WEB SERVICES

Extend the ASP.NET WebMethod Framework by Adding XML Schema Validation

Aaron Skonnard and Dan Sullivan

This article assumes you're familiar with XML, ASP.NET, and Web Services

Level of Difficulty 1 2 3

Download the code for this article: XMLSchemaValidation.exe (208KB)

SUMMARY WebMethods make the development of XML Web Services easier by encapsulating a good deal of functionality, but there is still a lot of underlying XML processing that you have to be responsible for. For example, WebMethods do not validate messages against the implied schema. Because they are not validated, the response that's returned can result in unintended consequences. To address this, the authors extend the WebMethod framework by adding XML Schema validation through a custom SoapExtension class.

SP.NET provides an infrastructure for mapping methods on ordinary classes, written in C# or Visual Basic® .NET, into Web Service operations (WebMethods) that support XML 1.0, XML Schema, SOAP, and Web Services Description Language (WSDL). This gives developers a familiar programming model for building Web Services without having to write any code to produce or process the XML messages used on the wire. Although WebMethods can greatly increase productivity, you must still pay attention to certain aspects of the underlying XML processing. WebMethods do not fully validate the messages they process against the implied schema. As a result, it's possible for a given WebMethod invocation to produce a false positive (a successful response with an incorrect or unintended result) when an exception would have been expected. Also, they don't offer built-in support for declaring additional message-level constraints on business rules. The lack of validation support complicates the error handling that must be built into each WebMethod.

In this article, we'll show you how to compensate for this issue by using the WebMethod extensibility hooks built into ASP.NET. The full sample is available for download from the link at the top of this article.

WebMethod Functionality

ASP.NET makes it possible to map traditional methods to Web Service operations through the System.Web.Services.WebMethod attribute. For example, the following class exposes the CalcArea method as a Web Service:

```
using System.Web.Services;

[WebService(Namespace="http://example.org/geometry/")]
public class Geometry
{
    [WebMethod]
    public double CalcArea(double length, double width)
    {
        return length * width;
    }
}
```

To invoke this WebMethod, you must provide an .asmx file that refers to the Geometry class, as shown here:

```
<!-- geometry.asmx -->
<%@ WebService class="Geometry" language="C#" %>
```

Although it's more common to keep the source code in a separate file, it's also possible to include the class definition within the .asmx file itself. In this case, the class is just-in-time compiled the first time the .asmx file is requested.

When a request comes in for geometry.asmx, the WebServiceHandler class (.asmx HTTP handler) translates the SOAP request into a traditional method call. For CalcArea, it expects the following SOAP message:

```
<soap:Envelope
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <CalcArea xmlns="http://example.org/geometry/">
      <length>2.3</length>
      <width>1.2</width>
    </CalcArea>
  </soap:Body>
</soap:Envelope>
```

This request produces the following SOAP response:

```
<soap:Envelope
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <CalcAreaResponse xmlns="http://example.org/geometry/">
      <CalcAreaResult>2.76</CalcAreaResult>
    </CalcAreaResponse>
  </soap:Body>
</soap:Envelope>
```

In a nutshell, the WebMethod infrastructure takes care of dispatching incoming SOAP requests to the appropriate class and method (based on either the request element's name or the SOAPAction header, configurable through SoapDocumentService.RoutingStyle) and serializing/deserializing the XML message into the common language runtime (CLR) objects, including the translation of .NET-style exceptions into SOAP Fault elements.

In addition, the infrastructure automatically generates a WSDL definition that describes the WebMethods using XML Schema types and WSDL constructs (http://localhost/geometry/geometry.asmx?wsdl). The System.Web.Services namespace contains a slew of additional attributes used to control dispatching as well as the XML Schema and WSDL mapping details.

What WebMethods Miss

If you don't take advantage of WebMethod, you have to write an HTTP handler that processes the SOAP request, extracts the width and length text representations, converts the text to a numeric value type, calculates the area, and then constructs the SOAP response. There is plenty of support in the Microsoft® .NET Framework for writing such a handler, but by using WebMethod all you have to do is calculate the area.

Although this is a significant productivity gain that most developers will want to take advantage of, it doesn't excuse you from having to deal with (or at least thinking about) certain aspects of the underlying message processing. Typically, you'll need more control over what your WebMethods will accept than what is provided by the default behavior. Consider this XML Schema complex type definition generated for the CalcArea WebMethod:

```
<s:element name="CalcArea">
  <s:complexType>
    <s:sequence>
      <s:element minOccurs="1" maxOccurs="1"
                 name="length" type="s:double" />
      <s:element minOccurs="1" maxOccurs="1"
                 name="width" type="s:double" />
    </s:sequence>
  </s:complexType>
</s:element>
```

The complex type definition states that the CalcArea element must contain a sequence of two required elements, length followed by width, both of which must be of type double.

Now consider the following unexpected SOAP request:

```
<soap:Envelope
    xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <CalcArea xmlns="http://example.org/geometry/">
        <Length>2.3</Length>
        <Width>1.2</Width>
    </CalcArea>
  </soap:Body>
</soap:Envelope>
```

At first glance this message looks fine but there's actually a subtle problem with the case of the length and width fields. However, instead of producing a SOAP fault, the WebMethod produces the following successful response:

```
<soap:Envelope
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
```

```
  <soap:Body>
    <CalcAreaResponse xmlns="http://example.org/geometry/">
      <CalcAreaResult>0</CalcAreaResult>
    </CalcAreaResponse>
  </soap:Body>
</soap:Envelope>
```

In fact, the following request, which is missing both the length and width elements, also produces a successful result with a value of 0:

```
<soap:Envelope
    xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <CalcArea xmlns="http://example.org/geometry/"/>
  </soap:Body>
</soap:Envelope>
```

Or consider this request, where the client clearly has the wrong idea about how the WebMethod works:

```
<soap:Envelope
   xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <CalcArea xmlns="http://example.org/geometry/">
        <Length>2.3</Length>
        <Width>1.2</Width>
        <Length>5.0</Length>
        <Width>3.4</Width>
        <Length>9.2</Length>
        <Width>2.8</Width>
    </CalcArea>
  </soap:Body>
</soap:Envelope>
```

In this case the WebMethod returns a successful response with the value of 2.76, as we illustrated in the very first example.

These situations, and many others, cause WebMethods to produce false positives. For certain applications it may not be a big deal, but for others it could be catastrophic and is therefore completely unacceptable. For these applications, some form of validation is required, regardless of the implementation technique that you use.

The reason for this behavior has to do with XmlSerializer, the underlying plumbing that takes care of object deserialization. XmlSerializer is very forgiving by design. It happily ignores XML nodes that it didn't expect and will use default CLR values for expected but missing XML nodes. It doesn't perform XML Schema validation and the consequence is that details like structure, ordering, occurrence constraints, or simple type restrictions are not enforced during deserialization. This issue is described in more detail in the November 2002 installment of The XML Files.

It's possible to force XmlSerializer to perform validation during deserialization by supplying an XmlValidatingReader object to the call to deserialize, as shown here:

```
XmlReader xr = new XmlTextReader(fileName);
XmlValidatingReader vr = new XmlValidatingReader(xr);
vr.Schemas.Add(schemaNamespace, schemaLocation);
object validatedObject = ser.Deserialize(vr);
```

However, within the WebMethod infrastructure, there are two problems. First, it's not possible to get your hands on the underlying XmlSerializer, and second, the schema file doesn't exist as part of the project (the schema is derived from the CLR attributes decorating the method, and it's only generated when clients issue a GET request with the ?wsdl query string).

WebMethod Extensions

Ideally, it would be possible to enable WebMethod validation by "flipping a switch," but unfortunately it's not. In some cases, it's not even possible to detect error conditions, like missing information, from within a WebMethod implementation. Therefore, you must take advantage of the provided extensibility points in order to add validation support.

There are two general WebMethod extension mechanisms: HTTP modules and SoapExtension classes. An HTTP module is simply a class derived from IHttpModule and configured to run on a particular virtual root (using web.config) or on the entire machine (using machine.config). HTTP modules can be used to extend any type of HTTP application, not just WebMethods. HTTP modules make it possible to perform pre- and post-processing on the HTTP messages traveling through the pipeline (see **Figure 1**).

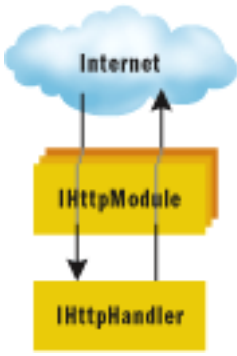

Figure 1 HTTP Modules

HTTP modules force you to work completely at the message level. Hence, it's a fairly low-level approach that requires you to implement SOAP details manually (like generating SOAP faults). Since HTTP modules are configured to run for an entire virtual root, it's quite difficult to enable functionality or supply extra configuration information on a class-per-class or method-per-method basis.

The other technique for extending WebMethods, managed entirely by the .asmx handler, is commonly referred to as the SoapExtension framework. To use this approach, you derive a new class from SoapExtension and override a few of its members. Then you configure the SoapExtension to run by using a configuration file or a custom attribute. The SoapExtension will be called before and after each WebMethod invocation, shown in **Figure 2**.

The hooks that allow the WebMethods to be extended are quite flexible. We are not going to look at all possible ways of using them, only at how to examine and validate messages associated with a particular method prior to invocation. The hook will validate the incoming message against the corresponding XML Schema. Based on the validation results, the message will either be allowed to proceed to the WebMethod or an appropriate SOAP fault will be returned to the client, describing the error condition. **Figure 3** shows the basic flow.

SoapExtension is the preferred framework for extending WebMethods because it gives you much more configuration control and makes it possible for you to customize extension behavior on a method-per-method basis.

Figure 3 Validating Messages Before Invocation

SoapExtension Configuration

You can configure SoapExtensions to run on an entire virtual root (similar to an HTTP module) or on a method-by-method basis. To configure a SoapExtension for an entire virtual root, you simply add an entry to web.config. Here's a sample web.config file that registers a custom SoapExtension called ValidationExtension:

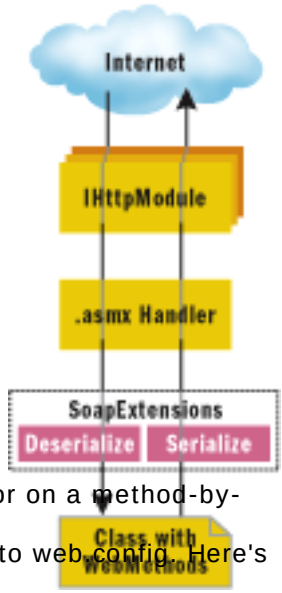

Figure 2 SoapExtensions

```
<configuration>
  <system.web>
    <webServices>
      <soapExtensionTypes>
        <add type="ValidationExtension,
                   DevelopMentor.Web.Services"
             priority="1" group="0" />
      </soapExtensionTypes>
    </webServices>
  </system.web>
</configuration>
```

In order to configure a SoapExtension on a method-by-method basis, you need to write a custom attribute derived from SoapExtensionAttribute (see **Figure 4**). The ExtensionType property returns the type of SoapExtension to use, in this case ValidationExtension. You should take note that this attribute can only be applied to a method, and then only once. This way the compiler will catch most of the places where this attribute might be misapplied. Limiting it to a single instance on a method just makes it easier to implement the extension.

Then you can decorate your WebMethods with this attribute to tell the infrastructure to call your SoapExtension at runtime. Note that if your attribute's name ends in "Attribute", you can leave it off when it's used:

```
[Validation]
[WebMethod]
public double CalcArea(double length, double width)
{
    return length * width;
}
```

You can also design your attributes to take additional configuration information through public fields or properties—you'll see an example of this shortly. Overall, this configuration technique ties in nicely with WebMethods.

SoapExtension Processing

The ValidationExtension class must derive from the SoapExtension base class and override the required members, as shown in **Figure 5**. **Figure 6** provides a helpful summary describing when each of the different SoapExtension methods is called by the infrastructure. However, deciding how to implement them requires a thorough understanding of the execution process.

Each time a WebMethod is invoked, the infrastructure creates a new instance of the SoapExtension class. The fact that a new SoapExtension instance is created on each invocation poses a state management issue. You can't store state in member variables since between calls they go away. You also don't want to do expensive initialization in the constructor for obvious performance reasons. Hence, the SoapExtension framework provides the GetInitializer and Initialize methods for performing one-time initialization.

The first time that any WebMethod in a virtual root is invoked, the infrastructure checks to see if any SoapExtensions have been configured in the applicable configuration files (web.config or machine.config). If so, the infrastructure creates an instance of each configured SoapExtension class and calls the first GetInitializer method shown in **Figure 6**. This is your chance to perform any necessary initialization, wrap the information in an object, and return the object to the infrastructure. The infrastructure stores the returned object and then passes it to Initialize, which is subsequently called on each WebMethod invocation after constructing the SoapExtension class. During this first WebMethod invocation, the infrastructure also inspects each WebMethod to see if it also has any attributes that derive from SoapExtensionAttribute. If it does, it makes an instance of each attribute (with any supplied information) and asks via the ExtensionType property for the SoapExtension-derived class that should be called on each invocation. For each of these classes, the infrastructure creates an instance and calls the second GetInitializer method described in **Figure 6**, passing in the supplied attribute. This is your chance to perform WebMethod-specific initialization. The object you return from GetInitializer is stored by the infrastructure for that particular WebMethod and is passed to Initialize during each future invocation. The entire initialization process is illustrated in **Figure 7**.

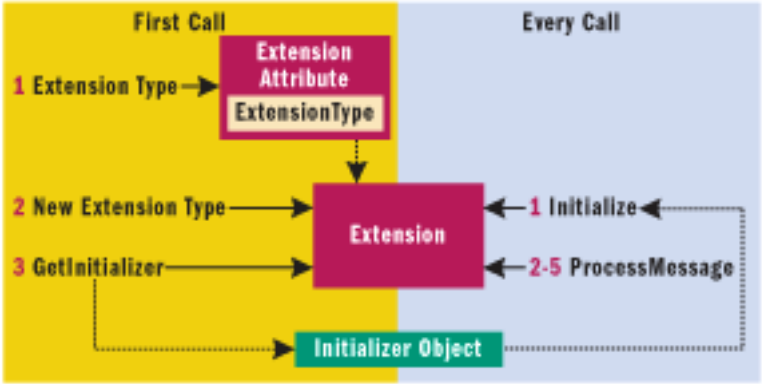

Figure 7 Initialization Process

There are two parts to initialization; one happens only on the first call and the other every time a method is called. The first initialization happens when GetInitializer is called and returns an initializer object which the runtime caches for future use. The second part happens when Initialize is called and also happens whenever a method is called, even the first time. Initialize is given the Initializer object for use by the ProcessMessage function. This allows the

time required to create the Initializer object to be amortized over all the calls made to a method. ProcessMessage is then called four times, one for each message processing state (BeforeDeserialize, AfterSerialize, BeforeSerialize, and AfterSerialize).

Inside Initialize, you typically just want to store the supplied object in a member variable for future use in ProcessMessage, where the real pre- and post-processing occurs. There are four stages in the processing of a WebMethod: BeforeDeserialize, AfterDeserialize, BeforeSerialize, and AfterSerialize (see **Figure 2**). You can write code to execute during any of these stages within ProcessMessage. The WebMethod implementation is invoked between AfterDeserialize and BeforeSerialize.

ValidationExtension Implementation

To implement a validation extension, we want to perform XML Schema validation before the System.Web.Services has done anything with the message. We'll hook our logic into the BeforeDeserialize stage, as illustrated in **Figure 8**. This implementation loads the raw message stream into an XmlTextReader and wraps it with an XmlValidatingReader. After loading the schema files into XmlValidatingReader's schema cache (Schemas property), it simply reads through the stream to perform validation. If XmlValidatingReader encounters a validation problem in the message, it throws an exception containing a description of the problem. The exception is translated into a SOAP Fault response message and returned to the client without invoking the WebMethod. Before returning from ProcessMessage, we also want to reset the stream's position; otherwise the infrastructure won't know how to deserialize the message properly. The only tricky part about ProcessMessage is loading the schemas into XmlValidatingReader.

The Schema Cache

In order to perform validation on the incoming requests, XmlValidatingReader needs access to the schema definitions that describe the messages. The question is, where does our code look for the schema definitions? There are a variety of possible answers. Probably the easiest solution is to load all schema files from a well-known location in the virtual root, like a child xsd directory. Then users of this extension can simply drop the required schema files into the directory and they'll be automatically picked up.

Since it could cost you some performance to load up all the schema files, this is a perfect example of where you'll want to use GetInitializer. Within GetInitializer we need to create an instance of XmlSchemaCollection and fill it up with all of the compiled schemas. We need to do the same thing regardless of which GetInitalizer version is called, so we'll benefit ourselves by creating a helper method, GetInitializerHelper, which both GetInitializer versions delegate to (see **Figure 9**).

Remember that the object returned from GetInitializer is passed into Initialize during each future WebMethod invocation. Our implementation of Initialize caches the XmlSchemaCollection object into a member variable for use within ProcessMessage:

```
public override void Initialize(object initializer)
```

```
{
    _context = (XmlSchemaCollection)initializer;
}
```

Then, going back to our implementation of ProcessMessage, we can associate the stored XmlSchemaCollection object with the XmlValidatingReader before processing the stream:

```
// ProcessMessage method
XmlTextReader tr = new XmlTextReader(message.Stream);
XmlValidatingReader vr = new XmlValidatingReader(tr);
vr.Schemas.Add(_context); // load schema cache
while (vr.Read()) ; // read through stream
```

With this, we should have enough foundation in place to begin using the ValidationExtension.

To experiment with the example described thus far, you'll need to follow these steps:

1. Add a reference in your project to the ValidationExtension assembly (DevelopMentor.Web.Services).
2. Annotate your WebMethods with [Validation].
3. Create an xsd directory within the vroot.
4. Download the SOAP 1.1 schema from http://schemas.xmlsoap.org/soap/envelope, and save it to the xsd directory.
5. Browse to the .asmx endpoint, press Service Description (.asmx?wsdl), and then save the WSDL file to the xsd directory.
6. Extract the schema definition from the saved WSDL file and save it to a new .xsd file (make sure you get all the namespace declarations).

Now, invoking CalcArea with an invalid request produces a SOAP Fault. For example, the invalid request containing uppercase letters (Length/Width) produces the response seen in **Figure 10**. Notice that the SOAP Fault contains precise information about what went wrong during validation. All false positive situations described earlier will now be detected thanks to our schema definition.

A Better Schema Cache

It's easy to get the ValidationExtension to execute by either annotating with [Validation] or adding an entry to web.config. The challenge is setting up the schema cache. To help simplify this process, we decided to add a few features to our extension.

First, we automatically load the SOAP 1.1 schema from the assembly resource file. There is a call to a helper function in GetInitalizerHelper that takes care of this and adds the SOAP 1.1 schema to the XmlSchemaCollection object:

```
LoadSchemaFromResourceFile(sc, "ValidationResources",
   "DevelopMentor.Web.Services", "SOAP1.1");
```

This takes care of the SOAP schema, but there's still the issue of loading up your own schema. Since it's a bit cumbersome to extract the schema definition from the WSDL file before you can use it, we also made it possible to load schemas directly from a complete WSDL file by extending the code that automatically loads .xsd files with the following:

```
string[] wsdlFiles = Directory.GetFiles(
  ctx.Server.MapPath("xsd"), "*.wsdl");
foreach (string wsdlFile in wsdlFiles)
{
    ServiceDescription sd=ServiceDescription.Read(wsdlFile);
    foreach(XmlSchema embeddedXsd in sd.Types.Schemas)
       sc.Add(embeddedXsd);
}
```

It's also possible to just drop the WSDL file into the xsd directory and be ready to go. Also, to give you more flexibility in terms of where you want to store your XML Schema files, we've defined a few additional attributes that can be used in conjunction with the ValidationExtension: ValidationSchemaAttribute and ValidationSchemaCacheAttribute, both of which are used on the class definition. ValidationSchemaAttribute allows you to specify a schema file location (relative to vroot) that you'd like to load into the cache:

```
[WebService(Namespace="http://example.org/geometry/")]
[ValidationSchema("schemas/geo/geo.xsd")]
[ValidationSchema("schemas/global/math.xsd")]
[ValidationSchema("schemas/global/simple.xsd")]
public class Geometry {
    •••
}
```

ValidationSchemaCache allows you to specify additional directories (other than xsd) that will contain a bunch of .xsd or .wsdl files, all of which will be loaded into the cache by GetInitializer. Here's an example of it in action:

```
[WebService(Namespace="http://example.org/geometry/")]
[ValidationSchemaCache("schemas")]
[ValidationSchemaCache("schemas/global")]
public class Geometry {
 •••
}
```

The attribute class definitions are straightforward—they derive from System.Attribute and provide a public property for storing the file/directory location, respectively. GetInitializerHelper then inspects the class (serviceType) to see if they were used and if so, it retrieves the information and loads from the specified locations. Here's the code taken from GetInitializerHelper:

```
// load schemas from user-defined file locations
   atts = serviceType.GetCustomAttributes(
   typeof(ValidationSchemaAttribute), true);
```

```
 foreach(ValidationSchemaAttribute vsa in schemaLocations)
   LoadSchemaFromFile(sc, ctx.Server.MapPath(vsa.Location));

// load schemas from user-defined directories
   object[] atts = serviceType.GetCustomAttributes(
   typeof(ValidationSchemaCacheAttribute), true);
foreach (ValidationSchemaCacheAttribute loc in atts)
    LoadSchemasFromDirectory(sc, loc.RelativeDirectory);
```

This gives the user much more control over where the schema and WSDL files reside in their system.

As a final simplification, we wanted to make it possible for developers to avoid having to manage schema files altogether. The .asmx handler provides support for generating a WSDL document that describes the endpoint. We can leverage this functionality to generate a WSDL document that contains an XML Schema definition that we can use for validation purposes. This makes it possible to turn on validation for a given WebMethod without requiring the developer to manage a schema file by hand. The following method illustrates this process:

```
internal void LoadReflectedSchemas(
    XmlSchemaCollection sc, Type type, string url)
{
    ServiceDescriptionReflector r=
          new ServiceDescriptionReflector();
        r.Reflect(type, url);
        foreach (XmlSchema xsd in r.Schemas)
            sc.Add(xsd);
        foreach (ServiceDescription sd in r.ServiceDescriptions)
            LoadSchemasFromServiceDescriptions(sc, sd);
}
```

You supply the class type for which you'd like to generate WSDL and then iterate through the generated Schema collections. You can call this from either the GetInitializer method by passing serviceType or methodInfo.DeclaringType, respectively. Note that user-defined schemas always take precedence over this technique.

With all this code in place, you can cross off the last four steps listed earlier; the extension will now work. Just add the assembly to your project, annotate with [Validation], and you're finished.

Customizing Schema Definitions

Just enabling XML Schema validation will simplify WebMethod error handling tremendously. It's now possible to constrain the message structure as well as the lexical and value spaces of text values. If you don't want to worry about managing schema files, you're basically stuck with what's produced by the infrastructure. But if you're willing to take control of your schemas (using the techniques I've just described), you can use more advanced schema features. For example, consider the following WebMethod:

```
public void AddEmployee(string name, string id, DateTime
  bday, double salary) { ... }
```

The generated schema for this WebMethod will automatically constrain the values using XML schema's built-in data

types. However, string and double don't precisely describe the expected formats for the id and salary values, respectively. Consider the following XSD simple type definitions:

```
<xs:simpleType name="SSN">
    <xs:restriction base="xs:string">
        <xs:pattern value="\d{3}-\d{2}-\d{4}" />
    </xs:restriction>
</xs:simpleType>
<xs:simpleType name="NewHireSalary">
    <xs:restriction base="xs:double">
        <xs:minExclusive value="0" />
        <xs:maxExclusive value="5000" />
    </xs:restriction>
</xs:simpleType>
```

These definitions precisely describe the data format expected by the operation. The SSN type utilizes a regular expression to constrain the allowed string format while NewHireSalary explicitly defines the allowed value space. To leverage these types during validation, simply modify your schema file to use them, as shown in **Figure 11**. Now if the client provides an id value that's not in Social Security Number format, or a salary that's not between 0 and 5000, the ValidationExtension will catch it and will return a descriptive SOAP Fault.

Business Rules

As we just illustrated, using the ValidationExtension enables you to take full advantage of the XML Schema language. You can control complex structural details, simple type value spaces, uniqueness, substitution mechanisms, and more as the language is quite extensive. XML Schema does, however, have some limits. One of the most apparent limitations is that you can't enforce co-occurrence constraints where the presence or value of a given node affects the constraints on other nodes.

Co-occurrence constraints are typical of most business rules in the real world. Business rules typically specify arbitrary constraints on the data beyond its type. What kind of arbitrary constraints might there be on the data? Consider the CalcArea operation where we're dealing with length and width elements. In the real world, the length is by definition greater than the width.

Although the choice is arbitrary, in a manufacturing process it is important that people have a consistent vocabulary so that boxes fit on belts and through openings, like making sure to put a tray lengthwise on the belt. Let's assume this is the case here. This rule is something that you can't enforce using XML Schema, but you can take care of it in your CalcArea method implementation. By adding a little procedural code, you can check the input parameters before you calculate the area. But since business rules typically come from corporate specifications, it would be much cleaner to declaratively include them in the class definitions.

XPath is an ideal language for defining these additional constraints. It was designed specifically for the purpose of evaluating the content of an XML document; it's easy to use and quite flexible. Furthermore, XPath is implemented for virtually all platforms, languages, and even databases in common use today. Therefore, if we use XPath as the

rule language, rules can be enforced on other platforms using other languages, tools, or even databases. This means that business rules can be pushed anywhere in the chain of processing, even to the client.

Using XPath to make assertions about XML documents is not a new idea. Schematron is a proposed language for doing this in a standard manner (see http://www.ascc.net/xml/schematron). Schematron can be integrated with XML Schema definitions and evaluated during a post-validation step. Daniel Cazzulino has written an open-source implementation of Schematron for the .NET Framework that you can find at http://sourceforge.net/projects/schematron-net.

We decided that it would be useful to build this type of XPath constraint processing right into the ValidationExtension class. To do so, we defined a new attribute class called AssertAttribute that allows you to specify XPath assertions to an individual method or the entire class (applies to all WebMethods). An example of these attributes in use is shown in **Figure 12**.

For more information on how this part of the project was implemented, stay tuned for our next piece where we'll continue describing the XPath assertion functionality as well as other cool topics like extending the WSDL generation process to include the business rule descriptions. However, if you're itching to take a look at these topics now, download the source code from the link at the top of this article and check out the sample.

Conclusion

The WebMethod infrastructure provides a powerful extensibility framework for adding additional user-defined behavior. We were able to add validation support to any WebMethod through the ValidationExtension and ValidationAttribute classes. This technique simplifies WebMethod error handling and fully utilizes the power and flexibility offered by XML Schema.

For related articles see:
The XML Files: A Quick Guide to XML Schema
The XML Files: A Quick Guide to XML Schema-Part 2
For background information see:
Real World XML Web Services: For VB and VB .NET Developers by Yasser Shohoud (Addison-Wesley, 2002)
Essential XML Quick Reference: A Programmer's Reference to XML, XPath, XSLT, XML Schema, SOAP, and More by Aaron Skonnard and Martin Gudgin (Addison-Wesley, 2001)
Building XML Web Services for the Microsoft .NET Platform by Scott Short (Microsoft Press, 2002)

Aaron Skonnard is an instructor and researcher at DevelopMentor, where he develops the XML and Web Service-related curriculum. Aaron coauthored *Essential XML Quick Reference* (Addison-Wesley, 2001) and *Essential XML* (Addison-Wesley, 2000). Reach him at http://staff.develop.com/aarons.

Dan Sullivan is an independent consultant and an instructor and course author at DevelopMentor. He has worked in all facets of computing, from designing processors to writing applications and system software. Dan now focuses on XML and developing Web Services. He can be contacted at dsullivan@net1plus.com.

From the July 2003 issue of MSDN Magazine.
Get it at your local newsstand, or better yet, *subscribe*.

Figure 4 SoapExtension per Method

```
[AttributeUsage(AttributeTargets.Method,
                AllowMultiple=false)]
public class ValidationAttribute : SoapExtensionAttribute
{
    int priority = 0;

    // used by soap extension to get the type
    // of object to be created
    public override System.Type ExtensionType
    {
        get { return typeof(ValidationExtension); }
    }
    public override int Priority
    {
        get { return priority; }
        set { priority = value;    }
    }
}
```

Figure 5 ValidationExtension Class

```
public class ValidationExtension : SoapExtension
{
    public override object GetInitializer(
      System.Type serviceType)
    {
        // TODO
}
    public override object GetInitializer(
      LogicalMethodInfo methodInfo,
      SoapExtensionAttribute attribute)
    {
        // TODO
}
    public override void Initialize(object initializer)
    {
        // TODO
}
    public override void ProcessMessage(SoapMessage message)
    {
        // TODO
}
}
```

Figure 6 SoapExtension Methods

Method	Description
GetInitializer(serviceType)	Called once per vroot (if registered in the configuration file) the first time any Web-Method is invoked); stores returned object

GetInitializer(methodInfo, attribute)	Called once per configured WebMethod (if configured via SoapExtensionAttribute) the first time any configured WebMethod is invoked; stores returned object
Initiailize(object)	Called once per configured WebMethod invocation (receives object returned by GetInitializer)
ProcessMessage(...)	Called four times per WebMethod invocation (this is where you do the pre/post processing work)

Figure 8 BeforeDeserialize

```
public override void ProcessMessage(SoapMessage message)
{
  if (SoapMessageStage.BeforeDeserialize == message.Stage)
  {
    try
    {
         // perform XML Schema validation here
         // configure XmlValidatingReader
         XmlTextReader tr = new XmlTextReader(message.Stream);
         XmlValidatingReader vr = new XmlValidatingReader(tr);
         ... // load schema files in XmlValidatingReader
         while (vr.Read()) ; // read through stream
    }
    finally
    {
      // reset stream position
      message.Stream.Position = 0;
    }
  }
}
```

Figure 9 GetInitializerHelper

```
public object GetInitializerHelper()
{
  // XML Schema cache for schema validation
  XmlSchemaCollection sc = new XmlSchemaCollection();
  // load schemas from vroot cache
  HttpContext ctx = HttpContext.Current;
  if (Directory.Exists(ctx.Server.MapPath(relativeDir)))
  {
      string[] schemaFiles = Directory.GetFiles(
        ctx.Server.MapPath("xsd"), "*.xsd");
      foreach (string schemaFile in schemaFiles)
      {
        XmlTextReader r = new XmlTextReader(filePath);
        XmlSchema schema = XmlSchema.Read(r, null);
        sc.Add(schema);
      }
  }
  return sc;
}
```

Figure 10 SOAP Fault

```
<soap:Envelope
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <soap:Fault>
      <faultcode>soap:Server</faultcode>
      <faultstring>Server was unable to process request. —&gt; Element
      'http://example.org/geometry/:CalcArea' has invalid child element
      'http://example.org/geometry/:Length'. Expected
      'http://example.org/geometry/:length'. An error occurred at
      (4, 3).</faultstring>
      <detail />
    </soap:Fault>
  </soap:Body>
</soap:Envelope>
```

Figure 11 New Types

```
<s:element name="AddEmployee">
  <s:complexType>
    <s:sequence>
      <s:element minOccurs="0" maxOccurs="1"
                 name="name" type="s:string" />
      <s:element minOccurs="0" maxOccurs="1"
                 name="id" type="tns:SSN" />
      <s:element minOccurs="1" maxOccurs="1"
                 name="bday" type="s:dateTime" />
      <s:element minOccurs="1" maxOccurs="1"
                 name="salary" type="tns:NewHireSalary" />
    </s:sequence>
  </s:complexType>
</s:element>
```

Figure 12 Assertions

```
[AssertNamespaceBinding("s",
  "http://schemas.xmlsoap.org/soap/envelope/")]
[AssertNamespaceBinding("t",
  "http://example.org/geometry/")]
[WebService(Namespace="http://example.org/geometry/")]
[Assert(@"//t:width >= 0", "width must be greater than 0")]
[Assert("(//t:length > //t:width)",
        "Length must be greater than width")]
public class Geometry
{
    [WebMethod]
    [Validation]
    [Assert("(//t:length * //t:width) > 100",
            "Area must be greater than 100")]
    [Assert("(//t:length div //t:width) = 2",
```

```
            "Length must be twice the size of width")]
    public double CalcArea(double length, double width)
    {
        return length * width;
    }
    [WebMethod]
    [Validation]
    public double CalcPerimeter(double length, double width)
    {
        return length * width;
    }
}
```
