ADO NET for Programmers

Source	http://www.dotnetspider.com/kb/Article1830.aspx

ADO.NET for the ADO Programmer

Summary: This article presents an overview of the data access architecture available through ADO.NET, and answers the questions mostly commonly asked by developers familiar with ADO.

Introduction

In the .NET Framework, Microsoft introduces ADO.NET, an evolution of the data access architecture provided by the Microsoft® ActiveX® Data Objects (ADO) programming model. ADO.NET does not replace ADO for the COM programmer; rather, it provides the .NET programmer with access to relational data sources, XML, and application data. ADO.NET supports a variety of development needs, including the creation of database clients and middle-tier business objects used by applications, tools, languages, and Internet browsers. Built with the ADO programmer in mind, ADO.NET contains many similarities to ADO, and also introduces several new concepts in its design. This article addresses the most common questions the ADO programmer encounters when first examining ADO.NET.

Overview

From its inception, ADO has provided an efficient, robust interface for COM programmers to work with data. ADO is widely used as an interface to a variety of different stores because it can be called from any automation language including Microsoft Visual Basic® 6.0, Microsoft

Visual C++®, and a variety of scripting interfaces.
ADO.NET is an evolution of ADO that provides better platform interoperability and scalable data access. Creating a new set of data access APIs in ADO.NET offered the following advantages over directly porting ADO to the .NET Framework.
Improved integration with XML: Subsequent to the design of ADO, XML began to play an increasingly significant role in the design of applications. ADO.NET was designed from the ground up to integrate with XML, and leverages XML in a fundamental way. In addition to persisting and loading both data and its relational structure as XML, ADO.NET relies on XML for remoting data between tiers or clients. The generic XML representation that ADO.NET uses provides a convenient method for transmitting data across any network, including those with restrictive security perimeters. ADO.NET also uses XML tools to perform validation, hierarchical queries, and data transformations on relational data.
Integration with the .NET Framework: ADO constructs, such as the Recordset, do not employ familiar programming constructs but instead are modeled to be database-oriented. For example, cursors, which are used to navigate and retrieve data, function differently than other data constructs such as arrays and collections. In ADO.NET, however, in-memory data can be exposed through common .NET Framework structures, including arrays and collections, providing you with common access methods when working with relational data.
Improved support for the disconnected business model: ADO provides limited support for disconnected access using the Recordset. ADO.NET introduces a new object, the DataSet, which serves as a common, in-memory representation of relational data. The DataSet is, by design, disconnected at all times. Because it holds no persistent connection to outside resources, it is ideal for packaging, exchanging, caching, persisting, and loading data.
Explicit control of data access behaviors: The design of ADO includes implicit behaviors that may not always be required in an application and that may therefore limit performance. ADO.NET provides well-defined, factored components with predictable behavior, performance, and semantics that enable you to address common scenarios in a highly optimized manner.
Improved design-time support: ADO derives information about data implicitly at run time, based on metadata that is often expensive to obtain. ADO.NET, on the other hand, leverages known metadata at design time in order to provide better run-time performance and more consistent run-time behavior.
The Design of ADO
To better understand the model and design of ADO.NET, it is helpful to review some of the core aspects of ADO.
ADO uses a single object, the Recordset, as a common representation for working with all types of data. The Recordset is used for working with a forward-only stream of results from a database, for scrolling through data held on a server, or for scrolling through a set of cached results. Changes made to data may be applied immediately to the database, or applied as a batch using optimistic search and update operations. You specify the desired functionality when you create the Recordset, and the behavior of the resulting Recordset can vary greatly depending on the properties you request.
Because ADO uses a single object that can behave in many different ways, it enables you to keep the object model of your applications very simple. However, it is difficult to write common, predictable, and optimized code because the behavior, performance, and semantics exhibited by that single object can vary greatly depending on how the object is generated and what data it is accessing. This is particularly true for generic components (such as a grid control) that attempt to consume data not generated by the component and for which the component has no ability to specify required behavior or functionality.
ADO.NET: Explicit and Factored
In designing ADO.NET, consideration was given to the tasks that developers commonly face when accessing and working with data. Rather than using a single object to perform a number of tasks, ADO.NET factors specific functionality into explicit objects that are

optimized to enable developers to accomplish each task.
The functionality that the ADO Recordset provides has been factored into the following explicit objects in ADO.NET: the DataReader, which provides fast, forward-only, read-only access to query results; the DataSet, which provides an in-memory relational representation of data; and the DataAdapter, which provides a bridge between the DataSet and the data source. The ADO.NET Command object also includes explicit functionality such as the ExecuteNonQuery method for commands that do not return rows, and the ExecuteScalar method for queries that return a single value rather than a row set.
To better understand how the design of ADO.NET is made up of objects that are optimized to perform explicit behavior, consider the following tasks that are common when working with data.
Forward-Only Read-Only Data Streams
Applications, particularly middle-tier applications, often process a series of results programmatically, requiring no user interaction and no updating of or scrolling back through the results as they are read. In ADO, this type of data retrieval is performed using a Recordset with a forward-only cursor and a read-only lock. In ADO.NET, however, the DataReader object optimizes this type of data retrieval by providing a non-buffered, forward-only, read-only stream that provides the most efficient mechanism for retrieving results from the database. Much of this efficiency is gained as a result of the DataReader having been designed solely for this purpose, without having to support scenarios where data is updated at the data source or cached locally as with the ADO Recordset.
Returning a Single Value
Often the only data to be retrieved from a database is a single value (for example, an account balance). In ADO, you perform this type of data retrieval by creating a Recordset object, reading through the results, retrieving the single value, and then closing the Recordset. In ADO.NET, however, the Command object supports this function through the ExecuteScalar method, which returns the single value from the database without having to introduce an additional object to hold the results.
Disconnected Access to Data
A frequent case for exposing data is a representation in which a user can navigate the data in an ad-hoc manner without holding locks or tying up resources on the server. Some examples of this scenario are binding data to a control or combining data from multiple data sources and/or XML. The ADO Recordset provides some support for these scenarios, using a client-side cursor location. However, in ADO.NET the DataSet is explicitly designed for such tasks.
The DataSet provides a common, completely disconnected data representation that can hold results from a variety of different sources. Because the DataSet is completely independent of the data source, it provides the same performance and semantics regardless of whether the data is loaded from a database, loaded from XML, or is generated by the application. A single DataSet may contain tables populated from several different databases and other non-database sources; to the consumer of the DataSet it all looks and behaves exactly the same. Within the DataSet you can define relations to navigate from a table populated from one database (for example, "Customers"), to a related table populated from an entirely different database (for example, "Orders"), and from there to a third table (for example, "OrderDetails") containing values loaded from XML. The relational capabilities of the DataSet provide an advantage over the Recordset, which is limited to exposing the results from multiple tables either as a single joined result, or by returning multiple distinct result sets, requiring the developer to handle and relate the results manually. Though the Recordset has the ability to return and navigate hierarchical results (using the MSDataShape provider), the DataSet provides much greater flexibility when dealing with related result sets. The DataSet also provides the ability to transmit results to and from a remote client or server in an open XML format, with the schema defined using the XML Schema definition language (XSD).
Retrieving and Updating Data from a Data Source

Based on customer feedback and common use cases it is clear that in most application development scenarios (with the exception of ad-hoc tools and generic data components) the developer knows certain things about the data at design time that technologies like ADO attempt to derive at run time. For example, in most middle-tier applications the developer knows, at the time of application development, the type of database to be accessed, what queries will be executed, and how the results will be returned. ADO.NET gives you the ability to apply this knowledge at design time in order to provide better run-time performance and predictability.

As an example, when using batch updating with ADO Recordset objects, you must submit changes to the database by executing appropriate INSERT, UPDATE, and DELETE statements for each row that has changed. ADO generates these statements implicitly, at run time, based on metadata that is often expensive to obtain. ADO.NET, however, enables you to explicitly specify INSERT, UPDATE, and DELETE commands, as well as custom business logic such as a stored procedure, that will be used to resolve changes in a DataSet back to the data source using the DataAdapter. This model provides you with greater control over how application data is returned and updated, and removes the expense of gathering the metadata at run time.

The DataAdapter provides the bridge between the DataSet and the data source. A DataAdapter is used to populate a DataSet with results from a database, and to read changes out of a DataSet and resolve those changes back to the database. Using a separate object, the DataAdapter, to communicate with the database allows the DataSet to remain completely generic with respect to the data it contains, and gives you more control over when and how commands are executed and changes are sent to the database. ADO performs much of this behavior implicitly, however the explicit design of ADO.NET enables you to fine-tune your interaction with a data source for best performance and scalability.

The implicit update behavior of ADO is also available in ADO.NET using a CommandBuilder object that, based on a single table SELECT, automatically generates the INSERT, UPDATE, and DELETE commands used for queries by the DataAdapter. However, the compromise for this convenience is slower performance and less control over how changes are propagated to the data source because, as with ADO, the commands are generated from metadata collected at run time.

Data Types

In ADO, all results are returned in a standard OLE Automation Variant type. This can hinder performance because, in addition to conversion overhead, variants are allocated using task-allocated system memory, which causes contention across the system. When retrieving results from a DataReader in ADO.NET, however, you can retrieve columns in their native data type, as a common Object class, without going through expensive conversions. Data values can either be exposed as .NET Framework types, or can be placed in a proprietary structure in the .NET Framework to preserve the fidelity of the native type. An example of this is the SQL Server .NET Data Provider, which can be used to expose Microsoft® SQL Server™ data as .NET Framework types, or as proprietary types defined by the classes in the System.Data.SqlTypes namespace.

Summary

ADO.NET is designed to build on the strength of the ADO programming model, while providing an evolution of data access technology to meet the changing needs of the developer. It is designed to leverage your existing knowledge of ADO, while giving you much finer control over the components, resources, and behavior of your applications when accessing and working with data.

Related Topics in the .NET Framework SDK

• Accessing Data with ADO.NET

Describes ADO.NET architecture and how to use the ADO.NET classes to manage application data and interact with data sources including Microsoft SQL Server, OLE DB, and XML.

• Overview of ADO.NET

Provides an introduction to the design and components of ADO.NET.
• Using .NET Data Providers to Access Data
Describes the components of a .NET data provider and how to use them to access relational data sources.
• Creating and Using DataSets
Describes the DataSet and how to use it to manage relational data in your application.
• XML and the DataSet
Describes how the DataSet interacts with XML as a data source, including loading and persisting the contents of a DataSet as XML and synchronizing a DataSet with an XmlDataDocument.
FAQ
This section answers the questions most commonly asked by developers familiar with ADO as they begin to familiarize themselves with ADO.NET.
What happened to the Recordset?
The ADO Recordset bundles functionality together into one object and handles much behavior implicitly. ADO.NET, on the other hand, has been designed to factor behavior into separate components and to enable you to explicitly control behavior. The following table describes the individual ADO.NET objects that provide the functionality of the ADO Recordset.
ADO.NET object Description
DataReader Provides a forward-only, read-only stream of data from a data source.
The DataReader is similar to a Recordset with CursorType = adOpenForwardOnly and LockType = adLockReadOnly.
DataSet Provides in-memory access to relational data.
The DataSet is independent of any specific data source and therefore can be populated from multiple and differing data sources including relational databases and XML, or can be populated with data local to the application. Data is stored in a collection of one or more tables, and can be accessed non-sequentially and without limits to availability, unlike ADO in which data must be accessed a single row at a time. A DataSet can contain relationships between tables, similar to the ADO Recordset in which a single result set is created from a JOIN. A DataSet can also contain unique, primary key, and foreign key constraints on its tables.
The DataSet is similar to a Recordset with CursorLocation = adUseClient, CursorType = adOpenStatic, and LockType = adLockOptimistic. However, the DataSet has extended capabilities over the Recordset for managing application data.
DataAdapter Populates a DataSet with data from a relational database and resolves changes in the DataSet back to the data source.
The DataAdapter enables you to explicitly specify behavior that the Recordset performs implicitly.
Related Topics in the .NET Framework SDK
• Retrieving Data Using the DataReader
Describes the DataReader and how to use it to return results from a data source.
• Creating and Using DataSets
Describes the DataSet and how to use it to manage relational data in your application.
• Populating a DataSet from a DataAdapter
Describes how to fill the contents of a DataSet from a relational data source using a DataAdapter.
• Updating the Database with a DataAdapter and the DataSet
Describes how to resolve changes to data in a DataSet back to a data source using a DataAdapter.
• Using .NET Data Providers to Access Data
Describes the components of a .NET data provider and how to use them to access relational data sources.

What happened to cursors?
In ADO it is possible, within a common Recordset object, to request multiple and differing cursor types (dynamic, keyset, static, and forward-only) with different properties that define how the cursor behaves, for example whether the cursor is updateable or is read-only, or whether it is implemented on the client or on the server. In ADO.NET, however, different classes are exposed that give you greater control over each type of interaction. The DataReader provides an extremely fast, forward-only, read-only cursor on the server side that enables you to retrieve a stream of results from a database. The DataSet provides a completely disconnected "client" cursor, through which you can scroll and update, that is equivalent to the static cursor in ADO. These objects, along with the DataAdapter that enables you to move data between the DataSet and a database, provide you with optimal access methods for the most common types of data interactions.
Note that ADO.NET version 1.0 does not expose a scrollable, updateable server-side cursor. Applications that require scrolling and positioned updates on the client side generally involve user interaction. Because server-side cursors require state to be held on the server, your application will not be robust or scalable if it must hold those valuable resources while users interact with the data on the client side. Most applications that currently use scrollable server-side cursors on the client could be much more efficiently written according to one of the following designs:
• Use stored procedures to handle custom logic, to run on the server instead of the client.
• Use a forward-only, read-only cursor to return data from the server, and execute commands to process any updates.
• Populate a DataSet with results, modify the data locally, and then propagate those changes back to the server.
Related Topics in the .NET Framework SDK
• Using .NET Data Providers to Access Data
Describes the components of a .NET data provider and how to use them to access relational data sources.
• Optimistic Concurrency
Describes the optimistic concurrency model and how to use ADO.NET to handle optimistic concurrency violations.
• Paging Through a Query Result
Provides an example of returning the results of a query as smaller sections, or "pages", of data.
How do I populate a DataSet with data from my database if it is disconnected from the data source?
How do I resolve changes to the DataSet back to the database?
The DataAdapter provides the bridge between the DataSet and the data source. You control the behavior for populating the DataSet and resolving inserts, updates, and deletes in the DataSet back to the data source by defining explicit commands that the DataAdapter will use.
The DataAdapter command properties are the SelectCommand, InsertCommand, UpdateCommand, and DeleteCommand. Each command corresponds directly to a SELECT, INSERT, UPDATE, and DELETE action at the data source. Additionally, these actions can be optimized as a stored procedure call. Once the DataAdapter commands have been defined, you can pass a DataSet to the Fill method of a DataAdapter to fill a DataSet with the results returned by the SelectCommand, or pass a DataSet to the Update method of a DataAdapter to propagate changes in the DataSet back to the data source. The InsertCommand will process rows that have been added to the DataSet. The UpdateCommand will process existing rows that have been modified in the DataSet. The DeleteCommand will process existing rows that have been deleted from the DataSet.
Related Topics in the .NET Framework SDK
• Updating the Database with a DataAdapter and the DataSet

Describes how to resolve changes to data in a DataSet back to a data source using a DataAdapter.
• Populating a DataSet from a DataAdapter
Describes how to fill the contents of a DataSet from a relational data source using a DataAdapter.
• Using .NET Data Providers to Access Data
Describes the components of a .NET data provider and how to use them to access relational data sources.
How do I use ADO.NET with XML, similar to adPersistXml?
The DataSet provides extensive support for using XML to load and persist the schema and data within a DataSet. You can load the contents of the DataSet from any XML format, and write the contents of a DataSet to an XML format that is much simpler and more generic than that of an ADO Recordset saved as XML. The schema, or relational structure, of a DataSet can easily be persisted as, or created from, a simple XML Schema definition language (XSD) schema. If an XML document has no XML Schema supplied, and no schema is defined within the DataSet, the DataSet can infer the schema from the XML elements in the XML document. Additionally, the DataSet gives you control over how rows and columns are written to and read from an XML document. Columns can be mapped as attributes, elements, or simple content, or can be hidden (not written out). Related rows can be nested within their parent element, or treated as sibling elements.
The DataSet can also be synchronized with an XmlDataDocument to provide simultaneous relational and hierarchical views of a single set of data. By synchronizing a DataSet with an XmlDataDocument, you also gain access to other XML functionality for the data in your DataSet such as the ability to perform XML Path Language (XPath) queries over the data or to apply an Extensible Stylesheet Language Transformation (XSLT transformation) to the data.
The SQL Server .NET Data Provider also provides the capability, using the SqlCommand, to return the results of FOR XML queries against Microsoft SQL Server 2000 or later directly as an XmlReader.
In addition to the XML capabilities provided with ADO.NET, SQLXML 2.0 (XML for SQL Server 2000) contains SQLXML Managed Classes that enable you to access the XML functionality of Microsoft SQL Server 2000 and later from the .NET Framework. For example, these classes allow you to execute XML templates, perform XPath queries over data at the server, or perform updates to data using Updategrams or Diffgrams.
Related Topics in the .NET Framework SDK
• Obtaining Data as XML from SQL Server
Describes how to return the results of a FOR XML query in Microsoft SQL Server 2000 or later as an XmlReader using the SQL Server .NET Data Provider.
• XML and the DataSet
Describes how the DataSet interacts with XML as a data source, including loading and persisting the contents of a DataSet as XML and synchronizing a DataSet with an XmlDataDocument.
• SQLXML 2.0 (XML for SQL Server 2000)
Provides the release of XML for Microsoft SQL Server 2000 (SQLXML 2.0), which includes SQLXML Managed Classes for use in the .NET Framework.
How do I use my existing COM components, which use ADO for data retrieval and updates, in the .NET Framework?
COM components that return or consume ADO objects are available in the .NET Framework using COM interop services. Additionally, the OLE DB .NET Data Provider includes overloads to the OleDbDataAdapter.Fill method which take as input an ADO Recordset or Record object returned by existing COM components, and populate a DataSet with the data contained in the ADO object. Updates to the data in the DataSet can be propagated back to the data source using a DataAdapter. You can also use an Extensible Stylesheet Language

Transformation (XSLT transformation) to transform between the XML format of the ADO Recordset and the XML format of the ADO.NET DataSet.
Related Topics in the .NET Framework SDK
• Accessing an ADO Recordset or Record from ADO.NET
Describes how to use the OleDbDataAdapter to fill the contents of a DataSet or DataTable from an ADO Recordset or Record object.
• Exposing COM Components to the .NET Framework
Describes how to access COM components from within the .NET Framework.

~~~ End of Article ~~~