CHAPTER 1

.NET Architecture

.NET security is not an island of technology, but a slice of a larger entity called the .NET Framework. Basic understanding of the .NET Framework is required before attempting .NET security programming. This chapter presents the basic concepts of the .NET Framework architecture and programming. This is an overview and is not intended to replace the independent study required for a mastery of this subject. (For a comprehensive discussion on the .NET Framework from a developer's perspective, I recommend *.NET Framework Essentials* by Thun L. Thai and Hoang Q. Lam, O'Reilly & Associates, February 2002.)

Microsoft .NET is not just a different spin on the Win32 operating model. Furthermore, despite reports to the contrary, it is not Java in wolf's clothing. You will never understand or adequately explain .NET simply by comparing it to existing products. .NET is new. As such, .NET introduces a fresh operating modality and perspective on computing software and devices.

Are there similarities to Java? Are there similarities to Win32? Yes, but there are many more differences. Successfully programming in .NET requires embracing this new technology as new and fully understanding the many things that make .NET unique. When object-oriented languages were introduced, developers faced a similar challenge and, unfortunately, mindset. Many programmers quickly learned the syntax and ported their C application to C++ or SmallTalk. However, without the requisite understanding of object-oriented programming, these new applications were procedural programs draped in the syntax of an object-oriented language. Some developers invested the time

to learn object-oriented programming—not just the syntax, but the philosophy and intent. Their resulting applications were true object-oriented programs that provided all the benefits envisioned for the new programming modality. Similarly, understanding the philosophy and architecture of .NET is essential for creating applications that offer new solutions. Some industry analysts assert that Microsoft has gambled the company on .NET. I would not agree. .NET does represent a massive investment. However, Microsoft is a diversified and multibillion dollar company with many products and a sizable market share in many segments of the software industry. In addition, Microsoft is no longer simply a software company, having expanded into many markets outside their traditional stronghold.

But recognizing that Microsoft is not teetering on a precipice named .NET does not diminish the importance of .NET. .NET does represent a new philosophy in product development. From .NET will emerge an entirely new family of products that will drive Microsoft sales into the stratosphere over the next 5 to 10 years. If the .NET initiative fails, or more likely is adopted slowly, Microsoft will recover and continue, although maybe with a little less luster. Importantly, .NET allows Microsoft to escape the Windows conundrum. Although Windows has been enormously successful, it is still a box. .NET helps Microsoft emerge from that box and develop applications for a universal audience. This new opportunity will fuel growth not just for Microsoft, but for software developers everywhere.

I attended the formal launch of Microsoft .NET at the Professional Developers Conference in Orlando, Florida several years ago. William Gates III (aka Bill) was the keynote speaker. Part of his speech included an entertaining video. The video portrayed .NET as a new standard that will allow software to run anywhere, at anytime, on any platform, and on devices large and small.

Anywhere. This has reported many times, but it is worth repeating: "Microsoft was late to realize the importance and then embrace the Internet." Recently, Microsoft has been making up for that late start. .NET marks the next major step in that journey. The Internet is not an adjunct of .NET, but is interwoven seamlessly into the product. The Internet was planned, integrated, and implemented into .NET—including the embracing of open standards such as XML and HTTP. Essentially, any platform that offers a browser that understands XML or HTML is a potential .NET client.

Anytime. The Internet is open 7 days per week and 24 hours per day. The Internet never closes. Since .NET leverages the Internet, .NET applications such as a Web service are fully accessible at anytime.

Any platform. .NET is a multilanguage and multiplatform operating environment. Compare this to Java, which is single-language and multiplatform. .NET offers C#, Visual Basic .NET, and many more .NET-compliant

languages. Programming in .NET does not require learning an entirely new language. To program to the Java Virtual Machine (JVM) requires learning the Java language. For many, this is a substantial drawback. The common language runtime is the common runtime of all .NET languages. In addition, Microsoft publishes the Common Language Infrastructure (CLI) document, which is a set of guidelines for creating a .NET common language runtime for any platform, such as Linux. To view one such initiative, visit www.go-mono.com. In the future, developers can create .NET applications in Windows and run them in Linux, Unix, Macintosh, or any platform that offers a common language runtime.

Devices large and small. NET marks Microsoft's first extensive support of open standards, even if it is rather tepid. Microsoft adopts HTTP, SMTP, SOAP, XML, and many more standards. This means that any device that supports these standards can actively participate in a .NET conversation. This will liberate personal digital assistants (PDAs), hand-held, and embedded devices. These devices lack the girth to run powerful applications, such as full-blown Microsoft Office. Using open standards, these devices can tap the power of a back-end server and run virtually any program. The embedded chip in your refrigerator could access Microsoft Word remotely, compose a grocery list, and print it to a networked printer. Refrigerators with word-processing capabilities—way cool!

.NET Is Web Enabled

Microsoft .NET is Web empowered. Developers can use ASP.NET, XML Web services, and ADO.NET to easily create feature-rich Web applications. This represents the front, middle, and bottom tier of an *n*-tiered enterprise application. Despite this, do not believe the rhetoric stating that Microsoft has abandoned client-side applications—some applications will never be well suited for server-side operations. Windows Forms, a new forms generation engine, and other additions in the .NET Framework make development of traditional Windows applications more intuitive, while adding additional features.

.NET Components

.NET introduces a new component model that is largely implicit. The messiness of COM (Component Object Model) is removed. In .NET, developers use standard language syntax to create, publish, and export components. There is nothing else to learn. .NET addresses many of the shortfalls of COM, including susceptibility to DLL Hell, language incompatibilities, reference counting, and more.

Coding COM at the API level is exacting. Interfaces such as IUnknown, IDispatch, IConnectionPoint, and system functions such as CoGetClassObject and CoFreeUnusedLibraries represent a massive learning curve. MFC (Microsoft Foundation Classes), ATL (Active Template Library), and Visual Basic provided some relief, but offered different solutions for creating COM objects for different languages.

How do you create a component in .NET? In the server application, you define a public class, using the syntax of the preferred .NET language. In the client, you import a reference to the component application and then create an instance of the component class. That is it. You can then use the component. The arcane syntax of COM is gone.

The developer controls the lifetime of a COM object. AddRef and Release live in infamy. They are methods of the IUnknown interface and control the persistence of the COM object. If implemented incorrectly, a COM object could be released prematurely or, conversely, never be released and be the source of memory leaks. The .NET memory manager, appropriately named the Garbage Collector, is responsible for managing component lifetimes for the application—no more AddRef and Release.

COM was largely a Windows standard, and building bridges to components in other platforms was difficult. COM was still a worthwhile endeavor and did indeed advance the concept of component development. However, now the torch has been handed to .NET. Using open standards, .NET components are potentially accessible to everyone at anytime.

Component versioning was a considerable problem with COM and contributed to DLL Hell. Another contributor was COM's reliance on the Windows Registry. Let us assume that two versions of the same in-process server (COM DLL) are installed on the same computer, and the newer version is installed first. The older version will override the Registry settings of the newer component, and clients will be redirected to the incorrect version. Clients using newer services will immediately break. .NET does not rely on the Registry for component registration, which diminishes the possibility of DLL Hell.

I have been teaching .NET to developers for some time and usually start class with a question for my students: Can someone describe the benefits of .NET for end users? Students quickly mention the common language runtime, ASP.NET, XML integration, Garbage Collection, and so on. They are all great things, but they benefit programmers rather than end users. Why would clients give a whit about .NET? The benefit of .NET to users is the new generation of software that .NET introduces: software that runs anywhere, at anytime, on any platform, and from devices large and small.

.NET Framework Architecture

.NET is tiered, modular, and hierarchal. Each tier of the .NET Framework is a layer of abstraction. .NET languages are the top tier and the most abstracted level. The common language runtime is the bottom tier, the least abstracted, and closest to the native environment. This is important since the common language runtime works closely with the operating environment to manage .NET applications. The .NET Framework is partitioned into modules, each with its own distinct responsibility. Finally, since higher tiers request services only from the lower tiers, .NET is hierarchal. The architectural layout of the .NET Framework is illustrated in Figure 1.1.

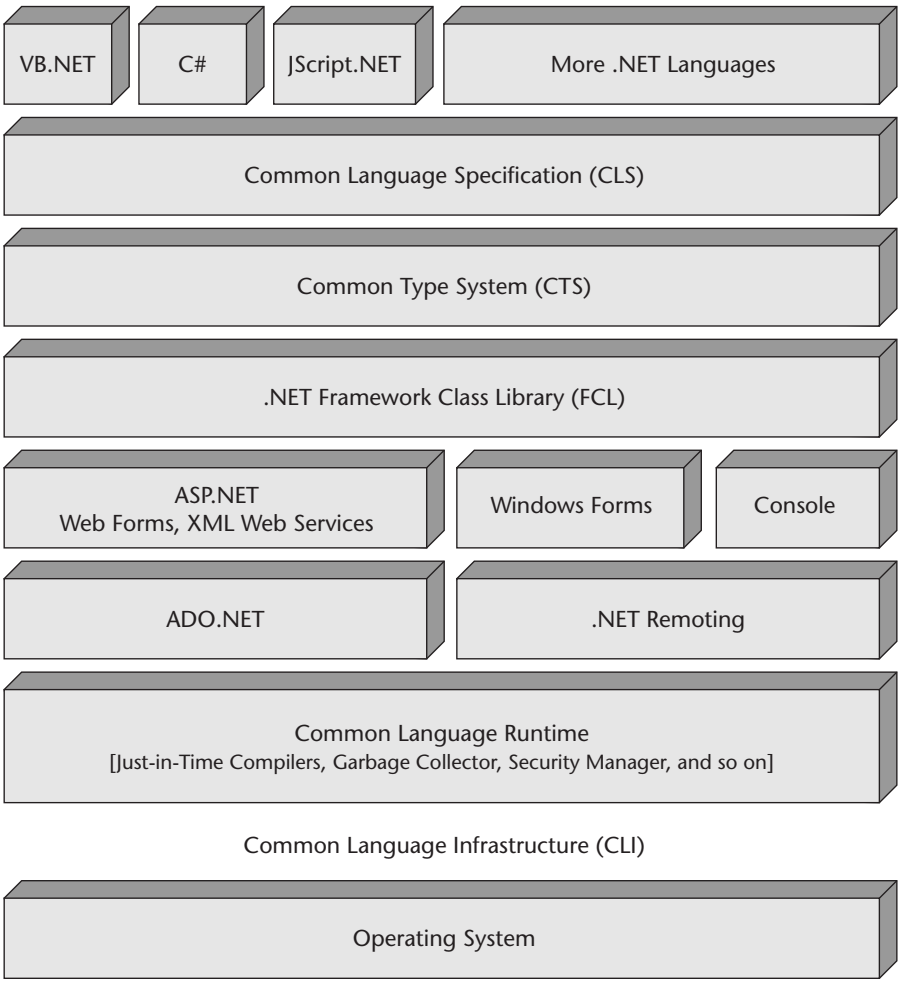

Figure 1.1 An overview of the .NET architecture.

The .NET Framework is a managed environment. The common language runtime monitors the execution of .NET applications and provides essential services. It manages memory, handles exceptions, ensures that applications are well-behaved, and much more.

Language interoperability is one goal of .NET. .NET languages share a common runtime (the common language runtime, a common class library), the Framework Class Library (FCL), a common component model, and common types. In .NET, the programming language is a lifestyle choice. Except for subtle differences, C#, VB.NET, or JScript.NET offer a similar experience.

.NET abstracts lower-level services, while retaining most of their flexibility. This is important to C-based programmers, who shudder at the limitations presented in Visual Basic 6 and earlier.

Let us examine each tier of the .NET Framework as it relates to a managed environment, language interoperability, and abstraction of lower-level services.

Managed Languages and Common Language Specification

.NET supports managed and unmanaged programming languages. Applications created from managed languages, such as C# and VB.NET, execute under the management of a common runtime, called the common language runtime.

There are several differences between a compiled managed application and an unmanaged program.

- Managed applications compile to Microsoft Intermediate Language (MSIL) and metadata. MSIL is a low-level language that all managed languages compile to instead of native binary. Using just-in-time compilation, at code execution, MSIL is converted into binary optimized both to the environment and the hardware. Since all managed languages ultimately become MSIL, there is a high degree of language interoperability in .NET.
- Metadata is data that describes data. In a managed application, also called an assembly, metadata formally defines the types employed by the program.
- Wave a fond goodbye to the Registry. Managed applications are sweeping away the Registry, Interface Definition Language (IDL) files, and type libraries with a single concept called metadata. Metadata and the related manifest describe the overall assembly and the specific types of an assembly.

- Managed applications have limited exposure to the unmanaged environment. This might be frustrating to many programmers, particularly experienced C gurus. However, .NET has considerable flexibility. For those determined to use unmanaged code, there are interoperability services.

NOTE

In .NET, a managed application is called an assembly. An *assembly* adheres to the traditional Portable Executable (PE) format but contains additional headers and sections specific to .NET. MSIL and metadata are the most important new additions to the .NET PE. When the .NET Framework is installed, a new program loader recognizes and interprets the .NET PE format. In future Windows operating systems, the first being .NET Server, the .NET loader is automatically provided.

What is a managed language? If someone wants to create Forth.NET, are there established guidelines? Common Language Specification (CLS) is a set of specifications or guidelines defining a .NET language. Shared specifications promote language interoperability. For example, CLS defines the common types of managed languages, which is a subset of the Common Type System (CTS). This removes the issue of marshaling, a major impediment when working between two languages.

Common Type System

The Common Type System (CTS) is a catalog of .NET types—System.Int32, System.Decimal, System.Boolean, and so on. Developers are not required to use these types directly. These types are the underlying objects of the specific data types provided in each managed language. The following is the code for declaring an integer in C# and Visual Basic .NET. Either syntax maps to a System.Int32 object.

```
// C# integer
int nVar=0;
' VB.NET
dim nVar as integer=0
```

Preferably, you should use the syntax of the language and not the underlying object type, leaving .NET the flexibility to select the most appropriate type and size for the operating environment.

The common type system is a pyramid with System.Object at the apex. .NET types are separated into value and reference types. Value types, which are mainly primitive types, inherit from System.ValueType and then System.Object. Reference types—anything not a value type—are derived from System.Object,

either directly or indirectly. Value types are short-term objects and are allocated on the stack. Reference types are essentially pointers and allocated on the managed heap. The lifetime of reference types is controlled by the Garbage Collector.

NOTE

There are many more differences between a value type and reference types, which is a subject beyond the context of this book.

Value types can be converted to reference types, and vice versa, through processes called boxing and unboxing, respectively. Boxing is helpful when a developer needs to change the memory model of an object.

The contributions of CTS extend well beyond the definitions of common data types. CTS helps with type safeness, enhances language interoperability, aids in segregating application domains, and more. Type verification occurs during just-in-time compilation, ensures that MSIL safely accesses memory, and confirms that there is no attempt to access memory that is not formerly defined in metadata. If so, the code is treated as a rogue application. CTS provides a shared type substratum for .NET, enhancing language interoperability. Finally, .NET introduces lightweight processes called application domains. Application domains are processes within a process. Application domains are more scalable and less expensive then traditional Win32 processes. .NET must police application domains and guarantee that they are good neighbors. Code verification, type safeness, and CTS play a role in guaranteeing that application domains are safe.

.NET Framework Class Library

The .NET Framework Class Library (FCL) is a set of managed classes that provide access to system services. File input/output, sockets, database access, remoting, and XML are just some of the services available in the FCL. Importantly, all the .NET languages rely on the same managed classes for the same services. This is one of the reasons that, once you have learned any .NET language, you have learned 40 percent of every other managed language. The same classes, methods, parameters, and types are used for system services regardless of the language. This is one of the most important contributions of FCL.

Look at the following code that writes to and then reads from a file. Here is the C# version of the program.

```
// C# Program
static public void Main()
{
     StreamWriter sw=new StreamWriter("date.txt", true);
     DateTime dt=DateTime.Now;
     string datestring=dt.ToShortDateString()+" "+
          dt.ToShortTimeString();
     sw.WriteLine(datestring);
     sw.Close();
     StreamReader sr=new StreamReader("date.txt");
     string filetext=sr.ReadToEnd();
     sr.Close();
     Console.WriteLine(filetext);
}
```

Next is the VB.NET version of the program.

```
' VB.NET
shared public sub Main()
     dim sw as StreamWriter=new StreamWriter("date.txt", true)
     dim dt as DateTime=DateTime.Now
     dim datestring as string=dt.ToShortDateString()+" " _
          +dt.ToShortTimeString()
     sw.WriteLine(datestring)
     sw.Close()
     dim sr as StreamReader=new StreamReader("date.txt")
     dim filetext as string=sr.ReadToEnd()
     sr.Close()
     Console.WriteLine(filetext)
end sub
```

Both versions of the program are nearly identical. The primary difference is that C# uses semicolons at the end of statements, while VB.NET does not. The syntax and use of StreamReader, StreamWriter, and the Console class are identical: same methods, identical parameters, and consistent results.

FCL includes some 600 managed classes. A flat hierarchy consisting of hundreds of classes would be difficult to navigate. Microsoft partitioned the managed classes of FCL into separate namespaces based on functionality. For example, classes pertaining to local input/output can be found in the namespace System.IO. To further refine the hierarchy, FCL namespaces are often nested; the

tiers of namespaces are delimited with dots. System.Runtime.InteropServices, System.Security.Permissions, and System.Windows.Forms are examples of nested namespaces. The root namespace is System, which provides classes for console input/output, management of application domains, delegates, garbage collection, and more.

Prefixing calls with the namespace can get quite cumbersome. You can avoid needless typing with the *using* statement, and the namespace is implicit. If two namespaces contain identically named classes, an ambiguity may arise from the *using* statement. Workarounds for class name ambiguity are provided by defining unique names with the *using* directive. Here is a simple program written without the *using* statement.

```
public class Starter
{
     static void Main()
     {
          System.Windows.Forms.MessageBox.Show("Hello, world!");
          System.Console.WriteLine("Hello, world");
     }
}
```

This the same program with the *using* statement. Which is simpler? Undeniably, the next program is simpler and more readable.

```
using System;
using System.Windows.Forms;
public class Starter
{
     static void Main()
     {
          MessageBox.Show("Hello, world!");
          Console.WriteLine("Hello, world");
     }
}
```

It is hard to avoid the FCL and write a meaningful .NET application. Developers should fight the tendency or inclination to jump to unmanaged code for services provided in .NET. It may appear simpler because you have used that unmanaged API a hundred times. However, your program then becomes less

portable, and security issues may arise later. When in Rome, do as the Romans do. When in .NET, use managed code.

ASP.NET

ASP.NET is used to create dynamic Web applications and is the successor to ASP. While IIS 5 and 6 support side-by-side execution of ASP and ASP.NET, ASP.NET is not merely an upgrade of ASP, as evidenced by the lack of upward compatibility. The view state, configuration files, validation controls, and a total reconstruction of the ASP architecture is the short list of numerous changes. The theme of these changes is scalability, extensibility, and improving the programmer experience.

Microsoft emphasizes scalability in ASP.NET. Free threads boost responsiveness and prevent internal bottlenecks. ASP.NET uses ADO.NET, server-side controls, and other techniques to promote a highly distributed and scalable model. Also, ASP.NET hosts Web applications in application domains within the worker process (aspnet_wp.exe) to heighten performance and lower overhead. Finally, ASP.NET uses compiled pages instead of interpreted pages to improve performance.

Speaking of scripting languages, many developers did not learn ASP because of the necessity of learning yet another language. In addition, some C-based programmers have an aversion to scripting languages on principle, such as VBScript. ASP.NET is coded in the managed language of choice: C#, VB.NET, SmallTalk .NET, or whatever. The complication of learning a new language solely for Web development has been removed. There is an additional, probably unintentional, but positive side effect. Converting a client-side application written in a managed language to a Web-based program is considerably easier since the language now remains the same. This removes the final excuse for not moving all your applications to the Web.

NOTE

Of course, this excludes device driver developers. You can continue to happily code to your platform.

Web Forms (see Figure 1.2), is the forms generator for Web applications in ASP.NET and replaces Visual Interdev. Web Forms closely resembles Windows Forms or the Visual Basic 6 forms engine, one more consideration that helps developers move between client and Web programming. A variety of server- and client-side controls are supported. Web Forms controls are server-side controls and typically more complex than HTML controls. The calendar control typifies a Web Form control that maps to multiple tags. HTML controls by default are client-side controls and map to a single tag. Web controls are instances of managed classes that write HTML tags. Developers create custom controls by having them inherit from the Control class and coding the Control.Render method to output desired tags.

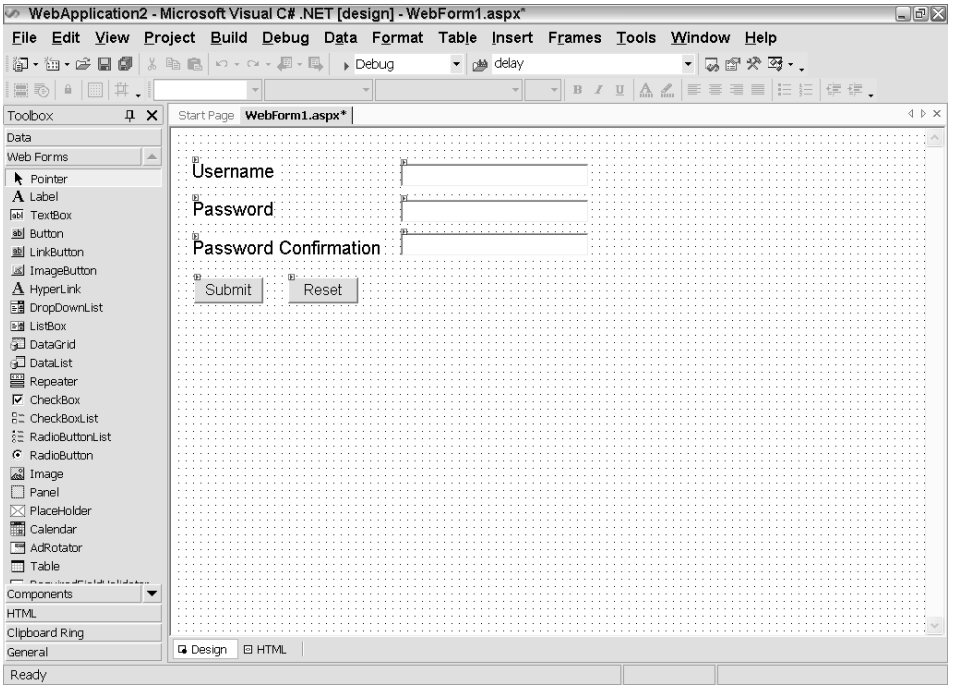

Figure 1.2 This Web form submits a username and password. This form is built from Web Forms controls.

Developers can view and edit the generated HTML by switching to the HTML pane, shown in Figure 1.3. Actually, developers can opt out of the Web forms and build controls directly into the HTML pane or even Notepad.

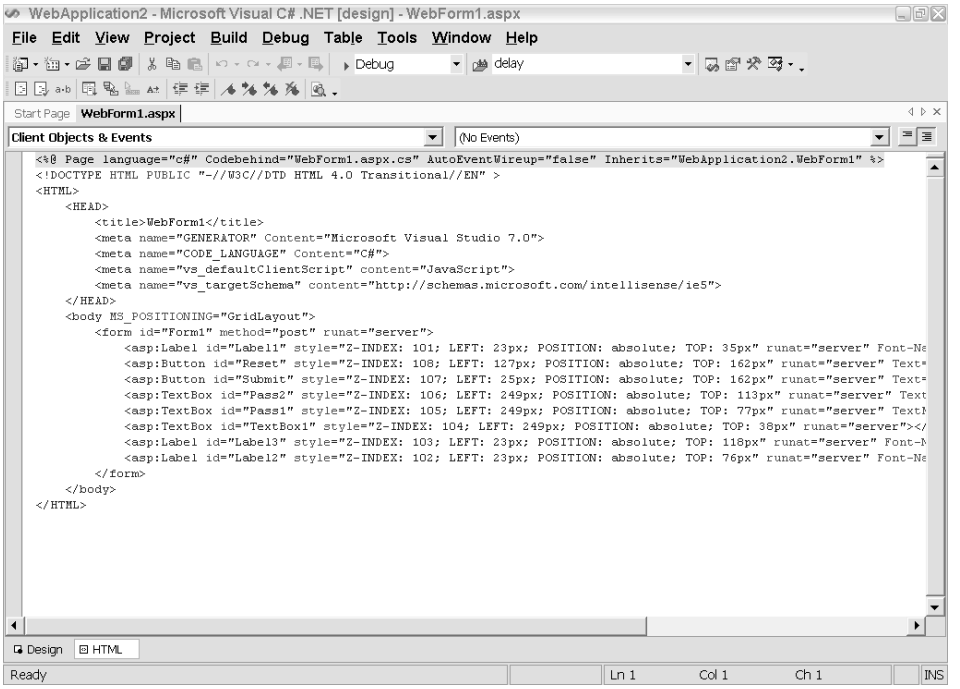

Figure 1.3 The HTML view of the username and password form.

XML Web Services

Web services are the basis of the programmable Web and distributed applications that transcend hardware and operating environments. Web services are not unique to Microsoft. Microsoft, IBM, Sun, and other vendors are promoting Web services as integral components of their recent initiatives. Web services promote remote function calls over the Internet. The promotion and hype surrounding Web services has been deafening.

A Web service exposes functionality to the entire world—any device at any time. Not coincidentally, this is the very definition of .NET and explains the importance of a Web server to .NET. Anyone with Web-enabled software, such as a browser, that understands HTML and HTTP can access a Web service. Any device, large or small, that is Web enabled can access a Web service. This revolutionizes embedded devices, bringing an array of services to these devices that were not previously practical. The Internet never closes, which means that Web services are available 24/7. The explosion of computers and PDAs that are continuously wired to the Internet creates a burgeoning audience for Web services.

XML Web services are part of ASP.NET and leverage open standards, such as HTTP and XML, to publish public functions to the Internet. In addition, creating Web services in .NET is remarkably easy. Web pages publish presentation data with limited functionality over the Internet. Open standards, namely HTML and HTTP, are the backbone of Web pages and deliver them to almost anyone with a browser. Web services expose functionality over the Internet using open standards—the combination of HTTP, XML, SOAP, and the Web Service Description Language (WSDL), which are the underpinning of this exciting technology. SOAP is the preferred protocol for XML Web services.

Visual Studio .NET removes the challenge of creating a Web service. Developers can create Web services with limited or no knowledge of SOAP, XML, or WSDL. The first step is to create an ASP.NET Web service project. The new project is a starter kit for a Web service application. A Web service class, sample Web method, web.config file, global.asx file, and the remaining plumbing of a Web service are provided. See Figure 1.4. The developer is left with only one task—writing the functions to be exposed as Web methods.

Consuming a Web service is equally easy. Start by creating a proxy from the WSDL of the Web service. Create the proxy in Visual Studio .NET by adding a Web reference or externally using the WSDL.exe tool. After creating the proxy, compile it into the Web service client or into a separate assembly, and then bind it to the client. In the client code, create an instance of the proxy and call the Web service methods as local functions. The proxy connects to the Web service, uses SOAP to invoke a remote method, and then return any values. The parameters, return value, and other Web service data are transported in XML envelopes.

```
{
    /// <summary>
    /// Summary description for Service1.
    /// </summary>
    public class Service1 : System.Web.Services.WebService
    {
        public Service1()
        {
            //CODEGEN: This call is required by the ASP.NET Web Services Designer
            InitializeComponent();
        }

        Component Designer generated code

        // WEB SERVICE EXAMPLE
        // The HelloWorld() example service returns the string Hello World
        // To build, uncomment the following lines then save and build the project
        // To test this web service, press F5

//      [WebMethod]
//      public string HelloWorld()
//      {
//          return "Hello World";
//      }
    }
}
```

Figure 1.4 The Web service class and method are generated for the ASP.NET Web service project.

Windows Forms and Console Applications

Windows Forms is the form generator for client-side applications and is similar to the forms engine of Visual Basic 6. Visual Basic programmers using VB.NET will be familiar with the look and feel of Windows Forms, but this similarity is largely cosmetic and there are substantial differences in the implementation. Windows Forms is new to Visual C++ programmers, who previously had to code every aspect of a program's graphical user interface. This excludes the dialog editor of Visual C++, which is a limited forms engine.

Microsoft Foundation Classes removed some of the drudgery, but it was far from a visual tool. C-based developers can now focus more on the application and less on the mechanics of creating edit boxes, coding buttons, managing a status bar, and attaching a menu to a window. Windows Forms is primarily a code generator, generating managed classes for forms, buttons, text boxes, menus, and other graphical user interface elements.

NOTE

C++ developers do not have access to Windows Forms as a forms generator. They are restricted to coding the graphical user interface from managed classes found in the Windows.Forms namespace. There is no visual assistance. C-based programmers must use C# for full access to Windows Forms visual tools.

Console applications have been available to C-based, but not to Visual Basic, programmers. In .NET, console applications are available to all managed languages. Console applications are useful for logging, instrumentation, and other text-based activities.

ADO.NET

ADO.NET is an exceptional and a worthy successor to ADO. ADO.NET accentuates disconnected data manipulation, is highly scalable, integrates open standards, and is perfected for Web application development. ADO.NET offers managed providers for Microsoft SQL and OLE DB databases and is a set of managed classes in the System.Data namespace. The System.Data.SqlClient namespace contains classes related to Microsoft SQL, while System.Data.OleDb encompasses classes pertaining to OLE DB providers.

The differences between ADO.NET and ADO are significant. In ADO, everything revolves around record sets. ADO.NET does not support record sets, which underscores that there are fundamental differences between ADO.NET and ADO. ADO.NET is an entirely different model. Record sets, client-side cursors, and server-side cursors are not supported in ADO.NET. Instead, these constructs are replaced by data readers on the server side and data sets for client-side disconnected management of data. Data readers are server side, and support read-only and forward-only access of the database. Data sets are client side, bidirectional, and modifiable.

ADO.NET is more scalable then its predecessor and embraces the disconnected model. Data sets are more flexible then records sets, supporting the client-side transfer of multiple tables and relations. The ability to transfer multiple tables of a database as a unit was not easily supported in ADO. With more relevant data on the client side, the application can remain disconnected longer, which improves scalability and enhances customer satisfaction.

XML is ubiquitous in the .NET Framework—no more so than in ADO.NET, where you bump into XML everywhere. Data sets particularly rely heavily on XML as the standard for transmitting data from the server to the client. In data sets, building relations between tables from different databases is trivial, since regardless of the origin, everything is ultimately XML. There is no concept of

an SQL or OleDb dataset. When the data reaches the data set, it is fully homogenized. This provides ADO.NET with unprecedented flexibility. Data sets are initialized using data adapters.

Data adapters play a critical role in ADO.NET and are the bridge between the data source and the client-side data set. The data adapter is the connection between the data source and the disconnected database. First, a data adapter fills the data set with content from the data source. Second, using command builder objects, data adapters update the data source with changes incurred in the data set. Data adapters are the glue that makes the disconnected model work.

An ADO.NET data set is an ensemble of collections, beginning with the tables collection. Iterate the tables collection to find the individual tables stored in the data set. Each table has a rows collection and a data column collection. The rows collection contains the records of the table, while the data column collection is the schema of the table. Each row has a collection of values, normally called fields. Iterate the fields to extract the individual values of each record. Therefore, browsing a database is series of iterations, which is made easier with the *foreach* statement.

The following code creates a data set from the authors table and then iterates each record, outputting the author names from within a *foreach* loop.

```
using System;
using System.Data;
using System.Data.SqlClient;
public class Starter
{
     public static void Main()
     {
          string conntext= "Initial Catalog=pubs;"
               +"Data Source=localhost;"
               +"Integrated Security=SSPI;";
          SqlConnection conn= new SqlConnection(conntext);
          SqlCommand command=new SqlCommand("SELECT au_lname,
               au_fname FROM authors", conn);
          SqlDataAdapter da= new SqlDataAdapter(command);
          DataSet ds=new DataSet();
          da.Fill(ds);
          DataTable dt=ds.Tables[0];
          foreach(DataRow row in dt.Rows)
          {
               Console.WriteLine(row["au_lname"]+
                    ", "+row["au_fname"]);
          }
          conn.Close();
     }
}
```

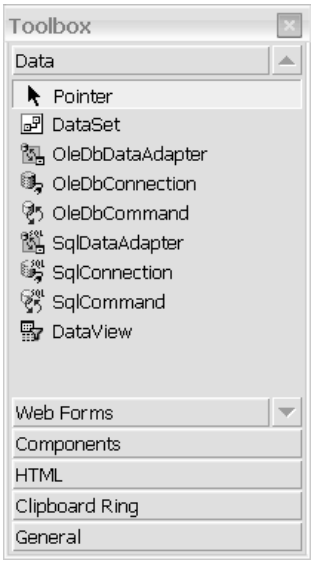

Figure 1.5 This is the data toolbox from a Web application project in Visual Studio .NET.

The Data toolbar of Visual Studio .NET shown in Figure 1.5 provides a full complement of controls for rapid application development using ADO.NET. ADO.NET code can be complicated and long. The data tools generate the code for you and can be a welcome time saver.

.NET Remoting

.NET Remoting is a second option for remoting objects in the .NET Framework. The first mentioned was XML Web services. .NET Remoting is similar to Web services conceptually. However, with .NET Remoting the developer chooses the transmission protocol, data protocol, data port, and other aspects of the remoting architecture necessary to open a channel for client-server communication. In essence, a developer is setting the specifications of the remoting infrastructure. In this way, .NET Remoting offers unlimited possibilities. Like Web services, .NET Remoting can leverage open standards, such as XML, HTTP, and SMTP. .NET Remoting is fully extensible. Custom or and proprietary standards can also be plugged in.

The namespace for .NET Remoting is System.Runtime.Remoting. Remoted objects can be copied or used as a reference (marshal-by-reference). If copied, the object is duplicated in the address area of the client and is then accessible as a local object. Alternately, remoted objects can be accessed by reference via a proxy that connects to the object on the server machine. The indirection of a proxy affects performance. However, not all remoted objects are good

candidates for copying. Some are either too large, have software dependencies on the server side, or require the server machine.

For server-side or referenced objects, .NET controls the activation of remoted objects as either Singleton or SingleCall objects. Singleton objects are created once, then shared among multiple clients. With SingleCall objects, each client receives a unique instance of the remoted object. RemotingConfiguration.RegisterWellKnownServiceType is the method that establishes the activation mode.

Like MSRPC (Microsoft RPC), the server application must be running for clients to connect. There is no Service Control Manager (SCM), as exists in COM, to bootstrap .NET Remoting server applications.

Example of .NET Remoting can be found in Chapter 5, "Role-Based Security."

Common Language Runtime

The common language runtime is the engine of .NET and the common runtime of all managed languages. In addition, as the final layer resting atop of the operating environment, the CLR provides the first level of abstraction. Since assemblies run within the context of the common language runtime, they are independent of the underlying operating environment and hardware. Assemblies or managed code are portable to any environment offering a .NET-compliant common language runtime, as defined by the Common Language Infrastructure (CLI).

Managed code is managed by the common language runtime. common language runtime manages security, code verification, type verification, exception handling, garbage collection, a common runtime, and other important elements of program execution. When an assembly is executed, mscoree.dll is loaded into the memory of the running process, and the imported entry point _CorExeMain is called. Mscoree contains the common language runtime, which then manages the executing application.

Of the many services offered by the common language runtime, we will focus on the two most important: code execution and memory management.

Just-in-Time Compilation

Assemblies contain MSIL, which is converted into native binary and executed at runtime, using a process aptly named Just-in-Time compilation, or jitting. An assembly is subjected to two compilations. First, managed code is compiled to create the actual assembly. Managed compilers, such as csc and vbc,

compile C# and VB.NET source code into an assembly that contains MSIL and metadata. Second, the assembly is compiled at load time, converting the MSIL into native binary that is optimized for the current platform and hardware. When an assembly is jitted, an in-memory cache of the binary is created and executed. Just-in-Time compilers are called Jitters. There is a Jitter for each supported hardware architecture.

Converting an entire program to native binary may be inefficient and compromise the performance of the application. The Jitter converts to binary only methods that are called; there's no need to convert methods that are never used to native code. Only the parts of the program used during the current execution are loaded into memory, which conserves resources. When the assembly is loaded, the class loader stubs the methods of each class. At this time, the stubs point to a Jitter routine that converts MSIL to native binary. Upon invocation of a stubbed method, the conversion routine is called that compiles MSIL into native binary, which is then cached in memory. The stub is then updated to point to the in-memory location of the native binary and the code is executed. Future calls to the same function skip the conversion routine and execute the native binary directly.

If you chart the runtime performance of a jitted assembly, you will see a series of small peaks and valleys. Based on usage patterns, the type of application, and the size of the application, this may not be the most efficient model for executing the assembly.

Consider a large application in which the user touches 80 percent of the functionality and where performance is a concern. A few peaks and valleys may not be noticeable, but a few hundred might be problematic. In this scenario, a different model will work better. In the alternate model, you compile and cache the native binary of the entire assembly. Running the assembly from the cache should improve performance. Instead of a blizzard of peaks and valleys, the new chart would show an acute spike at the beginning and then flat line for the remainder of the program. To adopt the later model, use the Ngen tool to generate the cached native image of the .NET assembly.

Garbage Collector

In .NET, the Garbage Collector manages the memory associated with new objects. Ceding memory management to the runtime is not a radical concept for Visual Basic programmers. However, this is indeed a radical concept for C-based programmers accustomed to managing their own memory. Despite the messiness, C-based programmers relish pointers. This approach will also be foreign to low-level COM programmers, who are accustomed to the IUnknown interface and to calling the AddRef and Release methods to manage

the lifetime of components. However, I believe there will be few COM programmers marching to protest the lost of AddRef and Release—these methods have never been a developer favorite. The Garbage Collector will hopefully eviscerate inadvertent memory leaks or problems caused by prematurely releasing an object. Memory management, regardless of the application, typical follows known patterns. There is absolutely no reason for every developer to implement a private memory management model. The Garbage Collector implements a common memory model that is applied to all managed code. The public interface of the Garbage Collector is the System.GC managed class.

Garbage collection manages the heap using a concept called generations, which is based on certain tenets:

- Small objects are generally short lived and frequently accessed.
- Larger objects are longer lived.
- Grouping like-sized objects together decreases heap fragmentation.
- Segmenting the heap offers efficiencies, such as defragmenting only a portion of the heap.

The managed heap is divided into generations. As new objects are created, they are placed in Generation 0. Eventually, memory reserved for Generation 0 will be exhausted. The generation is then compacted. If these steps do not free enough memory to satisfy the pending allocation request, the current generation is aged to Generation 1 and a new Generation 0 is started. Existing objects are now in Generation 1, and the pending object is allocated in the new Generation 0. The next time memory is depleted, these steps are repeated, now for Generation 0 and 1. If insufficient memory is recovered, the existing generations are aged and Generation 0 is reinitialized.

There are a maximum of three generations: Generation 0, 1, and 2. Sufficiently, large objects (larger than 20,000 bytes) are automatically copied to the large object heap, which is Generation 2. This model keeps short-lived objects group together in younger generations. Large objects are group in older generations. In addition, the smaller objects, which require more frequent aligning, can be compacted without having to recompact the entire heap.

In C#, a developer allocates memory for an object with the *new* statement. The Garbage Collector creates the object on the managed heap. However, there is no *free* or *delete* command to deterministically remove the object from memory. Cleanup of .NET objects is nondeterministic. When there are no outstanding references to the object, the Garbage Collector eventually removes the object from the managed heap. At that time, if the object has a destructor, the destructor is called and the object is afforded the opportunity to release unmanaged resources. This includes closing files, releasing tokens, and deconstructing socket connections.

Starting at the initialization, objects with destructors are weighted with non-trivial additional overhead. The management of objects with destructors at times seems counterintuitive and destined to draw the wrath of many developers. This is a perfect example of what happens when a good idea goes afoul. When an object with a destructor is instantiated, not only is the object allocated on the managed heap, but also a reference to the object is placed in the finalization queue. When there are no outstanding references to the object, the Garbage Collector does not immediately remove the object, as it would an object without a destructor. It spends the garbage collection cycle moving the object from the finalization queue to the freachable queue. The freachable queue contains references to objects pending finalization. There is a dedicated thread that services the freachable queue. This thread eventually wakes up and calls the destructors of objects found in the freachable queue. On the next cycle of garbage collection, these objects are finally deleted. Destructors should be omitted from all .NET objects unless required. Alternately, consider using an IDisposable.Dispose method, which can be called deterministically, without the overhead of a destructor.

This is a cursory explanation of garbage collection. For more details, consult MSDN, including two articles from Jeffrey Richter, "Garbage Collection: Automatic Memory Management in the Microsoft .NET Framework, Parts I and II."

Basic C# Application

C# is used throughout this book. Visual Basic .NET examples can be downloaded from the companion Web site for the book. Depending on the application, I switch between using NotePad and Visual Studio .NET as the development platform.

Chapter 1 of every programming book must have a Hello World application. This chapter offers the Hello World application twice.

```
using System;
using System.Windows.Forms;
public class Starter
{
     static void Main()
     {
          try
          {
               Console.WriteLine("Please enter your name : ");
               string name=Console.ReadLine();
               MessageBox.Show("Hello "+name);
               Console.WriteLine("Hello, {0}", name);
               Console.WriteLine("Please enter your age : ");
               string sAge=Console.ReadLine();
```

```
                XAge age=new XAge();
                age.IsYoung(int.Parse(sAge));
            }
            catch(Exception e)
            {
                MessageBox.Show(e.Message);
            }
        }
    }
    public class XAge
    {
        public void IsYoung(int _Age)
        {
            nAge=_Age;
            if(nAge<40)
                Console.WriteLine("You are very young!!");
            else
                Console.WriteLine("Well, you are young also.");
        }
        public XAge()
        {
            // Do nothing constructor
        }
        ~XAge()
        {
            // Do nothing destructor (costly)
        }
        private int nAge=0;
    }
```

This is a short, but representative C# program. Helpful notes on the hello program and general comments on C# follow.

- The using statements name the namespaces required in the program. System.Windows.Forms is required for the MessageBox.Show method.
- All methods must be contained in a class, including the entry point of Main. C# does not support global functions.
- The entry of a C# executable is Main. It must be a static method and there is only one entry point per program.
- The try block is guarded code, protecting against an exception. If an exception is raised, execution is transferred to the catch block and the error is displayed. System.Exception is the base exception and traps all exceptions.
- Finally, XAge is a public class that contains instance methods, such as a constructor, destructor, and IsYoung. There is also a single private data member called nAge. In Main, an instance of XAge is created and used. When the Main method ends, there is no outstanding references to the instance and it becomes a candidate for garbage collection.

- The program is compiled from the Visual Studio .Net command line using the csc compiler. As the result, hellorevised.exe, a console application is created through the following command: *csc hellorevised.cs*. Use the target option to compile to a library or Windows application. The reference option is used to bind to assemblies that contain external references required by the target through the following command: *csc /t:library /r:another.dll mylib.dll*.

With the example presented in this section, you should be able to understand most of the code presented in this book.

What's Next

This chapter offers the core concepts of the .NET Framework, while the next chapter covers the fundamental concepts of Win32 security. Win32 security is not replaced by .NET security. This is the first, but not the last, time this is mentioned in the book. Win32 and .NET security are partners, each providing a secured perimeter protecting securable resources.

The first gate usually entered is Win32 security. If Win32 security denies access to a securable resource or task, this will often preempt .NET security. Therefore, the importance of Win32 security is not diminished in .NET.

Win32 security consists of multiple systems working together to protect securable resources. Win32 security is end to end and starts at logon, when the user is authenticated and an access token is assigned. Later, processes inherit the access token and run in the security context of the related user. When processes attempt to access a secured resource on behalf of the user, authorization occurs and is coordinated by the security subsystem. The next chapter explains authentication, authorization, and impersonation in the Win32 environment.

The nuts and bolts of the security APIs and structures can be mind numbing and challenging even to an experienced Windows programmer. A discussion of security descriptors, security attributes, DACL, and ACEs, can be painful, but is nothing that a good explanation and some thorough examples will not rectify.

Finally, Windows XP Professional ships with an assortment of new security features and updates. The next chapter reviews what is new in Windows XP for security.