CloverLink SYSTEMS, INC.

Your Link to Tomorrow's Business

A Platform for Web Services

Microsoft Corporation — January 2001

An overview on the Web Services model for building applications.

Introduction

If you've been following the news about Microsoft® .NET, you'll know that Web Services play a major role in the .NET application architecture. In the .NET vision, an application is constructed using multiple Web Services that work together to provide data and services for the application. However, just because Web Services are usually discussed in the context of .NET, you should not assume that you must wait for the Microsoft .NET Framework or Microsoft® Visual Studio.NET to build, deploy, or consume Web Services. Web Services are a very general model for building applications and can be implemented for any operating system that supports communication over the Internet.

In this article, we will:

- Review the definition of Web Services.
- Define a generic architecture for Web Services and relate that architecture to Microsoft® Windows® DNA and .NET.
- Specify some requirements on a platform that provides good support for building, deploying, or consuming Web Services.
- Indicate the products and technologies provided by the Microsoft platform to address these requirements.

Web Services Defined

A Web Service is programmable application logic accessible using standard Internet protocols. Web Services combine the best aspects of component-based development and the Web. Like components, Web Services represent black-box functionality that can be reused without worrying about how the service is implemented. Unlike current component technologies, Web Services are not accessed via object-model-specific protocols, such as the distributed Component Object Model (DCOM), Remote Method Invocation (RMI), or Internet Inter-ORB Protocol (IIOP). Instead, Web Services are accessed via ubiquitous Web protocols and data formats, such as Hypertext Transfer Protocol (HTTP) and Extensible Markup Language (XML). Furthermore, a Web Service interface is defined strictly in terms of the messages the Web Service accepts and generates. Consumers of the Web Service can be implemented on any platform in any programming language, as long as they can create and consume the messages defined for the Web Service interface.

There are a few key specifications and technologies you are likely to encounter when building or consuming Web Services. These specifications and technologies address five requirements for service-based development:

- A standard way to represent data
- A common, extensible, message format
- A common, extensible, service description language
- A way to discover services located on a particular Web site
- A way to discover service providers

XML is the obvious choice for a standard way to represent data. Most Web Service-related specifications use XML for data representation, as well as XML Schemas to describe data types.

The Simple Object Access Protocol (SOAP) defines a lightweight protocol for information exchange. Part of the SOAP specification defines a set of rules for how to use XML to represent data. Other parts of the SOAP specification define an extensible message format, conventions for representing remote procedure calls (RPCs) using the SOAP message format, and bindings to the HTTP protocol. (SOAP messages can be exchanged over other protocols, but the current specification only defines bindings for HTTP.) Microsoft anticipates that SOAP will be the standard message format for communicating with Web Services.

Given a Web Service, it would be nice to have a standard way to document what messages the Web Service accepts and generates—that is, to document the Web Service contract. A standard mechanism makes it easier for developers and developer tools to create and interpret contracts. The Web Services Description Language (WSDL) is an XML-based contract language jointly developed by Microsoft and IBM. We anticipate that WSDL will be widely supported by developer tools for creating Web Services.

Note Over the past year Microsoft and IBM have proposed several contract languages: Service Description Language (SDL), SOAP Contract Language (SCL), and Network Accessible Services Specification Language (NASSL). While these are all superseded by WSDL, some early development tools use these languages. You might need to translate the provided contract into the contract language your development tool understands in order to consume a Web Service.

Developers will also need some way to discover Web Services. The Discovery Protocol (Disco) specification defines a discovery document format (based on XML) and a protocol for retrieving the discovery document, enabling developers to discover services at a known URL. However, in many cases the developer will not know the URLs where services can be found. Universal Description, Discovery, and Integration (UDDI) specifies a mechanism for Web Service providers to advertise the existence of their Web Services and for Web Service consumers to locate Web Services of interest.

For more information about Web Services and these key specifications, see Web Services Essentials in the MSDN Library.

Web Services, Windows DNA, and .NET

Recall that the .NET vision imagines applications will be constructed from multiple Web Services that work together to provide data and services for the application. This is shown in Figure 1.

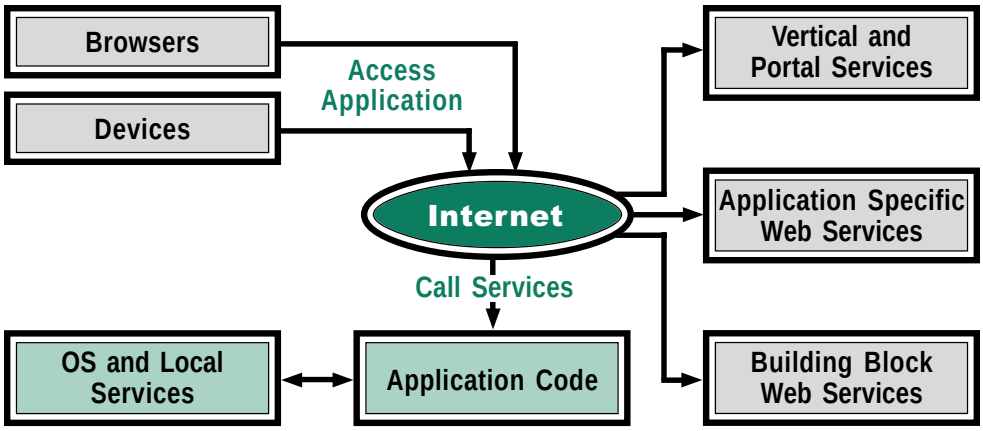

Figure 1. *.NET Application Architecture*

This diagram, as well as the definition of Web Services just provided, is concerned with the external appearance of the Web Services. After all, we've said that as long as a client application can create and consume the appropriate messages, it doesn't need to know anything about the internals of the Web Services it uses. Developers of Web Services will obviously care about the internal structure as well.

Figure 2 shows a generic architecture for a Web Service. The architecture is divided into five logical layers. Furthest from the client is the *data layer*, which stores information required by the Web Service. Above the data layer is the *data access layer*, which presents a logical view of the physical data to the business layer. The data access layer isolates business logic from changes to the underlying data stores and ensures the integrity of the data. The business layer implements the business logic of the Web Service. As in Figure 2, it is often subdivided into two parts: the *business façade* and the *business logic*. The business façade provides a simple interface that maps directly to operations exposed by the Web Service. The business façade uses services provided by the business logic layer. In a simple Web Service, all the business logic might be implemented by the business façade, which would interact directly with the data access layer. Client applications interact with the Web Service *listener*. The listener is responsible for receiving incoming messages containing requests for service, parsing the messages, and dispatching the request to the appropriate method on the business façade. If the service returns a response, the listener is also responsible for packaging the response from the business façade into a message and sending that back to the client. The listener also handles requests for contracts and other documents about the Web Service. If you think about it, the only part of the Web Service that knows it is part of a Web Service is the listener!

This architecture is very similar to the *n*-tier application architecture defined by Windows DNA. As shown in Figure 3, the Web Service listener is equivalent to the presentation layer of a Windows DNA application. A common development scenario is likely to include exposing functionality of an existing Web application for programmatic access—that is, adding a Web Service to an existing application. As this figure shows, that could be as simple as implementing a Web Service listener that accesses the existing business façade.

In the Windows DNA architecture, we're used to thinking about implementing the data layer using databases and the business layer using COM components. But what if the data access layer gets its data from a Web Service instead of a database? Or the business façade calls a Web Service to do part of its work? Suddenly our application architecture looks a lot like Figure 1. In some respects, the .NET application architecture simply extends the Windows DNA application architecture across the Web.

Note The requirements and usage patterns for programmatic access will often be sufficiently different from the requirements and usage patterns of the existing Web application that additional work is

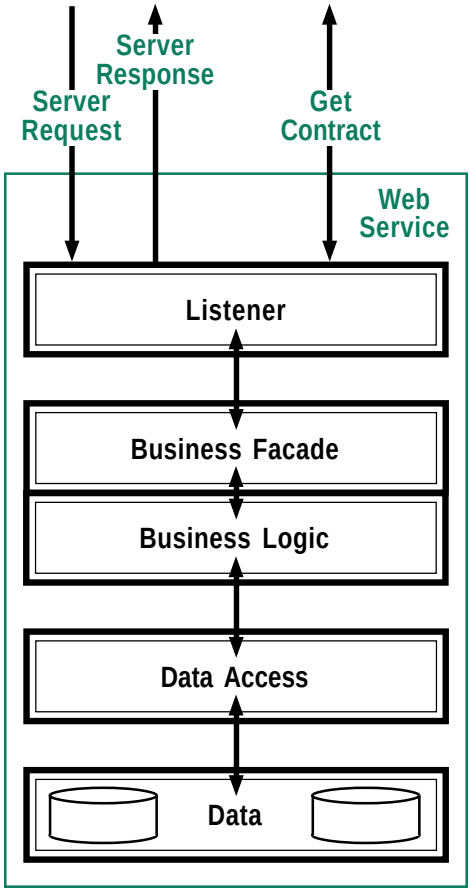

Figure 2. *Generic Web Services architecture*

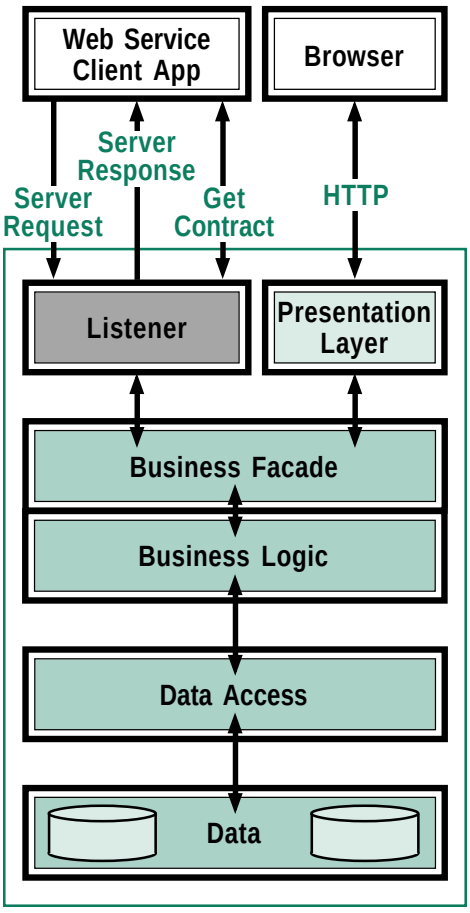

Figure 3. *Relationship of Web Service architecture to Windows DNA architecture*

required. These differences might impact all layers of the application architecture. For example, the physical data schema might not be designed to handle new kinds of queries exposed by a Web Service, or the volume of queries made by client applications. In addition, Web Services have extremely high reliability requirements. Unless your existing Web application is highly available, capable of dealing with unexpected input values, and so on, additional work might be needed to meet these new requirements.

Platform Requirements

Given the similarities between the Windows DNA architecture and the Web Services architecture, you should not be surprised to learn that developing Web Services is not all that different from developing Web applications, or that you can develop Web Services using today's Windows DNA technologies. However, there are some notable differences. In particular, in order to implement a Web Service listener, you'll probably need to understand SOAP messages and generate SOAP responses, provide a WSDL contract for your service and Disco file for your site, and advertise your service via UDDI. If you're consuming Web Services, you might need to use UDDI or Disco to locate services and service contracts, interpret a WSDL contract for the service, and generate appropriate SOAP messages and interpret SOAP responses.

In addition, because applications rely on Web Services, it is critical that these services are completely dependable. A Web Service should always be available. It must not make mistakes, lose requests, fault in the face of invalid requests, or corrupt persisted data. It should always be able to meet client demand with acceptable performance. In the rare event that a fault occurs, the Web Service should continue processing requests as best it can. In other words, a Web Service needs all the "abilities"—scalability, reliability, availability, and so on. If a Web Service is not dependable, application developers will not use it.

Note For more information on building scalable, reliable, and available Web applications, see the MSDN Online E-Commerce Resources page at http://msdn.microsoft.com/ecommerce/. While the resources on these pages talk about Web applications, the techniques are applicable to Web Services as well.

Systems administrators for service providers will not permit Web Services to be deployed unless the Web Services are easy to deploy and manage. It should be possible to deploy a Web Service without using special tools, either locally or remotely. The deployment process should be easy to learn and easy to replicate. It should be easy to deploy a new version of a Web Service, either side by side with the existing version or by replacing the existing version. Management tools should make it easy to monitor and tune Web Service performance, both in isolation and in combination with other services, as demand varies. It must be possible to secure a Web Service so that only authorized consumers can use it. Perhaps most importantly, deploying one Web Service must not impact the availability or performance of any other Web Service—even if multiple Web Services share implementation components.

The requirements of Web Services consumers and system administrators impose a considerable burden on Web Service developers that's not specifically related to any functionality provided by the service. Web Service developers will want infrastructure and tools support that make it easier to implement secure, reliable, scalable, manageable, and highly available Web Services. Developers will also want infrastructure and tools to help them debug, profile, and trace execution of their code and the infrastructure services they are using. Ideally, you will not need to learn a new programming language in order to use this infrastructure and these tools. In fact, the more you can leverage your existing skills, components, applications, and data stores, the better.

Much of the difficulty in implementing scalable, highly available code is connected with properly managing resources—such as processes, threads, and shared state—when multiple concurrent requests are received by a service. Middleware that manages these resources and lets you write service logic as if a single client is accessing it can greatly improve the reliability, scalability, and availability of Web Services, as well as greatly simplify development of the Web Service.

In practical terms, the middleware provides a standard hosting environment for Web Service implementation code. The hosting environment is responsible for:

- Listening for incoming HTTP requests.
- Performing security authentication and authorization checks.
- Dispatching authorized requests to the correct service.
- Ensuring services are isolated from each other and the hosting environment—that is, a service has its own memory, services cannot block other services from executing, and service faults cannot cause other services or the hosting environment to fault.
- Automatically recovering from service, hosting environment, and system failures.
- Providing administrative facilities for deploying, monitoring, and controlling services.
- Managing resources such as processes, threads, and shared state on behalf of each service.

Of course, some Web Services will have non-standard requirements that the standard hosting environment does not support. Thus, the environment must be flexible enough that you can replace features that don't meet your needs.

APIs for constructing and parsing messages will also enhance your productivity. At a minimum, APIs must support reading and writing XML streams. Specific APIs for standards such as SOAP, WSDL, Disco, and UDDI will improve productivity as well as overall reliability by eliminating the need for you to write parsing and formatting logic. The APIs also reduce the need for you to learn every detail of the specifications.

If you are implementing Web Services using component technologies, additional productivity gains can be achieved if the system provides services to activate objects on demand and map messages to object method calls—essentially implementing the Web Service listener for you. Similarly, if you're consuming Web Services, you'll want tools that construct proxy objects hiding the details of formatting and sending a service request, then interpreting the response.

A Roadmap to the Microsoft Platform

Microsoft Windows 2000 provides the basic infrastructure required to implement Web applications and Web Services. Windows 2000 Internet Information Services (IIS) and Component Services (a.k.a. COM+) provide a hosting environment that meets most of the requirements just listed. Windows 2000 also provides APIs to help you implement all layers of a Web Service:

- COM for implementing the business façade, business logic, and data access layers
- ADO, OLE DB, and ODBC for implementing data access to a variety of data stores
- MSXML to help construct and consume XML messages in the Web Service listener
- Active Server Pages (ASP) or ISAPI for implementing the Web Service listener

To improve availability and scalability of your Web Service, you can use the Network Load Balancing (NLB) and Clustering services of Windows 2000 Advanced Server and Datacenter Server. Access to Web Services can be restricted using IPSec, HTTP Basic authentication, Digest authentication, Kerberos 5 authentication, NTLM authentication, or your own custom scheme. IPSec, SSL, and Windows cryptography services can be used to ensure data privacy.

.NET Enterprise Servers

Microsoft also supplies several server products you might find useful when implementing and deploying Web Services. The latest versions of these products, collectively known as the .NET Enterprise Servers, have been enhanced to support the Web and XML. The .NET Enterprise Servers include:

- Application Center, for deploying and managing your Web applications and Web Services.
- BizTalk™ Server, for business process orchestration and document interchange. BizTalk Server contains extensive support for XML and SOAP-based messages, transmitted over a variety of protocols, including HTTP, SSL, and SMTP. Developers who are most interested in exposing business processes to their business partners will likely build solutions based on BizTalk Server that end up fitting the definition of a Web Service, rather than explicitly setting out to implement Web Services.
- Commerce Server, an application platform for building e-commerce Web applications.
- Exchange Server, for messaging and collaboration. Exchange includes the Microsoft Web Storage System, which provides hierarchical data storage of heterogeneous documents. XML is the native data format for many kinds of documents in the Web Storage System. Exchange is tightly integrated with IIS, to support SMTP, POP, and direct access to data via HTTP. Exchange provides a complete application platform for building collaborative workflow applications that work over the Web. As with BizTalk Server, developers who are most interested in building workflow applications will likely build solutions based on Exchange or Microsoft® SharePoint™ Portal Server that end up fitting the definition of a Web Service, rather than explicitly setting out to implement Web Services.
- Host Integration Server, for accessing mainframe applications and data stores. Host Integration Server is the evolution of Microsoft SNA Server.
- Internet Security and Acceleration Server, which provides firewall and Web caching services.
- Mobile Information Server, for wireless access to enterprise data.
- SQL Server™, for relational data storage. SQL Server includes extensive support for XML. Relational data can be queried and modified as XML, eliminating the need to handcraft formatting logic in your applications. You can also provide direct access to SQL Server data stores and OLAP cubes via HTTP, using the SQL Server XML services. If your Web Service simply handles queries for data without a great deal of business logic and you don't need to provide a SOAP-based interface, you might consider simply using SQL Server XML as the Web Service listener.

Today's Tools for SOAP

Note that, with the exception of BizTalk Server, these technologies and products do not support SOAP, WSDL, Disco, or UDDI. Developers creating SOAP-based Web Services with today's technologies have three basic choices:

- Roll-your-own, using MSXML, ASP, or ISAPI, etc. The downside to this, of course, is that you need to implement and test everything yourself—and figure out how to comply with the relevant specifications!
- Use the SOAP Toolkit for Visual Studio 6.0 to build a Web Service listener that connects to a business façade implemented using COM. (The SOAP Toolkit can be used if the business façade is not implemented as a COM component, but you cannot leverage the wizard that generates the listener as easily.) Note that the SOAP Toolkit is a sample provide by MSDN. The Toolkit understands SOAP over HTTP and SSL, but does not help you create Disco documents or UDDI registrations. It supports an older contract language called SDL, rather than WSDL.
- Use the Microsoft Soap Toolkit version 2 to build a Web Service listener that connects to a business façade implemented using COM. The Microsoft Soap Toolkit version 2 is scheduled to for publication on MSDN Online in the first quarter of 2001. Version 2 supports SOAP over HTTP, and can be used to create a WSDL file describing your service. You will still need to create Disco documents and UDDI registrations manually.

Both versions of the SOAP Toolkit provide tools to help developers consume Web Services as if they were COM components. The SOAP Toolkit provides a COM component called the Remote Object Proxy Engine (ROPE) that can be used by client applications. ROPE uses an SDL file to dynamically create Automation methods you can call on a proxy object. If you want to use a Web Service that doesn't supply an SDL file, you will need to create one. Version 2 of the SOAP Toolkit provides similar functionality, based on WSDL files.

Moving Forward with .NET

The .NET Framework is Microsoft's next-generation platform for building Web applications and Web Services. It is built from the ground up to meet the needs of Web Services developers and consumers, with pervasive support for Web standards such as XML and SOAP. Some of the key features of the .NET Framework for Web Services developers include:

- *A common language runtime* that manages the needs of running code written in any programming language and eliminates the need to implement special interfaces such as **IUnknown** and **IDispatch**. Developers simply implement classes in their chosen programming language. Classes are completely self-describing, so there's no need for separate type libraries or IDL files.
- *Interoperability with existing COM components*. Existing COM components look like managed classes to managed applications; managed classes look like COM components to unmanaged applications.
- *An improved application deployment model*, which lets you specify exactly what versions of dependent DLLs to use. Application configuration information can be specified in text files, simplifying administration of applications. Many applications can be deployed by simply copying files on to the target machine (sometimes called "Xcopy Deployment").
- *Integrated, pervasive security services* to ensure that unauthorized users cannot access code or perform unauthorized actions.
- *ADO.NET*, which provides classes to access XML documents and relational data stores. As the name implies, ADO.NET is the evolution of Microsoft ActiveX® Data Objects (ADO).
- *Lightweight application isolation* based on application domains. An application domain represents an isolation boundary. An isolated application can be independently stopped and debugged, cannot access code or resources of other applications, can fault without causing other applications to fail, and has a baseline set of authorization checks that can be performed before the application is launched. Multiple application domains can run within the same process.
- *A robust HTTP runtime* for processing HTTP requests, engineered to automatically recover as best it can from access violations, memory leaks, deadlocks, and so on. Web applications and Web Services run in application domains, so a fault in one application domain doesn't bring down other application domains or the hosting environment. Application domains are launched on demand; if a fault stops an application domain, the next incoming request simply launches a new one. The runtime also supports preemptive cycling of applications to improve overall system stability in the face of applications that leak resources. Application DLLs are never locked, so new versions can be deployed without shutting down the application or Web server—when a new DLL is detected, a new application domain is launched to handle new requests and any existing application domains are shut down when they have no outstanding requests.
- *ASP.NET*, which provides a low-level programming model equivalent to ISAPI (but easier to implement), along with high-level programming models for building Web applications (known as Web Forms) and Web Services. ASP.NET supports basic, digest, and NTLM authentication, as well as Microsoft Passport authentication and custom cookie-based authentication for applications that use a private account database for authenticating users.
- *.NET Remoting* for activating objects and making method calls across context, application domain, process, or machine boundaries. For cross-machine calls, .NET Remoting supports both a DCOM-like binary wire protocol over TCP/IP and the SOAP wire protocol over HTTP or SMTP. The architecture is extensible, so that additional wire protocols and transports can be supported.

ASP.NET Web Services are the preferred technology for implementing Web Services based on the .NET Framework. ASP.NET Web Services support service requests using SOAP over HTTP, as well as HTTP **GET** or **POST**. ASP.NET Web Services automatically generate WSDL and Disco files for your Web Services. You can use ASP.NET Web Services to implement a Web Service listener that accesses a business façade implemented as a COM component or managed class. The .NET Framework SDK also provides tools to generate proxy classes that client applications can use to access Web Services.

Note that ASP.NET Web Services, as with the other tools we have discussed, do not expose the server-side types to client applications. The implementation is completely hidden inside the Web Service. All the tools discussed assume a stateless programming model as well—that is, each incoming request is handled independently. The only state maintained between requests is anything persisted in data stores.

If you need a more tightly coupled, object-based programming model between client and server, you'll want to use .NET Remoting. .NET Remoting provides remote access to server-side objects with full type fidelity. Clients can obtain references to server-side objects and control the lifetime of those objects. If you use these object lifetime services, however, client applications will need to be implemented using .NET Remoting as well.

In addition to the features provided by the .NET Framework, Microsoft Visual Studio.NET provides additional tools to help you build, deploy, and consume Web Services. For example, the IDE supports UDDI and Disco for locating Web Services, and understands how to generate client-side proxies from WSDL files. Visual Studio.NET also includes ATL Server, which C++ developers using ATL can use to construct Web Service listeners that connect to a business façade implemented as a C++ class. ATL Server supports SOAP over HTTP and will automatically generate WSDL files for your Web Services. It also provides tools to generate C++ proxy classes that client applications can use to access Web Services.

As you can see, Microsoft provides a number of tools and technologies that will help you build, deploy, and consume Web Services.

Mary Kirtland
Microsoft Developer Network

CloverLink SYSTEMS, INC.

Your Link to Tomorrow's Business

CloverLink Systems, Inc. offers a full spectrum of Internet application development services. We specialize in analysis, design and development of Internet and Intranet based solutions, using leading-edge technologies backed by an experienced professional staff and a sound business strategy. We are dedicated to bringing your products and services to market on time and on budget!

Let us help you achieve your company's unique goals and vision by providing a development proposal to fit your needs.

Call today and put the CloverLink team to work for you.

Microsoft .net

Microsoft CERTIFIED Partner

Corporate Offices

CloverLink Systems, Inc.
1486 Sunshine Drive
Glendale, CA 91208

e-mail: **Sales@CloverLink.com**
http://www.CloverLink.com

800.378.8348