

SOAP is Not a Remote Procedure Call

I truly believe that [WebMethod] is one of the best features of the .NET Framework. Unfortunately, it is at the same time the very attribute which hides way too many Web Service features from you.

It's one of the best features as it allows developers to easily create web services. But I also think that we heavily need to change our thought patterns regarding web services to see them not just as *yet another means for remote procedure calls* but instead as a vehicle for transferring XML documents from point A to point B, potentially crossing several routing hops and processing pipelines.

The State of the XML Nation

Before into going the details of why [WebMethod] deprives you of some of the most important features of web services, let us first have a look at the state of the nation regarding the use of XML documents. After years of marketing and buzzwording, XML today is a broadly used technology which is employed in a massive number of business applications. I can remember attending a keynote presentation in the year 2000 with two CTOs in the row behind me talking about their company and concluding at the point that

"OurApplication will need to have XML support, else OurCompany won't exist in two years time." They didn't. They still exist. This corporation develops PBX phone switching systems.

Fortunately, we've left these times of ultimate XML hype behind and today can reasonably use this technology wherever and whenever it is applicable. We also tend to use it in a reasonable and extensible way. But do we?

If asked to persist some data to an XML file, I guess that most of today's developers will come up with a document similar to the one presented in Listing 1.

Listing 1 State of the Art for most developers

```
<PurchaseOrder>
  <Customer Id="123" />
  <ShippingAddress>
    ...
  </ShippingAddress>
  <Items>
    <Item>
      <ArticleId>1234</ArticleId>
      <Amount>3</Amount>
      <Price>34.50</Price>
      <Currency>EUR</Currency>
    </Item>
  </Items>
</PurchaseOrder>
```

Unfortunately, XML documents like these have a serious drawback which should be immediately addressed: they are not scoped. If a sec-

Ingo Rammer

<http://www.IngoRammer.com>



Ingo Rammer is independent consultant, mentor, and architect helping companies to adopt Microsoft .NET and Web service technologies.

He can help you to

- **Decrease the time** necessary for initial adoption of .NET and Web Services
- **Go for the best solution** right from the start instead of spending time with trying various approaches
- **Get a deeper understanding** of reliability, scalability and performance in the .NET Framework and how they apply to your application
- **Increase your own proficiency** in application architecture and design
- **Understand SOA** and how it applies to your current and future applications



and company comes up with a different format for their own `<PurchaseOrder>` entities, an application will not be able to immediately distinguish between these two different formats. After all, both of them are called `<PurchaseOrder>`. That's why XML introduced the notion of a namespace. This is a great step forward as it allows you to easily determine which data is contained in your XML document. You can do this simply by looking at the namespace URI of the `<PurchaseOrder>` entity. In fact, I tend to believe that an XML document without a namespace is not worth the paper it has been written on. Or such.

The next step to enable information interchange is to define some notion of a "service" which allows you to for example create new orders as shown in Listing 2.

Listing 2 Introduce the concept of a "Service"

```
<myns:NewOrder xmlns:myns="...">
  <myns:CreationDate>
    ...
  </myns:CreationDate>
  <myns:PurchaseOrder>
    <myns:CustomerId="123"/>
    ...
  </myns:PurchaseOrder>
</myns:NewOrder>
```

Here you basically took the namespace'd `<PurchaseOrder>` entity and embedded it inside a different entity, `<NewOrder>`. This allows an application which receives this document to determine that you actually want to create a new purchase order.

This version of the document allows you to easily send this document from point A to point B without having to worry about communicating out of band about which kind of service you want the receiver to perform.

The next step is to create some facility for addressing, routing, and transmitting your XML document. These extensibility points can later be used to pass additional information to the transport or processing framework. This is demonstrated in Listing 3.

As you can see, a classic SOAP envelope's `<soap:Body>` usually just wraps up some service oriented XML entity (`<NewOrder>`), which in turn contains the business oriented information (`<PurchaseOrder>`).

I think the best analogy is to compare SOAP with conventional letters sent by snail mail. Your XML document is the letter, and the SOAP envelope is just this—an envelope into which you

Listing 3 Add facilities for message headers

```
<soap:Envelope>
  <soap:Header>
    <myns:SomeHeader value="..." />
  </soap:Header>

  <soap:Body>
    <myns:NewOrder xmlns:myns="...">
      <myns:CreationDate>
        ...
      </myns:CreationDate>
      <myns:PurchaseOrder>
        <myns:CustomerId="123"/>
        ...
      </myns:PurchaseOrder>
    </myns:NewOrder>
  </soap:Body>
</soap:Envelope>
```

can put your document. Just as with normal letters, you can then specify various quality of service or delivery requirements¹ by attaching flags to its outside—in SOAP's case, the header.

The SOAP envelope shown above basically expresses the state of the world in common SOAP 1.1 systems. This is the part where [WebMethod] absolutely excels: it allows you to define a service accepting this exact document by writing code like the one shown in Listing 4 and 5.

Listing 4 Define the matching [WebMethod]

```
[WebService
 (Namespace="urn:ingorammer.com:orders:200309")]
public class OrderService: WebService
{
  [WebMethod]
  public void NewOrder(DateTime CreationDate,
                      Order Order)
  {
    // process the order
  }
}
```

Listing 5 Order and Item will be serialized correctly

```
public class Order
{
  public int Customer;
  public String ShippingAddress;

  [XmlElement(ElementName="Items")]
  [XmlArrayItem("Item", typeof(Item))]
  public ArrayList Items;
}

public class Item
{
  public int ArticleId;
  public int Amount;
  public double Price;
  public String Currency;
}
```

As soon as you generate a proxy for this web service (using "Add web reference ..."), you will be able to call it directly from any .NET client and have the framework generate the correspond-



ing XML entity *NewOrder* for you.

The .NET Framework basically allows you to invoke the web service as if it was just another procedure call. Even though this might be enough for some applications, it unfortunately deprives you of the best features of XML web services. It deprives you of XML.

Why use XML Web Services without XML?

One of the nicer features of XML is that it supports extensibility mechanisms. If the underlying XML Schema has been designed with this in mind, you will be able to extend any XML element with further information without breaking your interface contract. This will allow you for easier versioning and better decoupling between client and server.

The XML schema definition of an extensible "PurchaseOrder" type can for example look like the snippet shown in Listing 6.

Listing 6 Extensible schema using <xs:any>

```
<xs:complexType name="PurchaseOrder">
  <xs:sequence>
    <xs:element
      minOccurs="1"
      maxOccurs="1"
      name="Customer"
      type="xs:int" />
    <xs:element
      minOccurs="0"
      maxOccurs="1"
      name="ShippingAddress"
      type="xs:string" />
    <xs:element
      minOccurs="0"
      maxOccurs="1"
      name="Items"
      type="s0:ArrayOfItem" />
    <xs:any
      minOccurs="0"
      maxOccurs="unbounded" />
  </xs:sequence>
</xs:complexType>
```

In essence, this usage of <xs:any> as the last element of the "PurchaseOrder" complexType specifies that the sending application is allowed to include zero, one or more additional XML entities at this point in the XML document without breaking the schema contract.

But can you use this extensibility mechanism with ASP.NET Web Services as well? Yes, you can! The C# class which corresponds to the schema snippet above is shown in Listing 7.

As you can see in this listing, you will be able to access any additional entities which have been sent by the client by defining an array of

Listing 7 Expressing <xs:any> in C#

```
public class Order
{
  public int Customer;
  public String ShippingAddress;

  [XmlElement("Items")]
  [XmlArrayItem("Item", typeof(Item))]
  public ArrayList Items;

  [XmlAnyElementAttribute]
  public XmlElement[] Any;
}
```

XmlElement and marking it with the [XmlElementAttribute].

The Importance of Extensibility

But why would you want this kind of extensibility? Why would the client ever send some information which the server can not process as it doesn't know what to expect in the first place?

If we stay with the purchase order example, the client can for example include reference or collation information used to later match the shipment with its own purchase order tracking system. Listing 8 shows an XML document which uses this method.

Listing 8 Sending extended information

```
<soap:Envelope>
  <soap:Body>
    <NewOrder xmlns="...">
      <CreationDate>...</CreationDate>
      <Order>
        <Customer>42</Customer>
        <ShippingAddress>...</ShippingAddress>
        <Items>
          ...
        </Items>
        <PurchaseOrder xmlns="urn:client.com:PO">
          F00-42
        </PurchaseOrder>
        <AuthorizedBy xmlns="urn:client.com:PO">
          John Doe
        </AuthorizedBy>
      </Order>
    </NewOrder>
  </soap:Body>
</soap:Envelope>
```

If we have a look at the complete business process and its associated message exchange, we can envision that the server will at a later point in time create documents like "OrderAccepted" and "ShipmentCompleted". These would be transferred back to the company which originally placed the order. In our case, this new message can include the original *PurchaseOrder* XML document along with the additional entities created by the client. The original sender can then query this document, find the additional informa-



tion to correlate the shipment with its own ordering system.

The service would essentially just reflect all additional data back to the sender without ever trying to interpret the information.

But wait! Isn't this synchronous HTTP?

You might be wondering how scenarios like this are applicable in current web service applications. After all, if you invoke a service as shown in Listing 9, then how can you later receive response messages?

Listing 9 Simple Web Service invocation is not enough

```
OrderService svc = new OrderService();
svc.NewOrder(DateTime.Now, ord);
```

Well, you now have to change the way you think about Web Services. Forever!

Clicking "Add Web Reference ..." in Visual Studio.NET and writing code like the one presented above only shows you a very small subset of the possibilities of web services. It mandates a request/response RPC pattern to web services which limit you to the use of synchronous HTTP without the possibility to receive truly asynchronous responses.

In the future, we will regard web services mainly as a means for transferring XML documents. In the world of global business, this can for example mean to send *PurchaseOrder* documents from a customer to a vendor.

To go back to the previous analog between conventional snail mail letters and SOAP, you will also see different message exchange patterns, and not just today's standard request/response.

Following the purchase order example, this could mean that you will see three different XML messages exchanged between client and server:

1. Customer to Vendor: *NewOrder*
2. Vendor to Customer: *OrderAccepted* (maybe minutes or hours later. Just think along the lines of Amazon.com's "order confirmed" email which you receive literally hours after placing your order).
3. Vendor to Customer: *ShipmentCompleted* (hours, days, or even weeks after the initial request message. Equivalent to Amazon.com's shipment email which is sent as soon as the goods leave their warehouse)

In general, you will come across at least two very distinct message exchange patterns, and numerous combinations of them:

- Request/Response
- OneWay
- Dialog (combination)

In the example above, we are basically dealing with a *Dialog* exchange pattern consisting of three OneWay messages. In addition to enabling different message exchange patterns, you will also come across the fact that your XML document will have to pass through different routers and services along a processing pipeline before reaching your final destination. It is for example quite unlikely that Amazon.com would allow your incoming XML Order message direct, real time access to its backend systems. It will instead store them in some sort of message queue so that it can process these messages whenever capacity is available, and will send outgoing *OrderAccepted* message later on.

This is again pretty similar to the handling of snail mail letters. As soon as you hand your letter to some courier service, it is very unlikely that the person you initially handed the document to will be the same one who'd deliver it to the recipient. The letter will instead be queued, forwarded, routed, and handled by different persons and corporations. They might even add some more information to the outside of your envelope, for example to include better routing codes.

The only thing they will never do is to change your message. The same thing applies for SOAP: every intermediary can read, add to, act upon and even modify the SOAP headers but must not ever touch your message.

On the technical side, it's the task of protocols like WS-Addressing to allow for transparent routing by enabling you to store all necessary routing and destination information directly in the SOAP envelope as illustrated in Listing 10.

Listing 10 Store the destination in the envelope!

```
<soap:Envelope
  xml ns:wsa="http://.../addressing">
  <soap:Header>
    <wsa:To soap:mustUnderstand="1">
      http://www.ingorammer.com/MyService
    </wsa:To>
  </soap:Header>
  <soap:Body>
    <NewOrder xml ns="urn:... ">
      ...
    </NewOrder>
  </soap:Body>
</soap:Envelope>
```

In addition, this format also allows you to specify the reply endpoint which should be used by the server if it wants to respond to your mes-



sage. If you for example send a purchase order to the endpoint above and want the server to reply with its *OrderAccepted* and *ShipmentCompleted* messages to the URL <http://mycompany.com/reply>, then you could indicate this by sending a message similar to the one in Listing 11.

Listing 11 ReplyTo Addressing in the Header

```
<soap:Envelope
  xmlns:wsa="http://.../addressing">
  <soap:Header>
    <wsa:To soap:mustUnderstand="1">
      http://www.ingorammer.com/MyService
    </wsa:To>
    <wsa:ReplyTo soap:mustUnderstand="1">
      http://mycompany.com/reply
    </wsa:ReplyTo>
  </soap:Header>
  <soap:Body>
    <NewOrder xmlns="urn:... ">
      ...
```

The reasoning for all these features is to allow you to model complex message exchange patterns which directly reflect the underlying business processes.

Change your Thought Pattern!

Please let me restate that all these advanced Web Services features are built around one core assumption: that you want to transfer XML documents from a creator to an ultimate destination. Future Web Services will not be built around any Remote Procedure Call (RPC) semantics, even though this will still be possible as well. The reason is that you can always use XML messages as a foundation for RPC, but it doesn't work the other way round.

Summary

If you only looked at Web Services as a means for HTTP based remote procedure calls, then you have limited yourself to only a small subset of the possibilities. The true power of SOAP lies in its core design as a means for sending XML documents from point A to point B while potentially crossing multiple routers and processing pipelines on its way.

The combination of these features allows you to design asynchronous message exchange patterns which model the underlying business process better than any RPC or request/response oriented protocol ever could.

Did you like this issue of Ingo Rammer's Architecture Briefings? If yes, then please note that you are allowed to freely print, share and forward it to your coworkers, managers, friends, user groups, and whoever you think would profit from access to this document.

If you would like to receive future issues, you are invited to subscribe—for free—at

<http://www.ingorammer.com/NL>

¹ Even though I have to admit that I don't really believe anymore that sticking "Airmail" onto a letter actually accelerates delivery by any margin.