1

Introducing .NET

The .NET Framework and Visual Studio are the main components of .NET

What's required to create good software? While it's possible to write first-rate code in almost any environment, creating good software is much easier when the right platform and tools are available. For most Windows developers today, that platform is defined by .NET. While defining .NET clearly was once a challenge, it's now clear that the .NET label refers primarily to two things. They are:

- *The .NET Framework*, which consists of the *Common Language Runtime (CLR)* and the *.NET Framework class library.* The CLR provides a standard foundation for building applications, while the .NET Framework class library offers a large set of standard classes and other types that can be used by any .NET Framework application written in any language.
- *Visual Studio*, an integrated development environment (IDE) for creating Windows applications. While this tool can be used to build software that runs directly on

Perspective: .NET's Naming Journey

It makes sense today to think of the name ".NET" as primarily referring to the .NET Framework and Visual Studio. Things weren't always so simple, however. When .NET was first announced in the summer of 2000, Microsoft applied the term to a broad range of things. Today's .NET technologies were included, of course, but so were several other things. Many of Microsoft's server products, including SQL Server and BizTalk Server, were grouped together as the *.NET Enterprise Servers*, for example, and a wholly separate effort eventually known as *.NET My Services* was launched. There was even talk about a possible Windows .NET and Office .NET sometime in the future.

But was there a common technical underpinning for all of these things? Sadly, the answer was no. When Microsoft first sprang .NET on the world, it treated the term as a broad brand, one that could be applied to pretty much anything the company was doing. The result was a good deal of confusion among Microsoft's customers.

Thankfully, the story has gotten much simpler. The .NET Enterprise Servers are now considered part of the Windows Server System, and so they've lost the .NET tag. .NET My Services faded from the scene, while the branding boffins in Redmond decided against tacking the .NET brand onto either Windows or Office. Today, when somebody says ".NET," they're referring to the .NET Framework and Visual Studio.

Windows, its main focus is helping developers create .NET Framework applications. Visual Studio supports several programming languages for creating these applications, including C#[1], Visual Basic (VB), and C++.

Various versions exist for both of these technologies. The versions described in this book are those released by Microsoft

1. Pronounced "C sharp," as in the musical note.

in late 2005: version 2.0 of the .NET Framework and Visual Studio 2005.

The .NET Framework

The .NET Framework is a foundation for creating Windows applications

The heart of .NET is the .NET Framework. First released in 2002, it brought enormous change to the lives of those who write Windows software and the people who manage them. Figure 1-1 shows the Framework's two main parts: the CLR and the .NET Framework class library. A .NET application always uses the CLR, and it can also use whatever parts of the class library it requires.

The CLR provides a common basis for all .NET languages

Every application written using the Framework depends on the CLR. Among other things, the CLR provides a common set of data types, acting as a foundation for C#, VB, and all other languages that target the .NET Framework. Because this foundation is the same no matter which language they choose, developers see a more consistent environment.

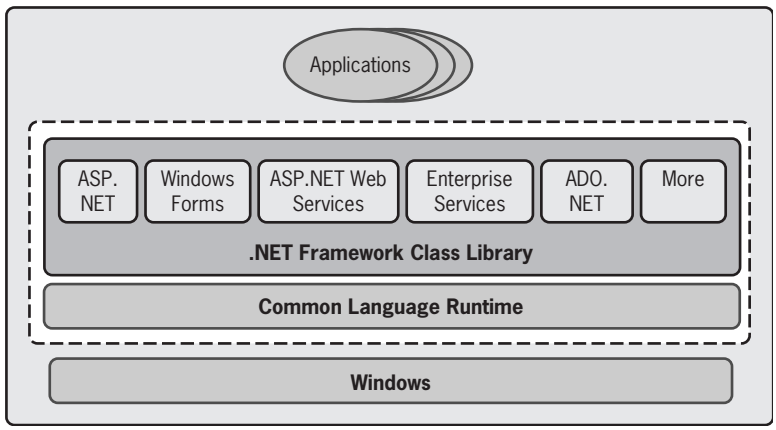
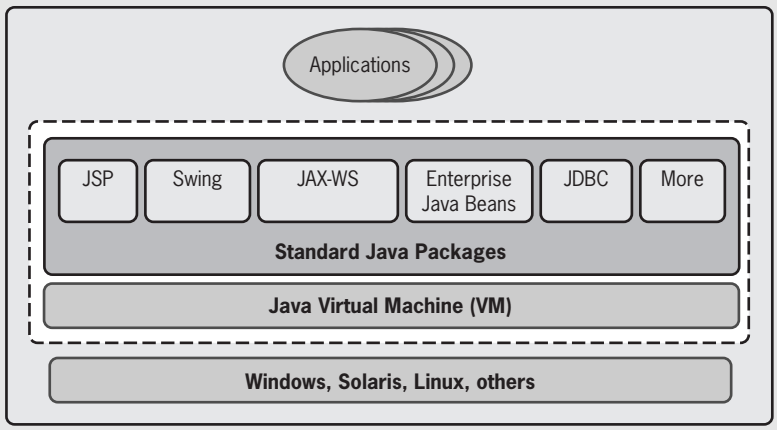


Figure 1-1 The .NET Framework consists of the Common Language Runtime (CLR) and the .NET Framework class library.

Perspective: The .NET Framework vs. the Java Environment



Mainstream software development today has split largely into two camps. Microsoft, promoting the .NET Framework, is in one, while most other vendors backing the Java environment are in the other. Each technology has its fans and detractors, and each has a substantial installed base today.

These competing worlds are strikingly similar. To see how similar, compare the figure above with Figure 1-1. Both environments support the same kinds of applications, and both provide a large standard library to help build those applications. The Java library, mostly known today as Java 2 Enterprise Edition (J2EE or just JEE) includes Java Server Pages (JSP) for Web scripting, Swing for building GUIs, JAX-WS (formerly JAX-RPC) for Web services–based communication, Enterprise JavaBeans (EJB) for building scalable server applications, JDBC for database access, and other classes. These technologies are analogous to the .NET Framework's ASP.NET, Windows Forms, ASP.NET Web Services, Enterprise Services, and ADO.NET, respectively. The Java virtual machine is also much like the .NET Framework's CLR, and even the semantics of the dominant languages—Microsoft's C# and VB vs. Java—are quite similar.

There are also differences, of course. One obvious distinction between the two is that the Java environment runs on diverse operating systems, while the .NET Framework focuses on Windows. The trade-off here is clear: Portability is good,

but it prevents tight integration with any one system, and integration is also good. You can't have everything, at least not all at the same time. Also, Java-based products are available from multiple vendors, while only Microsoft provides the .NET Framework. Different Java vendors can provide different extensions to the core specifications, so developers can get somewhat locked into a single vendor. Still, portability across different Java platforms is possible, while the .NET Framework unambiguously ties your application to Microsoft.

This bifurcation and the competition it engenders are ultimately a good thing. Both camps have had good ideas, and each has borrowed from the other. Having one completely dominant technology, whether the .NET Framework or Java, would produce a stultifying monopoly, while having a dozen viable choices would lead to anarchy. Two strong competitors, each working to outdo the other, is just right.

The .NET Framework class library provides standard code for common functions

Applications written in any .NET language can use the code in the .NET Framework class library. Among the most important technologies provided in this library are the following:

- **ASP.NET:** Classes focused on building browser-accessible applications.
- **Windows Forms:** Classes for building Windows graphical user interfaces (GUIs) in any CLR-based programming language.
- **ASP.NET Web Services (also called ASMX):** Classes for creating applications that communicate with other applications using Web services.
- **Enterprise Services:** Classes that provide distributed transactions, object instance control, and other services useful for building reliable, scalable applications.
- **ADO.NET:** Classes focused on accessing data stored in relational database management systems (DBMS).

The .NET Framework class library contains much more than this short list indicates. Among the other services it provides are support for creating and working with XML documents, a variety of security services, and mechanisms for interoperating with applications built using older Windows technologies such as the Component Object Model (COM).

The .NET Framework supports various kinds of applications

As this short description suggests, the .NET Framework class library can be used to create many different types of applications. And because all of the services in this library are built on the CLR, applications can combine them as needed. A browser application built using ASP.NET, for example, might use ADO.NET to access stored data and Enterprise Services to perform distributed transactions.

A .NET Framework application consists of managed code

Software that uses the .NET Framework (and thus relies on the CLR) is referred to as *managed code*. As Figure 1-2 shows, an application can be built solely from managed code, relying entirely on the CLR and the relevant parts of the .NET Framework class library. An application can also be built from a combination of managed code and ordinary unmanaged code, with the two interacting as necessary. This second option, shown on the

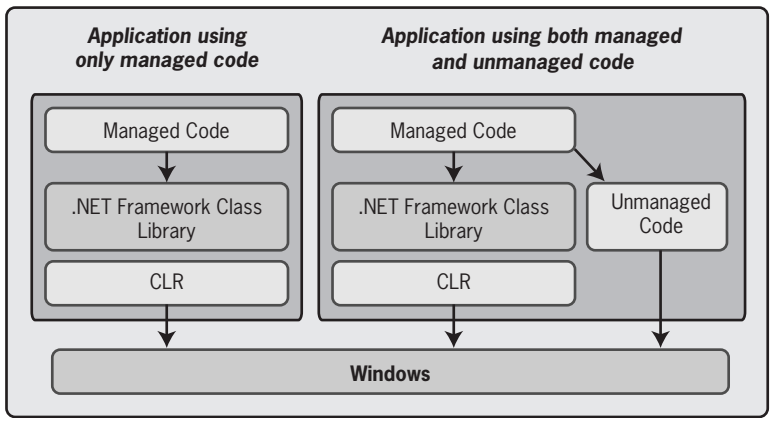


Figure 1-2 An application can be built entirely from managed code or from a combination of managed and unmanaged code.

right side of the figure, is especially important for existing applications. Most new Windows applications created today are built wholly in managed code, but it can also be useful to extend pre-.NET applications with managed code. And although it's not shown in the figure, it's still possible to create new Windows applications entirely in unmanaged code—using the .NET Framework isn't obligatory.

Managed code is typically built using managed objects

Managed code is typically object-oriented, so the objects it creates and uses are known as *managed objects.* A managed object can use and inherit from another managed object even if the two are written in different languages. This fact is a key part of what makes the .NET Framework class library an effective foundation: Objects written in any CLR-based language can inherit and use the class library's code. Given the fundamental role played by the CLR, understanding the .NET Framework begins with understanding this runtime environment.

The Common Language Runtime

Built from scratch to support modern applications, the CLR embodies a current view of what a programming environment should be. While it's hard to claim complete originality for any idea in computer science today, it is fair to say that this essential .NET technology takes an interesting new approach to programming languages.

What the CLR Defines

There's widespread agreement on the features a modern programming language should offer

Think about how a programming language is typically defined. Each language commonly has its own unique syntax, its own set of control structures, a unique set of data types, its own notions of how classes inherit from one another, and much more. The choices a language designer makes are driven by the target applications for the language, who its users are meant to be, and the designer's own sensibilities.

Yet most people agree on much of what a modern general-purpose programming language should provide. While opinions

Looking Backward: Windows DNA and COM

For most of the 1990s, application developers in the Microsoft environment relied on a set of technologies that became known as *Windows DNA.* Those technologies included the Component Object Model (COM) and Distributed COM (DCOM), a larger group of COM-based technologies known collectively as COM+, support for creating browser applications using Active Server Pages (ASP), data access support with ActiveX Data Objects (ADO), and others. The most commonly used languages for building Windows DNA applications were VB and C++, both supported by earlier versions of Visual Studio.

Tens of thousands of applications based on these technologies are in production today, providing solid evidence of Windows DNA's success. Yet the technologies Windows DNA includes were developed independently over a period of several years. Because of this, the integration among them wasn't as complete as it might have been. For example, while the Windows DNA environment let developers use various programming languages, each language has its own runtime libraries, its own data types, its own approach to building GUIs, and other differences. Applications written in different languages also accessed system services in different ways. C++ applications could make direct calls to the operating system through the Win32 interface, for instance, while VB applications typically accessed these services indirectly. These differences made life challenging for developers working in more than one language. COM, by defining common conventions for interfaces, data types, and other aspects of interaction among different software, was effectively the duct tape that held this complex environment together.

By providing a common foundation that can be used from all languages, the .NET Framework significantly simplified life for Windows developers. Applications built on the .NET Framework don't face many of the problems that COM addresses—.NET Framework applications all use the CLR, for example, which defines a common approach to interfaces and other data types—and so the glue between different languages that COM provides is no longer necessary. This is why COM technology isn't used in building pure .NET Framework applications. Instead, developers can build software that interacts in a more natural and substantially simpler way.

For the most part, the arrival of the .NET Framework was the death knell for Windows DNA and COM. It's taken quite a while for organizations to migrate to the new world of .NET, and many applications built with these older Windows technologies are still in production. Still, with few exceptions, serious new Windows development today uses .NET, not Windows DNA.

The CLR defines a common set of semantics that is used by multiple languages

on syntax differ—some developers love curly braces, others abhor them—there's widespread agreement on what semantics a language should offer. Given this, why not define a standard implementation of those semantics, then allow different syntaxes to be used to express those semantics?

The CLR also provides other common services

The CLR provides this standard implementation. By providing a common set of data types such as integers, strings, classes, and interfaces, specifications for how inheritance works, and much more, it defines a common set of semantics for languages built on it. The CLR says nothing about syntax, however. How a language looks, whether it contains curly braces or semicolons or anything else, is entirely up to the language designer. While it is possible to implement languages with varying behaviors on top of the CLR, the CLR itself provides a consistent, modern set of semantics for a language designer to build on.

Along with its standard types, the CLR provides other fundamental services. Those services include the following:

- Garbage collection, which automatically frees managed objects that are no longer referenced.
- A standard format for metadata, information about each type that's stored with the compiled code for that type.
- A common format, called *assemblies*, for organizing compiled code. An assembly can consist of one or more Dynamic Link Libraries (DLLs) and/or executables (EXEs), and it includes the metadata for the classes it contains.

A single application might use code from one or more assemblies, and so each assembly can specify other assemblies on which it depends.

Using the CLR

The CLR supports many different programming languages

The CLR was not defined with any particular programming language in mind. Instead, its features are derived largely from popular existing languages, such as C++, the pre-.NET version of VB, and Java. Today, Microsoft provides several CLR-based languages, including C#, the .NET version of VB, an extended

Perspective: Standardizing C# and the CLR

C# and a subset of the CLR called the Common Language Infrastructure (CLI) are now official international standards. Microsoft originally submitted them to the ECMA standards organization, and they've also been approved by the International Organization for Standardization (ISO). Along with C#, the standardized technologies include the syntax and semantics for metadata, MSIL (rechristened the Common Intermediate Language, or CIL), and parts of the .NET Framework class library. For more details on exactly what has been submitted and its current status, see http://msdn.microsoft.com/net/ecma.

The biggest effect of establishing C# and the CLI as standards is to let others more easily implement these technologies. The most visible non-Microsoft implementation of .NET is surely the Mono project (http://www.mono-project.com). Mono's ambitious goal is to implement at least a large part of what Microsoft has given to ECMA, including a C# compiler and the CLI, along with other parts of the .NET Framework. Mono's creators say that they were attracted to the CLR for technical reasons, which must please Microsoft. (In fact, it's possible to view the CLI as the specification of a system, while .NET's CLR is just the Microsoft implementation of this specification.) Now led by Novell, the Mono project qualifies as a success in many ways, with implementations available for Linux and other operating systems. If nothing else, I admire the ambition and ability of the people who created it.

version of C++, and others. Third parties also provide languages built on the CLR.

Managed code is always compiled first into MSIL

No matter what language it's written in, all managed code is compiled into *Microsoft Intermediate Language (MSIL)* rather than a machine-specific binary. MSIL (also referred to as just IL) is a set of CPU-independent instructions for performing typical operations such as loading and storing information and calling methods. Each DLL and EXE in an assembly contains MSIL rather than processor-specific code. Installing a .NET Framework application on your system really means copying to your disk files that contain MSIL rather than a machine-specific binary. When the application is executed, MSIL is transformed into native code before it's executed.

Each method is typically JIT compiled the first time it's invoked

Figure 1-3 illustrates the process of compiling and executing managed code. Source code written in VB, C#, or another language that targets the CLR is first transformed into MSIL by the appropriate language compiler. As the figure shows, the compiler

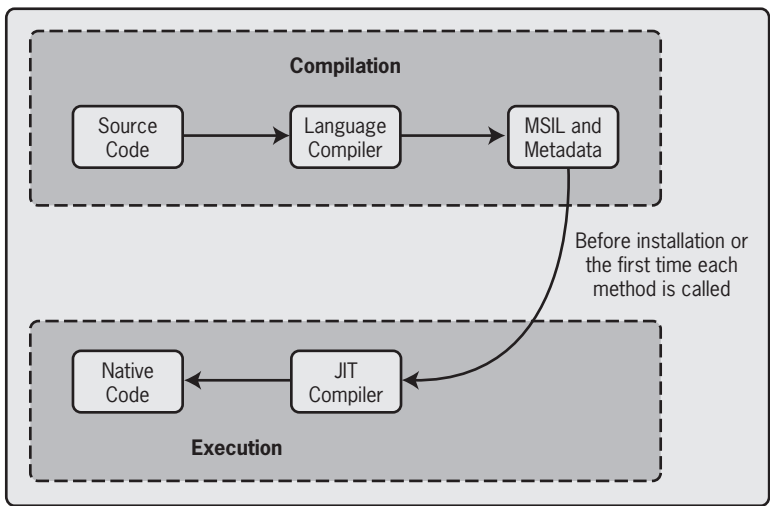


Figure 1-3 All managed code is compiled first to MSIL, then translated into native code before execution.

also produces metadata that's stored in the same file as the MSIL. Before execution, this MSIL is compiled into native code for the processor on which the code will run. By default, each method in a running application is compiled the first time that method is called. Because the method is compiled just in time to execute it, this approach is called *just in-time (JIT)* compilation.

All .NET Framework–based languages have about the same level of performance

One point worth noting is that any language built on the CLR should exhibit roughly the same performance as any other CLR-based language. Unlike the pre-.NET world, where the performance difference between VB and C++ was sometimes significant, a .NET Framework application written in C# isn't noticeably faster than the same application written in VB. While some compilers may produce better MSIL code than others, large variations in execution speed are unlikely.

The CLR is the foundation of everything else in the .NET Framework. All code in the .NET Framework class library depends on it, as do all Framework-based applications. Chapter 2 provides a more detailed look at the technology of the CLR.

The .NET Framework Class Library

The .NET Framework class library can be used from any CLR-based language

The .NET Framework class library is exactly what its name suggests: a library of classes and other types that developers can use to make their lives easier. While these classes are themselves written in C#, they can be used from any CLR-based language. Code written in C#, VB, C++, or any other language supported by the .NET Framework can create instances of these classes and call their methods. That code can also rely on the CLR's support for inheritance to inherit from the library's classes.

Surveying the Library

The .NET Framework class library is organized as a tree

The contents of the .NET Framework class library are organized into a tree of namespaces. Each namespace can contain types, such as classes and interfaces, and other namespaces. Figure 1-4

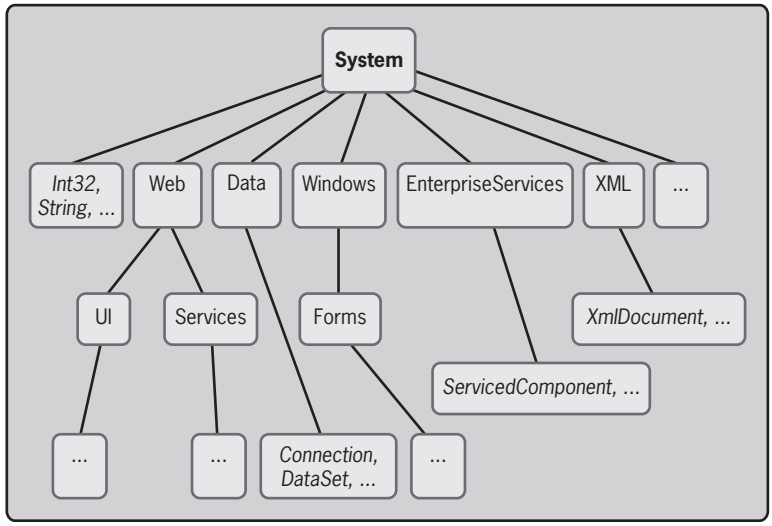


Figure 1-4 The .NET Framework class library is structured as a hierarchy of namespaces, with the System namespace at the root.

The .NET Compact Framework

While the .NET Framework is useful for writing applications on desktops and server machines, it can also be used with smaller devices, such as mobile phones, PDAs, and set-top boxes. Small devices are becoming more and more important, and they're an important piece of Microsoft's overall business strategy. These devices typically have less memory, however, so they're unable to run the complete .NET Framework. The .NET Compact Framework addresses this issue. By eliminating some parts of the .NET Framework class library, it allows use of the Framework in smaller devices.

The .NET Compact Framework targets the Windows CE operating system, but because it's built on the same foundation used in larger systems, developers can use Visual Studio as their development environment. Organizations that must create software for a range of devices can now use the same languages, the same tools, and much of the same development platform to target systems of all sizes.

shows a very small part of the .NET Framework class library's namespace tree. The namespaces shown include the following:

- **System:** The root of the tree, this namespace contains all of the other namespaces in the .NET Framework class library. System also contains the core data types used by the CLR (and thus by languages built on the CLR). These types include several varieties of integers, a string type, and many more.
- **System.Web:** This namespace contains types useful for creating Web applications, and like many namespaces, it has subordinate namespaces. Developers can use the types in System.Web.UI to build ASP.NET browser applications, for example, while those in System. Web.Services are used to build ASP.NET Web Services applications.
- **System.Data:** The types in this namespace comprise ADO.NET. For example, the Connection class is used to establish connections to a database management system (DBMS), while an instance of the DataSet class can be used to cache and examine the results of a query issued against that DBMS.
- **System.Windows.Forms:** The types in this namespace make up Windows Forms, and they're used to build Windows GUIs. Rather than relying on language-specific mechanisms, such as the older Microsoft Foundation Classes (MFC) in C++, .NET Framework applications written in any programming language use this common set of types to build graphical interfaces for Windows.
- **System.EnterpriseServices:** The types in this namespace provide services required for some kinds of enterprise applications. Implemented by COM+ in the pre-NET world, these services include distributed transactions, object instance lifetime management, and more. The most important type in this namespace, one from which

classes must inherit to use Enterprise Services, is the ServicedComponent class.

- **System.XML:** Types in this namespace provide support for creating and working with XML-defined data. The XmlDocument class, for instance, allows accessing an XML document using the Document Object Model (DOM). This namespace also includes support for technologies such as the XML Schema definition language (XSD) and XPath.

Learning the .NET Framework class library takes time

Many more namespaces are defined, providing support for file access, serializing an object's state, remote access to objects, and much more. In fact, the biggest task facing developers who wish to build on the .NET Framework is learning to use the many services that the library provides. There's no requirement to learn everything, however, so a developer is free to focus on only those things relevant to his or her world. Still, some parts will be relevant to almost everybody, and so the next sections provide a quick overview of some of this large library's most important aspects.

Building Web Applications: ASP.NET

ASP.NET applications rely on .aspx pages

Implemented in the System.Web namespace, ASP.NET is an important piece of the .NET Framework. The successor to the very popular Active Server Pages (ASP) technology, ASP.NET applications are built from one or more pages. Each page contains HTML and/or executable code, and typically has the extension *.aspx.* As Figure 1-5 shows, a request from a browser made via HTTP causes a page to be loaded and executed. Any output the page creates is then returned to the browser that made the request.

ASP.NET includes a number of things to help developers create Web applications

Building effective Web applications requires more than just the ability to combine code with HTML. Accordingly, ASP.NET provides a range of support, including the following:

- Web controls, allowing a developer to create a browser GUI in a familiar way. By dragging and dropping

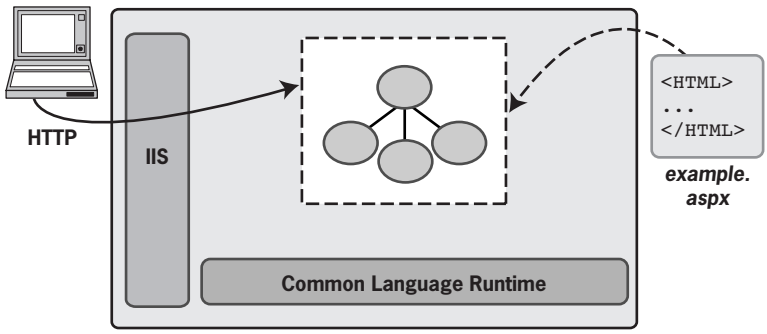


Figure 1-5 ASP.NET allows developers to create browser-accessible applications.

standard ASP.NET controls for buttons and other interface elements onto a form, it's possible to build GUIs for Web applications in much the same way as for local Windows applications.

- Mechanisms for managing an application's state information.
- Built-in support for maintaining information about an application's users, sometimes called *membership* information.
- Support for *data binding,* which allows easier access to information stored in a DBMS or some other data source.

Given the popularity of Web applications, ASP.NET probably impacts more developers than any other part of the .NET Framework class library. Chapter 5 provides more detail on this key component of the .NET Framework.

Accessing Data: ADO.NET

ADO.NET lets applications access stored data

ADO.NET lets applications work with stored data. As Figure 1-6 shows, access to a DBMS relies on a .NET Framework data provider, written as managed code. Providers that allow access to SQL Server, Oracle, and other DBMS are included with the

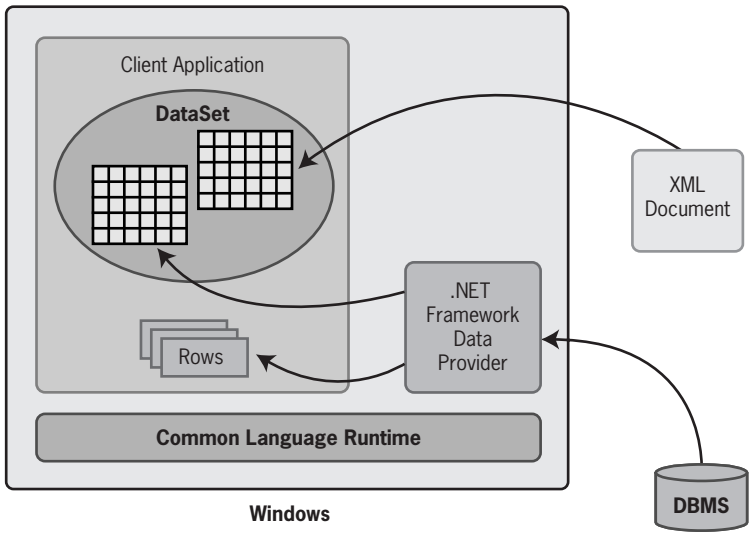


Figure 1-6 ADO.NET allows .NET Framework applications to access data stored in DBMS and XML documents.

.NET Framework. They allow a client application to issue commands against the DBMS and examine any results those commands return. The result of a Structured Query Language (SQL) query, for example, can be examined in two ways. Applications that need only read the result a row at a time can do this by using a DataReader object to march through the result one record at a time. Applications that need to do more complex things with a query result, such as send it to a browser, update information, or store that information on disk, can instead have the query's result packaged inside a DataSet object.

An ADO.NET DataSet acts as an in-memory cache for data

As Figure 1-6 illustrates, a DataSet can contain one or more tables. Each table can hold the result of a different query, so a single DataSet might potentially contain the results of two or more queries, perhaps from different DBMS. In effect, a DataSet acts as an in-memory cache for data. As the figure shows, however, DataSets can hold more than just the result of a SQL query. It's also possible to read an XML document directly into a table in a DataSet without relying on a .NET Framework data

Running the .NET Framework

The .NET Framework is meant to be the foundation for most Windows applications going forward. To make this possible, the Framework runs on many versions of Windows, including Windows 2000, Windows XP, Windows Server 2003, and Windows Vista. It's also available for the 64-bit versions of Windows XP, Windows Server 2003, and Windows Vista. The Framework doesn't run on older systems, however, such as Windows 95 or Windows NT. Given that it was released many years after these versions of Windows, this shouldn't be surprising.

The .NET Framework also supports an option called *side-by-side* execution. This allows simultaneous execution of not just multiple versions of the same application, but also multiple versions of the .NET Framework itself. For example, a single machine might have both version 1.1 and version 2.0 of the Framework installed, with each used to run applications written specifically for it. This lets organizations move forward with new versions of the .NET Framework without touching existing applications that run on earlier releases.

provider. Data defined using XML has also become much more important in the last few years, so ADO.NET allows accessing it directly. While not all .NET Framework applications will rely on ADO.NET for data access, a large percentage surely will. ADO.NET is described in more detail in Chapter 6.

Building Distributed Applications

Creating software that communicates with other software is a standard part of modern application development. Yet different applications have different communication requirements. To meet these diverse needs, the .NET Framework class library includes three distinct technologies for creating distributed applications. Figure 1-7 illustrates these choices.

ASP.NET Web Services allow communication via SOAP

ASP.NET Web Services, mostly defined in System.Web.Services, allows applications to communicate using Web services. Since it's part of ASP.NET, this technology lets developers use a similar

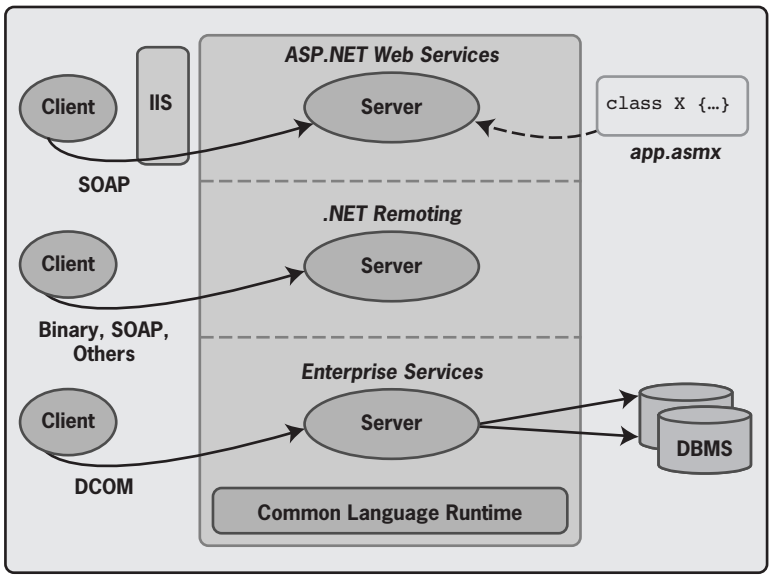


Figure 1-7 Distributed applications can use ASP.NET Web Services, .NET Remoting, or Enterprise Services.

model for creating distributed software. As Figure 1-7 shows, applications that expose methods as Web services can be built from files with the extension .asmx, each of which contains only code. Clients make requests using the standard Web services protocol SOAP[2], and the correct page is loaded and executed. Because this technology is part of ASP.NET, requests and replies also go through Internet Information Services (IIS), the standard Web server for Windows.

(chapter continues...)

2. "SOAP" was originally an acronym for "Simple Object Access Protocol." Today, the standards group that owns this technology has decided that SOAP no longer stands for anything—it's just a name.