ADO.NET Explored

by Guy Smith-Ferrier

ADO.NET is Microsoft's data access layer in Visual Studio.NET. At first sight it can be viewed as a progression of the ADO which we have come to know over the past few years. This article explores the new architecture, illustrates many of the differences between ADO and ADO.NET, and shows how ADO.NET can be supported in Delphi 4, 5 and 6 today.

The code shown in this article is based on .NET Public Beta 1. You can expect to see Public Beta 2 in July 2001, where some classes, assemblies and methods have been renamed, in addition to other changes. Where the changes are already known they will be pointed out so that you can translate the examples as necessary.

ADO.NET, at one time called ADO+ (and even earlier referred to as XDO because of its fundamental dependence on XML), is a redesign of ADO to better suit the needs of web developers. As you will see, there are many similarities between ADO and ADO.NET, but it is also true to say that there are just as many differences. As a consequence, porting ADO code to ADO.NET is likely to need a certain amount of reworking. To understand the philosophy behind this redesign we should first look at what shortcomings 'classic' ADO suffers from ('classic' and 'legacy' have been phenomenally successful in recent years in installing a sense of guilt and unworthiness in those who dare to stay with tried and trusted solutions instead of jumping ship and trying the latest and greatest).

ADO is based on COM and this is perhaps the single largest reason why its success is less than meteoric in web development. ADO's previous attempt at data transmission from server to client and client to server was Remote Data Services (RDS). RDS was not widely embraced by ADO developers. One of the problems is that RDS is based on COM, which mostly only runs on Windows. This is a problem for interoperability, particularly over the internet. In addition, the transmission of a recordset from one process to another (eg server to client) requires COM marshalling, and this incurs a performance penalty not only with the process of moving the data but also in the translation of database data types to COM data types and back again. But perhaps the biggest reason for COM's failure in web development is that COM calls cannot always penetrate firewalls.

➤ *Table 1: ADO and ADO.NET.*

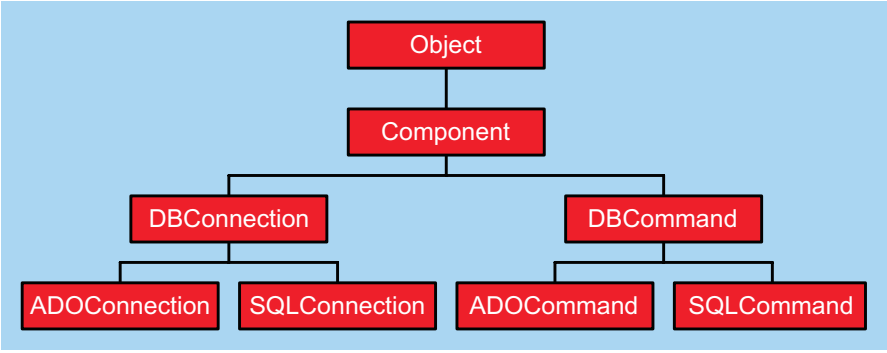
ADO.NET	Classic ADO Equivalent
Connection	Connection
Command	Command
DataSetCommand	Data Manipulation methods of Recordset
DataReader	Forwards only, read-only cursor
DataSet	(Set of DataTables)
DataTable	Disconnected, client-side Recordset
DataColumn	Field
DataRow	
DataRelation	SQL JOIN or MS Data Shape

Enter ADO.NET. ADO.NET uses a similar model to 'classic' ADO in that it has connections and commands, but it differs in that there is no `RecordSet`. ADO's `RecordSet` was considered to solve too many problems single-handedly, so it has been decomposed into smaller more focused classes. Where ADO.NET scores over classic ADO in web development is in its data transmission format. ADO.NET uses XML to communicate data between processes. This means that the client software can be from any vendor running on any platform. The same is true for the server software. It also means that there is no COM marshalling and no type conversion to or from COM data types. Lastly, as XML is just plain text, data can pass through firewalls.

Table 1 shows the fundamental ADO.NET classes and their nearest equivalent classic ADO classes. We will look at examples of most of these classes throughout this article.

ADO.NET offers a choice of classes. In Public Beta 1 there are two sets of classes: ADO and SQL. The ADO classes (renamed to OLEDB classes in Public Beta 2) are the closest link to classic ADO. They use traditional OLE DB providers and the same connection strings. The SQL classes are specifically written for SQL Server 7 and 2000 and do not use traditional OLE DB. Instead they communicate directly with SQL Server using Tabular Data Stream (TDS), providing significantly better performance. Figure 1 shows the class hierarchies for the `Connection` and `Command` classes.

Consequently, when you write an ADO.NET application, you make a conscious decision to commit yourself to one of the two sets of classes: ADO or SQL. Of course, this only refers to the code which defines the variables and constructs new objects, as the remaining code is compatible between sets of classes, because they share the same ancestors. However, it represents an interesting departure from the previous ADO philosophy. In addition to the specific



➤ *Figure 1: Connection and Command classes.*

classes for SQL Server you can expect to see explicit support for Oracle, Exchange 2000 and, possibly, Jet. It is likely that each of these will have their own sets of classes which, as a consequence of their names, will hard wire your applications into a specific database.

Supporting ADO.NET In Delphi

ADO.NET is a collection of 'managed' classes. As such they are hosted by the Common Language Runtime (CLR) and cannot be called directly by a non-managed application, such as a Delphi application. However, .NET includes a COM Interoperability service which allows .NET applications to use COM and allows non-managed applications to treat managed classes as COM classes. It is this latter facility which provides Delphi with the bridge to ADO.NET. .NET includes a utility called REGASM.EXE in

```
C:\WINNT\Microsoft.NET\
  Framework\v1.0.2204
```

which creates a COM wrapper for .NET managed classes and creates the appropriate registry entries to allow the classes to be treated as COM classes. The command in Listing 1 creates a type library in COMSystemData.DLL for the `System.Data` assembly (an assembly is akin to a managed DLL and `System.Data` is the namespace which contains both ADO and SQL classes).

This will take several minutes to execute, but once complete you will be able to register the type library (using TREGSVR or REGSVR32). Next, import the registered type library into Delphi (choose `Project | Import Type Library` | and select '`System_Data (Version 1.0)`').

You will need to make some changes before importing the type library. Firstly, I recommend unchecking the `Generate Component Wrapper` checkbox. Secondly, you will need to manually alter the names of the .NET components to prevent them from conflicting with Delphi's components. Specifically this affects Delphi's `TADOCommand`, `TADOConnection` and `TDataSet` classes. Now you can import the type library. Unfortunately the resulting files will not compile because the file ComRuntimeLibrary_TLB.PAS contains definitions of `Byte`, `Int64`, `Single` and `Double` which are all existing types in Delphi. You can either comment these definitions out (for a quick and dirty solution) or else rename these types (and all of the uses of these types in all imported .PAS files). Finally, change `Currency` to `Double` in System_XML_TLB.PAS.

You are now ready to use ADO.NET in Delphi.

Create a new application, add `System_Data_TLB` and `ComObj` to the `uses` clause, and add a button with the code in Listing 2.

This code simply executes `SELECT * FROM CUSTOMERS` and isn't even interested in catching the result set. Of course, in ADO.NET the use of the ADO classes to access SQL Server is a rather unwise choice (because you would use the SQL classes instead). As you can see, we treat the ADO.NET classes as regular COM classes using `CreateCOMObject` and the as operator (and suffering from having to call `Set_ConnectionString` and `Set_ActiveConnection` instead of making regular assignments). Notice, however, that we cannot pass parameters to the class constructors when using COM objects, so initial values have to be set separately on subsequent lines. This difference represents a significant problem for .NET classes which only allow certain values to be set in the constructor.

Using SQLConnection And SQLCommand

To use the SQL Server classes instead of the ADO classes simply replace `ADOConnection` with `SQLConnection` and `ADOCommand` with `SQLCommand`. In addition to this you will need to modify the connection string. Since the SQL classes know that they will be using SQL Server and only SQL Server, the specification of a provider is irrelevant and unwanted so you must remove

```
REGASM "C:\WINNT\Microsoft.NET\Framework\v1.0.2204\System.Data.DLL"
  /tlb: COMSystemData.DLL
```

➤ *Above: Listing 1*

➤ *Below: Listing 2*

```
procedure TForm1.Button1Click(Sender: TObject);
var
  oConn: ADOConnection;
  oComm: ADOCommand;
begin
  oConn:=CreateCOMObject(Class_ADOConnection) as ADOConnection;
  oConn.Set_ConnectionString(
    'Provider=SQLOLEDB.1;Persist Security Info = False;'+
    'Initial Catalog = Northwind;User ID = sa;');
  oConn.Open;
  oComm:=CreateCOMObject(Class_ADOCommand) as ADOCommand;
  oComm.Set_ActiveConnection(oConn);
  oComm.Set_CommandText('SELECT * FROM CUSTOMERS');
  oComm.Execute;
  oConn.Close;
end;
```

the Provider argument and add a Data Source argument giving the SQL Server name.

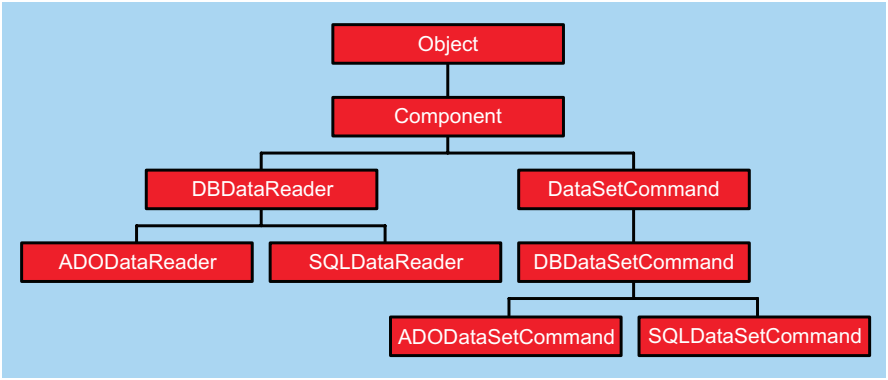
DataReaders

So far so good but we need a way to read the data. Enter the DataReader. DataReader is equivalent to a read-only, forwards only cursor. ADO developers sometimes refer to this as a firehose cursor. (In dbExpress this is equivalent to any of the TCustomSQLDataSet descendants.) Figure 2 shows the DataReader and DataSetCommand class hierarchy (we will return to the DataSetCommand later).

The Listing 3 code snippet adds the CompanyName field of every record to a ListBox using an ADOCommand supplied as an argument.

The command is executed and the DataReader attached to the result set. (In Public Beta 2 the Execute line will change to oDR:=oComm.Execute). Notice that the while loop does not contain a MoveNext as DataSetReader.Read reads the next record and returns False if the end of the result set has been reached.

In this Delphi example the Execute method is called Execute_2 because it has been overloaded and COM doesn't support overloading, so the type library import facility has had to generate a unique name for the overloaded method. In addition, we have to add a workaround as we cannot pass the oDR ADODataReader to Execute_2 because it is a different type to that which Execute_2 expects. So we pass oWorkAround and, on the next line, typecast it to an ADODataReader.



➤ *Figure 2: DataReader and DataSetCommand class hierarchy.*

DataSetCommands

The DataReader is fine if all you want to do is to read data. To read and update data you will need to use a DataSetCommand (which will be renamed to DataAdapter in Public Beta 2). Fortunately for Delphi programmers, the DataSetCommand is not a difficult class to comprehend, because we already have a very similar class in TUpdateSQL. TUpdateSQL has DeleteSQL, InsertSQL and ModifySQL properties to allow you to specify SQL which executes when there is a need to delete, insert or update the data. DataSetCommand has DeleteCommand, InsertCommand, SelectCommand and UpdateCommand properties which are all of type Command. One of the benefits that DataSetCommand has over TUpdateSQL is that, as the properties are Command objects instead of strings, these commands can execute stored procedures (which contain corresponding DELETE, INSERT, SELECT and UPDATE statements). Listing 4 shows a DataSetCommand being used to retrieve a dataset.

In this code we create a new ADODataSetCommand and set the SELECT statement and then the connection string. We then use FillDataSet to fill the oDS DataSet with data from the SELECT statement. Finally, we iterate through the collection of rows in the Customers table.

Although the DataSetCommand represents the standard way of providing a read/write dataset, I must confess a little disappointment with this solution. Like Delphi's TUpdateSQL, DataSetCommand uses statically generated SQL statements (either directly or indirectly by using stored procedures which contain statically generated SQL statements). Such statements are fragile and do not respond well to changes in the database structure. As we all

```
procedure TForm1.ShowCustomers(oComm: ADOCommand);
var
  oDR: ADODataReader;
  oWorkAround: IDataReader;
begin
  oComm.Execute_2(oWorkAround);
  oDR:=oWorkAround as ADODataReader;
  while oDR.Read do
    ListBox1.Items.Add(oDR.GetString(2));
  oDR.Close;
end;
```

➤ *Above: Listing 3*

➤ *Below: Listing 4*

```
var
  oConn: ADOConnection;
  oDSCommand: ADODataSetCommand;
  oDS: DataSet;
  oRows: RowsCollection;
  intRow: integer;
begin
  oConn:=CreateCOMObject(Class_ADOConnection) as ADOConnection;
  oConn.Set_ConnectionString(
    'Provider=SQLOLEDB.1;Persist Security Info = False;'+
    'Initial Catalog = Northwind;User ID = sa;');
  oConn.Open;
  oDSCommand:=
    CreateCOMObject(Class_ADODataSetCommand) as ADODataSetCommand;
  oDSCommand.SelectCommand.Set_CommandText(
    'SELECT * FROM CUSTOMERS');
  oDSCommand.SelectCommand.Set_ActiveConnection(oConn);
  oDS:=CreateCOMObject(Class_DataSet) as DataSet;
  oDSCommand.FillDataSet_2(oDS, 'Customers');
  oRows:=oDS.Tables.Get_Item_2('Customers').Rows;
  for intRow:=0 to oRows.Count - 1 do
    ListBox1.Items.Add(oRows.Item[intRow].Item[1]);
end;
```

know, database structures change very frequently and any change will require manual alteration of the SQL statements. We do not get this additional maintenance overhead with dynamically generated SQL statements and I foresee developers either wasting time pointlessly modifying SQL statements or finding a rather more slick solution.

When changes have been made to any or all tables in the DataSet, the entire batch can be applied using DataSetCommand.Update in a similar manner to ADO's batch updates. Thus all updates in the database are applied in a single operation. This is similar to TClientDataSet.ApplyUpdates, but it is a great step forwards over ADO's inadequate RecordSet.UpdateBatch.

DataSets, DataTables, DataColumns, DataRows And DataRelations

A DataSet is an in-memory database. I use 'database' in its correct sense here to describe a collection of related tables. It is not like a Delphi TDataSet, since TDataSet represents a single table and ADO.NET's DataSet represents a collection of related tables. The DataSet contains DataTables and DataRelations which describe the relationships between the DataTables. I, for one, am delighted with this representation of a database. At last we have an object-based representation of the entire database in memory. It seems incredible that we have had to wait until 2001 for such an obvious concept from a major vendor.

The collection of classes shown in Figure 3 is entirely database independent and library independent. These classes can be used with ADO classes and SQL classes alike without the need of a library specific prefix.

A DataTable is a set of rows and columns which might equate to a table in a database or a single SELECT statement. This is the closest ADO.NET gets to an ADO Recordset and is most like a disconnected client-side Recordset. Indeed, you can create in memory tables just as you can in classic ADO.

Listing 5 creates a DataSet called SalesDB and a table called States with two columns: STATE and NAME. It ends by adding the table to the DataSet. The DataSet is only included in this example to show the relationship with the DataTable; it is not required in order to create independent DataTables. To add data just add new rows, see Listing 6.

DataRelation is used to define the relationships between tables in a DataSet. Unfortunately, DataRelation is one of the few ADO.NET classes which accepts parameters in its class constructor which cannot be set after the object has been constructed. This represents a problem for .NET's COM Interop layer because parameters cannot be passed to constructors in COM. As a result, there appears to be no way to create new DataRelation objects through COM. However, this problem might get resolved in a future .NET beta or the problem will become irrelevant when Delphi supports .NET directly. For the time being, though, Listing 7 shows what the code might look like if we didn't suffer from this problem.

This is a simple example but DataRelation has a ChildKeyConstraint property which has AcceptRejectRule, DeleteRule and UpdateRule properties which allow you to specify whether deletes should set the child table's foreign keys to null and whether changing the parent's primary key should cascade through all children and so on.

Transporting The Data

DataSet supports many similar methods for getting the data into and out of a DataSet (see Table 2).

Some methods duplicate functionality of other methods for convenience (eg ReadXML is the same as ReadXMLSchema followed by ReadXMLData). The separation of reading the schema from reading the data allows applications to

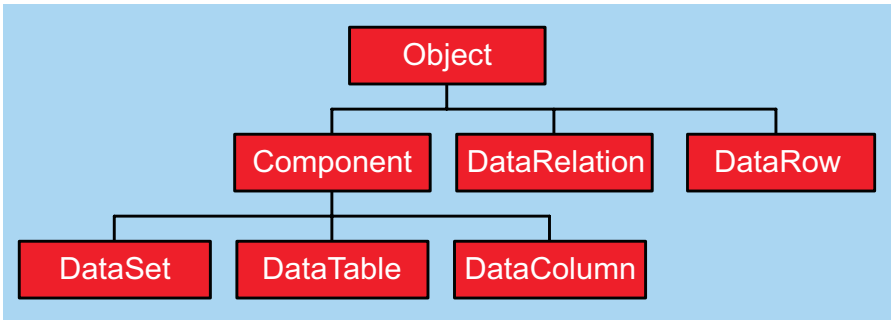
```
var
  oDataSet: DataSet;
  oTable: DataTable;
  oColumn: DataColumn;
begin
  oDataSet:=CreateCOMObject(Class_DataSet) as DataSet;
  oDataSet.Set_DataSetName('SalesDB');
  oTable:=CreateCOMObject(Class_DataTable) as DataTable;
  oTable.Set_TableName('States');
  oColumn:=CreateCOMObject(Class_DataColumn) as DataColumn;
  oColumn.Set_ColumnName('State');
  oTable.Columns.Add(oColumn);
  oColumn:=CreateCOMObject(Class_DataColumn) as DataColumn;
  oColumn.Set_ColumnName('Name');
  oTable.Columns.Add(oColumn);
  oDataSet.Tables.Add(oTable);
end;
```

➤ *Above: Listing 5*

➤ *Below: Listing 6*

```
oRow:=CreateCOMObject(Class_DataRow) as DataRow;
oRow.Set_Item_2('State', 'CA');
oRow.Set_Item_2('Name' , 'California');
oTable.Rows.Add_2(oRow);
oRow:=CreateCOMObject(Class_DataRow) as DataRow;
oRow.Set_Item_2('State', 'TX');
oRow.Set_Item_2('Name' , 'Texas');
oTable.Rows.Add_2(oRow);
```

➤ *Figure 3*



Method	Description
ReadDiffGram	Reads a DiffGram (a set of changes)
ReadXML	Reads the XML schema and XML data
ReadXMLData	Reads the XML data
ReadXMLRemoting	Reads an DiffGram or XML schema and data
ReadXMLSchema	Reads the XML schema
WriteDiffGram	Writes a DiffGram
WriteXML	Writes the XML schema and XML data
WriteXMLData	Writes the XML data
WriteXMLRemoting	Writes a DiffGram
WriteXMLSchema	Writes the XML schema

➤ *Table 2: DataSet methods for getting data in and out.*

```
var
  oRelation: DataRelation;
begin
  oDataSet.Relations.Add_4('CustomerStates',
  oStatesTable.Columns.Item_2['State'], oCustomersTable.Columns.Item_2['State']);
end;
```

➤ *Above: Listing 7*

➤ *Below: Listing 8*

```
var
  oDataSet: DataSet;
begin
  oDataSet:=CreateCOMObject(Class_DataSet) as DataSet;
  oDataSet.ReadXml_4('ABC.XML');
end;
```

read the schema locally (maybe even from XML hardwired into the program) and read the data remotely. As long as the schema information does not change or the client is always kept in synch with the server then there is no need to continually transport the same schema information over and over. In addition to these methods there are the `XML`, `XMLData` and `XMLSchema` properties which are all read/write.

The `WriteDiffGram` method writes a `DiffGram`, which is an XML document containing changes to the data (including the previous values). This is typically used to send changes from the client to the server. The changes are then read in on the server using `ReadDiffGram`.

The methods in Table 2 are all overloaded so can accept the XML source or destination in a variety of ways. A simple filename can be used for single tier applications which simply use an XML document as a database. This approach is also useful for briefcase applications. Other choices are to use `Streams`, `TextReaders` and `XMLReaders` (or `TextWriters` and `XMLWriters`). These are particularly useful for transporting the data across machines using HTTP, ASP `Request` and `Response` objects, or any other transport mechanism you care to employ. The XML supplied to the `Read` methods does not have to use Microsoft tag names so you can save XML using `TClientDataSet.SaveToFile` and then read it into a `DataSet`: see Listing 8.

Conclusion

ADO.NET is a significant redesign of ADO. Backwards compatibility with classic ADO is low and many existing ADO concepts have been dropped (including pessimistic locking and server-side cursors). The new design is certainly more elegant, and addresses the shortcomings of using classic ADO for web development. Existing ADO programmers have a head start in learning ADO.NET, as many of the basic building blocks can be seen in ADO today in the form of batch updates, disconnected recordsets, custom (fabricated) recordsets and DataShape.

Borland has already pledged support for .NET in a future version of Delphi (after Delphi 6) so this is a technology which is well worth watching.

Guy Smith-Ferrier is a Senior Delphi Consultant for Borland's Professional Services Organisation in the UK. Contact Guy at gsmithferrier@capellasoft.com