All About ASMX 2.0, WSE 3.0, and WCF

**The release of** the Microsoft® .NET Framework 2.0 reshapes the Web services landscape in several interesting, and perhaps confusing, ways. So this month I'm going to field some of the most common questions related to ASP.NET Web Services (ASMX) 2.0, Web Services Enhancements (WSE) 3.0, and Windows® Communication Foundation (WCF), which is part of WinFX®.

Q Can you explain how the new ASMX 2.0 support for interfaces works? Which of the attributes go on the interface and which go on the class?

Q Can you explain how the new ASMX 2.0 support for interfaces works? Which of the attributes go on the interface and which go on the class?

A The ASMX framework in the .NET Framework 2.0 introduces a significant programming model improvement: the ability to define your Web service contracts in .NET interface definitions. This feature allows you to annotate .NET interface definitions with the usual System.Web.Services attributes ([WebServiceBinding], [WebMethod], [SoapDocumentMethod], and so forth). Then you can implement the .NET interface on a .NET class to effectively implement the service contract.

A The ASMX framework in the .NET Framework 2.0 introduces a significant programming model improvement: the ability to define your Web service contracts in .NET interface definitions. This feature allows you to annotate .NET interface definitions with the usual System.Web.Services attributes ([WebServiceBinding], [WebMethod], [SoapDocumentMethod], and so forth). Then you can implement the .NET interface on a .NET class to effectively implement the service contract.

This approach is very similar to the one in WCF (the main difference being the attribute names). It allows you to decouple the service contract from the implementation code, thereby making things easier to manage and reuse over time, not to mention making the code much easier to read. **Figure 1** shows an example of this approach.

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifFigure 1 QuoteService ASMX Definitions

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl22_ctl00_ctl00_code');" \o "Copy Code)

using System.Web.Services;

[WebServiceBinding(

Name = "QuoteService",

Namespace = "http://example.org/stocks",

ConformsTo = WsiProfiles.BasicProfile1\_1,

EmitConformanceClaims=true)]

public interface IQuoteService

{

[WebMethod]

StockQuote GetQuote(string symbol);

[WebMethod]

List<StockQuote> GetQuotes(List<string> symbols);

}

[WebService(Namespace="http://example.org/stocks")]

public class QuoteService : IQuoteService

{

public StockQuote GetQuote(string symbol)

{

... // retrieve and return new StockQuote

}

public List<StockQuote> GetQuotes(List<string> symbol)

{

... // retrieve and return List<StockQuote>

}

}

As illustrated in **Figure 1**, [WebServiceBinding] and [WebMethod] go on the interface while [WebService] goes on the class. When employing this model, all attributes that affect the service contract definition must be placed on the interface definition and not on the class. This includes [SoapDocumentService], [SoapDocumentMethod], and the various System.Xml.Serialization attributes that you could apply to method signatures. You cannot mix and match these attributes across the interface and class definitions, and ASMX helps to ensure that you don't.

For example, once you've derived a class from an interface that contains these service contract attributes, ASMX will inspect the class at run time to make sure that you haven't used any additional attributes that might further change the service contract. If ASMX finds one, it will display an error indicating you've violated the model (see **Figure 2**).

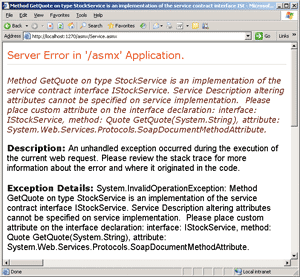


Figure 2 **ASMX Service Attribute Usage Error**

Although this minimizes the number of attributes you'll need to use on the derived class, there are still a few that come into play for configuring the service endpoint. You'll use [WebService] on the class to specify the <wsdl:service> details of the endpoint. This makes sense because you could have multiple services implementing the same service contract. Likewise, if you're using WSE 3.0, you'll use [Policy] on the class to apply a security policy configuration to the specific endpoint.

However, in some instances, you might also need to use [WebMethod] on the derived methods when you need to configure local processing behaviors through the various [WebMethod] properties (BufferResponse, CacheDuration, EnableSession, TransactionOption). For example, in the code in **Figure 3** I'm using [WebMethod] to configure caching of the responses.

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifFigure 3 Configuring WebMethod Caching

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl23_ctl00_ctl00_code');" \o "Copy Code)

[WebService(Namespace="http://example.org/stocks")]

public class QuoteService : IQuoteService

{

[WebMethod(CacheDuration=60)]

public StockQuote GetQuote(string symbol)

{

... // retrieve and return new StockQuote

}

[WebMethod(CacheDuration=60)]

public List<StockQuote> GetQuotes(List<string> symbol)

{

... // retrieve and return List<StockQuote>

}

}

The use of [WebMethod] here is not affecting the service contract. It's simply configuring local processing behavior. When using [WebMethod] in this context, you can only use these behavioral properties and must avoid anything that could modify the service contract.

Q How can I host an ASMX 2.0 class in a custom application (without using IIS)?

Q How can I host an ASMX 2.0 class in a custom application (without using IIS)?

A In my column in the December 2004 issue of *MSDN*®*Magazine* ([Service Station: Run ASMX Without IIS](http://msdn.microsoft.com/msdnmag/issues/04/12/ServiceStation/)), I showed how to take advantage of HTTP.sys and the new HttpListener class in the .NET Framework 2.0 to host the ASP.NET pipeline in your own application. This was a non-trivial task but the only way to reuse your ASMX classes outside of IIS in your own applications. WSE 3.0 changes all of this.

A In my column in the December 2004 issue of *MSDN*®*Magazine* ([Service Station: Run ASMX Without IIS](http://msdn.microsoft.com/msdnmag/issues/04/12/ServiceStation/)), I showed how to take advantage of HTTP.sys and the new HttpListener class in the .NET Framework 2.0 to host the ASP.NET pipeline in your own application. This was a non-trivial task but the only way to reuse your ASMX classes outside of IIS in your own applications. WSE 3.0 changes all of this.

WSE 3.0 merges the new and improved ASMX 2.0 programming model with its messaging layer and hosting model. This makes it possible to host ASMX endpoints within your own applications over any WSE-supported transport such as TCP. This means I can now host the ASMX code shown in **Figure 1** in a Windows service, a Windows Forms application, or even a console application. The code is simple—you just call SoapReceivers.Add and supply your ASMX class, as illustrated here:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl07_code');" \o "Copy Code)

class Program {

static void Main(string[] args) {

Uri uri = new Uri("soap.tcp://localhost:9393/quoteservice");

SoapReceivers.Add(uri, typeof(QuoteService));

Console.WriteLine("Listening...");

Console.ReadLine();

}

}

This is very similar to the WCF model where you use ServiceHost to host service types in any host environment. However, before you get too excited, I should point out that you cannot supply an HTTP address to SoapReceivers.Add like you can with ServiceHost in WCF. SoapReceivers doesn't provide direct integration with HTTP.sys, which is what would be required in this case. So to host your ASMX services over HTTP, you must revert to an <httpHandler> mapping or write the HTTP.sys integration code yourself.

Q What features does WCF provide for developers that ASMX 2.0 and WSE 3.0 don't?

Q What features does WCF provide for developers that ASMX 2.0 and WSE 3.0 don't?

A ASMX 2.0 and WSE 3.0 provide many key features that some developers mistakenly believe are unique to WCF. For example, both stacks provide a similar attribute-based programming model where you can author service contracts on .NET interface definitions (see the first question and answer in this column). Both stacks provide transport-neutral SOAP implementations that ship with multiple transport channels and custom transport hooks. Both stacks allow for hosting services in any Windows-based application (through SoapReceivers in WSE or ServiceHost in WCF). Both stacks provide multiple message encodings, including the two most widely supported: XML 1.0 and Message Transmission Optimization Mechanism (MTOM). And both stacks provide support for message-based security, configurable via simple configuration elements (turnkey security profiles). The table in **Figure 4** summarizes how the stacks compare.

A ASMX 2.0 and WSE 3.0 provide many key features that some developers mistakenly believe are unique to WCF. For example, both stacks provide a similar attribute-based programming model where you can author service contracts on .NET interface definitions (see the first question and answer in this column). Both stacks provide transport-neutral SOAP implementations that ship with multiple transport channels and custom transport hooks. Both stacks allow for hosting services in any Windows-based application (through SoapReceivers in WSE or ServiceHost in WCF). Both stacks provide multiple message encodings, including the two most widely supported: XML 1.0 and Message Transmission Optimization Mechanism (MTOM). And both stacks provide support for message-based security, configurable via simple configuration elements (turnkey security profiles). The table in **Figure 4** summarizes how the stacks compare.

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifFigure 4 ASMX 2.0 Plus WSE 3.0 Compared to WCF

|  |  |  |
| --- | --- | --- |
| **Feature** | **ASMX 2 .0 plus WSE 3. 0** | **WCF** |
| Hosting | IIS/ASP.NET (.asmx) SoapReceivers | IIS/ASP.NET (.svc) ServiceHost<T> WAS |
|  |  |  |
| Programming Model | [WebService], [WebMethod], and so on (supports interfaces, generics, and the like) | [ServiceContract], [OperationContract], and so on (supports interfaces, generics, and so on) |
| Message Exchange Patterns (MEP) | One-way Request-response Custom (using WSE API) | One-way Request-response First/last-operation Duplex Custom |
| XML Serialization | System.Xml .Serialization | System.Runtime .Serialization System.Xml.Serialization (you can choose) |
| Encodings | XML 1.0 MTOM DIME Custom | XML 1.0 MTOM Binary Custom |
| Transports | HTTP TCP Custom | HTTP TCP Named pipes MSMQ P2P Custom |
| Protocols | Security | Security Reliable messaging Transactions |
| Behaviors (enabled via attributes or configuration | Local DTC transactions HTTP buffering HTTP caching HTTP sessions Custom (via SoapExtensions, WSE filters) | Concurrency Instancing Throttling Thread-binding Exception handling and faults Impersonation Session management Transaction behaviors Custom (via behavior types) |

The main area where the stacks differ is in their support for the various WS-\* specifications. WSE 3.0 only supports the security framework while WCF supports the security, reliable messaging, and transaction frameworks. And as the table illustrates, WCF also provides more local processing behaviors and customization hooks. Virtually every layer in the WCF object model is extensible via code, attributes, or configuration.

Ultimately, WCF offers a more complete development framework that has been designed from the ground up around the various Web services protocols. Still, you can have a very similar experience today if you use ASMX 2.0 plus WSE 3.0. And the outlook for future migration looks promising.

Q Will it be difficult to port ASMX code to Windows Communication Foundation?

Q Will it be difficult to port ASMX code to Windows Communication Foundation?

A Not unless you consider changing attribute names difficult. Moving ASMX code forward to WCF is a mechanical process, one which you could even automate to some degree if desired. This process entails converting the various System.Web.Services and System.Web.Services.Protocols attributes to the equivalent System.ServiceModel attributes. **Figure 5** shows an example of how the ASMX code in **Figure 1** would port to WCF.

A Not unless you consider changing attribute names difficult. Moving ASMX code forward to WCF is a mechanical process, one which you could even automate to some degree if desired. This process entails converting the various System.Web.Services and System.Web.Services.Protocols attributes to the equivalent System.ServiceModel attributes. **Figure 5** shows an example of how the ASMX code in **Figure 1** would port to WCF.

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifFigure 5 Porting the QuoteService to WCF

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl26_ctl00_ctl00_code');" \o "Copy Code)

using System.ServiceModel;

[ServiceContract(

Name = "QuoteService",

Namespace = "http://example.org/stocks",

FormatMode = ContractFormatMode.XmlSerializer)]

public interface IQuoteService

{

[OperationContract]

StockQuote GetQuote(string symbol);

[OperationContract]

List<StockQuote> GetQuotes(List<string> symbols);

}

public class QuoteService : IQuoteService

{

public StockQuote GetQuote(string symbol)

{

... // retrieve and return new StockQuote

}

}

Both frameworks support annotating either the class definition or a separate interface definition, so additional refactoring isn't necessary, but may be desired. And by using FormatMode=ContractFormatMode.XmlSerializer, you don't have to port all of your message types over to the new [DataContract] model if you don't want to. Microsoft will publish more detailed migration details as WCF nears release.

Q When should I use WSE with ASMX and when should I use ASMX alone? I'd like to be clearer on this issue.

Q When should I use WSE with ASMX and when should I use ASMX alone? I'd like to be clearer on this issue.

A ASMX 2.0 provides support for the WS-I Basic Profile 1.1 and SOAP 1.2. This means support for XML 1.0, XML Schema Definition (XSD), Web Services Description Language (WSDL), SOAP 1.1 and SOAP 1.2, and basic profile conformance validation at compile time. WSE 3.0 picks up where ASMX 2.0 leaves off by providing support for some of the more advanced WS-\* protocols. Today, you use WSE 3.0 when you need to enhance your ASMX services with one or more of the following features:

* Message-base security (WS-Security)
* Efficient binary data transfer (MTOM)
* Alternative hosting environments
* Custom declarative policy pipeline

A ASMX 2.0 provides support for the WS-I Basic Profile 1.1 and SOAP 1.2. This means support for XML 1.0, XML Schema Definition (XSD), Web Services Description Language (WSDL), SOAP 1.1 and SOAP 1.2, and basic profile conformance validation at compile time. WSE 3.0 picks up where ASMX 2.0 leaves off by providing support for some of the more advanced WS-\* protocols. Today, you use WSE 3.0 when you need to enhance your ASMX services with one or more of the following features:

* Message-base security (WS-Security)
* Efficient binary data transfer (MTOM)
* Alternative hosting environments
* Custom declarative policy pipeline

Message-based security and MTOM are the number one and number two reasons people turn on WSE in their ASMX projects, and the good news is that you can do so without modifying the service logic. WSE 3.0 taps into ASMX through its provided extensibility points. This makes it possible for you to take advantage of various WSE 3.0 features without having to learn a new programming model—all you have to do is run the WSE Settings Tool and select the options you want.

The other reason to enable WSE 3.0 in your ASMX projects is to take advantage of its policy pipeline extensibility model. WSE makes it possible to inject filters that perform pre- and post-processing on the SOAP messages entering and leaving the endpoint. This extensibility point is easier to work with than the ASMX SoapExtension framework as it provides the ability to create custom declarative policies that can be combined with the existing WSE-supplied declarative security policies. You may decide to "turn on" WSE just to have this extensibility option, or at the very least to use the improved SOAP message diagnostic tracing feature.

With WSE 2.0, most developers turned away from ASMX when they needed to do custom hosting. Now that WSE 3.0 makes it possible to do custom hosting of ASMX types, this is no longer the case. The main scenario left that would lead you towards using the WSE messaging API standalone is the need to implement alternative message exchange patterns, such as duplex communications, multicast, or pub/sub scenarios. The messaging classes, SoapSender and SoapReceiver, make it possible to implement any type of advanced messaging pattern.

Q When I enable WSE 3.0 in my Web site, I no longer see the WSE SoapExtension in the config file. Instead I see a SoapServerProtocolFactory. What is that?

Q When I enable WSE 3.0 in my Web site, I no longer see the WSE SoapExtension in the config file. Instead I see a SoapServerProtocolFactory. What is that?

A By default, ASMX endpoints can be invoked using a few different server protocols, including HttpSoap, HttpSoap12, HttpGet, HttpPost, HttpPostLocalhost, and Documentation. You can enable or disable any of these protocols in Web.config. The default ASMX configuration in the .NET Framework 2.0 is shown here:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl11_code');" \o "Copy Code)

<webServices>

<protocols>

<clear />

<add name="HttpSoap12" />

<add name="HttpSoap" />

<add name="HttpPostLocalhost" />

<add name="Documentation" />

</protocols>

...

</webServices>

A By default, ASMX endpoints can be invoked using a few different server protocols, including HttpSoap, HttpSoap12, HttpGet, HttpPost, HttpPostLocalhost, and Documentation. You can enable or disable any of these protocols in Web.config. The default ASMX configuration in the .NET Framework 2.0 is shown here:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl12_code');" \o "Copy Code)

<webServices>

<protocols>

<clear />

<add name="HttpSoap12" />

<add name="HttpSoap" />

<add name="HttpPostLocalhost" />

<add name="Documentation" />

</protocols>

...

</webServices>

Each of these protocols expects a different type of HTTP message and implements a different ASMX invocation protocol. For example, in the case of HttpSoap, the incoming HTTP payload will always contain a <soap:Envelope> whereas with HttpGet the data comes across in the query string and the HTTP payload is empty. The former returns a SOAP envelope in the response, while the latter just returns plain XML.

In order to deal with these differences, the ASMX handler (System.Web.Services.Protocols.WebServiceHandler) uses what's called a ServerProtocol class to process the incoming message. There are several classes that derive from ServerProtocol, including SoapServerProtocol and HttpGetServerProtocol. These classes map to the different protocols that you see in the configuration file. The ASMX handler inspects the incoming HTTP message and chooses the appropriate ServerProtocol type. From that point on, the ServerProtocol instance manages the process of invoking the operation. The SoapExtension framework is one of the things it manages during the invocation process.

The SoapExtension framework is an extensibility point for developers to inject code between the ASMX handler and the target WebMethod. It essentially provides the mechanism for transforming incoming and outgoing message streams. When you need to modify messages in a SoapExtension, you must provide a new stream that it will hand to the next extension in the chain (this is wired up with SoapExtension.ChainStream in the beginning). Hence, a SoapExtension receives a stream, parses it, does its work, and writes to a stream that the next SoapExtension will read from, and this process repeats itself.

WSE 2.0 was implemented entirely as a SoapExtension. One of the problems with implementing WSE at this level is that it led to a great deal of serialization/deserialization overhead at the stream level. Moving to WSE 3.0, the WSE team wanted to optimize their SOAP processing model by moving up an abstraction layer and writing their own SoapServerProtocol implementation. So they asked the ASMX team if they would make it possible to replace the default SoapServerProtocol via configuration in ASMX 2.0. The ASMX developers provided a new configuration element called <soapServerProtocolFactory> that anyone can use to provide their own implementation (although this is not encouraged). The ASMX handler reads this type and uses it to create instances of the corresponding SoapServerProtocol type to process new requests. WSE 3.0 uses this hook and replaces the default SoapServerProtocol class with Microsoft.Web.Services3.WseProtocol.

WseProtocol manages both the SoapExtension mechanism and the WSE pipeline within the same class, thereby reducing some stream friction. WSE 3.0 still uses a SoapExtension (called WseProtocolExtension) but it never shows up in the configuration file because WseProtocol injects it dynamically at startup (to ensure it's always there). The WseProtocolExtension is responsible only for recognizing and parsing MTOM and XML 1.0 streams; everything else is done by the WseProtocol implementation.

So to make a long story short, the <soapServerProtocolFactory> is how WSE 3.0 wires itself into ASMX 2.0. WSE 3.0 still uses a SoapExtension, it's just not where it decided to do the bulk of its work in this version.

Q When hosting ASMX classes in my own app, what happens to HttpContext.Current? What about SoapExtensions?

Q When hosting ASMX classes in my own app, what happens to HttpContext.Current? What about SoapExtensions?

A Unfortunately, the ASMX design is tightly coupled with ASP.NET. For example, when Visual Studio® creates new ASMX classes, it derives them from System.Web.Services.WebService by default. This exposes the various System.Web.HttpContext properties through properties on the new ASMX class, thereby encouraging their use throughout WebMethods. Some developers are also accustomed to explicitly using the HTTP context via HttpContext.Current.

A Unfortunately, the ASMX design is tightly coupled with ASP.NET. For example, when Visual Studio® creates new ASMX classes, it derives them from System.Web.Services.WebService by default. This exposes the various System.Web.HttpContext properties through properties on the new ASMX class, thereby encouraging their use throughout WebMethods. Some developers are also accustomed to explicitly using the HTTP context via HttpContext.Current.

When you host an ASMX class using SoapReceivers in WSE 3.0, this type of code no longer works. In this case, your code is executing outside of IIS and the ASP.NET HTTP pipeline. HttpContext.Current will always be null, so this may require you to modify your existing WebMethod logic. In general, you'll want to avoid depending on anything in the HTTP pipeline throughout your WebMethod code if you plan to reuse that type across hosting environments outside of ASP.NET.

Also, when using SoapReceivers as a host, you can no longer depend on HttpModules or SoapExtensions to do the work for you. These intermediaries are no longer in play since you're now executing outside of the ASMX handler.

Q I can't find the filters tab in the WSE 3.0 settings tool. How can I access it?

Q I can't find the filters tab in the WSE 3.0 settings tool. How can I access it?

A The Filters tab is no longer in the WSE 3.0 Settings Tool because the <filters> element has been removed from the WSE 3.0 configuration section. In WSE 2.0, you used the <filters> element to define how you wanted the pipeline to be constructed with filters. One of the filters included by default was the policy filter. The policy filter looked at another configuration file (the policy file) to determine how to process incoming and outgoing messages according to its assertions (which were similar to filters).

A The Filters tab is no longer in the WSE 3.0 Settings Tool because the <filters> element has been removed from the WSE 3.0 configuration section. In WSE 2.0, you used the <filters> element to define how you wanted the pipeline to be constructed with filters. One of the filters included by default was the policy filter. The policy filter looked at another configuration file (the policy file) to determine how to process incoming and outgoing messages according to its assertions (which were similar to filters).

In WSE 3.0, these two concepts have been unified. So now policies consist of assertions and each assertion injects WSE filters into the pipeline. You place assertions in a policy to control how the filter pipeline is built and organized at run time. So if you want to inject a custom filter into the pipeline, you write the filter class, write a policy assertion class that uses the new filter, use the assertion class in a policy, and apply the policy to a service. In WSE 3.0, it's the policy framework that drives the pipeline so there is no longer a need for the <filters> section.

Q What transports are supported by WSE 3.0? Is it hard to write a custom transport?

Q What transports are supported by WSE 3.0? Is it hard to write a custom transport?

A WSE 3.0 supports HTTP and TCP. WSE provides a custom transport framework to allow third parties to plug in their own transport implementations. Since the release of WSE 2.0, several developers have provided sample WSE transports to illustrate how this can be done. There is one available for Microsoft Message Queue Server (MSMQ) (see [SoapMSMQ Transport](http://www.codeproject.com/cs/webservices/SoapMSMQ.asp)), SMTP (see [soap.smtp://](http://hyperthink.net/blog/CommentView,guid,d337a6f5-a0c8-45b8-920e-132391eedc31.aspx)), and UDP (see [Soap.Udp 0.1](http://www.dynamic-cast.com/mt-archives/000056.html)). All of these transports were written for WSE 2.0, but they port over to WSE 3.0 without many issues.

A WSE 3.0 supports HTTP and TCP. WSE provides a custom transport framework to allow third parties to plug in their own transport implementations. Since the release of WSE 2.0, several developers have provided sample WSE transports to illustrate how this can be done. There is one available for Microsoft Message Queue Server (MSMQ) (see [SoapMSMQ Transport](http://www.codeproject.com/cs/webservices/SoapMSMQ.asp)), SMTP (see [soap.smtp://](http://hyperthink.net/blog/CommentView,guid,d337a6f5-a0c8-45b8-920e-132391eedc31.aspx)), and UDP (see [Soap.Udp 0.1](http://www.dynamic-cast.com/mt-archives/000056.html)). All of these transports were written for WSE 2.0, but they port over to WSE 3.0 without many issues.

This code provides an example of using the SMTP transport to host a service in a console application:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl13_code');" \o "Copy Code)

SoapReceivers.Add(new Uri("soap.smtp://test@skonnard.com"),

typeof(ExpenseReportService));

Console.ReadLine();

Then on the client side you simply need to specify the SMTP address before invoking the method, as shown here:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl14_code');" \o "Copy Code)

ExpenseReportServiceWse svc = new ExpenseReportServiceWse();

svc.Url = "soap.smtp://test@skonnard.com";

svc.Submit(report);

Now the client will send messages to the test@skonnard.com e-mail address, and the receiver will pick them up from that same address. One of the interesting things about this example is that the client and receiver no longer have to be running at the same time. Notice how the developer experience is unaffected when using a custom transport. The only thing you have to do is tell WSE how to process the "soap.smtp" protocol scheme by adding a <transport> mapping to the WSE configuration section, as illustrated here:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl15_code');" \o "Copy Code)

<microsoft.web.services3>

<messaging>

<transports>

<add scheme="soap.smtp" type="SoapSmtpTransport,SoapSmtp"/>

</transports>

</messaging>

...

</microsoft.web.services3>

This element maps a protocol scheme to an ISoapTransport-derived class. When you call SoapReceivers.Add, it looks up the type that maps to the specified protocol scheme from the configuration section. Then it instantiates the specified type and uses the ISoapTransport methods to produce the input and output communication channels. The same thing happens on the client side. These channels hide all transport implementation details from the rest of the API.

Hence, writing a custom WSE transport entails implementing three classes: one that models the transport (ISoapTransport) and two for modeling input/output channels (ISoapInputChannel and ISoapOutputChannel). The transport serves up channels and manages resources shared across all channels, while the channels focus on sending/receiving messages via the transport. You're dealing with low-level transport details here, so I wouldn't say it's easy. The best way to get your head around how this works is to walk through the existing WSE transport samples I mentioned previously.

Q Can I use "Add Web Reference" to generate a proxy for a custom hosted WSE 3.0 service?

Q Can I use "Add Web Reference" to generate a proxy for a custom hosted WSE 3.0 service?

A If you have a service that is hosted in a Windows service using TCP for communications, the only way to retrieve the WSDL is by using the WseWsdl3.exe tool. This tool sends a SOAP message to the service requesting its metadata and generates the proxy from the WSDL returned in the SOAP response message.

A If you have a service that is hosted in a Windows service using TCP for communications, the only way to retrieve the WSDL is by using the WseWsdl3.exe tool. This tool sends a SOAP message to the service requesting its metadata and generates the proxy from the WSDL returned in the SOAP response message.

However, if you have the same service also hosted via ASP.NET (HTTP), you can use the typical "Add Web Reference" command in Visual Studio (or wsdl.exe) to generate a proxy (make sure you have WSE enabled in the client application before doing this so you get a WSE proxy). The generated WSE proxy can be used with any WSE transport, not just with HTTP. Simply change the address of the proxy to the TCP address using either the Url or Destination property and it will work just like a proxy generated via WseWsdl3.exe.

Q How should I organize my solution to share ASMX types across different hosts?

Q How should I organize my solution to share ASMX types across different hosts?

A When you need to share ASMX types across different hosts, you should put your ASMX classes in a shared class library. This seems kind of odd at first, since your ASMX code has probably always existed in Web projects, but there's no reason they can't be factored out into a library. I would even go a step further and organize all of my message serialization classes into a separate class library of their own, as you may want to share these types with clients and code generators (such as using a schema importer extension).

A When you need to share ASMX types across different hosts, you should put your ASMX classes in a shared class library. This seems kind of odd at first, since your ASMX code has probably always existed in Web projects, but there's no reason they can't be factored out into a library. I would even go a step further and organize all of my message serialization classes into a separate class library of their own, as you may want to share these types with clients and code generators (such as using a schema importer extension).

With this configuration, simply add a reference to your class libraries in each host project, including the Web site host. Then your Web site host will no longer need the service code in App\_Code. Instead your .asmx files will refer to the shared service classes by name. And your custom hosts can supply the same service type when calling SoapReceivers.Add.

Q My team developed a common library of XSD serialization classes that we want all of our developers to use when implementing or consuming our services. When they run "xsd.exe /c" or "wsdl.exe", the code generator should use our types instead of generating new ones. Is this possible?

Q My team developed a common library of XSD serialization classes that we want all of our developers to use when implementing or consuming our services. When they run "xsd.exe /c" or "wsdl.exe", the code generator should use our types instead of generating new ones. Is this possible?

A Yes, by implementing what's called a schema importer extension—a new extensibility point offered by the .NET Framework 2.0. Here's the gist of it. You write a class that derives from SchemaImporterExtension and override ImportSchemaType. In your implementation of this method, you define how the incoming XSD types should map to your existing .NET types. **Figure 6** shows an example.

A Yes, by implementing what's called a schema importer extension—a new extensibility point offered by the .NET Framework 2.0. Here's the gist of it. You write a class that derives from SchemaImporterExtension and override ImportSchemaType. In your implementation of this method, you define how the incoming XSD types should map to your existing .NET types. **Figure 6** shows an example.

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifFigure 6 Implementing a SchemaImporterExtension

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl32_ctl00_ctl00_code');" \o "Copy Code)

class CommonTypesSIE : SchemaImporterExtension

{

public override string ImportSchemaType(

string name, string ns, XmlSchemaObject context,

XmlSchemas schemas, XmlSchemaImporter importer,

CodeCompileUnit compileUnit,

CodeNamespace mainNamespace,

CodeGenerationOptions options,

CodeDomProvider codeProvider)

{

if (name.Equals("StockQuote") &&

ns.Equals("http://example.org/stocks"))

{

mainNamespace.Imports.Add(

new CodeNamespaceImport("CommonTypes"));

return "CommonTypes.StockQuote";

}

return null;

}

}

Then you configure your schema importer extension to be used by xsd.exe or wsdl.exe. You can do this in the <schemaImporterExtensions> element of the <system.xml.serialization> configuration section. Or you can do it by supplying the list of schema importer extensions to use when running xsd.exe or wsdl.exe (via the /parameters option).

Q Can you tell me what the file format is for /parameters in xsd.exe and wsdl.exe?

Q Can you tell me what the file format is for /parameters in xsd.exe and wsdl.exe?

A Both xsd.exe and wsdl.exe introduce a new switch, /parameters, for packaging up all of the command-line options you want to use into a single XML file. The format of the XML file is defined by an XML Schema definition that is available in the download for this column. The file must contain a top-level <wsdlParameters> element, which contains all of the options. **Figure 7** shows an example.

A Both xsd.exe and wsdl.exe introduce a new switch, /parameters, for packaging up all of the command-line options you want to use into a single XML file. The format of the XML file is defined by an XML Schema definition that is available in the download for this column. The file must contain a top-level <wsdlParameters> element, which contains all of the options. **Figure 7** shows an example.

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifFigure 7 Options.xml

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl33_ctl00_ctl00_code');" \o "Copy Code)

<!-- options.xml -->

<wsdlParameters xmlns='http://microsoft.com/webReference/'>

<language>c#</language>

<sharetypes>true</sharetypes>

<webReferenceOptions

xmlns="http://microsoft.com/webReference/">

<codeGenerationOptions>newAsync

</codeGenerationOptions>

<verbose>true</verbose>

<schemaImporterExtensions>

<type>CommonTypesSIE, CommonTypesSIE</type>

</schemaImporterExtensions>

</webReferenceOptions>

</wsdlParameters>

You would then pass that file to wsdl.exe, as shown here:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl20_code');" \o "Copy Code)

C:\demos> wsdl.exe /parameters:options.xml

http://localhost/stocks/quoteservice.asmx

This approach helps ensure a consistent set of options each time one of these code-generation tools is used. It's interesting to note that some of the xsd.exe and wsdl.exe features are only accessible via this mechanism.

Q I've heard speculation that version 3.0 might be the last WSE release. Is this true?

Q I've heard speculation that version 3.0 might be the last WSE release. Is this true?

A After WCF ships, WSE will most likely stop evolving with new features and move into support mode (bug fixes, service packs, and so forth). WCF is positioned as the new .NET Web services platform, which completely replaces the need for the functionality WSE has provided up to this point. And since WCF is getting closer to its release date, it's very likely that WSE 3.0 will be the last version to ship. The WSE development team has already been restructured within the WCF team to consolidate the efforts of the developers across the teams.

A After WCF ships, WSE will most likely stop evolving with new features and move into support mode (bug fixes, service packs, and so forth). WCF is positioned as the new .NET Web services platform, which completely replaces the need for the functionality WSE has provided up to this point. And since WCF is getting closer to its release date, it's very likely that WSE 3.0 will be the last version to ship. The WSE development team has already been restructured within the WCF team to consolidate the efforts of the developers across the teams.

However, it's still likely that something like WSE will be used to extend and "enhance" WCF after it hits the streets, although who knows what they will call it. Such a vehicle could be used to incorporate new protocols and behaviors as they become available using the various WCF extensibility points.