Data Points

Contrasting the ADO.NET DataReader and DataSet

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifContents

[Being Connected](http://msdn.microsoft.com/en-us/magazine/cc188717.aspx#S1)   
[Supporting Multiple Resultsets](http://msdn.microsoft.com/en-us/magazine/cc188717.aspx#S2)  
[The Disconnected Side](http://msdn.microsoft.com/en-us/magazine/cc188717.aspx#S3)  
[When to Use the DataSet](http://msdn.microsoft.com/en-us/magazine/cc188717.aspx#S4)

Iam often asked by developers whether the ADO.NET DataReader or the DataSet is the better tool. Some developers say the DataReader is better because it is lightweight, while still others say they prefer the DataSet for its inherent flexibility. The truth is that both have their place in Microsoft® .NET development as their usefulness depends on the situation.

The ADO 2.*x* recordset object is able to operate in either a connected or a disconnected mode. It can remain connected to the underlying database while traversing a forward-only rowset or it can retrieve a rowset into a client-side, in-memory cursor and disconnect itself from the database. Among the hurdles you may well encounter in migrating from classic ADO to ADO.NET is gaining a full understanding of how the operations that the ADO recordset performed are now handled in ADO.NET.

Instead of a single rowset container, ADO.NET offers two distinctly separate data storage objects: the DataReader and the DataSet. This month I'll focus on the purpose of these two data retrieval classes in ADO.NET and help you to decide which is the best choice to use in a particular situation. I will explore how to retrieve data into both the DataReader and the DataSet, beginning by discussing the DataReader's unique capabilities. I will also compare the connected DataReader to the disconnected DataSet, weighing the pros and cons of using each in different scenarios.

**Being Connected**

Before deciding when to use a DataReader, it is smart to understand its features and limitations. The DataReader has a defined set of operations that revolve around its connected, forward-only, read-only nature (the read-only DataReader is also known as the firehose cursor of ADO.NET). A DataReader is a stream of data that is returned from a database query. When the query is executed, the first row is returned to the DataReader via the stream. The stream then remains connected to the database, poised to retrieve the next record. The DataReader reads one row at a time from the database and can only move forward, one record at a time. As the DataReader reads the rows from the database, the values of the columns in each row can be read and evaluated, but they cannot be edited.

Unlike the DataSet, the DataReader is not implemented directly in the System.Data namespace. Rather, the DataReader is implemented through a specific managed provider's namespace such as System.Data.SqlClient.SqlDataReader. Because all DataReaders, including the OleDbDataReader, the SqlDataReader, and other managed provider's DataReaders implement the same IDataReader interface, they should all provide the same base set of functionality. Each DataReader is optimized for a specific data provider. If the database you are developing against has a managed provider for ADO.NET, then you should take advantage of it. Otherwise, you can use the System.Data.OleDb or the System.Data.Odbc namespaces, which expose more generic managed providers that can access a variety of data sources. If you are developing against SQL Server™ or Oracle, it would be more efficient to use the provider that was made specifically for these databases. In this column, I will query the SQL Server Northwind database using the System.Data.SqlClient namespace.

The fact that the SqlDataReader is part of a specific managed provider's feature set further differentiates it from the DataSet. The SqlDataReader can only retrieve one row at a time from the data source and in order for it to get the next record, it has to maintain its connection to the data source. The DataSet, however, doesn't need to know about where it gets its data. The DataReader can only get its data from a data source through a managed provider. The DataSet can also get its data from a data source via a managed provider, but the data source can also be loaded manually, even from an XML file on a hard drive. If the .NET Framework does not provide a managed provider that is specifically designed for your database, it is certainly worth checking to see if the manufacturer or a third party has one available since they should perform better than the generic OLE DB and ODBC providers.

In ASP.NET, DataReader objects can be used for more robust situations such as binding themselves to an ASP.NET DataGrid or a DropDownList server control. The following code demonstrates how to retrieve a list of products from the Northwind database using a SqlDataReader object:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl02_code');" \o "Copy Code)

string sSQL = "SELECT \* FROM Products";

string sConnString =

"Server=(local);Database=Northwind;Integrated Security=SSPI;";

using (SqlConnection oCn = new SqlConnection(sConnString))

{

SqlCommand oSelCmd = new SqlCommand(sSQL, oCn);

oSelCmd.CommandType = CommandType.Text;

oCn.Open();

SqlDataReader oDr = oSelCmd.ExecuteReader();

DataGrid1.DataSource = oDr;

DataGrid1.DataBind();

}

Both a SqlConnection and a SqlCommand object are created. The SqlConnection is opened and the SqlCommand object executes the SQL query, returning the first row to the SqlDataReader. At this point the connection to the database is still open and associated with the SqlDataReader. This code shows how a SqlDataReader can be bound to a bindable object such as an ASP.NET DataGrid.

Alternatively, a DataReader could be used to retrieve the rows and then loop through them manually, one by one. It can support several resultsets as well. For example, a list of products and categories could be retrieved from a database. The following code retrieves a SqlDataReader and loops through its rows, writing the first column's value for each row to the console:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl03_code');" \o "Copy Code)

SqlDataReader oDr = oCmd.ExecuteReader();

while(oDr.Read()) {

Console.WriteLine(oDr[0]);

}

**Supporting Multiple Resultsets**

The DataReader supports access to multiple resultsets, one at a time, in the order they are retrieved. This code is easily modified to handle multiple resultsets. The following code retrieves a SqlDataReader and loops through its rows, again writing the first column's value for each row to the console:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl04_code');" \o "Copy Code)

SqlDataReader oDr = oCmd.ExecuteReader();

do {

while(oDr.Read())

{

Console.WriteLine(oDr[0]);

}

Console.WriteLine(oDr[0]);

}

while(oDr.NextResult());

Once all of the rows from the first resultset are traversed, the rowset from the second query is retrieved and its rows are traversed and written. This process can continue for several resultsets using a single SqlDataReader.

The Read method of the SqlDataReader loads the next record so that it can be accessed and moves the position of the cursor ahead. It returns a Boolean value indicating the existence of more records. This feature can help circumvent a common problem in classic ADO development: the endless loop. In classic ADO, when looping through a recordset object developers would often omit the MoveNext method and run the code only to remember a second too late that this would cause the recordset to loop infinitely on the same record. ADO.NET is kind to developers as its DataReader object's Read method automatically moves the position to the next record so this situation can't occur. The NextResult method of the SqlDataReader object retrieves the subsequent rowset and makes it accessible to the SqlDataReader. It also returns a Boolean value indicating if there are additional resultsets to traverse, like the Read method does.

The DataReader in the previous code sample shows how to get the value for a column from the DataReader using its ordinal index position. Can the DataReader be indexed by the column name or can the index of the column be retrieved? The answer to both of these questions is yes. The code in **Figure 1** shows how a DataReader retrieves data and displays the CompanyName for each row in the diagnostics' output window.

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifFigure 1 Retrieving and Displaying Data

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl14_ctl00_ctl00_code');" \o "Copy Code)

string sConnString =

"Server=(local);Database=Northwind;Integrated Security=SSPI;";

using(SqlConnection oCn = new SqlConnection(sConnString))

{

SqlCommand oCmd = new SqlCommand("SELECT \* FROM Customers", oCn);

oCn.Open();

SqlDataReader oDr =

oCmd.ExecuteReader(CommandBehavior.CloseConnection);

while(oDr.Read())

{

System.Diagnostics.Debug.WriteLine("Company Name " +

oDr["CompanyName"]);

System.Diagnostics.Debug.WriteLine("Company Name: name = " +

oDr.GetName(1));

System.Diagnostics.Debug.WriteLine("Company Name: index = " +

oDr.GetOrdinal("CompanyName"));

}

}

To demonstrate these techniques, this code displays the CompanyName value, column name, and index. The first line in the loop writes the CompanyName using the value representing the name of the CompanyName column. This could also have been accomplished by passing the index of 1. I avoid this as it is less clear which column you are accessing, although using the string value is slower than using the index. The second line in the loop writes the name of the column at position 1 (CompanyName) using the GetName method. The third line gets the index of the CompanyName column using the GetOrdinal method of the SqlDataReader. You might also notice that the CommandBehavior is set to CloseConnection, ensuring that the connection will be closed when the SqlDataReader is closed. These methods are simple but they make the SqlDataReader a power tool.

**The Disconnected Side**

The DataSet is the main data storage tool in the ADO.NET disconnected architecture. Unlike the DataReader, the DataSet is not connected directly to a database through a Connection object when you populate it. Instead, to fill a DataSet from a database you first create a DataAdapter object (such as a SqlDataAdapter) for the provider and associate it with a SqlConnection object. Then the SqlDataAdapter can broker the data retrieval for the DataSet by issuing a SqlCommand against the database through the SqlConnection, retrieving the data, and filling the DataSet.

You can think of the SqlDataAdapter as a bridge between the connected and disconnected objects. One of its purposes is to serve as the route for a rowset to get from the database to the DataSet. For example, when the SqlDataAdapter's Fill method is executed it opens its associated SqlConnection object (if not already open) and issues its associated SqlCommand object against the SqlConnection. Behind the scenes, a SqlDataReader is created implicitly and the rowset is retrieved one row at a time in succession and sent to the DataSet. Once all of the data is in the DataSet, the implicit SqlDataReader is destroyed and the SqlConnection is closed.

The following code shows how a DataSet can be filled from the Products table of the Northwind database. Notice that there is no explicit SqlDataReader object in this code sample:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl07_code');" \o "Copy Code)

string sSQL = "SELECT \* FROM Products";

string sConnString =

"Server=(local);Database=Northwind;Integrated Security=SSPI;";

SqlDataAdapter oDa = new SqlDataAdapter();

DataSet oDs = new DataSet();

using(SqlConnection oCn = new SqlConnection(sConnString))

{

SqlCommand oSelCmd = new SqlCommand(sSQL, oCn);

oSelCmd.CommandType = CommandType.Text;

oDa.SelectCommand = oSelCmd;

oDa.Fill(oDs, "Products");

}

The DataSet can read and load itself from an XML document as well as export its rowset to an XML document. Because the DataSet can be represented in XML, it can be easily transported across processes, a network, or even the Internet via HTTP. Unlike the DataReader, the DataSet is not read-only. A DataSet can be modified, and rows can be added or deleted. Changes to a DataSet can be sent to the database via a managed provider's objects. Another key difference between the DataSet and the DataReader is that the DataSet is fully navigable. Its rows can be traversed forward or backward. The DataReader can be traversed forward only. In addition, a DataSet is highly flexible in that its DataTable objects can be filtered vertically or horizontally and they can be sorted or even searched. The DataSet is independent of any one data provider as it relies on a DataAdapter specific to each provider to broker the data between the DataSet and the database.

Not only can the DataSet be loaded from XML, it can also be loaded manually. Notice in **Figure 2** how a DataTable is created and its columns added manually. A primary key constraint is established as well as an auto-incrementing value for the key field. Then the DataTable is added to an empty DataSet (though is doesn't have to be empty) and the rows are added one by one to the DataTable, all without ever connecting to a data source.

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifFigure 2 Creating a DataSet Manually

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl16_ctl00_ctl00_code');" \o "Copy Code)

//-- Create the table

DataTable oDt = new DataTable("Employees");

DataRow oRow;

oDt.Columns.Add("EmployeeID", System.Type.GetType("System.Int32"));

oDt.Columns.Add("FirstName", System.Type.GetType("System.String"));

oDt.Columns.Add("LastName", System.Type.GetType("System.String"));

oDt.Constraints.Add("PK\_Employees", oDt.Columns["EmployeeID"], true);

oDt.Columns["EmployeeID"].AutoIncrement = true;

oDt.Columns["EmployeeID"].AutoIncrementSeed = -1000;

oDt.Columns["EmployeeID"].AutoIncrementStep = -1;

oDs.Tables.Add(oDt);

//-- Add the rows

oRow = oDs.Tables["Employees"].NewRow();

oRow["FirstName"] = "Haley";

oRow["LastName"] = "Smith";

oDs.Tables["Employees"].Rows.Add(oRow);

oRow = oDs.Tables["Employees"].NewRow();

oRow["FirstName"] = "Madelyn";

oRow["LastName"] = "Jones";

oDs.Tables["Employees"].Rows.Add(oRow);

//-- Bind it to a DataGrid

grdEmployees.DataSource = oDs.Tables["Employees"];

Because the DataSet is disconnected, its use can reduce the demand on database servers. It does, however, increase the memory footprint in the tier where it is stored, so be sure to account for that when designing around a DataSet as your data store. Scaling up on the middle tier using parallel servers and load balancing is a common way to handle the increased load so that session-based information can be stored in objects such as the DataSet.

**When to Use the DataSet**

Your situation will dictate when and where you'll use a DataSet versus a DataReader. For example, the DataSet's disconnected nature allows it to be transformed into XML and sent over the wire via HTTP if appropriate. This makes it ideal as the return vehicle from business-tier objects and Web services. A DataReader cannot be serialized and thus cannot be passed between physical-tier boundaries where only string (XML) data can go.

DataSet objects are a good choice when the data must be edited or rows added or deleted from the database. It is not the only choice, however, as a DataReader could be used to retrieve the data and changes would be sent to the database via a SqlDataAdapter through a separate, self-maintained DataRow array. That process can be quite messy because the SqlDataReader cannot allow edits as the DataSet can. As mentioned earlier, the DataSet is also a good choice when you need data manipulation such as filtering, sorting, or searching. Since the DataSet can be traversed in any direction, all of these features are available. This flexibility also makes the DataSet an easy choice when the situation calls for multiple iterations of the rowset. A DataReader can move forward only, so to loop through it more than once, the DataReader would be closed and reopened and the query would hit the database a second time.

If a rowset is intended to be bound to a single ASP.NET server control and the data is read-only, the DataReader could suffice. If a rowset is intended to be bound to more than one read-only ASP.NET server control, you should consider using a DataSet instead. If a DataReader was bound to more than one control (such as three DropDownList controls), the same query would hit the database three times since the DataReader can only move forward. The DataSet also works well when a rowset must be persisted between page calls to the Session or Cache objects.

Of course, because the DataReader is associated with a specific data source, it cannot be created, filled, or traversed without a connection to the data source. Unlike the DataReader, a DataSet can be created manually without a connection to the source. In a situation such as an online shopping cart in which a custom data store is required, a DataSet could be created manually and its rows added.

Another good use of the DataSet is the situation in which data must be retrieved and a complex action performed on each row. For example, an application might retrieve a hundred stock and mutual fund symbols from a 401k table that needs to be edited. This data might have to include the stock and mutual fund prices on screen, as well. A DataSet could be used to store the rowset and some code could loop through the DataSet and perform a lookup of each stock's price through a third-party Web service. Finally, one of the more compelling reasons to use a DataSet instead of a DataReader is that the DataSet can be serialized when the rowset needs to be passed around a network or the Internet. A DataReader cannot be serialized to XML due to its connected nature.

The DataSet is not the best solution in every situation, and there are several situations in which DataReaders should really be considered. One is when an application implements a .NET architecture without data binding—situations in which manual updates to the database are performed and controls are loaded by looping through rowsets. DataReaders are a good choice when an application has to be sensitive to changes in the underlying database.

There are other times when a DataReader can be the right choice, such as when populating a list or retrieving 10,000 records for a business rule. When a huge amount of data must be retrieved to a business process, even on a middle tier, it can take a while to load a DataSet, pass the data to it on the business tier from the database, and then store it in memory. The footprint could be quite large and with numerous instances of it running (such as in a Web application where hundreds of users may be connected), scalability would become a problem. If this data is intended to be retrieved and then traversed for business rule processing, the DataReader could speed up the process as it retrieves one row at a time and does not require the memory resources that the DataSet requires.

When output or return parameters need to be retrieved, a DataReader will not allow you to get them until the DataReader and its connection are closed. If data must be bound to a read-only list in a Web Form, a DataReader is a very good choice. Binding a DataReader to a DropDownList or even a read-only DataGrid in ASP.NET works well as the data can be retrieved and displayed in the list but does not need to be persisted for editing. DataSets are ideal if data needs to be edited, sorted, filtered, or searched.

As I have shown, when data must be passed without a connection to a database or when rich features for manipulating the data are required, a DataSet fits the bill nicely. The DataReader works well when a simpler purpose for the data exists such as populating a dropdown list or retrieving tens of thousands of rows for processing. Your decision should be based on the particular factors of the situation of the application.