1

Displaying Data on the Web

When the Web first appeared, people had to find a metaphor for how information should be presented on it. If you took a sample of web sites from that period, the content largely was based around what you'd find in traditional media such as books, magazines, and newspapers. This led to the Web serving the same purpose as those other formats: it provided a snapshot of information as it stood at the time the pages were created. Of course, there was nothing wrong with that, but it placed restrictions on what the Web could reasonably be used for.

Over time, the technologies powering the Web have matured, and it has changed from only being able to provide static *sites*, to providing dynamic *applications* as well. These applications invite their users to make choices about the information they're interested in, providing a customized user experience that can be modified in real time.

The key to these applications is the data they contain. Regardless of what it is – it could be a product catalogue, or a set of customer details, or a document repository – it's the data that makes them dynamic. In the past, providing data over the Web has been a harder task than providing it through traditional desktop applications, due both to the development tools and functionality available, and the nature of the Web itself, where users are far removed from the applications and data. Over time, and in particular (from our point of view) with the introduction of Microsoft's .NET Framework, this situation has been improved. Web application developers are now on a more equal footing with their desktop-developing counterparts.

In this first chapter, we'll provide a broad introduction to the topic of data-driven web sites, and how they are implemented in ASP.NET. It starts with a discussion of the advantages and disadvantages of data-driven sites, and then moves on to examine the sources that such data can come from. After that, we'll look at the .NET Framework's data access strategy of choice – ADO.NET – including its architecture, its classes, and how it fits into the structure of data-driven applications. We'll finish by covering the installation of a database server that we'll use throughout this book.

A data-driven web application is a web site that displays dynamic data. The user experience changes to reflect the information held in a data store.

Pros and Cons of Data-Driven Web Sites

Some of the advantages of having a data-driven system are immediately apparent, but there are others that are less tangible and not so readily evident. Naturally enough, there are also reasons why you might *not* want to attach a web site to a database. In this section, we'll examine the benefits and the drawbacks of creating a web site that's based around a data source.

Advantages

There are many secondary benefits of making a web site data-driven, such as the ability to reuse portions of functionality in other projects, and being able to share common pieces of information across systems – these tend to kick in when you start to work on your second or your third web application. Here, we're going to look at some of the advantages that can start to accrue as soon as you make the decision to create a data-driven site:

- *Quality and timeliness of content.* The most immediate advantages to making a site data-driven are the speed with which new information can be presented on the Web, and the controls that can be put in place to guarantee the quality of this information. Rather than having to get a web designer to create a page containing the information, and then get it uploaded again every time a price changes or a new product is added, a tool can be created that enables the instant publishing of new or updated information simply by modifying the database. This is one of the key benefits of the Web over traditional media – the ability to view information in real time, rather seeing than a snapshot of old data. By enforcing rules on who can add and amend data, how it is checked, and whether it is approved, data can be verified prior to being published in a much more rigorous manner, ensuring that the user only sees accurate details.
- *Functionality.* The other main benefit of storing all of the data required for a site in a database is that of improved functionality in terms of the actions that the user can perform on the system. Rather than producing 'catalogues', which (like this book) just have an index and a contents table as a means of searching, forms can be created that allow the user to specify what is being looked for, and have the system scour the database for that information. A great example of this is a search engine. Without a database, such a site would present only a manual categorization of other web sites, with a huge structure of pages that you could (try to) navigate between.
- *Maintenance.* With the data for a site stored in a separate location from the presentation code, there is no longer a need to maintain static links in HTML files between related sections of a site, forcing you to reapply formatting and menu structures to hundreds of pages each time the site is redesigned. In a data-driven system, web pages are typically **templates** that act for entire classes of pages, rather than having one page for each piece of information.

  As an example of this, you could imagine the on-screen appearance of a page that displays the details of a product for sale. Rather than this being a separate HTML page, in a data-driven system there would be *one* page containing fields and tables that could be populated with data regarding *any* product. This means that there is far less to do each time a redesign is implemented. Similarly, as the relationship between different pieces of information can be stored in the database (rather than hard-coded in the pages), links to related products and other information can be generated on the fly.

Disadvantages

Although there are many advantages to making a web site data-driven, some of them come at a price, and a data-driven site is not always the right solution to your problem. There are several hurdles that must be overcome in order to provide a richer experience to the end user, and it's important that you consider them before taking the plunge:

- *Development.* A large number of web sites that are now data-driven started out being static, and there are still many static sites being created to this day. The nature of the content you want to present is not always suited to a data-driven site, and the creation of a data-driven system requires extra time and skills, resulting in a product that is more complex, and (inevitably) more prone to errors. These costs have to be weighed up against the advantages that such a system provides.
- *Performance.* The performance of data-driven web sites is an issue that crops up regularly. If a site is entirely static, then there are no constraints on the way the system is organized, or on how it can expand to cater for higher volumes of users. The simplest way to increase performance is to buy a faster processor and more memory. When that stops being viable, multiple versions of the site can be created, and users redirected to whichever one is under least load. This can continue in a linear fashion, with the same increase in performance each time a new web server is added.

  With a data-driven site, this is not the case, because the entire system is dependent upon one resource: the database. If it's not carefully designed, the database can create a bottleneck in the system, whereby the rest of the application is held up while it waits for information to be retrieved. Removing this bottleneck is a difficult problem to solve – having multiple synchronized databases is one of the few real solutions, but it can prove very expensive, and the overheads involved in this synchronization are significant.
- *Cost.* In addition to the technical considerations mentioned above, there are also associated commercial issues. For a relatively static site, the time required to create a database and write the code to access it may be longer than it would take just to edit some HTML pages. Also, enterprise-class database systems are themselves expensive. Considering Microsoft's data storage solutions alone, it's well known that producing a solution using SQL Server (Microsoft's enterprise-level database server) provides many benefits over Access (its desktop database), such as higher performance and better support for industry standards, but comes with a price tag to match.

Data Sources

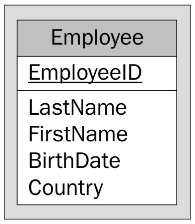
So: you've already considered some or all of the issues in the above lists, and you're still with us, which means that it's reasonable to assume you want to write a data-driven web application. The first question that needs to be answered, then, is where the information that will eventually end up on the user's screen is going to come from. Depending on factors such as the type of data, what operations are to be performed on the data, and the amount of use that is going to be made of the system, there are a multitude of options available. This section describes the reasons for and against using three of the most common data source types, along with an overview of the other types available.

Databases

When you start thinking about data sources, the most obvious one that springs to mind is the database, which will generally provide the most reliable, scaleable, and secure option for data storage. When you're dealing with large amounts of data, databases also offer the best performance. However, the very fact that other solutions exist is a sure indication that in some circumstances, they're not the best choice.

In general, databases are designed to store large amounts of data in a manner that allows arbitrary quantities of data to be retrieved in arbitrary order. For small collections of data, such as a set of contact details, the time and other costs involved in creating and accessing a database might outweigh the benefits that databases provide.

We'll have much more to say about the structure of databases in the next chapter, but as a quick example, wanting to store some information about a company employee in a database might move us to create a table called `Employee` that can contain the same pieces of data about a number of employees. Such information could include their `EmployeeID` (number), `LastName`, `FirstName`, `BirthDate`, and `Country`:



Throughout this chapter, comparisons and demonstrations will be made of how data can be stored and represented. For consistency, the same example is used throughout: that of storing details about the employees in an organization.

One thing to note when we display a database diagram, compared to the diagrams of other data sources, is that it's based on a *model* of the information being stored, rather than examples of the data. The way in which databases *actually* hold information is largely hidden from the outside world, leaving us to depict concepts rather than actual data items.

Text Files

At the opposite end of the scale from using databases to store information for a web site is the use of text files. Although text files can store information in almost any conceivable format, they are generally used for storing a set of data, one item on each line. If we were to capture the employee information detailed above, we could store the `LastName`, `FirstName`, `BirthDate`, and `Country` of two employees in a text file as follows:

```
Smith, John, 05-04-1979, UK
Bloggs, Joe, 29-09-1981, US
```

For simple information such as this, a text file provides an easy way of reading and writing data. If the data to be stored has more structure, however, it becomes far more time consuming. For example, it could be the case that each of these employees has placed an order for some office supplies. Rather than adding all of that information to the text file as well, it would be better to hold it separately, and then define relationships between the two sets of data.

When the data starts to gain 'structure' in this manner, a method of giving the file itself some structure must be found, and a way of retrieving it and representing it in memory must also implemented. One way of doing this is through the use of XML.

XML

In some ways, XML documents can be thought of as a stepping-stone between text files and databases; they store data using text files, but use a hierarchical and relational format that is both extensible and self-describing, providing a number of the benefits of a database system. Before we go any further in explaining the use of XML as a data source, a sample fragment of an XML document is shown below:

```
<company>
  <employees>
    <employee LastName="Smith" FirstName="John"
              BirthDate="05-04-1979" Country="UK" />
    <employee LastName="Bloggs" FirstName="Joe"
              BirthDate ="29-09-1981" Country="US" />
  </employees>
</company>
```

As you can see, the same information is being stored as in the text file, but there's also an indication of the nature of that information. You know that `29-09-1981` is the `BirthDate` of `Joe Bloggs`, because the data says so. Another benefit of XML is that it can contain multiple types of information in one document; a fragment like the one below could be inserted after `<employees>`:

```
<orders>
  <order ID="1">
    <product>Staples</product>
    <product>Pencils</product>
  </order>
  <order ID="2">
    <product>Biros</product>
    <product>Erasers</product>
  <order>
</orders>
```

Using the comprehensive functionality that's built into the XML-handling support provided by the .NET Framework (and other platforms), retrieving and manipulating the orders separately from the employees can be accomplished quite easily. This makes it possible to specify an order from the list for each employee by storing the ID of each order as part of the employee's details:

```
<employee LastName="Smith" FirstName="John"
          BirthDate="05-04-79" Country="UK" Order="2" />
```

XML is a powerful way of representing information, but in some circumstances performance can be a problem: updating and retrieving data from XML can be a time-consuming process. This is rarely an issue when a few users are accessing a small amount of data, but if there's a lot of data (or a lot of users) it can sometimes become one.

Other Sources

Between them, the three options enumerated above cover the main categories of data store, but there are many others that either fall between these, or follow a completely different paradigm. Most of the types that we haven't covered, though, are domain-specific – that is, that they've been developed to suit a specific task. On the Windows platform, typical examples of these include:

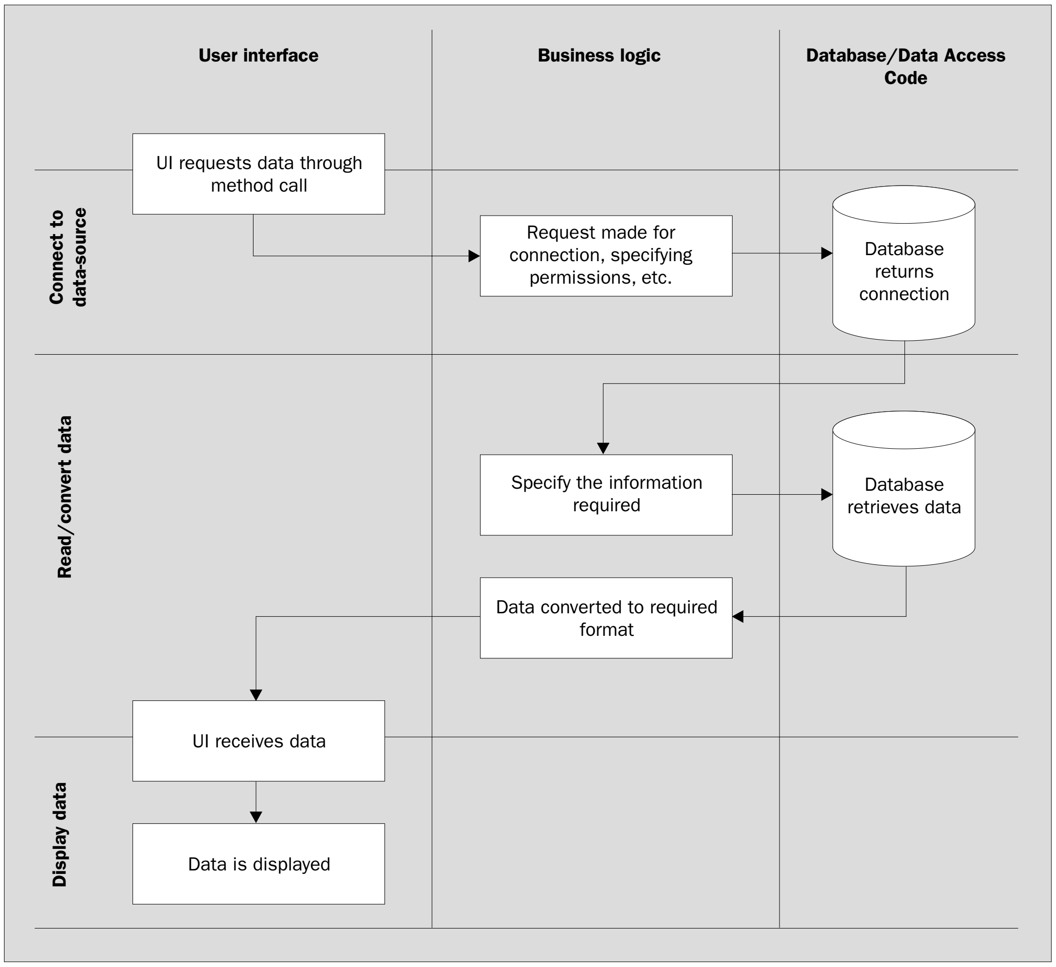
- *Microsoft Exchange Server* – the data store containing e-mail, calendar, and contact information
- *Active Directory* – the information that's stored by Windows-based servers regarding the users on the system, their permissions, etc.
- *Spreadsheets* – applications such as Excel store their data in a tabular format, a grid containing values that are used in tasks such as financial calculations.

In summary, although this book is focusing on databases (and uses them in the majority of its examples), it is important to remember that databases are not the only kind of data store, and that other mechanisms for storing data can often achieve the same goal more efficiently.

Retrieving Data from a Database

Regardless of the data store involved, there are three steps to using it that will be common to almost every web application you write. You need to 'connect' to the data source; you need to read the data (and possibly convert it or otherwise perform operations upon it); and you need to display the results. Before we begin to delve into the way that .NET deals with handling data, we'll elaborate on these three topics in a more general way, as a quick heads-up on how data-driven sites function.

The diagram below lays out the three steps mentioned above, and places them in context with the code that you'll need to write, and the data store itself.


User interface
Business logic
Database/Data Access Code
Connect to data-source
Read/convert data
Display data
UI requests data through method call
Request made for connection, specifying permissions, etc.
Database returns connection
Specify the information required
Database retrieves data
Data converted to required format
UI receives data
Data is displayed

Reading it from left to right, this diagram shows that there are three clearly separated aspects to the system: the application that requests the data, the code that communicates with the database and operates on the data, and the database itself. The details of this three-part structure and how it can be used to best effect are given later on in this chapter; for now, we'll continue on our top-to-bottom route.

- ❑ *Connecting to the data source.* Before we can issue any commands to the database or retrieve any data from it, we must create a **connection** to it. This provides a conduit through which we can send and retrieve data. To establish a connection, we need to specify such things as the type of database we are opening, where it is located, and any necessary security permissions. Once this has been done, and the connection has been opened, we can start to send instructions to the database.

- *Reading/converting the data*. Through the connection, we can tell the database to add, delete, and update records, to return information to us, and so on. As you can see from the diagram, there is more involved here than in the other steps. This is because the database expects commands to be in a different language from that of the application, and the application expects data to be returned in a different format from that stored in the database. Once information has *been* sent or retrieved, however, the connection to the database can usually be terminated. In special cases, an open connection is maintained, allowing data to be returned and displayed a little at a time, rather than all at once.
- *Displaying the data*. Once the data has been retrieved and converted into the correct format, it is usually transformed in some way to a format that's viewable by the user, such as an HTML table. Although there are far more *operations* during the reading and converting of the data, these largely happen behind the scenes, and in web applications it's often the case that presenting the information well takes the most time to implement. As we'll see later, however, ASP.NET offers us some help in this regard.

That's a lot of information, and lot of the things we'll be talking about in the first few chapters of this book where we will cover these issues more slowly, and in much greater depth. It also sounds like a lot of work – in fact, it *is* a lot of work – but mercifully we don't have to do all of it ourselves. Help is at hand in the form of ADO.NET, and that's our subject for the next section.

Introduction to ADO.NET

As described above, there are many different data stores that can provide information to an application. Microsoft realized a long time ago that having a single programming interface for accessing these diverse stores makes sense – it allows applications to make use of the latest versions of database servers with minimal changes to code, and it makes for interoperability between platforms.

With every new Microsoft platform comes a new way of accessing data stores. In the .NET Framework, the technology is called ADO.NET, but that builds upon the previous data-access functionality of technologies such as ADO, OLE DB, RDO, DAO, and ODBC. It provides an appropriate method for accessing data in modern applications that are more widely distributed than was previously the case.

After describing underlying technologies such as OLE DB and ODBC, this section will place ADO.NET into context with the technologies that came before it, and go on to explain the architecture of ADO.NET in a little detail.

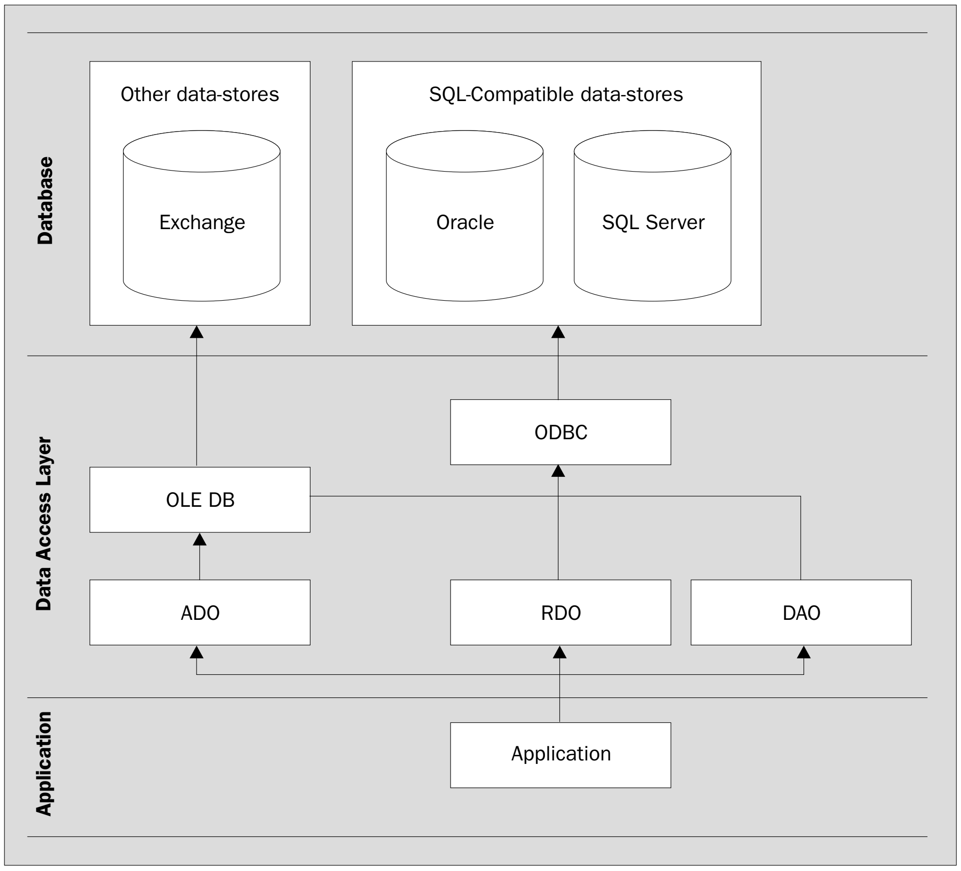
A History of Data Access on the Windows Platform

As soon as you start to think about accessing data on the Windows platform, you find yourself confronted with a list of abbreviations rather like the one in the second paragraph above. Here, we'll try to untangle the letters, and help you to understand how all of these technologies fit together – which, by and large, they do.

In the recent past, most applications communicated with data stores through the software objects provided by ADO, which made use of the lower-level technologies OLE DB and ODBC. In order for this to happen, ADO (and its replacement, ADO.NET) rely on a database conforming to an underlying set of standards. A significant difference between old and new is that ADO.NET has a less demanding, more flexible set of rules, allowing for a greater variety of data sources.

To allow applications to make connections to their databases, database vendors have to implement some common sets of functionality (interfaces) that have been devised for this purpose. One such interface is the highly successful **ODBC**, which is still supported by the vast majority of databases you'll come across. Another technology, **OLE DB**, was designed as the successor to ODBC, and was the cornerstone of Microsoft's Universal Data Access strategy. It too has become highly successful, and has gained broad support.

The diagram below shows a simplified version of how the various pre-.NET data access technologies connect to databases – and even in this diagram you can see that it was all getting a bit complicated! As well as ADO, OLE DB, and ODBC, technologies like RDO and DAO were getting involved too:



DAO

Let's start to put some meat on these bones. DAO (Data Access Objects) was Microsoft's first attempt at providing programmers with an object-oriented way of manipulating databases. It was invented for Access v1.0, and updated in later versions of Access and Visual Basic up to Access 97 and Visual Basic v5.0. Many of the original DAO commands have been retained through the years for backwards compatibility, meaning that the syntax required for performing operations can be quite ugly at times.

One of the biggest drawbacks of DAO is that it assumes data sources to be present on the local machine. While it can deal with ODBC connections to database servers such as Oracle and FoxPro, there are some things that make sense with remote data sources that cannot be achieved.

RDO

RDO (Remote Data Objects) is another object-oriented data access interface to ODBC. The methods and objects that it contains are similar in style to DAO, but they expose much more of the low-level functionality of ODBC. Although it doesn't deal very well with databases such as Access, its support for other large databases – Oracle, SQL Server, etc. – made it very popular with a lot of developers. This support focuses on its ability to access and manage the more complicated aspects of stored procedures (compiled commands used to maintain data in the database) and complex record sets (sets of data retrieved from the database).

ADO

ADO (ActiveX Data Objects) was first released in late 1996, primarily as a method of allowing ASP to access data, and initially supported only very basic client-server data-access functionality. Microsoft intended it eventually to replace DAO and RDO, and pushed everyone to use it, but at the time it only provided a subset of the features of two other technologies that were much more popular.

With the release of ADO v1.5, support for disconnected record sets was introduced, as was an OLE DB provider for Microsoft Access. With the release of ADO v2.0 in 1998, it went from being a subset of other technologies, to having a richer set of features. OLE DB drivers for both SQL Server and Oracle were released, meaning that access to enterprise-level database systems was feasible for the first time, and support for native providers (ones that didn't rely on ODBC) was added. Further increases in functionality were introduced in later versions up to v2.7.

ADO.NET

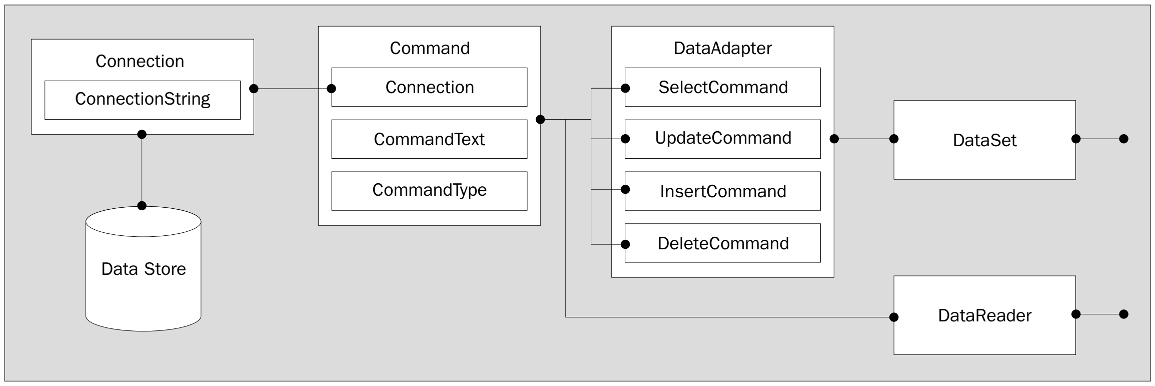
ADO.NET almost doesn't fit in this discussion – it's an entirely new data access technology that builds on the successes of ADO, but is only really related to it in name. The improvements lie in its support for different types of data store, its optimization for individual data providers, its utility in situations where the data is stored remotely from the client, and its ability to deal with applications where there are large numbers of users simultaneously accessing the data. The key to doing this is through features that separate it from the technologies that preceded it: the use of disconnected data, managed providers (we will look at both of these shortly), and XML.

ADO.NET Architecture

You now know that ADO.NET draws on a long history of data access. Almost inevitably, this means that there is quite a lot to learn. Thankfully, Microsoft has put a great deal of thought into its new data access technology, making it more logical and structured than previous attempts, while still providing a wealth of features.

ADO.NET is made up of a collection of objects, some of which are entirely new, and others of which have evolved from ADO. The main difference between these and their predecessors is that there is now generally only one way to accomplish a task – ADO was a little infamous for providing several means to exactly the same end!

The next few pages are concerned with taking a look at the main ADO.NET objects, and how they cooperate to provide data manipulation. Laid out below is a diagram of the five main object types that you'll be dealing with when you use ADO.NET:



If we work our way back from the database, taking the objects one by one, we can see how these objects work together, and what functions they perform:

- The **connection** object is the route through which all instructions to (and results from) the data store are sent. The user can specify which database to connect to, what authentication to use, and so on.
- The **command** object contains the instructions that specify what information should be sent to (or retrieved from) the database. It also contains a link to the connection that it's going to use.
- The **data reader** object provides a way of 'getting at' the information that's been retrieved by the command object. The information is provided on a read-only basis – so it can't be edited – and only one item of data is read at a time. Data readers provide an efficient (if inflexible) way of processing large amounts of data; they are sometimes described as providing **connected** access, since the connection with the database must remain open for as long as the data reader is in use.

- The **data adapter** object *represents* a set of commands and a database connection, providing an alternative method of retrieving data. It provides support for the data to be updated as well as just read, so in some ways it can be seen as a big brother to the data reader. Even so, the data adapter does not allow for direct editing of the data source; rather, it fills a dataset with a copy of information from the data source, and can then be used to write any changes to the data back to the database.
- The **dataset** can be thought of as a local copy of a portion of the data store. In this copy, rows of data can be read, added, edited, and deleted. Because the data is cached locally, it can be read in a random manner, as opposed to the forward-only manner of the data reader. When the required changes have been made to the data, they can be sent back to the data store through the data adapter. Until this point, the dataset is **disconnected** from the data store.

Looking to the far right of the diagram, you can see two unattached lines – this is where the 'front end' of your application connects to the ADO.NET architecture. The data that is returned here can be used in any way the developer chooses – displaying it to a web page, writing it out to a file, etc.

The twin concepts of "connected" and "disconnected" data are important ones, and while we've barely touched on them here, we'll be developing these ideas in later chapters – Chapters 4 and 5 in particular.

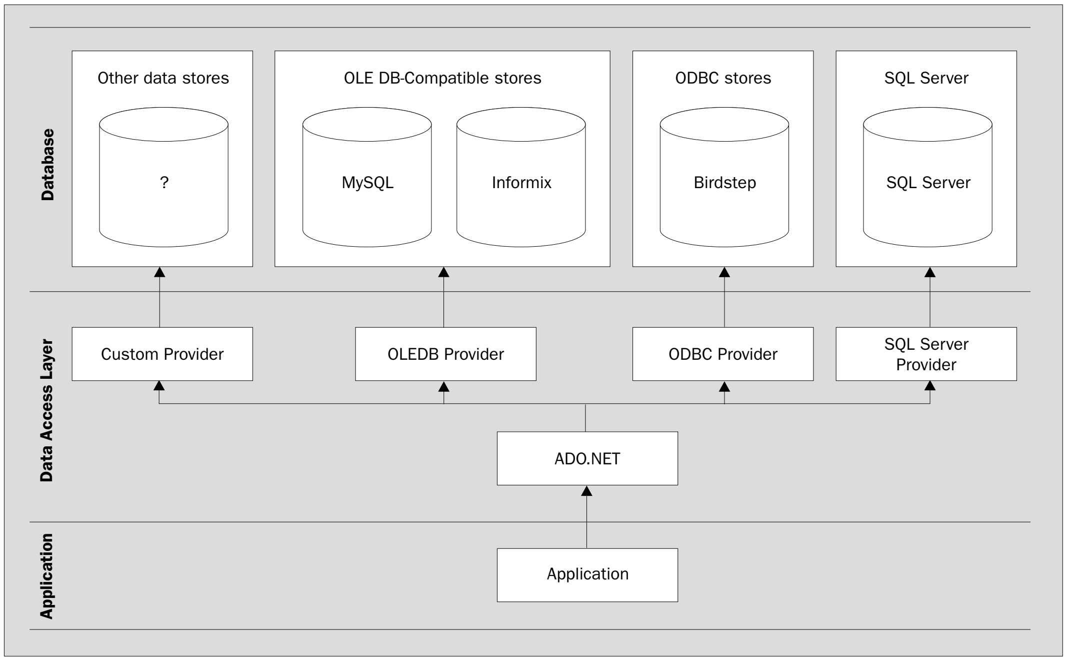
Data Providers

One of the key features of ADO.NET is that it's optimized for the various possible types of data store. Apart from the dataset, which is generic, the other objects in the above list have versions that are specifically geared towards accessing data of a particular type. For example, there are separate data reader classes for dealing with SQL Server and Microsoft Access databases. The umbrella term given to the 'set' of classes that deals with a particular type of data store is a **.NET data provider**.

As discussed, a data provider is a package of classes that implements a set of functionality allowing access to a specific type of data store. While there's a base set of functionality that a data provider must supply in order to be called as such, a particular data provider can have any number of extra properties and methods that are unique to the type of data store that is being accessed. This is very different from ADO, where there was a single set of classes that was used for accessing dissimilar data sources.

Where do Data Providers Fit in the Scheme of Things?

At this point, you're probably starting to think that you're getting a feel for the basic architecture of .NET, and that you can see why data providers allow for more types of data store to be accessed, but you don't know how the two relate to each other. Earlier on, we had a diagram of the technologies that were involved in data access before the introduction of ADO.NET. The following diagram shows how this changes – and how it gets simpler – under ADO.NET.



Standard Providers

Microsoft ships the .NET Framework with two data providers as standard: the SQL Server .NET data provider, and the OLE DB .NET data provider. The first of these provides a means of connecting to a SQL Server v7.0 (or later) database, and the classes that it comprises can be found in the `System.Data.SqlClient` namespace. The second allows access to any of the multitude of OLE DB-compatible data stores that are on the market, and implements similar functionality to the `SqlClient` provider; it resides in the `System.Data.OleDb` namespace.

A third data provider, which supports ODBC, is available but not installed by default; at the time of writing, it could be downloaded from http://msdn.microsoft.com/downloads/default.asp?URL=/downloads/sample.asp?url=/MSDN-FILES/027/001/668/msdncompositedoc.xml. Once installed, the classes for this provider can be found in the `Microsoft.Data.Odbc` namespace. Further data providers are under development, including one for Oracle that's in beta at the time of writing. This is also available for download from Microsoft's web site.

Data-driven Application Architecture

Although it's an important part of the puzzle, simply getting hold of a database and knowing the commands for retrieving and maintaining the data it contains does not guarantee the creation of an application that can achieve the goals of modern software development. In no particular order, these are:

- *Maintainability*. The ability to add and amend functionality to an application continually, without incurring large costs from having to revisit and re-implement existing portions of code.
- *Performance*. The application's ability to carry out its functionality responsively from the start, so users don't have to wait for lengthy periods for processing to complete, and resources on the machine are used responsibly.
- *Scalability*. The ability for an application to be extended in order to maintain performance when large numbers of users are accessing it simultaneously. Obviously, the more users there are, the more resources will be needed; scalability is the study of how rapidly the need for resources grows in relation to the increase in users, whether this need can be satisfied, and if so how this can be done.
- *Reusability*. The ultimate goal of software development; the ability to take functionality that has already been implemented and drop it into other projects and systems, removing the need for (re)development.

There have been many attempts to provide solutions to these problems, first through the introduction of proceduralization and modularization – the splitting of code into functions and separate files – and then through object-oriented techniques that hide the implementation from the user. Microsoft has produced guidelines on how applications developed on its platform should achieve the goals laid out above.

You can learn more about this subject in VB.NET Design Patterns Applied *(ISBN 1-86100-698-5), also from Wrox Press.*

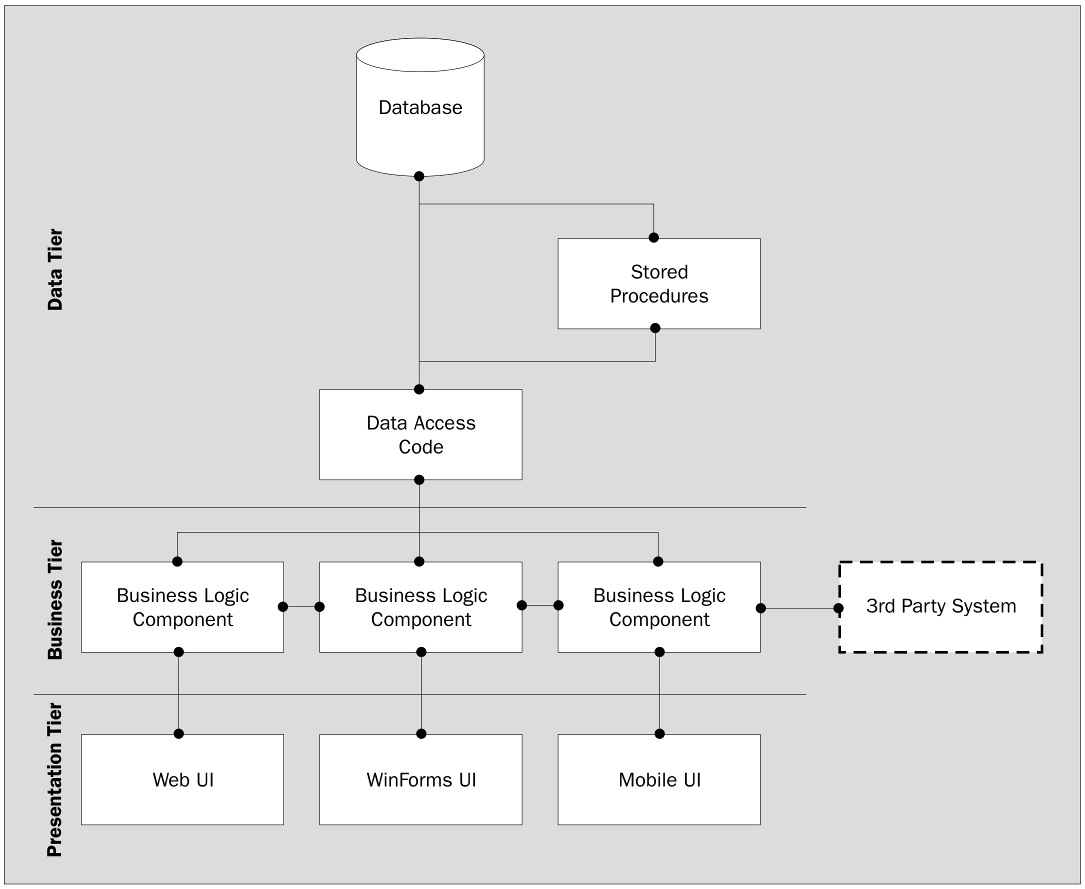
Earlier in the chapter, when we were talking about *Retrieving Data from a Database*, we briefly mentioned a "three-part structure" for web applications that deal with data access. In fact, this was a preview of the architecture that Microsoft (and many others) recommends. More formally, it's known as an **n-tier** architecture, which basically means that each application has code 'layers' that each communicate only with the layers around them. This section describes one of the most common approaches: the 3-tier model, which consists of the following:

- *Data tier*. Contains the database and stored procedures, and data access code.
- *Business tier*. Contains the business logic – methods that define the unique functionality of this system, and abstract these workings away from other tiers. This is sometimes referred to as the "middle tier".
- *Presentation tier*. Provides a user interface and control of process flow to the application, along with validation of user input.

While it could be argued that all applications make use of n-tier architectures, it has become commonplace to use the term to mean an application that has multiple tiers (where 'n' is greater than two) that abstract implementation details from each other.

Data Tier, Business Tier, and Presentation Tier

The 3-tier model has become the most popular due mainly to its simplicity. It's also the basis for all other models, which tend to break down the three tiers still further. The diagram below should help you to visualize the three tiers before we move on to their description. One thing you might want to keep in mind is that in ADO.NET, datasets are regularly passed between tiers, which means that the business and presentation tiers know more about the structure of the database than they would in a 'pure' model.



Data Tier

The data tier consists mainly of the table definitions, relationships, and data items that constitute the database itself, along with any pieces of code used for retrieving information from the database in its natural format. In SQL Server, these would be SQL statements that are usually kept in stored procedures.

One of the hardest issues to address is where the data tier ends, and where the business tier begins. This problem arises because it's quite easy to implement a lot of the functionality of the business logic (business tier) in the database code (as stored procedures). Where the line should be drawn is largely dependent upon the requirements of the application; whether rapid portability to other database server products is a key concern, or whether high performance is preferable. If the former is more important, then putting the majority of the functionality in the business tier is a preferable solution, and vice versa.

In the diagram above, there is a separate item called "data access code". This may simply represent the use of ADO.NET, or it may be a separate layer of functionality that hides ADO.NET from the other layers, preventing them from needing to know what type of data store(s) are being accessed.

Business Tier

The business tier of the application is where the majority of the application-specific functionality resides. Usually, this functionality consists of calling multiple atomic actions (`Read`, `Write`, `Delete` commands, etc.) in order to insulate the presentation tier from the complexities of the rules that the application must conform to. This tier also generally contains any links necessary to the methods exposed by third-party systems.

Contrary to popular object-oriented principles, any components created for the business tier of web applications should be **stateless** – that is, they should make use of functions and procedures that take in multiple parameters, rather than having properties that allow values to be set before making method calls. In .NET, the business tier is often implemented using class libraries, as we'll discuss in more detail in Chapter 10.

Presentation Tier

The presentation tier of the application is the only portion of the system that the end user gets to see, whether it's a collection of web pages, the interface to an application such as Outlook, or even a command prompt. This tier functions by making use of the functionality exposed to it via the business tier – it can never access the database (or any other portion of the data tier) directly. In this way, it is kept from being exposed to many of the details of how the application has been implemented, and can focus on providing the most usable presentation of the information and options possible.

As stated above, the nature of data access with ADO.NET is such that on occasion, these guidelines are not strictly adhered to. Indeed, we'll make use of this fact to abbreviate some of our early examples. But as we progress through this book, you'll find this architecture cropping up again and again.

Presenting Data with Controls

Since we just talked about presentation, it makes sense to take a moment here to discuss an important new feature of ASP.NET. One of the biggest hurdles to overcome in the past has been finding a common way of presenting the information that is being returned from the business tier. In a nutshell, this problem has arisen from the differences between the stateless world of web applications, and the world of client-server applications, where large amounts of data can be kept at the client.

Quite simply, data that was suited to one interface was often not suited to the other, due to the different mechanisms that were available for presenting it. Attempts were made to bridge this gap using ActiveX controls that could be created in languages such as Visual Basic, but these were specific to Windows machines running Internet Explorer, removing a lot of the benefit of having HTML-based applications.

Thankfully, this problem has been greatly lessened in .NET by the introduction of ASP.NET **web server controls**. These behave like ActiveX controls when you're developing code, but by the time the user sees them in their browser, they've been converted into standard HTML elements. The main advantage in doing this is that it brings development for all platforms into line (even minority platforms such as WAP applications, through the use of mobile controls). A single source of data can be used to populate all of the different control types, which include such things as drop-down lists, check box lists, and the data grid, which is a highly versatile kind of table.

This book contains examples of working with all of the most important web server controls, starting with the first exercises in Chapter 3.

Data Binding to Controls

The key to allowing a single source of data to provide all controls with exactly what they need is **data binding**. This is a technique where, when given a data source in the correct format, the control itself decides what pieces of information are useful to it, and which are irrelevant.

Due to the highly structured nature of the .NET Framework, many different data sources can be used for data binding, as long as they implement the right interfaces. As well as datasets, standard data structures such as arrays and hash tables also fit the bill. The same types of objects that can be used to add items to a drop-down list control can also be used to populate a data grid, or any other web server control.

If similar end-user functionality was to be implemented in classic ASP, the developer would generally have to loop through a set of data, making calls to `Response.Write()`, and creating lines of HTML with `<option>` tags interspersed throughout. From this alone, you should start to get an idea of just how much the use of web server controls allows the developer to forget about the actual target language (HTML) they're developing for, and concentrate on what it is that they are trying to present to the user. The ASP.NET web server controls can even adjust themselves to support different browsers, so the need for developers to consider what platform the application will run on is also greatly reduced.

As the internal workings of the controls that .NET provides are hidden from the developer, support is not only guaranteed for current browsers, but also can be implemented for future ones without requiring changes to the applications that use them. This will largely remove the need for browser and version testing, traditionally an expensive part of the development cycle.

Custom Controls

While the controls that come built into .NET provide us with a great deal of functionality, and cater for most situations, there are always circumstances where we require something unique. After all, if applications were all the same, then there would be none left to write by now! When we find ourselves in this situation, .NET doesn't leave us in the lurch. We can create our own controls that support all of the features of the built-in ones, and more. We don't even have to write these controls from scratch, because custom controls can be created:

- By deriving a new one from an existing control, and adding the required functionality
- By composing a new custom control using two or more existing controls
- By creating a control from scratch, making use of base controls such as `Tables` and `Input` elements

Although it's not a subject we'll be pursuing further in this book – we've got enough on our plate as it is! – this aspect of ASP.NET web server controls is yet another reason to be enthusiastic about the wealth of options available for data access under ASP.NET.

The Microsoft SQL Server Desktop Engine

As our final act in this chapter, we need to do something that will set us up for the rest of the book. During the course of the discussion so far, we've mentioned the names of a number of different databases, but it can't have escaped your attention that if we're going to demonstrate anything useful in the chapters to come, we need to set up a database of our own.

Our choice is to use the Microsoft SQL Server Desktop Engine (MSDE), which is a specialized version of SQL Server 2000. In this section, we'll explain what it is, why we've chosen to use it, and – most important of all – how you can get hold of it and install it.

A Smaller SQL Server

The first thing to say about MSDE is that it's entirely compatible with SQL Server, which is truly an enterprise-class database server. This means that the things you learn while using MSDE will stand you in good stead when you come to use SQL Server itself – it behaves in exactly the same way. From our perspective here, though, the immediate benefits of MSDE are:

- It's freely distributable
- It's currently sitting on your Visual Basic .NET discs, just waiting for you to install it

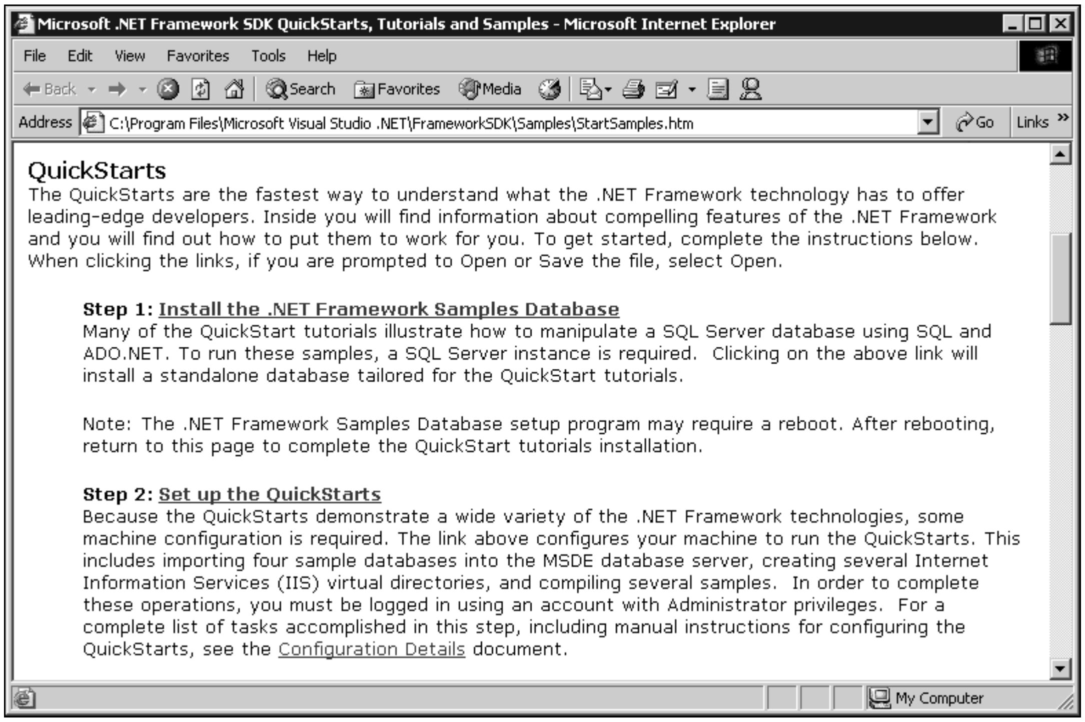
What this means is that as well as providing the perfect system for us to learn and experiment with, a complete web application can initially be produced and distributed without incurring any costs for the database server. If the system expands at a later date, it can be ported to the commercial distribution of SQL Server with next to no effort. The only features cut down from the full version of SQL Server are that the MSDE is optimized for (but not limited to) up to five connections at a time, that the maximum database size is limited to 2GB, and that some enterprise features are absent.

Throughout this book, the code samples and text will assume that MSDE is being used as the data provider, with the Northwind database (which is included with it) acting as the data source. To ensure that the code in the book will all function correctly, the next section details the installation of MSDE.

All of the features that MSDE supports are also supported by SQL Server. The converse is not true, however; some of the richer functionality of SQL Server is not present in MSDE. However, none of this functionality is required for any of the code in this book to operate correctly.

Obtaining and Installing MSDE

When Visual Basic .NET (or any of the various Visual Studio .NET products) is installed, an item called Microsoft .NET Framework SDK is added to your Start | Programs menu. Beneath this is an item called Samples and QuickStart Tutorials; if you select it, this is what you'll see:


Microsoft .NET Framework SDK QuickStarts, Tutorials and Samples - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address C:\Program Files\Microsoft Visual Studio .NET\FrameworkSDK\Samples\StartSamples.htm
Go Links
My Computer

QuickStarts

The QuickStarts are the fastest way to understand what the .NET Framework technology has to offer leading-edge developers. Inside you will find information about compelling features of the .NET Framework and you will find out how to put them to work for you. To get started, complete the instructions below. When clicking the links, if you are prompted to Open or Save the file, select Open.

Step 1: Install the .NET Framework Samples Database
Many of the QuickStart tutorials illustrate how to manipulate a SQL Server database using SQL and ADO.NET. To run these samples, a SQL Server instance is required. Clicking on the above link will install a standalone database tailored for the QuickStart tutorials.

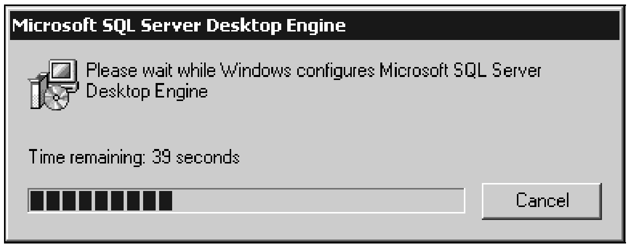
Note: The .NET Framework Samples Database setup program may require a reboot. After rebooting, return to this page to complete the QuickStart tutorials installation.

Step 2: Set up the QuickStarts
Because the QuickStarts demonstrate a wide variety of the .NET Framework technologies, some machine configuration is required. The link above configures your machine to run the QuickStarts. This includes importing four sample databases into the MSDE database server, creating several Internet Information Services (IIS) virtual directories, and compiling several samples. In order to complete these operations, you must be logged in using an account with Administrator privileges. For a complete list of tasks accomplished in this step, including manual instructions for configuring the QuickStarts, see the Configuration Details document.

This page is self-explanatory: clicking on the first link will install the MSDE engine; clicking on the second will cause the sample databases – including Northwind, which we'll be using throughout this book – to be created.

> *This page will only appear once, so if you (or someone else) have been here before, you won't see it. Don't worry: you'll find the* `instmsde.exe` *and* `configsamples.exe` *files that these links invoke beneath the* `FrameworkSDK\Samples` *folder of your Visual Studio installation.*

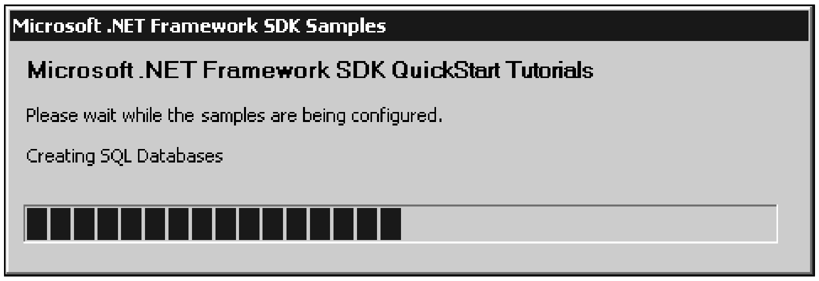
Ensure that you're logged on as a user with Administrator privileges on the current machine, and click on the first link (or run the executable). The following dialog will appear:


Microsoft SQL Server Desktop Engine

Please wait while Windows configures Microsoft SQL Server Desktop Engine

Time remaining: 39 seconds

When this has finished, there's no need to restart your machine. You can go straight on to the next step, which will produce another dialog:


Microsoft .NET Framework SDK Samples

Microsoft .NET Framework SDK QuickStart Tutorials

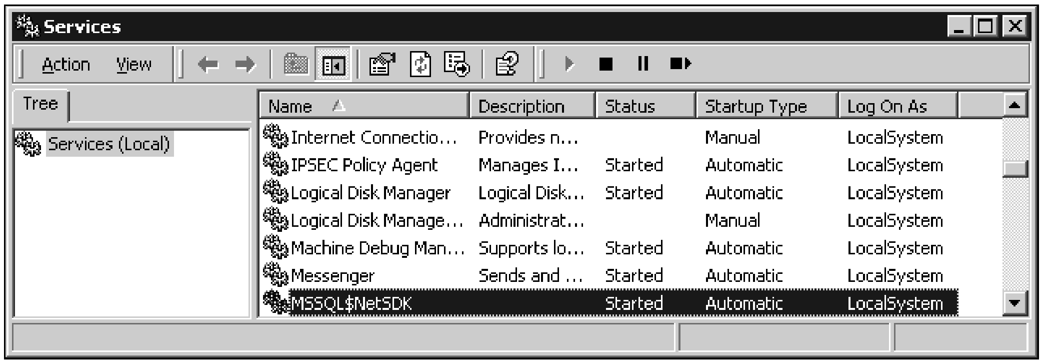
Please wait while the samples are being configured.

Creating SQL Databases

Once again, wait for this step to finish its work... and you're all done. But what exactly *have* you done? The best way to understand that is to open up Visual Basic .NET, ready for the quick tour in the next section.

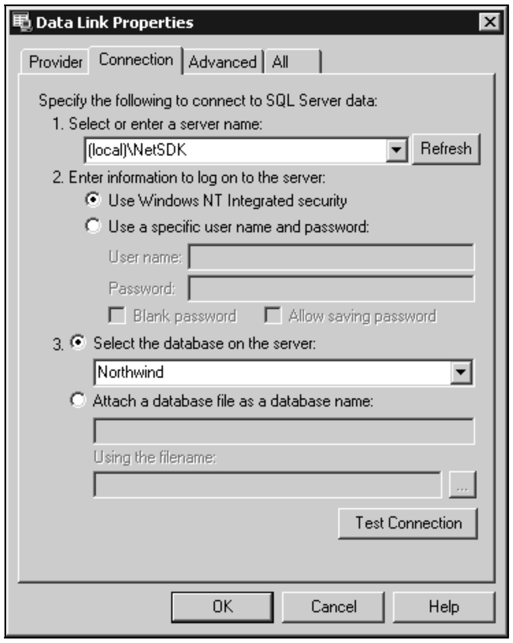
Using MSDE

Once it has been successfully installed on the local machine, you need to make sure that the MSDE service has started. This procedure differs slightly from platform to platform, but the instruction here, for Windows 2000, should tell you all you need to know. From the Start Menu, open the Control Panel, and go to Administrative Tools | Services:



Name	Description	Status	Startup Type	Log On As
Internet Connectio…	Provides n…		Manual	LocalSystem
IPSEC Policy Agent	Manages I…	Started	Automatic	LocalSystem
Logical Disk Manager	Logical Disk…	Started	Automatic	LocalSystem
Logical Disk Manage…	Administrat…		Manual	LocalSystem
Machine Debug Man…	Supports lo…	Started	Automatic	LocalSystem
Messenger	Sends and …	Started	Automatic	LocalSystem
MSSQL$NetSDK		Started	Automatic	LocalSystem

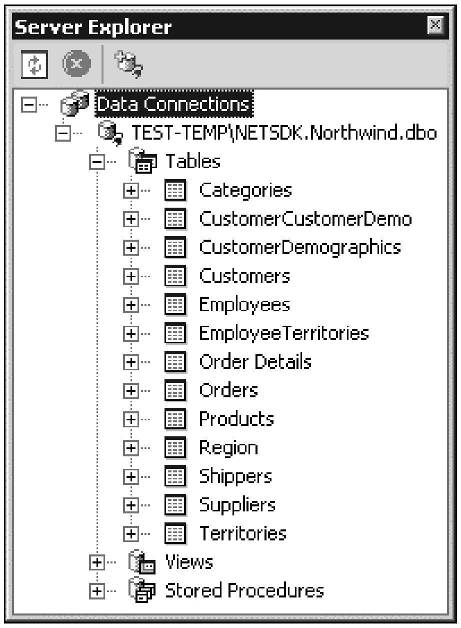
The service called MSSQL$NetSDK *is* MSDE. Make sure that the Status and Startup Type are set to Started and Automatic, as they are here, and you can rest assured that from now, when Windows is running, MSDE will be running too. Close this window, head into Visual Studio, and choose the View | Server Explorer menu item. Right-click on the Data Connections item at the top of the Server Explorer window, choose Add Connection from the context menu, and you'll see the following:



Specify the following to connect to SQL Server data:

1. Select or enter a server name: (local)\NetSDK
2. Enter information to log on to the server:
   - [✓ Use Windows NT Integrated security]
   - Use a specific user name and password:
   - User name:
   - Password:
   - Blank password   Allow saving password
3. [✓ Select the database on the server:] Northwind
   - Attach a database file as a database name:
   - Using the filename:

On choosing the appropriate settings in this dialog, the Server Explorer allows you to browse SQL Server databases, examining their content and performing some simple operations. To view the Northwind database that we just installed, you should make your dialog match the screenshot above. When you return to the Server Explorer, you'll be rewarded with a view of the database:



Having made it this far, you can be sure that the MSDE database is ready for action, and with that our work in this chapter is done. It's been quite a fast ride, but we've covered a lot of ground in the hope that it will look familiar when you see it again in the chapters ahead. That slower, more careful journey begins in the next chapter.

Summary

In this chapter, we have discussed the advantages and disadvantages of creating data-driven sites, showing that although they can provide a wealth of functionality, there are problems associated with their creation. This was followed by an introduction to data sources, and how they are handled using ADO.NET. The architecture of an application that involves data access was then covered, showing why the separation of data and presentation can provide many benefits to developers. Finally, details about MSDE – the database server we'll use in examples throughout this book – were given, and the product was installed.

Now that the background to the creation of data-driven web sites in .NET has been dealt with, we can move on to finding out the details of each of the technologies, and putting them to use. In the next chapter, we'll look at some of the theory involved in creating databases, along with a quick introduction to the SQL language, and a tour of the Northwind database. In Chapter 3, our brief description of connecting to databases will be expanded to cover the finer points, along with other topics such as the storing of connection strings. Following this, Chapters 4 and 5 are concerned with data readers and datasets, with Chapters 6 and 7 covering the adding, modification, and deletion of records. The rest of the book then makes use of this knowledge of ADO.NET, discussing the creation of components, applications, and stored procedures, and looking at performance issues. We finish with a case study, which pulls all this knowledge together in a practical application.