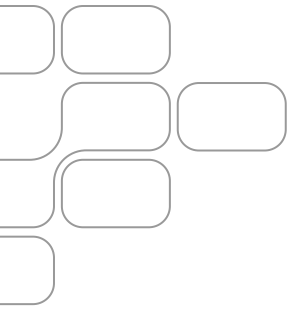
Microsoft® Visual Studio® .net™ Academic

Unleash the Power of .NET

An Architecture for Distributed Applications on the Internet: Overview of the Microsoft® .NET Platform



Introduction

One of today's most pressing computing challenges is application integration: taking different applications running on different operating systems built with different object models using different programming languages and integrating them into robust systems for supporting critical business processes or scientific research programs. Application developers increasingly want and need to reach beyond tightly-coupled client-server environments to access functionality on remote systems that are very different in design and implementation, and which are owned and managed by other organizations.

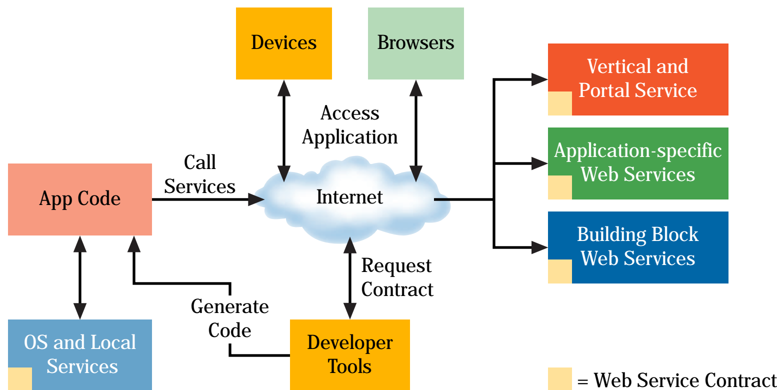
Because of its ubiquity, the Internet is driving this demand. Creating a viable architecture for web-based component interaction is one of the foremost challenges of distributed computing today, and is a major objective of the Microsoft® .NET platform. Microsoft's strategy is to support creation of a standards-based architecture for distributed applications on the Internet, and then to enable easy application development and deployment by providing a specially adapted toolset and runtime environment for creating and running highly distributed applications.

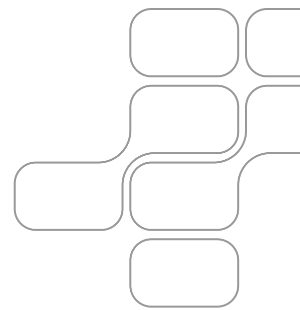
This paper describes the essential components of the new Microsoft® .NET framework, including the common language runtime, base class libraries, the services framework, and the programming models for building and integrating applications over the web. We will examine these components in general, but take a special look at .NET as a platform for building and supporting web-based applications of the kind now commonly called "Web Services".

A First Look at Web Services

Broadly speaking, a Web Service is simply a URL-addressable resource that programmatically returns information to clients who want to use it. One important feature of Web Services is that clients don't need to know how a service is implemented. In this sense Web Services represent black-box functionality that can be reused without worrying about how the service is implemented. Web Services provide well-defined interfaces, sometimes called contracts, that describe the services provided. Developers can assemble applications using a combination of remote services, local services, and custom code. For example, a company might assemble an online store using an internal authentication service, a third-party personalization service to adapt Web pages to each user's preferences, a credit-card processing service, a sales tax service to calculate tax on each transaction, package-tracking services from each shipping company, an in-house catalog service that connects to the company's internal inventory management applications, and a bit of custom code to make sure that their store stands out from the crowd. Figure 1 shows a model that illustrates how Web Services can be assembled to create fully functional distributed Web applications.

Figure 1 Web Services Application Model





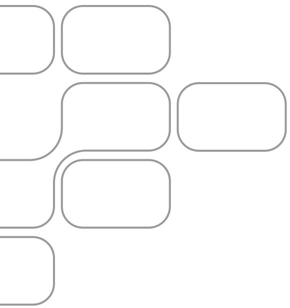
Web Services are, at their core, simply a component technology for the Internet. Unlike current component technologies, however, Web Services do not use object model-specific protocols such as DCOM, RMI, or IIOP that require specific, homogeneous infrastructures on both the client and service machines. While implementations tightly coupled to specific component technologies are perfectly acceptable in a controlled environment, they become impractical on the Web. As the set of participants in an integrated business process changes, and as technology changes over time, it becomes very difficult to guarantee a single, unified infrastructure among all participants. Web Services take a different approach; they communicate using ubiquitous Web protocols and data formats such as SOAP and XML. Any system supporting these protocols will be able to support Web Services.

Furthermore, a Web Service contract describes the services provided in terms of the messages the Web Service accepts and generates rather than how the service is implemented. By focusing solely on messages, the Web Services model is completely language, platform, and object model-agnostic. A Web Service can be implemented using the full feature set of any programming language, object model, and platform. A Web Service can be consumed by applications implemented in any language for any platform. As long as the contract that explains the service's capabilities and the message sequences and protocols it expects is honored, the implementations of Web Services and Web Service consumers can vary independently without affecting the application at the other end of the conversation.

The minimum infrastructure required by the Web Services model is purposefully low to help ensure that Web Services can be implemented on and accessed from any platform using any technology and programming language. The key to Web Service interoperability is reliance solely on Web standards. However, simply agreeing that Web Services should be accessed through standard Web protocols is not sufficient to make it easy for applications to use Web Services. Web Services become easy to use when a Web Service and Web Service consumer can rely on standard ways to represent data and commands, to represent Web Service contracts, and to figure out the capabilities a Web Service provides. Under the auspices of the W3C and other standards bodies, Microsoft and other companies have worked to define and support a variety of basic standards for interoperability and discoverability of Web services.

XML is the obvious choice for defining a standard yet extensible language to represent commands and typed data. While rules for representing commands and typed data using other techniques (such as encoding as a query string) could be defined, XML is specifically designed as a standard meta-language for describing data. The Simple Object Access Protocol (SOAP)[1] is rapidly becoming the industry standard for using XML to represent data and commands in communication among web service components. XML is also the basis of the Web Services Description Language (WSDL)[2] a grammar for documenting Web Service contracts. Universal Description, Discovery and Integration (UDDI)[3] describes a standard way for clients to locate web services, while WS-Inspection[4] (which embraces the earlier Disco standard) provides a standard mechanism for discovery from a provider of detailed information about available services.

A simple way to understand the roles of these standards is by analogy to the information sources that we use to take advantage of the services provided by a local business like a bank. UDDI is like the yellow pages where one can locate businesses of a particular kind. WS-Inspection/Disco is like a sign in a bank's window listing the specific services a particular business offers. WSDL is like the deposit slip that specifies what information is required (and in what format) to consummate a transaction with the bank, and SOAP is the protocol that defines a format for the electronic deposit slip transmitted to the bank. The goal of these standards as a set is to allow software clients to easily find, understand, and interact with services on the Internet without the need for human research of printed documentation and extensive hand coding of each relationship between client and service.



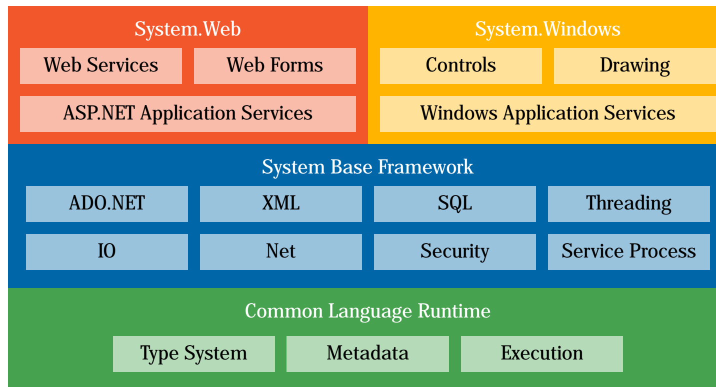
Standards like SOAP, WSDL, UDDI and WSInpection help developers since they do not need to understand and implement different ways to access each Web Service that they use. Even better, because they are documented and widely supported standards, well-tested, high-performance infrastructure supporting these standards can be supplied by development platforms, greatly simplifying the entire development process. All these standards are supported by Microsoft and, to the extent that they are complete and finalized, by the .NET platform.

The Microsoft .NET Framework

The goal of the Microsoft .NET Framework is to make it easy to build distributed applications, especially Web Services, on a wide variety of platforms. The Framework specification has been accepted by the standards section of the European Computer Manufacturer's Association (ECMA), thus permitting others to create their own implementations.[5]

Figure 2 shows the Microsoft .NET Framework architecture. Built on top of operating system services is a Common Language Runtime (CLR) that manages the needs of running code written in any supported programming language. This runtime supplies many services that help simplify code development and application deployment while also improving application reliability and security. The .NET Framework also includes a set of class libraries that developers can use from any supported programming language. Above that sit various application programming models that provide higher-level components and services targeted specifically at developing Web sites and Web Services. Let's take a closer look at each of these layers.

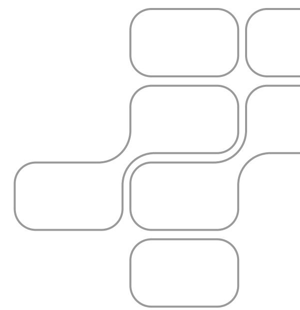
Figure 2 Microsoft .NET Framework Architecture



Common Language Runtime

The CLR, sometimes referred to as a "managed execution environment", loads and runs code written in any runtime-aware programming language. Code that targets the runtime is called managed code because the runtime itself assumes responsibility for tasks like creating objects, making method calls, enforcing security policies and performing garbage collection.

The architecture of the CLR is built on a two stage compilation process. Source code is compiled to an intermediate language, imaginatively named Microsoft Intermediate Language or MSIL. This intermediate code is then compiled at runtime to the target hardware platform by a JIT.[6] Thus all code running in the CLR is ultimately native code, and the only performance impact is the small latency due to compilation the first time an object is loaded into memory.



One of the most important features of the CLR is cross-language integration which allows complete language interoperability among any languages targeting the runtime.[7] To achieve this interoperability, the runtime makes use of a new common type system capable of expressing the semantics of most modern programming languages. The common type system defines a standard set of types and rules for creating new types. The runtime understands how to create and execute these types. Compilers and interpreters thus use runtime services to define types, manage objects, and make method calls instead of using tool or language-specific methods.

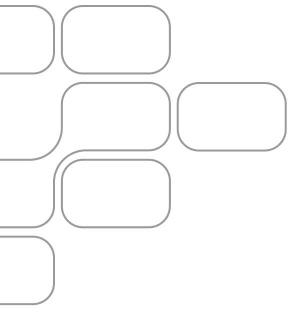
The primary design goal for the type system was to enable deep multi-language integration. Code written in one language can now inherit implementation from classes written in another language; exceptions can be thrown from code written in one language and caught in code written in another; and operations such as debugging and profiling work seamlessly regardless of the languages used to write the code. This means that developers of reusable class libraries no longer need to create versions for each programming language or compiler, and developers using class libraries are no longer limited to libraries developed for the programming language they are using.

Language interoperability is, however, only part of the goal of the .NET runtime. Full component interoperability also requires that adequate metadata describing each component be available to other components at runtime. Previous architectures like COM or CORBA store this metadata in external files that must be maintained separately from the component itself. The .NET Framework makes a significant departure from these earlier schemes by incorporating all metadata into the component package itself, making components completely self-describing. As a result, separate configuration information does not need to be deployed to identify developer requested service attributes. And best of all, since the metadata is generated from the source code during the compilation process and stored with the executable code, it is never out of sync with the executable.

In addition to improvements in deploying individual components, the Microsoft .NET Framework defines an application deployment model that addresses the complexities of application installation and DLL versioning (commonly known as "DLL Hell"). Addressing this problem, .NET introduces the notion of an assembly. An assembly is a group of resources and types, along with metadata about those resources and types, that is deployed as a unit. The metadata is called an assembly manifest and includes information such as a list of types and resources visible outside the assembly. The manifest also includes information about dependencies, such as the version of the assemblies used when the assembly was built. Developers can specify versioning policies to indicate whether the runtime should load the latest version of a dependent assembly installed on the system, a specific version, or the version used at build time.

Though it has always been possible for multiple copies of a software component to reside on the same system, in general only one of these copies can be registered with the operating system or loaded for execution. The policy for locating and loading components is global to the system. The .NET CLR adds the infrastructure necessary to support per-application policies that govern the locating and loading of components, generally referred to as side-by-side deployment.[8] In this scheme, assemblies can be private to an application or shared by multiple applications. Multiple versions of an assembly can be deployed on a machine at the same time. Application configuration information defines where to look for assemblies, thus the runtime can load different versions of the same assembly for two different applications that are running concurrently. This eliminates issues that arise from incompatibilities between component versions, improving overall system stability. If necessary, administrators can add configuration information, such as a different versioning policy, to assemblies at deployment time, but the original information provided at build time is never lost.

Because assemblies are self-describing, no explicit registration with the operating system is required. Application deployment can be as simple as copying files to a directory tree. Configuration information is stored in XML files that can be edited by any text editor.



Finally, the runtime also supplies integrated, pervasive security services to ensure that unauthorized users cannot access resources on a machine and that code cannot perform unauthorized actions. The Microsoft .NET Framework provides both code access security and role-based security. With code access security, developers can specify the required permissions their code needs to accomplish work. For example, code may need permission to write a file or access environment variables. This information is stored at the assembly level, along with information about the identity of the code. At load time and on method calls, the runtime verifies that the code can be granted the permissions it has asked for. If not, a security violation is reported. Policies for granting permissions, known as trust policies, are established by system administrators, and are based on evidence about the code such as who published the code and where it was obtained from, as well as the identity and requested permissions found in the assembly. Developers can also specify permissions they explicitly don't want granted, to prevent malicious use of their code by others. Programmatic security checks can be written even if the permissions required depend on information that isn't known until runtime.

In addition to code access security, the runtime supports role-based security, which builds on the same permissions model as code access security, except that the permissions are based on user identity rather than code identity. Roles represent categories of users and can be defined at development or deployment time. Policies for granting permissions are assigned to each defined role. At runtime, the identity of the user on whose behalf the code is running is determined. The runtime determines what roles the user is a member of and then grants permissions based on those roles.

Before looking at programming models in the Microsoft .NET Framework, let's take a moment to look at the services it provides atop the CLR.

The Base Class Libraries

As you may recall from Figure 2, on top of the common language runtime is the base framework, consisting of a large library of base classes which can be called from any supported programming language. Moreover, because learning class libraries is a major part of learning most any modern programming language, .NET has organized the base classes into a set of prescribed namespaces to reduce the learning curve for developers. There is not space here to review even a small part of the base class libraries, but after noticing the most important groups of classes, we'll look a bit more closely at a particular set of classes that are of particular importance to Web Services.

Some of the key base class libraries include libraries that developers would expect in a standard language library, such as collections, input/output, string, and numerical classes. In addition, the base class library provides classes to access operating system services such as graphics, networking, threading, globalization, and cryptography. The base framework also includes classes that development tools can use, such as debugging and profiling services.

These classes are important across all types of programming and applications. But one set of classes are particularly important to Web-based applications that most often need access to data across different platforms and environments. With XML rapidly emerging as the standard for this kind of data-interchange, Microsoft has thoroughly baked the power of XML into the .NET framework data access classes.

To provide data access, the services framework includes the ActiveX® Data Objects (ADO) class library, referred to as ADO.NET. This library provides high-performance stream APIs for connected, cursor-style data access, as well as a disconnected data model more suitable for returning data to client applications. Figure 3 illustrates the ADO.NET architecture and shows that any data-regardless of how it is actually stored-can be manipulated as XML or relational data, whichever is most appropriate for the application at a given point in time.

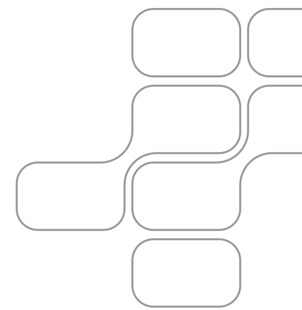
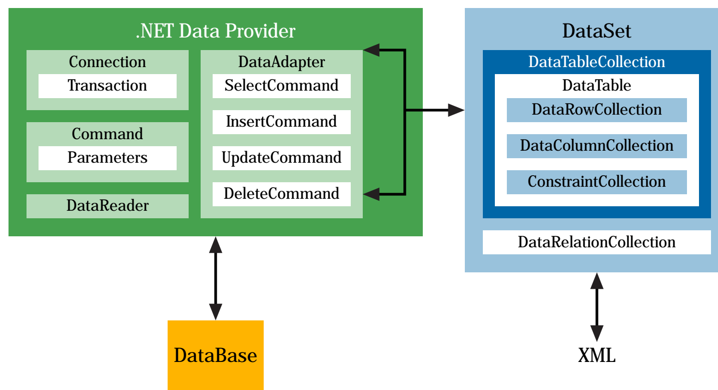


Figure 3 ADO.NET Architecture



One of the major innovations of ADO.NET is the introduction of the Dataset. More than just a disconnected record set, a Dataset in .NET is an in-memory data cache providing a relational view of retrieved data that can be manipulated using powerful components from the base libraries. Datasets know nothing about the source of their data-the Dataset may be created and populated programmatically or by loading data from a data store. No matter where the data comes from, it is manipulated using the same programming model and uses the same underlying cache.

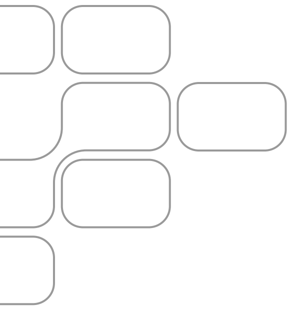
ADO.NET components have been designed to separate data access from data manipulation. Data access components include the Connection, Command, DataReader, and DataAdapter objects, and are used to fill data sets from data stores and to resolve changes to the data set back to the store. The .NET data provider is designed to be lightweight, creating a minimal layer between the data source and your code, increasing performance without sacrificing functionality.

Another critical innovation in ADO.NET is strong support for handling XML data. In ADO, all data can be viewed as XML, and just as DataReaders expose efficient stream access to relational data, XmlReaders expose efficient stream access to XML Data. Developers use a DataNavigator for scrolling and editing an in-memory XmlDocument. DataNavigators are functionally equivalent to the W3C Document Object Model (DOM), but are more efficient and provide an object model that maps nicely to the relational data view. DataNavigators support XPath syntax for navigating the data stream. ADO+ also provides an XMLDocument class for developers who want to continue to use the DOM as an object model for XML rather than the more efficient DataNavigator model.

Since all data can be viewed as XML, developers can take advantage of transformation and validation services for any data which are provided directly through the ADO.NET class libraries. ADO.NET supports schemas defined via DTDs, XSD, or XDR. The .NET Framework provides a specific transformation component that supports the W3C XSL Transformations (XSLT) specification, and also provides a validation engine that uses XML Schemas to validate an XmlReader.

Programming and Application Models in .NET

The ultimate reason for having an application architecture like .NET is, of course, to be able to create applications. In today's world there are at least two basic types of applications, which have, traditionally been programmed in sometimes very different ways.



One the one hand are traditional client applications with complete GUI interfaces, programmed using either command line compilers or visual development environments to create executable packages for local deployment. On the other hand web programming combines at least three different elements: a client side interface done through HTML, client-side scripting interwoven into the presentation code, and server side scripting in any of a number of different languages or implementation models from ASP to CGI scripting.

Though we have heretofore focused on .NET as an architecture for Internet-based web applications, the new platform programming model supports both standard programming of client applications and web application programming. But the new architecture seeks to simplify the developer's learning curve and task by offering a unified visual programming model that is the same across both domains. Let's take a brief look at programming in each of the two.

Windows® Forms

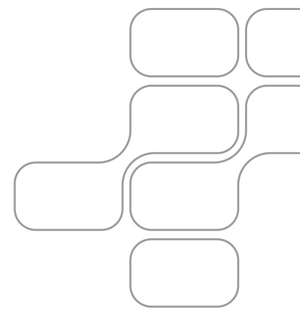
Developers writing client applications for Windows® can use what is now called the Win Forms application model which combines the best features of the popular forms-based visual programming style of Visual Basic® with the full power of traditional MFC user interface classes. In addition Win Forms provides full access to existing ActiveX controls and new features of Windows® 2000 or Windows® XP, such as transparent, layered, and floating windows.

Win Forms also takes advantage of the Microsoft .NET Framework runtime to provide full security and ease of deployment. The framework security model can safely execute applications and components on client machines provided they are written using the Win Forms model or used from a Win Forms application. At the same time, applications can be configured to use the versions of shared components they were built and tested with, rather than using whatever component versions happen to be installed on the client machine, improving application reliability and eliminating one of the major causes of application support calls: incompatible versions of user interface controls and other shared components.

Web Applications

Web applications built on the Microsoft .NET Framework share the Win Forms programming model, again based on the popular forms-oriented visual programming style of Visual Basic. But here there is a distinct application model, called ASP.NET, in which a Web application is seen as just a set of URLs rooted at some base URL. Thus it encompasses both Web applications that generate pages for display in a browser and Web Services. Like Win Forms, ASP.NET takes advantage of the common language runtime and services framework to provide a reliable, robust, scalable hosting environment for Web applications. ASP.NET also benefits from the common language runtime assembly model to simplify application deployment. In addition, it provides services to simplify application development with support for things like state-management.

At the core of ASP.NET is the HTTP runtime, a high-performance runtime for processing HTTP requests that is responsible for processing all incoming HTTP requests, resolving the URL of each request to an application, and then dispatching the request to the application for further processing. The HTTP runtime is multithreaded and processes requests asynchronously, so it cannot be blocked by bad application code from processing new requests. Furthermore, the HTTP runtime assumes that failures will occur, so it is engineered to automatically recover as best it can from access violations, memory leaks, deadlocks, and so on. Barring hardware failure, the runtime aims for 100 percent availability.



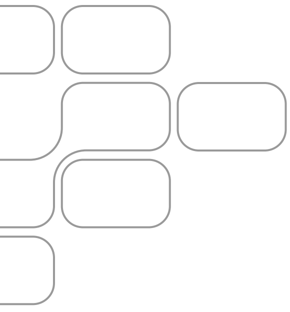
ASP.NET uses the Microsoft .NET Framework deployment model based on assemblies, thus gaining all its benefits such as XCOPY deployment, side-by-side deployment of assemblies, and XML-based configuration. ASP.NET also supports live updating of applications. An administrator doesn't need to shut down the Web server or even the application to update application files. Application files are never locked, so they can be overwritten even when the application is running. When files are updated, the system gracefully switches over to the new version. The system detects file changes, launches a new instance of the application using the new application code, and begins routing incoming requests to that application. When all outstanding requests being processed by the existing application instance have been handled, that instance is shut down.

Because the Web is a fundamentally stateless model with no correlation between HTTP requests, writing Web applications can be especially difficult, with programmers having to devise a variety of ad hoc mechanisms to maintain state information. ASP.NET enhances the state management services introduced by ASP to provide three types of state to Web applications: application, session, and user. Application state is specific to an application instance and is not persisted. Session state is specific to a user session with the application. In ASP.NET, session state is stored in a separate process and can even be configured to be stored on a separate machine. This makes session state usable when an application is deployed on a Web farm. User state resembles session state, but generally does not time out and is persisted. Thus user state is useful for storing user preferences and other personalization information. All the state management services are implemented as HTTP modules, so they can be added or removed from an application's pipeline easily. If additional state management services are required beyond those supplied by ASP.NET, they can be provided by a third-party module.

ASP.NET also provides caching services to improve performance. An output cache saves completely rendered pages, and a fragment cache stores partial pages. Classes are provided so applications, HTTP modules, and request handlers can store arbitrary objects in the cache as needed. Now let's take a quick look at the two higher-level programming models that build on the ASP.NET programming model: ASP.NET Web Forms and ASP.NET Web Services.

Web Forms

Just as Win Forms brings the productivity benefits of a forms-based programming model to client applications in any .NET supported language, Web Forms brings the same model to web applications, thus providing .NET with a unified programming model across all application domains and languages. Web Forms support traditional ASP syntax that mixes HTML content with script code, but it also promotes a more structured approach that separates application code from user interface presentation layer. Web Forms controls are responsible for generating the user interface, typically in the form of HTML. ASP.NET comes with a set of Web Forms controls that mirror the typical HTML user interface widgets (including listboxes, text boxes, and buttons), and an additional set of Web controls that are more complex (such as calendars and ad rotators). One important feature of these controls is that they can be written to adapt to client-side capabilities; the same pages can be used to target a wide range of client platforms and form factors. In other words, Web Forms controls can detect the level of the client that is hitting a form and return an appropriate user experience-maybe HTML 3.2 for a down-level browser and Dynamic HTML for Internet Explorer 5.0. The separation of code and content enables ASP.NET pages to be dynamically compiled into managed classes for fast performance. Each incoming HTTP request is delivered to a new page instance so that developers do not need to be concerned about thread safety in their code.



ASP.NET for Web Services

The ASP.NET Web Services infrastructure provides a high-level programming model for building Web Services with ASP.NET. While not required for building a Web Service in .NET, with ASP.NET developers don't need to understand the specifics of HTTP, SOAP, WSDL, or any other specifications for Web Services in order to use the programming model. You create a Web Service with ASP.NET by authoring a file with the extension .asmx and deploying it as part of a Web application. The ASMX file either contains a reference to a managed class defined elsewhere or the class definition itself. The class is derived from the WebService class supplied by ASP.NET. Public class methods are exposed as Web Service methods simply by marking them with the WebMethod attribute. These methods can then be invoked by sending HTTP requests to the URL of the ASMX file. This is all there is to it. ASP.NET inspects the class metadata to automatically generate an WSDL file when requested by the caller.

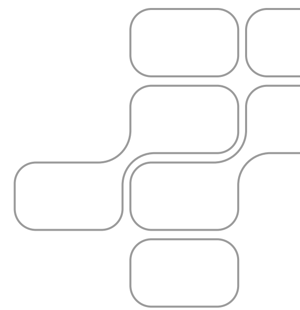
Clients may submit service requests via SOAP, HTTP GET, and HTTP POST. Conventions are defined for encoding methods and parameters as query strings for HTTP GET and form data for HTTP POST. The HTTP GET and HTTP POST mechanisms are not as powerful as SOAP, but they enable clients that don't support SOAP to access a Web Service.

The ASP.NET Web Services model assumes a stateless service architecture, because Stateless architectures are generally more scalable than stateful architectures. Thus, each time a service request is received, a new object is created, the request is converted into a method call, and the object is destroyed once the method call returns. Services can use the ASP.NET State Management services if they need to maintain state across requests. Web Services based on ASP.NET run in the Web application model, so they get all the security, deployment, and other benefits of that model.

Conclusion

As technology moves forward into the era of rich connectivity across a wide range of devices, there is a growing need for architectures that can support building large-scale loosely-coupled systems. Web Services provides a simple, flexible, standards-based model for binding applications together over the Internet that takes advantage of existing infrastructure and relies on well established protocols like HTTP. Web applications can be easily assembled with locally developed services and existing services, irrespective of the platform, development language, or object model used to implement any of the constituent services or applications.

But more than just an architecture for commercial application development, Microsoft's new platform technology offers many appealing features for teaching about computing. .NET supports many different computing languages, allowing students and instructors to choose the language most appropriate to their immediate learning goals. And using utilities like ildasm.exe that come with the .NET SDK, students and instructors can inspect intermediate language code directly to understand language and compiler design issues. It features a runtime environment that manages memory and provides strong security and easy deployment, so students can avoid many frustrating errors and begin to create satisfying real-world applications much sooner. .NET provides a single programming model across conventional and web applications, so students can learn new application models without having to struggle learning new programming models as well. And because support for things like networking and XML are baked into the foundation class libraries, students can concentrate on building distributed applications to solve problems without having to learn or create necessary "plumbing code". For all these reasons and many more, Microsoft's new platform may ultimately offer as much to teaching about computing as it does to the world of commercial application development.



[1] See full documentation and history at: http://www.w3.org/2000/xp/

[2] See full documentation and history at: http://www.w3.org/TR/wsdl

[3] See full documentation and history at: http://www.uddi.org/ . The UDDI project creates a platform-independent, open framework for describing services, discovering businesses, and integrating business services using the Internet, as well as an operational registry that is available today.

[4] WS-Inspection is a specification jointly proposed by Microsoft and IBM. See: http://www-106.ibm.com/developerworks/webservices/library/ws-wsilspec.html .

[5] In December 2000, C# and the Common Language Infrastructure were accepted by ECMA. For more information see: http://www.microsoft.com/PressPass/press/2001/Dec01/12-13ECMApr.asp . Microsoft itself is sponsoring one of the first implementations of the .NET framework on a non-windows platform. In partnership with Corel, Inc., Microsoft will deliver an implemention on FreeBSD Unix and will make complete source code available for teaching and research purposes. See http://www.microsoft.com/PressPass/press/2001/Jun01/06-27CorelPR.asp .

[6] The JIT compiler is a pluggable component of the CLR architecture, thus allowing the development of specialized compilers to be optimized for particular tasks.

[7] Project 7 was a research project supporting academic researchers targeting a variety of academic languages at the .NET CLR. For more information see: http://research.microsoft.com/project7.net/project_7.htm .

[8] The CLR also permits the execution of unmanaged code. Under these circumstances things are slightly more complicated because the CLR, by definition, does not have the ability to determine or enforce appropriate policies for such code.

©2002 Microsoft Corporation. All rights reserved. Microsoft, ActiveX, the .NET logo, Visual Basic, Visual Studio, the Visual Studio logo, and Windows are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.