesOrderID") into ords
    select new
    {
        CustomerID =
            order.Field<int>("SalesOrderID"),
        ords = ords.Count()
    };

foreach (var order in query)
{
    Console.WriteLine("CustomerID: {0}  Orders Count: {1}",
        order.CustomerID,
        order.ords);
}
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim orders = ds.Tables("SalesOrderHeader").AsEnumerable()
Dim details = ds.Tables("SalesOrderDetail").AsEnumerable()

Dim query = _
        From order In orders _
        Group Join detail In details _
            On order.Field(Of Integer)("SalesOrderID") _
            Equals detail.Field(Of Integer)("SalesOrderID") Into ords = Group _
        Select New With _
            { _
                .CustomerID = order.Field(Of Integer)("SalesOrderID"), _
                .ords = ords.Count() _
            }

For Each order In query
    Console.WriteLine("CustomerID: {0}  Orders Count: {1}", _
        order.CustomerID, order.ords)
Next
```

Example

This example performs a GroupJoin over the `Contact` and `SalesOrderHeader` tables. A group join is the equivalent of a left outer join, which returns each element of the first (left) data source, even if no correlated elements are in the other data source.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable contacts = ds.Tables["Contact"];
DataTable orders = ds.Tables["SalesOrderHeader"];

var query =
    from contact in contacts.AsEnumerable()
    join order in orders.AsEnumerable()
    on contact.Field<Int32>("ContactID") equals
    order.Field<Int32>("ContactID")
    select new
    {
        ContactID = contact.Field<Int32>("ContactID"),
        SalesOrderID = order.Field<Int32>("SalesOrderID"),
        FirstName = contact.Field<string>("FirstName"),
        Lastname = contact.Field<string>("Lastname"),
        TotalDue = order.Field<decimal>("TotalDue")
    };

foreach (var contact_order in query)
{
    Console.WriteLine("ContactID: {0} "
                    + "SalesOrderID: {1} "
                    + "FirstName: {2} "
                    + "Lastname: {3} "
                    + "TotalDue: {4}",
        contact_order.ContactID,
        contact_order.SalesOrderID,
        contact_order.FirstName,
        contact_order.Lastname,
        contact_order.TotalDue);
}
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim contacts As DataTable = ds.Tables("Contact")
Dim orders As DataTable = ds.Tables("SalesOrderHeader")

Dim query = _
    From contact In contacts.AsEnumerable(), order In orders.AsEnumerable() _
    Where (contact.Field(Of Integer)("ContactID") = _
        order.Field(Of Integer)("ContactID")) _
    Select New With _
        { _
            .ContactID = contact.Field(Of Integer)("ContactID"), _
            .SalesOrderID = order.Field(Of Integer)("SalesOrderID"), _
            .FirstName = contact.Field(Of String)("FirstName"), _
            .Lastname = contact.Field(Of String)("Lastname"), _
            .TotalDue = order.Field(Of Decimal)("TotalDue") _
        }

For Each contact_order In query
    Console.Write("ContactID: " & contact_order.ContactID)
    Console.Write(" SalesOrderID: " & contact_order.SalesOrderID)
    Console.Write(" FirstName: " & contact_order.FirstName)
    Console.Write(" Lastname: " & contact_order.Lastname)
    Console.WriteLine(" TotalDue: " & contact_order.TotalDue)
Next
```

Join

Example

This example performs a join over the `SalesOrderHeader` and `SalesOrderDetail` tables to get online orders from the month of August.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable orders = ds.Tables["SalesOrderHeader"];
DataTable details = ds.Tables["SalesOrderDetail"];

var query =
    from order in orders.AsEnumerable()
    join detail in details.AsEnumerable()
    on order.Field<int>("SalesOrderID") equals
        detail.Field<int>("SalesOrderID")
    where order.Field<bool>("OnlineOrderFlag") == true
    && order.Field<DateTime>("OrderDate").Month == 8
    select new
    {
        SalesOrderID =
            order.Field<int>("SalesOrderID"),
        SalesOrderDetailID =
            detail.Field<int>("SalesOrderDetailID"),
        OrderDate =
            order.Field<DateTime>("OrderDate"),
        ProductID =
            detail.Field<int>("ProductID")
    };

foreach (var order in query)
{
    Console.WriteLine("{0}\t{1}\t{2:d}\t{3}",
        order.SalesOrderID,
        order.SalesOrderDetailID,
        order.OrderDate,
        order.ProductID);
}
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim orders As DataTable = ds.Tables("SalesOrderHeader")
Dim details As DataTable = ds.Tables("SalesOrderDetail")


Dim query = _
    From order In orders.AsEnumerable() _
    Join detail In details.AsEnumerable() _
    On order.Field(Of Integer)("SalesOrderID") Equals _
            detail.Field(Of Integer)("SalesOrderID") _
    Where order.Field(Of Boolean)("OnlineOrderFlag") = True And _
            order.Field(Of DateTime)("OrderDate").Month = 8 _
    Select New With _
    { _
        .SalesOrderID = order.Field(Of Integer)("SalesOrderID"), _
        .SalesOrderDetailID = detail.Field(Of Integer)("SalesOrderDetailID"), _
        .OrderDate = order.Field(Of DateTime)("OrderDate"), _
        .ProductID = detail.Field(Of Integer)("ProductID") _
    }

For Each order In query
    Console.WriteLine(order.SalesOrderID & vbTab & _
        order.SalesOrderDetailID & vbTab & _
        order.OrderDate & vbTab & _
        order.ProductID)
Next
```

See also

- Loading Data Into a DataSet
- LINQ to DataSet Examples
- Standard Query Operators Overview (C#)
- Standard Query Operators Overview (Visual Basic)

Method-Based Query Examples (LINQ to DataSet)

9/7/2019 • 2 minutes to read • Edit Online

This section provides LINQ to DataSet programming examples in method-based query syntax that use the standard query operators. The DataSet used in these examples is populated by using the `FillDataSet` method, which is specified in Loading Data Into a DataSet. For more information, see Standard Query Operators Overview (C#) or Standard Query Operators Overview (Visual Basic).

In This Section

Projection
The examples in this topic demonstrate how to use the Select and SelectMany methods to query a DataSet.

Partitioning
The examples in this topic demonstrate how to use the Skip and Take methods to query a DataSet and partition the results.

Ordering
The examples in this topic demonstrate how to use the OrderBy, OrderByDescending, Reverse, and ThenByDescending methods to query a DataSet and order the results.

Set Operators
The examples in this topic demonstrate how to use the Distinct, Except, Intersect, and Union operators to perform value-based comparison operations on sets of data rows.

Conversion Operators
The examples in this topic demonstrate how to use the ToArray, ToDictionary, and ToList methods to immediately execute a query expression.

Element Operators
The examples in this topic demonstrate how to use the First and ElementAt methods to get DataRow elements from a DataSet.

Aggregate Operators
The examples in this topic demonstrate how to use the Average, Count, Max, Min, and Sum methods to query a DataSet and aggregate data.

Join
The examples in this topic demonstrate how to use the GroupJoin and Join methods to query a DataSet.

See also

- Query Expression Examples
- DataSet-Specific Operator Examples
- LINQ to DataSet Examples

Method-Based Query Syntax Examples: Projection (LINQ to DataSet)

9/7/2019 • 3 minutes to read • Edit Online

The examples in this topic demonstrate how to use the Select and SelectMany methods to query a DataSet using the method-based query syntax.

The `FillDataSet` method used in these examples is specified in Loading Data Into a DataSet.

The examples in this topic use the Contact, Address, Product, SalesOrderHeader, and SalesOrderDetail tables in the AdventureWorks sample database.

The examples in this topic use the following `using` / `Imports` statements:

```
using System;
using System.Linq;
using System.Linq.Expressions;
using System.Collections.Generic;
using System.Data;
using System.Data.SqlClient;
using System.Data.Common;
using System.Globalization;
```

```
Option Explicit On

Imports System.Linq
Imports System.Linq.Expressions
Imports System.Collections.Generic
Imports System.Data
Imports System.Data.SqlClient
Imports System.Data.Common
Imports System.Globalization
```

For more information, see How to: Create a LINQ to DataSet Project In Visual Studio.

Select

Example

This example uses the Select method to project the `Name`, `ProductNumber`, and `ListPrice` properties to a sequence of anonymous types. The `ListPrice` property is also renamed to `Price` in the resulting type.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable products = ds.Tables["Product"];

var query = products.AsEnumerable().
    Select(product => new
    {
        ProductName = product.Field<string>("Name"),
        ProductNumber = product.Field<string>("ProductNumber"),
        Price = product.Field<decimal>("ListPrice")
    });

Console.WriteLine("Product Info:");
foreach (var productInfo in query)
{
    Console.WriteLine("Product name: {0} Product number: {1} List price: ${2} ",
        productInfo.ProductName, productInfo.ProductNumber, productInfo.Price);
}
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim products As DataTable = ds.Tables("Product")

Dim query = products.AsEnumerable() _
    .Select(Function(product As DataRow) New With _
    { _
        .ProductName = product.Field(Of String)("Name"), _
        .ProductNumber = product.Field(Of String)("ProductNumber"), _
        .Price = product.Field(Of Decimal)("ListPrice") _
    })

Console.WriteLine("Product Info:")
For Each product In query
    Console.Write("Product name: " & product.ProductName)
    Console.Write("Product number: " & product.ProductNumber)
    Console.WriteLine("List price: $ " & product.Price)
Next
```

SelectMany

Example

This example uses the SelectMany method to select all orders where `TotalDue` is less than 500.00.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

var contacts = ds.Tables["Contact"].AsEnumerable();
var orders = ds.Tables["SalesOrderHeader"].AsEnumerable();

var query =
    contacts.SelectMany(
        contact => orders.Where(order =>
            (contact.Field<Int32>("ContactID") == order.Field<Int32>("ContactID"))
                && order.Field<decimal>("TotalDue") < 500.00M)
            .Select(order => new
            {
                ContactID = contact.Field<int>("ContactID"),
                LastName = contact.Field<string>("LastName"),
                FirstName = contact.Field<string>("FirstName"),
                OrderID = order.Field<int>("SalesOrderID"),
                Total = order.Field<decimal>("TotalDue")
            }));

foreach (var smallOrder in query)
{
    Console.WriteLine("Contact ID: {0} Name: {1}, {2} Order ID: {3} Total Due: ${4} ",
        smallOrder.ContactID, smallOrder.LastName, smallOrder.FirstName,
        smallOrder.OrderID, smallOrder.Total);
}
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim contacts = ds.Tables("Contact").AsEnumerable()
Dim orders = ds.Tables("SalesOrderHeader").AsEnumerable()

Dim query = _
    contacts.SelectMany( _
        Function(contact) orders.Where(Function(order) _
            (contact.Field(Of Int32)("ContactID") = order.Field(Of Int32)("ContactID")) _
                And order.Field(Of Decimal)("TotalDue") < 500D) _
            .Select(Function(order) New With _
            { _
                .ContactID = contact.Field(Of Integer)("ContactID"), _
                .LastName = contact.Field(Of String)("LastName"), _
                .FirstName = contact.Field(Of String)("FirstName"), _
                .OrderID = order.Field(Of Integer)("SalesOrderID"), _
                .Total = order.Field(Of Decimal)("TotalDue") _
            }))

For Each smallOrder In query
    Console.Write("ContactID: " & smallOrder.ContactID)
    Console.Write(" Name: " & smallOrder.LastName & ", " & smallOrder.FirstName)
    Console.Write(" Order ID: " & smallOrder.OrderID)
    Console.WriteLine(" Total Due: $" & smallOrder.Total)
Next
```

Example

This example uses the SelectMany method to select all orders where the order was made on October 1, 2002 or later.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

var contacts = ds.Tables["Contact"].AsEnumerable();
var orders = ds.Tables["SalesOrderHeader"].AsEnumerable();

var query =
    contacts.SelectMany(
        contact => orders.Where(order =>
            (contact.Field<Int32>("ContactID") == order.Field<Int32>("ContactID"))
                && order.Field<DateTime>("OrderDate") >= new DateTime(2002, 10, 1))
            .Select(order => new
            {
                ContactID = contact.Field<int>("ContactID"),
                LastName = contact.Field<string>("LastName"),
                FirstName = contact.Field<string>("FirstName"),
                OrderID = order.Field<int>("SalesOrderID"),
                OrderDate = order.Field<DateTime>("OrderDate")
            }));

foreach (var order in query)
{
    Console.WriteLine("Contact ID: {0} Name: {1}, {2} Order ID: {3} Order date: {4:d} ",
        order.ContactID, order.LastName, order.FirstName,
        order.OrderID, order.OrderDate);
}
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim contacts = ds.Tables("Contact").AsEnumerable()
Dim orders = ds.Tables("SalesOrderHeader").AsEnumerable()

Dim query = _
    contacts.SelectMany( _
        Function(contact) orders.Where(Function(order) _
            (contact.Field(Of Int32)("ContactID") = order.Field(Of Int32)("ContactID")) _
                And order.Field(Of DateTime)("OrderDate") >= New DateTime(2002, 10, 1)) _
            .Select(Function(order) New With _
                { _
                    .ContactID = contact.Field(Of Integer)("ContactID"), _
                    .LastName = contact.Field(Of String)("LastName"), _
                    .FirstName = contact.Field(Of String)("FirstName"), _
                    .OrderID = order.Field(Of Integer)("SalesOrderID"), _
                    .OrderDate = order.Field(Of DateTime)("OrderDate") _
                }))

For Each order In query
    Console.Write("Contact ID: " & order.ContactID)
    Console.Write(" Name: " & order.LastName & ", " & order.FirstName)
    Console.Write(" Order ID: " & order.OrderID)
    Console.WriteLine(" Order date: {0:d}", order.OrderDate)
Next
```

See also

- Loading Data Into a DataSet
- LINQ to DataSet Examples

- Standard Query Operators Overview (C#)
- Standard Query Operators Overview (Visual Basic)

Method-Based Query Syntax Examples: Partitioning (LINQ

9/7/2019 • 5 minutes to read • Edit Online

The examples in this topic demonstrate how to use the Skip, SkipWhile, Take, and TakeWhile methods to query a DataSet using the query expression syntax.

The `FillDataSet` method used in these examples is specified in Loading Data Into a DataSet.

The examples in this topic use the Contact, Address, Product, SalesOrderHeader, and SalesOrderDetail tables in the AdventureWorks sample database.

The examples in this topic use the following `using` / `Imports` statements:

```
using System;
using System.Linq;
using System.Linq.Expressions;
using System.Collections.Generic;
using System.Data;
using System.Data.SqlClient;
using System.Data.Common;
using System.Globalization;
```

```
Option Explicit On

Imports System.Linq
Imports System.Linq.Expressions
Imports System.Collections.Generic
Imports System.Data
Imports System.Data.SqlClient
Imports System.Data.Common
Imports System.Globalization
```

For more information, see How to: Create a LINQ to DataSet Project In Visual Studio.

Skip

Example

This example uses the Skip method to get all but the first five contacts of the `Contact` table.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable contacts = ds.Tables["Contact"];

IEnumerable<DataRow> allButFirst5Contacts = contacts.AsEnumerable().Skip(5);

Console.WriteLine("All but first 5 contacts:");
foreach (DataRow contact in allButFirst5Contacts)
{
    Console.WriteLine("FirstName = {0} \tLastname = {1}",
        contact.Field<string>("FirstName"),
        contact.Field<string>("Lastname"));
}
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim contacts As DataTable = ds.Tables("Contact")

Dim allButFirst5Contacts = contacts.AsEnumerable().Skip(5)

Console.WriteLine("All but first 5 contacts:")

For Each contact In allButFirst5Contacts
    Console.Write("FirstName = {0} ", contact.Field(Of String)("FirstName"))
    Console.WriteLine(vbTab & " LastName = " & contact.Field(Of String)("LastName"))
Next
```

Example

This example uses the Skip method to get all but the first two addresses in Seattle.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable addresses = ds.Tables["Address"];
DataTable orders = ds.Tables["SalesOrderHeader"];

var query = (
    from address in addresses.AsEnumerable()
    from order in orders.AsEnumerable()
    where address.Field<int>("AddressID") == order.Field<int>("BillToAddressID")
         && address.Field<string>("City") == "Seattle"
    select new
    {
        City = address.Field<string>("City"),
        OrderID = order.Field<int>("SalesOrderID"),
        OrderDate = order.Field<DateTime>("OrderDate")
    }).Skip(2);

Console.WriteLine("All but first 2 orders in Seattle:");
foreach (var order in query)
{
    Console.WriteLine("City: {0} Order ID: {1} Total Due: {2:d}",
        order.City, order.OrderID, order.OrderDate);
}
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim addresses As DataTable = ds.Tables("Address")
Dim orders As DataTable = ds.Tables("SalesOrderHeader")

Dim query = ( _
    From address In addresses.AsEnumerable() _
    From order In orders.AsEnumerable() _
    Where (address.Field(Of Integer)("AddressID") = _
           order.Field(Of Integer)("BillToAddressID")) _
         And address.Field(Of String)("City") = "Seattle" _
    Select New With _
       { _
           .City = address.Field(Of String)("City"), _
           .OrderID = order.Field(Of Integer)("SalesOrderID"), _
           .OrderDate = order.Field(Of DateTime)("OrderDate") _
       }).Skip(2)

Console.WriteLine("All but first 2 orders in Seattle:")
For Each addOrder In query
    Console.Write("City: " & addOrder.City)
    Console.Write(" Order ID: " & addOrder.OrderID)
    Console.WriteLine(" Order date: " & addOrder.OrderDate)
Next
```

SkipWhile

Example

This example uses OrderBy and SkipWhile methods to return products from the `Product` table with a list price greater than 300.00.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable products = ds.Tables["Product"];

IEnumerable<DataRow> skipWhilePriceLessThan300 =
    products.AsEnumerable()
        .OrderBy(listprice => listprice.Field<decimal>("ListPrice"))
        .SkipWhile(product => product.Field<decimal>("ListPrice") < 300.00M);

Console.WriteLine("Skip while ListPrice is less than 300.00:");
foreach (DataRow product in skipWhilePriceLessThan300)
{
    Console.WriteLine(product.Field<decimal>("ListPrice"));
}
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim products As DataTable = ds.Tables("Product")
Dim skipWhilePriceLessThan300 As IEnumerable(Of DataRow) = _
    products.AsEnumerable() _
        .OrderBy(Function(listprice) listprice.Field(Of Decimal)("ListPrice")) _
        .SkipWhile(Function(product) product.Field(Of Decimal)("ListPrice") < 300D)

Console.WriteLine("First ListPrice less than 300.00:")
For Each product As DataRow In skipWhilePriceLessThan300
    Console.WriteLine(product.Field(Of Decimal)("ListPrice"))
Next
```

Take

Example

This example uses the Take method to get only the first five contacts from the `Contact` table.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable contacts = ds.Tables["Contact"];

IEnumerable<DataRow> first5Contacts = contacts.AsEnumerable().Take(5);

Console.WriteLine("First 5 contacts:");
foreach (DataRow contact in first5Contacts)
{
    Console.WriteLine("Title = {0} \t FirstName = {1} \t Lastname = {2}",
        contact.Field<string>("Title"),
        contact.Field<string>("FirstName"),
        contact.Field<string>("Lastname"));
}
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim contacts As DataTable = ds.Tables("Contact")

Dim first5Contacts = contacts.AsEnumerable().Take(5)

Console.WriteLine("First 5 contacts:")
For Each contact In first5Contacts
    Console.Write("Title = " & contact.Field(Of String)("Title"))
    Console.Write(vbTab & "FirstName = " & contact.Field(Of String)("FirstName"))
    Console.WriteLine(vbTab & "LastName = " & contact.Field(Of String)("LastName"))
Next
```

Example

This example uses the Take method to get the first three addresses in Seattle.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable addresses = ds.Tables["Address"];
DataTable orders = ds.Tables["SalesOrderHeader"];

var query = (
    from address in addresses.AsEnumerable()
    from order in orders.AsEnumerable()
    where address.Field<int>("AddressID") == order.Field<int>("BillToAddressID")
         && address.Field<string>("City") == "Seattle"
    select new
    {
        City = address.Field<string>("City"),
        OrderID = order.Field<int>("SalesOrderID"),
        OrderDate = order.Field<DateTime>("OrderDate")
    }).Take(3);

Console.WriteLine("First 3 orders in Seattle:");
foreach (var order in query)
{
    Console.WriteLine("City: {0} Order ID: {1} Total Due: {2:d}",
        order.City, order.OrderID, order.OrderDate);
}
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim addresses As DataTable = ds.Tables("Address")
Dim orders As DataTable = ds.Tables("SalesOrderHeader")

Dim query = ( _
    From address In addresses.AsEnumerable() _
    From order In orders.AsEnumerable() _
    Where (address.Field(Of Integer)("AddressID") = _
        order.Field(Of Integer)("BillToAddressID")) _
         And address.Field(Of String)("City") = "Seattle" _
    Select New With _
    { _
        .City = address.Field(Of String)("City"), _
        .OrderID = order.Field(Of Integer)("SalesOrderID"), _
        .OrderDate = order.Field(Of DateTime)("OrderDate") _
    }).Take(3)

Console.WriteLine("First 3 orders in Seattle:")
For Each order In query
    Console.Write("City: " & order.City)
    Console.Write(" Order ID: " & order.OrderID)
    Console.WriteLine(" Order date: " & order.OrderDate)
Next
```

TakeWhile

Example

This example uses OrderBy and TakeWhile to return products from the `Product` table with a list price less than 300.00.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable products = ds.Tables["Product"];

IEnumerable<DataRow> takeWhileListPriceLessThan300 =
    products.AsEnumerable()
        .OrderBy(listprice => listprice.Field<decimal>("ListPrice"))
        .TakeWhile(product => product.Field<decimal>("ListPrice") < 300.00M);

Console.WriteLine("First ListPrice less than 300:");
foreach (DataRow product in takeWhileListPriceLessThan300)
{
    Console.WriteLine(product.Field<decimal>("ListPrice"));
}
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim products As DataTable = ds.Tables("Product")

Dim takeWhileListPriceLessThan300 As IEnumerable(Of DataRow) = _
    products.AsEnumerable() _
        .OrderBy(Function(listprice) listprice.Field(Of Decimal)("ListPrice")) _
        .TakeWhile(Function(product) product.Field(Of Decimal)("ListPrice") < 300D)

Console.WriteLine("First ListPrice less than 300.00:")
For Each product As DataRow In takeWhileListPriceLessThan300
    Console.WriteLine(product.Field(Of Decimal)("ListPrice"))
Next
```

See also

- Loading Data Into a DataSet
- LINQ to DataSet Examples
- Standard Query Operators Overview (C#)
- Standard Query Operators Overview (Visual Basic)

Method-Based Query Syntax Examples: Ordering (LINQ to DataSet)

9/7/2019 • 2 minutes to read • Edit Online

The examples in this topic demonstrate how to use the OrderBy, Reverse, and ThenBy methods to query a DataSet and order the results using the method query syntax.

The `FillDataSet` method used in these examples is specified in Loading Data Into a DataSet.

The examples in this topic use the Contact, Address, Product, SalesOrderHeader, and SalesOrderDetail tables in the AdventureWorks sample database.

The examples in this topic use the following `using` / `Imports` statements:

```
using System;
using System.Linq;
using System.Linq.Expressions;
using System.Collections.Generic;
using System.Data;
using System.Data.SqlClient;
using System.Data.Common;
using System.Globalization;
```

```
Option Explicit On

Imports System.Linq
Imports System.Linq.Expressions
Imports System.Collections.Generic
Imports System.Data
Imports System.Data.SqlClient
Imports System.Data.Common
Imports System.Globalization
```

For more information, see How to: Create a LINQ to DataSet Project In Visual Studio.

OrderBy

Example

This example uses the OrderBy method with a custom comparer to do a case-insensitive sort of last names.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable contacts = ds.Tables["Contact"];

IEnumerable<DataRow> query =
    contacts.AsEnumerable().OrderBy(contact => contact.Field<string>("LastName"),
                                    new CaseInsensitiveComparer());

foreach (DataRow contact in query)
{
    Console.WriteLine(contact.Field<string>("LastName"));
}
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim contacts As DataTable = ds.Tables("Contact")

Dim query As IEnumerable(Of DataRow) = _
    contacts.AsEnumerable().OrderBy(Function(contact) _
                contact.Field(Of String)("LastName"), _
                New CaseInsensitiveComparer())

For Each contact As DataRow In query
    Console.WriteLine(contact.Field(Of String)("LastName"))
Next
```

Reverse

Example

This example uses the Reverse method to create a list of orders where `OrderDate` is earlier than Feb 20, 2002.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable orders = ds.Tables["SalesOrderHeader"];

IEnumerable<DataRow> query = (
    from order in orders.AsEnumerable()
    where order.Field<DateTime>("OrderDate") < new DateTime(2002, 02, 20)
    select order).Reverse();

Console.WriteLine("A backwards list of orders where OrderDate < Feb 20, 2002");
foreach (DataRow order in query)
{
    Console.WriteLine(order.Field<DateTime>("OrderDate"));
}
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim orders As DataTable = ds.Tables("SalesOrderHeader")

Dim query = ( _
    From order In orders.AsEnumerable() _
    Where order.Field(Of DateTime)("OrderDate") < New DateTime(2002, 2, 20) _
    Select order).Reverse()

Console.WriteLine("A backwards list of orders where OrderDate < Feb 20, 2002")

For Each order In query
    Console.WriteLine(order.Field(Of DateTime)("OrderDate"))
Next
```

ThenBy

Example

This example uses OrderBy and ThenBy methods with a custom comparer to first sort by list price, and then perform a case-insensitive descending sort of the product names.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable products = ds.Tables["Product"];

IEnumerable<DataRow> query =
    products.AsEnumerable().OrderBy(product => product.Field<Decimal>("ListPrice"))
    .ThenBy(product => product.Field<string>("Name"),
            new CaseInsensitiveComparer());

foreach (DataRow product in query)
{
    Console.WriteLine("Product ID: {0} Product Name: {1} List Price {2}",
        product.Field<int>("ProductID"),
        product.Field<string>("Name"),
        product.Field<Decimal>("ListPrice"));
}
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim products As DataTable = ds.Tables("Product")

Dim query As IEnumerable(Of DataRow) = _
    products.AsEnumerable().OrderBy(Function(product) product.Field(Of Decimal)("ListPrice")) _
        .ThenBy(Function(product) product.Field(Of String)("Name"), _
                New CaseInsensitiveComparer())

For Each product As DataRow In query
    Console.WriteLine("Product ID: {0} Product Name: {1} List Price {2}", _
        product.Field(Of Integer)("ProductID"), _
        product.Field(Of String)("Name"), _
        product.Field(Of Decimal)("ListPrice"))
Next
```

See also

- Loading Data Into a DataSet
- LINQ to DataSet Examples
- Standard Query Operators Overview (C#)
- Standard Query Operators Overview (Visual Basic)

Method-Based Query Syntax Examples: Set Operators (LINQ to DataSet)

9/7/2019 • 5 minutes to read • Edit Online

The examples in this topic demonstrate how to use the Distinct, Except, Intersect, and Union operators to perform value-based comparison operations on sets of data rows.Loading Data Into a DataSet See Comparing DataRows for more information on DataRowComparer.

The `FillDataSet` method used in these examples is specified in Loading Data Into a DataSet.

The examples in this topic use the Contact, Address, Product, SalesOrderHeader, and SalesOrderDetail tables in the AdventureWorks sample database.

The examples in this topic use the following `using` / `Imports` statements:

```
using System;
using System.Linq;
using System.Linq.Expressions;
using System.Collections.Generic;
using System.Data;
using System.Data.SqlClient;
using System.Data.Common;
using System.Globalization;
```

```
Option Explicit On

Imports System.Linq
Imports System.Linq.Expressions
Imports System.Collections.Generic
Imports System.Data
Imports System.Data.SqlClient
Imports System.Data.Common
Imports System.Globalization
```

For more information, see How to: Create a LINQ to DataSet Project In Visual Studio.

Distinct

Example

This example uses the Distinct method to remove duplicate elements in a sequence.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

List<DataRow> rows = new List<DataRow>();

DataTable contact = ds.Tables["Contact"];

// Get 100 rows from the Contact table.
IEnumerable<DataRow> query = (from c in contact.AsEnumerable()
                              select c).Take(100);

DataTable contactsTableWith100Rows = query.CopyToDataTable();

// Add 100 rows to the list.
foreach (DataRow row in contactsTableWith100Rows.Rows)
    rows.Add(row);

// Create duplicate rows by adding the same 100 rows to the list.
foreach (DataRow row in contactsTableWith100Rows.Rows)
    rows.Add(row);

DataTable table =
    System.Data.DataTableExtensions.CopyToDataTable<DataRow>(rows);

// Find the unique contacts in the table.
IEnumerable<DataRow> uniqueContacts =
    table.AsEnumerable().Distinct(DataRowComparer.Default);

Console.WriteLine("Unique contacts:");
foreach (DataRow uniqueContact in uniqueContacts)
{
    Console.WriteLine(uniqueContact.Field<Int32>("ContactID"));
}
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim rows As List(Of DataRow) = New List(Of DataRow)

Dim contacts As DataTable = ds.Tables("Contact")

' Get 100 rows from the Contact table.
Dim query = ( _
    From c In contacts.AsEnumerable() _
    Select c).Take(100)

Dim contactsTableWith100Rows = query.CopyToDataTable()

' Add 100 rows to the list.
For Each row In contactsTableWith100Rows.Rows
    rows.Add(row)
Next

' Create duplicate rows by adding the same 100 rows to the list.
For Each row In contactsTableWith100Rows.Rows
    rows.Add(row)
Next

Dim table = _
        System.Data.DataTableExtensions.CopyToDataTable(Of DataRow)(rows)

' Find the unique contacts in the table.
Dim uniqueContacts = _
    table.AsEnumerable().Distinct(DataRowComparer.Default)

Console.WriteLine("Unique contacts:")
For Each uniqueContact In uniqueContacts
    Console.WriteLine(uniqueContact.Field(Of Integer)("ContactID"))
Next
```

Except

Example

This example uses the Except method to return contacts that appear in the first table but not in the second.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable contactTable = ds.Tables["Contact"];

// Create two tables.
IEnumerable<DataRow> query1 = from contact in contactTable.AsEnumerable()
                              where contact.Field<string>("Title") == "Ms."
                              select contact;

IEnumerable<DataRow> query2 = from contact in contactTable.AsEnumerable()
                              where contact.Field<string>("FirstName") == "Sandra"
                              select contact;

DataTable contacts1 = query1.CopyToDataTable();
DataTable contacts2 = query2.CopyToDataTable();

// Find the contacts that are in the first
// table but not the second.
var contacts = contacts1.AsEnumerable().Except(contacts2.AsEnumerable(),
                                               DataRowComparer.Default);

Console.WriteLine("Except of employees tables");
foreach (DataRow row in contacts)
{
    Console.WriteLine("Id: {0} {1} {2} {3}",
        row["ContactID"], row["Title"], row["FirstName"], row["LastName"]);
}
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim contactTable As DataTable = ds.Tables("Contact")

Dim query1 = _
    From contact In contactTable.AsEnumerable() _
    Where contact.Field(Of String)("Title") = "Ms." _
    Select contact

Dim query2 = _
    From contact In contactTable.AsEnumerable() _
    Where contact.Field(Of String)("FirstName") = "Sandra" _
    Select contact

Dim contacts1 = query1.CopyToDataTable()
Dim contacts2 = query2.CopyToDataTable()

' Find the contacts that are in the first
' table but not the second.
Dim contacts = contacts1.AsEnumerable().Except(contacts2.AsEnumerable(), _
                                               DataRowComparer.Default)

Console.WriteLine("Except of employees tables")

For Each row In contacts
    Console.WriteLine("Id: {0} {1} {2} {3}", _
            row("ContactID"), row("Title"), row("FirstName"), row("LastName"))
Next
```

Intersect

Example

This example uses the Intersect method to return contacts that appear in both tables.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable contactTable = ds.Tables["Contact"];

// Create two tables.
IEnumerable<DataRow> query1 = from contact in contactTable.AsEnumerable()
                              where contact.Field<string>("Title") == "Ms."
                              select contact;

IEnumerable<DataRow> query2 = from contact in contactTable.AsEnumerable()
                              where contact.Field<string>("FirstName") == "Sandra"
                              select contact;

DataTable contacts1 = query1.CopyToDataTable();
DataTable contacts2 = query2.CopyToDataTable();

// Find the intersection of the two tables.
var contacts = contacts1.AsEnumerable().Intersect(contacts2.AsEnumerable(),
                                                  DataRowComparer.Default);

Console.WriteLine("Intersection of contacts tables");
foreach (DataRow row in contacts)
{
    Console.WriteLine("Id: {0} {1} {2} {3}",
        row["ContactID"], row["Title"], row["FirstName"], row["LastName"]);
}
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim contactTable As DataTable = ds.Tables("Contact")

Dim query1 = _
    From contact In contactTable.AsEnumerable() _
    Where contact.Field(Of String)("Title") = "Ms." _
    Select contact

Dim query2 = _
    From contact In contactTable.AsEnumerable() _
    Where contact.Field(Of String)("FirstName") = "Sandra" _
    Select contact

Dim contacts1 = query1.CopyToDataTable()
Dim contacts2 = query2.CopyToDataTable()

Dim contacts = contacts1.AsEnumerable() _
    .Intersect(contacts2.AsEnumerable(), DataRowComparer.Default)

Console.WriteLine("Intersect of employees tables")

For Each row In contacts
    Console.WriteLine("Id: {0} {1} {2} {3}", _
            row("ContactID"), row("Title"), row("FirstName"), row("LastName"))
Next
```

Union

Example

This example uses the Union method to return unique contacts from either of the two tables.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

// Create two tables.
DataTable contactTable = ds.Tables["Contact"];
IEnumerable<DataRow> query1 = from contact in contactTable.AsEnumerable()
                              where contact.Field<string>("Title") == "Ms."
                              select contact;

IEnumerable<DataRow> query2 = from contact in contactTable.AsEnumerable()
                              where contact.Field<string>("FirstName") == "Sandra"
                              select contact;

DataTable contacts1 = query1.CopyToDataTable();
DataTable contacts2 = query2.CopyToDataTable();

// Find the union of the two tables.
var contacts = contacts1.AsEnumerable().Union(contacts2.AsEnumerable(),
                                              DataRowComparer.Default);

Console.WriteLine("Union of contacts tables:");
foreach (DataRow row in contacts)
{
    Console.WriteLine("Id: {0} {1} {2} {3}",
        row["ContactID"], row["Title"], row["FirstName"], row["LastName"]);
}
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim contactTable As DataTable = ds.Tables("Contact")

Dim query1 = _
    From contact In contactTable.AsEnumerable() _
    Where contact.Field(Of String)("Title") = "Ms." _
    Select contact

Dim query2 = _
    From contact In contactTable.AsEnumerable() _
    Where contact.Field(Of String)("FirstName") = "Sandra" _
    Select contact

Dim contacts1 = query1.CopyToDataTable()
Dim contacts2 = query2.CopyToDataTable()

Dim contacts = contacts1.AsEnumerable().Union(contacts2.AsEnumerable(), _
                                              DataRowComparer.Default)

Console.WriteLine("Union of employees tables")
For Each row In contacts
    Console.WriteLine("Id: {0} {1} {2} {3}", _
            row("ContactID"), row("Title"), row("FirstName"), row("LastName"))
Next
```

See also

- Loading Data Into a DataSet
- LINQ to DataSet Examples

- Standard Query Operators Overview (C#)
- Standard Query Operators Overview (Visual Basic)

Method-Based Query Syntax Examples: Conversion Operators (LINQ to DataSet)

9/7/2019 • 2 minutes to read • Edit Online

The examples in this topic demonstrate how to use the ToArray, ToDictionary, and ToList methods to immediately execute a query expression.

The `FillDataSet` method used in these examples is specified in Loading Data Into a DataSet.

The examples in this topic use the Contact, Address, Product, SalesOrderHeader, and SalesOrderDetail tables in the AdventureWorks sample database.

The examples in this topic use the following `using` / `Imports` statements:

```
using System;
using System.Linq;
using System.Linq.Expressions;
using System.Collections.Generic;
using System.Data;
using System.Data.SqlClient;
using System.Data.Common;
using System.Globalization;
```

```
Option Explicit On

Imports System.Linq
Imports System.Linq.Expressions
Imports System.Collections.Generic
Imports System.Data
Imports System.Data.SqlClient
Imports System.Data.Common
Imports System.Globalization
```

For more information, see How to: Create a LINQ to DataSet Project In Visual Studio.

ToArray

Example

This example uses the ToArray method to immediately evaluate a sequence into an array.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable products = ds.Tables["Product"];

IEnumerable<DataRow> productsArray = products.AsEnumerable().ToArray();

IEnumerable<DataRow> query =
    from product in productsArray
    orderby product.Field<Decimal>("ListPrice") descending
    select product;

Console.WriteLine("Every price from highest to lowest:");
foreach (DataRow product in query)
{
    Console.WriteLine(product.Field<Decimal>("ListPrice"));
}
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim products As DataTable = ds.Tables("Product")

Dim productsArray = products.AsEnumerable().ToArray()

Dim query = _
        From product In productsArray _
        Select product _
        Order By product.Field(Of Decimal)("ListPrice") Descending

Console.WriteLine("Every price From highest to lowest:")
For Each product In query
    Console.WriteLine(product.Field(Of Decimal)("ListPrice"))
Next
```

ToDictionary

Example

This example uses the ToDictionary method to immediately evaluate a sequence and a related key expression into a dictionary.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable products = ds.Tables["Product"];

var scoreRecordsDict = products.AsEnumerable().
    ToDictionary(record => record.Field<string>("Name"));
Console.WriteLine("Top Tube's ProductID: {0}",
    scoreRecordsDict["Top Tube"]["ProductID"]);
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim products As DataTable = ds.Tables("Product")

Dim scoreRecordsDict = products.AsEnumerable(). _
    ToDictionary(Function(record) record.Field(Of String)("Name"))

Console.WriteLine("Top Tube's ProductID: {0}", _
    scoreRecordsDict("Top Tube")("ProductID"))
```

ToList

Example

This example uses the ToList method to immediately evaluate a sequence into a List<T>, where `T` is of type DataRow.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable products = ds.Tables["Product"];

IEnumerable<DataRow> productList = products.AsEnumerable().ToList();

IEnumerable<DataRow> query =
    from product in productList
    orderby product.Field<string>("Name")
    select product;

Console.WriteLine("The product list, ordered by product name:");
foreach (DataRow product in query)
{
    Console.WriteLine(product.Field<string>("Name").ToLower(CultureInfo.InvariantCulture));
}
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim products As DataTable = ds.Tables("Product")

Dim productList = products.AsEnumerable().ToList()

Dim query = _
    From product In productList _
    Select product _
    Order By product.Field(Of String)("Name")

Console.WriteLine("The sorted name list:")

For Each product In query
    Console.WriteLine(product.Field(Of String)("Name").ToLower(CultureInfo.InvariantCulture))
Next
```

See also

- Loading Data Into a DataSet
- LINQ to DataSet Examples
- Standard Query Operators Overview (C#)
- Standard Query Operators Overview (Visual Basic)

Method-Based Query Syntax Examples: Element Operators (LINQ to DataSet)

3/12/2020 • 2 minutes to read • Edit Online

The examples in this topic demonstrate how to use the First and ElementAt methods to get DataRow elements from a DataSet using the query expression syntax.

The `FillDataSet` method used in these examples is specified in Loading Data Into a DataSet.

The examples in this topic use the Contact, Address, Product, SalesOrderHeader, and SalesOrderDetail tables in the AdventureWorks sample database.

The examples in this topic use the following `using` / `Imports` statements:

```
using System;
using System.Linq;
using System.Linq.Expressions;
using System.Collections.Generic;
using System.Data;
using System.Data.SqlClient;
using System.Data.Common;
using System.Globalization;
```

```
Option Explicit On

Imports System.Linq
Imports System.Linq.Expressions
Imports System.Collections.Generic
Imports System.Data
Imports System.Data.SqlClient
Imports System.Data.Common
Imports System.Globalization
```

For more information, see How to: Create a LINQ to DataSet Project In Visual Studio.

ElementAt

Example

This example uses the ElementAt method to retrieve the fifth address where `PostalCode` == "M4B 1V7".

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable addresses = ds.Tables["Address"];

var fifthAddress = (
    from address in addresses.AsEnumerable()
    where address.Field<string>("PostalCode") == "M4B 1V7"
    select address.Field<string>("AddressLine1"))
.ElementAt(5);

Console.WriteLine("Fifth address where PostalCode = 'M4B 1V7': {0}",
    fifthAddress);
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim addresses As DataTable = ds.Tables("Address")

Dim fifthAddress = ( _
    From address In addresses.AsEnumerable() _
    Where address.Field(Of String)("PostalCode") = "M4B 1V7" _
    Select address.Field(Of String)("AddressLine1")).ElementAt(5)

Console.WriteLine("Fifth address where PostalCode = 'M4B 1V7': " & _
        fifthAddress)
```

First

Example

This example uses the First method to return the first contact whose first name is 'Brooke'.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable contacts = ds.Tables["Contact"];

DataRow query = (
    from contact in contacts.AsEnumerable()
    where (string)contact["FirstName"] == "Brooke"
    select contact)
    .First();

Console.WriteLine("ContactID: " + query.Field<int>("ContactID"));
Console.WriteLine("FirstName: " + query.Field<string>("FirstName"));
Console.WriteLine("LastName: " + query.Field<string>("LastName"));
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim contacts As DataTable = ds.Tables("Contact")

Dim query = ( _
    From contact In contacts.AsEnumerable() _
    Where contact.Field(Of String)("FirstName") = "Brooke" _
    Select contact).First()

Console.WriteLine("ContactID: " & query.Field(Of Integer)("ContactID"))
Console.WriteLine("FirstName: " & query.Field(Of String)("FirstName"))
Console.WriteLine("LastName: " & query.Field(Of String)("LastName"))
```

See also

- Loading Data Into a DataSet
- LINQ to DataSet Examples
- Standard Query Operators Overview (C#)
- Standard Query Operators Overview (Visual Basic)

Method-Based Query Syntax Examples: Aggregate Operators (LINQ to DataSet)

9/7/2019 • 12 minutes to read • Edit Online

The examples in this topic demonstrate how to use the Aggregate, Average, Count, LongCount, Max, Min, and Sum operators to query a DataSet and aggregate data using method query syntax.

The `FillDataSet` method used in these examples is specified in Loading Data Into a DataSet.

The examples in this topic use the Contact, Address, Product, SalesOrderHeader, and SalesOrderDetail tables in the AdventureWorks sample database.

The examples in this topic use the following `using` / `Imports` statements:

```
using System;
using System.Linq;
using System.Linq.Expressions;
using System.Collections.Generic;
using System.Data;
using System.Data.SqlClient;
using System.Data.Common;
using System.Globalization;
```

```
Option Explicit On

Imports System.Linq
Imports System.Linq.Expressions
Imports System.Collections.Generic
Imports System.Data
Imports System.Data.SqlClient
Imports System.Data.Common
Imports System.Globalization
```

For more information, see How to: Create a LINQ to DataSet Project In Visual Studio.

Aggregate

Example

This example uses the Aggregate method to get the first 5 contacts from the `Contact` table and build a comma-delimited list of the last names.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

IEnumerable<DataRow> contacts = ds.Tables["Contact"].AsEnumerable();

string nameList =
    contacts.Take(5).Select(contact => contact.Field<string>("LastName")).Aggregate((workingList, next) =>
workingList + "," + next);

Console.WriteLine(nameList);
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim contacts As IEnumerable(Of DataRow) = _
    ds.Tables("Contact").AsEnumerable()

Dim nameList As String = _
        contacts.Take(5).Select(Function(contact) contact.Field(Of String)("LastName")). _
            Aggregate(Function(workingList, next1) workingList + "," + next1)

Console.WriteLine(nameList)
```

Average

Example

This example uses the Average method to find the average list price of the products.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

var products = ds.Tables["Product"].AsEnumerable();

Decimal averageListPrice =
    products.Average(product => product.Field<Decimal>("ListPrice"));

Console.WriteLine("The average list price of all the products is ${0}",
    averageListPrice);
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim products As IEnumerable(Of DataRow) = _
    ds.Tables("Product").AsEnumerable()

Dim averageListPrice As Decimal = _
    products.Average(Function(product) product. _
            Field(Of Decimal)("ListPrice"))

Console.WriteLine("The average list price of all the products is $" & _
    averageListPrice)
```

Example

This example uses the Average method to find the average list price of the products of each style.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

var products = ds.Tables["Product"].AsEnumerable();

var query = from product in products
            group product by product.Field<string>("Style") into g
            select new
            {
                Style = g.Key,
                AverageListPrice =
                    g.Average(product => product.Field<Decimal>("ListPrice"))
            };

foreach (var product in query)
{
    Console.WriteLine("Product style: {0} Average list price: {1}",
        product.Style, product.AverageListPrice);
}
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim products As IEnumerable(Of DataRow) = _
    ds.Tables("Product").AsEnumerable()

Dim query = _
    From product In products _
    Group product By style = product.Field(Of String)("Style") Into g = Group _
    Select New With _
        { _
            .Style = style, _
            .AverageListPrice = g.Average(Function(product) _
                    product.Field(Of Decimal)("ListPrice")) _
        }

For Each product In query
    Console.WriteLine("Product style: {0} Average list price: {1}", _
        product.Style, product.AverageListPrice)
Next
```

Example

This example uses the Average method to find the average total due.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable orders = ds.Tables["SalesOrderHeader"];

Decimal averageTotalDue = orders.AsEnumerable().
    Average(order => order.Field<decimal>("TotalDue"));
Console.WriteLine("The average TotalDue is {0}.",
    averageTotalDue);
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim orders As DataTable = ds.Tables("SalesOrderHeader")

Dim averageTotalDue As Decimal = orders.AsEnumerable(). _
    Average(Function(order) order.Field(Of Decimal)("TotalDue"))
Console.WriteLine("The average TotalDue is {0}.", _
    averageTotalDue)
```

Example

This example uses the Average method to get the average total due for each contact ID.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable orders = ds.Tables["SalesOrderHeader"];

var query =
    from order in orders.AsEnumerable()
    group order by order.Field<Int32>("ContactID") into g
    select new
    {
        Category = g.Key,
        averageTotalDue =
            g.Average(order => order.Field<decimal>("TotalDue"))
    };

foreach (var order in query)
{
    Console.WriteLine("ContactID = {0} \t Average TotalDue = {1}",
        order.Category,
        order.averageTotalDue);
}
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim orders As DataTable = ds.Tables("SalesOrderHeader")

Dim query = _
    From order In orders.AsEnumerable() _
    Group order By contactID = order.Field(Of Int32)("ContactID") Into g = Group _
    Select New With _
    { _
        .Category = contactID, _
        .averageTotalDue = g.Average(Function(order) order. _
                Field(Of Decimal)("TotalDue")) _
    }

For Each order In query
    Console.WriteLine("ContactID = {0} " & vbTab & _
        " Average TotalDue = {1}", order.Category, _
        order.averageTotalDue)
Next
```

Example

This example uses the Average method to get the orders with the average `TotalDue` for each contact.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable orders = ds.Tables["SalesOrderHeader"];

var query =
    from order in orders.AsEnumerable()
    group order by order.Field<Int32>("ContactID") into g
    let averageTotalDue = g.Average(order => order.Field<decimal>("TotalDue"))
    select new
    {
        Category = g.Key,
        CheapestProducts =
            g.Where(order => order.Field<decimal>("TotalDue") ==
                        averageTotalDue)
    };

foreach (var orderGroup in query)
{
    Console.WriteLine("ContactID: {0}", orderGroup.Category);
    foreach (var order in orderGroup.CheapestProducts)
    {
        Console.WriteLine("Average total due for SalesOrderID {1} is: {0}",
            order.Field<decimal>("TotalDue"),
            order.Field<Int32>("SalesOrderID"));
    }
    Console.WriteLine("");
}
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim orders As DataTable = ds.Tables("SalesOrderHeader")

Dim query = _
    From order In orders.AsEnumerable() _
    Group order By contactID = order.Field(Of Int32)("ContactID") Into g = Group _
    Let averageTotalDue = g.Average(Function(order) order.Field(Of Decimal)("TotalDue")) _
    Select New With _
    { _
        .Category = contactID, _
        .CheapestProducts = g.Where(Function(order) order. _
                Field(Of Decimal)("TotalDue") = averageTotalDue) _
    }

For Each orderGroup In query
    Console.WriteLine("ContactID: " & orderGroup.Category)
    For Each order In orderGroup.CheapestProducts
        Console.WriteLine("Average total due for SalesOrderID {1} is: {0}", _
            order.Field(Of Decimal)("TotalDue"), _
            order.Field(Of Int32)("SalesOrderID"))
    Next
    Console.WriteLine("")
Next
```

Count

Example

This example uses the Count method to return the number of products in the `Product` table.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

var products = ds.Tables["Product"].AsEnumerable();

int numProducts = products.Count();

Console.WriteLine("There are {0} products.", numProducts);
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim products As DataTable = ds.Tables("Product")

Dim numProducts = products.AsEnumerable().Count()

Console.WriteLine("There are " & numProducts & " products.")
```

Example

This example uses the Count method to return a list of contact IDs and how many orders each has.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable contacts = ds.Tables["Contact"];

var query = from contact in contacts.AsEnumerable()
            select new
            {
                CustomerID = contact.Field<int>("ContactID"),
                OrderCount =
                    contact.GetChildRows("SalesOrderContact").Count()
            };

foreach (var contact in query)
{
    Console.WriteLine("CustomerID = {0} \t OrderCount = {1}",
        contact.CustomerID,
        contact.OrderCount);
}
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim contacts As DataTable = ds.Tables("Contact")

Dim query = _
    From contact In contacts.AsEnumerable() _
    Select New With _
    { _
        .ContactID = contact.Field(Of Integer)("ContactID"), _
        .OrderCount = contact.GetChildRows("SalesOrderContact").Count() _
    }

For Each contact In query
    Console.Write("CustomerID = " & contact.ContactID)
    Console.WriteLine(vbTab & "OrderCount = " & contact.OrderCount)
Next
```

Example

This example groups products by color and uses the Count method to return the number of products in each color group.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable products = ds.Tables["Product"];

var query =
    from product in products.AsEnumerable()
    group product by product.Field<string>("Color") into g
    select new { Color = g.Key, ProductCount = g.Count() };

foreach (var product in query)
{
    Console.WriteLine("Color = {0} \t ProductCount = {1}",
        product.Color,
        product.ProductCount);
}
```

```
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim products As DataTable = ds.Tables("Product")

Dim query = _
    From product In products.AsEnumerable() _
    Group product By color = product.Field(Of String)("Color") Into g = Group _
    Select New With {.Color = color, .ProductCount = g.Count()}

For Each product In query
    Console.WriteLine("Color = {0} " & vbTab & "ProductCount = {1}", _
        product.Color, _
        product.ProductCount)
Next
```

LongCount

Example

This example gets the contact count as a long integer.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable contacts = ds.Tables["Contact"];

long numberOfContacts = contacts.AsEnumerable().LongCount();
Console.WriteLine("There are {0} Contacts", numberOfContacts);
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim contacts As DataTable = ds.Tables("Contact")

Dim numberOfContacts = contacts.AsEnumerable().LongCount()

Console.WriteLine("There are {0} Contacts", numberOfContacts)
```

Max

Example

This example uses the Max method to get the largest total due.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable orders = ds.Tables["SalesOrderHeader"];

Decimal maxTotalDue = orders.AsEnumerable().
    Max(w => w.Field<decimal>("TotalDue"));
Console.WriteLine("The maximum TotalDue is {0}.",
    maxTotalDue);
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim orders As DataTable = ds.Tables("SalesOrderHeader")

Dim maxTotalDue As Decimal = orders.AsEnumerable(). _
    Max(Function(w) w.Field(Of Decimal)("TotalDue"))
Console.WriteLine("The maximum TotalDue is {0}.", maxTotalDue)
```

Example

This example uses the Max method to get the largest total due for each contact ID.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable orders = ds.Tables["SalesOrderHeader"];

var query =
    from order in orders.AsEnumerable()
    group order by order.Field<Int32>("ContactID") into g
    select new
    {
        Category = g.Key,
        maxTotalDue =
            g.Max(order => order.Field<decimal>("TotalDue"))
    };

foreach (var order in query)
{
    Console.WriteLine("ContactID = {0} \t Maximum TotalDue = {1}",
        order.Category, order.maxTotalDue);
}
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim orders As DataTable = ds.Tables("SalesOrderHeader")

Dim query = _
   From order In orders.AsEnumerable() _
   Group order By contactID = order.Field(Of Int32)("ContactID") Into g = Group _
   Select New With _
   { _
       .Category = contactID, _
       .maxTotalDue = _
           g.Max(Function(order) order.Field(Of Decimal)("TotalDue")) _
   }
For Each order In query
    Console.WriteLine("ContactID = {0} " & vbTab & _
        " Maximum TotalDue = {1}", _
        order.Category, order.maxTotalDue)
Next
```

Example

This example uses the Max method to get the orders with the largest `TotalDue` for each contact ID.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable orders = ds.Tables["SalesOrderHeader"];

var query =
    from order in orders.AsEnumerable()
    group order by order.Field<Int32>("ContactID") into g
    let maxTotalDue = g.Max(order => order.Field<decimal>("TotalDue"))
    select new
    {
        Category = g.Key,
        CheapestProducts =
            g.Where(order => order.Field<decimal>("TotalDue") ==
                        maxTotalDue)
    };

foreach (var orderGroup in query)
{
    Console.WriteLine("ContactID: {0}", orderGroup.Category);
    foreach (var order in orderGroup.CheapestProducts)
    {
        Console.WriteLine("MaxTotalDue {0} for SalesOrderID {1}: ",
            order.Field<decimal>("TotalDue"),
            order.Field<Int32>("SalesOrderID"));
    }
}
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim orders As DataTable = ds.Tables("SalesOrderHeader")

Dim query = _
    From order In orders.AsEnumerable() _
    Group order By contactID = order.Field(Of Int32)("ContactID") Into g = Group _
    Let maxTotalDue = g.Max(Function(order) order.Field(Of Decimal)("TotalDue")) _
    Select New With _
    { _
        .Category = contactID, _
        .CheapestProducts = _
            g.Where(Function(order) order. _
                    Field(Of Decimal)("TotalDue") = maxTotalDue) _
    }

For Each orderGroup In query
    Console.WriteLine("ContactID: " & orderGroup.Category)
    For Each order In orderGroup.CheapestProducts
        Console.WriteLine("MaxTotalDue {0} for SalesOrderID {1} ", _
            order.Field(Of Decimal)("TotalDue"), _
            order.Field(Of Int32)("SalesOrderID"))
    Next
Next
```

Min

Example

This example uses the Min method to get the smallest total due.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable orders = ds.Tables["SalesOrderHeader"];

Decimal smallestTotalDue = orders.AsEnumerable().
    Min(totalDue => totalDue.Field<decimal>("TotalDue"));
Console.WriteLine("The smallest TotalDue is {0}.",
    smallestTotalDue);
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim orders As DataTable = ds.Tables("SalesOrderHeader")

Dim smallestTotalDue As Decimal = orders.AsEnumerable(). _
    Min(Function(totalDue) totalDue.Field(Of Decimal)("TotalDue"))

Console.WriteLine("The smallest TotalDue is {0}.", smallestTotalDue)
```

Example

This example uses the Min method to get the smallest total due for each contact ID.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable orders = ds.Tables["SalesOrderHeader"];

var query =
    from order in orders.AsEnumerable()
    group order by order.Field<Int32>("ContactID") into g
    select new
    {
        Category = g.Key,
        smallestTotalDue =
            g.Min(order => order.Field<decimal>("TotalDue"))
    };

foreach (var order in query)
{
    Console.WriteLine("ContactID = {0} \t Minimum TotalDue = {1}",
        order.Category, order.smallestTotalDue);
}
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim orders As DataTable = ds.Tables("SalesOrderHeader")

Dim query = _
    From order In orders.AsEnumerable() _
    Group order By contactID = order.Field(Of Int32)("ContactID") Into g = Group _
    Select New With _
    { _
       .Category = contactID, _
       .smallestTotalDue = g.Min(Function(order) _
                order.Field(Of Decimal)("TotalDue")) _
    }

For Each order In query
    Console.WriteLine("ContactID = {0} " & vbTab & _
        "Minimum TotalDue = {1}", order.Category, order.smallestTotalDue)
Next
```

Example

This example uses the Min method to get the orders with the smallest total due for each contact.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable orders = ds.Tables["SalesOrderHeader"];

var query =
    from order in orders.AsEnumerable()
    group order by order.Field<Int32>("ContactID") into g
    let minTotalDue = g.Min(order => order.Field<decimal>("TotalDue"))
    select new
    {
        Category = g.Key,
        smallestTotalDue =
            g.Where(order => order.Field<decimal>("TotalDue") ==
                        minTotalDue)
    };

foreach (var orderGroup in query)
{
    Console.WriteLine("ContactID: {0}", orderGroup.Category);
    foreach (var order in orderGroup.smallestTotalDue)
    {
        Console.WriteLine("Mininum TotalDue {0} for SalesOrderID {1}: ",
            order.Field<decimal>("TotalDue"),
            order.Field<Int32>("SalesOrderID"));
    }
    Console.WriteLine("");
}
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim orders As DataTable = ds.Tables("SalesOrderHeader")

Dim query = _
    From order In orders.AsEnumerable() _
    Group order By contactID = order.Field(Of Int32)("ContactID") Into g = Group _
    Let minTotalDue = g.Min(Function(order) order.Field(Of Decimal)("TotalDue")) _
    Select New With _
    { _
       .Category = contactID, _
       .smallestTotalDue = g.Where(Function(order) order. _
            Field(Of Decimal)("TotalDue") = minTotalDue) _
    }

For Each orderGroup In query
    Console.WriteLine("ContactID: " & orderGroup.Category)
    For Each order In orderGroup.smallestTotalDue
        Console.WriteLine("Mininum TotalDue {0} for SalesOrderID {1} ", _
            order.Field(Of Decimal)("TotalDue"), _
            order.Field(Of Int32)("SalesOrderID"))
    Next
    Console.WriteLine("")
Next
```

Sum

Example

This example uses the Sum method to get the total number of order quantities in the `SalesOrderDetail` table.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable orders = ds.Tables["SalesOrderDetail"];

double totalOrderQty = orders.AsEnumerable().
    Sum(o => o.Field<Int16>("OrderQty"));
Console.WriteLine("There are a total of {0} OrderQty.",
    totalOrderQty);
```

Example

This example uses the Sum method to get the total due for each contact ID.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable orders = ds.Tables["SalesOrderHeader"];

var query =
    from order in orders.AsEnumerable()
    group order by order.Field<Int32>("ContactID") into g
    select new
    {
        Category = g.Key,
        TotalDue = g.Sum(order => order.Field<decimal>("TotalDue")),
    };
foreach (var order in query)
{
    Console.WriteLine("ContactID = {0} \t TotalDue sum = {1}",
        order.Category, order.TotalDue);
}
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim orders As DataTable = ds.Tables("SalesOrderHeader")

Dim query = _
    From order In orders.AsEnumerable() _
    Group order By contactID = order.Field(Of Int32)("ContactID") Into g = Group _
    Select New With _
       { _
           .Category = contactID, _
           .TotalDue = g.Sum(Function(order) order. _
                Field(Of Decimal)("TotalDue")) _
       }

For Each order In query
    Console.WriteLine("ContactID = {0} " & vbTab & "TotalDue sum = {1}", _
        order.Category, order.TotalDue)
Next
```

See also

- Loading Data Into a DataSet
- LINQ to DataSet Examples
- Standard Query Operators Overview (C#)
- Standard Query Operators Overview (Visual Basic)

Method-Based Query Syntax Examples: Join (LINQ to DataSet)

9/7/2019 • 3 minutes to read • Edit Online

Joining is an important operation in queries that target data sources that have no navigable relationships to each other, such as relational database tables. A join of two data sources is the association of objects in one data source with objects that share a common attribute in the other data source. For more information, see Standard Query Operators Overview (C#) or Standard Query Operators Overview (Visual Basic).

The examples in this topic demonstrate how to use the Join method to query a DataSet using the method query syntax.

The `FillDataSet` method used in these examples is specified in Loading Data Into a DataSet.

The examples in this topic use the Contact, Address, Product, SalesOrderHeader, and SalesOrderDetail tables in the AdventureWorks sample database.

The examples in this topic use the following `using` / `Imports` statements:

```
using System;
using System.Linq;
using System.Linq.Expressions;
using System.Collections.Generic;
using System.Data;
using System.Data.SqlClient;
using System.Data.Common;
using System.Globalization;
```

```
Option Explicit On

Imports System.Linq
Imports System.Linq.Expressions
Imports System.Collections.Generic
Imports System.Data
Imports System.Data.SqlClient
Imports System.Data.Common
Imports System.Globalization
```

For more information, see How to: Create a LINQ to DataSet Project In Visual Studio.

Join

Example

This example performs a join over the `Contact` and `SalesOrderHeader` tables.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable contacts = ds.Tables["Contact"];
DataTable orders = ds.Tables["SalesOrderHeader"];

var query =
    contacts.AsEnumerable().Join(orders.AsEnumerable(),
    order => order.Field<Int32>("ContactID"),
    contact => contact.Field<Int32>("ContactID"),
    (contact, order) => new
    {
        ContactID = contact.Field<Int32>("ContactID"),
        SalesOrderID = order.Field<Int32>("SalesOrderID"),
        FirstName = contact.Field<string>("FirstName"),
        Lastname = contact.Field<string>("Lastname"),
        TotalDue = order.Field<decimal>("TotalDue")
    });

foreach (var contact_order in query)
{
    Console.WriteLine("ContactID: {0} "
                    + "SalesOrderID: {1} "
                    + "FirstName: {2} "
                    + "Lastname: {3} "
                    + "TotalDue: {4}",
        contact_order.ContactID,
        contact_order.SalesOrderID,
        contact_order.FirstName,
        contact_order.Lastname,
        contact_order.TotalDue);
}
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim contacts As DataTable = ds.Tables("Contact")
Dim orders As DataTable = ds.Tables("SalesOrderHeader")

Dim query = _
    contacts.AsEnumerable().Join(orders.AsEnumerable(), _
    Function(order) order.Field(Of Int32)("ContactID"), _
    Function(contact) contact.Field(Of Int32)("ContactID"), _
    Function(contact, order) New With _
        { _
            .ContactID = contact.Field(Of Int32)("ContactID"), _
            .SalesOrderID = order.Field(Of Int32)("SalesOrderID"), _
            .FirstName = contact.Field(Of String)("FirstName"), _
            .Lastname = contact.Field(Of String)("Lastname"), _
            .TotalDue = order.Field(Of Decimal)("TotalDue") _
        })

For Each contact_order In query
    Console.WriteLine("ContactID: {0} " _
                    & "SalesOrderID: {1} " _
                    & "FirstName: {2} " _
                    & "Lastname: {3} " _
                    & "TotalDue: {4}", _
        contact_order.ContactID, _
        contact_order.SalesOrderID, _
        contact_order.FirstName, _
        contact_order.Lastname, _
        contact_order.TotalDue)
Next
```

Example

This example performs a join over the `Contact` and `SalesOrderHeader` tables, grouping the results by contact ID.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable contacts = ds.Tables["Contact"];
DataTable orders = ds.Tables["SalesOrderHeader"];

var query = contacts.AsEnumerable().Join(orders.AsEnumerable(),
    order => order.Field<Int32>("ContactID"),
    contact => contact.Field<Int32>("ContactID"),
    (contact, order) => new
    {
        ContactID = contact.Field<Int32>("ContactID"),
        SalesOrderID = order.Field<Int32>("SalesOrderID"),
        FirstName = contact.Field<string>("FirstName"),
        Lastname = contact.Field<string>("Lastname"),
        TotalDue = order.Field<decimal>("TotalDue")
    })
        .GroupBy(record => record.ContactID);

foreach (var group in query)
{
    foreach (var contact_order in group)
    {
        Console.WriteLine("ContactID: {0} "
                        + "SalesOrderID: {1} "
                        + "FirstName: {2} "
                        + "Lastname: {3} "
                        + "TotalDue: {4}",
            contact_order.ContactID,
            contact_order.SalesOrderID,
            contact_order.FirstName,
            contact_order.Lastname,
            contact_order.TotalDue);
    }
}
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim contacts As DataTable = ds.Tables("Contact")
Dim orders As DataTable = ds.Tables("SalesOrderHeader")

Dim query = _
    contacts.AsEnumerable().Join(orders.AsEnumerable(), _
    Function(order) order.Field(Of Int32)("ContactID"), _
    Function(contact) contact.Field(Of Int32)("ContactID"), _
        Function(contact, order) New With _
        { _
            .ContactID = contact.Field(Of Int32)("ContactID"), _
            .SalesOrderID = order.Field(Of Int32)("SalesOrderID"), _
            .FirstName = contact.Field(Of String)("FirstName"), _
            .Lastname = contact.Field(Of String)("Lastname"), _
            .TotalDue = order.Field(Of Decimal)("TotalDue") _
        }) _
        .GroupBy(Function(record) record.ContactID)

For Each group In query
    For Each contact_order In group
        Console.WriteLine("ContactID: {0} " _
                        & "SalesOrderID: {1} " _
                        & "FirstName: {2} " _
                        & "Lastname: {3} " _
                        & "TotalDue: {4}", _
            contact_order.ContactID, _
            contact_order.SalesOrderID, _
            contact_order.FirstName, _
            contact_order.Lastname, _
            contact_order.TotalDue)
    Next
Next
```

See also

- Loading Data Into a DataSet
- LINQ to DataSet Examples
- Standard Query Operators Overview (C#)
- Standard Query Operators Overview (Visual Basic)
- Join Samples
- Dataset Samples

DataSet-Specific Operator Examples (LINQ to DataSet)

9/7/2019 • 2 minutes to read • Edit Online

The examples in this topic demonstrate how to use the CopyToDataTable method and the DataRowComparer class.

The `FillDataSet` method used in these examples is specified in Loading Data Into a DataSet.

The examples in this topic use the Contact, Address, Product, SalesOrderHeader, and SalesOrderDetail tables in the AdventureWorks sample database.

The examples in this topic use the following `using` / `Imports` statements:

```
using System;
using System.Linq;
using System.Linq.Expressions;
using System.Collections.Generic;
using System.Data;
using System.Data.SqlClient;
using System.Data.Common;
using System.Globalization;
```

```
Option Explicit On

Imports System.Linq
Imports System.Linq.Expressions
Imports System.Collections.Generic
Imports System.Data
Imports System.Data.SqlClient
Imports System.Data.Common
Imports System.Globalization
```

For more information, see How to: Create a LINQ to DataSet Project In Visual Studio.

CopyToDataTable

Example

This example loads a DataTable with query results by using the CopyToDataTable method.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

DataTable contacts1 = ds.Tables["Contact"];

IEnumerable<DataRow> query =
    from contact in contacts1.AsEnumerable()
    where contact.Field<string>("Title") == "Ms."
        && contact.Field<string>("FirstName") == "Carla"
    select contact;

DataTable contacts2 = query.CopyToDataTable();

foreach (DataRow contact in contacts2.AsEnumerable())
{
    Console.WriteLine("ID:{0} Name: {1}, {2}",
        contact.Field<Int32>("ContactID"),
        contact.Field<string>("LastName"),
        contact.Field<string>("FirstName"));
}
```

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim contactTable As DataTable = ds.Tables("Contact")

Dim query = _
    From contact In contactTable.AsEnumerable() _
    Where contact.Field(Of String)("Title") = "Ms." _
            And contact.Field(Of String)("FirstName") = "Carla" _
    Select contact

Dim contacts = query.CopyToDataTable().AsEnumerable()

For Each contact In contacts
    Console.Write("ID: " & contact.Field(Of Integer)("ContactID"))
    Console.WriteLine(" Name: " & contact.Field(Of String)("LastName") & _
                      ", " & contact.Field(Of String)("FirstName"))
Next
```

DataRowComparer

Example

This example compares two different data rows by using DataRowComparer.

```
// Fill the DataSet.
DataSet ds = new DataSet();
ds.Locale = CultureInfo.InvariantCulture;
FillDataSet(ds);

// Get two rows from the SalesOrderHeader table.
DataTable table = ds.Tables["SalesOrderHeader"];
DataRow left = (DataRow)table.Rows[0];
DataRow right = (DataRow)table.Rows[1];

// Compare the two different rows.
IEqualityComparer<DataRow> comparer = DataRowComparer.Default;

bool bEqual = comparer.Equals(left, right);
if (bEqual)
    Console.WriteLine("The two rows are equal");
else
    Console.WriteLine("The two rows are not equal");

// Get the hash codes of the two rows.
Console.WriteLine("The hashcodes for the two rows are {0}, {1}",
    comparer.GetHashCode(left),
    comparer.GetHashCode(right));
```

See also

- Loading Data Into a DataSet
- LINQ to DataSet Examples

Entity Data Model

11/7/2019 • 2 minutes to read • Edit Online

The Entity Data Model (EDM) is a set of concepts that describe the structure of data, regardless of its stored form. The EDM borrows from the Entity-Relationship Model described by Peter Chen in 1976, but it also builds on the Entity-Relationship Model and extends its traditional uses.

The EDM addresses the challenges that arise from having data stored in many forms. For example, consider a business that stores data in relational databases, text files, XML files, spreadsheets, and reports. This presents significant challenges in data modeling, application design, and data access. When designing a data-oriented application, the challenge is to write efficient and maintainable code without sacrificing efficient data access, storage, and scalability. When data has a relational structure, data access, storage, and scalability are very efficient, but writing efficient and maintainable code becomes more difficult. When data has an object structure, the trade-offs are reversed: Writing efficient and maintainable code comes at the cost of efficient data access, storage, and scalability. Even if the right balance between these trade-offs can be found, new challenges arise when data is moved from one form to another. The Entity Data Model addresses these challenges by describing the structure of data in terms of entities and relationships that are independent of any storage schema. This makes the stored form of data irrelevant to application design and development. And, because entities and relationships describe the structure of data as it is used in an application (not its stored form), they can evolve as an application evolves.

A `conceptual model` is a specific representation of the structure of data as entities and relationships, and is generally defined in a domain-specific language (DSL) that implements the concepts of the EDM. Conceptual schema definition language (CSDL) is an example of such a domain-specific language. Entities and relationships described in a conceptual model can be thought of as abstractions of objects and associations in an application. This allows developers to focus on the conceptual model without concern for the storage schema, and allows them to write code with efficiency and maintainability in mind. Meanwhile storage schema designers can focus on the efficiency of data access, storage, and scalability.

In This Section

Topics in this section describe the concepts of the Entity Data Model. Any DSL that implements the EDM should include the concepts described here. Note that the ADO.NET Entity Framework uses CSDL to define conceptual models. For more information, see CSDL Specification.

Entity Data Model Key Concepts

Entity Data Model: Namespaces

Entity Data Model: Primitive Data Types

Entity Data Model: Inheritance

association end

association end multiplicity

association set

association set end

association type

complex type

entity container

entity key

entity set

entity type

facet

foreign key property

model-declared function

model-defined function

navigation property

property

referential integrity constraint

See also

- ADO.NET Entity Data Model Tools
- .edmx File Overview
- CSDL Specification

Entity Data Model Key Concepts

11/7/2019 • 3 minutes to read • Edit Online

The Entity Data Model (EDM) uses three key concepts to describe the structure of data: *entity type*, *association type*, and *property*. These are the most important concepts in describing the structure of data in any implementation of the EDM.

Entity Type

The entity type is the fundamental building block for describing the structure of data with the Entity Data Model. In a conceptual model, entity types are constructed from properties and describe the structure of top-level concepts, such as a customers and orders in a business application. In the same way that a class definition in a computer program is a template for instances of the class, an entity type is a template for entities. An entity represents a specific object (such as a specific customer or order). Each entity must have a unique entity key within an entity set. An entity set is a collection of instances of a specific entity type. Entity sets (and association sets) are logically grouped in an entity container.

Inheritance is supported with entity types: that is, one entity type can be derived from another. For more information, see Entity Data Model: Inheritance.

Association Type

An association type (also called an association) is the fundamental building block for describing relationships in the Entity Data Model. In a conceptual model, an association represents a relationship between two entity types (such as Customer and Order). Every association has two association ends that specify the entity types involved in the association. Each association end also specifies an association end multiplicity that indicates the number of entities that can be at that end of the association. An association end multiplicity can have a value of one (1), zero or one (0..1), or many (*). Entities at one end of an association can be accessed through navigation properties, or through foreign keys if they are exposed on an entity type. For more information, see foreign key property.

In an application, an instance of an association represents a specific association (such as an association between an instance of Customer and instances of Order). Association instances are logically grouped in an association set. Association sets (and entity sets) are logically grouped in an entity container.

Property

Entity types contain properties that define their structure and characteristics. For example, a Customer entity type may have properties such as CustomerId, Name, and Address.

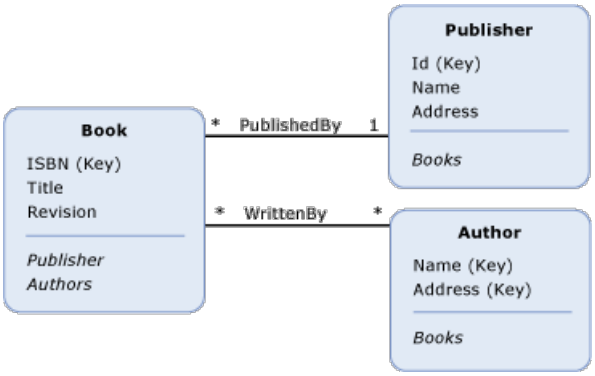
Properties in a conceptual model are analogous to properties defined on a class in a computer program. In the same way that properties on a class define the shape of the class and carry information about objects, properties in a conceptual model define the shape of an entity type and carry information about entity type instances.

A property can contain primitive data (such as a string, an integer, or a Boolean value), or structured data (such as a complex type). For more information, see Entity Data Model: Primitive Data Types.

Representations of a Conceptual Model

A *conceptual model* is a specific representation of the structure of some data as entities and relationships. One way to represent a conceptual model is with a diagram. The following diagram represents a conceptual model

with three entity types (`Book` , `Publisher` , and `Author`) and two associations (`PublishedBy` and `WrittenBy`):



This representation, however, has some shortcomings when it comes to conveying some details about the model. For example, property type and entity set information are not conveyed in the diagram. The richness of a conceptual model can be conveyed more clearly with a domain-specific language (DSL). The ADO.NET Entity Framework uses an XML-based DSL called *conceptual schema definition language* (CSDL) to define conceptual models. The following is the CSDL definition of the conceptual model in the diagram above:

```
  <Schema xmlns="http://schemas.microsoft.com/ado/2008/09/edm"
          xmlns:cg="http://schemas.microsoft.com/ado/2006/04/codegeneration"
          xmlns:store="http://schemas.microsoft.com/ado/2007/12/edm/EntityStoreSchemaGenerator"
          Namespace="BooksModel" Alias="Self">
    <EntityContainer Name="BooksContainer" >
      <EntitySet Name="Books" EntityType="BooksModel.Book" />
      <EntitySet Name="Publishers" EntityType="BooksModel.Publisher" />
      <EntitySet Name="Authors" EntityType="BooksModel.Author" />
      <AssociationSet Name="PublishedBy" Association="BooksModel.PublishedBy">
        <End Role="Book" EntitySet="Books" />
        <End Role="Publisher" EntitySet="Publishers" />
      </AssociationSet>
      <AssociationSet Name="WrittenBy" Association="BooksModel.WrittenBy">
        <End Role="Book" EntitySet="Books" />
        <End Role="Author" EntitySet="Authors" />
      </AssociationSet>
    </EntityContainer>
    <EntityType Name="Book">
      <Key>
        <PropertyRef Name="ISBN" />
      </Key>
      <Property Type="String" Name="ISBN" Nullable="false" />
      <Property Type="String" Name="Title" Nullable="false" />
      <Property Type="Decimal" Name="Revision" Nullable="false" Precision="29" Scale="29" />
      <NavigationProperty Name="Publisher" Relationship="BooksModel.PublishedBy"
                          FromRole="Book" ToRole="Publisher" />
      <NavigationProperty Name="Authors" Relationship="BooksModel.WrittenBy"
                          FromRole="Book" ToRole="Author" />
    </EntityType>
    <EntityType Name="Publisher">
      <Key>
        <PropertyRef Name="Id" />
      </Key>
      <Property Type="Int32" Name="Id" Nullable="false" />
      <Property Type="String" Name="Name" Nullable="false" />
      <Property Type="String" Name="Address" Nullable="false" />
      <NavigationProperty Name="Books" Relationship="BooksModel.PublishedBy"
                          FromRole="Publisher" ToRole="Book" />
    </EntityType>
    <EntityType Name="Author">
      <Key>
        <PropertyRef Name="Name" />
        <PropertyRef Name="Address" />
      </Key>
      <Property Type="String" Name="Name" Nullable="false" />
      <Property Type="String" Name="Address" Nullable="false" />
      <NavigationProperty Name="Books" Relationship="BooksModel.WrittenBy"
                          FromRole="Author" ToRole="Book" />
    </EntityType>
  <Association Name="PublishedBy">
        <End Type="BooksModel.Book" Role="Book" Multiplicity="*" />
        <End Type="BooksModel.Publisher" Role="Publisher" Multiplicity="1" />
    </Association>
    <Association Name="WrittenBy">
      <End Type="BooksModel.Book" Role="Book" Multiplicity="*" />
      <End Type="BooksModel.Author" Role="Author" Multiplicity="*" />
    </Association>
  </Schema>
```

See also

- Entity Data Model

Entity Data Model: Namespaces

11/7/2019 • 2 minutes to read • Edit Online

A namespace in the Entity Data Model (EDM) is an abstract container for entity types, complex types, and associations. Namespaces in the EDM are similar to namespaces in a programming language: they provide context for the objects that they contain and they provide a way to disambiguate objects that have the same name (but are contained in different namespaces).

Example

The ADO.NET Entity Framework uses a domain-specific language (DSL) called conceptual schema definition language (CSDL) to define conceptual models. The following CSDL code uses a namespace to identify a type that is defined in a different conceptual model. The example defines an entity type (`Publisher`) that has a complex type property (`Address`) that is imported from the `ExtendedBooksModel` namespace. Note that the `Using` element indicates that a namespace has been imported. Also note that the type of the `Address` property is defined by using its fully qualified name (`ExtendedBooksModel.Address`), indicating that this type is defined in the `ExtendedBooksModel` namespace.

```
  <Schema xmlns="http://schemas.microsoft.com/ado/2008/09/edm"
          xmlns:cg="http://schemas.microsoft.com/ado/2006/04/codegeneration"
          xmlns:store="http://schemas.microsoft.com/ado/2007/12/edm/EntityStoreSchemaGenerator"
          Namespace="BooksModel" Alias="Self">

    <Using Namespace="BooksModel.Extended" Alias="BMExt" />

<EntityContainer Name="BooksContainer" >
      <EntitySet Name="Publishers" EntityType="BooksModel.Publisher" />
    </EntityContainer>

<EntityType Name="Publisher">
      <Key>
        <PropertyRef Name="Id" />
      </Key>
      <Property Type="Int32" Name="Id" Nullable="false" />
      <Property Type="String" Name="Name" Nullable="false" />
      <Property Type="BMExt.Address" Name="Address" Nullable="false" />
    </EntityType>

</Schema>
```

See also

- Entity Data Model Key Concepts
- Entity Data Model

Entity Data Model: Primitive Data Types

9/7/2019 • 2 minutes to read • Edit Online

The Entity Data Model (EDM) supports a set of abstract primitive data types (such as String, Boolean, Int32, and so on) that are used to define properties in a conceptual model. These primitive data types are proxies for actual primitive data types that are supported in the storage or hosting environment, such as a SQL Server database or the common language runtime (CLR). The EDM does not define the semantics of operations or conversions over primitive data types; these semantics are defined by the storage or hosting environment. Typically, primitive data types in the EDM are mapped to corresponding primitive data types in the storage or hosting environment. For information about how the Entity Framework maps primitive types in the EDM to SQL Server data types, see SqlClient for Entity FrameworkTypes.

> **NOTE**
> The EDM does not support collections of primitive data types.

For information about structured data types in the EDM, see entity type and complex type.

Primitive Data Types Supported in the Entity Data Model

The table below lists the primitive data types supported by the EDM. The table also lists the facets that can be applied to each primitive data type.

PRIMITIVE DATA TYPE	DESCRIPTION	APPLICABLE FACETS
Binary	Contains binary data.	MaxLength, FixedLength, Nullable, Default
Boolean	Contains the value `true` or `false`.	Nullable, Default
Byte	Contains an unsigned 8-bit integer value.	Precision, Nullable, Default
DateTime	Represents a date and time.	Precision, Nullable, Default
DateTimeOffset	Contains a date and time as an offset in minutes from GMT.	Precision, Nullable, Default
Decimal	Contains a numeric value with fixed precision and scale.	Precision, Nullable, Default
Double	Contains a floating point number with 15 digit precision.	Precision, Nullable, Default
Float	Contains a floating point number with seven digit precision.	Precision, Nullable, Default
Guid	Contains a 16-byte unique identifier.	Precision, Nullable, Default
Int16	Contains a signed 16-bit integer value.	Precision, Nullable, Default

PRIMITIVE DATA TYPE	DESCRIPTION	APPLICABLE FACETS
Int32	Contains a signed 32-bit integer value.	Precision, Nullable, Default
Int64	Contains a signed 64-bit integer value.	Precision, Nullable, Default
SByte	Contains a signed 8-bit integer value.	Precision, Nullable, Default
String	Contains character data.	Unicode, FixedLength, MaxLength, Collation, Precision, Nullable, Default
Time	Contains a time of day.	Precision, Nullable, Default

See also

- Entity Data Model Key Concepts
- Entity Data Model

Entity Data Model: Inheritance

11/7/2019 • 2 minutes to read • Edit Online

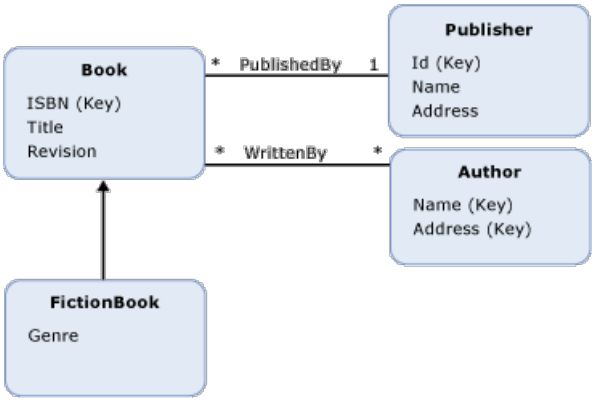
The Entity Data Model (EDM) supports inheritance for entity types. Inheritance in the EDM is similar to inheritance for classes in object-oriented programming languages. Like with classes in object-oriented languages, in a conceptual model you can define an entity type (a *derived type*) that inherits from another entity type (the *base type*). However, unlike classes in object-oriented programming, in a conceptual model the derived type always inherits all the properties and navigation properties of the base type. You cannot override inherited properties in a derived type.

In a conceptual model you can build inheritance hierarchies in which a derived type inherits from another derived type. The type at the top of the hierarchy (the one type in the hierarchy that is not a derived type) is called the *root type*. In an inheritance hierarchy, the entity key must be defined on the root type.

You cannot build inheritance hierarchies in which a derived type inherits from more than one type. For example, in a conceptual model with a `Book` entity type, you could define derived types `FictionBook` and `NonFictionBook` that each inherit from `Book`. However, you could not then define a type that inherits from both the `FictionBook` and `NonFictionBook` types.

Example

The following diagram shows a conceptual model with four entity types: `Book`, `FictionBook`, `Publisher`, and `Author`. The `FictionBook` entity type is a derived type, inheriting from the `Book` entity type. The `FictionBook` type inherits the `ISBN (Key)`, `Title`, and `Revision` properties, and defines an additional property called `Genre`.



The ADO.NET Entity Framework uses a domain-specific language (DSL) called conceptual schema definition language (CSDL) to define conceptual models. The following CSDL defines an entity type, `FictionBook`, that inherits from the `Book` type (as in the diagram above):

```
<EntityType Name="FictionBook" BaseType="BooksModel.Book" >
  <Property Type="String" Name="Genre" Nullable="false" />
</EntityType>
```

See also

- Entity Data Model Key Concepts
- Entity Data Model

association end

11/7/2019 • 2 minutes to read • Edit Online

An *association end* identifies the entity type on one end of an association and the number of entity type instances that can exist at that end of an association. Association ends are defined as part of an association; an association must have exactly two association ends. Navigation properties allow for navigation from one association end to the other.

An association end definition contains the following information:

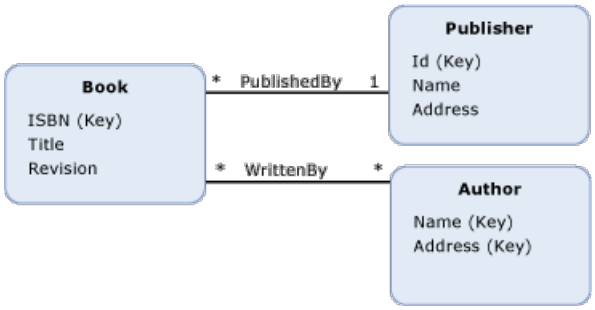
- One of the entity types involved in the association. (Required)

  > **NOTE**
  > For a given association, the entity type specified for each association end can be the same. This creates a self-association.

- An association end multiplicity that indicates the number of entity type instances that can be at one end of the association. An association end multiplicity can have a value of one (1), zero or one (0..1), or many (*).
- A name for the association end. (Optional)
- Information about operations that are performed on the association end, such as cascade on delete. (Optional)

Example

The diagram below shows a conceptual model with two associations: `PublishedBy` and `WrittenBy`. The association ends for the `PublishedBy` association are the `Book` and `Publisher` entity types. The multiplicity of the `Publisher` end is one (1) and the multiplicity of the `Book` end is many (*), indicating that a publisher publishes many books and a book is published by one publisher.



The ADO.NET Entity Framework uses a domain-specific language (DSL) called conceptual schema definition language (CSDL) to define conceptual models. The CSDL below defines the `PublishedBy` association shown in the diagram above. Note that the type, name, and multiplicity of each association end are specified by XML attributes (the `Type`, `Role`, and `Multiplicity` attributes, respectively). Optional information about operations performed on an end is specified in an XML element (the `OnDelete` element). In this case, if a publisher is deleted, so are all associated books.

```
  <Association Name="PublishedBy">
    <End Type="BooksModel.Book" Role="Book" Multiplicity="*" />
    <End Type="BooksModel.Publisher" Role="Publisher" Multiplicity="1" >
  <OnDelete Action="Cascade" />
</End>
  </Association>
```

See also

- Entity Data Model Key Concepts
- Entity Data Model

association end multiplicity

11/7/2019 • 2 minutes to read • Edit Online

Association end multiplicity defines the number of entity type instances that can be at one end of an association.

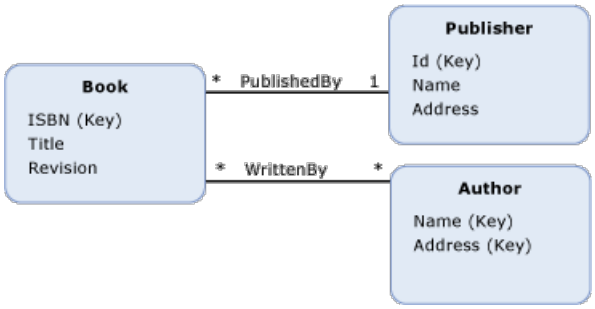
An association end multiplicity can have one of the following values:

- one (1): Indicates that exactly one entity type instance exists at the association end.
- zero or one (0..1): Indicates that zero or one entity type instances exist at the association end.
- many (*): Indicates that zero, one, or more entity type instances exist at the association end.

An association is often characterized by its association end multiplicities. For example, if the ends of an association have multiplicities one (1) and many (*), the association is called a one-to-many association. In the example below, the `PublishedBy` association is a one-to-many association (a publisher publishes many books and a book is published by one publisher). The `WrittenBy` association is a many-to-many association (a book can have multiple authors and an author can write multiple books).

Example

The diagram below shows a conceptual model with two associations: `PublishedBy` and `WrittenBy`. The association ends for the `PublishedBy` association are the `Book` and `Publisher` entity types. The multiplicity of the `Publisher` end is one (1) and the multiplicity of the `Book` end is many (*).



The ADO.NET Entity Framework uses a domain-specific language (DSL) called conceptual schema definition language (CSDL) to define conceptual models. The following CSDL defines the `PublishedBy` association shown in the diagram above:

```
<Association Name="PublishedBy">
      <End Type="BooksModel.Book" Role="Book" Multiplicity="*" />
      <End Type="BooksModel.Publisher" Role="Publisher" Multiplicity="1" />
    </Association>
```

See also

- Entity Data Model Key Concepts
- Entity Data Model

association set

11/7/2019 • 2 minutes to read • Edit Online

An *association set* is a logical container for association instances of the same type. An association set is not a data modeling construct; that is, it does not describe the structure of data or relationships. Instead, an association set provides a construct for a hosting or storage environment (such as the common language runtime or a SQL Server database) to group association instances so that they can be mapped to a data store.

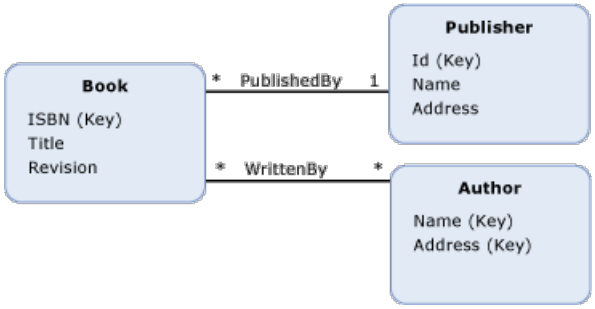
An association set is defined within an entity container, which is a logical grouping of entity sets and association sets.

A definition for an association set contains the following information:

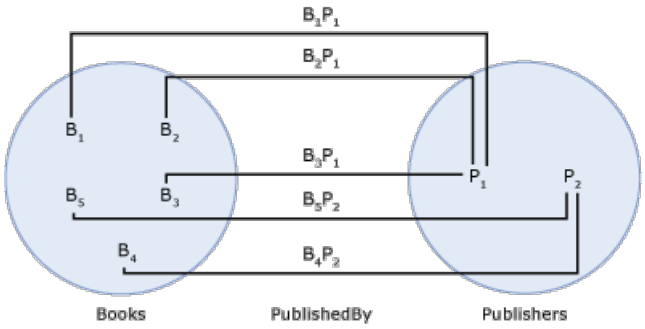
- The association set name. (Required)
- The association of which it will contain instances. (Required)
- Two association set ends.

Example

The diagram below shows a conceptual model with two associations: `PublishedBy`, and `WrittenBy`. Although information about association sets is not conveyed in the diagram, the next diagram shows an example of association sets and entity sets based on this model.



The following example shows an association set (`PublishedBy`) and two entity sets (`Books` and `Publishers`) based on the conceptual model shown above. Bi in the `Books` entity set represents an instance of the `Book` entity type at run time. Similarly, Pj represents a `Publisher` instance in the `Publishers` entity set. BiPj represents an instance of the `PublishedBy` association in the `PublishedBy` association set.



The ADO.NET Entity Framework uses a domain-specific language (DSL) called conceptual schema definition language (CSDL) to define conceptual models. The following CSDL defines an entity container with one association set for each association in the diagram above. Note that the name and association for each association set are defined using XML attributes.

```
<EntityContainer Name="BooksContainer" >
  <EntitySet Name="Books" EntityType="BooksModel.Book" />
  <EntitySet Name="Publishers" EntityType="BooksModel.Publisher" />
  <EntitySet Name="Authors" EntityType="BooksModel.Author" />
  <AssociationSet Name="PublishedBy" Association="BooksModel.PublishedBy">
    <End Role="Book" EntitySet="Books" />
    <End Role="Publisher" EntitySet="Publishers" />
  </AssociationSet>
  <AssociationSet Name="WrittenBy" Association="BooksModel.WrittenBy">
    <End Role="Book" EntitySet="Books" />
    <End Role="Author" EntitySet="Authors" />
  </AssociationSet>
</EntityContainer>
```

It is possible to define multiple association sets per association, as long as no two association sets share an association set end. The following CSDL defines an entity container with two association sets for the `WrittenBy` association. Notice that multiple entity sets have been defined for the `Book` and `Author` entity types and that no association set shares an association set end.

```
<EntityContainer Name="BooksContainer" >
  <EntitySet Name="Books" EntityType="BooksModel.Book" />
  <EntitySet Name="FictionBooks" EntityType="BooksModel.Book" />
  <EntitySet Name="Publishers" EntityType="BooksModel.Publisher" />
  <EntitySet Name="Authors" EntityType="BooksModel.Author" />
  <EntitySet Name="FictionAuthors" EntityType="BooksModel.Author" />
  <AssociationSet Name="PublishedBy" Association="BooksModel.PublishedBy">
    <End Role="Book" EntitySet="Books" />
    <End Role="Publisher" EntitySet="Publishers" />
  </AssociationSet>
  <AssociationSet Name="WrittenBy" Association="BooksModel.WrittenBy">
    <End Role="Book" EntitySet="Books" />
    <End Role="Author" EntitySet="Authors" />
  </AssociationSet>
  <AssociationSet Name="FictionWrittenBy" Association="BooksModel.WrittenBy">
    <End Role="Book" EntitySet="FictionBooks" />
    <End Role="Author" EntitySet="FictionAuthors" />
  </AssociationSet>
</EntityContainer>
```

See also

- Entity Data Model Key Concepts
- Entity Data Model
- foreign key property

association set end

11/7/2019 • 2 minutes to read • Edit Online

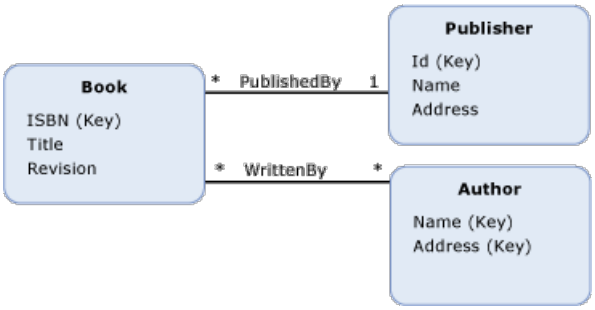
An *association set end* identifies the entity type and the entity set at the end of an association set. Association set ends are defined as part of an association set; an association set must have exactly two association set ends.

An association set end definition contains the following information:

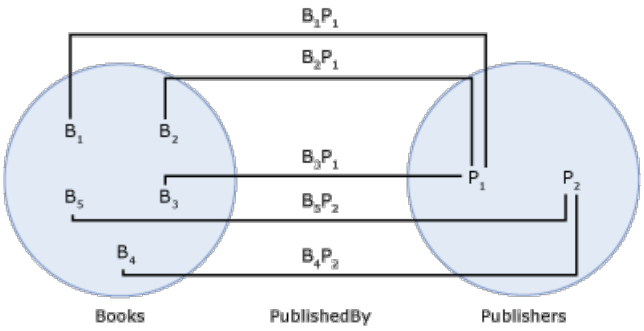
- One of the entity types involved in the association set. (Required)
- The entity set for the entity type involved in the association set. (Required)

Example

The diagram below shows a conceptual model with two associations: `WrittenBy` and `PublishedBy`.



The following diagram shows an association set (`PublishedBy`) and two entity sets (`Books` and `Publishers`) based on the conceptual model shown above. The association set ends are the `Books` and `Publishers` entity sets. Bi in the `Books` entity set represents an instance of the `Book` entity type at run time. Similarly, Pj represents a `Publisher` instance in the `Publishers` entity set. BiPj represents an instance of the `PublishedBy` association in the `PublishedBy` association set.



The ADO.NET Entity Framework uses a DSL called conceptual schema definition language (CSDL) to define conceptual models. The following CSDL defines an entity container with one association set for each association in the diagram above. Note that association set ends are defined as part of each association set definition.

```
<EntityContainer Name="BooksContainer" >
  <EntitySet Name="Books" EntityType="BooksModel.Book" />
  <EntitySet Name="Publishers" EntityType="BooksModel.Publisher" />
  <EntitySet Name="Authors" EntityType="BooksModel.Author" />
  <AssociationSet Name="PublishedBy" Association="BooksModel.PublishedBy">
    <End Role="Book" EntitySet="Books" />
    <End Role="Publisher" EntitySet="Publishers" />
  </AssociationSet>
  <AssociationSet Name="WrittenBy" Association="BooksModel.WrittenBy">
    <End Role="Book" EntitySet="Books" />
    <End Role="Author" EntitySet="Authors" />
  </AssociationSet>
</EntityContainer>
```

See also

- Entity Data Model Key Concepts
- Entity Data Model

association type

11/7/2019 • 2 minutes to read • Edit Online

An *association type* (also called an association) is the fundamental building block for describing relationships in the Entity Data Model (EDM). In a conceptual model, an association represents a relationship between two entity types (such as `Customer` and `Order`). In an application, an instance of an association represents a specific association (such as an association between an instance of `Customer` and an instance of `Order`). Association instances are logically grouped in an association set.

An association definition contains the following information:

- A unique name. (Required)
- Two association ends, one for each entity type in the relationship. (Required)

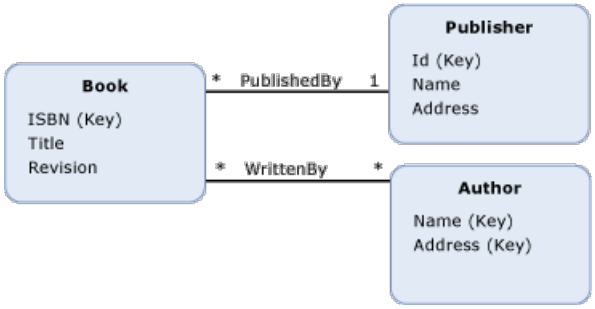
  > **NOTE**
  > An association cannot represent a relationship among more than two entity types. An association can, however, define a self-relationship by specifying the same entity type for each of its association ends.

- A referential integrity constraint. (Optional)

Each association end must specify an association end multiplicity that indicates the number of entity type instances that can be at one end of the association. An association end multiplicity can have a value of one (1), zero or one (0..1), or many (*). Entity type instances at one end of an association can be accessed through navigation properties or foreign keys if they are exposed on an entity type. For more information, see Entity Data Model: Foreign Keys.

Example

The diagram below shows a conceptual model with two associations: `PublishedBy` and `WrittenBy`. The association ends for the `PublishedBy` association are the `Book` and `Publisher` entity types. The multiplicity of the `Publisher` end is one (1) and the multiplicity of the `Book` end is many (*), indicating that a publisher publishes many books and a book is published by one publisher.



The ADO.NET Entity Framework uses a domain-specific language (DSL) called conceptual schema definition language (CSDL) to define conceptual models. The following CSDL defines the `PublishedBy` association shown in the diagram above:

```
<Association Name="PublishedBy">
      <End Type="BooksModel.Book" Role="Book" Multiplicity="*" />
      <End Type="BooksModel.Publisher" Role="Publisher" Multiplicity="1" />
    </Association>
```

See also

- Entity Data Model Key Concepts
- Entity Data Model

complex type

11/7/2019 • 2 minutes to read • Edit Online

A *complex type* is a template for defining rich, structured properties on entity types or on other complex types. Each template contains the following:

- A unique name. (Required)

  > **NOTE**
  > The name of a complex type cannot be the same as an entity type name within the same namespace.

- Data in the form of one or more properties. (Optional.)

  > **NOTE**
  > A property of a complex type can be another complex type.

A complex type is similar to an entity type in that a complex type can carry a data payload in the form of primitive type properties or other complex types. However, there are some key differences between complex types and entity types:

- Complex types do not have identities and therefore cannot exist independently. Complex types can only exist as properties on entity types or other complex types.
- Complex types cannot participate in associations. Neither end of an association can be a complex type, and therefore navigation properties cannot be defined on complex types.

Example

The ADO.NET Entity Framework uses a domain-specific language (DSL) called conceptual schema definition language (CSDL) to define conceptual models. The following CSDL defines a complex type, Address, with the primitive type properties `StreetAddress`, `City`, `StateOrProvince`, `Country`, and `PostalCode`.

```
<ComplexType Name="Address" >
  <Property Type="String" Name="StreetAddress" Nullable="false" />
  <Property Type="String" Name="City" Nullable="false" />
  <Property Type="String" Name="StateOrProvince" Nullable="false" />
  <Property Type="String" Name="Country" Nullable="false" />
  <Property Type="String" Name="PostalCode" Nullable="false" />
</ComplexType>
```

To define the complex type `Address` (above) as a property on an entity type, you must declare the property type in the entity type definition. The following CSDL declares the `Address` property as a complex type on an entity type (Publisher):

```
<EntityType Name="Publisher">
      <Key>
        <PropertyRef Name="Id" />
      </Key>
      <Property Type="Int32" Name="Id" Nullable="false" />
      <Property Type="String" Name="Name" Nullable="false" />
      <Property Type="BooksModel.Address" Name="Address" Nullable="false" />
      <NavigationProperty Name="Books" Relationship="BooksModel.PublishedBy"
                          FromRole="Publisher" ToRole="Book" />
    </EntityType>
```

See also

- Entity Data Model Key Concepts
- Entity Data Model

entity container

11/7/2019 • 2 minutes to read • Edit Online

An *entity container* is a logical grouping of entity sets, association sets, and function imports.

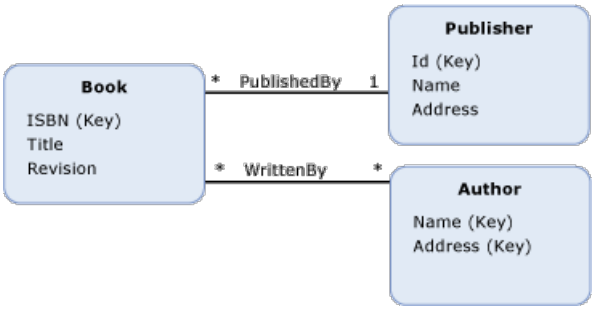
The following must be true of an entity container defined in a conceptual model:

- At least one entity container must be defined in each conceptual model.
- The entity container must have a unique name within each conceptual model.

An entity container can define entity sets or association sets that use entity types or associations defined in one or more namespaces. For more information, see Entity Data Model: Namespaces.

Example

The diagram below shows a conceptual model with three entity types: `Book`, `Publisher`, and `Author`. See the next example for more information.



Although the diagram does not convey entity container information, the conceptual model must define an entity container. The ADO.NET Entity Framework uses a DSL called conceptual schema definition language (CSDL) to define conceptual models. The following CSDL defines an entity container for the conceptual model shown in the diagram above. Note that the entity container name is defined in an XML attribute.

```
<EntityContainer Name="BooksContainer" >
  <EntitySet Name="Books" EntityType="BooksModel.Book" />
  <EntitySet Name="Publishers" EntityType="BooksModel.Publisher" />
  <EntitySet Name="Authors" EntityType="BooksModel.Author" />
  <AssociationSet Name="PublishedBy" Association="BooksModel.PublishedBy">
    <End Role="Book" EntitySet="Books" />
    <End Role="Publisher" EntitySet="Publishers" />
  </AssociationSet>
  <AssociationSet Name="WrittenBy" Association="BooksModel.WrittenBy">
    <End Role="Book" EntitySet="Books" />
    <End Role="Author" EntitySet="Authors" />
  </AssociationSet>
</EntityContainer>
```

See also

- Entity Data Model Key Concepts
- Entity Data Model

entity key

11/7/2019 • 2 minutes to read • Edit Online

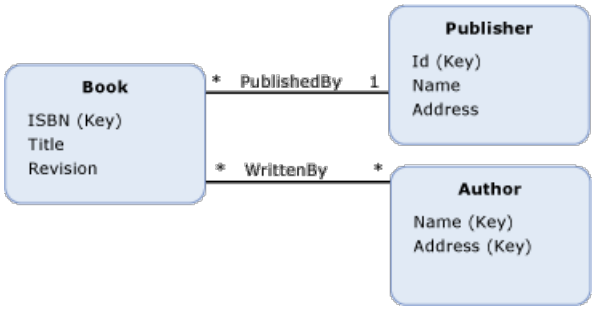
An *entity key* is a property or a set of properties of an entity type that are used to determine identity. The properties that make up an entity key are chosen at design time. The values of entity key properties must uniquely identify an entity type instance within an entity set at run time. The properties that make up an entity key should be chosen to guarantee uniqueness of instances in an entity set.

The following are the requirements for a set of properties to be an entity key:

- No two entity keys within an entity set can be identical. That is, for any two entities within an entity set, the values for all of the properties that constitute a key cannot be the same. However, some (but not all) of the values that make up an entity key can be the same.
- An entity key must consist of a set of non-nullable, immutable, primitive type properties.
- The properties that make up an entity key for a given entity type cannot change. You cannot allow more than one possible entity key for a given entity type; surrogate keys are not supported.
- When an entity is involved in an inheritance hierarchy, the root entity must contain all the properties that make up the entity key, and the entity key must be defined on the root entity type. For more information, see Entity Data Model: Inheritance.

Example

The diagram below shows a conceptual model with three entity types: `Book`, `Publisher`, and `Author`. The properties of each entity type that make up its entity key are denoted with "(Key)". Note that the `Author` entity type has an entity key that consists of two properties, `Name` and `Address`.



The ADO.NET Entity Framework uses a domain-specific language (DSL) called conceptual schema definition language (CSDL) to define conceptual models. The CSDL below defines the `Book` entity type shown in the diagram above. Note that the entity key is defined by referencing the `ISBN` property of the entity type.

```
<EntityType Name="Book">
  <Key>
    <PropertyRef Name="ISBN" />
  </Key>
  <Property Type="String" Name="ISBN" Nullable="false" />
  <Property Type="String" Name="Title" Nullable="false" />
  <Property Type="Decimal" Name="Revision" Nullable="false" Precision="29" Scale="29" />
  <NavigationProperty Name="Publisher" Relationship="BooksModel.PublishedBy"
                      FromRole="Book" ToRole="Publisher" />
  <NavigationProperty Name="Authors" Relationship="BooksModel.WrittenBy"
                      FromRole="Book" ToRole="Author" />
</EntityType>
```

The `ISBN` property is a good choice for the entity key because an International Standard Book Number (ISBN) uniquely identifies a book.

The CSDL below defines the `Author` entity type shown in the diagram above. Note that the entity key consists of two properties, `Name` and `Address`.

```
<EntityType Name="Author">
  <Key>
    <PropertyRef Name="Name" />
    <PropertyRef Name="Address" />
  </Key>
  <Property Type="String" Name="Name" Nullable="false" />
  <Property Type="String" Name="Address" Nullable="false" />
  <NavigationProperty Name="Books" Relationship="BooksModel.WrittenBy"
                      FromRole="Author" ToRole="Book" />
</EntityType>
```

Using `Name` and `Address` for the entity key is a reasonable choice, because two authors of the same name are unlikely to live at the same address. However, this choice for an entity key does not absolutely guarantee unique entity keys in an entity set. Adding a property, such as `AuthorId`, that could be used to uniquely identify an author would be recommended in this case.

See also

- Entity Data Model Key Concepts
- Entity Data Model

entity set

11/7/2019 • 2 minutes to read • Edit Online

An *entity set* is a logical container for instances of an entity type and instances of any type derived from that entity type. (For information about derived types, see Entity Data Model: Inheritance.) The relationship between an entity type and an entity set is analogous to the relationship between a row and a table in a relational database: Like a row, an entity type describes data structure, and, like a table, an entity set contains instances of a given structure. An entity set is not a data modeling construct; it does not describe the structure of data. Instead, an entity set provides a construct for a hosting or storage environment (such as the common language runtime or a SQL Server database) to group entity type instances so that they can be mapped to a data store.

An entity set is defined within an entity container, which is a logical grouping of entity sets and association sets.

For an entity type instance to exist in an entity set, the following must be true:

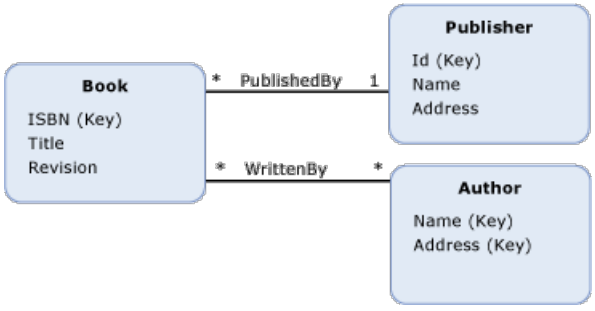
- The type of the instance is either the same as the entity type on which the entity set is based, or the type of the instance is a subtype of the entity type.
- The entity key for the instance is unique within the entity set.
- The instance does not exist in any other entity set.

> **NOTE**
> Multiple entity sets can be defined using the same entity type, but an instance of a given entity type can only exist in one entity set.

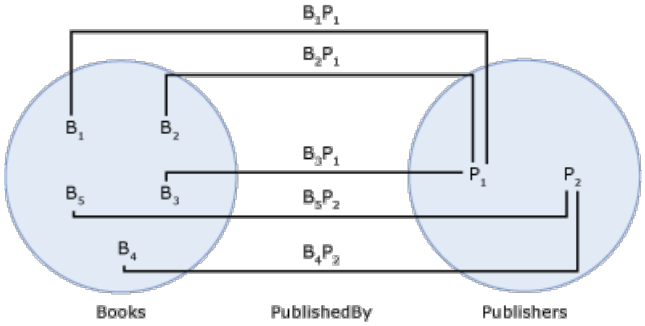
You do not have to define an entity set for each entity type in a conceptual model.

Example

The diagram below shows a conceptual model with three entity types: `Book`, `Publisher`, and `Author`.



The following diagram shows two entity sets (`Books` and `Publishers`) and an association set (`PublishedBy`) based on the conceptual model shown above. Bi in the `Books` entity set represents an instance of the `Book` entity type at run time. Similarly, Pj represent a `Publisher` instance in the `Publishers` entity set. BiPj represents an instance of the `PublishedBy` association in the `PublishedBy` association set.



The ADO.NET Entity Framework uses a domain-specific language (DSL) called conceptual schema definition language (CSDL) to define conceptual models. The following CSDL defines an entity container with one entity set for each entity type in the conceptual model shown above. Note that the name and entity type for each entity set are defined using XML attributes.

```
<EntityContainer Name="BooksContainer" >
  <EntitySet Name="Books" EntityType="BooksModel.Book" />
  <EntitySet Name="Publishers" EntityType="BooksModel.Publisher" />
  <EntitySet Name="Authors" EntityType="BooksModel.Author" />
  <AssociationSet Name="PublishedBy" Association="BooksModel.PublishedBy">
    <End Role="Book" EntitySet="Books" />
    <End Role="Publisher" EntitySet="Publishers" />
  </AssociationSet>
  <AssociationSet Name="WrittenBy" Association="BooksModel.WrittenBy">
    <End Role="Book" EntitySet="Books" />
    <End Role="Author" EntitySet="Authors" />
  </AssociationSet>
</EntityContainer>
```

It is possible to define multiple entity sets per type (MEST). The following CSDL defines an entity container with two entity sets for the `Book` entity type:

```
<EntityContainer Name="BooksContainer" >
  <EntitySet Name="Books" EntityType="BooksModel.Book" />
  <EntitySet Name="FictionBooks" EntityType="BooksModel.Book" />
  <EntitySet Name="Publishers" EntityType="BooksModel.Publisher" />
  <EntitySet Name="Authors" EntityType="BooksModel.Author" />
  <AssociationSet Name="PublishedBy" Association="BooksModel.PublishedBy">
    <End Role="Book" EntitySet="Books" />
    <End Role="Publisher" EntitySet="Publishers" />
  </AssociationSet>
  <AssociationSet Name="BookAuthor" Association="BooksModel.BookAuthor">
    <End Role="Book" EntitySet="Books" />
    <End Role="Author" EntitySet="Authors" />
  </AssociationSet>
</EntityContainer>
```

See also

- Entity Data Model Key Concepts
- Entity Data Model

entity type

11/7/2019 • 2 minutes to read • Edit Online

The *entity type* is the fundamental building block for describing the structure of data with the Entity Data Model (EDM). In a conceptual model, an entity type represents the structure of top-level concepts, such as customers or orders. An entity type is a template for entity type instances. Each template contains the following information:

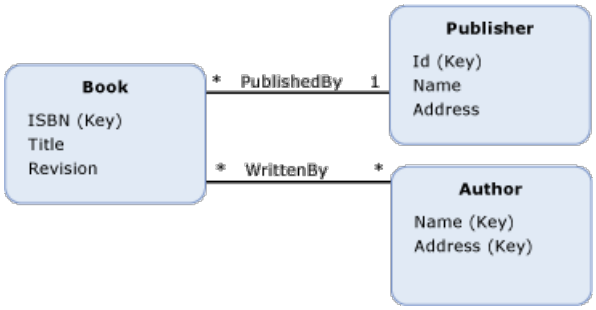
- A unique name. (Required.)
- An entity key defined by one or more properties. (Required.)
- Data in the form of properties. (Optional.)
- Navigation properties that allow for navigation from one end of an association to the other end. (Optional)

In an application, an instance of an entity type represents a specific object (such as a specific customer or order). Each instance of an entity type must have a unique entity key within an entity set.

Two entity type instances are considered equal only if they are of the same type and the values of their entity keys are the same.

Example

The diagram below shows a conceptual model with three entity types: `Book`, `Publisher`, and `Author`:



Note that the properties of each entity type that make up its entity key are denoted with "(Key)".

The ADO.NET Entity Framework uses a domain-specific language (DSL) called conceptual schema definition language (CSDL) to define conceptual models. The following CSDL defines the `Book` entity type shown in the diagram above:

```
<EntityType Name="Book">
  <Key>
    <PropertyRef Name="ISBN" />
  </Key>
  <Property Type="String" Name="ISBN" Nullable="false" />
  <Property Type="String" Name="Title" Nullable="false" />
  <Property Type="Decimal" Name="Revision" Nullable="false" Precision="29" Scale="29" />
  <NavigationProperty Name="Publisher" Relationship="BooksModel.PublishedBy"
                      FromRole="Book" ToRole="Publisher" />
  <NavigationProperty Name="Authors" Relationship="BooksModel.WrittenBy"
                      FromRole="Book" ToRole="Author" />
</EntityType>
```

See also

- Entity Data Model Key Concepts
- Entity Data Model
- facet

facet

11/7/2019 • 2 minutes to read • Edit Online

A *facet* is used to add detail to a primitive type property definition. A property definition contains information about the property type, but often more detail is necessary. For example, an entity type in a conceptual model might have a property of type `String` whose value cannot be set to null. Facets allow you to specify this level of detail.

The table below describes the facets that are supported in the EDM.

> **NOTE**
> The exact values and behaviors of facets are determined by the run-time environment that uses an EDM implementation.

FACET	DESCRIPTION	APPLIES TO
`Collation`	Specifies the collating sequence (or sorting sequence) to be used when performing comparison and ordering operations on values of the property.	`String`
`ConcurrencyMode`	Indicates that the value of the property should be used for optimistic concurrency checks.	All primitive type properties
`Default`	Specifies the default value of the property if no value is supplied upon instantiation.	All primitive type properties
`FixedLength`	Specifies whether the length of the property value can vary.	`Binary`, `String`
`MaxLength`	Specifies the maximum length of the property value.	`Binary`, `String`
`Nullable`	Specifies whether the property can have a null value.	All primitive type properties
`Precision`	For properties of type `Decimal`, specifies the number of digits a property value can have. For properties of type `Time`, `DateTime`, and `DateTimeOffset`, specifies the number of digits for the fractional part of seconds of the property value.	`DateTime`, `DateTimeOffset`, `Decimal`, `Time`,
`Scale`	Specifies the number of digits to the right of the decimal point for the property value.	Decimal
`Unicode`	Indicates whether the property value is stored as Unicode.	`String`

Example

The ADO.NET Entity Framework uses a domain-specific language (DSL) called conceptual schema definition language (CSDL) to define conceptual models. The following CSDL defines a `Book` entity type. Note that facets are implemented as XML attributes. The facet values indicate that no property can be set to null, and that the `Scale` and `Precision` of the `Revision` property are each set to 29.

```
<EntityType Name="Book">
  <Key>
    <PropertyRef Name="ISBN" />
  </Key>
  <Property Type="String" Name="ISBN" Nullable="false" />
  <Property Type="String" Name="Title" Nullable="false" />
  <Property Type="Decimal" Name="Revision" Nullable="false" Precision="29" Scale="29" />
  <NavigationProperty Name="Publisher" Relationship="BooksModel.PublishedBy"
                      FromRole="Book" ToRole="Publisher" />
  <NavigationProperty Name="Authors" Relationship="BooksModel.WrittenBy"
                      FromRole="Book" ToRole="Author" />
</EntityType>
```

See also

- Entity Data Model Key Concepts
- Entity Data Model

foreign key property

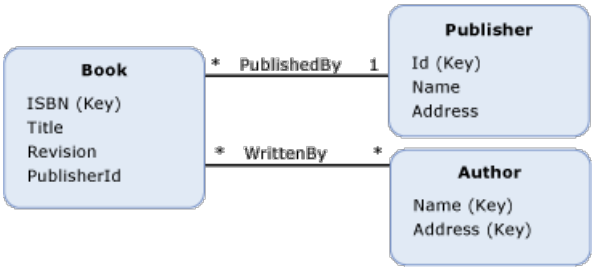
11/7/2019 • 2 minutes to read • Edit Online

A *foreign key property* in the Entity Data Model (EDM) is a primitive type property (or a set of primitive type properties) on an entity type that contains the entity key of another entity type.

A foreign key property is analogous to a foreign key column in a relational database. In the same way that foreign key columns are used in a relational database to create relationships between rows in tables, foreign key properties in a conceptual model are used to establish associations between entity types. A referential integrity constraint is used to define an association between two entity types when one of the types has a foreign key property.

Example

The diagram below shows a conceptual model with three entity types: `Book`, `Publisher`, and `Author`. The `Book` entity type has a property, `PublisherId`, that references the entity key of the `Publisher` entity type when you define a referential integrity constraint on the `PublishedBy` association.



The ADO.NET Entity Framework uses a domain-specific language (DSL) called conceptual schema definition language (CSDL) to define conceptual models. The following CSDL uses the foreign key property `PublisherId` to define a referential integrity constraint on the `PublishedBy` association shown in the conceptual model shown above.

```
<Association Name="PublishedBy">
  <End Type="BooksModel.Book" Role="Book" Multiplicity="*" >
  </End>
  <End Type="BooksModel.Publisher" Role="Publisher" Multiplicity="1" />
  <ReferentialConstraint>
    <Principal Role="Publisher">
      <PropertyRef Name="Id" />
    </Principal>
    <Dependent Role="Book">
      <PropertyRef Name="PublisherId" />
    </Dependent>
  </ReferentialConstraint>
</Association>
```

See also

- Entity Data Model Key Concepts
- Entity Data Model

model-declared function

9/7/2019 • 2 minutes to read • Edit Online

A *model-declared function* is a function that is declared in a conceptual model, but is not defined in that conceptual model. The function might be defined in the hosting or storage environment. For example, a model-declared function might be mapped to a function that is defined in a database, thus exposing server-side functionality in the conceptual model.

The declaration of a model-declared function contains the following information:

- The name of the function. (Required)
- The type of the return value. (Optional)

  > **NOTE**
  > If no return value is specified, the return type is void.

- Parameter information, including parameter name and type. (Optional)

Example

The ADO.NET Entity Framework uses a domain-specific language (DSL) called conceptual schema definition language (CSDL) to define conceptual models. In CSDL, one implementation of a model-declared function is a function import (using the FunctionImport element). The following CSDL defines an entity container with a function import definition. Note that the return type for the function is void since no return type is specified.

```
<FunctionImport Name="UpdatePublisher">
  <Parameter Name="PublisherId" Mode="In" Type="Int32" />
  <Parameter Name="PublisherName" Mode="In" Type="String" />
</FunctionImport>
```

See also

- Entity Data Model Key Concepts
- Entity Data Model

model-defined function

11/7/2019 • 2 minutes to read • Edit Online

A *model-defined function* is a function that is defined in a conceptual model. The body of a model-defined function is expressed in Entity SQL, which allows for the function to be expressed independently of rules or languages supported in the data source.

A definition for a model-defined function contains the following information:

- A function name. (Required)
- The type of the return value. (Optional)

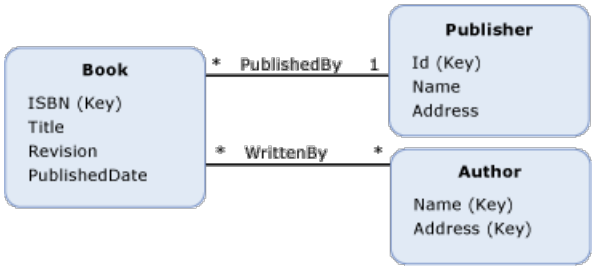
  > **NOTE**
  > If no return type is specified, the return value is void.

- Parameter information. (Optional)
- An Entity SQL expression that defines the body of the function.

Note that model-defined functions do not support output parameters. This restriction is in place so that model-defined functions can be composed.

Example

The diagram below shows a conceptual model with three entity types: `Book`, `Publisher`, and `Author`.



The ADO.NET Entity Framework uses a domain-specific language (DSL) called conceptual schema definition language (CSDL) to define conceptual models. The following CSDL defines a function in the conceptual model that returns the numbers of years since an instance of a `Book` (in the diagram above) was published.

```
<Function Name="GetYearsInPrint" ReturnType="Edm.Int32" >
      <Parameter Name="book" Type="BooksModel.Book" />
      <DefiningExpression>
        Year(CurrentDateTime()) - Year(cast(book.PublishedDate as DateTime))
      </DefiningExpression>
    </Function>
```

See also

- Entity Data Model Key Concepts
- Entity Data Model
- Entity Data Model: Primitive Data Types

Navigation property

2/8/2020 • 2 minutes to read • Edit Online

A *navigation property* is an optional property on an entity type that allows for navigation from one end of an association to the other end. Unlike other properties, navigation properties do not carry data.

A navigation property definition includes the following:

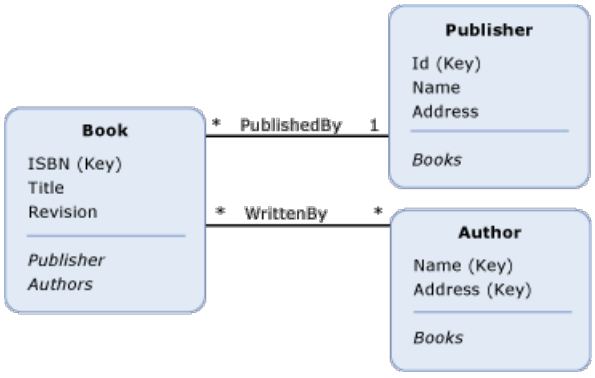
- A name. (Required)
- The association that it navigates. (Required)
- The ends of the association that it navigates. (Required)

Navigation properties are optional on both entity types at the ends of an association. If you define a navigation property on one entity type at the end of an association, you do not have to define a navigation property on the entity type at the other end of the association.

The data type of a navigation property is determined by the multiplicity of its remote association end. For example, suppose a navigation property, `OrdersNavProp`, exists on a `Customer` entity type and navigates a one-to-many association between `Customer` and `Order`. Because the remote association end for the navigation property has multiplicity of many (*), its data type is a collection (of `Order`). Similarly, if a navigation property, `CustomerNavProp`, exists on the `Order` entity type, its data type would be `Customer`, because the multiplicity of the remote end is one (1).

Example

The diagram below shows a conceptual model with three entity types: `Book`, `Publisher`, and `Author`. The navigation properties `Publisher` and `Authors` are defined on the Book entity type. Navigation property `Books` is defined on both the Publisher entity type and the `Author` entity type.



The ADO.NET Entity Framework uses a domain-specific language (DSL) called conceptual schema definition language (CSDL) to define conceptual models. The following CSDL defines the `Book` entity type shown in the diagram above:

```
<EntityType Name="Book">
  <Key>
    <PropertyRef Name="ISBN" />
  </Key>
  <Property Type="String" Name="ISBN" Nullable="false" />
  <Property Type="String" Name="Title" Nullable="false" />
  <Property Type="Decimal" Name="Revision" Nullable="false" Precision="29" Scale="29" />
  <NavigationProperty Name="Publisher" Relationship="BooksModel.PublishedBy"
                      FromRole="Book" ToRole="Publisher" />
  <NavigationProperty Name="Authors" Relationship="BooksModel.WrittenBy"
                      FromRole="Book" ToRole="Author" />
</EntityType>
```

XML attributes are used to communicate the information necessary to define a navigation property: The attribute `Name` contains the name of the property, `Relationship` contains the name of the association it navigates, and `FromRole` and `ToRole` contain the ends of the association.

See also

- Entity Data Model Key Concepts
- Entity Data Model
- Relationships, navigation properties, and foreign keys

property

11/7/2019 • 2 minutes to read • Edit Online

Properties are the fundamental building blocks of entity types and complex types. Properties define the shape and characteristics of data that an entity type instance or complex type instance will contain. Properties in a conceptual model are analogous to properties defined on a class. In the same way that properties on a class define the shape of the class and carry information about objects, properties in a conceptual model define the shape of an entity type and carry information about entity type instances.

> **NOTE**
>
> Properties, as described in this topic, are different from navigation properties. For more information, see navigation properties.

A property definition contains the following information:

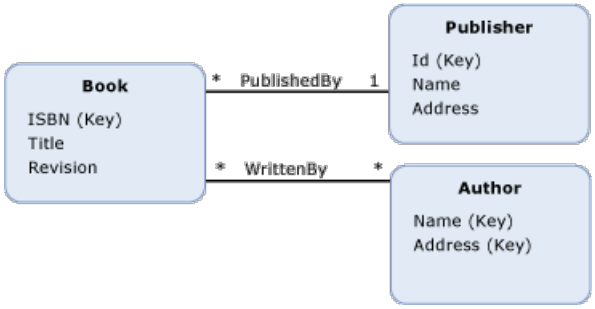
- A property name. (Required)
- A property type. (Required)
- A set of facets. (Optional)

A property can contain primitive data (such as a string, an integer, or a Boolean value), or structured data (such as a complex type). Properties that are of primitive type are also called scalar properties. For more information, see Entity Data Model: Primitive Data Types.

> **NOTE**
>
> A complex type can, itself, have properties that are complex types.

Example

The diagram below shows a conceptual model with three entity types: `Book`, `Publisher`, and `Author`. Each entity type has several properties, although type information for each property is not conveyed in the diagram. Properties that are entity keys are denoted with (Key).



The ADO.NET Entity Framework uses a domain-specific language (DSL) called conceptual schema definition language (CSDL) to define conceptual models. The following CSDL defines the `Book` entity type (as shown in the diagram above) and indicates the type and name of each property by using XML attributes. An optional facet, `Nullable`, is also defined by using an XML attribute.

```
<EntityType Name="Book">
  <Key>
    <PropertyRef Name="ISBN" />
  </Key>
  <Property Type="String" Name="ISBN" Nullable="false" />
  <Property Type="String" Name="Title" Nullable="false" />
  <Property Type="Decimal" Name="Revision" Nullable="false" Precision="29" Scale="29" />
  <NavigationProperty Name="Publisher" Relationship="BooksModel.PublishedBy"
                      FromRole="Book" ToRole="Publisher" />
  <NavigationProperty Name="Authors" Relationship="BooksModel.WrittenBy"
                      FromRole="Book" ToRole="Author" />
</EntityType>
```

It is possible that one of the properties shown in the diagram is a complex type property. For example, the `Address` property on the `Publisher` entity type could be a complex type property composed of several scalar properties, such as `StreetAddress`, `City`, `StateOrProvince`, `Country`, and `PostalCode`. The CSDL representation of such a complex type would be as follows:

```
<ComplexType Name="Address" >
  <Property Type="String" Name="StreetAddress" Nullable="false" />
  <Property Type="String" Name="City" Nullable="false" />
  <Property Type="String" Name="StateOrProvince" Nullable="false" />
  <Property Type="String" Name="Country" Nullable="false" />
  <Property Type="String" Name="PostalCode" Nullable="false" />
</ComplexType>
```

See also

- Entity Data Model Key Concepts
- Entity Data Model

referential integrity constraint

11/7/2019 • 2 minutes to read • Edit Online

A *referential integrity constraint* in the Entity Data Model (EDM) is similar to a referential integrity constraint in a relational database. In the same way that a column (or columns) from a database table can reference the primary key of another table, a property (or properties) of an entity type can reference the entity key of another entity type. The entity type that is referenced is called the *principal end* of the constraint. The entity type that references the principal end is called the *dependent end* of the constraint.

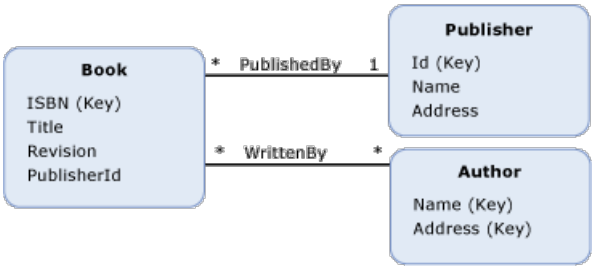
A referential integrity constraint is defined as part of an association between two entity types. The definition for a referential integrity constraint specifies the following information:

- The principal end of the constraint. (An entity type whose entity key is referenced by the dependent end.)
- The entity key of the principal end.
- The dependent end of the constraint. (An entity type that has a property or properties that reference the entity key of the principal end.)
- The referencing property or properties of the dependent end.

The purpose of referential integrity constraints in the EDM is to ensure that valid associations always exist. For more information, see foreign key property.

Example

The diagram below shows a conceptual model with two associations: `WrittenBy` and `PublishedBy`. The `Book` entity type has a property, `PublisherId`, that references the entity key of the `Publisher` entity type when you define a referential integrity constraint on the `PublishedBy` association.



The ADO.NET Entity Framework uses a domain-specific language (DSL) called conceptual schema definition language (CSDL) to define conceptual models. The following CSDL defines a referential integrity constraint on the `PublishedBy` association shown in the conceptual model above.

```
<Association Name="PublishedBy">
  <End Type="BooksModel.Book" Role="Book" Multiplicity="*" >
  </End>
  <End Type="BooksModel.Publisher" Role="Publisher" Multiplicity="1" />
  <ReferentialConstraint>
    <Principal Role="Publisher">
      <PropertyRef Name="Id" />
    </Principal>
    <Dependent Role="Book">
      <PropertyRef Name="PublisherId" />
    </Dependent>
  </ReferentialConstraint>
</Association>
```

See also

- Entity Data Model Key Concepts
- Entity Data Model

Oracle and ADO.NET

2/4/2020 • 2 minutes to read • Edit Online

> **NOTE**
> The types in System.Data.OracleClient are deprecated. The types remain supported in the current version of.NET Framework but will be removed in a future release. Microsoft recommends that you use a third-party Oracle provider.

This section describes features and behaviors that are specific to the .NET Framework Data Provider for Oracle.

The .NET Framework Data Provider for Oracle provides access to an Oracle database using the Oracle Call Interface (OCI) as provided by Oracle Client software. The functionality of the data provider is designed to be similar to that of the .NET Framework data providers for SQL Server, OLE DB, and ODBC.

To use the .NET Framework Data Provider for Oracle, an application must reference the System.Data.OracleClient namespace as follows:

```
Imports System.Data.OracleClient
```

```
using System.Data.OracleClient;
```

You also must include a reference to the DLL when you compile your code. For example, if you are compiling a C# program, your command line should include:

```
csc /r:System.Data.OracleClient.dll
```

In This Section

System Requirements
Describes requirements for using the .NET Framework Data Provider for Oracle, and describes a number of issues to be aware when using it.

Oracle BFILEs
Describes the OracleBFile class, which is used to work with the Oracle BFILE data type.

Oracle LOBs
Describes the OracleLob class, which is used to work with Oracle LOB data types.

Oracle REF CURSORs
Describes support for the Oracle REF CURSOR data type.

OracleTypes
Describes structures you can use to work with Oracle data types, including OracleNumber and OracleString.

Oracle Sequences
Describes support for retrieving the server-generated key Oracle Sequence values.

Oracle Data Type Mappings
Lists Oracle data types and their mappings to the OracleDataReader.

Oracle Distributed Transactions

Describes how the OracleConnection object automatically enlists in an existing distributed transaction if it determines that a transaction is active.

Related Sections

Securing ADO.NET Applications

Describes secure coding practices when using ADO.NET.

DataSets, DataTables, and DataViews

Describes how to create and use `DataSets`, typed `DataSets`, `DataTables`, and `DataViews`.

Retrieving and Modifying Data in ADO.NET

Describes how to work with data in ADO.NET.

SQL Server and ADO.NET

Describes how to work with features and functionality that are specific to SQL Server.

DbProviderFactories

Describes generic classes that allow you to write provider-independent code in ADO.NET.

See also

- ADO.NET
- ADO.NET Overview

System Requirements for the .NET Framework Data Provider for Oracle

3/12/2020 • 2 minutes to read • Edit Online

The .NET Framework Data Provider for Oracle requires Microsoft Data Access Components (MDAC) version 2.6 or later. MDAC 2.8 SP1 is recommended.

You must also have Oracle 8i Release 3 (8.1.7) Client or later installed.

Oracle Client software prior to version Oracle 9i cannot access UTF16 databases because UTF16 is a new feature in Oracle 9i. To use this feature, you must upgrade your client software to Oracle 9i or later.

Working with the Data Provider for Oracle and Unicode Data

The following is a list of Unicode-related issues that you should consider when working with the .NET Framework Data Provider for Oracle and Oracle client libraries. For more information, see your Oracle documentation.

Setting the Unicode Value in a Connection String Attribute

When working with Oracle, you can use the connection string attribute

```
Unicode=True
```

to initialize the Oracle client libraries in UTF-16 mode. This causes the Oracle client libraries to accept UTF-16 (which is very similar to UCS-2) instead of multi-byte strings. This allows the Data Provider for Oracle to always work with any Oracle code page without additional translation work. This configuration only works if you are using Oracle 9i clients to communicate with an Oracle 9i database with the alternate character set of AL16UTF16. When an Oracle 9i client communicates with an Oracle 9i server, additional resources are required to convert the Unicode **CommandText** values to the appropriate multi-byte character set that the Oracle9i server uses. This can be avoided when you know that you have the safe configuration by adding `Unicode=True` to your connection string.

Mixing Versions of Oracle Client and Oracle Server

Oracle 8i clients cannot access **NCHAR**, **NVARCHAR2**, or **NCLOB** data in Oracle 9i databases when the server's national character set is specified as AL16UTF16 (the default setting for Oracle 9i). Because support for the UTF-16 character set was not introduced until Oracle 9i, Oracle 8i clients cannot read it.

Working with UTF-8 Data

To set the alternate character set, set the Registry Key HKEY_LOCAL_MACHINE\SOFTWARE\ORACLE\HOMEID\NLS_LANG to UTF8. See the Oracle Installation notes on your platform for more information. The default setting is the primary character set of the language from which you are installing the Oracle Client software. Not setting the language to match the national language character set of the database to which you are connecting will cause parameter and column bindings to send or receive data in your primary database character set, not the national character set.

OracleLob Can Only Update Full Characters.

For usability reasons, the OracleLob object inherits from the .NET Framework Stream class, and provides **ReadByte** and **WriteByte** methods. It also implements methods, such as **CopyTo** and **Erase**, that work on sections of Oracle **LOB** objects. In contrast, Oracle client software provides a number of APIs to work with character **LOB**s (**CLOB** and **NCLOB**). However, these APIs work on full characters only. Because of this difference, the Data Provider for Oracle implements support for **Read** and **ReadByte** to work with UTF-16 data in a byte-wise manner. However, the other methods of the **OracleLob** object only allow full-character operations.

See also

- Oracle and ADO.NET
- ADO.NET Overview

Oracle BFILEs

3/12/2020 • 2 minutes to read • Edit Online

The .NET Framework Data Provider for Oracle includes the OracleBFile class, which is used to work with the Oracle BFile data type.

The Oracle **BFILE** data type is an Oracle **LOB** data type that contains a reference to binary data with a maximum size of 4 gigabytes. An Oracle **BFILE** differs from other Oracle **LOB** data types in that its data is stored in a physical file in the operating system instead of on the server. Note that the **BFILE** data type provides read-only access to data.

Other characteristics of a **BFILE** data type that distinguish it from a **LOB** data type are that it:

- Contains unstructured data.
- Supports server-side chunking.
- Uses reference copy semantics. For example, if you perform a copy operation on a **BFILE**, only the **BFILE** locator (which is a reference to the file) is copied. The data in the file is not copied.

The **BFILE** data type should be used for referencing LOBs that are large in size, and therefore, not practical to store in the database. More client, server, and communication overhead is involved when using a **BFILE** data type compared with the **LOB** data type. It is more efficient to access a **BFILE** if you only need to obtain a small amount of data. It is more efficient to access database-resident LOBs if you need to obtain the entire object.

Each non-NULL **OracleBFile** object is associated with two entities that define the location of the underlying physical file:

1. An Oracle DIRECTORY object, which is a database alias for a directory in the file system, and
2. The file name of the underlying physical file, which is located in the directory associated with the DIRECTORY object.

Example

The following C# example demonstrates how you can create a **BFILE** in an Oracle table and then retrieve it in the form of an **OracleBFile** object. The example demonstrates using the OracleDataReader object and the **OracleBFile Seek** and **Read** methods. Note that in order to use this sample, you must first create a directory named "c:\\bfiles" and file named "MyFile.jpg" on the Oracle server.

```
using System;
using System.IO;
using System.Data;
using System.Data.OracleClient;

public class Sample
{
   public static void Main(string[] args)
   {
      OracleConnection connection = new OracleConnection(
        "Data Source=Oracle8i;Integrated Security=yes");
      connection.Open();

      OracleCommand command = connection.CreateCommand();
      command.CommandText =
        "CREATE or REPLACE DIRECTORY MyDir as 'c:\\bfiles'";
      command.ExecuteNonQuery();
      command.CommandText =
        "DROP TABLE MyBFileTable";
      try {
        command.ExecuteNonQuery();
      }
      catch {
      }
      command.CommandText =
        "CREATE TABLE MyBFileTable(col1 number, col2 BFILE)";
      command.ExecuteNonQuery();
      command.CommandText =
        "INSERT INTO MyBFileTable values ('2', BFILENAME('MyDir', " +
        "'MyFile.jpg'))";
      command.ExecuteNonQuery();
      command.CommandText = "SELECT * FROM MyBFileTable";

        byte[] buffer = new byte[100];

      OracleDataReader reader = command.ExecuteReader();
      using (reader) {
          if (reader.Read()) {
                OracleBFile bFile = reader.GetOracleBFile(1);
                using (bFile) {
                  bFile.Seek(0, SeekOrigin.Begin);
                  bFile.Read(buffer, 0, 100);
              }
          }
      }

      connection.Close();
   }

}
```

See also

- Oracle and ADO.NET
- ADO.NET Overview

Oracle LOBs

3/12/2020 • 5 minutes to read • Edit Online

The .NET Framework Data Provider for Oracle includes the OracleLob class, which is used to work with Oracle **LOB** data types.

An **OracleLob** may be one of these OracleType data types:

DATA TYPE	DESCRIPTION
Blob	An Oracle **BLOB** data type that contains binary data with a maximum size of 4 gigabytes. This maps to an **Array** of type **Byte**.
Clob	An Oracle **CLOB** data type that contains character data, based on the default character set on the server, with a maximum size of 4 gigabytes. This maps to **String**.
NClob	An Oracle **NCLOB** data type that contains character data, based on the national character set on the server with a maximum size of 4 gigabytes. This maps to **String**.

An **OracleLob** differs from an OracleBFile in that the data is stored on the server instead of in a physical file in the operating system. It can also be a read-write object, unlike an **OracleBFile**, which is always read-only.

Creating, Retrieving, and Writing to a LOB

The following C# example demonstrates how you can create LOBs in an Oracle table, and then retrieve and write to them in the form of **OracleLob** objects. The example demonstrates using the OracleDataReader object and the **OracleLob Read** and **Write** methods. The example uses Oracle **BLOB**, **CLOB**, and **NCLOB** data types.

```
using System;
using System.IO;
using System.Text;
using System.Data;
using System.Data.OracleClient;

// LobExample
public class LobExample
{
   public static int Main(string[] args)
   {
      //Create a connection.
      OracleConnection conn = new OracleConnection(
         "Data Source=Oracle8i;Integrated Security=yes");
      using(conn)
      {
         //Open a connection.
         conn.Open();
         OracleCommand cmd = conn.CreateCommand();

         //Create the table and schema.
         CreateTable(cmd);

         //Read example.
         ReadLobExample(cmd);
```

```
      //Write example
      WriteLobExample(cmd);
   }

   return 1;
}

// ReadLobExample
public static void ReadLobExample(OracleCommand cmd)
{
   int actual = 0;

   // Table Schema:
   // "CREATE TABLE tablewithlobs (a int, b BLOB, c CLOB, d NCLOB)";
   // "INSERT INTO tablewithlobs values (1, 'AA', 'AAA', N'AAAA')";
   // Select some data.
   cmd.CommandText = "SELECT * FROM tablewithlobs";
   OracleDataReader reader = cmd.ExecuteReader();
   using(reader)
   {
      //Obtain the first row of data.
      reader.Read();

      //Obtain the LOBs (all 3 varieties).
      OracleLob blob = reader.GetOracleLob(1);
      OracleLob clob = reader.GetOracleLob(2);
      OracleLob nclob = reader.GetOracleLob(3);

      //Example - Reading binary data (in chunks).
      byte[] buffer = new byte[100];
      while((actual = blob.Read(buffer, 0, buffer.Length)) >0)
         Console.WriteLine(blob.LobType + ".Read(" + buffer + ", " +
           buffer.Length + ") => " + actual);

      // Example - Reading CLOB/NCLOB data (in chunks).
      // Note: You can read character data as raw Unicode bytes
      // (using OracleLob.Read as in the above example).
      // However, because the OracleLob object inherits directly
      // from the .NET stream object,
      // all the existing classes that manipluate streams can
      // also be used. For example, the
      // .NET StreamReader makes it easier to convert the raw bytes
      // into actual characters.
      StreamReader streamreader =
        new StreamReader(clob, Encoding.Unicode);
      char[] cbuffer = new char[100];
      while((actual = streamreader.Read(cbuffer,
        0, cbuffer.Length)) >0)
         Console.WriteLine(clob.LobType + ".Read(
           " + new string(cbuffer, 0, actual) + ", " +
           cbuffer.Length + ") => " + actual);

      // Example - Reading data (all at once).
      // You could use StreamReader.ReadToEnd to obtain
      // all the string data, or simply
      // call OracleLob.Value to obtain a contiguous allocation
      // of all the data.
      Console.WriteLine(nclob.LobType + ".Value => " + nclob.Value);
   }
}

// WriteLobExample
public static void WriteLobExample(OracleCommand cmd)
{
   //Note: Updating LOB data requires a transaction.
   cmd.Transaction = cmd.Connection.BeginTransaction();

   // Select some data.
   // Table Schema:
```

```
   // Table Schema:
   // "CREATE TABLE tablewithlobs (a int, b BLOB, c CLOB, d NCLOB)";
   // "INSERT INTO tablewithlobs values (1, 'AA', 'AAA', N'AAAA')";
   cmd.CommandText = "SELECT * FROM tablewithlobs FOR UPDATE";
   OracleDataReader reader = cmd.ExecuteReader();
   using(reader)
   {
      // Obtain the first row of data.
      reader.Read();

      // Obtain a LOB.
      OracleLob blob = reader.GetOracleLob(1/*0:based ordinal*/);

      // Perform any desired operations on the LOB
      // (read, position, and so on).

      // Example - Writing binary data (directly to the backend).
      // To write, you can use any of the stream classes, or write
      // raw binary data using
      // the OracleLob write method. Writing character vs. binary
      // is the same;
      // however note that character is always in terms of
      // Unicode byte counts
      // (for example, even number of bytes - 2 bytes for every
      // Unicode character).
      byte[] buffer = new byte[100];
      buffer[0] = 0xCC;
      buffer[1] = 0xDD;
      blob.Write(buffer, 0, 2);
      blob.Position = 0;
      Console.WriteLine(blob.LobType + ".Write(
        " + buffer + ", 0, 2) => " + blob.Value);

      // Example - Obtaining a temp LOB and copying data
      // into it from another LOB.
      OracleLob templob = CreateTempLob(cmd, blob.LobType);
      long actual = blob.CopyTo(templob);
      Console.WriteLine(blob.LobType + ".CopyTo(
         " + templob.Value + ") => " + actual);

      // Commit the transaction now that everything succeeded.
      // Note: On error, Transaction.Dispose is called
      // (from the using statement)
      // and will automatically roll back the pending transaction.
      cmd.Transaction.Commit();
   }
}

// CreateTempLob
public static OracleLob CreateTempLob(
  OracleCommand cmd, OracleType lobtype)
{
   //Oracle server syntax to obtain a temporary LOB.
   cmd.CommandText = "DECLARE A " + lobtype + "; "+
                  "BEGIN "+
                     "DBMS_LOB.CREATETEMPORARY(A, FALSE); "+
                     ":LOC := A; "+
                  "END;";

   //Bind the LOB as an output parameter.
   OracleParameter p = cmd.Parameters.Add("LOC", lobtype);
   p.Direction = ParameterDirection.Output;

   //Execute (to receive the output temporary LOB).
   cmd.ExecuteNonQuery();

   //Return the temporary LOB.
   return (OracleLob)p.Value;
}
```

```
   // CreateTable
   public static void CreateTable(OracleCommand cmd)
   {
      // Table Schema:
      // "CREATE TABLE tablewithlobs (a int, b BLOB, c CLOB, d NCLOB)";
      // "INSERT INTO tablewithlobs VALUES (1, 'AA', 'AAA', N'AAAA')";
      try
      {
         cmd.CommandText   = "DROP TABLE tablewithlobs";
         cmd.ExecuteNonQuery();
      }
      catch(Exception)
      {
      }

      cmd.CommandText =
        "CREATE TABLE tablewithlobs (a int, b BLOB, c CLOB, d NCLOB)";
      cmd.ExecuteNonQuery();
      cmd.CommandText =
        "INSERT INTO tablewithlobs VALUES (1, 'AA', 'AAA', N'AAAA')";
      cmd.ExecuteNonQuery();
   }
}
```

Creating a Temporary LOB

The following C# example demonstrates how to create a temporary LOB.

```
OracleConnection conn = new OracleConnection(
  "server=test8172; integrated security=yes;");
conn.Open();

OracleTransaction tx = conn.BeginTransaction();

OracleCommand cmd = conn.CreateCommand();
cmd.Transaction = tx;
cmd.CommandText =
  "declare xx blob; begin dbms_lob.createtemporary(
  xx, false, 0); :tempblob := xx; end;";
cmd.Parameters.Add(new OracleParameter("tempblob",
  OracleType.Blob)).Direction = ParameterDirection.Output;
cmd.ExecuteNonQuery();
OracleLob tempLob = (OracleLob)cmd.Parameters[0].Value;
tempLob.BeginBatch(OracleLobOpenMode.ReadWrite);
tempLob.Write(tempbuff,0,tempbuff.Length);
tempLob.EndBatch();
cmd.Parameters.Clear();
cmd.CommandText = "myTable.myProc";
cmd.CommandType = CommandType.StoredProcedure;
cmd.Parameters.Add(new OracleParameter(
  "ImportDoc", OracleType.Blob)).Value = tempLob;
cmd.ExecuteNonQuery();

tx.Commit();
```

See also

- Oracle and ADO.NET
- ADO.NET Overview

Oracle REF CURSORs

9/7/2019 • 2 minutes to read • Edit Online

The .NET Framework Data Provider for Oracle supports the Oracle **REF CURSOR** data type. When using the data provider to work with Oracle REF CURSORs, you should consider the following behaviors.

> **NOTE**
> Some behaviors differ from those of the Microsoft OLE DB Provider for Oracle (MSDAORA).

- For performance reasons, the Data Provider for Oracle does not automatically bind **REF CURSOR** data types, as MSDAORA does, unless you explicitly specify them.
- The data provider does not support any ODBC escape sequences, including the {resultset} escape used to specify REF CURSOR parameters.
- To execute a stored procedure that returns REF CURSORs, you must define the parameters in the OracleParameterCollection with an OracleType of **Cursor** and a Direction of **Output**. The data provider supports binding REF CURSORs as output parameters only. The provider does not support REF CURSORs as input parameters.
- Obtaining an OracleDataReader from the parameter value is not supported. The values are of type DBNull after command execution.
- The only **CommandBehavior** enumeration value that works with REF CURSORs (for example, when calling ExecuteReader) is **CloseConnection**; all others are ignored.
- The order of REF CURSORs in the **OracleDataReader** depends on the order of the parameters in the **OracleParameterCollection**. The ParameterName property is ignored.
- The PL/SQL **TABLE** data type is not supported. However, REF CURSORs are more efficient. If you must use a **TABLE** data type, use the OLE DB .NET Data Provider with MSDAORA.

In This Section

REF CURSOR Examples
Contains three examples that demonstrate using REF CURSORs.

REF CURSOR Parameters in an OracleDataReader
Demonstrates how to execute a PL/SQL stored procedure that returns a REF CURSOR parameter, and reads the value as an **OracleDataReader**.

Retrieving Data from Multiple REF CURSORs Using an OracleDataReader
Demonstrates how to execute a PL/SQL stored procedure that returns two REF CURSOR parameters, and reads the values using an **OracleDataReader**.

Filling a DataSet Using One or More REF CURSORs
Demonstrates how to execute a PL/SQL stored procedure that returns two REF CURSOR parameters, and fills a DataSet with the rows that are returned.

See also

- Oracle and ADO.NET

- ADO.NET Overview

REF CURSOR Examples

3/12/2020 • 2 minutes to read • Edit Online

The REF CURSOR examples are comprised of the following three Microsoft Visual Basic examples that demonstrate using REF CURSORs.

SAMPLE	DESCRIPTION
REF CURSOR Parameters in an OracleDataReader	This example executes a PL/SQL stored procedure that returns a REF CURSOR parameter, and reads the value as an OracleDataReader.
Retrieving Data from Multiple REF CURSORs Using an OracleDataReader	This example executes a PL/SQL stored procedure that returns two REF CURSOR parameters, and reads the values using an **OracleDataReader**.
Filling a DataSet Using One or More REF CURSORs	This example executes a PL/SQL stored procedure that returns two REF CURSOR parameters, and fills a DataSet with the rows that are returned.

To use these examples, you may need to create the Oracle tables, and you must create a PL/SQL package and package body.

Creating the Oracle Tables

These examples use tables that are defined in the Oracle Scott/Tiger schema. The Oracle Scott/Tiger schema is included with most Oracle installations. If this schema does not exist, you can use the SQL commands file in {OracleHome}\rdbms\admin\scott.sql to create the tables and indexes used by these examples.

Creating the Oracle Package and Package Body

These examples require the following PL/SQL package and package body on your server. Create the following Oracle package on the Oracle server.

```
CREATE OR REPLACE PACKAGE CURSPKG AS
    TYPE T_CURSOR IS REF CURSOR;
    PROCEDURE OPEN_ONE_CURSOR (N_EMPNO IN NUMBER,
                               IO_CURSOR IN OUT T_CURSOR);
    PROCEDURE OPEN_TWO_CURSORS (EMPCURSOR OUT T_CURSOR,
                                DEPTCURSOR OUT T_CURSOR);
END CURSPKG;
/
```

Create the following Oracle package body on the Oracle server.

```
CREATE OR REPLACE PACKAGE BODY CURSPKG AS
    PROCEDURE OPEN_ONE_CURSOR (N_EMPNO IN NUMBER,
                               IO_CURSOR IN OUT T_CURSOR)
    IS
        V_CURSOR T_CURSOR;
    BEGIN
        IF N_EMPNO <> 0
        THEN
             OPEN V_CURSOR FOR
             SELECT EMP.EMPNO, EMP.ENAME, DEPT.DEPTNO, DEPT.DNAME
                  FROM EMP, DEPT
                  WHERE EMP.DEPTNO = DEPT.DEPTNO
                  AND EMP.EMPNO = N_EMPNO;

        ELSE
             OPEN V_CURSOR FOR
             SELECT EMP.EMPNO, EMP.ENAME, DEPT.DEPTNO, DEPT.DNAME
                  FROM EMP, DEPT
                  WHERE EMP.DEPTNO = DEPT.DEPTNO;

        END IF;
        IO_CURSOR := V_CURSOR;
    END OPEN_ONE_CURSOR;

    PROCEDURE OPEN_TWO_CURSORS (EMPCURSOR OUT T_CURSOR,
                                DEPTCURSOR OUT T_CURSOR)
    IS
        V_CURSOR1 T_CURSOR;
        V_CURSOR2 T_CURSOR;
    BEGIN
        OPEN V_CURSOR1 FOR SELECT * FROM EMP;
        OPEN V_CURSOR2 FOR SELECT * FROM DEPT;
        EMPCURSOR  := V_CURSOR1;
        DEPTCURSOR := V_CURSOR2;
    END OPEN_TWO_CURSORS;
END CURSPKG;
/
```

See also

- Oracle REF CURSORs
- ADO.NET Overview

REF CURSOR Parameters in an OracleDataReader

9/7/2019 • 2 minutes to read • Edit Online

This Microsoft Visual Basic example executes a PL/SQL stored procedure that returns a REF CURSOR parameter, and reads the value as an OracleDataReader.

```
Private Sub Button1_Click(ByVal sender As Object, _
  ByVal e As System.EventArgs) Handles Button1.Click

  Dim connString As New String(_
      "Data Source=Oracle9i;User ID=scott;Password=tiger;")
  Using conn As New OracleConnection(connString)
    Dim cmd As New OracleCommand()
    Dim rdr As OracleDataReader

    conn.Open()
    cmd.Connection = conn
    cmd.CommandText = "CURSPKG.OPEN_ONE_CURSOR"
    cmd.CommandType = CommandType.StoredProcedure
    cmd.Parameters.Add(New OracleParameter(
      "N_EMPNO", OracleType.Number)).Value = 7369
    cmd.Parameters.Add(New OracleParameter(
      "IO_CURSOR", OracleType.Cursor)).Direction = ParameterDirection.Output

    rdr = cmd.ExecuteReader()
    While (rdr.Read())
        REM do something with the values
    End While

    rdr.Close()
  End Using
End Sub
```

See also

- Oracle REF CURSORs
- ADO.NET Overview

Retrieving Data from Multiple REF CURSORs Using an OracleDataReader

3/12/2020 • 2 minutes to read • Edit Online

This Microsoft Visual Basic example executes a PL/SQL stored procedure that returns two REF CURSOR parameters, and reads the values using an OracleDataReader.

```
Private Sub Button1_Click( _
  ByVal sender As Object, ByVal e As System.EventArgs)  _
  Handles Button1.Click

  Dim connString As New String( _
      "Data Source=Oracle9i;User ID=scott;Password=tiger;")
  Using conn As New OracleConnection(connString)
    Dim cmd As New OracleCommand()
    Dim rdr As OracleDataReader

    conn.Open()
    cmd.Connection = conn
    cmd.CommandText = "CURSPKG.OPEN_TWO_CURSORS"
    cmd.CommandType = CommandType.StoredProcedure
    cmd.Parameters.Add(New OracleParameter( _
      "EMPCURSOR", OracleType.Cursor)).Direction = _
      ParameterDirection.Output
    cmd.Parameters.Add(New OracleParameter(_
      "DEPTCURSOR", OracleType.Cursor)).Direction = _
      ParameterDirection.Output

    rdr = cmd.ExecuteReader(CommandBehavior.CloseConnection)
    While (rdr.Read())
        REM do something with the values from the EMP table
    End While

    rdr.NextResult()
    While (rdr.Read())
        REM do something with the values from the DEPT table
    End While
    rdr.Close()
  End Using
End Sub
```

See also

- Oracle REF CURSORs
- ADO.NET Overview

Filling a DataSet Using One or More REF CURSORs

9/7/2019 • 2 minutes to read • Edit Online

This Microsoft Visual Basic example executes a PL/SQL stored procedure that returns two REF CURSOR parameters, and fills a DataSet with the rows that are returned.

```
Private Sub Button1_Click(ByVal sender As Object, _
  ByVal e As System.EventArgs) Handles Button1.Click

  Dim connString As New String(_
    "Data Source=Oracle9i;User ID=scott;Password=tiger;")
  Dim ds As New DataSet()
    Using conn As New OracleConnection(connString)
    Dim cmd As New OracleCommand()

    cmd.Connection = conn
    cmd.CommandText = "CURSPKG.OPEN_TWO_CURSORS"
    cmd.CommandType = CommandType.StoredProcedure
    cmd.Parameters.Add(New OracleParameter( _
      "EMPCURSOR", OracleType.Cursor)).Direction = _
      ParameterDirection.Output
    cmd.Parameters.Add(New OracleParameter( _
      "DEPTCURSOR", OracleType.Cursor)).Direction = _
       ParameterDirection.Output

    Dim da As New OracleDataAdapter(cmd)
    da.TableMappings.Add("Table", "Emp")
    da.TableMappings.Add("Table1", "Dept")
    da.Fill(ds)

    ds.Relations.Add("EmpDept", ds.Tables("Dept").Columns("Deptno"), _
      ds.Tables("Emp").Columns("Deptno"), False)

    DataGrid1.DataSource = ds.Tables("Dept")
  End Using
```

See also

- Oracle REF CURSORs
- ADO.NET Overview

OracleTypes

3/12/2020 • 2 minutes to read • Edit Online

The .NET Framework Data Provider for Oracle includes several structures you can use to work with Oracle data types. These include OracleNumber and OracleString.

> **NOTE**
> For a complete list of these structures, see System.Data.OracleClient.

The following C# examples:

- Create an Oracle table and load it with data.
- Use an OracleDataReader to access the data, and use several OracleType structures to display the data.

Creating an Oracle Table

This example creates an Oracle table and loads it with data. You must run this example before running the next example.

```
public void Setup(string connectionString)
   {
   OracleConnection conn = new OracleConnection(connectionString);
   try
      {
      conn.Open();
      OracleCommand cmd = conn.CreateCommand();
      cmd.CommandText ="CREATE TABLE OracleTypesTable " +
        "(MyVarchar2 varchar2(3000),MyNumber number(28,4) " +
        "PRIMARY KEY ,MyDate date, MyRaw raw(255))";
      cmd.ExecuteNonQuery();
      cmd.CommandText ="INSERT INTO OracleTypesTable VALUES " +
        "( 'test', 2, to_date('2000-01-11 12:54:01','yyyy-mm-dd " +
        "hh24:mi:ss'), '0001020304' )";
      cmd.ExecuteNonQuery();
      }
   catch(Exception)
   {
   }
   finally
   {
      conn.Close();
   }
}
```

Retrieving Data from the Oracle Table

This example uses an **OracleDataReader** to access the data, and uses several **OracleType** structures to display the data.

```
public void ReadOracleTypesExample(string connectionString)
   {
   OracleConnection myConnection =
      new OracleConnection(connectionString);
   myConnection.Open();
   OracleCommand myCommand = myConnection.CreateCommand();

   try
      {
      myCommand.CommandText = "SELECT * from OracleTypesTable";
      OracleDataReader oracledatareader1 = myCommand.ExecuteReader();
      oracledatareader1.Read();

      //Using the oracle specific getters for each type is faster than
      //using GetOracleValue.

      //First column, MyVarchar2, is a VARCHAR2 data type in Oracle
      //Server and maps to OracleString.
      OracleString oraclestring1 =
        oracledatareader1.GetOracleString(0);
      Console.WriteLine("OracleString " + oraclestring1.ToString());

      //Second column, MyNumber, is a NUMBER data type in Oracle Server
      //and maps to OracleNumber.
      OracleNumber oraclenumber1 =
        oracledatareader1.GetOracleNumber(1);
      Console.WriteLine("OracleNumber " + oraclenumber1.ToString());

      //Third column, MyDate, is a DATA data type in Oracle Server
      //and maps to OracleDateTime.
      OracleDateTime oracledatetime1 =
        oracledatareader1.GetOracleDateTime(2);
      Console.WriteLine("OracleDateTime " + oracledatetime1.ToString());

      //Fourth column, MyRaw, is a RAW data type in Oracle Server and
      //maps to OracleBinary.
      OracleBinary oraclebinary1 =
        oracledatareader1.GetOracleBinary(3);
      //Calling value on a null OracleBinary throws
      //OracleNullValueException; therefore, check for a null value.
      if (oraclebinary1.IsNull==false)
      {
         foreach(byte b in oraclebinary1.Value)
         {
            Console.WriteLine("byte " + b.ToString());
         }
      }
      oracledatareader1.Close();
   }
   catch(Exception e)
   {
       Console.WriteLine(e.ToString());
   }
   finally
   {
       myConnection.Close();
   }
}
```

See also

- Oracle and ADO.NET
- ADO.NET Overview

Oracle Sequences

3/12/2020 • 3 minutes to read • Edit Online

The .NET Framework Data Provider for Oracle provides support for retrieving the server-generated key Oracle Sequence values after performing inserts by using the OracleDataAdapter.

SQL Server and Oracle support the creation of automatically incrementing columns that can be designated as primary keys. These values are generated by the server as rows are added to a table. In SQL Server, you set the Identity property of a column; in Oracle you create a Sequence. The difference between auto-increment columns in SQL Server and sequences in Oracle is that:

- In SQL Server, you mark a column as an auto-increment column and SQL Server automatically generates new values for the column when you insert a new row.
- In Oracle, you create a sequence to generate new values for a column in your table, but there is no direct link between the sequence and the table or column. An Oracle sequence is an object, like a table or a stored procedure.

When you create a sequence in an Oracle database, you can define its initial value and the increment between its values. You can also query the sequence for new values before submitting new rows. That means your code can recognize the key values for new rows before you insert them into the database.

For more information about creating auto-increment columns by using SQL Server and ADO.NET, see Retrieving Identity or Autonumber Values and Creating AutoIncrement Columns.

Example

The following C# example demonstrates how you can retrieve new sequence values from Oracle database. The example references the sequence in the INSERT INTO query used to submit the new rows, and then returns the sequence value generated using the RETURNING clause introduced in Oracle10g. The example adds a series of pending new rows in a DataTable by using ADO.NET's auto-increment functionality to generate "placeholder" primary key values. Note that the increment value ADO.NET generated for the new row is just a "placeholder". That means the database might generate different values from the ones ADO.NET generates.

Before submitting the pending inserts to the database, the example displays the contents of the rows. Then, the code creates a new OracleDataAdapter object and sets its InsertCommand and the UpdateBatchSize properties. The example also supplies the logic to return the server-generated values by using output parameters. Then, the example executes the update to submit the pending rows and displays the contents of the DataTable.

```
public void OracleSequence(String connectionString)
{
   String insertString =
      "INSERT INTO SequenceTest_Table (ID, OtherColumn)" +
      "VALUES (SequenceTest_Sequence.NEXTVAL, :OtherColumn)" +
      "RETURNING ID INTO :ID";

   using (OracleConnection conn = new OracleConnection(connectionString))
   {
      //Open a connection.
      conn.Open();
      OracleCommand cmd = conn.CreateCommand();

      // Prepare the database.
      cmd.CommandText = "DROP SEQUENCE SequenceTest_Sequence";
      try { cmd.ExecuteNonQuery(); } catch { }
```

```
        cmd.CommandText = "DROP TABLE SequenceTest_Table";
        try { cmd.ExecuteNonQuery(); } catch { }

        cmd.CommandText = "CREATE TABLE SequenceTest_Table " +
                      "(ID int PRIMARY KEY, OtherColumn varchar(255))";
        cmd.ExecuteNonQuery();

        cmd.CommandText = "CREATE SEQUENCE SequenceTest_Sequence " +
                          "START WITH 100 INCREMENT BY 5";
        cmd.ExecuteNonQuery();

        DataTable testTable = new DataTable();
        DataColumn column = testTable.Columns.Add("ID", typeof(int));
        column.AutoIncrement = true;
        column.AutoIncrementSeed = -1;
        column.AutoIncrementStep = -1;
        testTable.PrimaryKey = new DataColumn[] { column };
        testTable.Columns.Add("OtherColumn", typeof(string));
        for (int rowCounter = 1; rowCounter <= 15; rowCounter++)
        {
           testTable.Rows.Add(null, "Row #" + rowCounter.ToString());
        }

        Console.WriteLine("Before Update => ");
        foreach (DataRow row in testTable.Rows)
        {
           Console.WriteLine("   {0} - {1}", row["ID"], row["OtherColumn"]);
        }
        Console.WriteLine();

        cmd.CommandText =
          "SELECT ID, OtherColumn FROM SequenceTest_Table";
        OracleDataAdapter da = new OracleDataAdapter(cmd);
        da.InsertCommand = new OracleCommand(insertString, conn);
        da.InsertCommand.Parameters.Add(":ID", OracleType.Int32, 0, "ID");
        da.InsertCommand.Parameters[0].Direction = ParameterDirection.Output;
        da.InsertCommand.Parameters.Add(":OtherColumn", OracleType.VarChar, 255, "OtherColumn");
        da.InsertCommand.UpdatedRowSource = UpdateRowSource.OutputParameters;
        da.UpdateBatchSize = 10;

        da.Update(testTable);

        Console.WriteLine("After Update => ");
        foreach (DataRow row in testTable.Rows)
        {
           Console.WriteLine("   {0} - {1}", row["ID"], row["OtherColumn"]);
        }
        // Close the connection.
        conn.Close();
    }
}
```

See also

- Oracle and ADO.NET
- ADO.NET Overview

Oracle Data Type Mappings

9/7/2019 • 3 minutes to read • Edit Online

The following table lists Oracle data types and their mappings to the OracleDataReader.

ORACLE DATA TYPE	.NET FRAMEWORK DATA TYPE RETURNED BY ORACLEDATAREADER.GETVALUE	ORACLECLIENT DATA TYPE RETURNED BY ORACLEDATAREADER.GETORACLEVALUE	REMARKS
BFILE	Byte[]	OracleBFile	
BLOB	Byte[]	OracleLob	
CHAR	String	OracleString	
CLOB	String	OracleLob	
DATE	DateTime	OracleDateTime	
FLOAT	Decimal	OracleNumber	This data type is an alias for the **NUMBER** data type, and is designed so that the OracleDataReader returns a **System.Decimal** or OracleNumber instead of a floating-point value. Using the .NET Framework data type can cause an overflow.
INTEGER	Decimal	OracleNumber	This data type is an alias for the **NUMBER(38)** data type, and is designed so that the OracleDataReader returns a **System.Decimal** or OracleNumber instead of an integer value. Using the .NET Framework data type can cause an overflow.
INTERVAL YEAR TO MONTH	Int32	OracleMonthSpan	
INTERVAL DAY TO SECOND	TimeSpan	OracleTimeSpan	
LONG	String	OracleString	
LONG RAW	Byte[]	OracleBinary	
NCHAR	String	OracleString	
NCLOB	String	OracleLob	

ORACLE DATA TYPE	.NET FRAMEWORK DATA TYPE RETURNED BY ORACLEDATAREADER.GETVALUE	ORACLECLIENT DATA TYPE RETURNED BY ORACLEDATAREADER.GETORACLEVALUE	REMARKS
NUMBER	Decimal	OracleNumber	Using the .NET Framework data type can cause an overflow.
NVARCHAR2	String	OracleString	
RAW	Byte[]	OracleBinary	
REF CURSOR			The Oracle **REF CURSOR** data type is not supported by the OracleDataReader object.
ROWID	String	OracleString	
TIMESTAMP	DateTime	OracleDateTime	
TIMESTAMP WITH LOCAL TIME ZONE	DateTime	OracleDateTime	
TIMESTAMP WITH TIME ZONE	DateTime	OracleDateTime	
UNSIGNED INTEGER	Number	OracleNumber	This data type is an alias for the **NUMBER(38)** data type, and is designed so that the OracleDataReader returns a **System.Decimal** or OracleNumber instead of an unsigned integer value. Using the .NET Framework data type can cause an overflow.
VARCHAR2	String	OracleString	

The following table lists Oracle data types and the .NET Framework data types (**System.Data.DbType** and OracleType) to use when binding them as parameters.

ORACLE DATA TYPE	DBTYPE ENUMERATION TO BIND AS A PARAMETER	ORACLETYPE ENUMERATION TO BIND AS A PARAMETER	REMARKS
BFILE		BFile	Oracle only allows binding a **BFILE** as a **BFILE** parameter. The .NET Data Provider for Oracle does not automatically construct one for you if you attempt to bind a non-**BFILE** value, such as **byte[]** or OracleBinary.

ORACLE DATA TYPE	DBTYPE ENUMERATION TO BIND AS A PARAMETER	ORACLETYPE ENUMERATION TO BIND AS A PARAMETER	REMARKS
BLOB		Blob	Oracle only allows binding a **BLOB** as a **BLOB** parameter. The .NET Data Provider for Oracle does not automatically construct one for you if you attempt to bind a non-**BLOB** value, such as **byte[]** or OracleBinary.
CHAR	AnsiStringFixedLength	Char	
CLOB		Clob	Oracle only allows binding a **CLOB** as a **CLOB** parameter. The .NET Data Provider for Oracle does not automatically construct one for you if you attempt to bind a non-**CLOB** value, such as **System.String** or OracleString.
DATE	DateTime	DateTime	
FLOAT	Single, Double, Decimal	Float, Double, Number	Size determines the **System.Data.DBType** and OracleType.
INTEGER	SByte, Int16, Int32, Int64, Decimal	SByte, Int16, Int32, Number	Size determines the **System.Data.DBType** and OracleType.
INTERVAL YEAR TO MONTH	Int32	IntervalYearToMonth	OracleType is only available when using both Oracle 9i client and server software.
INTERVAL DAY TO SECOND	Object	IntervalDayToSecond	OracleType is only available when using both Oracle 9i client and server software.
LONG	AnsiString	LongVarChar	
LONG RAW	Binary	LongRaw	
NCHAR	StringFixedLength	NChar	
NCLOB		NClob	Oracle only allows binding a **NCLOB** as a **NCLOB** parameter. The .NET Data Provider for Oracle does not automatically construct one for you if you attempt to bind a non-**NCLOB** value, such as **System.String** or OracleString.

ORACLE DATA TYPE	DBTYPE ENUMERATION TO BIND AS A PARAMETER	ORACLETYPE ENUMERATION TO BIND AS A PARAMETER	REMARKS
NUMBER	VarNumeric	Number	
NVARCHAR2	String	NVarChar	
RAW	Binary	Raw	
REF CURSOR		Cursor	For more information, see Oracle REF CURSORs.
ROWID	AnsiString	Rowid	
TIMESTAMP	DateTime	Timestamp	OracleType is only available when using both Oracle 9i client and server software.
TIMESTAMP WITH LOCAL TIME ZONE	DateTime	TimestampLocal	OracleType is only available when using both Oracle 9i client and server software.
TIMESTAMP WITH TIME ZONE	DateTime	TimestampWithTz	OracleType is only available when using both Oracle 9i client and server software.
UNSIGNED INTEGER	Byte, UInt16, UInt32, UInt64, Decimal	Byte, UInt16, Uint32, Number	Size determines the **System.Data.DBType** and OracleType.
VARCHAR2	AnsiString	VarChar	

The **InputOutput**, **Output**, and **ReturnValue ParameterDirection** values used by the Value property of the OracleParameter object are .NET Framework data types, unless the input value is an Oracle data type (for example, OracleNumber or OracleString). This does not apply to **REF CURSOR**, **BFILE**, or **LOB** data types.

See also

- Oracle and ADO.NET
- ADO.NET Overview

Oracle Distributed Transactions

10/29/2019 • 2 minutes to read • Edit Online

The OracleConnection object automatically enlists in an existing distributed transaction if it determines that a transaction is active. Automatic transaction enlistment occurs when the connection is opened or retrieved from the connection pool. You can disable auto-enlistment in existing transactions by specifying

```
Enlist=false
```

as a connection string parameter for an OracleConnection.

See also

- Oracle and ADO.NET
- ADO.NET Overview

ADO.NET Entity Framework

2/7/2019 • 2 minutes to read • Edit Online

The docs.microsoft.com/ef/ site is now the main location for the Entity Framework content.

The content for this topic is now available on the following page: Introducing Entity Framework.

SQL Server and ADO.NET

2/4/2020 • 2 minutes to read • Edit Online

This section describes features and behaviors that are specific to the .NET Framework Data Provider for SQL Server (System.Data.SqlClient).

System.Data.SqlClient provides access to versions of SQL Server, which encapsulates database-specific protocols. The functionality of the data provider is designed to be similar to that of the .NET Framework data providers for OLE DB, ODBC, and Oracle. System.Data.SqlClient includes a tabular data stream (TDS) parser to communicate directly with SQL Server.

> **NOTE**
> To use the .NET Framework Data Provider for SQL Server, an application must reference the System.Data.SqlClient namespace.

In This Section

SQL Server Security
Provides an overview of SQL Server security features, and application scenarios for creating secure ADO.NET applications that target SQL Server.

SQL Server Data Types and ADO.NET
Describes how to work with SQL Server data types and how they interact with .NET Framework data types.

SQL Server Binary and Large-Value Data
Describes how to work with large value data in SQL Server.

SQL Server Data Operations in ADO.NET
Describes how to work with data in SQL Server. Contains sections about bulk copy operations, MARS, asynchronous operations, and table-valued parameters.

SQL Server Features and ADO.NET
Describes SQL Server features that are useful for ADO.NET application developers.

LINQ to SQL
Describes the basic building blocks, processes, and techniques required for creating LINQ to SQL applications.

For complete documentation of the SQL Server Database Engine, see SQL Server Books Online for the version of SQL Server you are using.

SQL Server Books Online

See also

- Securing ADO.NET Applications
- Data Type Mappings in ADO.NET
- DataSets, DataTables, and DataViews
- Retrieving and Modifying Data in ADO.NET
- ADO.NET Overview

DataSets, DataTables, and DataViews

9/7/2019 • 2 minutes to read • Edit Online

The ADO.NET DataSet is a memory-resident representation of data that provides a consistent relational programming model regardless of the source of the data it contains. A DataSet represents a complete set of data including the tables that contain, order, and constrain the data, as well as the relationships between the tables.

There are several ways of working with a DataSet, which can be applied independently or in combination. You can:

- Programmatically create a DataTable, DataRelation, and Constraint within a DataSet and populate the tables with data.
- Populate the DataSet with tables of data from an existing relational data source using a `DataAdapter`.
- Load and persist the DataSet contents using XML. For more information, see Using XML in a DataSet.

A strongly typed DataSet can also be transported using an XML Web service. The design of the DataSet makes it ideal for transporting data using XML Web services. For an overview of XML Web services, see XML Web Services Overview. For an example of consuming a DataSet from an XML Web service, see Consuming a DataSet from an XML Web Service.

In This Section

Creating a DataSet
Describes the syntax for creating an instance of a DataSet.

Adding a DataTable to a DataSet
Describes how to create and add tables and columns to a DataSet.

Adding DataRelations
Describes how to create relations between tables in a DataSet.

Navigating DataRelations
Describes how to use the relations between tables in a DataSet to return the child or parent rows of a parent-child relationship.

Merging DataSet Contents
Describes how to merge the contents of one DataSet, DataTable, or DataRow array into another DataSet.

Copying DataSet Contents
Describes how to create a copy of a DataSet that can contain schema as well as specified data.

Handling DataSet Events
Describes the events of a DataSet and how to use them.

Typed DataSets
Discusses what a typed DataSet is and how to create and use it.

DataTables
Describes how to create a DataTable, define the schema, and manipulate data.

DataTableReaders
Describes how to create and use a DataTableReader.

DataViews

Describes how to create and work with `DataViews` and work with DataView events.

Using XML in a DataSet

Describes how the DataSet interacts with XML as a data source, including loading and persisting the contents of a DataSet as XML data.

Consuming a DataSet from an XML Web Service

Describes how to create an XML Web service that uses a DataSet to transport data.

Related Sections

What's New in ADO.NET

Introduces features that are new in ADO.NET.

ADO.NET Overview

Provides an introduction to the design and components of ADO.NET.

Populating a DataSet from a DataAdapter

Describes how to load a **DataSet** with data from a data source.

Updating Data Sources with DataAdapters

Describes how to resolve changes to the data in a **DataSet** back to the data source.

Adding Existing Constraints to a DataSet

Describes how to populate a **DataSet** with primary key information from a data source.

See also

- ADO.NET
- ADO.NET Overview